Web
Performance
IN ACTION

BUILDING FAST WEB PAGES

Jeremy L. Wagner

Foreword by Ethan Marcotte

/// MANNING

Web Performance in Action: Building Fast Web Pages

by [Jeremy L. Wagner](#)
Publisher: [Manning Publications](#)
Release Date: December 2016
ISBN: 9781617293771
Topic: [Web Development](#)

Book Description

Summary

Web Performance in Action is your companion guide to making websites faster. You'll learn techniques that speed the delivery of your site's assets to the user, increase rendering speed, decrease the overall footprint of your site, as well as how to build a workflow that automates common optimization techniques.

About the Technology

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.

About the Book

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

About the Reader

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

About the Author

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

Copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Brief Table of Contents

Table of Contents

Foreword

"May you live in interesting times," goes the old half-curse. And I don't know about you, but it feels like the web is perpetually stuck in interesting times. We're designing for an ever-expanding number of mobile devices, each one more powerful than most laptops I've owned throughout my career. But we're also designing for a web that travels over the aging infrastructure of developed economies, as well as to cheaper, low-powered mobile devices in younger, emerging markets.

In other words, the web is more broadly accessed today than ever before—but over a network that's far more fragile than we might like to think. Once a user requests one of our web pages, any number of things can fail. Maybe a connection drops, or a network's latency is too high for an asset to load. Or maybe the users exceeded their data allotment for the month.

We're building digital experiences—some responsive, some not—that are more beautiful than anything produced at any other point in the web's history. But we need to start designing for *performance* as well. We need to create sites and services optimized for the fragility of the network, as well as the widths of our users' screens.

Thankfully, you've begun reading *Web Performance in Action*, a book that can help you do just that. Jeremy Wagner has written an invaluable, accessible reference for the modern web developer, one that demystifies even the most arcane-sounding acronyms and frames even the most arcane-seeming web optimization tricks in approachable, plain language. In interesting times like these, Jeremy's guide is indispensable: As you travel through these pages, you'll gain the skills to ensure your sites are as beautiful as they are fast, nimble, and bandwidth-friendly.

Ethan Marcotte

Designer, Ethanmarcotte.com

Author of *Responsive Web Design*

Preface

Well before I ever entertained the notion of writing a book, the idea that websites ought to be fast was a high priority in all my projects. In my humble opinion, slow websites are not a mere inconvenience. They are a critical sort of user experience problem. Until a website loads, no user experience exists. The longer it takes for a site to load, the more this absence is felt by the user.

When I proposed this book to Manning in 2015, I was hardly the first to write on the topic of web performance. Many authors before me had written in this space, and I knew that I would be standing on the shoulders of giants. My goal with *Web Performance in Action* was to provide a modern guide for today's web developers that would give them the knowledge they need to make their websites faster than ever. I think this book meets that goal.

When web performance is discussed, it's often tied to financial concepts. The idea that a poorly performing website can affect sales or ad revenues is hardly new. What we don't hear enough about, however, is how such a website can be potentially costly for the user on a restricted data plan. Or how slow websites are an impassible sort of barrier for people mired in an antiquated internet infrastructure. So much of the world has such a difficult time accessing the web. While infrastructure is slowly improving, we as developers can move the needle for users by developing sites with performance in mind.

I wrote *Web Performance in Action* to help you meet your goals, and the folks at Manning participated in refining it. In an age where the web is becoming increasingly complex, the time has never been more appropriate to tackle this problem. I think this book will help you get to where you want to be.

Acknowledgments

It takes a ton of people, beyond the author, to put a book together. The people at Manning played a huge role in getting this book from a mere proposal to what you're reading right now. Brace yourself, because this section is brimming with gratitude.

I'll start by thanking the first person from Manning I talked to, an acquisitions editor named Frank Pohlmann. The proposal phase of this book took quite some time, and Frank coached me on what to do and what not to do, and most importantly, let me know exactly what I was getting into. Thank you, Frank, for guiding me in the early part of this process.

As this is my first book, I'd like to extend my gratitude to Manning's publisher Marjan Bace, who saw fit to grant me this opportunity. Green-lighting any book proposal is a risk, and that's especially true when the proposal comes from someone who hasn't made a name for himself prior to that point, so it took some courage to take that risk. Thank you, Marjan.

Behind every author is an editor pushing them to write the best manuscript they can. Susanna Kline was the development editor for this book, and here I offer my sincerest thanks and gratitude for her hard work and indispensable guidance on this project. Susanna not only played the role of an editor, she was also a great coach who understood the vulnerability I felt in this vast and new undertaking, especially when there was so much uncertainty in the early stages of development. Her guidance in this project was essential to its success. Thank you, Susanna, for all your help.

Every technical book, of course, needs a technical editor. Nick Watts did a superb job in this role. His informed perspective, valuable input, and willingness to challenge my assertions and points of view certainly contributed positively to the quality of the final text. Thank you, Nick.

This book was also reviewed by many people at various stages in its development, including Alexey Galiullin, Amit Lamba, Birnou Sebarte, Daniel Vasquez, John Huffman, Justin Calleja, Kevin Liao, Matt Harting, Michael Martinsson, Michael Sperber, Narayanan Jayaratchagan, Noreen Dertinger, Omer Faruk Celebi, Simone Cafiero, and William Ross. Their feedback gave valuable insight into what public perception of the book could be. I would like to thank them for their input and suggestions, which made this book better than what I could have achieved on my own.

The final polish of a book is also very important. I'd like to thank David Fombella Pombal for his thorough and excellent technical proofing of the manuscript, which identified issues I would have otherwise missed. Sharon Wilkey meticulously combed through and copyedited the final manuscript, which further refined it, for which I'm grateful. Elizabeth Martin filed off the rough edges and reined in some of my excesses with keen precision. On top of all this, Kevin Sullivan did a great job of coordinating the preproduction and production

phases. Thank you so much, guys. You did great work in that last, critical mile of the project.

I also wish to extend my gratitude to Ethan Marcotte, a person whose work has irrevocably changed how we all develop for the web. When I contacted Ethan to see if he would be interested in writing the foreword to *Web Performance in Action*, I was pleasantly surprised that he had time to reply, let alone read the manuscript. A foreword is not a matter to be taken lightly. It's an endorsement of a book's quality. To know that Ethan endorses the material in this book is one of the proudest moments of my professional career. Thank you, Ethan.

I'd like to thank my father Luke and my mother Georgia for supporting me in all that I've done and attempted to do, even the harebrained stuff. I'd also like to thank my brother Lucas who has always been an incredible example to me, and led the way in showing what's possible if you're willing to work hard for something. Thank you so much.

Lastly, I owe gratitude and thanks to my wife Alexandria. Her unwavering support and selflessness has been a source of strength for me throughout this endeavor. Her gentle encouragement and belief in my potential has helped more than she knows.

To anyone else I may have overlooked, know that you were a part of making this book what it is. For that, I thank *you*.

About this Book

The purpose of *Web Performance in Action* is to teach you how to create faster websites, and through the course of this book, I'll help you get here. The techniques you'll learn as you read should also come in handy for improving performance on existing websites.

Who should read this book

This book focuses heavily (though not exclusively) on improving website performance on the client side. This means that it's targeted toward front-end developers who have a good command of HTML, CSS, and JavaScript. You, the reader, should be comfortable working with these technologies.

This book occasionally strays into the server side where appropriate. For instance, some server-side code examples are in PHP. These examples are intended to be illustrative of a concept, and are often peripheral to the task at hand. Chapter 10 covers server compression, including the new Brotli compression algorithm, which fits into the server-side category. Chapter 11 explains HTTP/2, so having an interest in how this new protocol can affect how you optimize your site can be helpful.

You should also be somewhat comfortable on the command line, but even if you're not, you'll still be able to follow along in the examples provided. Now, let's talk about how this book is structured.

Roadmap

Unlike most other Manning titles, this book is not divided into parts, but it does follow a logical flow of sorts. Chapter 1 is an introduction to the fundamentals of web performance—bedrock stuff, such as minification, server compression, and so forth. If you're already a performance-minded developer, this chapter will feel familiar to you. It's intended for the front-end developer who's new to the concept of web performance. Chapter 2 covers performance assessment tools, both online and in the browser, with a focus on using Chrome's developer tools.

From there, we'll venture into the realm of optimizing CSS. Chapter 3 is a grab bag of topics and examples of how you can make your CSS leaner, and use native CSS features that can help increase the responsiveness of your website to user input. Chapter 4 is about critical CSS, a technique that can give your site's rendering performance a real shot in the arm.

Then, we'll tackle image optimization. Chapter 5 focuses on different image types and how to use them, as well as how to deliver them optimally to different devices both in CSS and

inline in HTML. Chapter 6 covers how to reduce the file size of images, automating the creation of image sprites, Google's WebP image format, and how to lazy load images by writing a custom lazy loading script.

After all of that, we'll turn our focus away from images toward fonts. Chapter 7 covers optimizing fonts. This ranges from creating an optimal @font-face cascade to font subsetting, using the unicode-range CSS property, compressing legacy font formats on the server, and how to control the loading and display of fonts with CSS and JavaScript.

Chapters 8 and 9 focus on JavaScript. Chapter 8 speaks more to the need for minimalism in JavaScript by advocating the use of in-browser features, rather than relying on jQuery and other libraries. For those who can't abandon jQuery, I talk about jQuery-compatible alternatives that offer a subset of what jQuery does, but with less overhead. This chapter also talks about proper placement of the <script> tag, as well as how to use the async attribute, and animating with the requestAnimationFrame method. Chapter 9 ventures into the territory of JavaScript service workers. In this chapter, you'll learn how you can serve content to users who are offline, as well as how you can improve the performance of pages for online users with this technology.

Chapter 10 is yet another grab bag of topics. It covers the impact of poorly configured server compression, the new Brotli compression algorithm, resource hints, configuring caching policies, and the benefits of using CDN-hosted resources.

Chapter 11 covers HTTP/2, the performance problems that it solves, how optimization practices differ between it and HTTP/1, Server Push, and a proof of concept of how you can adapt the delivery of your website's content to accommodate both versions of the protocol.

Chapter 12 takes a good chunk of what you've learned and automates it with the gulp task runner. In this chapter, you'll learn how to automate various aspects of optimizing your website's performance, which will help you optimize your sites as you code them, saving you valuable time.

There are two appendixes. Appendix A is a tools reference. Appendix B highlights common jQuery functions and shows you how to accomplish the same tasks by native means.

Tools used in this book

While following examples in this book, you'll have open your favorite text editor and a command-line window. Beyond that, two tools are used consistently throughout, so you'll want to have them installed.

Node.js

Node.js, sometimes referred to as Node, is a JavaScript runtime that allows you to use JavaScript outside of the browser. It can be used for all kinds of crazy stuff that, some years ago, no one would have thought that JavaScript would be used for. I'm talking about task runners, image processors, and even web servers. All these things are installed using the Node Package Manager (npm).

In your optimization efforts throughout the book, you'll use Node for all of that. You'll often use it to run local web servers using the Express framework for examples that we'll work through together. In chapter 11, you'll even use it to run a local HTTP/2 server. You'll use Node in chapter 6 to optimize images in bulk, and in chapter 12, you'll use it to automate common optimization tasks with gulp. It's used in nearly every chapter for one kind of function or another.

If you're serious about working your way through this book, you'll need to have Node installed. If you don't have it set up, go to https://nodejs.org and head to the downloads section. If you're feeling trepidation because you don't know Node, don't worry! Everything is explained, and if you follow the directions, you should be fine. But, if you feel that you'd benefit from a deep dive into how Node works later on, check out *Node.js in Action*, another title from Manning (https://www.manning.com/books/node-js-in-action). Just know that a deep knowledge of Node is not necessary to navigate through this text.

Git

Git is a version control system used for keeping track of changes in software applications. There's a good chance you've used it, but if not, you'll get to use it in this book. Git is used to download code for examples from this book's collection of GitHub repositories hosted at https://github.com/webopt. You can download Git at https://git-scm.com.

Why use Git instead of downloading zip files of code examples? For one, using a version control system like Git on the command line makes it easier to grab things and go. The biggest advantage, though, is that you'll be able to easily skip ahead to finished code examples if you get stuck or just want to see the final result.

If you've never used Git, don't sweat it. All the instructions for using it are delineated clearly, and you'll be able to follow along. If you prefer not to use Git, you can go to https://github.com/webopt and download zip files from each repository.

Other tools

Most of the tools you'll use in this book will be installed by Node Package Manager, and are thus dependent on Node. There are two instances, however, where you'll get an opportunity to use tools beyond Node.

In [chapter 3](), there's an example where you'll apply the DRY (don't repeat yourself) principle to CSS, which entails combining redundant rules under multiple selectors. A Ruby-based tool named csscss is used in this example to detect redundancies. If you have a Mac or you're running any other UNIX-like operating system, this may already be available for you. If you're running Windows, you'll have to download Ruby at [http://rubyinstaller.org]().

In [chapter 7](), there's an example where you'll subset fonts to make them smaller. You'll use a Python-based tool called pyftsubset. Like Ruby, there's a good chance that on UNIX-like systems, Python will already be available. If you use Windows, you'll want to head over to [www.python.org]() and grab the installer.

Code conventions

Code in this book is written in a fashion that most developers will be comfortable with. All source code in the book is in a fixed-width font like this, which sets it off from the surrounding text. In code snippets throughout the book, relevant portions are annotated for clarity. Changed portions of an existing snippet are typically set in **bold font like this**. Regarding the code you download from GitHub, indentations are done with tabs. When it comes to how many spaces you want a tab character to represent, that's up to you. When I wrote the examples, I went with four spaces. The code snippets in the book follow that convention.

Source code for all working examples is available on the publisher's website ([www.manning.com/books/web-performance-in-action]()) as well as GitHub ([https://github.com/webopt]()).

Author Online

Purchase of *Web Performance in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to [www.manning.com/books/web-performance-in-action](). This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the Author

Jeremy Wagner is a professional front-end web developer with over ten years of experience in various agencies and large companies. In addition to his writings on web performance, he also speaks at conferences on a variety of web development‑related topics. He can be found on the web at https://jeremywagner.me or @malchata on Twitter.

About the Cover Illustration

The figure on the cover of *Web Performance in Action* is captioned "Man from Bednja, near Zagreb, Croatia." The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian's retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It's now hard to tell apart the inhabitants of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Chapter 1. Understanding web performance

- 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.

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.
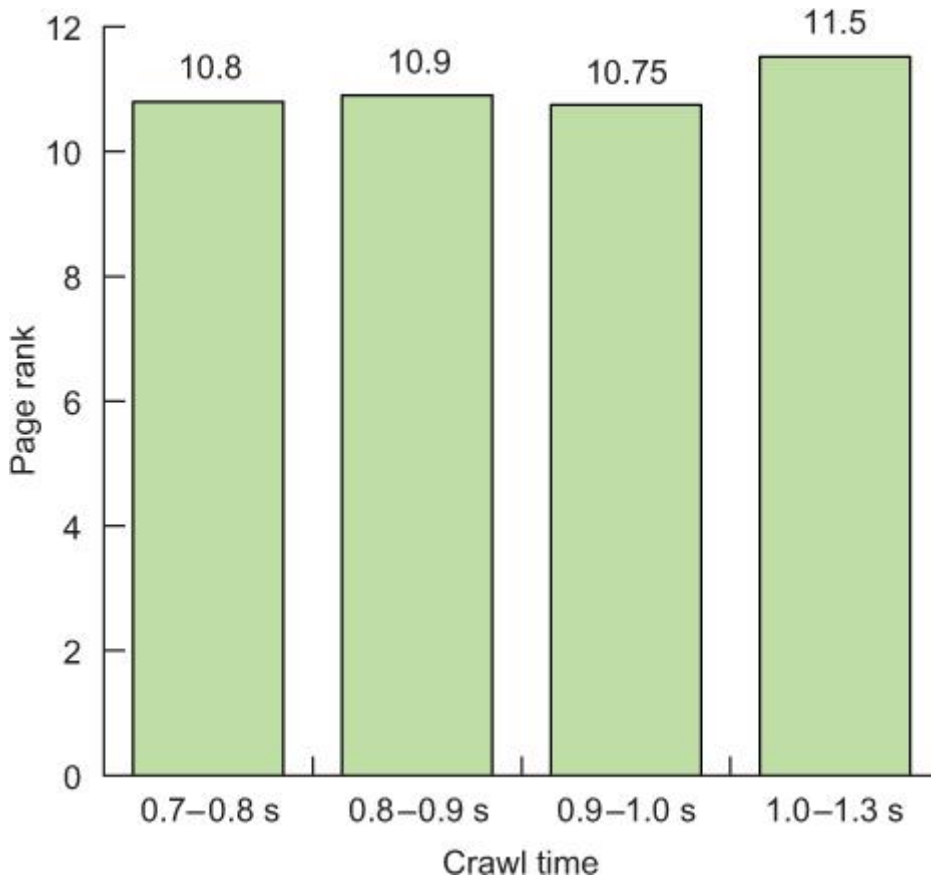
Search rankings remain stable, but when crawl times start straying beyond a second, the ranking slips. It pays to take performance seriously. If you're running a content-driven site such as a blog, your organic search rankings are the greatest source of traffic you have. Reducing your website's load 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.

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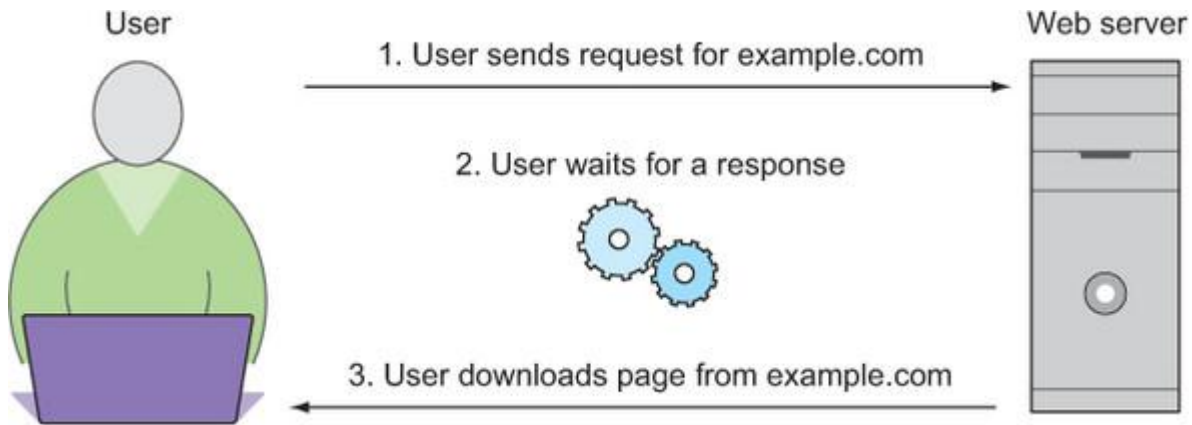
When it's said that web performance focuses on making websites load faster, the primary focus is on reducing load time. The most simple interpretation of *load time* is the time between the instant a user requests a website and the instant it appears on the user's screen. The mechanism driving this is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

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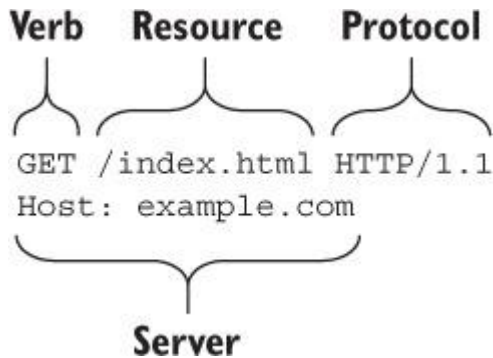
After making the request, you receive a response code of 200 OK, which assures you that the resource you've requested exists, along with a response containing the contents of /index.html. The content of /index.html is then downloaded and interpreted by the web browser.

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
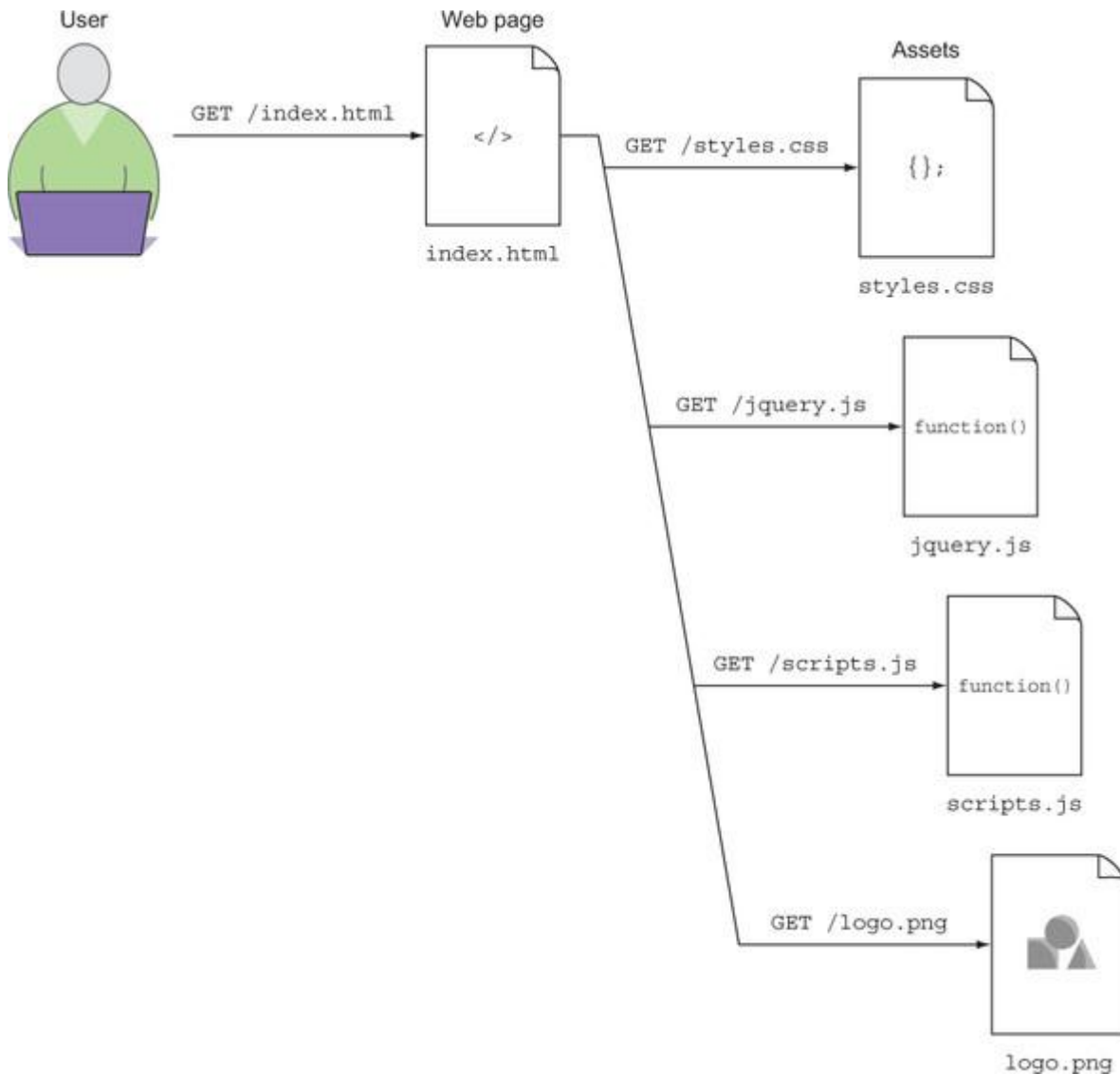
After the browser downloads index.html, it discovers a <link> tag to a style sheet, a couple of <script> tags linking to JavaScript files, and an <img> tag referring to an image. When the browser discovers these references to other files, it makes new HTTP requests on the user's behalf to retrieve them. 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.
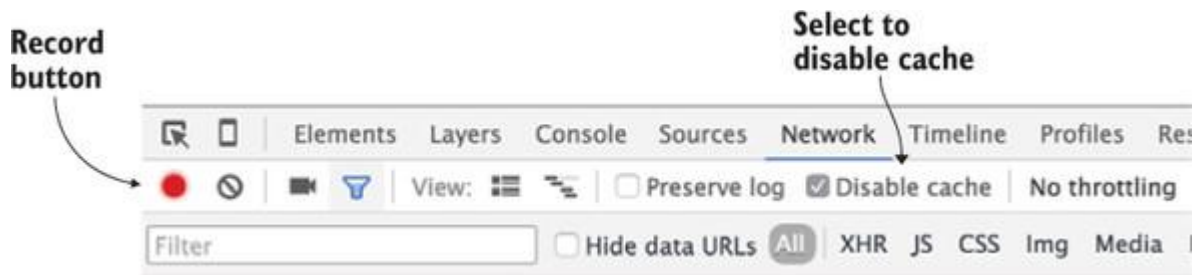
The first thing you'll want to do in the Network tab is ensure that the Disable cache check box is selected. When a website is first visited, none of the assets are cached, and this is the scenario that you want to be able to replicate. Otherwise, the site's assets will be served from the cache. Although 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.

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.
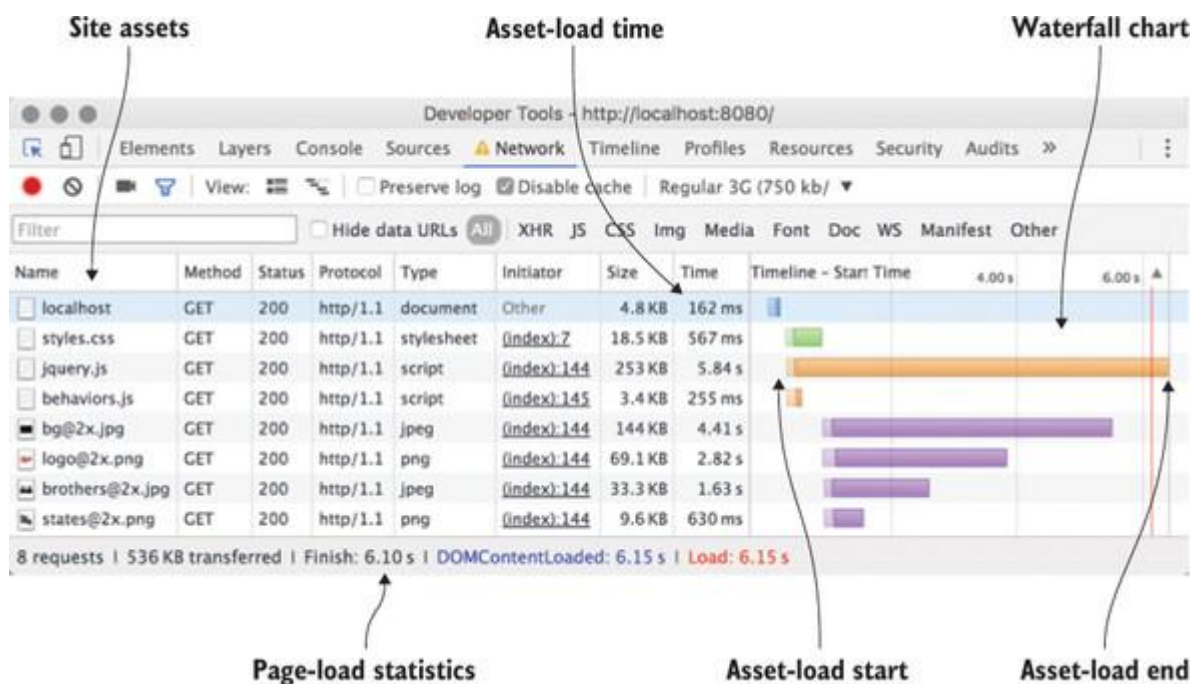
The waterfall chart generated for your client's site shows eight requests. Although this isn't an obscene number of requests, 536 KB of data is spread across them, and that's a significant amount for a small site like this. Because of the amount of data, the site loads in about 6.15 seconds on the Regular 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. 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 the images on the site without compromising their visual integrity. Finally, you'll finish by employing compression on the server for text assets.

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!

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.

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.

Unminified: 98 bytes

```
.logo
{
    width: 282px;
    height: 186px;
    position: absolute;
    top: 0;
    left: -54px;
    z-index: 11;
}
```

```
.logo{width:282px;height:186px;position:absolute;top:0;left:-54px;z-index:11}
```

Minified: 77 bytes
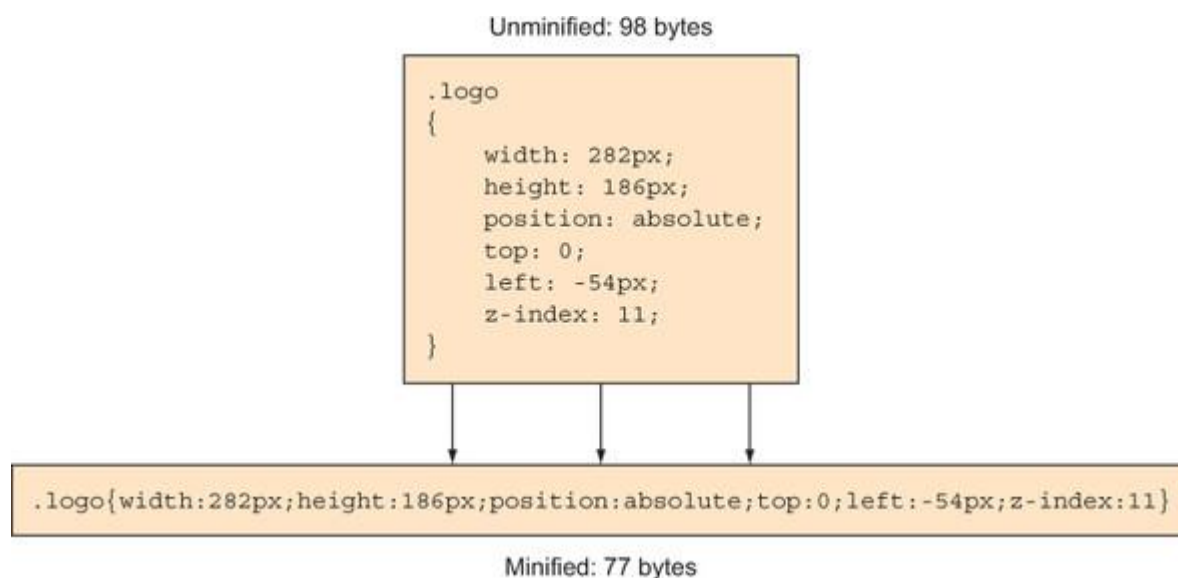
Many human-readable files such as CSS and JavaScript contain whitespace and characters that are inserted by developers during development. We use line breaks and indentation in our CSS and JavaScript to make them easier to read, as well as using comments in source code for documentation purposes.

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.

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.

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.

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*.

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


```
1. User requests                              2. Web server sends
   compressed content                            compressed response

              GET /index.html
              Accept-Encoding: gzip, deflate

                        </>

                      index.html

              Content-Encoding: gzip
```

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");        ◁─┐ Compression module is
var app = express();                                 imported into the script.

// Run static server
app.use(compression());                          ◁─┐ Script hooks the compression
app.use(express.static(__dirname));                  module into the web server.
app.listen(8080);
```

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%

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.

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%.

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                    Compresses files that match
                               module is loaded.                        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.

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.

Unoptimized
(30.87 KB)

Optimized
(11.69 KB)

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.

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

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 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%

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.

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%.

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%

Device type	Page weight before	Page weight after	Reduction
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%

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

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.

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.

Chapter 2. Using assessment tools

This chapter covers

- Using Google PageSpeed Insights
- Using network request inspectors to view timing information for assets
- Using rendering profilers to diagnose poor performance
- Benchmarking JavaScript code
- Emulating devices and internet connections

Now that you have a handle on the idea of web performance and have had a chance to optimize a client's site, it's time to go deeper. That starts with learning about tools that identify performance issues. These exist both online and in the browser, starting with Google's PageSpeed Insights, and ending with the tools available in Chrome and other desktop browsers.

2.1. Evaluating with Google PageSpeed Insights

It won't surprise you to know that Google cares about web performance. As early as 2010, Google indicated in a blog post that performance is a factor in a site's ranking in organic search results. If you're running a content-driven site that gets most of its traffic from search engines, this should give you pause. Fortunately, Google has an assessment tool: PageSpeed Insights.

2.1.1. Appraising website performance

Google PageSpeed Insights (https://developers.google.com/speed/pagespeed/insights/) analyzes a website and gives tips on how to improve its performance and user experience. When PageSpeed Insights renders its analysis, it does so twice: once with a mobile user agent and then with a desktop user agent. It analyzes performance with two criteria in mind: the time it takes for above-the-fold content to load, and the time it takes for the entire page to load. Figure 2.1 illustrates this concept of above-the-fold content versus below-the-fold content.

Figure 2.1. Google PageSpeed Insights checks two aspects of page speed: the load time of above-the-fold content, which is what the user sees immediately upon visiting a page, and the load time of the entire page.

The tool gives a score for both user agents from 0 to 100, and color codes its recommendations based on the severity of the issues it finds. Yellow indicates minor problems that you should fix if time allows, whereas red indicates problems that you should definitely fix. Performance aspects that pass are indicated in green. Figure 2.2 shows a sample report. The solutions to issues that PageSpeed Insights identifies are numerous and covered throughout this book in later chapters. Some are steps you took in optimizing the client site from chapter 1, such as minifying assets, configuring compression, and optimizing images.

Figure 2.2. Google PageSpeed Insights results for the mobile view of a website. A user enters a URL and gets performance tips grouped by severity for both mobile and desktop states.

One way to get your feet wet with PageSpeed Insights is to run it against the client website from chapter 1. Because PageSpeed Insights can't examine URLs on your local machine, I've hosted the unoptimized and optimized versions of the client website on a public web server. Enter the following URLs into PageSpeed Insights and compare the output of each report:

- http://jlwagner.net/webopt/ch01-exercise-pre-optimization
- http://jlwagner.net/webopt/ch01-exercise-post-optimization

When you enter a URL and click Analyze, generating the report takes a minute. After PageSpeed Insights finishes, you'll see tabs for the Mobile and Desktop profiles and scores for each. The output from the program looks similar to figure 2.3.

Figure 2.3. The PageSpeed Insights report for the client website from chapter 1 prior to (left) and after (right) your optimizations.

Most of the suggestions are aspects of performance you fixed in chapter 1. These suggestions consist of changes including minifying text assets such as HTML, CSS, and JavaScript; enabling compression; and so forth.

You'll likely see a persistent issue in the report for the optimized version of the site that prevents you from getting a higher score. This is because the <link> tag that's used to load the CSS blocks rendering of the page until the style sheet loads. You can fix this by inlining the CSS in the HTML inside <style> tags so that the CSS is downloaded at the same time as the HTML.

Inlining in general is considered somewhat of an antipattern and has a detrimental effect on caching. But it does cut down on HTTP requests, which is good for HTTP/1 servers, and it increases rendering speed of the document.

[Chapter 4](#) covers a technique called *critical CSS* that increases rendering speed of a page. It's an effective technique for HTTP/1 client/server interactions, but a feature in HTTP/2 called s*erver push* fixes this antipattern. To learn more about it, check out [chapter 10](#).

Next, you'll learn how to use Google Analytics to retrieve PageSpeed Insights data for more than one page, which gives you a broader perspective of your entire site's performance.

2.1.2. Using Google Analytics for bulk reporting

If you're a professional web developer, chances are good that you've used Google Analytics. This reporting tool provides data on your site's visitors, such as where they're located, how they got to your site, how much time they've spent there, and other statistics. Pertinent to this chapter is the PageSpeed Insights data available in this tool.

If you have Google Analytics on your site already, all you have to do is log in and follow along. If you haven't installed it on your site, sign in with a Google account at www.google.com/analytics and follow the instructions. The process takes little time and involves pasting a small bit of JavaScript code into your site's HTML. From there, you need to wait a day or two for Google Analytics to gather data.

Legal implications

Be warned that adding Google Analytics to your site comes with legal implications. When you install the tracking code, you're accepting the terms of a legal agreement. If you're the sole owner of a site, that's one thing, but be sure to get consent of the site's owner otherwise. This is important if you're a developer in a large company, where legal review is a common process.

After you've logged in, you'll be redirected to the website's dashboard. Go to the Behavior section in the left-hand menu and expand it to reveal a submenu, as shown in [figure 2.4](#).

Figure 2.4. PageSpeed Insights reporting information can be accessed in Google Analytics by navigating to the Behavior section on the left menu and clicking the Speed Suggestions link.

Upon entering this section, you'll see a dashboard with performance statistics, as shown in figure 2.5. This includes a line graph plotting the average load times of all visits for pages on the site in the last reporting period, as well as a table with the following columns:

- *Page—* The URL of the page.
- *Pageviews—* The number of views a page has received in the reporting period. The reporting period is usually the preceding month but can be changed to a custom time period.
- *Avg. Page Load Time—* The average number of seconds the page has taken to load.
- *PageSpeed Suggestions—* The number of suggestions PageSpeed Insights has to improve the performance of the associated page URL. Clicking this value directs you to a new window containing a PageSpeed Insights report for that specific URL.
- *PageSpeed Score—* The score given by the PageSpeed Insights report. This score is expressed in a range from 1 to 100, with lower scores indicating room for improvement, and higher scores indicating positive performance characteristics.

Figure 2.5. The reporting table of performance statistics in Google Analytics. Note the two rightmost columns with PageSpeed Insights–specific data and links to reports for associated page URLs.

	Page	Pageviews	Avg. Page Load Time (sec)	PageSpeed Suggestions	PageSpeed Score
1.	/eq-tips/	1,961	9.08	8 total	75
2.	/marshall-shoppers-guide-1/	1,812	7.81	8 total	71
3.	/vintage-style-pickups-explored-2/	1,791	6.01	7 total	75
4.	/edward-van-halen-brown-sound/	1,406	6.15	7 total	74
5.	/	1,099	5.38	8 total	70
6.	/marshall-shoppers-guide-2/	1,019	12.97	9 total	72

Unfortunately, you can't sort the PageSpeed Suggestions and PageSpeed Score columns, but you can sort the other three. When tackling issues, sort by the number of page views in descending order and fix issues for your most popular content.

Now that you've learned a bit about PageSpeed Insights and how to use it, you're ready to learn about the tools that live right in your own browser.

2.2. Using browser-based assessment tools

Numerous tools are available in your desktop browser. All browsers ship with a set of developer tools. All of them share functionality, but each has or lacks something in comparison to its competitors. The browsers we touch upon in this section are Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge, with a specific focus on Chrome's Developer Tools.

Similarities in developer tools across browsers

Unless otherwise noted, accessing similar features in different browsers is similar to what's shown in Chrome. Take Firefox, for example: As in Chrome, you'll find timing details by opening the Firefox Developer Tools, clicking the Network tab, and then clicking an entry in the waterfall graph. The same is also true of Microsoft Edge.

Opening the developer tools in any browser is the same. On Windows systems, they can be opened with the F12 key, and on a Mac by pressing Cmd-Alt-I.

The goal of this chapter isn't to be an encyclopedic resource of all the nooks and crannies of every browser's developer tools. Such a resource could easily be its own book. Instead, the goal is to highlight the common aspects in these tools that are available across all browsers, starting with Chrome, while highlighting some of the notable differences in other browsers.

2.3. Inspecting network requests

You'll recall in the client's website from chapter 1 that you used Chrome's network utility to generate a waterfall chart of the site's assets and to measure page load time. Most network inspection tools in the browser work similarly to Chrome's in that they generate waterfall charts, but the functionality only begins there. This section explains how to use the utility to view timing information of individual assets, as well as how to view HTTP headers.

2.3.1. Viewing timing information

At the start of chapter 1, we discussed how web browsers talk with web servers, and the latency inherent in this exchange. All of the steps depicted in figure 2.6 incur latency. One important metric is known as *Time to First Byte* (TTFB), the amount of time between the moment a user requests a web page and the moment the first byte of the response arrives. This is distinct from load time which is the amount of time it takes for an asset to finish downloading altogether. Figure 2.6 (repeated from chapter 1) illustrates this concept.

Figure 2.6. The process of a web browser's request to a web server. Latency occurs in each step of the process. The amount of time between the instant the user makes a request to the time the response arrives is known as Time to First Byte (TTFB).

Causes behind a long TTFB vary. It may be due to network conditions such as the physical distance of the server from the user, poor server performance, or issues in the application back end. The longer it takes for content to start downloading, the longer the user waits.

To find out how long a request is taking, you'll look at how it's done in Chrome, which is similar to how this information can be found in most browsers (save for Safari, which we'll get to later). To start, open the Network tab in the Chrome Developer Tools and follow these steps:

1.   Populate the waterfall graph with data if you haven't already. You can get a detailed walk-through of this in chapter 1, but the easiest way is to reload the site while the developer tools are open and the Network tab is active.

2.   After the waterfall graph is populated, you can click any of the asset entries and view the timing information for it.

After you do this, you'll see something similar to figure 2.7. You can see in the figure that the TTFB value is labeled clearly in Chrome. Prior to the request being made, a few steps occur, such as queueing the request, DNS lookup, connection setup, and the SSL handshake.

Figure 2.7. Timing information for a site asset. The TTFB in this example is 174.56 ms.

A note on DNS lookups

To eliminate latency in DNS lookups, browsers create a DNS lookup cache. If a domain's corresponding IP address isn't in the cache, the IP address lookup will incur latency. On repeat requests, however, the IP address will be cached to eliminate latency in further requests. In Chrome, you can look at the DNS cache by going to chrome://netinternals#dns.

Most browsers allow access to this kind of information in a similar fashion, but Safari is a bit different. To begin with, you may have to enable the developer tools. A quick way to see whether they're enabled is to look for the Develop menu at the top of the screen, shown in figure 2.8.

Figure 2.8. The Safari Developer Tools can be used only if the Develop option is visible in the menu bar when the Safari web browser window is in focus. If you don't see this menu, you have to enable the developer tools.

If the Develop menu isn't visible, click the Safari menu and then Preferences. When the window opens, go to the Advanced tab and select the Show Develop menu in menu bar check box, as shown in figure 2.9. After you've toggled the Develop menu, exit the Preferences window, and open the developer tools by pressing Cmd-Alt-I.

Figure 2.9. You can enable the Safari Developer Tools by choosing Safari > Preferences from the menu bar. In the window that appears, click the Advanced tab and select the check box.

In the developer tools, you can click the Network tab and go to the optimized client website from chapter 1 at http://jlwagner.net/webopt/ch01-exercise-post-optimization. You'll see in figure 2.10 that the Safari version of the Network tab lacks a waterfall graph, but it does show a table of timing data. In this case, you have the Latency, Start Time, and Duration columns.

Figure 2.10. The network request information for a website in the Network tab in Safari's Developer Tools. Note the lack of a waterfall graph in this view in favor of columns for timing information.

The Latency column isn't the same as the TTFB value you see in other browser tools. TTFB doesn't include steps such as DNS lookup and the time spent connecting to the web server. It includes only the time it takes for the request to be made, and when the asset starts to download. Latency includes TTFB, plus all steps in the process prior to the request being made.

In the next section, you'll go further with browser developer tools and use them to inspect HTTP headers for site assets, which allows you to view detailed information in requests for content, and how the server responds to those requests.

2.3.2. Viewing HTTP request and response headers

Another useful aspect of the developer tools across all browsers is the ability to inspect HTTP headers that travel with browser requests for content, and the responses from web servers. In figure 2.11, you see the typical request/response diagram, showing sample headers that accompany the request and the response (albeit with much more brevity than in practice).

Figure 2.11. HTTP headers are sent by the browser in the initial request *and* by the server in its response. In this figure, a simplified set of headers is shown. The network inspection utilities in the developer tools for every browser allow the user to examine these headers.

These headers include basics such as response codes, supported media types, the host of the request, and so on. But headers also can include performance indicators. Figure 2.12 shows how HTTP headers can be viewed in Chrome. Under the Network tab, clicking an asset name reveals its request and response header in a separate pane to the right.

Figure 2.12. Viewing HTTP headers in Chrome's Developer Tools. Accessing HTTP headers for an asset can be done by clicking the asset name. A new pane to the right opens, with the header information contained within the Headers tab.

An example of a performance-related header is the Content-Encoding response header. This header tells you whether a resource is compressed by the web server. When you set up your own server, chances are good that you'll know whether compression is enabled. If you're working in an unfamiliar hosting environment and lack certainty, response headers are the place to check. Figure 2.13 shows response headers for jquery.js in the optimized client website.

Figure 2.13. The `Content-Encoding` response header from the web server lets you know that the asset is compressed, as well as the compression algorithm used (gzip in this example).

▼ Response Headers
    Accept-Ranges: bytes
    Cache-Control: max-age=31536000
    Content-Encoding: gzip

Indicates a compressed asset

When the server compresses content, it replies with a Content-Encoding header. Using the developer tools, you can inspect the response to see this in action. Of course, this isn't all that you'd use this tool for. It's useful for checking headers related to caching, cookies, and other information. Consider header inspection to be one of the many tools in your developer tool belt!

Most tools in other browsers present this information in much the same way as Chrome does. Firefox's Developer Tools use the same flow as Chrome's. Microsoft Edge uses the same flow, but instead of opening the informational window on the right-hand side, it requires the user to open it explicitly by clicking a small toggle button, as seen in .

Figure 2.14. Viewing HTTP headers in Microsoft Edge requires the user to click a small toggle button at the farright side of the window in the Network tab.

Safari also requires the user to toggle the right-hand pane through a small toggle button, in about the same relative location as Microsoft Edge. The end result of these tools is the same: you get to view HTTP headers for site assets, which can be useful for troubleshooting.

In the next section, you'll tackle the task of understanding how browsers render web pages, and how to use the developer tools to audit pages for rendering performance issues.

2.4. Rendering performance-auditing tools

Although minimizing load time is a big concern, another aspect of performance is a page's rendering speed. The initial rendering of a page is important, but it's also important that interactions with web pages after they render are smooth. In this section, you'll learn the process by which pages render. You'll also learn how to use Chrome's Timeline tool, how to spot poor rendering performance, and how to mark points in the timeline with JavaScript. Finally, you'll get an overview of similar tools in other browsers.

2.4.1. Understanding how browsers render web pages

When a user visits a website, the browser interprets the HTML and CSS and renders it to the screen. Figure 2.15 shows a basic overview of this process.

Figure 2.15. The page-rendering process.

In detailed terms, the steps in this process are:

1.   *Parse HTML to create the Document Object Model (DOM)*—When the HTML is downloaded from the web server, it's parsed by the browser to build the DOM, which is a hierarchical representation of the HTML document's structure.

2.   *Parse CSS to create the CSS Object Model (CSSOM)*—After the DOM is built, the browser parses the CSS and creates the CSSOM. This is similar to the DOM, except it represents the way that CSS rules are applied to the document.

3.   *Lay out elements*—The DOM and CSSOM trees are combined to create a render tree. The render tree then goes through the layout process, where CSS rules are applied and elements are laid out on the page to create the UI.

4.   *Paint page*—After the document has finished the layout process, the cosmetic aspects of the page are applied from the CSS and media in the page. At the end of the painting process, the output is converted into pixels (rasterized) and displayed on the screen.

The bulk of the rendering for many sites is done when the page first loads, but more can occur beyond that point. As a user interacts with elements on a page, changes can occur to the page. These changes can trigger re-rendering.

Next, you'll learn how to use the Timeline tool to profile page activity and identify undesirable behaviors that occur on the page.

2.4.2. Using Google Chrome's Timeline tool

Chrome's Timeline tool records the loading, scripting, rendering, and painting activity of a page. It can be daunting at first glance, but this section will help you understand this tool, make sense of the data it collects, and use it to identify performance problems. Figure 2.16 shows an overview of the tool's interface.

Figure 2.16. The Timeline tool in the populated state.

A lot is going on in figure 2.16, so let's take it step by step: to start, browse to an online version of the client website from chapter 1 at http://jlwagner.net/webopt/ch01-exercise-post-optimization and profile it. After the page loads, access the Timeline tool in the developer tools by clicking the Timeline tab. The timeline is empty, so you need to populate it.

Let's start by recording what happens when the page loads. To do this, reload the page by pressing Ctrl-R (Cmd-R on a Mac). When this is done with the Timeline tool in focus, it'll automatically start recording. Once the page has been loaded, you can stop recording by pressing Ctrl-E (Cmd-E on a Mac). Once finished, the timeline populates with data.

You're going to see a lot of data in the activity overview and in the flame chart. The sheer amount of information can be overwhelming, but let's start with the basics. The tool captures four specific types of events, each of them color coded:

- *Loading (Blue)*— Network-related events such as HTTP requests. It also includes activity such as the parsing of HTML, CSS, and image decoding.
- *Scripting (Yellow)*— JavaScript-related events. These can range from DOM-specific activity, to garbage collection, to site-specific JavaScript, and to other activity.
- *Rendering (Purple)*— Any and all events relating to page rendering. Events in this category are activities such as applying CSS to the page HTML, and events that cause re-rendering such as changes to the page's HTML triggered by JavaScript.
- *Painting (Green)*— Events related to drawing the layout to the screen, such as layer compositing and rasterization.

Before diving into the flame chart, let's look at the event summary. This shows the amount of CPU time in the session that was spent in each one of the aforementioned categories. You can see the summary at the bottom of the tool pane under the Summary tab, as shown in figure 2.17.

Figure 2.17. The breakdown of session activity as recorded by the Timeline tool

Range: 110 ms – 645 ms

21.4 ms ■ Loading — Network send/receive events and HTML, image, and CSS parsing activity

264.3 ms ■ Scripting — JavaScript-related activity

30.7 ms ■ Rendering — Layout and reflow activity from CSS, changes to HTML, and changes to CSS by JavaScript

2.7 ms ■ Painting — Drawing and rasterization of the web page to the screen

112.2 ms ■ Other — Activity not able to be broken down by Chrome

104.1 ms ▢ Idle

Total: 535.44 ms

Narrowing things down

When the Timeline tool initially populates with data, it automatically selects a range of time for you. If you want to tweak the range, you can do so by using your mouse wheel to constrict or expand it, or by clicking and dragging its edges in the activity overview panel as shown in <u>figure 2.16</u>. When you adjust the range, you'll notice that the flame chart and summary will also change to reflect the selected portion of activity.

The summary reports the CPU time spent in each event category. In this example, you'll notice that a significant portion of time is spent in scripting and Other activity. The Other category is a type of activity separate from the four events types we covered, and consists of CPU activity that Chrome is unable to break down and present in the flame chart.

Keeping tabs on visible pages

Web browsers excel at budgeting CPU time. If you're running the Timeline tool on a browser tab that's not currently visible, the browser won't spend any time rendering or painting the page. So make sure the tab of the page you wish to profile is the one that's currently visible!

Now onto the flame chart itself. A *flame chart* is a kind of chart used to represent the events that occur in a computer program. With Chrome's Timeline tool, it arranges this data in a call stack. With respect to flame charts, a *call stack* is a hierarchical

representation of recorded page activity. Figure 2.18 shows a call stack of the client website's HTML parsing and of the activity that originates from it.

Figure 2.18. An isolated call stack from the flame chart view in the Timeline tool. The top event is a loading event where the HTML was parsed. Underneath it are events originating from it, such as the `DOMContentLoaded` event that fires when the DOM is ready, and scripting and rendering events.

When you find a call stack in the flame chart that you want to dive into, you can interact with it by clicking its layers. When you click a layer, the summary view at the bottom of the Timeline tool window updates with information specific to the selected event. Figure 2.19 shows information related to the site's behaviors.js script being evaluated by the browser.

Figure 2.19. The breakdown of a scripting event. You can see information related to the event, such as the amount of CPU time used, the event type, and its origination. This data is also visualized in a pie chart.

With this information, you can see what the browser is up to in this specific part of the call stack. Although this is helpful toward achieving familiarity with how the browser works under the hood, it doesn't show you how to identify performance problems on the page. In the next section, you'll look at some of the cues the Timeline tool gives you when pages are being sluggish.

2.4.3. Identifying problem events: thy enemy is jank

Before you can identify performance issues, you have to define your primary goal with respect to page performance. The goal is simple: to minimize the amount of time the browser spends loading and rendering the page. To do this, you must defeat a single enemy: jank.

Jank is the effect of interactions and animations that stutter or otherwise fail to render smoothly. Even a page that loads quickly from the network is subject to the effects of jank if suboptimal programming techniques are used.

So what causes jank? It occurs when too much CPU time is consumed during a single frame. A *frame* is the amount of work the browser does in one frame per second of display time.

When I say *work*, I'm talking about the events described earlier, such as loading, scripting, painting, and rendering.

One janky frame out of many won't cause much trouble, but when frames pile up, the frame rate drops. This can be due to scripts that fire too often, loading events that take too long, and any other activity that causes inefficient or superfluous rendering and painting operations.

Spotting jank

Spotting jank is a difficult task if you're not sure what it looks like. The nature of the printed page doesn't allow for a meaningful visual representation of motion, either. Thankfully, there's a helpful game by Google developer Jake Archibald that helps to train your eyes to recognize jank. It's playable at http://jakearchibald.github.io/jank-invaders.

The optimal frame rate for a typical display is 60 frames per second (FPS), but this isn't always possible on all devices, depending on the hardware capabilities of the device and/or the complexity of the page. Look at this figure as a goal, but know that it may not be possible for every single device to reach.

Frame rate is measured in most developer tools, but the way it's represented is different in each browser. It's shown as a graph in some browsers (as in the activity overview at the top of the Timeline tool in Chrome), but others may represent it only as a number without a visual.

The Timeline tool measures duration in milliseconds. Because there are 60 FPS and 1000 milliseconds in a second, simple math dictates a budget of 16.66 ms per frame. Because the browser has overhead in each frame, Google recommends a 10 ms budget per frame.

To get started on your jank hunt, you'll pick up where you left off with the client website from [chapter 1](). Instead of continuing with the same codebase, you'll download the new starting point from GitHub. Type the following commands in a folder of your choosing:

```
git clone https://github.com/webopt/ch2-jank.git
cd ch2-jank
npm install
node http.js
```

Then navigate to http://localhost:8080 and record a new session. As the session records, click the Schedule Appointment button to launch the scheduling modal. After the modal slides into view, stop recording. The Timeline tool should populate with something that looks similar to [figure 2.20]().

Figure 2.20. A timeline recording of the modal opening on the client website. A range of janky frames is denoted with red markers in the activity overview, and highlighted in red and clickable in the flame chart.

After you stop the recording, you'll see a range of red frames in the activity overview, and specific frame(s) marked in red in the flame chart. If you see this as a bad thing, your instinct serves you well. When you see red on either the activity overview or the flame chart, it's an indicator of low frame rate, which is a precursor to jank. In the flame chart, you can click any of the problem frames you see, and the summary at the bottom of the page will update with a warning about jank, similar to what's shown in figure 2.21.

Figure 2.21. The summary view of a janky frame. Note the explicit warning and the low frame rate.

In this case, the average FPS of this frame is a measly 3. That needs fixing, sure, but how can you figure out what's causing the issue? You know that the modal uses an animation to slide into position, so maybe it's something to do with that. To drill down a bit more,

you can click the Event Log tab shown in <u>figure 2.21</u>. When you click the event log, you'll see every single event related to the frame, as seen in <u>figure 2.22</u>.

Figure 2.22. The event log filtered by scripting events. The text box can be used to filter events by the contents of their activity, filtered by a specific length of time in the duration drop-down, and/or by type.

When you go to the event log, you'll see a lot of events labeled Timer Fired. Anytime a setTimeout or setInterval call is recorded, it's logged with a label of Timer Fired. Because you're seeing a lot of these when the modal slides in, it's likely due to the jQuery animate function. The Timeline tool can be somewhat inscrutable at times when it comes to pinning down the exact origination point of an event, but you know that the appointment scheduling modal element has a click event bound to it that calls animate to bring the modal into view. Let's open behaviors.js in the js folder in your text editor, and look for calls to the animate method. You should see a single call to this method on line 10, and it will look something like this:

```
$(".modal").animate({
    "top": topPlacement
}, 500);
```

animate invokes a timer when it animates properties, and it's this timer that's causing a fair bit of jank. So you need to see what else you can use to make this work a bit better. Because this is a simple, linear animation where the modal is sliding in from the top, it seems a little silly to use jQuery to animate it when CSS transitions are built into the browser. CSS transitions are a technology native to CSS that are perfect for linear animations. Because they're built into the browser, they also have none of the overhead of jQuery, and can perform better than timer-based animations that use setTimeout and setInterval, such as jQuery's animate method.

If you don't know anything about CSS transitions, don't worry. They're covered in detail in [chapter 3](#).

Want to skip ahead?

If you get stuck or feel like skipping ahead, you can do so by entering git checkout -f css-transition and the finished code will be downloaded to your computer. Be sure to back up your work if you have any changes you'd like to keep.

The short version is that the CSS transition property allows you to animate changes in CSS properties (for example, color and width) over a period of time. To fix the jank, you'll use a CSS transition to slide the modal into view, rather than the animate property. To achieve this, you'll use a CSS class that animates the modal's position when it's added or removed.

To start, open styles.css in your text editor and go to line 557, which is the CSS rule for the modal styling for desktop devices. In this rule, perform the following actions:

1.   Change the top property from top: -150%; to this:

transform: translateY(-150%);

2.   Add this property on the next line:

transition: transform .5s;

3.   Add a new CSS rule after the one you just modified:

div.modal.open{
transform: translateY(10%);
}

4.   Inside the styling breakpoint for mobile devices near line 1040, you need to add a version of the same rule that's styled for mobile devices:

div.modal.open{
transform: translateY(0);
}

Let's go over these steps. Rather than use the top property to position the element, you've changed this to a transform property by using the translateY method. Like the top property, this transform method repositions the element on the y-axis. The difference, though, is that transforms animate better, and with less jank, which is what you're after.

Next, you add a transition property that works on the element's transform property. Changes in this property are animated with a half-second duration. To top it off, you confine the changes on the transform property to a separate class named open. When this class is toggled on the div.modal element, the position of the modal animates so that it enters the viewport when the class is added, and exits the viewport when it's removed.

All that's left is to update the JavaScript to add/remove this class when the modal is opened and closed. This involves changing two pieces of JavaScript in behaviors.js:

1.   In the openModal function, you'll see the following call to the animate function:

```
$(".modal").animate({
"top": topPlacement
}, 500);
```

2.   This is the animate call that's responsible for the jank you want to fix. You'll replace this to add the open class to the div.modal element, which causes the transition property to kick in and slide the modal into view:

```
$(".modal").addClass("open");
```

3.   Update the closeModal function to remove the open class from the element to reverse the effect when the user dismisses the modal. To do this, replace this code

```
$(".modal").hide(0, function(){
$(".modal").removeAttr("style");
});
```

with this code:

```
$(".modal").removeClass("open");
```

Next, test to ensure that the modal still works. Then retest in the Timeline tool to see how the new code performs. Your results should be similar to figure 2.23.

Figure 2.23. Modal animation performance after CSS transitions have been implemented. Janky frames still exist, but much less so than before, resulting in an overall improved experience.

Do janky frames still exist in the animation? Sure, but they're reduced, and the animation is improved overall. Moreover, the activity summary shows reduced CPU usage. Figure 2.24 shows CPU usage with jQuery animations versus the CSS transition you've put in place.

Figure 2.24. CPU usage summary of jQuery animations (left) compared to CSS transitions (right)

As a result of converting the jQuery animation to a simple CSS transition, you've not only solved the jittery animation problem of the modal, but also saved CPU time in the process.

Are CSS transitions *always* the solution for an animation requirement? Nope. At times your requirements need more than what CSS transitions can provide, but CSS transitions are a great solution for linear transitions. Because they're a part of CSS, no additional overhead is incurred. Chapter 3 goes over CSS transitions in more detail. For now, you'll move on to how to mark specific points in timelines by using JavaScript.

2.4.4. Marking points in the timeline with JavaScript

On websites with lots of activity, it can be hard to pry out what you're looking for with the Timeline tool. It *can* be easier to find specific events if a site doesn't have much going on. If a flurry of activity is taking place, however, finding what you're after can be a whole other story.

Thankfully, Chrome's Developer Tools allow developers to mark parts of a timeline via JavaScript. This can be done with the console object's timeStamp method which takes one argument, a string that labels the marker on the timeline. It's akin to a mile marker on a highway. You can invoke this method in either the console or in your site's JavaScript.

To try this out, open behaviors.js in the js folder of the exercise you worked on, and go to line 33. This line should contain a jQuery click event binding that opens the scheduling modal:

```
$("#schedule").click(function(){
```

Inside the function for this event handler, add a new line with the following code:

```
console.timeStamp("Modal open.");
```

While a new session is being recorded, this method will place a marker on the timeline when you launch the scheduling modal. When you stop recording and select the entire range of the recording, a yellow marker above the flame chart appears, as illustrated in figure 2.25.

Figure 2.25. A marker added to the timeline. The associated call stack is selected, and the timestamp event call is shown in the event log.

By digging near this marker in the call stacks, you can find the corresponding layer in the stack and find the marker event in the event log. By using markers, you can narrow down what you're looking for to a specific time without having to slog through the entire set of data.

Next, you'll take a quick look at rendering profilers in other browsers and compare them to Chrome's own profiler.

2.4.5. Rendering profilers in other browsers

Other browsers have their own equivalents of Chrome's Timeline tool. The way you use them is usually the same: you can either reload a page or press a key combination (usually Ctrl-E for Windows and Cmd-E for Mac) to begin recording a session, and press the same key combination to stop recording.

Firefox's tool is similar, except that instead of the tool being under a tab labeled Timeline, it's labeled Performance. From there, it's used the same way. A timeline overview shows the frame rate of the application as well as useful statistics such as minimum, maximum, and average frame rate of the session. In addition to a flame chart, the session data is viewable as a waterfall chart.

Edge's profiling tool is similar to both Chrome's and Firefox's, but boasts a bit more specificity in a well-designed UI, as you can see in figure 2.26. Like Firefox, it too resides under a tab labeled Performance and has a timeline overview showing the frame rate. The major difference is that Edge breaks performance into segments, which shows you what the CPU was doing in each rendering frame. Chrome and Firefox eschew this approach for a much more fluid representation of the data.

Figure 2.26. An annotated overview of Microsoft Edge's performance profiler

A big plus is that all browsers covered in this chapter support the console.timeStamp method for marking the timeline. So no matter what tool you use, you can label a point in the session to help you find the activity you're looking for. Next, you'll learn how to use the console object in Chrome to benchmark snippets of JavaScript code.

2.5. Benchmarking JavaScript in Chrome

Benchmarking JavaScript gives you the ability to compare approaches to the problems you're trying to solve, and tease out which is the best performing. By choosing the best-performing solutions, you'll be creating pages that will render faster and respond more quickly to user input.

The console object in most browsers gives you the ability to benchmark code by using the time and timeEnd methods. These methods accept a string that's used to label the benchmark session, similar to the timeStamp method. To demonstrate how to use this feature, you'll open the jank exercise from earlier in this chapter and play around in the console in Chrome's Developer Tools. To access the console, click the Console tab.

A typical use case of the time and timeEnd methods is to compare the execution time of two pieces of code. In this example, you'll compare the speed of jQuery's selection of a DOM element versus that of JavaScript's native document.querySelector method.

To get started, reopen the jank exercise from before, and run the following two command batches in the console:

1.

console.time("jQuery"); jQuery("#schedule"); console.timeEnd("jQuery");

2.

console.time("querySelector"); document.querySelector("#schedule");
console .timeEnd("querySelector");

Note that the string parameters sent to the time and timeEnd methods are identical for each test. The string you enter is a label for the session. For the benchmark to terminate, the string label you use in the time method *must* be the same as the one you use in the timeEnd method. When you run these two benchmarks, the console output should look something like <u>figure 2.27</u>.

Figure 2.27. The results of two benchmarks you've run of jQuery's DOM selection versus that of the native `document.querySelector` **method. Results are circled.**

As you can see, the benchmark results appear in the console. In this instance, you can see that the document.querySelector method is faster than jQuery's own CSS selector engine. This isn't surprising, because native JavaScript methods are usually faster than user-defined ones.

Benchmarking tip

When benchmarking, it's important to understand that a single test result isn't enough. You should run multiple sessions and average the results for the best possible accuracy.

Benchmarking in the console is great for small tests, but it's impractical when you have larger pieces of code you need to evaluate. To get around this, use the time and timeEnd methods in your application JavaScript, and the output will appear in the console when the code executes. It's also worth noting that this method is available across the four browsers covered in this chapter, and the way they're used is the same regardless of the platform.

Next, you'll learn how to use Chrome's Device Mode to simulate the appearance of websites on various devices, such as tablets and phones, as well as how to inspect pages on physical devices and monitor their behavior.

2.6. Simulating and monitoring devices

As a developer, you spend a lot of time doing the initial testing for your websites in a desktop environment. This is typical, but further testing should be done with tools that simulate how your pages might look on mobile devices, and finally, on actual physical devices. This testing can range from cursory style checks across CSS breakpoints, or performance testing on real devices. In this section, you'll learn how to do both.

2.6.1. Simulating devices in the desktop web browser

The most simple way of checking your website's appearance is by using device simulation tools across desktop web browsers. These tools cover only high-level characteristics such as device resolution and pixel density.

In Chrome, it's easy to use Device Mode. To try it, you'll navigate to a website—in this case, the Manning Publications website at www.manning.com. With the developer tools open, you can hit the Ctrl-Shift-M key combination (Cmd-Shift-M on a Mac), or click the mobile device icon to the left of the Elements tab. When you do this, the interface changes, as shown in figure 2.28.

Figure 2.28. The device simulation mode in Chrome viewing the Manning Publications website

From this interface, you can pick a device profile from the drop-down list of presets, and simulate the characteristics of a selected preset in the current page. As you can see in figure 2.28, you have several things to tinker with. You can switch to a canned device profile (for example, iPhone or Galaxy Nexus), key in a custom resolution, change the device pixel ratio to debug issues related to high-density displays, and more.

Other web browsers have similar utilities. Safari has an iOS-centric device simulation utility called Responsive Design Mode. You invoke this mode from the Develop menu by clicking Enter Responsive Design Mode, or by hitting Alt-Cmd-R. This utility is similar to Chrome's in capability, but with a different UI. Firefox's Responsive Design Mode is similar to

Chrome's, but with fewer options overall. Edge is similar as well, and focuses on simulating Microsoft-centric mobile devices and Internet Explorer.

Although simulating devices in your desktop browser can be useful, don't forget to test on mobile devices to catch problems that browser-based tools may miss. Next, you'll learn how to attach Android devices and monitor their activity in the desktop version of Chrome.

2.6.2. Debugging websites remotely on Android devices

Sometimes you need to test your site on a real device. Browser-based tools such as the ones covered in the previous section are great for debugging and performance profiling, but desktop devices have much more memory and processing power to work with than mobile devices. It's important to test on the real thing to see whether performance problems exist on those platforms.

The way to do this is to connect your mobile device to your desktop computer, and debug it by using the developer tools in one of the browsers. The way this is accomplished depends on the device you have. For Android devices, you'll use Chrome.

Chrome calls this feature *remote debugging*. To use it, connect your Android device to your machine with a USB cable, and open Chrome on both your mobile and desktop devices. Follow these directions, and your Android device will show up in the device list in Chrome's remote debugger on your desktop, as shown in figure 2.29.

Figure 2.29. The Chrome device list showing an open web page on a connected Android phone
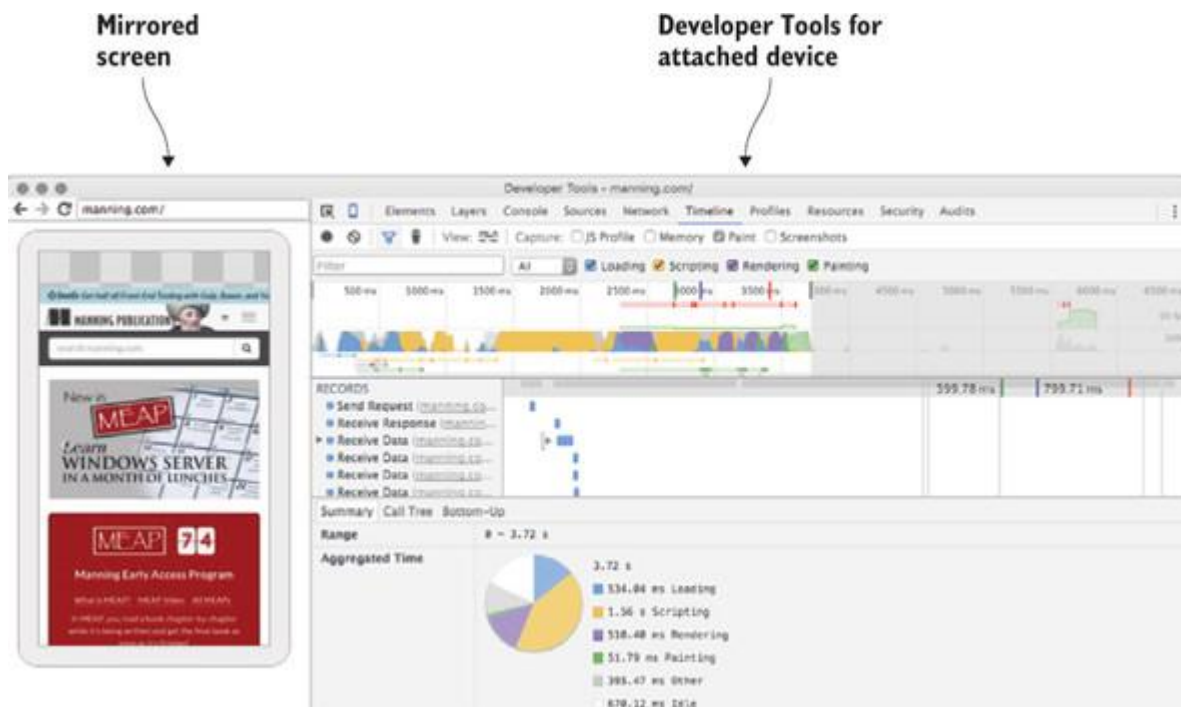

To get started with remote debugging, complete the following steps:

1.   *Enable the developer options on the Android device*—This entails choosing Settings > About Device and tapping the build number field seven times (seriously).

2.   *Enable USB debugging*—On the Android device, choose Settings > Developer Options and then select the USB Debugging check box.

3.   *Allow device authorization*—In Chrome on your desktop, go to the URL chrome://inspect#devices and ensure that the Discover USB Devices check box is selected. This enables you to receive an authorization request inside Chrome on the attached device. Tap OK to accept it.

4.   *Inspect the web page open on the device*—After a device appears in the device list as shown in figure 2.29, click the Inspect link underneath the device in the list.

After all of this rigmarole, the developer tools will launch on the desktop machine. The window that pops up is identical to tools you're used to seeing, except that the device's screen is mirrored in a pane to the left, as shown in figure 2.30.

Figure 2.30. The Developer Tools profiling rendering activity of a page on an Android phone. In this view, the device's display is mirrored on the host machine, and the Developer Tools are focused on the device's current page rather than a session active on the desktop.

When the remote debugging session is active, you can do anything that you normally did with the developer tools on desktop sessions, except now context of the tools is that of a session on your Android phone.

When your Android device is connected and the developer tools are open on the host device, try things like benchmarking load times on your mobile network connection, or using the Timeline tool to see how your device performs. With the same knowledge you used throughout this chapter for Chrome, you can do any of those things, but now for the attached device!

Next, you'll learn how to debug mobile devices by using Safari on a Mac, and Mobile Safari on an iOS device.

2.6.3. Debugging websites remotely on iOS devices

You can also debug pages on iOS devices, and it's simpler than remote debugging websites on Android phones. First, connect your iOS device to your Mac with a USB cable, and instead of using Chrome, you'll use Safari on both the desktop and the mobile device. After you have Safari open on both, go to www.manning.com on the attached device and follow along:

1.   *Authorize your Mac to access your device*—In Safari on your Mac, go to the Develop menu and you'll see the name of your iOS device (for example, Jeremy's iPhone). Underneath that menu, you'll see the Use for Development option. Click this option and you'll see a prompt on your iOS device to trust the computer that it's connected to. Tap the Trust option to allow your Mac to communicate with the device.

2.   *Inspect the web page open on the device*—After authorizing your Mac, go back to the Develop menu, choose your device, and in that submenu you'll see a list of the web pages that are open in Safari on the attached device. Choose the device that's focused on the Manning Publications page.

After you choose from the list of pages on the iOS device, the developer tools will launch for that website. As in remote debugging with Android devices in Chrome, you can use any of the tools available for debugging pages on your desktop to find performance issues on web pages on your iOS device.
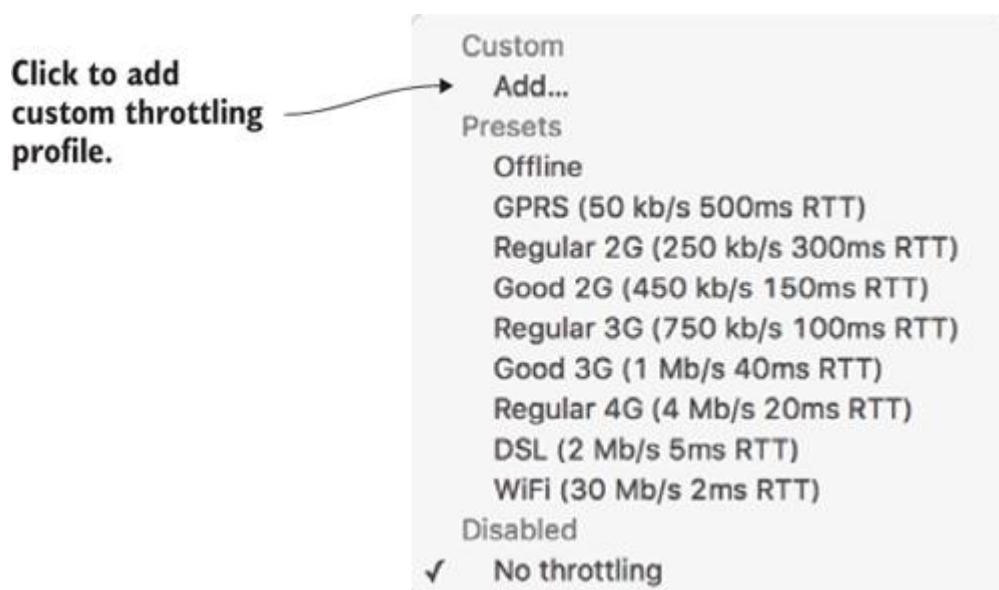
2.7. Creating custom network throttling profiles

Early in chapter 1, you used the network throttling tool in Chrome. This tool allows you to simulate certain internet connections, such as 3G or 4G connections. This is valuable for determining page-load times in scenarios you may not be able to otherwise replicate.

Out of the four browsers covered in this book, Chrome is the only one that has this function. Because [chapter 1](#) covered how to use the throttling tool, we'll go over how to further extend its usefulness by defining a custom profile.

Using the presets that ship with Chrome allows you to approximate the performance of many internet connection types. Unless you have to test for a specific scenario, the throttling presets that are built in will suffice for most situations. It's especially useful for performance testing on sites running on your local computer, which run without network bottlenecks.

If you do need to test for a specific scenario, you can add a custom profile via the Add option, as seen in [figure 2.31](#). Click this, and you'll be sent to the Network Throttling Profiles settings screen, where you can add a new profile by clicking the Add Custom Profile button. When you do this, a screen like [figure 2.32](#) appears.

Figure 2.31. The throttling profiles that ship with Chrome, with the option to add custom profiles

Figure 2.32. Adding a new throttling profile in Chrome. The profile requires four bits of information: a profile name, the download and upload speeds (inKbits/sec), and the latency in milliseconds.
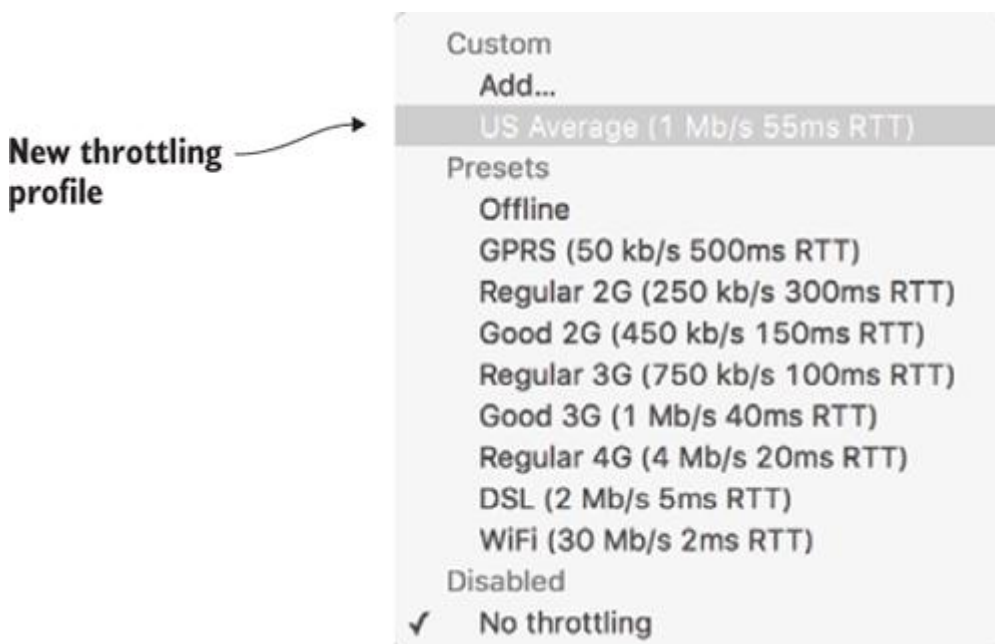

This screen displays the following set of fields:

- *Profile Name*— A name for the profile. What you enter here will appear by this name in the throttling profile drop-down list.
- *Throughput*— The connection speed of the profile in kilobytes.
- *Latency*— The connection latency of the profile in milliseconds.

After the profile has been added, it'll will be visible in the drop-down list, as shown in figure 2.33. Now you can use your custom profile and see how it affects the load times of websites. When you use it, watch the Network tab as sites load to see your new profile in action.

Figure 2.33. Your new custom network throttling profile is now in the list.

Now that you have a handle on how to use the performance assessment tools in different browsers, we'll bring this chapter to a close with a short summary of the techniques you learned.

2.8. Summary

We covered a lot in this chapter about some of the performance assessment tools out there, but it looks like you made it! Let's take stock of what you've learned:

- Google PageSpeed Insights is a useful online tool that analyzes a URL and gives you a list of performance issues for that URL that you can act upon to make your site faster.
- Although it's useful, PageSpeed Insights can analyze only one URL at a time. If you need a bulk assessment of your site, you can turn to Google Analytics, which provides PageSpeed Insights reports for all pages on a particular site.
- Gathering timing information on network requests can be done in most every browser's set of developer tools. This information allows you to examine how long it takes for a given asset on a site to download, and breaks down that period into specific stages that you can use to diagnose server performance issues.
- Developer tools in all browsers allow you to examine HTTP request and response headers. You can use this information to examine many aspects of requests and responses, including performance indicators such as server compression headers.
- Chrome's Timeline tool enables you to record a period of time and examine the various types of activity that occur. Using this information, you can identify the activity causing performance problems and then set about fixing those issues in your code. You can also use JavaScript to mark specific points in the timeline to help you nail down a time in a recording that you want to examine.
- Using JavaScript, you can perform simple benchmarking via the `console` object's `time` and `timeEnd` methods. This allows you to quantify the amount of time that a piece of JavaScript takes to execute.
- In various browsers, you can simulate the characteristics of mobile devices inside the browser itself. Using these tools, you can get a quick idea of how any given page might look on a similar device.
- In Chrome and Safari, you have the ability to inspect open pages on Android and iOS devices, respectively. When these devices are attached, you can use the developer tools on the host computer to find and diagnose performance issues.
- Chrome's network throttling utility comes with useful presets, but you can create your own custom profiles that enable you to simulate specific network conditions that the canned presets may not cover.

With this chapter at an end, you can now move toward optimizing specific parts of your site. In chapter 3, you'll begin with some useful tips and methods for optimizing your site's CSS.

Chapter 3. Optimizing CSS

This chapter covers

- Reducing the size of your CSS by taking advantage of shorthand CSS properties, using shallow CSS selectors, and implementing the DRY principle
- Segmenting your CSS by using unique page templates
- Understanding the importance of mobile-first responsive web design
- Knowing what makes a page mobile-friendly, and how this matters to Google search rankings
- Improving the performance of your CSS by avoiding bad practices and using higher-performing CSS selectors, the flexbox layout engine, and CSS transitions

Now that you've learned how to assess performance by using developer tools available in the browser, you can learn to optimize the various aspects of your websites. This starts with optimizing your CSS. In this chapter, you'll learn to write efficient CSS, understand the importance of mobile-first responsive design, and gain tips for performance-tuning your CSS.
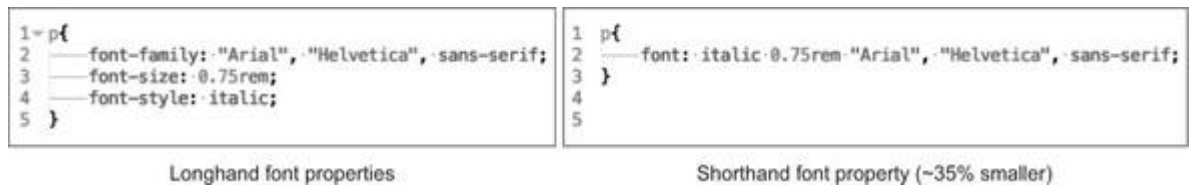
3.1. Don't talk much and stay DRY

When you begin learning a new topic in the realm of web development, the first thought is, "What's new and shiny?" Although the topic of web performance certainly brings new tools to the table for CSS, the best piece of advice is to be as terse as possible when you write CSS. Doing this requires no new tools, only the desire to learn terser expressions and the discipline to use them consistently.

In this section, you'll see the importance of keeping your CSS properties and selectors terse. You'll learn how to remove redundant CSS as part of the DRY (don't repeat yourself) principle, explore the potential benefits of segmenting CSS on your site, and keep your site's frameworks as slim as possible by customizing framework downloads.

3.1.1. Write shorthand CSS

Using *shorthand CSS* means using the least verbose properties and values where possible. This approach doesn't save you a ton in the short term, but when used consistently in large style sheets, it can add up. In figure 3.1, for example, the rule on the left uses a set of verbose typography styles that takes up 94 bytes, and the rule on the right combines them into a single font property that takes up 60 bytes.

Figure 3.1. An example of shorthand CSS via the `font` property

Longhand font properties   Shorthand font property (~35% smaller)

Although a gain of 34 bytes isn't hugely significant on its own, consistent use of this approach in projects with large style sheets has the potential to save considerable space. Space saved equals fewer bytes transferred over the wire. That translates to less time the user spends waiting for your website to load. This is *especially* true of mobile connections, which tend to be slower than broadband internet connection speeds you experience in your home or office.

Let's look at a website that you can improve with shorthand properties. This website is, once again, the Coyle Appliance Repair website from the previous two chapters. When you apply shorthand properties to its CSS, you'll further reduce its size by 28%.

Download and run the client's website on your local machine by running these commands in a folder of your choosing:
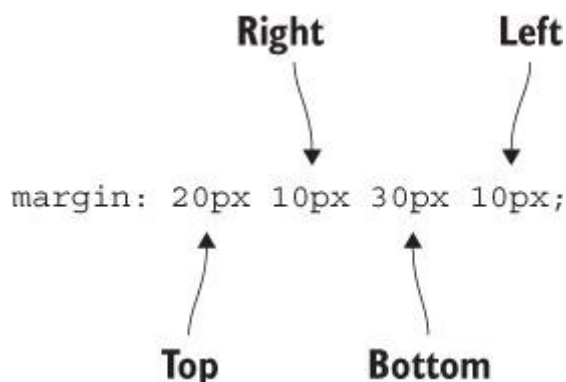
```
git clone https://github.com/webopt/ch3-css.git
cd ch3-css
npm install
node http.js
```

To get started, open styles.css in the css folder. You'll start by using a couple of shorthand properties that are easy to use and understand. In the style sheet, search for the div.pageWrapper selector, which should look like this:

```
div.pageWrapper{
    width: 100%;
    max-width: 906px;
    margin-top: 0;
    margin-right: auto;
    margin-bottom: 0;
    margin-left: auto;
}
```

Seems reasonable enough, right? You're setting margins for this element, but there's a much terser way to express this same CSS. The first shorthand property you'll start with is the margin property. Figure 3.2 shows this property and how it works.

Figure 3.2. The `margin` shorthand property takes one to four values: those for `margin-top`, `marginright`, `margin-bottom`, and `margin-left`.

The margin property is a replacement for the margin-top, margin-right, margin-bottom, and margin-left properties. The padding property works the same way with respect to its specific properties (for example, padding-top). Not all four arguments for shorthand rules such as margin and padding are required. The number that you can omit depends on the number of values that are unique. The following is a guide for shorthand properties using this syntax:

- *Use one value* when all four sides of an element have the same value. If all four sides of an element have a margin of $20px$, you can abbreviate to margin: $20px$;.
- *Use two values* when the top/bottom and right/left values are the same. If an element has a margin of $10px$ on the top and bottom sides, and $20px$ on the right and left sides, you can abbreviate to margin: $10px$ $20px$;.
- *Use three values* when only the right/left values are the same, but the top and bottom values are different. If an element has a top margin of $10px$, a right/left margin of $20px$, and a bottom margin of $30px$, you can write margin: 10px 20px 30px;.
- *Use all four values* when all of the values are unique.

With this approach, you can reduce the contents of the div.pageWrapper selector as follows:

```
div.pageWrapper{
    width: 100%;
    max-width: 906px;
    margin: 0 auto;
}
```

Here you're taking four properties that say, "Set the top/bottom margins of the div.pageWrapper element to 0, and the left/right margins to auto" and reducing them to one property that says the same thing. An atypical use of this property that you can apply in styles.css would be setting three values when the margin-right and margin-left values are the same, but the margin-top and margin-bottom values are unique. Take this CSS:

```
header div.phoneNumber h1.number{
```

```
    font-size: 55px;
    font-weight: normal;
    color: #fff;
    margin-top: 0;
    margin-right: 0;
    margin-bottom: -8px;
    margin-left: 0;
}
```

This rule can be expressed more succinctly:

```
header div.phoneNumber h1.number{
    font-size: 55px;
    font-weight: normal;
    color: #fff;
    margin: 0 0 -8px;
}
```

You can omit the last 0 in this shorthand, because the margin-left and margin-right values
are the same. This can seem unintuitive at first, but with practice, it becomes second
nature.

margin and padding are the easiest of the shorthand properties to understand, because the
only aspects they control are spacing and dimensions. Other shorthand properties exist for
visual elements such as borders. For example, take the a img selector in the client site
CSS:

```
a img{
    border-top: 0;
    border-right: 0;
    border-bottom: 0;
    border-left: 0;
}
```

You can express these border styles in a much more succinct way:

```
a img{
    border: 0;
}
```

Condensing these border properties into a single one not only culls unnecessary styles,
but also is more convenient. Be aware that unlike shorthand properties such as margin and
padding, the border property can be used only to set *all* borders on an element. If any side
of the element requires a different border style, you need to use the more specific
border-top, border-right, border-bottom, and border-left properties.

The shorthand properties you'll use in cleaning up the client's site CSS are margin, padding, border, background, and border-radius. These are only a handful among many available, and you can find a more complete list of them via Google. My preferred resource is on the W3C wiki at www.w3.org/community/webed/wiki/CSS_shorthand_reference.

Work your way through these properties and shorten what you can. I was able to reduce the CSS from its original size of 18.5 KB to 13.33 KB. If you get stuck and want to see how I did the work, you can skip ahead by entering git checkout -f shorthand, and the finished code will be downloaded to your computer.
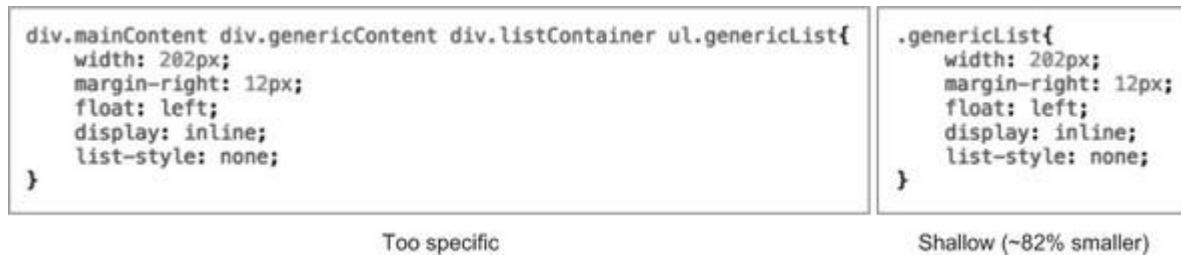
Now that you have a sense of how to write terser CSS with shorthand properties, you'll be able to keep your CSS files slimmer by simple discipline. Next, you'll learn about the importance of shallow CSS selectors, and how they can also contribute to a leaner style sheet.

3.1.2. Use shallow CSS selectors

If there's ever a time where shallowness is a virtue, it's when you write CSS. *Shallowness* refers to the specificity of a CSS selector. Overly specific selectors are those that are many levels deep, whereas shallow selectors are less so, specifying only what's necessary to match an element.

In big style sheets, keeping CSS selectors brief can save space. By reducing complexity, you can keep style sheets lean and load times low, thus boosting the page's performance. In figure 3.3, you can see an overqualified selector compared to a shallower one for the same element.

Figure 3.3. An example of an overly specific CSS selector (left) versus a more succinct one (right). The selector at the left is 67 characters, whereas the one at the right is at 12 characters.


```
div.mainContent div.genericContent div.listContainer ul.genericList{
    width: 202px;
    margin-right: 12px;
    float: left;
    display: inline;
    list-style: none;
}
```
Too specific

```
.genericList{
    width: 202px;
    margin-right: 12px;
    float: left;
    display: inline;
    list-style: none;
}
```
Shallow (~82% smaller)

Although this example represents a reduction of only 55 characters, this is for only a single selector. When applied across an entire style sheet, the benefits become more apparent.

3.1.3. Culling shallow selectors

A way to check for overqualified selectors is to scan your CSS for selectors that have more than one element. Ideally, your selector depth should be limited to only the element you're targeting. Although this isn't always practical, you should strive for as little specificity as possible.

Continuing from where you left off, open styles.css from the css folder and poke around. As you can see, most of the selectors are far too specific. The CSS weighs in at around 13.3 KB. This isn't massive, but considering the specific nature of the selectors, you can stand to whittle this down. When you finish this section, you'll have reduced the client's site CSS by 38%.

An example of a selector that you can make less specific can be found by searching styles.css for the div.marqueClass selector. The full selector on this line is written as follows:

header div.phoneNumber h3.numberHeader

This could be rewritten to something much shorter:

.numberHeader

After you make this change, save and reload the page. Note that the page looks the same. Repeat this process throughout the entire CSS file from top to bottom, eliminating all the specificity from the selectors that you can find without breaking the page. Whenever you make substantial changes, reload the page in your browser and verify that nothing has been

broken. If you get stuck or want to see the end result, you can do so by entering git checkout -f selectors; the finished code will be downloaded to your computer.

After you've finished, you can go a bit further by using a Node program called uncss to remove all the unused CSS from the style sheet. With these two commands, you can install the program globally and run it against the CSS file from the root folder of the client's site:

```
npm install -g uncss
uncss http://localhost:8080 -i .modal.open > css/styles.clean.css
```

This command takes an argument for a URL. In this case, you're telling the program to look at the client website that you're running locally. The -i option is an argument you use to tell the program which selectors you should keep. In this case, you want uncss to leave alone the .modal.open class that slides the modal window into view.

When uncss finishes, you can either switch the <link> tag in index.html over to the newly generated styles.clean.css, or copy the contents of it into styles.css. When you've done this step, the client site's CSS will have been pruned by 38%, from 13.24 KB to 8.2 KB.

3.1.4. LESS is more and taming SASS

CSS precompilers feature prominently in the front-end developer's toolkit. Precompilers provide features not available in plain CSS, including variables, functions (called *mixins*) for reuse of styles, and importing capabilities to help make your CSS more modular. These tools then compile files written in the precompiler language down to plain CSS that can be understood by the browser. Popular precompilers are LESS (http://lesscss.org) and SASS (http://sass-lang.com).
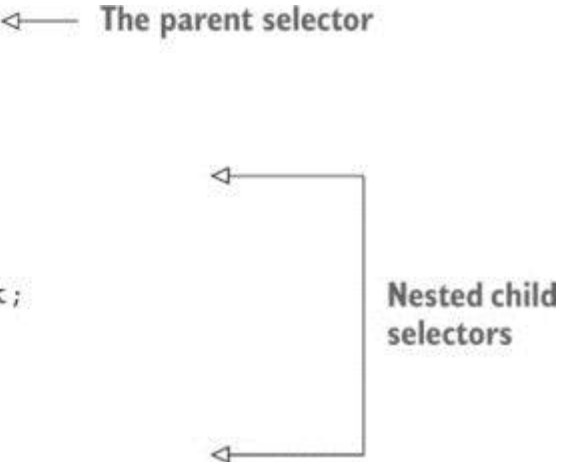
If you use these tools instead of writing plain CSS, you may be taking advantage of a nested selector feature.

Listing 3.1. LESS and SASS selector nesting

```
#main{                              ◁──── The parent selector
     max-width: 1280px;
     width: 100%;

     #mainColumn{                                ◁────────┐
           width: 65%;
           margin: 0 2% 0 0;
           display: inline-block;               Nested child
           float: left;                         selectors
     }

     #sideColumn{                                ◁────────┘
           width: 33%;
           display: inline-block;
           float: left;
     }
}
```

This looks nice, but it's more of a service to the developer than anything else. It *is* more readable, because it mimics the hierarchical structure of the HTML, but this convenience comes at a performance cost. When this code is compiled into plain CSS, it looks like this.

Listing 3.2. LESS/SASS nested selectors after compilation

```
#main{
    max-width: 1280px;
    width: 100%;
}

#main #mainColumn{
    width: 65%;
    margin: 0 2% 0 0;
    display: inline-block;
    float: left;
}

#main #sideColumn{
    width: 33%;
    display: inline-block;
    float: left;
}
```

After compilation, the CSS selectors are too specific because of the nesting in the original LESS/SASS code. In this case, *every* child of #main is now too specific. The deeper this nesting goes, the more problematic it'll be. Compression and minification *do* mitigate this somewhat, but these overly specific selectors can slow rendering time as well. Limit your use of this feature as much as practically possible, because what you can't see *can* hurt you.
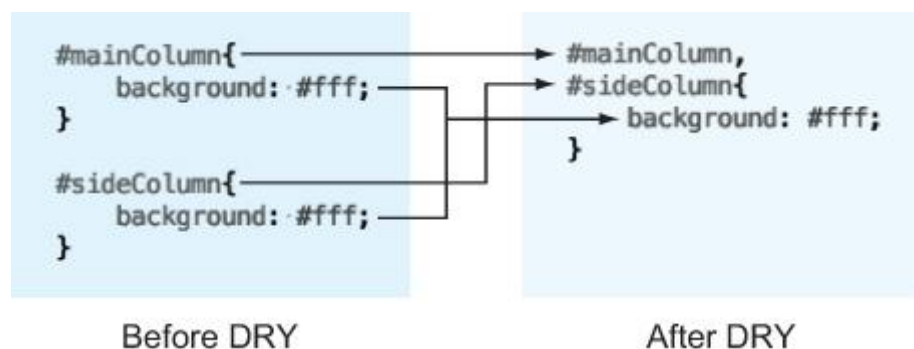
3.1.5. Don't repeat yourself

Another problem that front-end developers encounter in CSS is that properties are often duplicated across selectors. An example is multiple selectors that specify the same background color or font style. By minimizing the number of times a property is declared, you can cut down on bloat and make your CSS more maintainable.

3.1.6. Going DRY

The DRY principle is simple in that it seeks to reduce redundancy in CSS wherever practical and possible. <u>Figure 3.4</u> shows DRY in action.

Figure 3.4. An example of the DRY principle. Two selectors have the same background property. To save space and eliminate redundancy, the background property and the selectors are combined.

This example illustrates a basic application of DRY. Two selectors contain identical background rules. DRY dictates that you should combine these not only to save space, but also to provide increased maintainability. One method for finding redundancy is to look at common rules and combine them under multiple selectors.

If you're familiar with your project's CSS, this isn't a bad way of approaching things. You can make a list of common properties and selectors used in the project, and group them

in a way that makes sense to you. Your approach depends on the methodology that you prefer. Some, including myself, prefer to use selector names that describe the content (for example, #navigation or .siteHeader) rather than those that describe the document's structure (for example, Bootstrap's .col-md-1, .col-md-offset-3, and similar selector names). The HTML 5.1 draft specification encourages developers to choose selector names that describe the nature of the content it applies to rather than its presentation.

Some developers prefer a presentational style. The popular CSS framework Bootstrap makes heavy use of this style for selector names in its CSS. However you decide to write your CSS, the good news is that neither method prevents you from using DRY.

Unfortunately, finding redundancies can be an unwieldy task on its own. That's where a CSS redundancy checker comes in handy.

3.1.7. Finding redundancies with csscss

csscss is a command-line tool that finds redundancies in your CSS. It's a good place to start when refactoring your CSS. To install csscss, you need Ruby's gem installer, which is similar to Node's npm executable, but for Ruby packages. OS X comes preloaded with Ruby. If you have SASS installed, gem is already available to you.

If you don't have Ruby installed, doing so is a menial task. Moreover, the value that csscss provides is worth the effort. To install Ruby on Windows, go to http://rubyinstaller.org/downloads and grab the installer that's right for your system. Installing the software is a simple and guided process. After Ruby is installed, you can install csscss with gem by typing in the following command:

gem install csscss

After a moment, the gem package manager will install csscss, and you'll be able to run it against a CSS file. Try running it on styles.css from the client's site:

csscss styles.css -v --no-match-shorthand

This command examines styles.css for redundant rules by using two arguments. The -v argument tells the program to be verbose and print out the matching rules. The --no-match-shorthand argument keeps the program from expanding any matching shorthand rules such as border-bottom into more-explicit rules such as border-bottom-style. If you want to expand those rules, remove that switch. The program output will show all redundant styles across multiple elements. This listing is an example of one of these rules.

Listing 3.3. A portion of `csscss` output

```
{#okayButton}, {#schedule} AND {.submitAppointment a} share 12 declarations
  - background: #c40a0a
  - border-bottom: 4px solid #630505
  - border-radius: 8px
  - color: #fff
  - display: inline-block
  - font-size: 20px
  - font-weight: 700
  - letter-spacing: -0.5px
  - line-height: 22px
  - padding: 12px 16px
  - text-decoration: none
  - text-transform: uppercase
```

This rule is a good one to start with, because the CSS for these selectors is consistent on all devices. From here, you'll employ a lather-rinse-repeat methodology starting from the top. By the end of this short exercise, you'll be able to shave off an additional 10% from styles.css. Starting with the rule in listing 3.3 as an example, do the following:

1.   *Combine selectors and rules*—Combine the selectors #okayButton, #schedule, and .submitAppointment a into a single, comma-separated selector, and copy/paste the suggested rules from the program output. When you create the new rule at the end of styles.css, it should look like this.

Listing 3.4. Combined CSS rule from `csscss` output

```
#okayButton,
#schedule,
.submitAppointment a{
background: #c40a0a;
border-bottom: 4px solid #630505;
border-radius: 8px;
color: #fff;
display: inline-block;
font-size: 20px;
font-weight: 700;
letter-spacing: -0.5px;
line-height: 22px;
padding: 12px 16px;
text-decoration: none;
```

```
text-transform: uppercase;
}
```

2.   *Clean up the matching rules from the individual selectors*—Go back and remove the redundant rules from the original #okayButton, #schedule, and .submitAppointment a selectors.

3.   *Rerun* csscss*, examine the output, and repeat*—After you've cleaned up the redundant rules in the old selectors, rerun csscss to verify that the rule you've optimized is stricken from the list.

In some of the suggestions that csscss makes, you'll notice that rules are duplicated or in conflict with one another in different breakpoints because the CSS for those elements change as screen width does. The following listing shows a suggestion that illustrates this problem.

Listing 3.5. Problematic csscss output

This rule returns different redundancies for the same properties. This occurs because csscss sees a redundancy in one breakpoint for the desktop CSS, and then another in the mobile CSS. You can attempt to combine these values, but it may be an unwieldy task to do so. The best approach for responsive sites is to combine values that are common across all breakpoints.

After you've whittled down this list and csscss runs out of suggestions, you'll have reduced the CSS by an additional 10% to 7.42 KB. Your mileage may vary on different projects, but 10% saved is no small portion. As a result of your efforts since the beginning of the chapter, you've reduced the site's CSS by roughly 60%, from 18.5 KB to 7.42 KB. Not too shabby! In the next section, you'll learn about the importance of segmenting your CSS.

3.1.8. Segment CSS

One way to optimize your CSS is to segment it. *Segmentation* splits up CSS by styles specific to particular page templates. It *can* make sense to combine all of your site's CSS into one file so that the user already has all of the site's CSS cached on the first visit.

Serving your CSS this way can be a gamble, however, because your users may never navigate to subpages. A portion of your users would be forced to download CSS for pages they'll never see. This slows the initial visit to your site. The safer bet is to spread the weight across a few pages, but intelligently. This is shown in figure 3.5.

Figure 3.5. A user navigation flow to pages with CSS segmented by page template. The browser downloads only the CSS it needs for the current page.

A data-driven method of determining how to segment your site's CSS is to look at its analytics, and look at the path users take through your website. With a tool such as Google Analytics, you can visualize this information and use it to make informed decisions on segmentation.

If you have Google Analytics set up for your website, you can access this information by logging into Google Analytics. Then find the visitor-flow information by navigating to the Behavior section in the left-hand menu, and selecting the Behavior Flow option in the submenu, as illustrated in figure 3.6.

Figure 3.6. The Behavior section of the left-hand menu in Google Analytics. The visitor flows can be seen by clicking the Behavior Flow link in the Behavior submenu.

After clicking this option, the right-hand pane populates. In <u>figure 3.7</u>, you see a simple user flow through the site, with the majority landing on the main page (index.html) and few people proceeding to the subpages.

Figure 3.7. The visitor flow chart in Google Analytics. Starting at the left, you see where users enter the site. In this case, you see that the vast majority of users are entering on the site's main page. Few visitors click through to the subpages.

With this information in hand, it's a simple task to segment your website's CSS for optimal delivery. In this case, it could make sense to pull styles for the second-level pages out of the main style sheet and into a separate file.

How you go about this depends on the site itself and how specific your CSS is. If the templates for most of your pages are all similar and the styling is highly generalized, it makes sense to stick with one style sheet. But if you have many distinct page templates with specific styles, examine your users' behaviors and decide from there.

Say your site has a search results page, and only a small portion of users visit it. Logic dictates that you should separate the CSS specific to that page, place it into a separate file, and include it on the relevant page. Modern tools such as LESS and SASS make modularizing your CSS a trivial task, and the performance benefits are worth considering.

3.1.9. Customize framework downloads

CSS frameworks are a big part of the front-end development sphere, and with good reason. They can be time-saving tools that offer a tremendous service to the developer. If the benefit of using a CSS framework translates into a benefit for the user, they're worth considering.

You *can* have too much of a good thing, though, and it makes sense to prune what you don't need from these libraries. Popular frameworks such as Bootstrap and Foundation allow the developer to customize downloads, as you can see in [figure 3.8](). Don't need print media CSS in Bootstrap? Ditch it. Don't need table styles? Delete them. These features are great, but become performance liabilities when you force users to download code for them and then never use them.

Figure 3.8. The download customization screen on the Twitter Bootstrap website. Bootstrap allows the developer to specify which parts of the framework the user wants in a custom download.

After downloading the customized framework code, don't be afraid to go further and remove anything else you don't need. These frameworks can come at a significant up-front cost to the user. If at the end of a project you find that a lot can be pruned, you're doing your visitors a service by removing unnecessary code from your website.

Now that you know how to segment your CSS and understand the importance of pruning unnecessary code from frameworks, we can talk about the importance of mobile-first responsive web development.
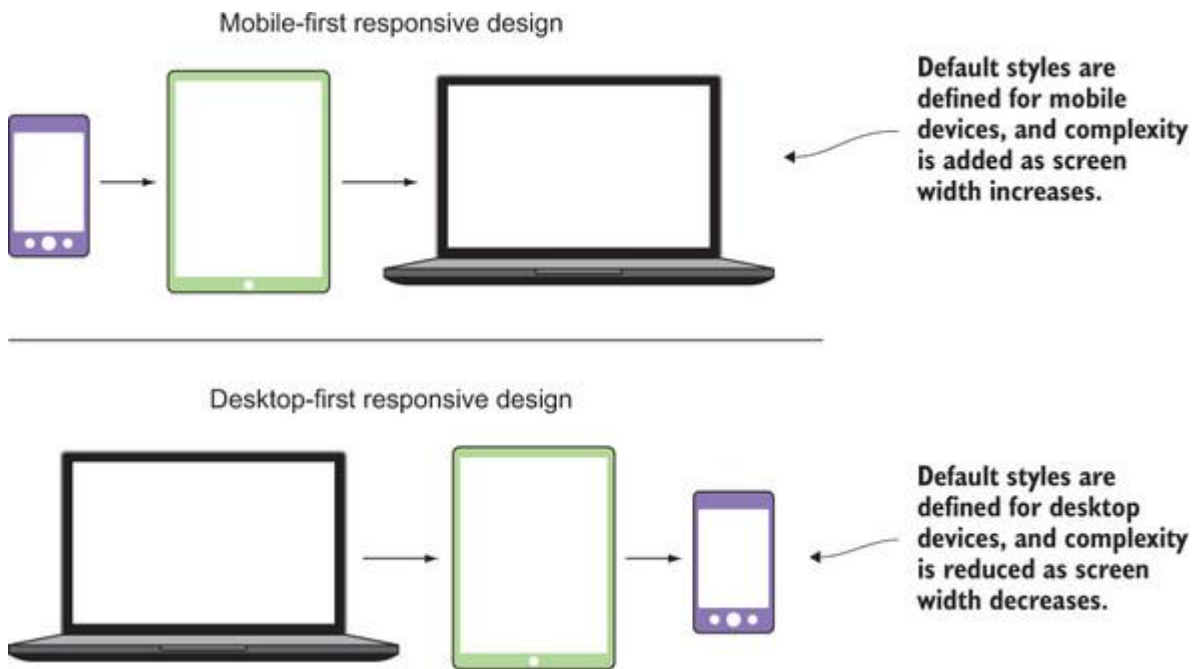
3.2. Mobile-first is user-first

In years past, front-end development has gone from a simple discipline to a more nuanced one. This is due in part to the emergence of the responsive web design principle pioneered by designer Ethan Marcotte. In the past, developers would create separate sites for mobile devices with fewer capabilities than their desktop counterparts. This approach has fallen out of favor, with developers embracing responsive web design instead.

Responsive web design uses one set of markup and modifies its presentation via CSS with respect to the device's display dimensions. These dimensions (usually the width) are examined by using a *media query*, and evaluated against a min-width or max-width value. Because media queries are flexible, two methods of responsive web design arose: desktop-first and mobile-first responsive design. In this section, you'll learn the differences between these two approaches, as well as the importance that Google places on having a mobile-friendly website.

3.2.1. Mobile-first vs. desktop-first

Mobile-first and desktop-first are the two approaches to responsive web design. Figure 3.9 depicts each method.

Figure 3.9. Mobile-first versus desktop-first responsive design flows

Mobile-first responsive design

Default styles are defined for mobile devices, and complexity is added as screen width increases.

Desktop-first responsive design

Default styles are defined for desktop devices, and complexity is reduced as screen width decreases.

Both techniques start with a foundational set of CSS. This CSS isn't contained within any media queries and defines the default appearance of the website. Using the mobile-first method, the default appearance is the mobile version of the site. In desktop-first sites, the default appearance is the desktop version of the site.

Your choice of which technique to use should be made with users in mind, and desktop-first responsive design isn't user-first. Using the mobile-first method, you're building the least complex presentation of a website first, and adding complexity as you scale up. Consider that an increasing number of users are accessing the web by using mobile devices, as shown in figure 3.10.

Figure 3.10. The trend of internet traffic on mobile devices versus laptop devices. Toward the end of 2015, nearly half of all traffic on the internet occurred on mobile devices. This trend is continuing (Data from StatCounter Global Stats).

The advantage of mobile-first CSS is that you're serving CSS intended for the devices that are most likely to consume it. Because mobile devices often have less processing power and memory than desktop devices, the mobile device shouldn't have to apply desktop styles, interpret media queries, and then apply mobile styles.

There's a service to the developer in the long haul, too. The benefit is in the ability to scale up as you go, rather than scaling down. If you start from the point of least complexity, you can optimize much more easily than if you remove pieces as you scale down.

Starting with mobile-first CSS is easy. Most of the time you'll develop for three device types: mobile phones, tablets, and desktops. All of these except for the foundational CSS lie within their own media query (commonly referred to as a *breakpoint*). A *media query* is a particular point in which new styles are applied. This point is typically a change in the screen's width (although height media queries do exist). In the case of mobile-first CSS, the mobile styles are the foundation, whereas the tablet and desktop are the breakpoints where changes in the layout occur. Figure 3.11 shows a set of breakpoints across three device types.

Figure 3.11. The flow of layout complexity across breakpoints on a mobile-first website

You'll notice in <u>figure 3.11</u> that the breakpoints are set using em units rather than px units. An em is a relative unit that's calculated based on the document's default font-size value (typically, 16px). Calculating em values is done with a simple formula:

px / default font size = em

In this case, the tablet breakpoint of 600px is divided by the default document font-size of 16px to arrive to a value of 37.5em. The desktop breakpoint of 1000px is converted using the same formula to a value of 62.5em.

A note on EMs and REMs

The em is a context-specific unit. In a media query, the context is the default font-size value for the HTML document. When ems are used deeper in a document's hierarchy, their context can change. If an element's parent has a font-size value of 12px, the em value is calculated by dividing the original px value by 12. The rem unit is similar to em, except that its context always refers to the document's default font-size value, no matter what its parent element's font-size is. rems have broad support, but aren't universal yet. Consider passing on using rems if your project needs to support legacy browsers.

This listing shows a starting point for a simple mobile-first responsive website.

Listing 3.6. Mobile-first CSS boilerplate

In this boilerplate, CSS resets go first. A popular reset is Eric Meyer's CSS reset, which you can download at http://meyerweb.com/eric/tools/css/reset. These are styles that reset margins, padding, and other properties on elements to normalize inconsistent default styles between browsers. Then come the mobile styles that act as the foundational CSS, the tablet styles, and finally, the desktop styles.

On choosing breakpoints

It's tempting to use common device widths when coding responsive sites. Resist the urge to do this, and instead pick thresholds that are pertinent to your design. Don't be afraid to add minor breakpoints as you code. The prevailing adage is that you resize the browser window until the layout breaks, add another breakpoint, and fix layout problems inside the new breakpoint.

At this point, you'll include your CSS as you normally would inside a <link> tag. To ensure that devices properly display your fancy new responsive CSS, you should add the following <meta> tag inside your <head> element:

<meta name="viewport" content="width=device-width, initial-scale=1">

This <meta> tag tells the browser two things: that the device should render the page at the same width as the device's screen, and that the initial scale of the page should be 100%. You can specify additional behaviors such as disabling zooming, but don't prevent your users from doing this; it could create accessibility issues for users with eyesight problems.

With this boilerplate, you can approach your responsive web design projects with a minimalist mindset. Remember: your users are most important. Developing visually rich and

engaging websites isn't a crime, but developing slow websites is! Starting from a place of minimalism is the best way to ensure that even complex websites load as fast as possible.

3.2.2. Mobilegeddon

In February 2015, Google announced a change in its search results ranking method that took effect two months later. This change gives preference in mobile search results to sites that are considered mobile-friendly.

It makes sense to incentivize developers and content creators to deliver good mobile experiences. For many, Google is the primary gateway through which content is accessed. By emphasizing the importance of how content is delivered on mobile devices, Google is putting the user first. This places responsibility on the developer to provide that experience for everyone.

3.2.3. Using Google's mobile-friendly guidelines

Google's guidelines for mobile-friendly sites are simple. When Google looks at your site, it's looking for two indicators of a good mobile user experience. Let's bring up a version of my personal website at http://jlwagner.net/webopt/ch03-test-site to see what traits are important in mobile-friendly websites

- ***A properly configured viewport—*** As discussed earlier, the $\langle$meta$\rangle$ viewport tag is used by browsers to size content to the device's screen. Figure 3.12 shows my website on a mobile device with and without this tag.

Figure 3.12. A responsive site on a mobile device without the `<meta>` viewport tag (left) and the same site with it (right). Even though the site pictured is a mobile-first responsive site, it won't display in the proper breakpoint without this crucial tag in place, and the user will be forced to zoom out to view the entire site.

Without `<meta>` viewport   With `<meta>` viewport

- *Responsiveness*— A site needs to respond to the size of the viewport as it changes. Users are fine with vertical scrolling, but horizontal scrolling is usually a bad user experience, and Google checks that content fits on a device's screen without horizontal scrolling. Although you should try to make your sites responsive in a mobile-first fashion for performance reasons, any responsive design approach is better than none. If you resize the browser window with my personal website pulled up, you can see that it adapts to the window size.

Google also checks for other things when determining mobile-friendliness, such as legible type sizes and the proximity of tap targets. But by and large, the two preceding criteria are the foundational aspects of any site with a good mobile user experience.

3.2.4. Verifying a site's mobile-friendliness

After its announcement, Google rightly figured that businesses would want to assess the mobile-friendly status of their websites. To help site owners, Google developed the

Mobile-Friendly Test tool, available at https://www.google.com/webmasters/tools/mobile-friendly/. As you can see in figure 3.13, this tool prompts the user to enter a URL to be analyzed. Let's use my personal website (https://jeremywagner.me/) as an example of how to use this tool and interpret its output.

Figure 3.13. The results page of Google's Mobile-Friendly Test tool after examining a website

After the site is analyzed, you'll see that it passes the mobile-friendly test, and a success message is shown. For sites that aren't mobile-friendly, the tool will return a list of reasons that the site failed the test, along with next steps for correcting issues. If your site isn't mobile-friendly, your next steps will be to add the <meta> viewport tag to your site, and make your site responsive for all devices.

In the next section, you'll see how to performance-tune your CSS in order to avoid common problems that can cause delays in page loading and improve page rendering.

3.3. Performance-tuning your CSS

Beyond writing terse and mobile-first responsive CSS, it's vital that you tune your CSS to be as high performing as possible so that users will have a fast and smooth experience. Achieving this starts with applying a set of techniques designed to improve loading and rendering times.

3.3.1. Avoiding the @import declaration

You may have seen use of the @import directive in CSS. This practice should be avoided, because unlike the <link> tag, @import directives in a style sheet aren't processed until the entire style sheet is downloaded. This behavior causes a delay in the total load time for a web page.

3.3.2. @import serializes requests

One of the goals of a performance-oriented website is to parallelize as many HTTP requests as possible. *Parallel requests* are those that are made at, or near, the same time. *Serialized requests* are the opposite, occurring one after another. When used inside an external CSS file, @import serializes requests, as illustrated in figure 3.14.

Figure 3.14. Downloads for two style sheets are serialized one after the other because of an `@import` directive in styles.css that requests fonts.css.

When @import is used to load a CSS file from within an external style sheet, the request for the initial style sheet must be loaded before the browser discovers the @import directive within it. In the case of figure 3.14, styles.css contains this line:

```
@import url("fonts.css");
```

This contributes to a poor performance pattern; requests are serialized one after the other. This increases the overall loading and rendering time for a page. Ideally, you should seek to bundle as many files of the same type as you possibly can. But some CSS includes in your website can come from third parties, making bundling impractical. In these cases, you should rely on the HTML <link> tag instead of @import.

3.3.3. <link> parallelizes requests

The HTML <link> tag is the best native method for loading CSS. Rather than serializing requests as @import does, it loads requests in parallel. After the HTML document is scanned, all <link> tags found in the document are loaded as illustrated in <u>figure 3.15</u>.

Figure 3.15. Two requests for style sheets made by using the <link> tag. The <link> tags are found by the browser after downloading the HTML, and the browser executes these two requests at the same time.

Unlike the @import directive's behavior in CSS files, whereby references to external files can be discovered only after a style sheet is downloaded, the <link> tag references are discovered when the HTML file downloads.

Technically, you *can* use @import inside <style> tags in HTML without any detriment to performance, but mixing this method with the <link> tag or using @import inside a CSS file will cause requests to become serialized. In practice, it's better to stick with the <link>

tag, because its behavior is predictable and relegates the task of importing CSS to the HTML.

The meaning of @import inside LESS/SASS files

In LESS/SASS, @import has a different function. In these languages, @import is read by the compiler and used to bundle LESS/SASS files. This is so that you can take advantage of modularizing your styles during development and bundling when compiling to CSS. The behavior I'm talking about in this section has to do with using @import in regular CSS.

3.3.4. Placing CSS in the <head>

You should place references to your CSS as early in the document as possible, and the earliest place you can load your CSS is in the <head> tag. By doing this, you mitigate two issues: the Flash of Unstyled Content effect, and improving the rendering performance of the page on load.

3.3.5. Preventing the Flash of Unstyled Content

A compelling reason to keep CSS in the <head> of the HTML is that it prevents your users from seeing your site in an unstyled state. This phenomenon is called a *Flash of Unstyled Content*, and it occurs when your users briefly (but noticeably) see your website without any CSS applied to it. Figure 3.16 shows this unsettling effect as the CSS is loaded too late in the document.

Figure 3.16. A rendering timeline in Chrome showing the Flash of Unstyled Content effect at left. The document eventually renders as intended, but with a brief display of the unstyled content. In this case, the effect is due to a `<link>` tag referencing a style sheet being placed at the end of the document.

This occurs because browsers read HTML from top to bottom. As the HTML document is read, the browser finds references to external assets. In the case of CSS, browsers are so fast at rendering that the browser has a chance to render the unstyled page before the external CSS is loaded.

Mitigating this problem is easy: load the style sheet by using a `<link>` tag in your HTML's `<head>` element and you'll avoid the problem entirely.

3.3.6. Increasing rendering speed

Placing your CSS in the `<head>` of your HTML does more than prevent unstyled content from appearing; it also speeds up the rendering of your site on the initial page load. The reason for this is that browsers are fast at rendering pages. If a style sheet is included later in the document, the browser has to do more work than if the style sheet was loaded in the `<head>`, because it has to re-render and repaint the entire DOM.

To test this, I downloaded my personal website onto my machine, pulled it up in Chrome, and used the DSL throttling profile. I then ran 10 tests with the style sheet `<link>` include in the `<head>` of the document, and then another 10 with the same include in the footer. I captured the rendering summary in Chrome's Timeline profiler and averaged the results. Figure 3.17 shows the rendering and painting times for each scenario.

Figure 3.17. Rendering performance at load time of my personal website in Chrome with styles placed in the `<head>` **versus at the end**

The payoff is big, and all because of a small HTML adjustment. If you're working on a site and you see `<link>` tags outside the `<head>`, relocate them to the `<head>` of the document.

3.3.7. Using faster selectors

Earlier in this chapter, you simplified CSS selectors in a client's website. Although this saved space by removing cruft, it can also aid in faster rendering. To see which selector types were the fastest, I compiled a benchmark pitting them against each other. This benchmark can be seen at http://jlwagner.net/webopt/ch03-selectors and is described next.

3.3.8. Constructing and running the benchmark

To determine the browser's rendering capability, you need a sound methodology. I created several HTML files that had the same general markup structure with identical styles. In each file, I styled the document by using different types of CSS selectors. Figure 3.18 shows the general structure of the test markup.

Figure 3.18. The structure of the test HTML document. The test markup is contained within a `div.contentContainer`. Within it are four `<section>` elements arranged in four columns, each containing two `<ul>` elements with 51 `<li>` elements. The block of four `<section>` elements is then repeated approximately 50 times. The total number of elements in each test document is about 21,000.

In figure 3.18, you can see that the test markup is large. Each of these documents contains an inline style sheet that styles the HTML with a variety of selector types. Though the selector types used are different, the end result is an identical appearance for all of the tests.

Viewing the test

If you want to look at the test code, you can find the tests at http://jlwagner.net/webopt/ch03-selectors and run the benchmark in each test page by opening the console and executing the bench() function. You can use the Timeline tool to get data of the test's activity. The data for all of the tests is also available at that page as a Microsoft Excel spreadsheet.

The benchmarking was done with a JavaScript function that stores the innerHTML of the div.contentContainer element to a variable when the document is loaded. Using a chain of

setTimeout calls, the content of that element is removed and reinserted 100 times. This causes a huge number of rendering calculations because of the document being reflowed. This activity was recorded using Chrome's Timeline tool, and the rendering and painting times were recorded.

This procedure was repeated 10 times over eight scenarios, using different selectors. Table 3.1 lists these selectors and how they're used.

Table 3.1. Selector types used in the test, and examples of those selectors in the test

Selector type	Test case example (targeted)
Tag	li
Descendant	section ul li
Class	.listItem
Direct child	section > ul > li
Overqualified	div.contentContainer section.column ul.list li.listItem
Sibling	li + li
Pseudo	li:nth-child(odd), li:nth-child(even)
Attribute	[data-list-item]

With the tests run and the results recorded, let's examine the results.

3.3.9. Examining the benchmark results

With the tests run and the results recorded, the rendering and painting figures were combined into one figure. I had planned to report both, but found that the amount of time Chrome spent repainting ended up being about 200 ms on average for all tests. As a percentage of the total time, this accounted for only 1%–2%. It's the rendering that's the most CPU-intensive, and therefore the most telling. Figure 3.19 shows the results.

Figure 3.19. The performance of the CSS selectors test in Chrome. On the left are the selector types, and on the bottom is the amount of time each selector type took to complete the test in seconds. All values are the sum of rendering and painting processes.

The conclusions you can draw from this are that overall performance for most selector types are similar, but specialized selector types such as the sibling, pseudo class, and attribute selector types are especially expensive.

These tests should be considered a loose guideline for the types of selectors to use, but real-world performance is always preferable. Profile the performance of your website in the developer toolkit of your choice and then make determinations about how to improve.

ID selectors (for example, #mainColumn) *weren't* benchmarked; elements with IDs tend to be few in practice because they're singular and unique items in a document, whereas elements with classes can be used repeatedly.

Continuing on in our efforts to increase rendering performance in CSS, let's look at the performance differences between box model layout and the newer flexbox layout engine.

3.3.10. Using flexbox where possible

For years, laying out content on the web was a combination of floating elements, manipulating their CSS display properties, and using margins and padding. *Flexbox* is a new CSS layout engine available in modern browsers. It simplifies laying out elements on a page. It automatically takes care of spacing, alignment, and justification on both axes. It not only is a more robust way of laying out elements on a page, but also tends to perform better than traditional methods.

3.3.11. Comparing box model and flexbox styles

The way to test flexbox rendering performance is similar to the selector rendering tests in the previous section. You have two test documents, and both are identically styled as a four-column gallery of list items. The first document uses the box model to lay out elements, and the second uses flexbox. The structure of the HTML is a single <ul> element containing a little more than 3,000 <li> elements. Each <li> contains an <img> element and a <p> element. The benchmark is run 10 times, and the figures are averaged.

Viewing the test

If you want to view the test, it's available at http://jlwagner.net/webopt/ch03-box-model-vs-flexbox. As with the selectors test in the previous section, you can run the benchmark by running the bench() function in the console and profile activity to draw your own conclusions.

The CSS is the same, except for the way the list and list item elements are styled. In the box model version, the styling for these elements can be seen here.

Listing 3.7. Box model styling

```
.list{
    margin: 0 auto;
    width: 100%;
```

```
    font-size: 0;
}


.item{
    width: 24.25%;
    list-style: none;
    border: .0625rem solid #000;
    margin: 0 1% 1rem 0;
    display: inline-block;
    vertical-align: top;
}


.item:nth-child(4n+4){
    margin: 0 0 1rem;
}
```

This is typical for a box model–styled list. To space everything out perfectly, margins
are used. A :nth-child selector on every fourth element removes the margin so that the width
and margins of all elements per row add up to 100%. In this listing, these elements are
equivalently styled using flexbox instead.

Listing 3.8. Flexbox styling with flexbox properties in bold

In the test, flexbox is applied to the .list element with a display: flex; rule. This turns
every <li> in the list into a flex item. Using the flex-flow property, you tell the browser
to lay out the items in a row and wrap them onto new lines. The justify-content property
is then used to space the elements out to the edges of the container with the space-between
value. Finally, the flex-basis property replaces the width property from the box model
version, and instructs the browser to render the items at a specific width. You can see
that this code has no :nth-child selector to remove the right margin on every fourth item.
In fact, no items have right margins applied. Flexbox handles all of that for you.

With the environment for the tests defined, let's check out the results!

3.3.12. Examining the benchmark results

Similar to the benchmarks in the CSS selectors test, the rendering and painting figures are combined into one figure. In each test, the painting represents about 60 ms of time, which is a tiny sliver of the overall work Chrome does. The test is run 10 times on each rendering mode, and the results are averaged and graphed in figure 3.20.

Figure 3.20. Benchmark results of box model layout performance versus flexbox layout in Chrome. Lower is better.

The conclusion you can draw is that when it comes to rendering content, flexbox tends to be a better-performing solution. Better yet, it enjoys broad support without vendor-specific prefixes. When used with vendor prefixes, support only increases. If you're not using flexbox on your websites, it's rather trivial to retrofit in most cases.

The next section dives into CSS transitions. We briefly visited this concept in the preceding chapter as part of using Chrome's rendering profiler, but here we'll delve into other concepts that we left untouched.

3.4. Working with CSS transitions

In [chapter 2](#), you used CSS transitions to fix a janky modal window on a client's website. This section covers how to use CSS transitions and the benefits they can provide.

3.4.1. Using CSS transitions

CSS transitions are a solid choice for simple, linear animations on websites with few animation requirements. Here are a few advantages of this native CSS feature:

- *Wide support—* Unlike years past, CSS transitions enjoy broad browser support. All up-to-date browsers support them, and most older browsers such as Internet Explorer (IE) 10 and above do with vendor prefixes.
- *More-efficient CPU usage when reflowing complex DOMs—* In large DOM structures, the CPU was more efficient when using CSS transitions. This is owed to the reduction in thrashing during intensive DOM reflows, and the fact that CSS transitions don't incur scripting overhead. My tests indicated 22% overall better CPU performance.
- *No overhead—* CSS transitions come with no overhead, as they ship with the browser. For websites with simple animation requirements, it makes more sense to use a built-in feature instead of adding the overhead of a JavaScript library to the page.

To get your feet wet, you'll look at a simple use case of this property. Navigate to http://jlwagner.net/webopt/ch03-transition and you'll see a page with a blue box. Hover over this box with your mouse to see the box turn into a circle ([figure 3.21](#)).

Figure 3.21. The `.box` element on the page before and after a transition on its border-radius property

Before transition   After transition

The transition effect is achieved by applying a transition property on the box's border-radius property. Initially, the box has no border-radius applied to it, but when it's hovered over, it's given a border-radius value of 50%. Here is the CSS that drives this effect.

Listing 3.9. Simple CSS hover state transition

```
.box{
    width: 128px;
    height: 128px;
    background: #00a;
    transition: border-radius 2s ease-out;          The transition
}                                                    property

.box:hover{                                          Trigger the
    border-radius: 50%;                              transition.
}
```

Using this CSS, the transition property animates the .box element's border-radius to a value of 50% over a duration of 2 seconds when the user hovers over the box. This changes the element from a square to a circle, and the animation comes to a smooth conclusion with the ease-out timing function.

With a basic use case of this property demonstrated, let's learn more about transition. The property itself is shorthand that sets several CSS properties at once in the following format:

```
transition: transition-property transition-duration transition-timing-function
transition-delay
```

The properties in this shorthand are:

- `transition-property`—The CSS property being animated. This can be any valid property such as `color`, `border-radius`, and so on. Some properties can't be animated, such as the `display` property.
- `transition-duration`—The time the transition takes to complete. Can be expressed in seconds or milliseconds (for example, $2.5s$ or $250ms$).
- `transition-timing-function`—The easing effect used in the transition. This can be expressed using presets such as `linear` or `ease`, segmented using the `steps` function, or can provide more nuanced easing behavior via the `cubic-bezier` function. Omitting this will animate the transition by using the default `ease` preset.
- `transition-delay`—The delay time in seconds or milliseconds before the transition begins. Omit this if no delay is needed.

It's also possible to transition more than one property on an element. If you also wanted to transition the width and the height of the .box element, you could add more to the transition property:

```
.box{
    width: 64px;
    height: 64px;
    transition: width 2s ease-out, height 2s ease-out;
```

With these additional properties, the element will transition the width and height properties of the .box element to whatever new width and height values are added to that element's hover state.

Next, you'll observe the performance of the CSS transition versus jQuery-driven animations.

3.4.2. Observing CSS transition performance

I prepared an animation benchmark that tests the performance of CSS transitions and jQuery-driven animations. I created two identical HTML documents, each with a <ul> element populated with 128 list items. In each test, I animated the list items to grow from a width and height of 5rems to 24rems. In the first test, I used the jQuery animate() method, and in the second test I used a CSS transition. The test is structured this way in order to cause a massive amount of DOM reflow. I tested each of these scenarios by using Google Chrome's Timeline tool five times and recorded the average memory usage, CPU time, and average frame rate of each run. Table 3.2 shows the averaged results.

Table 3.2. Benchmark results of CSS transitions vs. jQuery's animate method in Google Chrome

Transition type	jQuery animate()	CSS transition	Performance gain
Avg. memory usage	5.10 MB	2.32 MB	+54.51%
CPU time	2011.53 ms	1572.02 ms	+22%
Avg. frames per second	44.4	41.1	+8%

The performance advantages in this scenario are clear, but they're not ideal for all situations. Although CSS transitions have their place and increase animation performance at no cost to the user, they work best in simple situations and with simple UI effects such as hovers and off-canvas navigation transitions. Depending on the website you're building, you may require much more complex animation behaviors, and in that case there are better-performing JavaScript solutions that use the requestAnimationFrame() method, which we touch on in chapter 8.

Don't let this dissuade you from using CSS transitions, however, as they're high performing, work well for simple purposes, and have no overhead in the form of extra data your user has to download. If your requirements are simple and you can implement high-performance transitions using CSS rather than adding more weight to the page via a JavaScript animation library, then use CSS transitions.

The next section covers how to inform the browser of the elements you intend to animate with CSS transitions, and how this is beneficial.

3.4.3. Optimizing transitions with the will-change property

When the browser first executes a CSS transition, it must determine which aspects of that element will change. When this happens, the browser has to do some work before the transition executes for the first time. Although not necessarily suboptimal in and of itself, this can have a negative impact on rendering performance.

To get around this, developers discovered a CSS hack that would promote the targeted element to a new stacking context by using the translateZ property. When an element is given this new status in the browser, it's a roundabout sort of way to hint to the browser that this element's rendering should be handled by the GPU in the event that it's animated with CSS.

As with any useful hack, however, translateZ is now targeted for obsolescence with the new will-change property. The problem with the translateZ hack is that it tells the browser, "Something's going to happen here, but I can't tell you what." With the will-change property, you can inform the browser as to which aspects of the element will change.

Consider this property to be complementary to the transition property. You'll recall with transition that you can specify which style properties of the target element will change, such as color, width, or height. The will-change property's syntax works similarly:

will-change: **property, [property]**...

will-change accepts any valid CSS property, or a comma-separated list of properties that can be animated. But be careful: understand that using it improperly can affect the way that resources are allocated on a device. For example, you might be tempted to try to activate this property on all DOM elements to optimize all transitions on a page, like so:

```
*,
*::before,
*::after{
    will-change: all;
}
```

Don't do this. This can have a detrimental effect on page performance, especially in heavily layered and complex pages. If you use this, what you're doing is preparing the browser for the possibility that every element on the page will change. This is just plain bad for performance. The will-change property is a hint, and like all hints, it should be used with discretion.

Another thing to consider when using will-change is that you need to give the property enough time to work. This is a poor usage of the will-change property:

```
#siteHeader a:hover{
    background-color: #0a0;
    will-change: background-color;
}
```

The problem with this is that the browser won't have time to apply the optimizations necessary for any benefit to be realized. A better use of the property in this case is to apply it to a parent element's :hover state so that the browser can anticipate what will happen:

```
#siteHeader:hover a{
    will-change: background-color;
}
```

This gives the browser enough time to prepare for changes to the element, because by the time the user's mouse enters the #siteHeader element and hovers on the link, all of the a elements within it will have been prepared at the time of the #siteHeader element's hover event.

You can also use JavaScript to programmatically add will-change on demand. If a modal window opens and there are background-color transitions on the <button> elements within it, you could use something similar to the following code:

```
document.querySelector("#modal").style.display = "block";
document.querySelector("#modal button").style.willChange = "background-color";
```

After the modal is closed, you can remove the will-change property from the affected elements. Working with this property is finicky, but if you're steadfast, you can optimize transitions on elements in an intelligent fashion without affecting overall page performance. The key thing to remember about this property is that you anticipate *potential* changes on elements rather than assuming they'll happen.

3.5. Summary

You covered a lot of ground in this chapter, and you've learned the following:

- CSS shorthand properties not only are convenient, but also offer us a way to reduce the size of our style sheets by cutting down on excessive and verbose rules.
- Using shallow CSS selectors can also significantly reduce the size of a style sheet, as well as make code more maintainable and modular.
- Applying the DRY principle with the csscss redundancy checker can further winnow bloated CSS files by enabling you to remove superfluous properties.
- Segmenting your CSS based on behavioral data can ensure that users who visit your site for the first time aren't downloading CSS for page templates they may never see.
- Mobile-first responsive web design is important, and starting with minimalism is best for creating high-performance websites.
- A mobile-friendly website is a factor in Google search rankings. By ensuring that your site is mobile-friendly, you can avoid detrimental effects to your site's page rank.
- Avoiding the @import declaration, and placing your CSS in the <head> of the document, confers a positive impact on your site's rendering and load speed.
- Using efficient CSS selectors and the flexbox layout engine can improve the rendering speed of your website.
- CSS transitions for simple linear animations can be high performing, and are offered to the end user with no practical overhead, because they require no external libraries to use.
- Informing the browser of element state changes with the will-change property allows you to selectively boost the animation performance of certain elements, but only if you do so in a predictive and intelligent way. Trying to optimize animation for *all* elements with will-change is not only wasteful, but also potentially dangerous to performance.

The next chapter covers critical CSS, a technique for improving the perceived rendering performance of pages. This technique expedites the rendering of above-the-fold content and uses JavaScript to asynchronously load the rest of the page styles to give users the impression that the page is loading faster.

Chapter 4. Understanding critical CSS

This chapter covers

- Understanding critical CSS and the problem that it solves
- Understanding how critical CSS works
- Using critical CSS in your projects
- Knowing the benefits of critical CSS before and after implementation

With some CSS optimization techniques under your belt, it's a good time to learn an advanced CSS optimization task that speeds up the rendering of a page by prioritizing rendering of above-the-fold content. This technique is called *critical CSS*.

4.1. What does critical CSS solve?

Critical CSS is an optimization task that enables you to rethink how CSS is loaded by the browser by prioritizing the CSS for above-the-fold content ahead of below-the-fold content. When done properly, the user senses a perceived decrease in page-load time owing to faster page rendering. But understanding critical CSS requires an understanding of what *the fold* is.

4.1.1. Understanding the fold

When we talk about the fold, we think of print media. It makes sense to think of the fold this way, because that's where the concept originates. When newspapers are printed, the most important story is printed at the top of the front page. This content strategy ensures that when the papers are folded, bundled, and distributed, the lead story is seen on top.

Designers, marketers, and content strategists have long stressed the importance of placing the most important content *above the fold*, and developers have been tasked with building websites that meet this goal. The difference, though, is that the fold on the printed page is always statically placed. After a design has been printed, the content's job of adapting to the medium is done. The page is folded, and the designer moves on.

The web is a much different medium. The fold changes position depending on the device's resolution, its orientation, and in the case of desktop devices, the size of the browser window. Figure 4.1 illustrates this concept.

Figure 4.1. A depiction of above- and below-the-fold content on an array of devices. The above-the-fold content begins at the top of a website and ends at the bottom of the screen. Anything that's out of the browser's view is below the fold.

When understanding where the *fold* is on the user's screen, you anticipate as best as you can the size of the user's screen. This decision is informed by knowing common device resolutions.

Why is this so important to know? Because critical CSS as a technique is dependent upon knowing what is above and below the fold, and falls into two categories:

- *The critical CSS, or above-the-fold content*— These are styles for content that the user sees immediately and that need to be loaded as fast as possible.
- *The noncritical CSS, or below-the-fold content*— These are styles for content that users don't see until they begin scrolling down the page. This CSS should be loaded as quickly as possible too, but not before the critical CSS.

Now that you know where the fold is and how CSS is categorized in accordance with this concept, you can begin to understand the limitations of conventional CSS delivery. This requires a quick overview of how browser rendering is blocked when style sheets are downloaded and parsed.

4.1.2. Understanding render blocking

Render blocking is any activity that keeps the browser from painting content to the screen on a page's initial load. This has often been considered an unavoidable fact of life on

the web. But as browsers and front-end development technology have matured, this undesired behavior has become more avoidable.

In the case of CSS, render blocking began as a preferred behavior. Without it, the Flash of Unstyled Content occurs, and we see an unstyled page for a brief moment before the CSS is applied. Left to go on for too long, however, render blocking delays the display of a site's content to the screen. Knowing that time is of the essence and that your users won't wait for long, you should seek to minimize render blocking.

Varying degrees of render blocking occur, depending on where CSS is placed in the document, and the method by which it's loaded. Render blocking occurs when external CSS is loaded with the @import directive or the <link> tag. In chapter 3, you discovered how @import can delay rendering, and that the <link> tag is preferable. But the truth is that although <link> is a fine way to load CSS, it too blocks rendering.

To see render blocking in action, open the Coyle Appliance Repair website from chapter 1. While the site loads, capture the activity in Chrome's Timeline profiler (as you learned how to do in chapter 2). After the profiler populates with data, you can go to bottom of the pane, click the Event Log tab, and filter out all but the painting events. If you sort the Start Time column by ascending order, you'll see the Time to First Paint event on the page, as in figure 4.2.

Figure 4.2. Chrome's Timeline profiler when the document's first painting event occurs. The event can be found under the Event Log tab by filtering out all but the painting events.

Waiting about 860 ms is a tad long for the document to begin painting. So how do you fix this? For starters, you can inline the website's CSS directly into index.html, inside the <style> tags. This reduces the time it takes for content to begin rendering, as you can see in figure 4.3.

Figure 4.3. Chrome's Timeline profiler showing an improved paint time after the contents of the site's CSS have been inlined into the HTML

This approach cuts both ways, and the problem is that it works on only single-page websites, where it makes sense to do away with a separate CSS file. On larger and more complex sites, it's only half of a cogent solution.

Inlining and HTTP/2

Although inlining is a suitable practice for HTTP/1 servers and clients, it shouldn't be used with HTTP/2 servers. This functionality can be achieved by using HTTP/2's server push feature while maintaining cachability. To learn more about server push and HTTP/2, check out [chapter 11](#).

4.2. How does critical CSS work?

Critical CSS separates styles into two categories: styles for above-the-fold content and styles for the rest of the page. In this short section, you'll learn how to load the styles for each.

4.2.1. Loading above-the-fold styles

In the preceding section, we discussed the problem of render blocking when using the <link> tag. By inlining CSS into the <style> tag as illustrated in [figure 4.4](#), you can fix this issue.

Figure 4.4. Inline styles loaded for above-the-fold content. The CSS for the above-the-fold content is inlined into the HTML for faster parsing, which translates into a faster Time to First Paint.

If you inlined the CSS from the Coyle Appliance Repair website into the HTML in the preceding section, congratulate yourself, because you've already performed half of what's required for the critical CSS technique to work on more-complex sites.

The reason inlining CSS works so well is that the browser doesn't have to wait as long. When the HTML for a page is loaded, the document is parsed, and URLs to other assets are found. If the styles are externally loaded via a <link> tag, the rendering is blocked while the browser has to wait for the CSS. But when the styles are inlined into the HTML, the user needs to wait only for the HTML to load before the CSS is parsed and the page is rendered.

This is wonderful, but it comes with a detriment: when you load all of the CSS for a site in this way, you lose its portability. You end up duplicating CSS on every page load, which means that you're bloating every subsequent page load with something that you're not caching effectively. The <link> tag takes advantage of caching to optimize return visits.

Critical CSS does account for this to a degree. You bucket *only* the styles for above-the-fold content into the <style> tags and inline that into the HTML, leaving the rest of the styles to be loaded from an external file.

Does this make some of your CSS redundant in subsequent page loads? Sure, but only for a small portion of your site's global above-the-fold content. The decreased Time to First Paint will offset the detriment of that redundancy. Even if you're using CSS frameworks, you can still inline the portions of the framework that are being used for a particular page. When the browser begins painting the page more quickly, the desired effect is achieved, and the user won't notice the performance detriment of a small amount of redundant CSS.

4.2.2. Loading below-the-fold styles

The other half of critical CSS is to load styles for below-the-fold content. These styles are loaded using a <link> tag, but instead of using it in the usual way, you'll use a preload resource hint to load CSS without blocking rendering. You'll also load a script that polyfills preload functionality for browsers that don't support it.

This seems like overkill, but it yields results, especially when combined with inline CSS for above-the-fold content. The browser renders the above-the-fold CSS immediately, while the preload resource hint grabs the styles for the rest of the page in the background.

Want to know more about resource hints?

The preload resource hint is only one of a set of hints that help you fine-tune the loading of assets, not only critical CSS. To learn more about resource hints, see chapter 10.

You may say, "JavaScript blocks rendering, too!" In the case of externally loaded scripts, you're correct. In this case, though, you inline a tiny 1.5 KB script developed by the Filament Group called loadCSS to do the job. With this, you can use preload to load CSS for below-the-fold content by using a single syntax for all browsers, as shown in figure 4.5.

Figure 4.5. The `preload` resource hint loading external CSS for below-the-fold content. This method loads an external style sheet in a way that doesn't block rendering. When the CSS has finished loading, an `onload` event fires and flips the `rel` value of the `<link>` so that the styles render.

The way this method works is ingenious. Instead of using a <link> tag to load the CSS as you normally do, you use a preload hint, like so:

```
<link rel="preload" href="css/styles.min.css" as="style"
onload="this.rel='stylesheet'">
```

This has the effect of loading the CSS without blocking rendering. The onload event handler on the tag fires when the CSS finishes downloading. Once downloaded, the rel attribute's value is flipped from preload to a value of stylesheet. This changes the <link> tag from a resource hint to that of a normal CSS include, which applies the CSS to the below-the-fold content. The JavaScript polyfill is there in case the preload hint is unsupported. Easy as pie!

With the two methods of loading CSS for above- and below-the-fold content clear, you can now set out to implement this technique on a client's recipe website.

4.3. Implementing critical CSS

Now you'll learn how to implement critical CSS on a single page of a mobile-first responsive recipe website. The work you do on the site will take you through the following steps:

1.   Setting up the website to run on your local machine

2.   Identifying the above-the-fold CSS in each of the breakpoints

3.   Separating the above-the-fold CSS from the rest of the CSS, and inlining it into the HTML

4.   Using preload to load the rest of the site's CSS without blocking rendering

4.3.1. Getting the recipe website up and running

To complete the work in this chapter, you'll continue to use git, npm, and node to download and run this website. In addition, you'll use LESS, a popular CSS precompiler.

A note for SASS users

I understand that some developers may prefer SASS over LESS, but for clarity's sake, this website example uses LESS rather than trying to cater to users of both precompilers. If you've used SASS, LESS will feel familiar. Even if you've never used a CSS precompiler, using LESS won't impede your progress. The precompiler used is inconsequential to the takeaways of the work you'll do in this chapter.

Downloading and running the recipe website

A friend of yours is running a recipe website and has asked whether you can make it render faster. The recipe website space is filled with stiff competition, so speed is vital to maintaining the engagement of the site's visitors. Sounds like a job for critical CSS! Start by using git to download and run the site on a local web server with the following terminal commands:

```
git clone https://github.com/webopt/ch4-critical-css.git
cd ch4-critical-css
npm install
node http.js
```

As with prior examples, this installs the Node packages and runs the website on your local machine at http://localhost:8080. After the server is up and running, the site will look like figure 4.6.

Figure 4.6. The recipe website in Chrome. This is the tablet breakpoint at roughly 750 pixels wide.

With the site running, use Chrome's Timeline tool to discover the Time to First Paint for the site. Because you're running from a local web server, you'll want to simulate an internet connection by using the network throttling tool. This will allow you to identify performance improvements in a consistent fashion. Use the Regular 3G throttling profile.

The Time to First Paint doesn't differ based on which breakpoint the page is displayed in. It's a matter of how quickly the browser can fetch and process the CSS and the capabilities of the device. At the end of your efforts, you can expect to see a 30–40% improvement in the time it takes for the browser to begin painting the page.

Next, you'll briefly review the folder structure of the website, so you can be familiar with where all of the site assets live and what they do.

Reviewing the project structure

The structure of the site should be a familiar setup for most developers. The HTML is at the site's root folder and is named index.html. The js folder contains a couple of pertinent JavaScript files, such as scripts.min.js, which has a few simple behaviors for the site, and the minified preload resource hint polyfill in loadcss.min.js and cssrelpreload.min.js. You'll invoke this polyfill in [section 4.3.3](#).

The less folder contains the LESS files for the project. The main.less file generates the styles.min.css file that resides in the css folder. This file is already loaded via a <link> tag in index.html. The critical.less file is used to generate the critical.min.css file that will be inlined into index.html. Each of these files grabs componentized, breakpoint-specific files in the components subfolder. These files are categorized as follows:

- *Global components—* These initially contain all of the styles for the website:
    - global_small.less
    - global_medium.less
    - global_large.less
- *Critical components—* These are initially empty but will be the destination for the critical above-the-fold CSS:
    - critical_small.less
    - critical_medium.less
    - critical_large.less

Now that you know how the website is structured, and what files reside where, you can move on to figuring out where the fold exists for the recipe website.

4.3.2. Identifying and separating above-the-fold CSS

In this section, you'll be tasked with separating the critical CSS for above-the-fold content from the main CSS, and inlining it into index.html. To start, you'll identify where the fold exists.

Identifying the fold

Identifying and bucketing above-the-fold CSS in the document is an exercise of looking at the page in the browser and identifying the elements that are visible on the screen when the page first loads. Anything that's visible is above the fold. Sounds simple, right?

That's correct in theory, but in practice it's a bit more complex. The devices you use aren't always the ones that everyone else uses. If you use a laptop that has a resolution of 1280 x 800 and the window is maximized, your fold is 800 pixels minus the height of the browser-interface elements (toolbars, address bar, and so forth). That doesn't assume that the window is sized differently, or that it's even on a laptop to begin with. One user may be using an iPad, and the next might be browsing on an Android phone.

Thankfully, there's a great website with a sortable list of resolutions for various devices at http://mydevice.io/devices. To see where the fold exists on these devices, you can sort the CSS Height column in descending order, as illustrated in figure 4.7.

Figure 4.7. A chart of common device resolutions on mydevice.io, sorted in descending order by CSS height. The site also offers information for devices other than mobile phones. The physical resolution differs from CSS resolution in that they're both normalized to the same scale for consistency.

Common Smartphones values

name	phys. width	phys. height	CSS width	CSS height
Leap	720	1280	390	695
iPhone 6	750	1334	375	667
Xperia P	540	960	360	640
Xperia S	720	1280	360	640
G4	1440	2560	360	640
G3	1440	2560	360	640

Using this data, you can make determinations about the location of the fold for your site. To assist you in visualizing where this line is on a page, I've made a bookmarklet called VisualFold! You can find this tool at http://jlwagner.net/visualfold. To use it, drag the bookmarklet to your bookmarks, click it, and enter a number where you want a line drawn, as shown in figure 4.8. You can also draw multiple guides at once by entering a comma-separated list of numbers.

Figure 4.8. The VisualFold! bookmarklet in action. The user enters a number in a dialog box (left) indicating the desired location of a guideline to be drawn on the page (right). This assists the user in locating the fold. By resizing the window, the user can see how the content flows with respect to this line.

With this tool, draw guides at positions of 480, 667, 768, 800, 900, 1024, and 1280 pixels. These are common vertical resolutions for popular devices, and most devices are covered anywhere in between. After making these guides, resize the browser window to see where the content falls on each breakpoint.

You'll see that in all breakpoints, the 1280 pixel line falls somewhere within the recipe steps section. In the medium and large breakpoints, this line also falls over the right-hand column content. 1280 pixels seems reasonable, as it covers how the content is displayed on all devices.

Using this approach, you now have a threshold set for your critical CSS, and can begin the process of separating those styles from the main CSS and placing them into your critical CSS.

Identifying the critical components

The next step is to examine the page in each breakpoint and to take inventory of the components that are above the fold. Some of these components exist above the fold in all breakpoints.

Automating the process

The process of determining the critical CSS on a page can be automated by using the Filament Group's CritcalCSS Node program at https://github.com/filament-group/criticalCSS. Using this tool isn't covered in this chapter, so you can learn to identify critical components on your own. Some idiosyncrasies in the program also may break the appearance of your site. If you decide to go this route, be sure to examine the output!

Start by resizing the viewport to the mobile breakpoint. If you haven't placed a guide at 1280 pixels, use VisualFold! now to do so. After the line is in place, inventory the critical components on the page above the guideline, as shown in figure 4.9.

Figure 4.9. The mobile breakpoint of the page with labels of the critical components

Figure 4.9's component inventory is relevant only for this site. When you do this for your own site, your inventory will vary. With this step complete, you can enlarge the window into the larger breakpoint and number the new critical components on the page, as shown in figure 4.10.

Figure 4.10. The large breakpoint with components labeled that were below the fold on the mobile version

Note that when you cross into the large breakpoint, the content splits into two columns. Figure 4.10 highlights five additional critical components that appear below the fold in the mobile breakpoint. Thus, these become critical components on larger screens.

Normally, you would take stock of the largest breakpoint, but in this case, no new critical components appear above the fold in this breakpoint. The header changes, but the rest of the page expands until the page container's max-width of 1024px is satisfied.

Now that you have an inventory of the components that are above the designated fold line, you can move on to separating the critical CSS from the main style sheet.

Separating the critical CSS

With the critical components determined, you can strip out their related styles from the breakpoint-specific includes referenced by main.less, and place them into the includes referenced by critical.less. With this mobile-first website, much of the default styling is defined in the global_small.less file, and trickles up to the medium and large breakpoints. Table 4.1 lists the inventoried components and their related parent container selectors.

Table 4.1. Critical components and their related parent container selectors. These selectors can be used to search for styles for the components in the site's LESS files.

Critical component	Related parent container selector
Site header	header
Content hubs	.destinations
Recipe title	.recipeName
Content container	#content
Recipe image	#masthead
Recipe attributes	.attributes
Social buttons	.actions
Recipe description	.description
Section header	.sectionHeader
Ingredient List	.ingredientList
Banner ad	.ad
Recipe steps	.stepList
Main column	#mainColumn
Right column	aside
Content list/collection list	.contentList
Right column ad	.ad

Before diving into the contents of table 4.1, you'll need to move the reference to reset.less from line 2 of main.less to line 2 of critical.less. reset.less is a global component derived from Eric Meyer's CSS reset (http://meyerweb.com/eric/tools/css/reset) that resets the default styling of many elements for more-consistent rendering across

browsers. Because all elements on the page inherit from this component, these styles are definitely critical.

Once finished, save both files and compile main.less. How you compile depends on your operating system. On UNIX-like systems such as OS X and Linux, run less.sh at the root of the project. On Windows systems, run less.bat instead. You'll run this script every time you make changes to any of the project's .less files.

A smart thing to do before you start moving styles to the critical CSS component files is to comment out line 7 of index.html. This line is the <link> tag reference that brings in the site's styles. This leaves the page unstyled, but it makes visualizing the critical CSS much easier when you begin inlining critical.min.css into the page.

With the CSS reset module moved to critical.less, the next step is to employ and repeat a procedure for each of the critical components and their selectors listed in table 4.1.

Beginning with the header component, open the global_small.less file in the components folder and find the header selector. Cut and paste the header selector into the critical_small.less file in the critical folder, save all files, and rebuild main.less.

After the LESS files rebuild, open critical.min.css in the css folder in your text editor and copy the contents of it into <style> tags in the <head> of index.html. When you do this, the page should look something like figure 4.11.

Figure 4.11. The appearance of the recipe website after you've inlined the header selector CSS into the HTML. It's partially styled, but much is still missing.

Clearly you're still missing many styles. The <header> element on the page appears to be somewhat styled, but it has many child elements, all with their own styles. In order for this critical component to be fully added to the critical CSS, it's necessary to dive into the HTML, take an inventory of which elements are children of the <header> element, and locate the associated CSS selectors. The following is a list of selectors in global_small.less that contain styling for the <header> element's children:

- #logo
- #innerHeader
- nav
- nav:hover .nav
- #navIcon
- #navIcon > div
- .nav
- .show
- .navItem

When you cut and paste the CSS for these elements from global_small.less into critical_small.less and rebuild main.less, you're left with what you see in <u>figure 4.12</u>.

Figure 4.12. The critical CSS after all of the header styles have been inlined into index.html

As you can tell, styles are looking much better for the header. After the CSS for the component has been moved into the critical CSS in the small breakpoint, repeat the same task across all the breakpoints. Progress into the global_medium.less and global_large.less files and move the header-related styles into the critical_medium.less and critical_small.less files, respectively. After you've done this for each of the breakpoints, recompile main.less and re-inline the contents of critical.min.css into index.html.

Repeat these steps until you've worked your way through all the critical components in table 4.1, each time recompiling and re-inlining the critical CSS into index.html. The finished page should appear as it did before you began your efforts, with most of the styling missing for components below the 1280 pixel line.

Want to skip ahead?

If you're stuck, you can skip ahead and use git to see how the critical CSS has been implemented. To do this, open the terminal and enter git checkout -f criticalcss, and the completed work will be downloaded.

With the critical CSS separated from the global CSS, you can go about loading the rest of the page CSS by using the preload resource hint on the <link> tag.

4.3.3. Loading below-the-fold CSS

The last step is to asynchronously load the CSS for the below-the-fold content that's left in styles.min.css. You may be inclined to accomplish this with a standard <link> tag include, but as we discussed in section 4.1.2, <link> tags block rendering of the page. You want to avoid this, so you'll be employing the aforementioned preload resource hint.

Loading CSS asynchronously with the preload resource hint

As discussed earlier, the preload resource hint instructs the browser to begin fetching an asset as soon as possible. In the case of critical CSS, you use this hint to asynchronously load the less important CSS for the below-the-fold content without blocking rendering of the page. To do this for the recipe website, you'll remove any <link> tag in index.html and add the two lines in this listing right after the inlined CSS.

Listing 4.1. Using the preload resource hint to asynchronously load a CSS file

This not only asynchronously loads the CSS, but also covers users who have JavaScript disabled by loading the noncritical CSS via traditional means from within the <noscript> tag, shown in the last line. Those users will be afflicted by the render-blocking behavior of the old CSS-loading behavior, but they won't be left with an unstyled page.

Polyfilling the preload resource hint

Not all browsers support resource hints, and support for the feature is generally limited to Chromium-based browsers such as Chrome and Opera. Therefore, this approach will fail for a large subset of your users. To get around this, you'll use a polyfill available from the Filament Group called loadCSS, available at https://github.com/filamentgroup/loadcss.

I've included the scripts for this polyfill with the GitHub repo for the recipe site in the js folder. These files are cssrelpreload.min.js, which polyfills the preload resource hint functionality, and loadcss.min.js, which provides the asynchronous CSS-loading behavior for when preload resource hint functionality is unavailable.

Using the polyfill is easy. You could include loadcss.min.js and cssrelpreload.min.js by using <script> tags in that order, but that would block rendering, which is what you're trying to avoid. Instead, you should inline these scripts in the order indicated within a single <script> tag, and place the inlined scripts *after* the code indicated in [listing 4.1](#). When you do this, you can test the loading behavior in browsers that don't support the preload behavior, such as Safari. You should find that the CSS for below-the-fold content should render (and before, without the polyfill scripts, it wouldn't).

With the critical CSS method fully implemented into the recipe website, you can go on to analyze the benefits of your work.

4.4. Weighing the benefits

Before we began, I claimed that you would see a 30–40% decrease in Time to First Paint. Using Chrome, I assessed this performance indicator across several throttling profiles. You can see the results in [figure 4.13](#).

Figure 4.13. Time to first paint performance in Google Chrome before and after implementing critical CSS

Time to first paint before and after implementing critical CSS
(lower is better)

As you can see, as connection speed increases and latency decreases, the returns diminish. This is true of any kind of front-end optimization you make. Not every connection is created equal, and it's particularly important to optimize for mobile users who are the most likely to be on low-quality internet connections.

For mobile devices accessing the recipe website from a shared host, the benefits were a bit more modest, showing roughly a 20% improvement, as you can see in figure 4.14.

Figure 4.14. Time to First Paint in Mobile Safari on an iPhone 6S over a remote shared host before and after prioritizing critical CSS

Mobile Safari time to first paint before and after
implementing critical CSS
(lower is better)

One statistic to remember is that 0.1 seconds is the limit for a user to feel like an interface is reacting instantaneously. Decreasing your Time to First Paint *in addition* to other techniques you've already learned (and more that you'll learn later) will make that user feel like the site is responding quickly. If that's important to you, it's worth considering applying critical CSS to your website. The sooner users *feel* that your site can be interacted with, the more likely they'll stick around to see what you have to offer.

4.5. Making maintainability easier

The biggest obstacle to maintainability with critical CSS is inlining. It's not efficient to copy and paste critical CSS into the <head> of the document every time it changes. It's also a pain to inline the polyfill scripts. If something changes, you'd have to re-inline the changed code. Ideally, you want the maintainability of separate files but to have them automatically inlined so you can reap the rendering benefits of resource inlining.

One way to cut down on the mundane work of copying and pasting code is to use a server-side language to inline files into your HTML. PHP's file_get_contents function is perfect for this task. This function reads a file from the disk and allows you to inline it in a document. This is how to inline critical CSS in the <head> of a document by using this function.

Listing 4.2. Using PHP to inline a style sheet

```
                                              The critical CSS is inlined on the server
                                              side by using file_get_contents.
<style>
    <?php echo(file_get_contents("./css/critical.min.css")); ?>       <—
</style>
<link rel="preload" href="css/styles.min.css" as="style"
      onload="this.rel='stylesheet'">
<noscript><link rel="stylesheet" href="css/styles.min.css"></noscript>
<script>
    <?php
    echo(file_get_contents("./js/loadcss.min.js"));
    echo(file_get_contents("./js/cssrelpreload.js"));      The preload resource hint
    ?>                                                     polyfills are also inlined
</script>                                                  on the server side with
                                                           file_get_contents.
```

This approach allows you the modularity of separate files, while also enjoying the benefits that inlining provides. PHP doesn't have the monopoly on this capability, either. Any widely used server-side technology will have an equivalent method you can use to achieve the same result.

4.6. Considerations for multipage websites

This chapter walked you through implementing critical CSS on one page, but what about multipage sites? The approach is similar, but as shown in <u>figure 4.15</u>, the focus is on modularity.

Figure 4.15. A modularized approach to critical CSS. Template A and Template B have their own critical CSS that's inlined only for those pages, but both inline globally common critical styles.

In figure 4.15, you can see two page templates: Template A and Template B. Both are unique in that they have different CSS for their own above-the-fold content. For greater efficiency, it makes sense that the critical CSS for each page template is split into separate files. Those files are then inlined for only the pages that need them.

But there are critical styles for components that exist on every page on a website, such as the header, navigation, headline styles, and so forth. It makes sense to bucket those styles separately and inline them on all pages across the site.

When implementing critical CSS on a large website with several templates, the idea is to avoid combining the critical CSS for every unique page template on *every* page. Bucket those styles accordingly, and inline only the CSS that you need for a particular page.

The good news is that this doesn't change how you implement critical CSS. The process is the same; you're repeating it for each page template. More importantly, research your site analytics and consider using this method on your high-value pages, where the dividends will

be greater. Critical CSS requires a good deal of effort, and the benefits are worth the trouble, but prioritize it for your most important content pages above all else.

4.7. Summary

In this chapter, you learned the importance of critical CSS. As a part of this broad, overarching concept, you learned these smaller concepts and methods:

- The fold is a flexible concept. It refers to the cutoff point at which content is not visible on the screen. It also changes based on the device viewing the page.
- `<link>` tags block the rendering of a web page, which creates delays in document painting. Critical CSS allows you to eliminate this behavior.
- Critical CSS works by prioritizing the loading of CSS for above-the-fold content over CSS for below-the-fold content. The critical CSS is inlined into the site's HTML, and the noncritical styles are loaded in deferred fashion. When you defer the loading of the noncritical styles, you sidestep the effects of render blocking on the page.
- Implementing critical CSS not only gives the user the *impression* that the page is loading faster, but also is a measurable phenomenon. A page's Time to First Paint value decreases when critical CSS is used, and you can compare the effect of critical CSS on this metric by using Chrome's Timeline tool.

Now that you understand how to implement critical CSS and have witnessed the benefits it provides to the user, you can move onto the next chapter. Next, you'll learn the importance of serving images according to the capabilities of the devices that are requesting them.

Chapter 5. Making images responsive

This chapter covers

- Using CSS media queries to deliver the right background images for a user's device
- Delivering responsive images in HTML using `srcset` and the `<picture>` element
- Using Picturefill to polyfill `srcset` and `<picture>` in certain browsers
- Using SVG images in CSS and HTML

Now that you've learned useful CSS optimization techniques, you can dive into the importance of managing the images on your website.

Images often compose the largest portion of a website's total payload, and that trend shows no sign of changing. Though internet connection speeds are continually improving, many devices are shipping with high DPI displays. In order for images to display optimally on these devices, higher-resolution imagery is required. Because you still need to support devices with less-capable screens, however, you still require lower-resolution images and must pay attention to the way images are delivered to various devices.

When you think about image delivery, the intent isn't only to deliver the best visual experience possible, but also to deliver images that are appropriate for a device's capabilities. Knowing how to properly deliver images ensures that devices with less capability are never burdened with more than what they need, while ensuring that the most capable devices are receiving the best possible experience. Maintaining this balance ensures an appropriate mix of visual appeal and performance.

5.1. Why think about image delivery?

One component of web performance as it pertains to imagery means delivering the right image sizes and types to the devices that can best use them. This section introduces the importance of properly delivering images—in CSS as well as in HTML. When we talk about image delivery, what we're really talking about is serving images *responsively.*

Responsive images matter if you care about the performance of your website. It's important for your CSS to be responsive so that your site is viewable on as many devices as possible. But it's also important for your images to be responsive for these two reasons: scaling and file size.

When images are served with responsiveness in mind, users are getting the best experience that their devices are capable of. For instance, one large image can scale down well for all devices, but it isn't the best choice even if it looks great for all devices. The device has to take an image that's grossly oversized and rescale it to fit the screen. This image

will also have a larger file size, thus taking more time to download. This impairs performance.

Instead, it makes more sense to serve images to best fit the device's needs. This entails maintaining multiple sets of images, but the effort is worthwhile because processing and downloading times are minimized. Figure 5.1 illustrates inefficient versus efficient methods of image scaling.

Figure 5.1. Two examples of scaling an image to a mobile phone. At the top, a 170-KB image with a width of 1440 pixels is scaled down to the width of the phone's high DPI display. At the bottom, a 41-KB image with a width of 750 pixels is delivered to the screen without having to be scaled; this process is more efficient.

1440 x 900
170 kilobytes

750 x 1334
Retina screen

Inefficient:
image scaling
required

750 x 469
41 kilobytes

750 x 1334
Retina screen

Efficient:
no scaling
required

The performance impacts of responsive imagery must be examined as well. Google Chrome's Timeline tool comes in handy once again in measuring the rendering and painting performance of inefficient versus efficient image scaling. This test was run five times for each scenario and captured two aspects: rendering and painting time. Figure 5.2 shows the results.

Figure 5.2. A comparison of rendering and painting times for a single image in Chrome. For the scaling scenario, the source image of 1440 x 900 is scaled to fit a container 375 pixels wide. In the no-scaling scenario, an image resized to fit the container is used and triggers no scaling. Rendering and painting times are faster with no scaling.

Rendering and painting time:
image scaling vs. no scaling
(lower is better)

In this scenario, I observed a 15% improvement in rendering time and a 36% improvement in painting time. Though this represents an improvement of only a millisecond, it's important to remember that this measurement was taken for a single image. Implementing responsive imagery across an entire website can help make pages feel more responsive to user input by reducing processing time.

Is it possible to deliver perfectly scaled images for all devices? Anything is possible, but such a goal *isn't* practical. The best approach is to define an array of image widths that covers the entire spectrum of your needs, with the understanding that some overlap will exist between images in the array to cover the needs of different devices and display densities. Some scaling is always going to happen. You want to minimize how much of it occurs.

How you use responsive images depends on *where* they're being used. As you well know, images are most often referenced in CSS and HTML. As you work through the rest of this chapter, you'll learn about the various types of images, their best uses, and ways to use them in a responsive fashion in both CSS and HTML.

5.2. Understanding image types and their applications

Using images on the internet was once simple: a few formats existed, but all were bitmap (also known as *raster*) images. Some images were better suited than others for certain tasks. These rules hold true today, but the landscape has changed, and the playing field is larger than it used to be. In this section, you'll learn about the two major image types: raster and SVG.

5.2.1. Working with raster images

As you likely know, the most common type of image used on the web is a *raster*. These are sometimes called *bitmap images*. Typical examples are JPEG, PNG, and GIF images. These comprise pixels aligned on a two-dimensional grid. Figure 5.3 shows an example of a YouTube favicon enlarged from 16 x 16 pixels to 512 x 512 pixels to illustrate the concept.

Figure 5.3. A 16 x 16 raster image of a YouTube favicon. On the left is the native size of the image, and on the right is the enlarged version. Each pixel is part of a two-dimensional grid.

Raster images are used to depict many things on the web: logos, icons, photos, and so on. In HTML, they're displayed using <img> tags. In CSS, they're often used in the background property, but also find use in other lesser-used properties such as list-style-image.

Raster images come in a few formats, and each is best suited to a particular kind of content. In this section, you'll categorize these images by the way they're compressed. Recall that in chapter 1 you used server compression to achieve smaller file sizes for style sheets, scripts, and HTML assets. In this section, you'll learn about compression as it pertains to raster images, which fit into two categories: lossy and lossless.

Lossy images

Lossy images use compression algorithms that discard data from an uncompressed image source. The idea behind these image types is that some level of quality loss is acceptable in exchange for smaller file sizes.

A good example of lossy images happens right in your digital camera. When you download photos from a digital camera's storage card, you're typically downloading them as JPEGs. When the camera takes a photo, it stores an uncompressed version of the photograph in memory and uses lossless compression to convert the uncompressed source to a JPEG image.

JPEG images are nearly everywhere on the web. A prolific example of the JPEG format put to use is on the popular photo-sharing website Flickr, shown in figure 5.4.

Figure 5.4. JPEG images in use on the popular photo cataloging and sharing site Flickr. Photographic content is best suited to the JPEG format.

A drawback of this format is that extreme compression can be noticeable. These file types are also susceptible to generational loss if they're not saved from an uncompressed source such as a PSD file. Generational loss occurs when an already-compressed file is recompressed, resulting in further visual degradation. In practice, however, the degradation shouldn't be noticeable if compression is used with care, and these image types are saved from an uncompressed source.

Figure 5.5 shows two versions of a photograph. The left image is the uncompressed source, and the right image is a copy after a reasonable amount of JPEG compression has been applied to it.

Figure 5.5. A comparison of the same image in uncompressed (TIFF) and compressed (JPEG) formats with their respective file sizes. The JPEG version has some subtle degradation at a quality setting of 30, but is acceptable for this scenario.

Uncompressed
(269 KB)

JPEG
(12 KB)

Photo credit: NASA Jet Propulsion Laboratory.

The visual quality shows subtle signs of degradation from the uncompressed source to the compressed JPEG. Considering that the JPEG is about 96% smaller than its uncompressed TIFF version, this loss in visual quality is an acceptable trade-off. The output quality of the JPEG algorithm is expressed on a scale of 1 to 100, where 1 is the lowest and 100 is the highest.

JPEG isn't the sole lossy image format used on the web. Other such formats exist, such as Google's new WebP image format (covered in chapter 6). Now let's look at lossless image types.

Lossless images

On the other end of the spectrum are *lossless images*. These image types use compression algorithms that don't discard data from the original image source. A good example of a lossless image in action is the Facebook logo on the desktop version of its site, as shown in figure 5.6.

Figure 5.6. The Facebook logo is a PNG image, which is a lossless image format. PNG images are well suited for lossless formats.

Unlike lossy image formats, lossless formats are perfect when image quality is important. This makes lossless formats a great candidate for content such as icons. Lossless image types usually fall into two categories:

- *8-bit (256-color) images*— These are formats such as the GIF and 8-bit PNG formats and support only 256 colors and 1-bit transparency. Despite their color limitations, they're great candidates for icons and pixel art images that don't require a lot of colors or sophisticated transparency. The 8-bit PNG format tends to be more efficient than GIF images. Unlike GIFs, however, PNGs have no support for animation.
- *Full-color images*— The only formats that support more than 256 colors are the full-color PNG format and the lossless version of the WebP format. Both support full alpha transparency and up to 16.7 million colors. The full-color PNG format enjoys wider support than WebP. These image types are well suited for icons and photos, but their lossless nature means that photographic content is usually best left to the JPEG format unless transparency support is a must.

Results of lossless image formats depend on the subject of the image. Figure 5.7 provides a generalized comparison of lossless image compression methods.

Figure 5.7. A comparison of lossless image-compression methods. The differences between the uncompressed and full-color PNG and WebP versions are imperceptible, whereas the 8-bit lossless image is throttled down to 256 colors.

| Uncompressed | 8-bit PNG/GIF | Full-color PNG | Lossless WebP |
| (269 KB) | (26 KB) | (51 KB) | (140 KB) |

Photo credit: NASA Jet Propulsion Laboratory.

The various lossless formats have their advantages in certain types of content, and are overall well-suited for line art, iconography, and photography. The trick to finding the right fit requires experimentation, but the basic rules of using these formats are generally simple: Simple images with few colors should use 8-bit lossless formats. Images not suitable for lossy formats and/or in need of full transparency should use the full-color PNG format.

The next section covers an entirely different class of image aside from raster images: scalable vector graphics.

5.2.2. Working with SVG images

Another type of image format used on the web is *Scalable Vector Graphics*, commonly referred to as SVG. These images use a vector artwork format. They're different from raster images in that they can be scaled to any size, because they're composed of mathematically calculated shapes and sizes. Figure 5.8 demonstrates this effect.

Figure 5.8. A cartoon vector image at different sizes. Notice that the larger version doesn't lose any visual quality as it scales up. This is the primary advantage of vector images over raster images.

Vector images scale so well because of the way they're rendered. Although all device screens are pixel-driven, and thus all display output eventually ends up being represented by pixels, vector images go through a different process than raster images when they're displayed on a screen. They're parsed, their mathematical properties are evaluated, and then they're mapped to the pixel-based display through a process called *rasterization*. This occurs every time the image is scaled, ensuring the best possible visual integrity every time.

If you're familiar with creating vector artwork, you're familiar with programs such as Adobe Illustrator. Although file formats native to these programs are expressed in binary, the SVG format is expressed as XML, which is a text format. The SVG media type image/svg+xml reflects its roots in XML. This characteristic allows you to edit SVG files in a text editor, place SVG files inline in HTML, and even to use CSS and media queries in SVG files.

Although SVG has been a W3C standard since 1999, its use in websites has been only relatively recent. The fact that SVG works so flawlessly across different device resolutions and display densities makes it a popular image format.

SVG isn't a silver bullet, however, and it's limited in its applications. SVG images *aren't* suitable for photographs, and are most effective when used to depict logos, iconography, or line art. Images with solid colors and geometric shapes are well served by this flexible image format.

5.2.3. Knowing what image formats to use

With all of the image types available, knowing which type makes the most sense for a certain kind of content can be daunting. Although vector images are a standalone category with SVG as the prevailing format, raster images are classified by the two categories of compression (lossy and lossless) and within them are a slew of formats. Table 5.1 should give you some ideas on the kinds of content that fit best with each image type

In the next section, you'll optimize image delivery in CSS for a website's masthead image.

Table 5.1. You can choose an Image format based on the type of content for your site.

Each image for mat varies in color restrictions, image type, and compression category.

(*Full* color indicates a range of 16.7 million or more colors, 24/32-bit.)

Image format	Colors	Image type	Compression	Best fit
PNG	Full	Raster	Lossless	Content that may or may not require a full range of colors. Quality loss is unacceptable, and/or the content requires full transparency. Accommodates any image format, but may not compress as well as JPEG for photographs.
PNG (8-bit)	256	Raster	Lossless	Content that doesn't require a full range of colors but might require single-bit transparency—such as icons and pixel art.

Image format	Colors	Image type	Compression	Best fit
GIF	256	Raster	Lossless	Same as 8-bit PNG with somewhat lower-compression performance, but supports animation.
JPEG	Full	Raster	Lossy	Content that requires a full range of colors and for which quality loss and lack of transparency are acceptable—such as photographs.
SVG	Full	Vector	Uncompressed	Content that may or may not require a full range of colors, and for which quality loss is unacceptable when scaled. Best for line art, diagrams, and other generally nonphotographic content. Requires the least amount of development effort for optimal display on all devices.
WebP (Lossy)	Full	Raster	Lossy	Same as JPEG, but also supports full transparency, with the potential for better compression performance.
WebP (Lossless)	Full	Raster	Lossless	Same as full-color PNG, with the potential for better compression performance.

5.3. Image delivery in CSS

Image delivery in CSS is a great place to start, because it involves CSS properties and features that you're familiar with if you've ever developed responsive websites. The main CSS feature you'll use to properly deliver images is the media query.

This time around, you'll optimize the delivery of a masthead image for Legendary Tones, a website introduced in [chapter 1](#) that publishes articles of interest to guitarists. The images in this site are poorly managed, resulting in low quality on larger screens. You want to deliver higher-quality images to these users, but you want to do so in a way that doesn't overburden them with images that are too large, no matter the device they could using. You also want to ensure that users with high DPI displays are getting the best possible experience on their screens.

You'll start by downloading this website and running it locally. To do this, you need to perform these commands in a folder of your choosing:

```
git clone https://github.com/webopt/ch5-responsive-images.git
cd ch5-responsive-images
npm install
node http.js
```

After the website has been downloaded, you can pull it up on your local machine at http://localhost:8080. It should look something like [figure 5.9](#).

Figure 5.9. The Legendary Tones website as it appears in the browser

When the website is running on your local machine, you can segment images and form a plan for targeting displays by device width, device DPI, as well as how to use SVGs in CSS.

5.3.1. Targeting displays in CSS by using media queries

The goal of this section is for you to improve the visual quality of the Legendary Tones website masthead image by using a set of background images supplied in the img folder to create an experience that's optimal for *all* devices.

When implementing responsive images using CSS, the best tool for the job is the media query. With media queries, you can decide at which screen width you should change a background-image rule for a particular selector.

On the Legendary Tones website, only one selector has a background-image property set, and that's the #masthead selector. It sets the styling for the top of the page, which has a large background image, logo, and site tagline. The CSS for this selector is shown here.

Listing 5.1. The #masthead styling for the Legendary Tones website

```
#masthead{
    padding: .5rem 0 0;
    height: 10rem;
    background-size: cover;
    background-image: url("../img/masthead-xxxsmall.jpg");
    background-position: 50% 50%;
    position: relative;
}
```

Ensures the background image covers the entirety of the container no matter its size.

Default mobile-first background image for the masthead

The #masthead element spans the width of the browser window and has a background-image value of masthead-xxxsmall.jpg. If you load this website on large screens, you'll immediately notice how poor the visual quality of the background image is. This is because the default styling is mobile-first, so the smallest image meant for mobile devices is served.

Let's take stock of the images in the img folder which contains an array of masthead background images that you'll plug into media query breakpoints for the #masthead selector. Table 5.2 lists these images and the media queries they'll be targeted for.

Table 5.2. Images, their resolutions, and their target media query breakpoints in the website's CSS

Image name	Image resolution	Media query
masthead-xxxsmall.jpg	320 x 135	None (default image)
masthead-xxsmall.jpg	640 x 269	(min-width: 30em)
masthead-xsmall.jpg	768 x 323	(min-width: 44em)
masthead-small.jpg	1024 x 430	(min-width: 56em)
masthead-medium.jpg	1440 x 604	(min-width: 77em)
masthead-large.jpg	1920 x 805	(min-width: 105em)
masthead-xlarge.jpg	2560 x 1073	(min-width: 140em)
masthead-xxlarge.jpg	3840 x 1609	Not used

As you can see, each image falls into a specific device-resolution range that allows it to be scaled without being stretched to the point of being pixelated. The first image, masthead-xxxsmall.jpg, starts at a width of 320 pixels and stretches up to 479 pixels. At the 480-pixel breakpoint, the 640-pixel version of the image kicks in and scales larger and larger to a width of 703 pixels. At 704 pixels, a new breakpoint kicks in and a higher-resolution image is substituted. This continues until the maximum resolution is hit. Note that the masthead-xxlarge.jpg file isn't used. You'll use it later on this section when you target high DPI screens. For now, you'll ignore it.

Open the styles.css file from the css folder in your text editor. You'll notice that it's a mobile-first example, and that a slew of media queries exist at the bottom of the file. The styles in these media queries serve to change the size of the site logo, the site tagline copy, and the height of the #masthead container. The first breakpoint set on line 183 at 480px (or 30em, because you're using ems instead of pixels) looks like the following listing.

Listing 5.2. Media query breakpoint

```
@media screen and (min-width: 30em){ /* 480px/16px */
    #masthead{
            height: 12rem;
    }

    #logo{
            width: 70%;
    }

    #tagline{
            font-size: 1.25rem;
    }
}
```

Media query for a screen width of 480 pixels

The element you'll be adding a background image to in this and further breakpoints

What you want to do with this code is modify the #masthead selector's content to provide a higher-resolution background image than masthead-xxxsmall.jpg when this breakpoint kicks in. So go ahead and change the content of the #masthead selector to the following:

```
#masthead{
    height: 12rem;
    background-image: url("../img/masthead-xxsmall.jpg");
}
```

After you add the background-image property for the new image, save the document and reload the page. Then observe the improvement in image quality, as illustrated in .

Figure 5.10. The masthead background image at the 480-pixel (30em) breakpoint before (left) and after (right) adding the new background image. Note the improved visual quality in the after image.

Before adding background image   After adding background image

All that's left to do from here is to repeat this process for each breakpoint listed in table 5.2. When you're finished, you should have a masthead with a background image that transitions well from a mobile phone resolution all the way to a large desktop device resolution.

Want to skip ahead?

Having a bit of trouble? Or want to skip ahead and see how everything works? You can do this by using git to switch to the finished state of the project, where you can view the code. Enter git checkout responsive-images -f at the command line to skip ahead. Be warned that any changes you have in your local copy may be lost!

In the next section, you'll learn how to target high DPI displays and use media queries to deliver higher-resolution images to those more capable devices.

5.3.2. Targeting high DPI displays with media queries

One aspect of implementing responsive images that you must keep in mind are *high DPI displays*, such as 4K and 5K ultra HD displays. One well-known example of this technology is Apple's Retina Display, but it's certainly not limited to Apple devices. Most devices now ship with screens that can be considered high DPI. A comparison of standard versus high DPI displays can be seen from our masthead in figure 5.11.

Figure 5.11. An enlarged visual representation of graphics on standard displays versus high DPI displays

High DPI displays deliver enhanced visual experiences but present a new challenge for the developer: delivering images efficiently for them. Figure 5.12 provides a comparison of properly delivering images to these displays.

Figure 5.12. A comparison of two versions of a background image on two display types. On the left, a background image intended for use on standard displays appears on a high DPI display. On the right, the proper resolution image is used for the high DPI display, creating a better visual experience.

In a continuation of the efforts you began earlier in this section, you'll implement background images for the #masthead element, but this time for high DPI screens. All this requires is to target not only device widths as you did earlier, but also their pixel density with a combination of media queries. Here's an example of a basic high DPI screen media query:

```
@media screen (-webkit-min-device-pixel-ratio: 2),
              (min-resolution: 192dpi){
    /* Put High DPI Styles Here */
}
```

Here you see two media queries in action: the vendor-prefixed -webkit-min-device-pixel-ratio media query is the WebKit implementation for high DPI display support for older browsers, whereas the min-resolution media query is used for modern

browser support (although newer browsers tend to recognize the vendor-prefixed media query as well).

The -webkit-min-device-pixel-ratio media query checks for a simple ratio of pixel density in which 1 equals 96 pixels. In this case, you're making sure that the display has at least 192 DPI of pixel density before you have it download the higher-resolution images. The min-resolution media query takes a more straightforward value of 192dpi.

At this point, you need to assign the proper background image for each new breakpoint. This means taking the background images in table 5.2 and reworking them so that you're adjusting the background images on the #masthead element for high DPI screens. Table 5.3 is the result of this effort.

Table 5.3. Background images for the `#masthead` selector in the CSS, their resolution, and the high DPI screen media queries

Image name	Image resolution	Standard DPI media query	High DPI media query
masthead-xxxsmall.jpg	320 x 135	None (default)	Not used
masthead-xxsmall.jpg	640 x 269	(min-width: 30em)	(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi)
masthead-xsmall.jpg	768 x 323	(min-width: 44em)	(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 30em)
masthead-small.jpg	1024 x 430	(min-width: 56em)	(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 44em)
masthead-medium.jpg	1440 x 604	(min-width: 77em)	@media screen(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 56em)
masthead-large.jpg	1920 x 805	(min-width: 105em)	@media screen (-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 77em)
masthead-xlarge.jpg	2560 x 1073	(min-width: 140em)	@media screen (-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 105em)
masthead-xxlarge.jpg	3840 x 1609	Not Used	(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi), and (min-width: 140em)

If you compare [table 5.3](#) to [table 5.2](#), it looks similar at first, except each image has been shifted up so that higher-resolution images are used for smaller screen widths. You might remember that masthead-xxxsmall.jpg was used as the default background image for the masthead. For higher-resolution screens, this has been bumped up to the next largest image, which is masthead-xxsmall.jpg. For standard DPI displays, the image masthead-xxlarge.jpg was left unused. This image has now been put into play at the largest breakpoint to cover high DPI displays on large-screen devices.

To begin using the higher-resolution background images, find the start of the high DPI media queries in styles.css. This starts on line 280. You'll see a media query that looks like this:

```
@media screen (-webkit-min-device-pixel-ratio: 2),
              (min-resolution: 192dpi){ /* High DPI Default */
    #masthead{
    }
}
```

This media query is similar to your mobile-first styles, except that it defines the default styles for the page on high DPI screens. Within this media query, you'll need to change the content of the #masthead selector to include a new background-image property:

```
#masthead{
    background-image: url("../img/masthead-xxsmall.jpg");
}
```

When you make this change, the increase in quality will be noticeable, as it is in [figure 5.12](#). After you've made this adjustment, you need only to work through the list of images in [table 5.3](#) and apply them to their respective media queries.

Want to skip ahead?

If you're having some trouble or want to skip ahead to see how the code works, you can do so by going to your terminal or command line and using git. Go to the folder that the website is running in and enter the command git checkout hi-dpi-images-f. Be wary that you could lose any changes you have in your local copy!

In the next section, you'll learn about using SVG images in your CSS, and the advantage this vector image format has over raster images when it comes to efficiently delivering high-resolution images to all types of displays.

5.3.3. Using SVG background images in CSS

Sometimes it may be preferable to use SVGs in background images, depending on the content of the image. As I said earlier in this chapter, if you have an image that contains a lot of line art, SVGs are perfect. When using these image types in CSS, no media queries are required for the image to display flawlessly across all resolutions and display densities.

Find the SVG file in the img folder named masthead.svg. Open styles.css in your text editor and go to line 66 to the #masthead selector and change the background-image property in this selector to the following:

```
background-image: url("../img/masthead.svg");
```

Then remove all the media queries in the document, starting from line 180, so the background image overrides in those media queries don't override the SVG background you've set. After you make these changes, save and reload the page to see the new background image.

When the new SVG goes into effect, resize the window and observe how the image scales flawlessly at all resolutions. This is the primary advantage of SVG files at work. No media queries required, the image scales properly at all device widths and screen types, high DPI or otherwise.

Again, it's important to remember that this image type isn't suitable for all kinds of content. You'll still be better served by JPEGs and full-color PNG files for photographs. SVGs are best used for content such as logos, line art, and patterns. If in doubt, try SVG to see whether it works. Be sure to take note of the file size and see whether it's efficient for the intended device. If a user can be better served by a smaller raster image, consider switching. Most of the time, the SVG will be the most efficient if the content of the image is well suited to the format.

In the next section, you'll learn about various techniques for implementing responsive images in HTML.

5.4. Image delivery in HTML

Although CSS is useful for managing images in a responsive manner, it doesn't solve the problem of making images responsive when they're referenced from HTML. Since the birth of HTML, the <img> tag has been the vehicle for images on the web. Because responsive web design is ubiquitous, it makes sense that there should be a solution to make images responsive when they're used in HTML.

In this section, you'll learn about two approaches to using responsive images in HTML, both useful for different scenarios. These are the <picture> element and the <img> tag's

srcset attribute. Although these features are native to HTML, they aren't fully supported in all browsers, so you'll also learn how to polyfill these features in older browsers that don't support them.

Before you do that, however, we have to cover an important CSS rule that should be added to your style sheets.

5.4.1. The universal max-width rule for images

The one rule you should always have in your CSS for any website, responsive or not, is this simple code.

Listing 5.3. The universal `max-width` rule for all `img` elements

```
img{
        max-width: 100%;
}
```

This terse rule does a whole lot of good. It says that any <img> element will render to its natural width, *except* when it exceeds its container. If it does, this rule will limit its width to that of its container. [Figure 5.13](#) is an example of this in action.

Figure 5.13. A comparison of image behaviors with and without `max-width` restrictions. The example on the left is the default behavior: If the image is larger than its container, it'll exceed the boundaries. On the right is an image with a `max-width` of 100%, which constricts the image to the width of the container.

Without max-width   With max-width

With this rule in place, you're ensuring that images will behave as they normally do, except when they're larger than their parent containers. With a solid starting point in place, you can move on to using responsive images in HTML in earnest.

5.4.2. Using srcset

One method for displaying responsive images is an HTML5 feature baked into the <img> tag, and it goes by the name of srcset. This optional attribute for the <img> tag doesn't replace the src attribute but is used in addition to it.

Specifying images with srcset

One example of srcset in action follows:

```
<img src="image-small.jpg"
      srcset="image-medium.jpg 640w,
                  image-large.jpg 1280w">
```

In this example, the src attribute is used for the default image, which in mobile-first sites, should be the smallest in the image set. This acts as a fallback for browsers that don't support srcset. The srcset attribute here refers to two higher-resolution images (in bold in the preceding example). The format that the srcset attribute takes is the image URL, and the width of the image, separated by a space. Image names are in the format you'd normally use in an <img> tag's src attribute, and the width is denoted using the suffix w. For example, an image 512 pixels in width would be denoted as 512w. Additional images and dimensions can be added. Just separate them with a comma!

The strength of srcset is that it doesn't need media queries to work. The browser takes the information it's given and chooses the best image based on the current state of the viewport. This makes srcset great for circumstances where all of your images used in a given <img> tag have the same treatment but are in different aspect ratios. That's an important bit to remember. If you provide images that aren't in the same aspect ratio as one another, srcset *isn't* going to work well and will produce unexpected results. If you need a responsive image approach that allows you to have different treatments at different screen sizes, you can use media queries in CSS, or skip ahead and read about the <picture> element.

In this section, you'll continue with your image delivery optimization efforts, and implement srcset for an article image on the Legendary Tones website. But first you need to update your working copy of the website to a new branch by using git. To do this, run the following command:

```
git checkout srcset -f
```

Next, you can open index.html in your text editor and see on line 26 that a new feature image has been added. If you navigate to the website in your browser, you'll see something similar to figure 5.14.

Figure 5.14. The new feature image as it appears on the Legendary Tones website

The annotated feature image displayed in figure 5.14 resides in the img folder, and its filename is amp-xsmall.jpg. The goal of the work on this website at this stage is to get this image to scale to the width of the container while maintaining a reasonable resolution. Before you go about using srcset to achieve what you want, you'll take an inventory of images in the img folder and their widths so you can craft the value for the srcset attribute. This inventory is in table 5.4.

Table 5.4. An inventory of images in the website's img folder and their widths, which will be used for the `srcset` attribute

Image name	Image width (pixels)
amp-xsmall.jpg	320w (already referenced in src attribute)
amp-small.jpg	512w
amp-medium.jpg	768w
amp-large.jpg	1280w

Using the information in table 5.4, you can construct the content for the srcset attribute. In line 26 of index.html, change the <img> tag to the following:

<img src="img/amp-xsmall.jpg" class="articleImageFull"

```
    srcset="img/amp-small.jpg 512w,
            img/amp-medium.jpg 768w,
            img/amp-large.jpg 1280w">
```

With this srcset value, you'll have a feature image that will span all breakpoints in browsers that support it. The best part is that you don't have to write any media queries to make this work. The browser makes the best choice it can, and does all the leg work for you.

Where optimization is concerned, srcset is efficient. This is because the browser downloads only what's needed for the best visual quality. In this example, if you load the page at a large screen size, the browser will load amp-large.jpg, but if you scale down, the browser won't request amp-medium.jpg, amp-small.jpg, and so forth. The browser will adapt the image that has already been loaded. This prevents unnecessary fetching of image assets. Optimization win!

If you load the page at a smaller screen size and scale upward, the browser will download what it needs to ensure good image quality. So you get to have your cake *and* eat it too. Bottom line: srcset grabs only what it needs *when* it needs it.

Getting more granular with sizes

You may need more flexibility than what srcset alone provides. Maybe you need an image to change sizes depending on the screen's width. This is where the sizes attribute comes in.

Like srcset, the sizes attribute is used in the <img> tag. It accepts a set of media queries and widths. The media query, like a typical CSS media query, defines a point at which an image should change. The width that comes after it sets how wide the image should appear when that media query takes place. Multiple pairs of these media queries and image sizes can be separated by commas. An example is shown here:

```
<img src="image-small.jpg"
    srcset="image-medium.jpg 640w, image-large.jpg 1280w"
    sizes="(min-width: 704px) 50vw, 100vw">
```

The content of the sizes attribute in this example does two things. On screens 704px and wider, the image is instructed to take up 50% of the width of the viewport. To tell the browser this, you use viewport width (vw) units, which are a percentage of the viewport's current width. The next comma-separated rule after this, without the media query, is the default width of the image. If none of the media queries match, the image will be instructed to fill the viewport. Because of the max-width: 100% rule on all images (covered earlier in this section), the image will never exceed the width of the its container. To try the sizes attribute out for yourself, change the <img> tag on line 26 of index.html to the following:

```
<img src="img/amp-xsmall.jpg" class="articleImageFull"
    srcset="img/amp-small.jpg 512w,
            img/amp-medium.jpg 768w,
            img/amp-large.jpg 1280w"
    sizes="(min-width: 704px) 50vw, (min-width: 480px) 75vw, 100vw">
```

With the sizes attribute added to this <img> tag, the image behavior is affected in different breakpoints, as shown in figure 5.15.

Figure 5.15. The effect of the `sizes` attribute on the article image in Google Chrome. On the 704 px breakpoint, the image takes up 50% of the viewport, at the 480 px breakpoint the image takes up 75%, and the default image behavior below 480 px is to occupy the entire viewport.

After you make this change, resize the browser window and see how the image adapts to the viewport as you progress through the media queries. As with any responsive image approach, tweaking yields the best results. One rule that's helpful to follow in using sizes is that your media queries should be congruent with what you're using in your CSS. You *can* deviate, but always be sure to test, test, test!

Most of the time, srcset (and maybe sizes) ought to be fine, but in some cases you may need a responsive image approach that allows you to have different treatments at varying screen sizes. This is where the <picture> element comes in handy.

5.4.3. Using the <picture> element

srcset is capable, but it falls short when you need art direction for your images. *Art direction* is a technique that applies to responsive images, and refers to the practice of giving an image a different cropping and focal point for different screens. This is done when an image for a larger screen isn't optimal for smaller screens, for example. [Figure 5.16](#) is an example of art direction in action, with my neurotic cat as the subject of the image set.

Figure 5.16. An example of art direction across a trio of images. In the largest version, the subject has more context and surrounding details, because larger screens can accommodate more. As the screen width decreases, the image is cropped differently so the subject is still visible on smaller screens.

The srcset feature as used on the <img> tag isn't a good fit for images that need different treatments in different breakpoints, because it requires that a set of images maintain the same aspect ratio. The <picture> element has no such requirement, and will display any image at any defined transition point.

Before you embark on learning how to use the <picture> element, you need to switch to a new branch of the website's code by using git. If you have work that you'd like to keep, save it before switching. Then run this command:

git checkout picture -f

This switches you over to a new branch of code, where you'll get to experiment with the <picture> element. With the new code in place, you can start using the <picture> element to give an article image varying treatment across different devices.

Using art-directed images on the Legendary Tones website

Reload the Legendary Tones website in your browser and scroll down. You'll see a new article image of a guitar amplifier, as shown in figure 5.17. On screens smaller than 704 pixels, the image sits between the paragraphs and is centered in the viewport. On screens 704 pixels and wider, the image floats to the right, and the text wraps around it.

Figure 5.17. Image behaviors on the Legendary Tones website. On small screens (left), the image centers in the viewport and breaks between paragraphs. On large screens (right), the image floats to the right, and the text wraps around it.

The image in figure 5.17 is set in a `<picture>` element, which you can find on line 30 of index.html.

Listing 5.4. The `<picture>` element on the Legendary Tones website

What you want to add to this setup are higher-resolution images for displays that can use them, and provide an alternate cropping of the image for smaller devices. To accomplish this, you add a few `<source>` tags to the `<picture>`, thereby defining more images for the browser to use. The code added to the `<picture>` element is annotated next.

Listing 5.5. Adding new image treatments for different devices via `<picture>`

```
<picture>
    <source media="(min-width: 704px)"
            srcset="img/amp-medium.jpg 384w" sizes="33.3vw">
    <source srcset="img/amp-cropped-small.jpg 320w" sizes="75vw">
    <img src="img/amp-small.jpg">                    ◁─┐
</picture>                                               │ If <picture> isn't supported by the
                                                        │ browser, <img> will still work.
```

To achieve your goal, you add two <source> tags for different images. The first <source> tag contains a media attribute that takes effect when the screen is 704px or wider. When this condition is met, the srcset attribute will provide an image 384px in width, and render it at a third of the viewport size.

When the screen width is less than 704px, the second <source> tag kicks in. This <source> has an srcset attribute to bring in a different image treatment 320px in width, and sizes it to 75% of the viewport width. <u>Figure 5.18</u> shows the effect of your new code.

Figure 5.18. Image behaviors of the website after modifications to the `<picture>` element.

Note that small screens (left) offer a different treatment of the image based on the screen resolution.

Small screens   Large screens

The power of the <picture> element isn't necessarily in itself. It's merely a container for other elements that dictate responsive image behavior, namely the <source> and <img> elements. The <source> elements are where the image configuration is done, and the <img> tag is a fallback for browsers that don't support the <picture> element. Though the <img> tag provides the fallback behavior, it's also necessary for the <picture> element to work, so it should never be omitted.

What you've done works well enough for some lower DPI displays, but you should pump this up a bit so that high DPI displays can benefit from higher-quality images.

Targeting high DPI displays

You can target high DPI displays with the `<picture>` element quite easily. Doing so requires a bit more tweaking of the srcset attributes on the `<source>` tags. For this website, you can modify the contents of the `<picture>` element to provide a better experience for these displays. Modifications are in bold in the following listing.

Listing 5.6. Adding images for high-DPI displays by using `<picture>`

These small tweaks do two things: on larger screens, the browser will choose between an image 384px or 512px in width, and on smaller screens, the browser will choose between an image appropriate for low-DPI displays (amp-cropped-small.jpg) and an image appropriate for high-DPI displays (amp-cropped-medium.jpg).

In order to signal to the browser which image should be for which type of display, an x value is used in place of a width value in the srcset attribute. Think of it as a simple multiplier. 1x marks images appropriate for standard DPI screens, and 2x or higher signals images that are appropriate for higher DPI screens. If you wanted to, you could even use multipliers of 3x or higher, since 5K displays are out in the wild now.

Next, you'll use the `<picture>` element to specify fallbacks for different file types, and see how this can be used to take advantage of new image formats while maintaining compatibility with browsers that don't support them.

Using the type attribute for fallback images

The `<picture>` element also has the ability to reference a series of fallback images based on their type. This is useful when you want to take advantage of any up-and-coming image formats, but want to ensure that less-capable browsers will still be able to use common formats.

You can see a good example of this feature in action with Google's WebP format. WebP is a capable format that, depending on the image content, offers lower file sizes than equivalent formats.

To create a series of fallbacks in the <picture> element, you use the type attribute on <source> elements. type accepts an image's file type as an argument for an image specified in the srcset attribute.

Continuing on, you'll use the type attribute within <picture> to establish a preference for a WebP image, and fall back to a JPEG image for browsers that don't support WebP. In your text editor, go to line 30 and change the content of the <picture> element to the following:

```
<picture>
    <source srcset="img/amp-small.webp" type="image/webp">
    <img src="img/amp-small.jpg">
</picture>
```

With this, you get the best of both worlds: browsers that can handle WebP get WebP, and browsers that can't will fall back to JPEG. Furthermore, because the <img> tag in the <picture> element is the fallback, it'll work in browsers that don't support <picture> itself, making this method a great way to use any kind of format without having to worry about incompatibility.

Now that you've explored the usefulness of the <picture> element, you'll cover how to polyfill the <picture> element and the srcset attribute for older browsers by using the Picturefill library.

5.4.4. Polyfilling support with Picturefill

Although srcset and <picture> are both useful, their browser support isn't universal. Thankfully, you can take advantage of these markup patterns in browsers that don't support them by using a small 11 KB script called *Picturefill*.

Using Picturefill

Like any good polyfill, the strength of Picturefill is in its transparency. You get to use new browser features the way they were intended to be used, and all browsers will play nice. Browsers that support <picture> and srcset will use the native implementation, and other less capable browsers will use Picturefill.

Download Picturefill from https://scottjehl.github.io/picturefill and place it into your project. To see how Picturefill is used firsthand, you can switch to a new branch that already has it in place by entering this command:

```
git checkout picturefill -f
```

After the branch has loaded, open index.html in your text editor and take a peek at lines 7 and 8. You'll see these two <script> blocks in the <head>:

```
<script>document.createElement("picture");</script>
<script src="js/picturefill.min.js" async></script>
```

The first <script> block is for browsers that don't recognize the <picture> element, and prevents problems from occurring if the browser parses them in the HTML before Picturefill has finished loading. The second block then loads the Picturefill library, but does so without blocking page rendering with the async attribute (async is covered in detail in chapter 8).

Believe it or not, this is all it takes for Picturefill to work. After the script is loaded, browsers that didn't support these new image-delivery features in HTML will now support them just fine.

Unfortunately, this approach doesn't prevent browsers that support <picture> and srcset from unnecessarily downloading Picturefill. Next, you'll see how to use Modernizr to check for <picture> and srcset support, and load Picturefill for *only* the browsers that need it.

Conditionally loading Picturefill with Modernizr

Modernizr (http://modernizr.com) is a robust feature-detection library that provides a simple way to detect browser support for an array of features. In this section, I've created a 1.8 KB custom Modernizr build that contains only feature-detection code for <picture> and srcset.

You'll use Modernizr to avoid loading the 11 KB Picturefill library in modern browsers, by first checking to see whether the browser needs it. If tests for either feature fail, you load Picturefill. If they both succeed, you don't load Picturefill, and save those browsers the hassle of downloading unnecessary code.

To start, you remove line 8 of index.html, which is the <script> block that loads picturefill.min.js. You then add the following code just before the closing </body> tag:

```
<script src="js/modernizr.custom.min.js"></script>
<script>
    if(Modernizr.srcset === false || Modernizr.picture === false){
```

```
            var picturefill = document.createElement("script");
            picturefill.src = "js/picturefill.min.js";
            document.body.appendChild(picturefill);
    }
</script>
```

The preceding code begins by including the custom Modernizr build. Then, in a separate
<script> tag, you write a bit of code that checks for srcset and <picture> support in the
Modernizr object. If *either* of these checks fails, you create another <script> tag, set
its src attribute to point to Picturefill, and inject it into the DOM. When you examine
the network tab in the developer tools in different browsers, you can see that the
less-capable browser downloads picturefill.min.js, whereas the modern browser doesn't
load it. This is seen in figure 5.19.

Figure 5.19. Conditional loading of Picturefill as seen in two browsers' network request

inspectors. On the left is a version of Safari that doesn't support the <picture> **or** srcset

features and therefore loads Picturefill. On the right is Chrome, which fully supports

these features and therefore skips loading Picturefill.

With this little bit of code, you're saving those users with better browsers the trouble
of having to download Picturefill if they don't need it. Fewer requests and less code mean
faster load times for users who can support these newer features.

The next section covers how to use SVG in HTML, and the inherent flexibility of the format
when it comes to responsive imagery.

5.4.5. Using SVG in HTML

Similar to the way SVG behaves when used in CSS, SVG in HTML is the best choice when it comes to responsive imagery, assuming that what you want to depict translates well to the format.

Warning: An HTTP/2 antipattern is discussed in this section!

This section briefly discusses inlining SVGs, which is an appropriate optimization technique for sites hosted on HTTP/1 servers, but should be avoided on HTTP/2. Always ensure that the optimization techniques you choose are appropriate for the protocol version your site runs on.

As with SVG in CSS, the advantages for using SVG in HTML are in the format's flexibility. If you have image content that comports well to the format, you should seriously consider using it, because it'll save you the trouble of having to configure multiple image sources for optimal display across different devices. It's a one-size-fits-all format.

You know how to use the ⟨img⟩ tag, so you'll feel right at home using SVG in HTML. You have two nearly universally supported options when it comes to using this image format in HTML, and both methods provide the same benefits that you've seen when placing SVGs into your CSS, in terms of visual quality and ease of use in responsive web pages. The two options are:

- *Use the* ⟨img⟩ *tag*—This is the most simple method, and you can try it for yourself in index.html. The logo in the masthead points to a PNG version:

  ⟨img src="img/logo.png" alt="Legendary Tones" id="logo"⟩.

  An SVG version of the logo exists alongside logo.png in the img folder, named logo.svg. To use it, point to img/logo.svg in the ⟨img⟩ tag's src attribute:

  ⟨img src="img/logo.svg" alt="Legendary Tones" id="logo"⟩.

  When you make this change and reload the page, you'll see the SVG version of the logo in use. When you use an SVG file in an HTML ⟨img⟩ tag, there's rarely any reason to use it in a ⟨picture⟩ element or with srcset, unless you're using it in a ⟨picture⟩ element as a part of a series of image fallbacks.

- *Inline the SVG file*—SVGs are XML, which is syntactically similar to and compatible with HTML. Therefore, it's possible to copy and paste SVG files directly into HTML. Option 2 has a pro and a con. The pro is that inlining an SVG *could* help reduce page-load time by removing an HTTP request, provided that your site isn't hosted on an HTTP/2-enabled server. The downside is that this also makes the resource less cacheable from page to page.

Weigh the benefits to see which approach is better. You'll try inlining the contents of logo.svg into index.html as an example. To get started, switch to the `inline-svg` branch with this command:

```
git checkout -f inline-svg
```

Inlining an SVG image is easy. For this website, all you need to do is open logo.svg in your text editor from the img folder, copy the contents of the file to your clipboard, and replace the logo's <img> tag with the SVG file contents. Ensure that the *only* thing you're copying are the <svg> tag and its contents. Leave out anything else, such as the <?xml> header at the top of the file. The final result should look like the following listing, only without truncation.

Listing 5.7. Inlined SVG in HTML

```
<section id="masthead">
    <svg id="logo" xmlns=http://www.w3.org/2000/svg
        viewBox="0 0 216.7 34">...</svg> /
    <h2 id="tagline">Your Source for Great Guitar Tone</h2>
</section>
```

The inlined contents of logo.svg (truncated for brevity)

This scenario isn't the most optimal but illustrates the concept. A good scenario for inlining an SVG is a resource that appears on one page as part of some content. Vectorized infographics are a potential use case for inlining SVG, for example.

Even so, it's crucial to remember that the typical bottleneck of an internet connection is its latency. Inlining reduces load time by spreading latency across fewer requests. Still, effective caching is important too. Weigh your options and see what makes sense for your site.

5.5. Summary

In this chapter, you learned the importance of delivering images to devices that can best use them. As a part of this concept, you learned about these smaller, related topics:

- Delivering responsive images in CSS with media queries, and how supplying the correct image sources to the proper devices can positively impact loading and processing time
- Delivering responsive images in HTML by using the `srcset` attribute and the `<picture>` element
- Providing polyfill support for the `<picture>` element and `srcset` attribute for older browsers in an optimal fashion
- Using SVG images in both CSS and HTML, and the convenience and flexibility inherent to the format when it comes to optimal display on all devices

With these concepts under your belt, you're optimizing your projects so that your users are downloading only the image content that they need, while ensuring that they're receiving the best possible experience. Now it's time to learn image-optimization concepts and methods, such as image spriting, new image compression methods, and using the WebP format.

Chapter 6. Going further with images

This chapter covers

- Creating image sprites from multiple image files by using automated tools
- Reducing the file size of images without significantly degrading their visual quality
- Using the WebP image format from Google, and understanding how it compares to older formats
- Deferring the loading (lazy loading) of images that aren't in the viewport

In the preceding chapter, you learned the importance of optimal image delivery. This entailed using media queries to deliver images in CSS according to the user's device capabilities, as well as using new features in HTML to accomplish the same goal.

In this chapter, you'll go a step further in working with images. This entails reducing HTTP requests by combining images into sprites, reducing the size of raster and vector images through new compression methods, using Google's WebP image format, and understanding the benefits of lazy loading images.

6.1. Using image sprites

As a front-end developer, you're constantly looking for ways to improve your site's performance. With images representing a large portion of your page weight, it makes sense to want to tame these unruly critters and make them more manageable.

Warning: This section discusses an HTTP/2 antipattern!

Image sprites combine images to reduce HTTP requests, which is a form of concatenation. Although you *should* use image sprites on HTTP/1 to improve page-load times, you should *avoid* using them on HTTP/2. Check out chapter 11 for more information.

Surely you've noticed the iconography peppered throughout the sites you visit: images such as stars for ratings, social media icons, action icons that encourage the user to share content, and so on. Chances are good that these images are part of what's called an *image sprite*.

So, what *is* an image sprite? An image sprite is a collection of previously separate image files used throughout a website that have been assembled into one image file. These images are often global elements such as icons. Figure 6.1 shows an example image sprite.

Figure 6.1. An image sprite of various social media icons

Once created, an image sprite is referenced by the CSS background-image property, and manipulated by the background-position property to reveal only the relevant portion of that image within an element. The element's bounding box excludes the rest of the sprite from view, giving the impression that the element is displaying only a single image in the background.

The benefit is that you're taking what would normally be a larger number of images and reducing them to a single image. This results in more-efficient delivery of those resources and reduces the load time of the page by opening fewer connections to the web server.

In this section, you'll create an image sprite for a recipe website that has six SVG icons. Four are social media icons, and two are star icons used to represent recipe ratings. Using a command-line utility, you'll generate a sprite and the CSS necessary to use it. You'll then place this CSS into the project and replace all of the icons with the new sprite, to bring the request count of the page down from 25 to 20, and a load time of roughly 500 ms (using Chrome's Good 3G throttling profile) for these icons down to about 90 ms. After you're finished, you'll create a PNG fallback for older browsers that don't support SVG.

6.1.1. Getting up and running

Before you create the sprite, you need to download a utility to generate it. Then you'll download the website code with git. You can install the sprite generator with this command:

```
npm install -g svg-sprite
```

After this finishes installing, download the website code and run it on your local machine with the following commands in a folder of your choosing:

```
git clone https://github.com/webopt/ch6-sprites.git
cd ch6-sprites
npm install
node http.js
```

This launches a web server running the recipe website at http://localhost:8080.

Let's get started!

6.1.2. Generating the image sprite

In the img folder is a subfolder named icon-images. This contains the six separate SVG images that you'll combine into a sprite. Table 6.1 details these images.

Table 6.1. SVG icons in the recipe website that you'll combine into an image sprite

Image name	Image function	Image size (bytes)
icon_facebook.svg	Facebook icon	600
icon_google-plus.svg	Google Plus icon	938
icon_pinterest.svg	Pinterest icon	563
icon_star-off.svg	Rating star (inactive)	299
icon_star-on.svg	Rating star (active)	302
icon_twitter.svg	Twitter icon	759

These images represent six requests. By the end of this section, you'll have whittled this down to one. To generate the sprite, you'll use the svg-sprite command as follows from the root of the recipe website folder:

```
svg-sprite --css --css-render-less --css-dest=less
        --css-sprite=../img/icons.svg
        --css-layout=diagonal img/icon-images/*.svg
```

A lot is going on here, so let's go through each argument, diagrammed in figure 6.2.

Figure 6.2. The anatomy of the `svg-sprite` command as used to generate an SVG sprite with LESS mixins

When this command finishes, the generated sprite will be in the img folder, and a new LESS file named sprite.less will be in the less folder. The generated sprite should look like [figure 6.3](#).

Figure 6.3. The newly generated image sprite with annotations showing the names of the standalone files prior to being added to the sprite

Image sprites can be used for more than icons (although this is a common use of the technique). You can sprite other elements such as nonrepeatable backgrounds, button images, or other imagery that's not content-specific. With this step completed, you'll continue by updating the recipe website's CSS to use the generated sprite.

6.1.3. Using the generated sprite

The next step is to include the generated sprite.less file into main.less, both of which are in the less folder. At the beginning of main.less, add this line:

@import "sprite.less";

This adds LESS mixins for the sprite to the CSS. From here, you can replace the individual image icon references with these LESS mixins to put the sprite to use. You need to make six changes. What you'll do is search for the string .svg in your text editor, and replace each background-image reference with the corresponding LESS mixins shown in table 6.2.

Table 6.2. Icon images and the LESS mixins needed to replace them

Image name	LESS mixin
icon_facebook.svg	.svg-icon_facebook;
icon_google-plus.svg	.svg-icon_google-plus;
icon_pinterest.svg	.svg-icon_pinterest;
icon_star-off.svg	.svg-icon_star-off;
icon_star-on.svg	.svg-icon_star-on;
icon_twitter.svg	.svg-icon_twitter;

To help you with this, I'll walk you through replacing one of the images with the new sprite. Take the Facebook icon image in the first row of table 6.2, and do the following:

1.   Search for icon_facebook.svg in global_small.less in your text editor, and replace the line it appears on with the .svg-icon_facebook mixin. In this example, the line you're replacing looks like this:

background-image: url("../img/icon-images/icon_facebook.svg");

Replace this line with the .svg-icon_facebook mixin:

.svg-icon_facebook;

2.   Compile the LESS files. On UNIX-like systems, run less.sh. On Windows, run less.bat.

After completing these steps, reload the page. You'll notice that the Facebook icon looks the same as it did which is exactly what you want. Now look to see that the image sprite is in use by inspecting the image element in the browser's developer tools, and check its CSS.

At this point, it's a matter of repeating this process for the rest of the images listed in table 6.2. When finished, you'll have reduced the total number of requests from 25 to 20, and the load time for the sprite assets down from approximately 500 ms to about 90 ms.

Although the gains aren't as pronounced in this instance with just six icons, the positive effects on performance scale up as the number of images added to the sprite increase. A good example of image sprites at work is with Facebook, which uses image sprites to serve many images, such as the icons used through the site, button images, backgrounds, and so forth. If all of these icons were served separately, performance could be diminished.

6.1.4. Considerations for image sprites

So far, you've learned how to create sprites by using an SVG sprite generator. But you should keep some considerations in mind when creating them.

As I said earlier in this section, sprites are used to combine global visual elements of a page, such as iconography. Creating sprites for content-specific images isn't a fruitful endeavor. Content-specific images are usually relegated to the page they're relevant to, whereas global images appear on every page on the site. Creating sprites that contain content-specific imagery penalizes users by forcing them to download content for pages that may not use it. Figure 6.4 shows the recipe website for which you created the image sprite in this section, and annotates the images that are suitable for inclusion in sprites.

Figure 6.4. An overview of images on the recipe website that are or aren't candidates for inclusion in an image sprite. Iconography is marked for inclusion, whereas imagery such as recipe images and ads isn't.

You shouldn't be too strict as far as deciding what constitutes *global* imagery. Some iconography may not be used on all pages, but these kinds of images should still be included, because they add little to the sprite's file size. Loading them up front speeds the loading of subsequent pages, because the images will be in the browser cache by the time they're used. Each scenario is unique, so do an inventory of images and create a sprite that best fits your website.

6.1.5. Falling back to raster image sprites with Grumpicon

Earlier in this section, you created an SVG sprite by using a command-line utility. Most of the time when you're creating sprites, the images you're working with (icons and so forth) are good candidates for SVG, because they tend to comport with the capabilities of the format.

Although most browsers have SVG support, it may be necessary to specify a fallback to a more traditional format that's more widely supported. That's where Grumpicon comes in handy.

Grumpicon is a web-based tool that accepts SVG files and generates a PNG version of the sprite with fallback options. What you're interested in is converting your icons.svg sprite to a PNG version for older browsers. To get started, you'll switch to a new branch of code with the following command:

git checkout -f png-fallback

After the new code is downloaded to your computer, head over to http://grumpicon.com and upload icons.svg from the img folder. You can do this by browsing to the file on your computer, or by dragging and dropping the file on the Grumpicon beast shown in figure 6.5.

Figure 6.5. SVG files can be converted to PNG by dragging and dropping SVG files on the Grumpicon beast (or by browsing to them).

After you upload icons.svg, a zip file automatically begins downloading. Open this file and go to the png folder inside it. A file named icons.png is there. Copy this file to the img folder of the recipe website.

From here, you'll tweak the LESS mixin in sprite.less to fall back to this file in the event that SVGs aren't supported. The way you achieve this fallback is by using a cascade of background-image declarations. Open sprite.less in your text editor and find

the `.svg-common()` mixin on line 1. Change the content of this mixin to what you see in this listing.

Listing 6.1. Fallback to PNG for browsers without SVG support


```
.svg-common(){
    background: url("../img/icons.png") no-repeat;
    background: none, url("../img/icons.svg");
}
```

The fallback to the PNG version is specified first.

A multiple background image reference with an SVG fallback

This code does two things: In the first background-image declaration, you specify your fallback to icons.png. On the next line, you specify multiple backgrounds, the last of which is your SVG image sprite. After you've made the change to incorporate the fallback image, recompile your LESS files by running less.sh (or less.bat for Windows machines).

This fallback works because older browsers will read the first background property and apply it to the page. When an older browser attempts to interpret the second background property, it will fail because older browsers can't parse multiple backgrounds. These less-capable browsers will then default to the initial reference to icons.png. More-capable browsers will pick up and use icons.svg just fine, and ignore the icons.png file. This works because the browser will anticipate that icons.png isn't needed because it's overridden by the reference to icons.svg, and the browser will choose to not download the PNG file.

Now that you understand image sprites, their benefits, and how to create them for all browsers, you can learn how to reduce the file size of your images.

6.2. Reducing images

Imagine a client who has a website with a recipe collections page that's heavy with image content. This client has noticed that on all devices, this page takes a long time to load, even though the images are responsive. These pages are strong when it comes to driving traffic to other pages on the site, but the client knows that if you can cut the load time of this page, you may be able to coax the client's more-impatient users further into the site.

Reduce the size of your images automatically!

This section teaches you how to write Node scripts that will perform bulk optimizations on images in an example project. If you're interested in automating the techniques taught in this section, check out chapter 12 on using gulp.

That's where *image reduction* comes in. This process reduces the file size of images without significantly degrading their visual quality. Many image-editing programs don't produce output that's optimal for the web. A good example is Photoshop's ironically named Save for Web dialog box. Although Save for Web has useful presets and options, it doesn't quite compare to what's possible with modern image-reduction algorithms.

To get started, you need to download the client's recipe website and get it running on your machine with these commands:

```
git clone https://github.com/webopt/ch6-image-reduction.git
cd ch6-image-reduction
npm install
node http.js
```

Next, browse to http://localhost:8080 and you should see the client's recipe website, as shown in figure 6.6.

Figure 6.6. The client's recipe website as it appears in the tablet breakpoint

With the website up and running, you'll optimize JPEG images by using a Node program called imagemin. Then, you'll go further and learn how to use it to optimize PNG images. Finally, you'll learn how to use the svgo Node program to optimize an SVG image.

6.2.1. Reducing raster images with imagemin

The tool of choice for optimizing images on this site is imagemin, which is a generalized image-optimization module written in Node. It's capable of optimizing all types of images used on the web. In this section, you'll write a small Node program that uses imagemin to optimize all the JPEG images in the recipe website's img folder.

Mind your role

Maybe this goes without saying, but whether you apply these techniques to the sites you work on depends on the responsibilities that come with your role. If you play the part of both designer and developer, you have more latitude in the choices you can make. But if your organization's roles are more granular, and your responsibilities are relegated to development, loop in the project's designer on optimizations you want to make.

Unlike the work you've done up to this point, this tool requires you to write a little bit of JavaScript for Node, rather than running command-line tools installed by npm. Don't worry, though! The process is simple, and I'll walk you through it step-by-step.

Optimizing JPEG images

Before you start, let's take an inventory of the project's images. If you peek in the site's img folder, you'll see 34 recipe images ending with either -1x.jpg or -2x.jpg. The comes out to 17 pairs of JPEGs. You might have correctly guessed that these signify images intended for standard DPI and high DPI screens, respectively. This page uses the srcset attribute you learned about in chapter 5 to deliver images according to a device's capabilities.

You don't have a single baseline for how much of your page weight is represented by images, but rather two. It changes based on the type of screen and the device resolution. Table 6.3 lists screen DPI, total image payload, and the corresponding load time on the Good 3G network throttling profile in Chrome's network utility for the website.

Table 6.3. Screen DPI as it relates to the size of images and the total load time of the page

Screen DPI	Image payload	Load time

Screen DPI	Image payload	Load time
High	2089 KB	11.5 seconds
Standard	732 KB	4.38 seconds

Although devices with standard screens load the site in a somewhat expedient fashion, high DPI devices are definitely suffering. Let's give them a shot in the arm with our pal imagemin. To install imagemin for this project, run the following commands:

```
npm install imagemin imagemin-jpeg-recompress
mkdir optimg
```

The first command installs two packages: the imagemin module, and a JPEG optimization plugin for imagemin named jpeg-recompress. The second command creates a new folder named optimg that the imagemin code will write the optimized images to. After these are installed, you'll write a small Node program that does the work. In the root folder of the website, create a new file named reduce.js and add the following contents.

Listing 6.2. Using `imagemin` to optimize all JPEGs in a folder

The preceding code is simple: The require statements import the imagemin modules. You use these modules to create an imagemin object that processes all of the JPEGs in img and writes the optimized output to optimg. When you're finished, save reduce.js and run it with Node:

```
node reduce.js
```

This can take a little time. On my laptop, this command took about 10 to 15 seconds. Setting the accurate flag in listing 6.2 to false can cut processing time if the script is taking too long.

After the program finishes, you'll see that the optimg folder is populated with optimized images. Let's look at the unoptimized and optimized output of chicken-tacos-2x.jpg in each folder. Figure 6.7 depicts the two side by side.

Figure 6.7. A comparison of the unoptimized (left) and optimized versions of chicken-tacos-2x.jpg. The optimized version is about 55% smaller, but the visual differences are virtually imperceptible.

Unoptimized (181.9 KB)   Optimized (79.41 KB)

With the JPEGs optimized, you need to point index.html to them. To do this, you can copy and paste images from the optimg folder to the img folder, and choose to overwrite for all conflicts. This method involves no changes to index.html. If you want to preserve the unoptimized files, you can change the <img> tag references to point to files in the optimg folder.

After you open the page and check its load time in Chrome's Network tab, you'll notice that the load speed of the page should improve drastically, while none of the images should look much, if at all, different from their unoptimized versions.

Because of your imagemin script, you've achieved a 59% reduction of file sizes for images in both categories. The performance improvements are noted in figure 6.8, which represent about a 50% reduction in page-load times.

Figure 6.8. Website load times before and after the optimization of images for the recipe website using the Good 3G networking throttling profile in Google Chrome

With these results, you've achieved far beyond your client's expectations. All but the most impatient users ought to be satisfied with the site's improved performance, which offers them unimpeded access to more recipes through this content portal.

It's possible to further tweak the output of this program by adding and tweaking imagemin-jpeg-recompress options in reduce.js. All options are documented at the imagemin-jpeg-recompress npm package page at www.npmjs.com/package/imagemin-jpeg-recompress. Be aware that aggressive optimizations can cause noticeable degradation of images, so always compare the optimized output with the unoptimized input to ensure that the results are up to your standards.

Additionally, imagemin-jpeg-recompress isn't the only JPEG optimization library out there. You can learn more about many others at www.npmjs.com/browse/keyword/imageminplugin.

Optimizing PNG images

Optimization of PNGs with imagemin is largely the same as optimizing JPEGs, but it's beneficial to get hands-on with optimizing these image types as well. To get started, you'll pull down a new branch of code by entering this command:

git checkout -f pngopt

Other than bringing in the optimized JPEGs from earlier in this section, the only thing this command changes is that the site logo is swapped out from an SVG to a PNG image set. Two PNG files are added, logo.png for standard DPI screens and logo-2x.png for high DPI screens, which are 4.81 KB and 8.83 KB, respectively. To start optimizing, you'll need to download the imagemin-optipng plugin with this command:

npm install imagemin-optipng

After installation finishes, open reduce.js and change it to the content in this listing.

Listing 6.3. Using `imagemin` to optimize PNGs

```
var imagemin = require("imagemin"),
    optipng = require("imagemin-optipng");

imagemin(["img/*.png"], "optimg", {
    plugins: [optipng()]
});
```

Imports the optipng plugin for imagemin.

Imports the optipng plugin, processes PNGs in the img directory, and writes the output to the optimg folder.

This code works similarly to the JPEG optimizer, except you're processing PNG files instead. To test it, run reduce.js with node, like so:

node reduce.js

You should then see the optimized PNG files in the optimg directory. Figure 6.9 shows the file sizes before and after the optimizer runs.

Figure 6.9. A comparison of the logo.png and logo-2x.png files before and after optimization

Because of your optimizations, the file sizes of logo.png and logo-2x.png are reduced by about 33% to 37%, respectively. The visual quality of the images didn't suffer in the process.

It's possible to coax more out of this program by using the optimizationLevel option in the imagemin-optipng plugin. This option is the only one available in this plugin, and accepts an integer from 0 to 7. Higher values *can* further reduce file sizes. After a certain point, though, this will fail to yield better results. The default value for optimizationLevel is 2, and even when raised to the maximum value of 7 in this instance, no more gains were realized than with the default. Different images may yield different results, so experiment to see what's possible.

Other PNG optimization plugins are available at www.npmjs.com/browse/keyword/imageminplugin.

6.2.2. Optimizing SVG images

The mechanism behind optimizing SVG images is a bit different than it is with raster images. The reason behind this is that whereas raster images are binary files, SVG files are text files—so optimizations such as minification and server compression can be used on them.

The client is happy with the work you've done and has let you off the hook. Your colleague, sensing that you have time to kill, emails you about an SVG logo she's designed for her client, a timber and pulp company named Weekly Timber. She has designed a great logo, but it's 40 KB. She'd like to see if anything can be done to trim it down.

Fortunately for you, a command-line tool in Node named svgo is also available as an imagemin plugin. Because you're optimizing one file, the command-line tool will be more convenient than writing a JavaScript routine. To install svgo on your system, use npm:

npm install -g svgo

This installs svgo globally on your system so that you can use it anywhere. Then you need to grab the SVG at http://jlwagner.net/webopt/ch06/weekly-timber.svg. After it's downloaded, go to the folder it resides in at your terminal and try the following command:

svgo -o weekly-timber-opt.svg weekly-timber.svg

The format of this command is simple. It starts with the -o parameter, which is the name of the file that svgo will write the optimized output to. After that is the name of the unoptimized SVG file. When you run this command, you'll receive this output:

39.998 KiB - 28.4% = 28.656 KiB

Not too bad! Turns out svgo's default behavior optimizes a lot by simplifying the SVG's content as well as minifying it. This yields about a 28% savings compared to the original. Let's see how this impacts image quality by opening the optimized and unoptimized versions in the browser, and comparing the two. You can see this comparison in figure 6.10.

Figure 6.10. The Weekly Timber logo before (left) and after optimization with `svgo` **using the default options**

Unoptimized (39.99 KB)   Optimized (28.66 KB)

The image quality is virtually unaffected. You can open both files in a program such as Photoshop or Illustrator and compare. If you don't have imaging software, you can open SVGs in any modern browser. You shouldn't notice much, if any, difference between the two. If you do notice any differences, they should be minor. Aggressively optimized SVG images are characterized by a lack of fine details, particularly in the quality of Bézier curves.

svgo is a powerful program with a lot of options. Maybe we should dive in to see if you can further optimize this image. Type svgo -h to see additional options. One that sticks out is the -p argument, which you can use to control the precision of floating-point numbers. Try setting this value to 1, and see what the output looks like with this command:

svgo -p 1 -o weekly-timber-opt.svg weekly-timber.svg

When this command runs, you should see the following output:

39.998 KiB - 53.9% = 18.42 KiB

This yields another 25%! Let's not get too hasty in declaring victory, though. You should observe the output to see whether any anomalies have been introduced. Figure 6.11 compares the output of the original unoptimized image and your further optimized version.

Figure 6.11. The Weekly Timber logo before (left) and after (right) optimizing even further by reducing decimal precision with `svgo` **to a value of 1**

Unoptimized (39.99 KB)   Optimized (18.42 KB)

You can observe *some* noticeable differences by inspecting these images more closely, but they're still relatively minor. You *can* go too far, though. provides a closer look at what happens when you remove all precision from the same unoptimized SVG.

Figure 6.12. An examination of the unoptimized logo.svg (left) compared to an overoptimized version. All precision is stripped from the SVG shapes, resulting in a loss of fidelity, especially with Bézier curves.

This version of the SVG weighs in at 10.81 KB, which is the smallest yet, but the lower file size isn't worth the degradation. Optimizing too aggressively has drawbacks, so always, *always* ensure that the results are satisfactory to you, and *especially* to your client! You can usually identify any detrimental effects of SVG optimization by zooming in on the artwork and comparing it to the unoptimized version.

With image optimization techniques for all types of web images under your belt, you're ready to discover the usefulness of Google's WebP image format.

6.3. Encoding images with WebP

Since the early days of the commercial internet, the only available options for raster images were the JPG, GIF, and PNG formats. Little has changed in this landscape as far as new formats until somewhat recently, when Google introduced WebP.

Your recipe website client has caught wind of this new image format and wonders whether any gains can be made by using it. The client wants you to check out what gains could be made on the recipe collections page in particular.

Luckily for you, the imagemin program you've been using has a plugin for converting images to the WebP format, appropriately named imagemin-webp. Using this plugin involves using the same pattern as you've used before, so this won't be anything new for you at all.

Unlike other image formats, WebP can be encoded in both lossy and lossless formats. In this section, you'll use the imagemin-webp plugin to encode both lossy and lossless WebP images. You'll also use the <picture> element to provide a fallback for WebP-incapable browsers.

6.3.1. Encoding lossy WebP images with imagemin

Encoding lossy WebP images is easy with imagmein. It's the same pattern you used before to optimize JPEGs, except in this case, you're using it to convert JPEGs into WebP images. To get started, switch over to a new branch of the recipe client's website with this command:

git checkout -f webp

When this command finishes, you'll need to install imagemin and the imagemin-webp plugin:

npm install imagemin imagemin-webp

Now you'll write the WebP image-conversion code, which is like the other imagemin programs you've written. Create a file called reduce-webp.js and enter the following code into it.

Listing 6.4. Encoding JPEG images into lossy WebP with `imagemin`

```
var imagemin = require("imagemin"),          ◁——  Include the imagemin-
    webp = require("imagemin-webp");               webp plugin.

imagemin(["img/*.jpg"], "optimg", {
    plugins: [webp({
        quality: 40                          ◁——  Set the quality of the WebP
    })]                                            encoder to 40 out of l00.
});
```

Run this script by typing node reduce-webp.js. After it runs, all the JPEGs in the img folder will be encoded to WebP and saved to the optimg folder. Next, you'll compare the quality of one of the optimized JPEGs from section 6.2.1 to the WebP output. Figure 6.13 shows the comparison.

Figure 6.13. A JPEG optimized by using imagemin's `jpeg-recompress` plugin (left) compared to a WebP image encoded from the unoptimized JPEG at a quality setting of 40.

Optimized JPEG (79.41 KB)　　　　　　　WebP (67.67 KB)

There doesn't appear to be much in the way of huge differences. WebP has some drawbacks in that lower-quality settings can produce visual artifacts, but this is also true of JPEGs. After all images have been converted to WebP, switch all the references to JPEGs in index.html over to the WebP files. Using the Good 3G throttling profile in Chrome, let's compare the loading performance of the WebP files to the optimized and unoptimized JPEGs. Figure 6.14 shows this comparison.

Figure 6.14. A comparison of load times on the recipe website of JPEG and WebP images on standard and high DPI screens. The WebP images offer better loading performance in comparison to both the optimized and unoptimized JPEG images.

These optimizations have yielded a 35% and 20% decrease in page-load time for high-DPI and standard DPI screens, respectively, when compared to load times for the optimized JPEGs. This definitely signifies that WebP is worth the effort, even if the support for it isn't universal.

But you haven't yet investigated the potential of WebP when it comes to lossless images. In the next section, you'll investigate how WebP stacks up against PNG files.

6.3.2. Encoding lossless WebP Images with imagemin

WebP also supports lossless encoding similar to the full-color PNG format, supporting 24-bit color with full transparency. In this short section, you'll convert the PNG version of the recipe website's logo to WebP. You need to tweak only a few parts of your reduce-webp.js script. Modified lines are annotated and in bold in the following listing.

Listing 6.5. Encoding PNG images into lossless WebP with `imagemin`

```
var imagemin = require("imagemin"),
    webp = require("imagemin-webp");

imagemin(["img/*.png"], "optimg", {
    plugins: [webp({
        lossless: true
    })]
});
```

Change first argument in imagemin call.

Replace options with the lossless option and set it to true.

All you've changed here is the file wildcard in the first argument in the imagemin call to point to PNG files in the img folder, and replaced the options in the webp object with the lossless: true option, which tells imagemin to losslessly encode WebP images. After making the changes, rerun the script.

When the script finishes converting the PNG images, they'll be placed into the optimg folder. Figure 6.15 compares the file sizes of the lossless WebP files to the optimized PNG files from section 6.2.1 and the original unoptimized PNG files.

Figure 6.15. A comparison of unoptimized PNGs, optimized PNGs, and lossless WebP images

Without sacrificing the visual quality of the logo, you've managed to further optimize these images by about 40% and 33% for logo.png and logo-2x.png, respectively. Next you'll learn how to instruct browsers that don't support WebP to gracefully fall back to images they *can* support by using the <picture> element.

6.3.3. Supporting browsers that don't support WebP

Although WebP is a great image format that you can start using today, its support isn't as wide as with established image formats. If you have an audience reliant on browsers such as Firefox or Safari, they'll see something similar to figure 6.16.

Figure 6.16. Safari failing to display a WebP image

That's no good! You need to specify a fallback that other browsers can handle. That's where the <picture> element's ability to fall back to images based on their type comes in handy.

You learned about this method in <u>chapter 5</u>, but here, you'll apply it to the recipe website so that browsers that support WebP can benefit. Those that can't will be able to fall back to a JPEG image. To start, you'll switch to a new branch of code for the website by using git:

git checkout webp-fallback

After the code downloads, open the site in Chrome, and open it again in another browser that doesn't support WebP (such as Safari or Firefox). You'll notice that the images work in Chrome but fail to load in the other browser, as shown in <u>figure 6.16</u>.

At this time, open index.html in your text editor and look for the reference to the logo.webp file. The line will look something like the following code:

<img src="optimg/logo.webp"
    srcset="optimg/logo.webp 1x, optimg/logo-2x.webp 2x"
    alt="AllTheFoods" id="logo">

You'll take this <img> tag and rework it by using the <picture> element with type attribute fallbacks, as you see in the following listing.

Listing 6.6. Establishing fallbacks with `<picture>`

Note that the 〈img〉 element still retains the id attribute value of logo. This is nothing more than to make sure that the styling given to the #logo selector applies to the image. When styling 〈picture〉 elements, you should direct styles to the 〈img〉 tag, because that tag is what 〈picture〉 assigns the chosen 〈source〉 element's image to.

Using this pattern, work your way through index.html, and rework the 〈img〉 tags for the hero and collection gallery images.

Want to skip ahead?

If you'd like to skip ahead and see the end result, you can do so by entering git checkout -f webp-picture-fallback.

After you've worked your way through all the images in the HTML, open the Network tab in the developer tools for each of the browsers, and note what happens when you reload the page. Chrome will download the WebP images, and Firefox will fall back to the image types it supports. Figure 6.17 shows this in action.

Figure 6.17. The network request inspector for two web browsers for our recipe collection page. Chrome (left) can use the WebP images, but Firefox (right) can't, so it falls back to image types it supports.

Google Chrome loads WebP images   Firefox falls back to JPEGs and PNGs.

Here, you're doing two things: you're serving a beneficial image format to a significant slice of users on Chrome, and you're still giving image content to those with other browsers.

Now that you know how to use WebP images, and how to provide fallbacks to browsers that don't support them, you're ready to learn about deferred image loading, also known as *lazy loading*.

6.4. Lazy loading images

Your client appreciates the strides you've made in optimizing the images on the website but they have one last request. The client has noticed that a competitor's website loads images only when they're in the viewport. Your client is wondering whether that's something you can do, too.

The client doesn't want this functionality just because it's something shiny and new. It's a well-established technique that loads images only when they're needed. When you lazy load images, you're preventing the unnecessary loading of images in situations where users may not even see them. This saves bandwidth and decreases the site's initial load time.

In this section, you'll learn how to write a simple lazy loader in JavaScript, implement it in the client's recipe collection page, and then add basic functionality for browsers without JavaScript support. To get started writing your lazy loader, you'll download a

new repository of code via git. Run these commands to download the code and start the local web server:

```
git clone https://github.com/webopt/ch6-lazyload.git
cd ch6-lazyload
npm install
node http.js
```

When everything is running, you'll start by defining a pattern for images in your markup. If you get stuck at any point, you can type git checkout -f lazyload to view the complete index.html and lazyloader.js files. Let's begin by setting up the markup.

6.4.1. Configuring the markup

Setting up the markup for the lazy loader is the least time-consuming part of the task, but it's crucial. You need a pattern that prevents the browser from loading images by default.

To start, let's look at the page and see what images you should lazy load. You should look at what's above and below the fold on the page, and lazy load those images that fall or *could* fall below the fold. To identify where the fold is for your users, consider using the VisualFold! bookmarklet from chapter 4 at http://jlwagner.net/visualfold. Figure 6.18 shows which images you want to load normally, and has suggestions for what images you *should* lazy load.

Figure 6.18. An audit of which images make sense to lazy load and which ones don't

When you audit the page, you can immediately see two images that aren't candidates for lazy loading: the logo image and the large main image, known as a *hero image* in marketing parlance. These will be above the fold no matter what. The recipe collection thumbnails in the .collection elements, however, are perfect; they're likely to lie underneath the fold on most devices because of the space the large hero image occupies. Even if they *are* above the fold, the lazy loader will grab them on page load when the script initializes and load them anyway. You can choose to tweak which elements to lazy load later, if need be.

Four .collection elements are on the page, each with four image thumbnails in <picture> elements. One of these elements looks like this:

```
<picture>
    <source srcset="img/fish-and-chips-2x.webp 2x,
                    img/fish-and-chips-1x.webp 1x"
            type="image/webp">
```

```
            <source srcset="img/fish-and-chips-2x. jpg 2x,
                            img/fish-and-chips-1x. jpg 1x"
                    type="image/jpeg">
            <img src="img/fish-and-chips-1x. jpg" class="recipeImage">
</picture>
```

You need to do two things: move the srcset and src attributes to data attributes so that the images don't load, and add a class to the <img> element that the lazy loader script can attach to. Let's change this markup as shown here.

Listing 6.7. Preparing images for the lazy loading script

```
<picture>
    <source data-srcset="img/fish-and-chips-2x.webp 2x,
                         img/fish-and-chips-1x.webp 1x"          Points to images
            type="image/webp">                                  to lazy load.
    <source data-srcset="img/fish-and-chips-2x.jpg 2x,
                         img/fish-and-chips-1x.jpg 1x"
            type="image/jpeg">
    <img src="img/blank.png"                                    Points to image to lazy load.
         data-src="img/fish-and-chips-1x.jpg"                   src points to a placeholder.
         class="recipeImage lazy">                     ◁
</picture>                                                       Adds the class lazy, which the
                                                                lazy loader will attach to later.
```

The changes to the markup are simple. You take all srcset and src attributes on the <source> and <img> elements, and change them into data-srcset and data-src attributes. Storing the image URLs in these placeholder attributes keeps track of the image sources and keeps them from loading until you want them to.

Then, you create a new src attribute on the <img> tag that points to a 16 x 9 pixel placeholder PNG with a gray background color. This keeps shifting of the layout to a minimum by introducing a placeholder that occupies the same amount of space. The last step is to add the class lazy to the <img> tag. This is what the lazy loader script will target when it needs to load the image.

From here, you'll want to make the same changes to every other <picture> element inside the .collection elements. When you're finished, you're ready to write the lazy loading script.

6.4.2. Writing the lazy loader

With the markup pattern defined on your collection page, you can now begin writing the lazy loader script. If you have trepidation, don't worry. I'll explain every step of the way in a series of listings. Let's begin!

Laying the foundations

You'll begin by getting the foundations together. This involves creating a closure for your lazyLoader object, which isolates the scope of the script from other scripts on the page.

Create a new JavaScript file in the js folder named lazyloader.js. Then enter the contents of the following listing into the file.

Listing 6.8. Beginning the lazy loader

Okay, so there's already a quite a bit going on here. You're building an object for the lazy loader that you assign to the lazyLoader variable, and encapsulating everything inside a closure. This object will be a collection of properties and functions that facilitate the lazy loading behavior. You'll use the contents of the lazyClass property to select all image elements with a class of lazy, and store that collection later in the images property.

The processing property signifies whether the lazy loader is scanning the document for images, which is checked against later to prevent excessive script activity. The throttle property is the amount of time in milliseconds that the lazy loader needs to wait before it can scan for images again. The buffer property tops it off by specifying the number of pixels beyond the bottom edge of the viewport that will trip the lazy loading behavior for a particular image. Figure 6.19 demonstrates this property's effect.

Figure 6.19. The buffer property specifies how far out of the viewport the lazy loader will look for images to load. By extending what the lazy loader looks for beyond the viewport, you can begin loading images as you approach them to give the browser a head start.

After you have the skeleton of this structure in place, you can build the methods that initialize and destroy the lazy loader's behavior.

Building the initializer and destroyer

That heading seems a little bit like a self-important progressive metal album. The Alpha and The Omega: The Initializer and The Destroyer! All feeble attempts at humor aside, they're important parts of this script. Without them, the script doesn't have an origin from where it can perform its lazy loading goodness. Without a function to destroy the lazy loading behavior, the script will just keep going, uselessly burning up CPU time even after all the images have been loaded.

Continuing on, you'll write two new object properties. These are the init and destroy properties, each tied to a method that initializes and destroys the lazy loader, respectively. This code is shown next, leaving off from where the buffer property was defined.

Listing 6.9. The initializing and destroying functions

```
                              Start lazy loading
                                     behavior.              Get all elements by the class
         buffer: 50,                               specified in the lazyClass property.
Run      init: function(){                  ◄
scan-        lazyLoader.images = [].slice.call(document.getElementsByClassName
Images       ⟹ (lazyLoader.lazyClass));                          ◄
on init.     lazyLoader.scanImages();
         ┌─▷ document.addEventListener("scroll", lazyLoader.scanImages);     Run scan-
Run      │   document.addEventListener("touchmove", lazyLoader.scanImages); ◄─ Images for
scan-    },                                                      touchscreens.
Images   destroy: function(){   ◄───  Remove lazy loading behavior from page.
on       ┌─▷ document.removeEventListener("scroll", lazyLoader.scanImages);
scroll.  │   document.removeEventListener("touchmove", lazyLoader.scanImages);  ◄─┐
         },
         Remove scroll                               Remove scroll event
         event from page.                            for touchscreens.
```

With these additions, you're pulling up to the gas pump, so to speak. You're getting the framework laid down for the lazy load behavior to be attached to the appropriate image elements. The next piece of the puzzle is the scanImages method.

Scanning the document for images

The previous snippet of code refers to the scanImages method in many places, but you have yet to write that method. This method is fired from the scroll event (and touchmove event on mobile devices) and checks whether images with the lazy class are within 50 pixels of the viewport's bottom edge. This listing illustrates how to define the scanImages method.

Listing 6.10. Defining the `scanImages` method

```
                                                         Checks whether any images
                                                         are left to lazy load.
         scanImages: function(){
Checks       if(document.getElementsByClassName(lazyLoader.lazyClass).length === 0){  ◄
if the           lazyLoader.destroy();           ◄
document         return;                            Destroys the lazy loader
is being     }                                     if all images are loaded.
scanned
for images. ├─▷ if(lazyLoader.processing === false){
                 lazyLoader.processing = true;       ◄
Delays                                                  Processing flag is set to true to
processing ┌─▷ setTimeout(function(){                   block further code execution.
of a code  │       for(var i in lazyLoader.images){    ◄
block by a                                              Loops over all the images
specified                                               in the collection.
time.
```

```
                                                              Checks if element contains
                                                                      the lazy class.
 Checks if image      if(lazyLoader.images[i]
 element is in the      .className.indexOf(lazyLoader.lazyClass) !== -1){
     viewport.           if(lazyLoader.inViewport(lazyLoader.images[i])){
                            lazyLoader.loadImage(lazyLoader.images[i]);
                       }
                      }
                                                          Passes current element to
 Turns off      }                                         the loadImage method.
 processing
    flag.          lazyLoader.processing = false;        Specifies timeout period
                }, lazyLoader.throttle);                  via the throttle property.

               }
             },
```

This method first checks whether any more images need to be lazy loaded. If there aren't more, the destroy method is run and the lazy loader is finished. If there are more images to process, a setTimeout call is run with a for loop that uses the inViewport method to go through all the images and see whether they're in the viewport. If this method returns true for an image, it then loads the image. This behavior is protected from excessive calls from the scroll event listener by setting the processing property to true. From here, you'll need to write the inViewport and loadImage methods that the scanImages references.

Writing the core lazy loading methods

To trigger the lazy loading behavior, you need a cross-browser-compatible method to be able to determine whether a given image element is within the viewport. This is the inViewport method.

Listing 6.11. Defining the `inViewport` method

```
                                             inViewport method
                                             definition.
inViewport: function(img){
    var top = ((document.body.scrollTop ||
              document.documentElement.scrollTop) + window.innerHeight) +
              lazyLoader.buffer;
    return img.offsetTop <= top;                Finds the viewport's position and
},                                              height and the buffer threshold.

        Checks if the given image
        element is in the viewport.
```

The inViewport method is simple. It checks to see how far the user has scrolled down the page. It does this by conditionally tapping into either the document.body.scrollTop or the document.documentElement.scrollTop properties. The reason you check either one of these is that IE 9 has a compatibility problem with document.body.scrollTop always returning 0, so you fall back to a similar property using the || operator. This value is then added to the height of the window to track the bottom of the user's viewport, and then an additional buffer value is added to trigger loading of images when they're near, but not quite in,

the viewport. Figure 6.20 shows how these calculations relate to the browser viewport and the image element passed to inViewport.

Figure 6.20. The position calculations of the `inViewport` method, and how they relate to the viewport and the targeted image element. In this case, the calculation of the viewport height plus the amount of buffer space given exceeds the top boundary of the image element, resulting in a return value of `true`.

From here, you continue on to write the central piece of the program that drives the lazy loading behavior itself: the loadImage method, shown next.

Listing 6.12. Defining the `loadImage` method

```
var imgSrc = img.getAttribute("data-src"),
    imgSrcset = img.getAttribute("data-srcset");        ┐  data-src and data-srcset
                                                         │  grabbed from <img>.
if(imgSrc !== null){                  ◄───┐  Checks if data-src
                                           │  exists on <img>.

  Checks if a data-srcset exists
  on the <img> element.
    img.setAttribute("src", imgSrc);              ┐  src set to the value of
    img.removeAttribute("data-src");              │  data-src on <img>.
}

└─►  if(imgSrcset !== null){
        img.setAttribute("srcset", imgSrcset);    ┐  srcset replaced with
        img.removeAttribute("data-srcset");       │  content of data-srcset.
     }

     lazyLoader.removeClass(img, lazyLoader.lazyClass);   ◄─┐  lazy class is removed
},                                                          │  from <img> element.
```

The loadImage method first checks whether it's a direct child of a <picture> element. If this is true, the neighboring <source> elements are scanned, and their data-src and data-srcset attributes are flipped to src and srcset attributes. After this completes, the <img> element's data-src and data-srcset attributes are processed and flipped to src and srcset attributes. The browser then sends a request to the web server for those assets. The way this function is written allows the lazy loader to work on images specified by the <picture> element, as well as standard <img> with optional support for srcset. After the attributes are changed and the image loading starts, the lazy class is removed using a new method that you must define, called the removeClass method, which you can see in the next listing.

Listing 6.13. Defining the `removeClass` property

```
className
string is                                            Defines the
converted            removeClass: function(img, className){   ◄── removeClass method.
to an array. └─►        var classArr = img.className.split(" ");

                        for(var i = 0; i < classArr.length; i++){   ◄── Array is looped over.
  If this array  ┌─►        if(classArr[i] === className){
  element is the  │             classArr.splice(i, 1);             ◄─┐  ...the array item
  "lazy" class... │          }                                       │  is removed.
                  │       }

                        img.className = classArr.toString().replace(",", " ");   ◄─┐
                     }                                                              │
                                                                    Array converted into
                                                                    string and reassigned.
```

This method converts the image element's className string into an array and iterates over it. If the lazy class is found, it's removed, and the array is converted back into a string and assigned back to the image element's className property.

Turning the key and running the script

Now that everything in the object is defined, you can fire the init method inside an onreadystate change event after the lazyLoader object, like so:

document.onreadystatechange = lazyLoader.init;

This event waits for the DOM to load, and after it does, the lazy loading behavior is attached to the specified image elements. All that's left to do is to load the script by placing a <script> tag referencing lazyloader.js after the reference for scripts.min.js:

<script src="js/lazyloader.js"></script>

After the script is loaded, and assuming no syntax errors exist, you should be able to scroll down the page and see images load as they scroll into view. To see how the lazy loader works, try opening the network tab of the browser you're using and reload the page. Wait for the page to load and scroll the page. You should see new network requests appear in the waterfall graph as images lazy load. Figure 6.21 shows this behavior.

Figure 6.21. The network waterfall graph showing lazy loaded images

With the lazy loader written and finished, one last piece of the puzzle remains: providing a fallback to users who have JavaScript turned off.

6.4.3. Accommodating users without JavaScript

As small of a segment as they may seem, there are users who have JavaScript turned off or otherwise don't have it available. With the lazy loader script in place, those users won't see anything other than the image placeholders. Figure 6.22 shows this effect.

Figure 6.22. The effect of lazy loading a script on browsers with JavaScript turned off. The images never load because the JavaScript never runs.

This isn't acceptable, even if it affects such a small segment of users. The good news is that the fix is easy, thanks to your friend <noscript>. You can modify your markup by adding a <noscript> tag with the image sources set explicitly in the src and srcset attributes, like so:

```
<noscript>
    <picture>
        <source srcset="img/fish-and-chips-2x.webp 2x,
                    img/fish-and-chips-1x.webp 1x"
            type="image/webp">
        <source srcset="img/fish-and-chips-2x.jpg 2x,
                    img/fish-and-chips-1x.jpg 1x"
            type="image/jpeg">
        <img src="img/fish-and-chips-1x.jpg" class="recipeImage">
    </picture>
</noscript>
```

You'll note that the contents of the `<noscript>` tag are the `<picture>` elements as they were before you modified them for the lazy loader. When you add this code, you should add it immediately after the `<picture>` element that's lazy loaded. Try turning off JavaScript in your browser and then reload the page. You'll see something that looks like figure 6.23.

Figure 6.23. The `<noscript>` tag at work. Both the image placeholder and the image loaded in the `<noscript>` tag are visible because the image placeholder is never hidden when JavaScript is disabled.

Both the lazy loader image placeholder and the image loaded via the `<noscript>` tag are visible. This isn't going to fly with your client, so you need to make sure that the image placeholder is hidden when JavaScript is turned off. The solution to this is simple. First, add a class of no-js to the `<html>` element:

```
<html class="no-js">
```

You'll use this class to target the lazy loaded images in the markup with CSS. To do this, you add a simple rule at the end of global_small.less in the less/components folder:

```
.no-js .lazy{
    display: none;
```

```
}
```

After adding this, recompile the LESS files via less.sh (or less.bat for Windows users).

So what's going on here? By adding a no-js class to the <html> tag, and adding a style that hides the .lazy elements in the DOM, you're ensuring that you don't see both the image placeholder and the recipe image when JavaScript is turned off.

But this means that those elements will be hidden for browsers that *do* have JavaScript enabled! To fix this, you add a small bit of inline script that removes the no-js class from the <html> tag when JavaScript is available. Open index.html in your text editor, and add this one-liner script right before the closing </head> tag:

```
<script>document.getElementsByTagName("html")[0].className="";</script>
```

The end result is that browsers with JavaScript available will benefit from lazy loading, and those that have JavaScript disabled will still display the images. Those users won't benefit from the lazy loading script, but they'll receive an acceptable experience.

A note on removing HTML element classes

The preceding method employs a scorched-earth policy in removing the no-js class by emptying the entire <html> class attribute. If you have other classes you need to preserve (Modernizr classes, for example), you can use a jQuery function such as removeClass() to selectively remove the no-js class. Better yet, consider using the native classList method, which is covered in chapter 8.

With your client happy, let's summarize what you've learned and get ready to move on to the next chapter.

6.5. Summary

In this chapter, you learned the following image-optimization techniques and their performance benefits:

- Image sprites are a method of concatenating multiple images into a single file, which can save on HTTP requests. SVG sprites can easily be generated from a set of individual SVG images by using the svg-sprite Node utility.
- Not all browsers support SVG images. If you have a segment of users in your audience who can't use SVG, you can provide a PNG fallback by using the Grumpicon online utility at www.grumpicon.com.
- Images can account for a large portion of the data that your users download when they visit your site. You can reduce the size of your site's images via the imagemin Node utility, along with imagemin plugins specific to various image formats.

- If you have a large segment of users using Chrome or Chrome-derived browsers, you can serve even smaller images with equivalent visual quality by using Google's WebP image format. Using the `imagemin-webp` plugin in Node, you can even generate lossy WebP versions in place of JPEGs, and lossless WebP versions in place of PNGs.
- Not everyone uses Chrome, so you can't just slap WebP images up on your site and expect them to work for everyone. Using the `<picture>` element in conjunction with the `<source>` element's `type` attribute, you can specify fallbacks to established image types for browsers that don't support WebP.
- Deferred loading of images, or lazy loading, is a great way to reduce the initial load time of your site. This technique also saves bandwidth by avoiding loading images that your visitors may never see. By writing your own lazy loading script, you can implement this behavior on your own site. You also can accommodate users with JavaScript turned off by coming up with a solution that uses the `<noscript>` tag.

With these techniques under your belt, you'll be able to ensure that your websites accommodate rich visual content without sacrificing the speed and compatibility your site visitors demand. You're now ready to move on to the world of fonts, and learn how to optimize their delivery to the user!

Chapter 7. Faster fonts

This chapter covers

- Limiting the number of fonts through selection
- Rolling your own @font-face cascade
- Understanding the benefits of server compression for older font formats
- Limiting the size of fonts by subsetting
- Using the unicode-range CSS property to serve font subsets
- Managing the loading of fonts through JavaScript APIs

In the preceding chapter, you learned how to optimize images, but as it turns out, many other aspects of a page benefit from optimization as well. In this chapter, you'll explore yet another asset type commonly found in websites: fonts. Fonts can represent a significant portion of the payload of many websites, and the manner in which they're delivered is worth careful consideration.

The state of support for fonts has been somewhat fractured since their introduction into the developer's toolkit. Although support for the CSS @font-face property is ubiquitous, the font formats available for embedding have varying degrees of support.

The TrueType and Embedded OpenType formats enjoy wide support in browsers both antiquated and modern, so you may be tempted to call it a day and go with one of these. But these font formats aren't optimal for the web, because they're uncompressed. Newer formats such as WOFF and WOFF2 have a smaller footprint and are the most optimal for embedding. This doesn't mean that you shouldn't use older font formats. They *should* be part of a @font-face cascade, but only as last resorts for browsers that don't support newer and more optimal formats.

As you work your way through this chapter, you'll start with the basics and learn to select the fewest fonts and font variants necessary for a given page, and how to create an optimal @font-face cascade. You'll also explore how server compression can reduce the size of older font formats, and how to reduce the size of all font formats through subsetting. Finally, you'll take on the task of controlling the way fonts are displayed by using the CSS font-display property, as well as falling back to the native font-loading API in JavaScript, and then finally to the Font Face Observer library for older browsers. Let's begin!

7.1. Using fonts wisely

An optimal use of fonts begins with selection. Although font providers such as Google Fonts and Adobe Typekit do a lot to guide you in selecting fonts, at times you'll need to host fonts yourself—either because the font you need isn't available on these services, or your client has specific requirements that may prohibit you from using them.

Speaking of clients, your client from Legendary Tones is back. They've bought advertising for one of their more popular articles, and they want to spruce up that page with some nicer typefaces. In the course of this section, you'll choose the fonts and font variants you need for the site, convert them to the proper formats, and roll your own @font-face cascade. Before you start, you need to download the client's site and run it locally with these commands:

```
git clone https://github.com/webopt/ch7-fonts.git
cd ch7-fonts
npm install
node http.js
```

With the site downloaded and the web server running, point your browser to http://localhost:8080 and you'll get started!

7.1.1. Selecting fonts and font variants

At a glance, the client's website design could be improved by adding fonts. The designer on the project has recommended a nice sans serif font named Open Sans. To help out, the designer placed the Open Sans font family in a subfolder of the css folder named open-sans. A glance into this folder reveals 10 styles. Clearly, you need to be choosy; otherwise, page-load times could be significantly increased.

Want to skip ahead?

If you get stuck and want to skip ahead (or you're impatient and want to see the results), you can use the git command. Type git checkout -f fontface and you'll see the end result of the work done in this section.

So how do you tease out what font variants you need? The first step is easy. The Open Sans font family comes in two styles: italic and normal. For this site's content, you need only the normal, nonitalicized variants, so this narrows the field to a slimmer selection of five fonts of varying weights, from Light to Extra Bold.

Five isn't terribly excessive, but you can weed out some of the unnecessary font weights. This requires talking with the designer to find out what the client needs. Fortunately, they've been pretty clear about their needs and have given you a small diagram showing you what font variants you should use. Figure 7.1 shows this diagram, which annotates all of the font variants (font weights, in this case).

Figure 7.1. The client's content page with all of the font weights annotated

As you can see, the variants are determined by their CSS font-weight property value. font-weight specifies how "heavy" the affected text should be. This specification can be made in presets such as normal, bold, bolder, or lighter, or through more-specific integer values in increments of 100, starting at the lightest value of 100, and ending at the heaviest value of 900. The default value for most elements is normal, which is equivalent to a value of 400. Table 7.1 maps font-weight values with their corresponding Open Sans font variants and indicates whether you'll use them on the page.

Table 7.1. The available font variants in the Open Sans font family, their font-weight values, and whether they'll be used on the page

Font weight value	Font variant filename	Use on page?
300	OpenSans-Light.ttf	Yes
400	OpenSans-Regular.ttf	Yes
600	OpenSans-SemiBold.ttf	No

Font weight value	Font variant filename	Use on page?
700	OpenSans-Bold.ttf	Yes
800	OpenSans-ExtraBold.ttf	No

Knowing which font variants you need, you can discard those you don't and use these three: OpenSans-Light.ttf, OpenSans-Regular.ttf, and OpenSans-Bold.ttf.

By being choosy and selecting only what you need, you're lightening the load for the user by serving the necessities. Fonts aren't always diminutive assets, so it behooves you to be selective. When you're finished, you can move on to writing your own @font-face cascade.

7.1.2. Rolling your own @font-face cascade

With your font variants identified and the corresponding font files selected, you can begin embedding them into the client's website. But before you can write your @font-face declarations, you need to convert the TrueType font files to the other formats that you need.

Converting fonts

Because you have only the TrueType (TTF) fonts for Open Sans available, you need to convert them to the three other formats you need. Table 7.2 lists those formats and their browser support.

Table 7.2. Font formats, along with their file extensions and browser support. Opera Mini doesn't support custom fonts.

Font format	Extension	Browser support
TrueType	ttf	All except for and IE8 and below
Embedded OpenType	eot	IE6+
WOFF	woff	All except for Android Browser 4.3 and below, and IE8 and below

Font format	Extension	Browser support
WOFF2	woff2	Firefox 39+, Chrome 36+, Opera 23+, Android Browser 4.7+, Chrome and Firefox for Android, and Opera Mobile 36+

You can use various tools, available as web services or via download, for conversion. Conveniently enough, you can acquire some command-line utilities with npm. These are as follows:

- `ttf2eot`—Converts TTF to Embedded OpenType (EOT)
- `ttf2woff`—Converts TTF to WOFF
- `ttf2woff2`—Converts TTF to WOFF2

If fonts for your website are available only in OpenType (OTF) format, you can download the otf2ttf Node package to convert your OTF files to TTF prior to continuing. In the case of Open Sans, however, you're starting off with TTF, so you don't need this utility. To install these utilities globally so that you can use them anywhere on your system, you run this command:

```
npm install -g ttf2eot ttf2woff ttf2woff2
```

This could take a minute or so, depending on your internet connection and, but after it's finished, you'll be able to run these commands from any folder. After npm finishes, you can begin converting fonts.

Warning: Mind the licensing agreement!

The terms of use for fonts you want to use can vary from font to font, so you *need* to read the licensing agreements that come with them. Open Sans is free in every sense of the word and gives clear permissions about its use. Other font creators may require you to pay use rights. Even then, restrictions may exist on embedding. *Always* check the terms of use on the fonts you want to use and make sure that you're in compliance!

To convert the Open Sans fonts, go to the css/open-sans folder in your terminal, and start by generating EOT files for IE:

```
ttf2eot OpenSans-Light.ttf OpenSans-Light.eot
ttf2eot OpenSans-Regular.ttf OpenSans-Regular.eot
ttf2eot OpenSans-Bold.ttf OpenSans-Bold.eot
```

This should generate all of the EOT fonts you need. To generate the WOFF fonts with ttf2woff, the process and syntax are the same as with ttf2eot:

```
ttf2woff OpenSans-Light.ttf OpenSans-Light.woff
ttf2woff OpenSans-Regular.ttf OpenSans-Regular.woff
ttf2woff OpenSans-Bold.ttf OpenSans-Bold.woff
```

Finally, you'll create the WOFF2 files by using tt2woff2, which has a somewhat different syntax:

```
cat OpenSans-Light.ttf | ttf2woff2 >> OpenSans-Light.woff2
cat OpenSans-Regular.ttf | ttf2woff2 >> OpenSans-Regular.woff2
cat OpenSans-Bold.ttf | ttf2woff2 >> OpenSans-Bold.woff2
```

UNIX-like systems vs. Windows systems

In the preceding example, the cat command is used to output the contents of font files via the pipe operator to the ttf2woff2 program. The equivalent of cat on a Windows system is type.

With these commands, you've generated all of the fonts you need for an optimal @font-face cascade. Let's move on to embedding these fonts!

Building the @font-face cascade

How you build the @font-face cascade is important. When done right, it hints to the browser which formats are available and provides the optimal format. For modern browsers, you can reap the benefits of highly compressed formats such as WOFF and WOFF2, and for older browsers, you can safely fall back to less-optimal EOT and TTF formats.

A caveat on SVG fonts

If you've had experience with embedding fonts, you might be wondering where SVG fonts fit in. The short answer is that they no longer do. SVG fonts are deprecated or in the process of deprecation in future releases of major browsers. It's best to avoid them altogether.

Let's get started with writing the @font-face code by opening styles.css in the css folder. The following listing shows the @font-face code for the first font you need, which is the regular font weight for Open Sans. You'll want to place this at the start of styles.css.

Listing 7.1. `@font-face` declaration for Open Sans Regular

In this @font-face cascade, you're specifying the most optimal scenario to the least
optimal one. Let's look at the src property: this property takes a comma-separated list
of sources for the specified font. The sources are loaded in the order they are specified.
You start with a local() declaration that checks for the font on the user's system. This
is the most optimal outcome, because it saves the browser from having to download anything
altogether.

If a local source isn't found, the browser downloads one out of a set of font formats that
you converted earlier in this section. Which format is downloaded is based on the capability
of the browser. You want to start off on the right foot, so you begin with the best format,
which is the WOFF2 version. To achieve greater support, you need to fall back to increasingly
less-optimal formats for less-capable browsers. Figure 7.2 details this process.

Figure 7.2. The process of a user's browser processing a `@font-face` cascade. The browser searches for a locally installed version (if specified,) and if it can't find one, it will iterate through all of the `@font-face src()` calls for various formats of the same font.

When the request for the WOFF2 font format fails, the browser will check for the next format in the list, which is the slightly less optimal WOFF version. Most browsers succeed by this point and load the WOFF version. Other browsers will fall back to the EOT or TTF files.

After the @font-face declaration is written, you'll want to modify the font-family property on the body selector to reference this font as the default for the document, like so:

font-family: "Open Sans Regular", Helvetica, Arial, sans-serif;

This property specifies a number of fonts in order of preference. Open Sans Regular is specified first, and is thus the preferred font. The next few are fallbacks in case the @font-face you're depending on doesn't load. After you've specified the font, you can reload the document in different browsers and notice that the font is now in use.

If you check the network tab in various browsers, you'll see that Firefox and Chrome use the WOFF2 file. Safari uses the WOFF file. Older browsers such as IE8 download the EOT file. If you install the TTF font files to your system, you'll notice that no fonts at all download, and are instead referenced from your computer. This behavior can be circumvented by removing all local() sources, but this approach isn't optimal for production websites. Always be sure to test that your remote font files work properly by removing all local() source references, and add them before you deploy your site to production.

The last thing you need to do is to create @font-faces for the other two font weights, shown here.

Listing 7.2. `@font-face` declarations for remaining Open Sans font variants

```
@font-face{
    font-family: "Open Sans Light";
    font-weight: 300;
    font-style: normal;
    src: local("Open Sans Light"),
         local("OpenSans-Light"),
         url("open-sans/OpenSans-Light.woff2") format("woff2"),
         url("open-sans/OpenSans-Light.woff") format("woff"),
         url("open-sans/OpenSans-Light.eot") format("embedded-opentype"),
         url("open-sans/OpenSans-Light.ttf") format("truetype");
}

@font-face{
    font-family: "Open Sans Bold";
    font-weight: 700;
    font-style: normal;
    src: local("Open Sans Bold"),
         local("OpenSans-Bold"),
         url("open-sans/OpenSans-Bold.woff2") format("woff2"),
         url("open-sans/OpenSans-Bold.woff") format("woff"),
         url("open-sans/OpenSans-Bold.eot") format("embedded-opentype"),
         url("open-sans/OpenSans-Bold.ttf") format("truetype ");
}
```

After these @font-faces are written, you need to search styles.css for font-weight properties of 300 and 700. For selectors with font-weight properties of 700, add this rule:

```
font-family: "Open Sans Bold";
```

For selectors with font-weight properties of 300, add this rule:

```
font-family: "Open Sans Light";
```

Now the site should have the Open Sans font family displaying all text in the document in your varying font weights. Congratulations! You embedded fonts in a way that favored the smallest and best-performing font formats *first*, which lowers load times for your users. Even though older browsers receive less-optimal formats, they'll still display the custom typeface. Of course, these older formats aren't trivial to load. That's where server compression comes in!

7.2. Compressing EOT and TTF font formats

You may recall that your @font-face cascade starts off fine with high-performing formats such as WOFF2 and WOFF. But the two formats after that, although well-supported, are less optimal. The reason behind this is that the WOFF2 and WOFF formats are internally compressed. A compression algorithm is intrinsic to those formats, and server compression isn't necessary.

TTF and EOT font formats *aren't* compressed, so they're great candidates for server compression. Server compression can carry overhead for binary file types, but the process is worth it to speed the delivery of these less-optimized formats.

By default, these formats are compressed when you use the local Node web server with the compression module. But different web servers may or may not compress these formats by default, and may require further configuration. For instance, Apache web servers need to use mod_deflate to compress these files. This listing shows a portion of an Apache server configuration that specifies compression for TTF and EOT fonts.

Listing 7.3. Apache server configured to compress TTF and EOT fonts

This is just one example of a web server being configured to compress fonts. Configuring other web servers to enable this same functionality requires research. The point of this section is to point out the performance gains in compressing these file types. The benefits of compressing these formats can be seen in figure 7.3, where the OpenSans-Regular.ttf and OpenSans-Regular.eot font files are compared before and after their compression.

Figure 7.3. The size of the Open Sans Regular font before and after compression. The gains in this example are about 45%, from 212.26 KB to 113.76 KB, over the uncompressed versions. EOT compression ratios are similar.

There's a clear benefit in using server compression on these older formats. Compressing them achieves a file size similar to WOFF equivalents, so it's something of an equalizer for browsers that don't support WOFF. WOFF2 still beats out compressed TTF and EOT files, but again, not every browser supports WOFF2. The goal is to deliver the best result possible for *every* browser, and this is another method that gets you closer to achieving that goal.

For smaller assets, compression is efficient, but for larger ones such as TTF and EOT fonts, compression can take longer, resulting in a longer TTFB. Always be sure to test to see which scenario yields lower load times. For smaller files, compression may not be worth the processing time.

7.3. Subsetting fonts

Your client is happy that the site looks spiffier, but is curious about whether there's a way to lighten the load that these fonts are adding. The unfortunate reality of adding fonts to any site is that more data is going to end up being transferred over the wire. By adding three fonts, you've added around 185 KB of extra weight on browsers that download the WOFF2 fonts. Less-capable browsers fall back to WOFF, and other versions that represent approximately 260 KB of extra data. That's a *lot*, and there must be a way to trim the fat.

Fortunately, you can use a technique to reduce font size: subsetting. *Subsetting* is the practice of selecting only the characters you need in a font file and discarding the rest. A practical application of this technique involves subsetting a font by language. For example, if a site's content is in English, Latin characters should suffice. If you've ever used Google Fonts, you've taken advantage of subsetting, because it's part of the service's settings dialog box after you choose a font, as shown in figure 7.4.

Figure 7.4. Google subsetting fonts by language

Though services like Google Fonts and Adobe Typekit offer subsetting, some scenarios prevent the use of third-party services, particularly if a site's design calls for fonts not found on font services or if a font requires a subscription fee. For these and any number of potential reasons, it's good to know how to subset fonts on your own so that you can be in charge of optimal font delivery for a website in your care. In this section, you'll learn how to manually subset fonts by using a command-line tool, as well as how to use the unicode-range CSS property to serve fonts for multilingual websites.

7.3.1. Manually subsetting fonts

Subsetting fonts can be done on the command line with a Python-based set of utilities named fonttools. In this section, you'll learn about Unicode ranges, install Python (if it is not preloaded), and use the pip package manager to install the fonttools library. You'll also use the pyftsubset command-line utility to generate subsets for your fonts.

Understanding Unicode ranges

To understand the mechanism by which subsetting works, you need to know what Unicode is and how glyphs for various languages exist in predefined Unicode ranges.

If you've spent any time in web development, you've heard of Unicode, but maybe you don't know exactly what it is. *Unicode* is a standard that normalizes the way that characters for all languages are represented. More than 120,000 Unicode reservations exist for characters across various languages, and the standard is continually evolving to accommodate more.

The idea of Unicode isn't just to accommodate such a large range of characters; it's to reserve space for them in a consistent way when a Unicode character set is used. The best example of a widely used Unicode character set is UTF-8, which is the de facto standard used on the web. When a Unicode character set is used, the reserved space for a letter is the same in all documents. For example, a lowercase *p* is always located at a Unicode code point of U+0070. Figure 7.5 shows this code point among others in a table.

Figure 7.5. A portion of a table of Unicode characters from unicode.org, showing glyphs and their code points. The lowercase *p* is identified by its Unicode code point of U+0070.

You need an understanding of Unicode characters because fonts use Unicode code points to reserve spaces for specific characters.

You also use Unicode code points to subset fonts. Some fonts contain far more characters than are necessary for most uses cases. If you write your content in English, it stands

to reason that you don't need the Cyrillic characters that a font may provide. By subsetting, you export only the glyphs appropriate for a website's content. This results in smaller font file sizes, and as you know by now, smaller assets translate to lower page-load times.

The typical way of subsetting a font is by using a *Unicode range*. This range is expressed as two Unicode code points and includes all code points between them. A popular Unicode range is the Basic Latin range, which includes lower and uppercase characters from the English alphabet, numbers zero through nine, and a multitude of special symbols such as punctuation marks. This range is specified as U+0000 to U+007F. When this range is fed into a subsetting tool, it exports the specified range to a smaller file.

Finding other Unicode ranges

The Unicode Consortium's official site for the Unicode standard (unicode.org) contains an exhaustive listing of all the languages that the standard reserves space for. To find a Unicode range, go to http://unicode.org/charts and browse the listing. Click the language you're looking for, and the PDF chart for that language will have the range in the upper-left and upper-right corners of the document. For example, the range for Armenian characters is U+0530-058F.

Later in this section, you'll subset the Open Sans font to the Basic Latin Unicode range by using a command-line tool named pyftsubset, which is a part of the fonttools library. But before you can use this tool, you need to install it.

Installing fonttools

Installing fonttools is easy. In , you downloaded and installed Ruby so you could use the gem package manager to install the uncss utility. In this short section, you'll do something similar, only you'll be installing Python, which gives access to the pip package manager that you can use to install the fonttools package. This package is host to a command-line font subsetting utility named pyftsubset.

If you're a Mac user, Python comes preinstalled. Many distributions of Linux also have Python preinstalled. The easiest way to see whether Python is already installed on any system is to run the python --version command. If Python is installed, the version number will display on the screen, and you're good to go. The developer of fonttools states that the program requires Python 2.7, or 3.3 or later.

Windows users won't have the convenience of Python being preinstalled, but this is a minor obstacle. To install Python, go to http://python.org/downloads and get the installer. The installer will simplify the process for you, and requires nothing more than going through a series of steps.

After Python is installed, ensure that the pip package manager is available by typing pip
-V at the command line. If you receive an error and don't see a version number, you need
to install pip. Because Python is available by this point, this is easily remedied by running
the easy_install pip command. After it's finished, the pip installer will be available,
and you can install the fonttools package by typing pip install fonttools.

After fonttools is installed, you can check whether the pyftsubset utility is available
by entering pyftsubset --help at the command line. If your screen buffer fills up with help
text, the utility is installed, and you're ready to start subsetting fonts!

Subsetting fonts with pyftsubset

Now that fonttools is installed and pyftsubset is working, it's time to get down to business.
Because the content is in English, you want to subset the Basic Latin Unicode range. This
range contains all the letters and numbers used in the English alphabet, as well as all
the symbols you'll need, such as punctuation. When you look up the Basic Latin range on
the official Unicode website, you find that the range is U+0000 to U+007F. This is the
information you'll feed to the pyftsubset utility to generate your subsets.

To begin subsetting fonts with this utility, open a terminal window and traverse to the
css/open-sans directory within the client website folder. In order to work, pyftsubset
requires TTF, OTF, or WOFF files. For simplicity, you'll subset the original TTF files
and use the converters from section 7.1 to reconvert the subsetted TTF fonts to the other
formats that you need.

After you're in the correct folder, you'll start by subsetting the OpenSans-Regular.ttf
font file with the following command:

pyftsubset OpenSans-Regular.ttf --unicodes=U+0000-007F
--output-file=OpenSans-Regular-BasicLatin.ttf --name-IDs='*'

Quite a bit is going on in this command, so let's break it down. Figure 7.6 diagrams each
of these options.

Figure 7.6. Subsetting a font with `pyftsubset`**. The input file is specified first, followed by the Unicode range of characters you want to subset from the input font, and then the output filename. The last option is used to preserve all name table entries, which ensures better compatibility with the font converters.**

After a short wait, the program will finish and output OpenSans-Regular-BasicLatin.ttf as specified in the --output-file flag. If you look at the size of this file as compared to OpenSans-Regular.ttf, you'll observe that you've shrunk the file by about 90%, from 212.26 KB to 17.68 KB. This is a *huge* reduction in the font's overall size. What's more, this isn't even the optimal font format; Open Sans has a lot of characters because of its broad support for many languages. You won't always get this kind of mileage from subsetting your fonts, but it can pay to take time to see what's possible.

Before you declare victory, you still have to convert this subsetted font to EOT, WOFF, and WOFF2 formats with these commands:

```
ttf2eot OpenSans-Regular-BasicLatin.ttf OpenSans-Regular-BasicLatin.eot
ttf2woff OpenSans-Regular-BasicLatin.ttf OpenSans-Regular-BasicLatin.woff
cat OpenSans-Regular-BasicLatin.ttf | ttf2woff2 >>
    OpenSans-Regular-BasicLatin.woff2
```

After you finish converting all fonts, repeat the same subsetting procedure by using pyftsubset for OpenSans-Bold.ttf and OpenSans-Light.ttf, and convert those files to their respective EOT, WOFF, and WOFF2 versions. Then you need to update the @font-face sources in styles.css to reference the new subsetted font files.

On special symbols

When subsetting a font, bear in mind the content of the site that the font is destined for. A site about coffee and coffee products may use words from other languages—such as *café*, which has an accented *e* character. The Basic Latin range lacks these characters, so you may want to take care to include the glyphs you may need down the road.

With this effort, you've managed to peel quite a bit off the fonts' file sizes. Depending on the font's format, you've slimmed your fonts down anywhere from 85% to 90% of their original size. This translates into a rather large improvement in page-load time, as you can see in figure 7.7.

Figure 7.7. Load times before and after subsetting fonts. Load times are improved by well over 200%. Load times include load times for all assets on site. True-Type fonts were compressed by the server in these trials. (EOT omitted due to incompatibility; file sizes are nearly identical to TTF.)

By now, you're probably feeling unstoppable. The next challenge is subsetting by language with the unicode-range property in your @font-face cascade. With this property, you can target specific languages via Unicode ranges as you did with pyftsubset.

7.3.2. Delivering font subsets by using the unicode-range property

The client wants to present the content in more than one language. The site is, for whatever reason, particularly popular in some Eurasian countries, especially Russia.

This presents a bit of a challenge, because the client needs to get the content translated. But you can work while the content translators do their thing, because the client has given you Russian placeholder copy to work with.

The problem is that this site needs to be able to serve the Cyrillic characters that the Russian language uses. You can achieve this in one of two ways:

- You can serve up the entire unsubsetted font file so that all languages will have the characters they need at their disposal, no matter the circumstances.
- You can serve up only the subsets that are needed by that page.

Which way do you think is better? If you guessed option two, you're correct. As you saw in the previous section on manual subsetting, serving the unsubsetted font hinders performance, so you'll want to subset for other languages appropriately. In this case, you don't want to force English-speaking users to download the Cyrillic subset of the font.

This is where the unicode-range CSS property comes in. This property is specified within a @font-face definition, and its value is a range and/or set of Unicode code points in the same format as those you fed into the pyftsubset program. If the browser detects that the content of the page contains characters that fall into this range, it'll download the font. If it doesn't, it won't download the font.

This property doesn't have universal browser support, so if you intend to use it, you may need to have a fallback strategy. No well-maintained polyfills exist for unicode-range at this time, so fallbacks often require an alternate approach rather than a fallback. I'll demonstrate an alternate approach later in this section.

In this section, you'll learn how to use pyftsubset to generate a new font subset containing the Cyrillic characters you need. After you convert them, you'll then embed them in a new @font-face and use the unicode-range property to inform the browser of which characters the associated @font-face applies to. Then, you'll discuss a fallback method using JavaScript.

To get started in this subsetting exercise, you need to switch over to a new branch of code by using git. Type in the command git checkout -f unicoderange. The first thing you'll

see is that there are two HTML files: the English version of the article that you've seen before (index-en.html), and a Russian version (index-ru.html) using Cyrillic characters. Let's get started!

Generating the Cyrillic font subsets

Before you can use unicode-range to deliver the proper font subset to Russian viewers, you need to create the subset of those characters by using pyftsubset.

As you may suspect, the method for generating the Cyrillic subset with this program is much the same as it was when you generated the Basic Latin subset. The only difference is that you need a different Unicode point range to grab the characters.

Whereas the Basic Latin Unicode range is simple, the Cyrillic Unicode range is complex. It's three distinct comma-separated ranges that you pass to pyftsubset's --unicodes option. I'll provide these ranges for you in this example, but they can be found on the Unicode website. To create the Cyrillic subsets for the OpenSans-Regular.ttf font file, type in this command from within the css/open-sans folder at the command line:

```
pyftsubset OpenSans-Regular.ttf --unicodes=U+0400-045F,U+0490-0491,U+04B0-04B1
--output-file=OpenSans-Regular-Cyrillic.ttf --name-IDs='*'
```

The only differences between this command and the one you used to generate the Basic Latin subset for Open Sans Regular are the Unicode ranges you pass to the --unicodes option and the output filename. After this finishes, you'll see that a new file by the name of OpenSans-Regular-Cyrillic.ttf will be in the folder. As before, you need to convert this font into the EOT, WOFF, and WOFF2 versions:

```
ttf2eot OpenSans-Regular-Cyrillic.ttf OpenSans-Regular-Cyrillic.eot
ttf2woff OpenSans-Regular-Cyrillic.ttf OpenSans-Regular-Cyrillic.woff
cat OpenSans-Regular-Cyrillic.ttf | ttf2woff2 >> OpenSans-Regular-Cyrillic.woff2
```

After these conversions finish, repeat the process of generating Cyrillic subsets for OpenSans-Light.ttf and OpenSans-Bold.ttf to OpenSans-Light-Cyrillic.ttf and OpenSans-Bold-Cyrillic.ttf, respectively. Then convert those files to the formats you need for their respective @font-face definitions. Now you're ready to learn about the unicode-range property and use it in your new Cyrillic @font-faces!

Using the unicode-range property

Using the unicode-range property in CSS isn't much different from passing Unicode ranges to the --unicodes option when you subset fonts with pyftsubset. If you open styles.css from

the client's website in a text editor and look at the @font-face declarations, you'll notice that unicode-range has already been used for the Basic Latin subsets:

```
unicode-range: U+0000-007F;
```

The format for this property is simple but flexible. It accepts any number of single Unicode code points, ranges, and/or wildcards. Variations of this property's use are shown here.

Listing 7.4. unicode-range values

```
/* Single value */
unicode-range: U+0026;
```
Single Unicode code point expressed as a singular value

```
/* Range */
unicode-range: U+0000-007F;
```
Range of Unicode code points separated by a hyphen

```
/* Wildcard Range */
unicode-range: U+002?
```
Wildcard range of Unicode code points is specified using question marks.

```
/* Multiple Values */
unicode-range: U+0000-007F, U+0100, U+02??;
```
Multiple values separated by commas

With a Unicode range specified for the Latin subset, it stands to reason that if you visit the Russian version of the page in your browser at http://localhost:8080/index-ru.html, the Basic Latin subset shouldn't load at all, correct? As you can see in <u>figure 7.8</u>, this isn't the case.

Figure 7.8. The Basic Latin font subsets are loaded on the Russian version of the page, despite having a `unicode-range` property set to use these fonts only for pages displaying characters from the Basic Latin subset.

Name	Method	Status
index-ru.html	GET	200
styles.css	GET	200
logo.svg	GET	200
ohm.svg	GET	200
OpenSans-Light-BasicLatin.woff2	GET	200
OpenSans-Regular-BasicLatin.woff2	GET	200
OpenSans-Bold-BasicLatin.woff2	GET	200

Loaded fonts { OpenSans-Light-BasicLatin.woff2, OpenSans-Regular-BasicLatin.woff2, OpenSans-Bold-BasicLatin.woff2 }

"Wait a minute! What gives?" might be your first thought, but the fact is that this version of the site *is* using characters from the Basic Latin subset you created. This font subset contains things that are common not just in English but in Russian, too—things like punctuation and numerical characters. For numerous reasons, characters in the Basic Latin Unicode range are common in many languages. So the unicode-range property in this instance is working exactly as it ought to: it's getting the font subset only when it's needed!

What you want to do is prevent the Cyrillic font subsets from being downloaded on pages that *don't* need them. To do this, you need to establish a new @font-face for each of the new Cyrillic font subsets with their own unicode-range value. The following listing shows a @font-face declaration for the Cyrillic subset of Open Sans Regular that you'll add to styles.css.

Listing 7.5. `@font-face` for Open Sans Regular Cyrillic subset

```
@font-face{
    font-family: "Open Sans Regular";
    font-weight: 400;
    font-style: normal;
    src: local("Open Sans Regular"),
         local("OpenSans-Regular"),
         url("open-sans/OpenSans-Regular-Cyrillic.woff2") format("woff2"),
         url("open-sans/OpenSans-Regular-Cyrillic.woff") format("woff"),
         url("open-sans/OpenSans-Regular-Cyrillic.eot")
         format("embedded-opentype"),
         url("open-sans/OpenSans-Regular-Cyrillic.ttf") format("truetype");
    unicode-range: U+0400-045F,U+0490-0491,U+04B0-04B1;
}
```

Multiple character sets can be used within same font-family.

Source formats for the Cyrillic font subset

unicode-range the font applies to.

After adding this new font to styles.css, add the remaining @font-face declarations for the Cyrillic subsets of Open Sans Light and Open Sans Bold. When adding this, make sure to update the font-family and local() source names appropriately. They'll be the same as the values for the Basic Latin subsets of those font variants.

When you complete the remaining @font-faces, you'll be able to see how unicode-range affects font delivery. You have index-ru.html open, so open index-en.html in another tab and check the network utility in the Developer Tools for each page in Chrome. A comparison of the Network tab output for the English and Russian versions of the page can be seen in figure 7.9.

Figure 7.9. The fonts downloaded by the Russian version of the page (left) as compared to the English version (right), even though they both use the same style sheet. The `unicode-range` property detects whether any characters in the document exist in the defined ranges, and if so, the related `@font-face` resource is served up.

Name	Method	Status	Name	Method	Status
OpenSans-Light-Cyrillic.woff2	GET	200	OpenSans-Light-BasicLatin.woff2	GET	200
OpenSans-Regular-Cyrillic.woff2	GET	200	OpenSans-Regular-BasicLatin.woff2	GET	200
OpenSans-Bold-Cyrillic.woff2	GET	200	OpenSans-Bold-BasicLatin.woff2	GET	200
OpenSans-Light-BasicLatin.woff2	GET	200			
OpenSans-Bold-BasicLatin.woff2	GET	200			
OpenSans-Regular-BasicLatin.woff2	GET	200			

Russian version   English version

You'll notice that the Russian page pulls down the Cyrillic subsets with the Basic Latin ones, whereas the English page, not having any need for Cyrillic characters, heeds the unicode-range property and ignores that subset.

This technique has utility for any multilingual website with languages that use different character ranges. Most western languages such as German, Spanish, and French do fine with a more inclusive Latin subset, but languages such as Greek and Russian benefit from subsetting because of their different alphabets. Asian languages especially benefit from this method, because Asian languages can have thousands of characters.

Not all browsers support this property, so it pays to think about how to fall back to methods that are more compatible with older browsers.

Fallbacks for older browsers

Although unicode-range is a great feature, its overall support isn't universal. Although well supported in newer WebKit browsers and Firefox, others may not have support for it by the time you read this. These browsers will ignore the unicode-range property and download all of the font subsets found in a CSS file without discretion. Figure 7.10 shows Safari 9's behavior with the English version of the page; all fonts load on the page as though the unicode-range property never existed.

Figure 7.10. Cyrillic subsets loading on the English version of the page, regardless of the `unicode-range` property. The behavior shown is in Safari.

Name	Domain	Type	Method
OpenSans-Regular-Cyrillic.woff	localhost	Font	GET
OpenSans-Regular-BasicLatin.woff	localhost	Font	GET
OpenSans-Light-Cyrillic.woff	localhost	Font	GET
OpenSans-Light-BasicLatin.woff	localhost	Font	GET
OpenSans-Bold-Cyrillic.woff	localhost	Font	GET
OpenSans-Bold-BasicLatin.woff	localhost	Font	GET

So what can you do for browsers that don't support unicode-range? One possible approach is to create broader subsets if the increased number of glyphs won't be too detrimental to page performance. Both versions of the page would still download the extra characters, but instead of six requests over three font variants, the weight would be spread across three requests over the same number of font variants.

This approach doesn't work for content in languages such as Japanese, in which the number of glyphs can push font file sizes into the massive category. Although developers who code for sites in these languages may expect to have a large amount of a site's payload dedicated to fonts, it's not okay to push these subsets onto users who don't need them. It's not a nice thing to do to your visitors. The solution, therefore, lies in JavaScript.

For multilingual sites, developers use the <html> tag's lang attribute to define the document's language. These language codes conform to the ISO 639-1 standard. In index-ru.html, this looks like <html lang="ru">. You can write a small bit of inline JavaScript that checks the language code in this tag. If the language code is what you're looking for, you load a separate, smaller style sheet that contains the @font-face declarations for the subsets that you want to defer loading.

To implement this for Russian content, you start by moving the Cyrillic @font-face definitions into a separate CSS file named ru.css. You then reference ru.css with a <link> tag containing a placeholder data-href attribute that stores its location, along with a data-lang attribute that stores the content language code it's intended for. This prevents the CSS from being loaded at all until it's evaluated by the following <script> block. If the script determines that the <html> lang attribute's value matches that of the <link> tag's data-lang attribute, it'll download and parse that style sheet immediately. This listing shows this mechanism in action.

Listing 7.6. Deferring loading of font subsets with JavaScript


```
<!doctype html>
<html lang="ru">                          ◁——  lang attribute is set to a language
    <head>                                  code of "ru" (Russian).
        <title>Легендарные Тонизирует - Этого не случится.</title>
        <link rel="stylesheet" href="css/styles.css" type="text/css">
        <link rel="stylesheet" data-href="css/ru.css"
              data-lang="ru" type="text/css">    ◁——  @font-faces for the Cyrillic
        <script>                                      subsets are moved to
            (function(document){                      another CSS file.
                var documentLang = document.querySelector("html")
                ➥ .getAttribute("lang"),
```

<html> tag's lang attribute is saved to a variable.

| <link>

```
                    linkCollection = document
<link> tag              .querySelectorAll("link[data-href]");        tags with
collection is                                                        data-href
looped over.        for(var i = 0; i < linkCollection.length; i++){  attributes
                        var linkLang = linkCollection[i]             are saved to
<link> tag's data-lang      .getAttribute("data-lang"),             a variable.
attribute is captured.  linkHref = linkCollection[i]
                            .getAttribute("data-href");              <link> tag's data-href
                                                                     attribute is captured.
data-lang attribute is  if(documentLang === linkLang){
checked to see if it matches    linkCollection[i].setAttribute("href", linkHref);
the document language.  }
                    }
                }) (document);                      Appropriate <link> tag's data-href
            </script>                               attribute is changed to an href attribute.
            <noscript>
                <link rel="stylesheet" href="css/ru.css" type="text/css">
            </noscript>
        </head>
                                                    A <noscript> fallback to
                                                    download the font subsets
```

Because this <script> block is near the beginning of the document, the browser will discover it almost immediately, so it's executed sooner. This keeps delays to a minimum.

The script also has the capability of handling multiple <link> tags that follow the data-href pattern. This gives the code the flexibility to contain references to as many <link> tags as necessary for additional font subsets. You could place this code in the <head> of every single page of a multilingual website, and only the CSS and font subsets you need for that page's language would be loaded.

You should consider users who may have JavaScript disabled. To cover this, you'll serve the font subset via a <link> tag nested in a <noscript> element. This fallback isn't optimal because it won't discriminate based on the <html> lang attribute's value, but it'll ensure that users get the font subset necessary for the page.

The effect on your client's website is that you can have the same end result as with the unicode-range property, only in every browser. Figure 7.11 illustrates the effect of this script on both the Russian and English versions of the article as loaded in Safari.

Figure 7.11. The contents of the network tab in Safari on both the English (left) and Russian (right) versions of the content page, with your fallback script enabled on each page. The English version downloads only the fonts it needs, whereas the Russian version grabs the additional ru.css and the font subsets contained therein.

Name	Name
index-en.html	index-ru.html
styles.css	styles.css
logo.svg	ru.css
ohm.svg	logo.svg
OpenSans-Regular-BasicLatin.woff	ohm.svg
OpenSans-Light-BasicLatin.woff	OpenSans-Regular-Cyrillic.woff
OpenSans-Bold-BasicLatin.woff	OpenSans-Regular-BasicLatin.woff
	OpenSans-Light-Cyrillic.woff
	OpenSans-Light-BasicLatin.woff
	OpenSans-Bold-Cyrillic.woff
	OpenSans-Bold-BasicLatin.woff

English content page   Russian content page

Is this solution remotely as optimal as unicode-range? Nope! It simply illustrates that JavaScript solutions can be crafted if there's a genuine concern. Simpler JavaScript solutions for simpler scenarios could be written. A server-side approach may make more sense to you as well. For example, you could store the language code in a cookie, and use a server-side language such as PHP to inject the <link> element for the font subset into the document based on a condition.

As time marches on, unicode-range will gain more support until it's eventually preferable to allow older browsers less-optimal experiences. The idea is that you have options in case unicode-range *isn't* one of them.

In the next and final section of this chapter, you'll learn to control the way fonts are displayed via CSS and JavaScript mechanisms.

7.4. Optimizing the loading of fonts

Loading any asset on a website involves pitfalls that vary based on the asset's type. For instance, loading CSS with the <link> tag blocks rendering until the style sheet is downloaded and parsed and the styles are applied to the document. <script> tags that reference external JavaScript files similarly block rendering of the page when they're placed toward the top of the document.

Fonts are no different, and loading them causes no shortage of issues that can have ramifications on the readability of your site. In this section, you'll learn about the visual anomalies that can occur as fonts load. You'll then learn how to control the way fonts are displayed by using the font-display CSS property, and then fall back to using the JavaScript-based font-loading API when font-display is unavailable. If neither method is available to the browser, you'll learn how to fall back to a third-party script to achieve the same results.

7.4.1. Understanding font-loading problems

The Legendary Tones owners have sent an email saying that, although they're happy with the way the fonts look, they're noticing that text on the page seems to take a while to render on slow connections. This is understandable, but the reality is that this is how some browsers work when it comes to downloading fonts. The phenomenon the client is referring to is called the *Flash of Invisible Text*.

Flash of Invisible Text (henceforth referred to as *FOIT*) is similar to the Flash of Unstyled Content (FOUC) anomaly, only instead of unstyled content, you're dealing with text being invisible until the document's fonts are fully loaded. It's noticeable on even fast connections if you pay close attention. As connection speed decreases and network latency increases, the problem becomes more evident. Mobile devices on slow mobile networks such as 2G and 3G are more susceptible to this phenomenon. You can see this problem in figure 7.12.

Figure 7.12. As a page loads embedded fonts, the text is initially invisible (left) until the fonts fully load, at which point the text styled in those font faces appears.

This seems like an annoying bug, but it's how browsers are designed to behave. Browsers wait to render text while downloading fonts in order to avoid an effect known as the *Flash of Unstyled Text* (*FOUT*). FOUT is similar to FOUC talked about in chapter 3, except that instead of an unstyled page, text initially loads in a system typeface and then suddenly

re-renders with the custom typeface applied. The browser will hide text for only so long while a font loads, and when this period of time is exceeded, the unstyled text shows up before the font has finished loading. After the font has loaded, the unstyled text is styled. This is shown in figure 7.13.

Figure 7.13. When the download time for a font is too long, the text will eventually become visible, but is unstyled because of the still-loading font resource (left). After all of the fonts load, the text will become styled (right). This is known as FOUT.

The browser's intentions are good, but if a connection stalls, the user could be left waiting for 3 seconds or longer to see text on the page. In browsers such as Safari, the content may never show if the request stalls. If the user aborts loading the page, or the font assets otherwise fail to load, the content may remain permanently invisible until the page is refreshed. This holds true even if the developer of the website has specified fallbacks to system fonts in the font-family properties. Newer versions of Chrome try to mitigate this issue automatically, but it's not perfect, nor does every browser try to remedy the issue under the hood.

So what can you do? You embrace the FOUT on page load, and use the CSS font-display property. This property lets you ensure that your content will appear as soon as possible, and won't leave your users in a lurch with hidden text. Let's get started!

7.4.2. Using the CSS font-display property

The font-display property in CSS provides a convenient way to control the display of fonts with a minimum of effort. Though this approach is limited to Chrome browsers at the time of this writing, it's the best first resort in your plan to control the display of fonts. To get started, let's check out a new branch of the website code with git:

```
git checkout font-display
```

Want to skip ahead?

If you get stuck or want to skip ahead to see how the completed font-loading API code looks and behaves, you can do so by entering git checkout -f font-display-complete at the command line.

After the code has downloaded to your computer from GitHub, open index.html and styles.css in your text editor.

Controlling how and when fonts display

To get started, open the Developer Tools in Chrome and change your network throttling profile to Regular 3G. You'll be able to see the FOIT effect quite easily. As a general rule, the slower the connection, the more noticeable this effect. To pinpoint the moment when fonts become visible on the screen, you can toggle the Capture Screenshots button in the Network tab, as shown in figure 7.14.

Figure 7.14. The toggle button to capture screenshots in Chrome Developer Tools

When this button is toggled and the page is reloaded, a roll of screenshots of the page load populates above the network request waterfall chart. With this, you can pinpoint the exact moment that fonts appear on the page. With the Regular 3G throttling profile selected, the text doesn't appear until about 875 milliseconds from the time the page begins to download. This is okay, but depending on connection speed and latency, this could fluctuate. The goal is to allow the user to see content as soon as possible.

Learn a bit more about font-display on the web!

You can learn a little bit more about the font-display property, including how to detect support for the property, by checking out an article I wrote for CSS-Tricks at https://css-tricks.com/font-display-masses.

One way to control this behavior is via the font-display property in CSS. Although not universally supported, this property gives you a great degree of control over the way fonts are displayed. This property is placed inside a @font-face declaration and accepts one of the following values:

- auto—The default value. In most browsers, this is analogous to block.
- block—Blocks the rendering of text until the associated font is loaded. This is the effect described in the previous section, and what you're trying to overcome.
- swap—Fallback text is shown first. After the font is loaded, the custom typeface is swapped in.
- fallback—A compromise between auto and swap. For a short time (roughly 100 ms), text is invisible. If this elapses and the font isn't yet loaded, the fallback text appears. After the font is loaded, the custom typeface is swapped in.
- optional—Exactly like fallback, except that the browser is given more latitude to decide whether a font is downloaded or applied. This setting kicks in when the user's internet connection is sufficiently slow. This setting is particularly useful for sites where custom typefaces are considered entirely optional.

On the Legendary Tones article page, you'll use a font-display value of swap. To set this property, open styles.css in the css folder and locate the @font-face declarations at the top of the file. Inside the first @font-face declaration, add the font-display property.

Listing 7.7. Using the font-display **property**

```
@font-face{
    font-family: "Open Sans Light";
    font-weight: 300;
    font-style: normal;
    src: local("Open Sans Light"),
        local("OpenSans-Light"),
        url("open-sans/OpenSans-Light-BasicLatin.woff2") format("woff2"),
        url("open-sans/OpenSans-Light-BasicLatin.woff") format("woff"),
        url("open-sans/OpenSans-Light-BasicLatin.eot")
            format("embedded-opentype"),
        url("open-sans/OpenSans-Light-BasicLatin.ttf") format("truetype");
    font-display: swap;                    ⟵⎤  font-display used inside an
}                                            ⎦  @font-face declaration
```

With this one property in place, reload the page on a slower network-throttling profile and you'll see that the text displays progressively without being hidden by the browser.

This setting represents the easiest possible solution for controlling the rendering of fonts when you have control over the CSS that delivers them, but it doesn't have wide support in browsers, nor does it allow you to control the way that fonts are displayed when they're referenced from third-party providers such as Typekit or Google Fonts. That's when you

can fall back to a more widely supported JavaScript solution, known as the font-loading API.

7.4.3. Using the font-loading API

The *font-loading API* is a JavaScript-based tool that controls how fonts are loaded. Its open-ended nature gives you a lot of latitude in determining how to apply typefaces to a document, whether they're hosted on your own server or with a font provider such as Google Fonts. To use the font-display CSS property discussed in the previous section, you need to have control over the CSS that serves fonts. This is a luxury you don't have when you use third-party font providers. The font-loading API gives you a similar ability to control the display of fonts regardless of their origin, but through JavaScript rather than CSS.

Before you get started, you need to use git to switch to a new branch of code. Go into your terminal window and type git checkout -f font-loader-api. When this is complete, you're ready to go

Want to skip ahead?

If you want to skip ahead and see the work at the end of this section, you can do so by typing git checkout –f font-loader-api-complete.

To get started, look in styles.css for font-family definitions that use custom typefaces. For this site, you have three font variants: Open Sans Light, Open Sans Regular, and Open Sans Bold. Table 7.3 lists these font-family definitions and the selectors they apply to.

Table 7.3. Embedded fonts' font-family property values and their associated CSS selectors

font-family	Associated CSS selectors
Open Sans Light	.navItem a
Open Sans Regular	body
Open Sans Bold	.articleTitle.sectionHeader

You'll use this information to do two things. The first is to replace the font-family properties for all of these with system fonts. For this website, you'll use the following property and value for the associated selectors in table 7.3:

```
font-family: "Helvetica", "Arial", sans-serif;
```

This removes the Open Sans font families from the page, which allows the content to be seen immediately, because these fonts aren't downloaded from the web server.

Second, you nest these selectors under a class that you'll put on the `<html>` element when the fonts have been loaded. But before you write the font-loading script, you'll write the CSS for applying the Open Sans fonts after this class is applied to the `<html>` element. This is shown next.

Listing 7.8. Controlling font display by using the `fonts-loaded` class

```
.fonts-loaded body{
    font-family: "Open Sans Regular";
}

.fonts-loaded .navItem a{
    font-family: "Open Sans Light";
}

.fonts-loaded .articleTitle,
.fonts-loaded .sectionHeader{
    font-family: "Open Sans Bold";
}
```

By placing this small snippet of CSS at the end of styles.css, you can control *when* you apply the typefaces you're loading with the font-loading API. Because you've specified the system fonts as the initial font set, the unstyled text will be immediately visible when the page first renders, and the custom typefaces will be applied after they've loaded.

Open index.html in your text editor to start writing your font-loading script. After the `<link>` tag that imports styles.css (which imports our fonts,) add the code in the following listing.

Listing 7.9. Using the font-loading API

```
(function(document){                              Check for the presence
    if(document.fonts){                           of the font-loading API.
Font-loading API uses     document.fonts.load("1em Open Sans Light");
the load() method to      document.fonts.load("1em Open Sans Regular");
   load the fonts.        document.fonts.load("1em Open Sans Bold");

ready.then method         document.fonts.ready.then(function(fontFaceSet){
runs when all the             document.documentElement.className += " fonts-loaded";
specified fonts           });                          fonts-loaded class added to
have loaded.          }                                   the <html> element.
                      else{
                          document.documentElement.className += " fonts-loaded";
                      }                          If the font-loading API is unavailable, add the
    })(document);                                 fonts-loaded class to the <html> element.
```

Because you're managing three font variants, you initiate a separate call to load each typeface. The core property of the font-loading API that you're using to achieve this is the font object's load method. Rather than using the API to explicitly load a font by its URL, you rely on CSS to define @font-faces for the document. But just because the @font-faces are defined doesn't mean that the browser downloads those fonts. Browser behavior is well optimized, and modern browsers will inspect the document to see whether any defined @font-faces are in use. If they are, they'll be downloaded, but because you initially set all of your font-family values to use system fonts, none of the font variants are downloaded until you tell the browser to do so via the load() method shown in listing 7.9.

When all of the fonts have loaded, the fonts-loaded class is added to the <html> element. This defeats the browser's initial FOIT, allowing the content to be read as soon as the document is loaded and the CSS is applied. *Then* the fonts are applied to the document when they're available. This ensures that no matter what may happen on the user's end, the text will be visible as soon as possible, and if a font fails to load, the text will remain that way.

One drawback of this method is that it *does* cause a repainting of text elements on the page, but the increased accessibility is worth the trade-off. If you choose system fonts that are similar to custom typefaces, document reflow can be minimized.

Optimizing for repeat visitors

Our solution works great for first-time visitors, but you need to optimize for repeat visits when the fonts are already in the user's browser cache. With the code as it is now, the FOUT occurs on subsequent page visits even with the font in the cache. You can overcome this by using a cookie and modifying your font-loading code slightly.

Let's modify two parts of the code you've written. On the line where you check for the font-loading API, you add a condition to check for the presence of a cookie:

```
if(document.fonts && document.cookie.indexOf("fonts-loaded") !== -1){
```

This change adds a check for a cookie you'll define later. This cookie's name is fonts-loaded and it has no particular value. You check for its existence by using the indexOf string method, which returns (somewhat unintuitively) a value of -1 if the search string isn't found. This ensures that the font-loading code you've written runs only if the font-loading API is available *and* a fonts-loaded cookie hasn't been set.

But now you need to *set* that cookie somewhere. To do that, you add this bit of code after the line where you add the fonts-loaded class to the <html> element:

```
document.cookie = "fonts-loaded=";
```

This adds an empty cookie by the name of fonts-loaded for the current domain. When this cookie is set, and the user navigates to subsequent pages, the font-loading code doesn't run again. The else condition therefore takes effect immediately and adds the fonts-loaded class to the <html> element.

This reintroduces the FOIT, but the risks of the effect are now mitigated because the fonts are in the user's browser cache. This is okay so long as you can be assured that the fonts will load. Now that the fonts are in the cache, the assurance can be made that the effect won't block the user from ever seeing the content on the page.

JavaScript is a fine way to check for the cookie and apply the fonts-loaded class. If you *really* want to be speedy about it, you could use a back-end language (for example, PHP) to modify the document so that the fonts-loaded class is on the <html> element when the content is sent by the server. You remove the else condition that adds the class in the JavaScript, and modify the output by checking for the cookie on the back end. Here's how this is done in PHP.

Listing 7.10. Conditionally adding the `fonts-loaded` class via PHP

```
<?php if(isset($_COOKIE["fonts-loaded"])){
    ?><html class="fonts-loaded">
<?php }
else {
    ?><html>
<?php } ?>
```

Check if the fonts-loaded cookie is set via the isset() function.

If the cookie is set, add the fonts-loaded class on the <html> element.

If the cookie isn't set, don't modify the <html> element.

By using a back-end language to modify the response, you're changing the <html> element before it's sent to the client. That said, the JavaScript solution is serviceable, so both

approaches are reasonable solutions. It all depends on the tools you have at your disposal, your skill set, and the time available to you.

Accommodating users with JavaScript disabled

As always, it comes back to users with JavaScript disabled. Because of the way you've developed this solution, the @font-faces you've specified will never take effect because the font-loading scripts never run and apply the fonts-loaded class to the document. As a result, the content will be displayed using the system fonts you specified as the first to appear.

If you or your organization doesn't care that this small segment of users never gets to see your fancy new font faces in action, feel free to call it a day. But you or your organization may well take umbrage with this, so let's go over a quick fix that involves our old friend the <noscript> tag. You can use <noscript> to trigger the default browser-loading behavior by nesting an inline <style> tag that applies the Open Sans font families as the default.

Listing 7.11. `<noscript>` alternative to JavaScript font loading

```
                Buckets styles for users
                with JavaScript disabled

<noscript>                              Assigns fonts to elements without
    <style>                             respect to the fonts-loaded class
        body{
            font-family: "Open Sans Regular", "Helvetica", "Arial", sans-serif;
        }

        .navItem a{
            font-family: "Open Sans Light", "Helvetica", "Arial", sans-serif;
        }

        .articleTitle,
        .sectionHeader{
            font-family: "Open Sans Bold", "Helvetica", "Arial", sans-serif;
        }
    </style>                            Buckets styles for users
</noscript>                             with JavaScript disabled
```

By adding this little bit of inline CSS, you're returning the user without JavaScript to the browser's default font-loading behavior. This means that although they won't be able to reap the benefits of the font-loading API, they'll at least be provided with a base level of functionality. Continuing on, you'll learn about the Font Face Observer library to polyfill what the font-loading API provides.

7.4.4. Using Font Face Observer as a fallback

The unfortunate reality is that the font-loading API isn't yet universally supported. It has strong support in modern browsers, but some browsers (for example, IE) are lacking. Your client would appreciate it if you could make sure that more browsers receive an optimal font-loading experience. This is where a polyfill such as Font Face Observer comes in.

Font Face Observer (https://github.com/bramstein/fontfaceobserver) is a font-loading library by Danish developer Bram Stein. Although it's not a direct polyfill in the sense that you can drop it into a page and have your existing font-loading API code work without a hitch, it gives the developer similar ability to manage font loading.

In this section, you'll write a script that kicks in when the font-loading API isn't available and that loads two external scripts: the Font Face Observer script, and a script that loads the fonts via Font Face Observer. To get started, you need to download new code from GitHub. Type git checkout -f fontface-observer, and after the code has downloaded, you'll be ready to start!

Conditionally loading the external scripts

After downloading the new branch with git, you'll notice a js folder containing two scripts: fontfaceobserver.min.js, which is the minified Font Face Observer library, and fontloading.js, which contains an empty closure where you'll place the alternative font-loading behavior. Because you don't want to invoke the overhead of the Font Face Observer script in all browsers, you want to load it and the script with your fallback-loading behavior only when the font-loading API isn't available. To do this, you add the code in the following listing between the initial if conditional that checks for the document.fonts object and the fonts-loaded cookie, and the else conditional that follows it.

Listing 7.12. Conditionally loading Font Face Observer and font-loading scripts

The preceding code is simple. If the font-loading API isn't available *and* the fonts-loaded cookie hasn't been set, you then create new <script> elements for both Font Face Observer and the font-loading script, and set their src attributes to their respective locations. To ensure that they don't block page rendering, you set the defer attribute for both. To set everything up, you instruct the browser to load these scripts by appending them to the end of the <head> element.

Writing the font-loading behavior

Open js/fontloading.js in your text editor and you'll notice that the content of this file is an empty JavaScript closure. Starting at line 2, add the contents of this listing to the file.

Listing 7.13. Using Font Face Observer to control the loading of fonts

Font Face Observer's syntax is similar to that of the font-loading API, but with slight differences. You define a FontFaceObserver object for each font variant you want to load. Then, through a JavaScript promise, you wait until all fonts have loaded. After the fonts have loaded, you apply the fonts-loaded class to the <html> element and set the fonts-loaded cookie. This allows you to reuse the mechanism by which you control the display of your fonts that you used in the font-loading API.

The result of this effort is an effective and widely compatible method that uses a native API where available, but then falls back to a capable polyfill. With this code in place, your client is happy with the font rendering, and as you know, a happy client is the only kind that you want.

7.5. Summary

In this chapter, you learned the following font optimization and delivery techniques:

- You can proactively lighten page weight by selecting only the font variants you need. Although it seems like common sense, it pays to audit your font selections. Doing so can improve your site's load times.
- Building an optimal @font-face cascade can help your site's performance by preferring locally installed fonts first, and then falling back to a set of the most optimal formats to the least optimal.
- You can compensate somewhat for the shortcomings of the TTF and EOT formats by compressing them on the server.
- Subsetting fonts can reduce the size of font files by limiting them to only the characters you need for the language of your site's content.
- Using the unicode-range property in modern browsers can assist you in using only the necessary font subsets as per your site's content language.
- If you need to selectively serve font subsets, you can write a script that can be used to serve subsets when unicode-range isn't an option.
- You can control how fonts are displayed by using the font-display property in CSS. Failing that, you can use the font-loading API to control how fonts display when font-display isn't available, or if you don't have control of the CSS that serves fonts, as in the case of third-party font providers such as Google Fonts or Typekit.
- If the font-loading API isn't available, you still have the ability to control how fonts are loaded and displayed through the use of the third-party Font Face Observer library.

Now that you're comfortable with these techniques, you can go forward in your web projects and apply them for the benefit of your clients (with your team's blessing, of course). In the next chapter, you'll learn how to optimize your application's JavaScript through techniques such as controlling the loading behavior of <script> elements, using high-performance native JavaScript APIs, working with leaner alternatives to jQuery, and more.

Chapter 8. Keeping JavaScript lean and fast

This chapter covers

- Affecting the loading behavior of the `<script>` tag
- Replacing jQuery with smaller and faster API-compatible alternatives
- Using native JavaScript methods to replace jQuery functionality
- Animating with the `requestAnimationFrame` method

The world of JavaScript has exploded into a mélange of libraries and frameworks, leaving us with a slew of options for developing websites. In our excitement to use learn and use them, we often forget that the surest path to a fast website is a willingness to embrace minimalism.

This isn't to say these tools don't have a place in the web development landscape. They can be quite useful and can save developers hours of writing code. The goal of this chapter, however, is to promote minimalism in your website's JavaScript for the benefit of your users.

In this chapter, you'll dive into what *you* can do to improve the performance of script loading on your website. You'll also spend time learning about jQuery-compatible libraries that do much of what jQuery does, but with smaller file sizes and better performance. You'll go one step further and investigate how to replace jQuery with in-browser APIs that deliver much of what jQuery provides, but without the overhead. Finally, you'll learn to use the requestAnimationFrame method to code high-performance animations. Let's get started!

8.1. Affecting script-loading behavior

As with the `<link>` tag when loading CSS, the `<script>` tag can hinder the rendering of a page, depending on the tag's placement in the document. You can also modify script-loading behavior via the element's async attribute. Let's look at these aspects of script loading, and get a feel for how they can impact performance. You'll start by revisiting the Coyle Appliance Repair website. Download and run it from GitHub with the following commands:

```
git clone https://github.com/webopt/ch8-javascript.git
cd ch8-javascript
npm install
node http.js
```

Let's start by experimenting with the placement of the `<script>` tag.

8.1.1. Placing the <script> element properly

As you may recall from chapters 3 and 4, the placement, and even the presence, of the <link> tag can block rendering of a page when loading CSS. The <script> tag is responsible for this same kind of behavior as well, but because scripts don't impact the appearance of a page like CSS imports, you have more flexibility in placing <script> tags. Figure 8.1 diagrams this behavior.

Figure 8.1. Browsers read HTML documents from top to bottom. When links to external resources (such as scripts, in this case) are found, the browser stops to parse them. When parsing occurs, rendering is blocked.

This behavior can have an impact on when the browser first paints the page. If the browser detects a <script> tag in the <head>, for example, it pauses what it's doing to download and parse the script. As this goes on, the browser puts the rendering of the page on the back burner. In your client website's code, the <script> tags for jquery.min.js and behaviors.js will be in the document's <head> tag, which induce render blocking. You can measure this effect by checking the document's Time to First Paint. The way to measure this was first described in chapter 4, but let's recap the process.

To measure the Time to First Paint for the client's website, select the Regular 3G throttling profile to simulate page loading on a slower connection. Then go to the Timeline tab and head to http://localhost:8080. When the page loads and the timeline populates, go to the bottom of the Timeline pane and switch to the Event Log tab. Once there, filter out all event types except for painting events. The first event to appear in the list is the Time to First Paint, which should look something like figure 8.2.

Figure 8.2. The Time to First Paint in Chrome for the Coyle Appliance Repair website with `<script>` tags in the `<head>` of the document.

The average Time to First Paint for the client's website is roughly 830 ms when the `<script>` tags are in the `<head>`. This seems like a long time for the page to start painting. Experiment and see how that figure changes when you move these scripts to the end of index.html, just before the closing `</body>` tag, as shown in figure 8.3.

Figure 8.3. The Time to First Paint in Chrome for the Coyle Appliance Repair website with the `<script>` tags at the end of the document

In my testing, I was able to achieve an approximate Time to First Paint of 500 ms, which translates to roughly a 40% reduction overall. Well, in this example, anyway. As with most optimizations, your mileage will vary. The size and number of scripts as well as the length of the HTML document can play a role.

The good news about this approach is that it's consistent in nearly all browsers, so it's an easy fix that requires minimal effort. There are other things you can do with the `<script>` tag to influence how scripts load, such as the async attribute.

8.1.2. Working with asynchronous script loading

Modern browsers support a method for changing the loading behavior of external scripts. This change is via the `<script>` tag's async attribute. async (short for *asynchronous*) tells the browser to execute a script as soon as it loads, rather than loading each script in order and waiting to execute them in sequence. Figure 8.4 compares these behaviors.

Figure 8.4. A comparison of loading scripts with and without the user of the `async` attribute. The main difference is that scripts loaded with `async` won't wait for other scripts to finish loading before they execute.

`<script>` tags with the async attribute behave differently than those without it in that they'll execute immediately on download. They also won't block rendering while they download.

8.1.3. Using async

To use async, add it to `<script>` tags that you want to execute asynchronously. In this case, doing this will slash your client website's Time to First Paint by approximately 40%. Try it on the jquery.min.js and behaviors.js scripts, as shown in bold here:

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

Seems easy enough, right? You can reload the page and check whether things still work. Except that they don't. After reloading, you'll see a console error, as shown in <u>figure 8.5</u>.

Figure 8.5. The `async` attribute creates a problem in which behaviors.js fails because it executes before its dependency jquery.min.js is available.

Well, this is awful, isn't it? What's the point of async if it breaks stuff? There is a benefit in using async, but things get hairy when scripts are dependent on each other. When you use async with interdependent scripts, they enter into what's called a race condition, where two scripts can run out of sequence. In this example, a race condition occurs between jquery.min.js and behaviors.js. Because behaviors.js is much smaller than jquery.min.js, it'll always win the race and run first. Because behaviors.js is dependent on jquery.min.js, behaviors.js will always fail due to an unavailable jQuery object. This is because jquery.min.js hasn't loaded and executed before behaviors.js does. Figure 8.6 illustrates this race condition.

Figure 8.6. A race condition between jquery.min.js and behaviors.js always results in a failure, because behaviors.js loads and executes before its dependency is available.

This doesn't always occur. For example, if none of your scripts have dependencies, you can use async freely. It's when scripts have dependencies that things get tricky.

A way of getting around this is to combine your dependent scripts so that those dependencies are wrapped into a single asset. In this case, you can combine jquery.min.js and behaviors.js, in that order. From your command line, you can run this command to combine both scripts into scripts.js:

cat jquery.min.js behaviors.js > scripts.js

Because cat is available only on UNIX-like systems, Windows users will go this route:

type jquery.min.js behaviors.js > scripts.js

This command will finish quickly, and when it does, you get rid of both <script> tags in index.html and replace them with one <script> tag pointing to scripts.js:

<script src="js/scripts.js" async></script>

When you reload, you'll notice that the page works again, but you're probably thinking, "What's the benefit?" The benefit is quite noticeable. Figure 8.7 shows a further improved Time to First Paint after using async.

Figure 8.7. The Time to First Paint value in Chrome for the Coyle Appliance Repair website with scripts bundled and loaded using the `async` attribute

In my testing, async yielded an average Time to First Paint of roughly 300 ms with the scripts bundled, which outperforms placing the independent scripts without async at the bottom of the page by about 200 ms. There's a clear benefit in using async, but it doesn't end there. Without async, the DOM isn't available until about 1.4 seconds after the page begins to load. With async, this figure falls to 300 ms.

If you can manage your dependencies, it pays to use async. It's also highly supported, being available in all major browsers, even in IE10 and above. If async isn't supported, you can leave your <script> tags in the footer and they'll load in older browsers the normal way. Everyone wins! Except only sort of, which we'll discuss shortly.

8.1.4. Using async reliably with multiple scripts

You may take issue with bundling, but it's a good optimization practice for HTTP/1 servers and clients, because it can help alleviate the head-of-line blocking issue inherent to that protocol version.

HTTP/2 connections, however, benefit from assets being served in a more granular fashion as opposed to bundling them. More-granular resources make caching more effective. You can get away with this because HTTP/2 solves the head-of-line blocking problem by being able to serve more concurrent requests than HTTP/1. The particulars of this are explored in chapter 11. The point of this short section is to show you how to load scripts asynchronously while maintaining dependencies. To do this, you'll use a module loader called Alameda.

Alameda is an Asynchronous Module Definition (AMD) module loader written by Mozilla developer James Burke. Despite its capability of supporting complex projects with many dependencies, it's a small script, weighing in at only about 4.6 KB after minification and compression. As far as overhead goes, that's not adding too much to ensure that scripts load asynchronously while respecting their dependencies.

Using Alameda for this task is easy, and because it's part of the GitHub repository you downloaded at the beginning of this section, you can use it without tracking it down. The GitHub repository for Alameda is at https://github.com/requirejs/alameda. The first thing you should do is remove any <script> tags from index.html and add the following <script> right before the closing </body> tag:

<script src="js/alameda.min.js" data-main="js/behaviors" async></script>

Here you see three attributes:

- src loads the Alameda script.
- data-main includes the behaviors.js script. Alameda refers to scripts without their .js extensions, which is the syntax for AMD modules.
- async asynchronously loads Alameda, preventing blocking of page rendering.

It's not enough to slap this script on the page and assume everything's going to work. You need to open behaviors.js, add configuration code, and define your jQuery behaviors as an AMD module.

Listing 8.1. Configuring Alameda and defining behaviors.js as an AMD module

You're doing two things here: you define a configuration that tells Alameda where jquery.min.js lives, and then you wrap the behaviors.js script in a module definition that specifies jQuery as a dependency in the first argument. The dependent code in the second argument then runs when its dependencies are met.

When you reload the page with these changes and check the page's Time to First Paint, you'll notice that you're still hovering around the same mark as when you bundled scripts and used async. The big difference, though, is that you're keeping your scripts separate while still respecting the dependency of behaviors.js on jquery.min.js.

Alameda requires a modern browser!

Alameda is an update of RequireJS that requires functionality native to modern browsers to work, such as JavaScript promises. If you need support on a wider array of browsers, you can use RequireJS in place of Alameda. RequireJS and Alameda share a fully compatible API, so you can drop either in place of the other and they should work. Plus, RequireJS is only about 2 KB larger than Alameda when minified and gzipped. Check it out at http://requirejs.org.

Now that you know how to optimize script loading, let's look at alternatives to jQuery that provide a compatible API but offer less overhead and faster execution.

8.2. Using leaner jQuery-compatible alternatives

jQuery burst onto the scene years ago, during a time when accomplishing simple tasks such as selecting DOM elements and binding events required complex code to check for different methods available across browsers. Few methods were unified, and jQuery capitalized on this by providing a consistent API that worked regardless of the browser that used it.

Understandably, jQuery persists because of its utility and convenient syntax. But there are many reasons to consider alternatives that share portions of jQuery's API, but provide a smaller footprint and greater performance.

This section presents alternatives to jQuery. You'll compare their size and performance, and choose one of these options to use on the Coyle Appliance Repair website, as well as cover caveats of these alternatives.

8.2.1. Comparing the alternatives

Many JavaScript libraries are jQuery-compatible. *jQuery-compatible* doesn't mean that every single jQuery method is provided in these alternatives; it means that numerous methods present in jQuery are available in the alternative, and with the same syntax. The idea is that some measure of file size is traded off for less functionality. Some of these libraries also provide better performance.

8.2.2. Exploring the contenders

Here you'll compare three distinct jQuery-compatible libraries: Zepto, Shoestring, and Sprint. Here's a rundown of each:

- *Zepto* is described as a lightweight jQuery-compatible JavaScript library. Of all the jQuery alternatives, it's the most feature-rich out of the box and can be extended to do more. It's the most popular alternative in this field. Find out more at http://zeptojs.com.
- *Shoestring* is written by the Filament Group. It offers less capability than Zepto, but it provides most of the core DOM traversal and manipulation methods that jQuery does, as well as limited support for the $.ajax method. Find out more about Shoestring at https://github.com/filamentgroup/shoestring.
- *Sprint* is the lightest and most feature-bare alternative to jQuery, but it's high performing. Although it doesn't provide nearly as much functionality, it's great when you want to start out with very little, but are open to adding a more capable library when the need arises. Find out more about Sprint at https://github.com/bendc/sprint.

When comparing these libraries, you'll consider the size of each and the performance of equivalent methods.

8.2.3. Comparing file size

The biggest motivation for using these alternatives lies in the savings that they offer in file size. As you know from earlier in this book, the best thing you can do to decrease the load time of a website is to limit the amount of data that you send to the user. If you use jQuery in your web projects, these libraries are a perfect place to start cutting the fat. Figure 8.8 compares the file size of these libraries to jQuery. All file sizes assume minification and server compression.

Figure 8.8. A comparison of file sizes of jQuery and its alternatives

File size comparison of jQuery and compatible libaries
(file sizes refect minifcation and compression)

jQuery isn't huge, but its alternatives are far smaller. If you could lighten the load of your jQuery-dependent website by at least 20 KB, wouldn't you? Of course you would. The benefits don't stop there, though. Benefits also exist in performance.

8.2.4. Comparing performance

In this short section, you'll compare the execution times of common jQuery tasks: selecting elements by class name, toggling a class on an element with the addClass and removeClass methods, and toggling an attribute via the attr and removeAttr methods.

To measure performance in these cases, I selected a JavaScript library named Benchmark.js, which you can find out more about at https://benchmarkjs.com/. This library allows you to see the number of executions per second that a snippet of JavaScript is capable of.

I won't get into how Benchmark.js works under the hood. It's a highly accurate tool, and if you want to see how I wrote the test scripts, check out the GitHub repo for these tests at https://github.com/webopt/ch8-benchmark. I only want to show how the alternatives selected compare to jQuery for a handful of commonly used methods.

In the element selection test, you select an element of div.myDiv on the page by its class name. Figure 8.9 shows how the gallery fares with this simple task.

Figure 8.9. Performance of jQuery versus its alternatives when selecting an element by its class

Performance of selecting an element by its class
(higher is better)

Sprint is the definite winner here, with jQuery outperforming both Zepto and Shoestring. Figure 8.10 shows how everything stacks up when you toggle a class on an element.

Figure 8.10. Performance of jQuery versus its alternatives when toggling a class on an element

Performance of toggling a class on an element
(higher is better)

Things are more nuanced here. Sprint is still the clear winner. jQuery comes in third, losing to Shoestring, but still beats out Zepto. You can see how the gallery fares in figure 8.11 when you toggle attributes.

Figure 8.11. Performance of jQuery versus its alternative when toggling an attribute on an element

Performance of toggling an attribute on an element
(higher is better)

Again, Sprint dominates. Zepto loses, with jQuery in third and Shoestring in second. It should be noted that this is an arbitrary sampling of methods. Each method will compare differently, but some trends do persist. Sprint seems to be the fastest, but it's worth noting that Sprint's API isn't 100% compatible with jQuery's. Zepto covers most jQuery methods, and has plugins to further extend its capability. Although Sprint seems attractive performance-wise, it's not the easiest of the alternatives to retrofit into your jQuery-centric web project.

Now that you have a taste of what these libraries offer in weight and capability, let's go one step further by retrofitting the Coyle Appliance Repair website with one of these alternatives.

8.2.5. Implementing an alternative

Using a jQuery alternative is simple for sites that make light use of it: just drop the alternative in jQuery's place. In this short section, you'll do just that. Before you start, you may need to undo any changes you made earlier in the chapter. To do this, type git reset --hard.

8.2.6. Using Zepto

The alternative that you'll go with for Coyle is Zepto. Though not the highest-performing library of the alternatives, it has support for everything you need, and it's a little less than a third of the size of jQuery. You can take the site's payload down from 122 KB to 102 KB easily.

Another good reason to use Zepto in this case is that it represents the least amount of effort because it's the most compatible with jQuery, whereas libraries such as Shoestring and Sprint require refactoring to work. In environments where time is key (and when is that never true?), you can drop in Zepto in most cases with less effort than the alternatives.

Included in the repository's js folder is a copy of Zepto. To change out jQuery and replace it with Zepto, you need only to update the src attribute from js/jquery.min.js to this:

js/**zepto**.min.js

When you reload the page, you'll notice that there are no console errors, and all of the functionality on the page should be present, including the jQuery AJAX-driven form submission for the appointment-scheduling modal.

8.2.7. Understanding caveats on using Shoestring or Sprint

"That's it? Nothing else?" is likely what you're thinking. In this case, yes, that's pretty much it. Remember we picked Zepto because it's the most compatible with jQuery out of all the alternatives.

If you drop in Shoestring or Sprint, you'll need to refactor to make them work. The Coyle Appliance Repair website uses jQuery's $.ajax method to send an appointment-scheduling email to the site owner, and Shoestring's implementation of the $.ajax method isn't fully jQuery-compatible. Because Sprint has no $.ajax implementation, it can't make the cut.

It's not just the $.ajax method that can be problematic. jQuery alternatives don't support *everything* that jQuery does. Shoestring doesn't support the toggleClass method, but Sprint does. Sprint doesn't support the bind method, but Shoestring does. Many of these incompatibilities can be refactored by using workarounds or native JavaScript methods.

This is fine, though! The idea is that if you're beginning development of a new website with jQuery in mind, you should start with a minimalist library such as Sprint. If Sprint eventually fails to provide what you need, you can try Shoestring or Zepto. If you get to a point where those options are no longer cutting it, *then* you should go to jQuery.

If you start with minimalism in mind, you can ensure that you're keeping things as lean as possible. This mindset contributes to a faster site for your users. This isn't true of only jQuery, but also all aspects of web development. Always ask, "Do I *need* that hot new library for this site?" Chances are that this may lead you down a different path than what you initially intended.

In the next section, you'll go one step further and remove the need for jQuery and any alternatives altogether, and use native JavaScript to accomplish your goals. This is a significant undertaking if you're used to jQuery's methods, but it will allow you to eliminate all overhead associated with the library and to provide an even faster experience for your client.

8.3. Getting by without jQuery

jQuery and its alternatives are great, but many methods have been implemented (or are being implemented) into browsers that provide much of the same functionality. Tasks such as element selection and event binding that were once a burden to write for cross-browser compatibility now have a unified syntax, thanks to tstandardization efforts.

This section covers how to check for DOM readiness, select elements with query-Selector and querySelectorAll, bind events with addEventListener, manipulate classes on elements using classList, modify attributes and element contents with setAttribute and innerHTML, and use the Fetch API to make AJAX calls.

Want to skip ahead?

If you get stuck at any time doing the work in this section, you can skip ahead by typing git checkout -f native-js at the command line to see the finished work.

Before you start, you need to undo any work you've done on the client website in your local repository by entering git reset --hard at the command line. This reverts all local changes. You'll convert all of the code piece by piece, leaving jQuery in place until everything is replaced by native JavaScript. When done, you'll be able to remove the reference to jquery.min.js, load behaviors.js by using the async attribute, and subsequently benefit from faster load times. Let's open behaviors.js in your text editor and get to work!

8.3.1. Checking for the DOM to be ready

If you're familiar with jQuery, you know that you must check for the DOM to be ready before you can execute code. This isn't true of only jQuery, but also DOM-dependent scripts in general. You need to do this because the DOM doesn't fully load before the scripts run,

resulting in events not being bound to elements and critical behaviors not functioning. The following listing provides a truncated version of behaviors.js, showing how jQuery checks for the DOM to be ready.

Listing 8.2. jQuery checking for DOM readiness

```
$(function(){                                    ◁—— Checks for DOM readiness
    /* Content of behaviors.js truncated for brevity. */
});
```

In jQuery, anything encapsulated in $(function(){}); isn't executed until the document is loaded and ready. To achieve this in native JavaScript, you'll use addEvent-Listener (which you'll also use to bind other events later) to check for DOM readiness. The following listing shows this in action.

Listing 8.3. Checking for DOM readiness with addEventListener

```
document.addEventListener("readystatechange", function(){   ◁┐ Listens for
    /* Content of behaviors.js truncated for brevity. */        │ readystatechange,
};                                                              │ which waits for the
                                                               │ DOM to be ready.
```

That's it! The addEventListener method is available in IE9 and above, so this has a high level of compatibility.

Getting deeper support

If you need to support IE versions prior to 9, you can use the document.onreadystatechange method to monitor for DOM readiness. This method works in newer browsers as well.

Next, you'll investigate how to use the querySelector and querySelectorAll methods to select elements on a page, as well as how to take the addEventListener method further by binding events to these elements.

8.3.2. Selecting elements and binding events

The lion's share of jQuery's usefulness is in its ability to select elements and bind events to them. When it comes to selecting elements natively, the querySelector and querySelectorAll methods are the go-to solution. Like jQuery's core $ method, these two methods accept a CSS selector string as an argument. That string is used to return a node

in the DOM that you can work with. The difference between the two is that querySelector returns the first element that matches the expression, whereas querySelectorAll returns *all* elements that match.

Both methods have strong support across browsers, including IE9 and above, with partial support in IE8. This listing compares these two methods to their jQuery equivalents.

Listing 8.4. `querySelector` **and** `querySelectorAll` **vs. jQuery's core $ method**

```
/* Selecting one element. */
var element = document.querySelector("div.item");
var jqElement = $("div.item").eq(0);

/* Selecting a set of elements */
var elements = document.querySelectorAll("div.item");
var jqElements = $("div.item");
```

The first line in <ins>listing 8.4</ins> selects the first matching div.item element querySelector and the second line selects the first matching div.item element via jQuery. Line 3 selects all matching div.item elements via querySelectorAll and the final line selects all matching div.item elements via jQuery.

When element(s) are returned with either of these methods, you can then use the addEventListener method to attach events to those elements. The next listing shows a simple use of addEventListener to bind a click event on an item returned with querySelector.

Listing 8.5. Binding a click event on an item with `addEventListener`

```
document.querySelector("#schedule").addEventListener("click", function(){
    /* Code to execute on click. */
});
```

Using a combination of these methods will help you eliminate most of the jQuery-dependent code in behaviors.js. This code fires the appointment-scheduling modal.

Listing 8.6. jQuery-centric appointment scheduling modal launch code

```
// Scheduling modal open
$("#schedule").bind("click", function(){        ◁──┐  jQuery element selection
    $("body").addClass("locked");                   │  and click event binding
    openModal();
});
```

The part of the code you want to focus on here is the first line, which selects the appointment-scheduling button element (#schedule) and the bind method that binds a click event to that element. Using a combination of querySelector and add-EventListener, you can convert this to what you see next.

Listing 8.7. Appointment-scheduling modal event binding using native JavaScript

```
// Scheduling modal open
document.querySelector("#schedule").addEventListener("click", function(){    ◄─┐
    $("body").addClass("locked");
    openModal();                                        Native element selection
});                                                     and click event binding
```

When you reload, you'll still be able to trigger the modal by clicking the scheduling button. Although the code within the event handler is still driven by jQuery, you're getting closer to your goal of removing jQuery altogether.

In your text editor, switch out the remaining bind events to use addEventListener, as in listing 8.7. There should be three remaining calls to bind that you'll be able to replace.

Next, you'll replace calls to jQuery's addClass and removeClass methods with the native JavaScript classList method.

8.3.3. Using classList to manipulate classes on elements

The client website's JavaScript makes extensive use of jQuery's addClass and removeClass methods to add and remove classes. A native method called classList gives you this same functionality. This listing shows this method compared to its jQuery counterparts.

Listing 8.8. `classList` **vs. jQuery's** `removeClass` **and** `addClass` **methods**

```
                                  jQuery's addClass method              Adding a class with
                                                                        classList's add method
            /* Adding Classes */
            $(".modal").addClass("show");   ◄──┐
            document.querySelector(".modal").classList.add("show");        ◄──
  jQuery's
 removeClass                                                  Removing a class with
   method   ┌─► /* Removing Classes */                        classList's remove method
            └─► $(".modal").removeClass("show");
            document.querySelector(".modal").classList.remove("show");     ◄──

                                                             Toggling a class with
            /* Toggling Classes */                           classList's toggle method
  jQuery's  ┌─► $(".modal").toggleClass("show")
 toggleClass
   method   │  document.querySelector(".modal").classList.toggle("show");  ◄──
```

Although your client's website doesn't use the toggleClass method, it's important to note that classList has a toggle method. Unfortunately, this method isn't supported well in IE. Support for the classList method otherwise is good overall, with IE10 and above supporting it. This listing shows the openModal function that opens the scheduling modal.

Listing 8.9. The jQuery-dependent `openModal` function

```
function openModal(){
    window.scroll(0, 0);
    $(".pageFade").removeClass("hide");        <-- Removing hide class on the .pageFade element
    $(".modal").addClass("open");              <-- Adding open class on the .modal element
}
```

Achieving the same result with the classList method is a bit more involved. You need to convert the jQuery element selection code to use the querySelector method instead. This listing shows how to transform the code from listing 8.9 into fully jQuery-independent code.

Listing 8.10. The jQuery-independent `openModal` function

```
function openModal(){
    window.scroll(0, 0);
    document.querySelector(".pageFade").classList.remove("hide");   <-- Removing the hide class on the .pageFade element
    document.querySelector(".modal").classList.add("open");         <-- Adding the open class on the .modal element
}
```

Next, you need to search for all uses of the removeClass and addClass methods in behaviors.js, and update them to use classList. When you do, be sure to update the jQuery $ selection methods to use the querySelector method.

What if classList isn't supported?

It's possible that you may need to support IE9 or below. If this is the case, you can use the className property instead. This property doesn't have methods for adding, removing, or toggling classes. Instead, it's a string that you can use to assign whatever classes you need to the targeted element—not as convenient as classList, but it works in a pinch.

After you've changed all the code to use the classList method, it's time to replace the jQuery attribute and content modification methods with their native JavaScript counterparts.

8.3.4. Reading and modifying element attributes and content

Another piece of functionality that the client's website relies on jQuery to accomplish is the reading and modifying of attributes of elements, as well as modifying element contents. These behaviors can be replaced by native JavaScript methods easily. The next listing compares jQuery's attribute manipulation methods to their native JavaScript equivalents.

Listing 8.11. Modifying attributes with jQuery vs. native JavaScript

You'll also need to be able to know how to read and/or modify the contents of an element, because the client's website also uses this method. Here's how jQuery reads the contents of an element as compared to JavaScript's innerHTML property.

Listing 8.12. jQuery's `html` method vs. JavaScript's `innerHTML` property

The syntax between jQuery's html method and the native innerHTML property is different, in that the html method is a function, whereas innerHTML is a property that you assign a value to.

There aren't a whole lot of places in the client website's JavaScript that need to modify attributes or set content on elements, but it does happen during a key moment: when the user submits an appointment request, and the confirmation modal appears. This is in a success callback in a jQuery ajax call.

Listing 8.13. Attribute and element content modification via jQuery

```
                                                    ┌─ Message text is placed
                                                    │  into the status text area.
success: function(data){
    $("#status").html(data.message);          ◄────┘
    document.querySelector(".statusModal").classList.add("show");
    document.querySelector(".modal").classList.remove("open");

    if(data.status === true){
        $("#okayButton").attr("data-status", "success");
        $("#headerStatus").html("Thank You!");          ◄──┐
    }                                                      │  Header is updated
    else{                                                  │  to reflect success
        $("#okayButton").attr("data-status", "failure");  │  or failure.
        $("#headerStatus").html("Error");               ◄─┘
    }
}
```

data-status attribute is set on okayButton.

In the listing, the message text from the appointment-scheduling emailer is placed into the status text area. The data-status attribute is set on the okay button, which is used to determine what the button does in the context of success or failure. The header of the status modal is updated to reflect the success or failure of appointment submission.

With some modifications, you can take what you've learned so far and turn this into something like this listing, which runs without jQuery.

Listing 8.14. Attribute and element content modification via native JavaScript

```
success: function(data){
    document.querySelector("#status").innerHTML = data.message;
    document.querySelector(".statusModal").classList.add("show");
    document.querySelector(".modal").classList.remove("open");

    if(data.status === true){
        document.querySelector("#okayButton")
            .setAttribute("data-status", "success");     ◄──┐
        document.querySelector("#headerStatus")            │  Sets status of
            .innerHTML = "Thank You!";                     │  scheduling
    }                                                      │  operation on
    else{                                                  │  the data-
        document.querySelector("#okayButton")              │  status
            .setAttribute("data-status", "failure");     ◄─┤  attribute.
        document.querySelector("#headerStatus")            │
            .innerHTML = "Error";                          ┘
    }
}
```

Assigns message and status text from appointment scheduler.

You need to update one more spot that uses jQuery's attr method to read an attribute. This occurs when the user clicks the okay button in the status modal. Here are the relevant lines of this code.

Listing 8.15. Getting an attribute via jQuery's `attr` method

This spot is a little tricky because jQuery's $(this) object is used inside the click binding's code to refer to the #okayButton element. When you convert this to the addEvent-Listener syntax that you used earlier, this code will break. You need to use the event object (assigned to e in the function call) to replace the reference to the $(this) object, and use the getAttribute method to retrieve the value of the data-status attribute instead. Here is a working replacement of both methods.

Listing 8.16. Getting an attribute via the `getAttribute` method

```
document.querySelector("#okayButton").addEventListener("click", function(e){
    if(e.target.getAttribute("data-status") === "failure"){
```

The top line is the converted click binding with the event object in the function call. In the last line, the e.target attribute refers to the element that the click binding was attached to, with the getAttribute method retrieving the data-status attribute's value.

In the absence of jQuery's $(this) object, you can use the event object's target method to refer to the element that the event was bound to inside the event code itself. It's weird to get used to if you've been used to jQuery, but it'll quickly become second nature.

Before you can wrap things up and remove jQuery from the project, you'll replace the last bit of jQuery-dependent functionality left, which is the $.ajax call used to send an AJAX request to the server to schedule an appointment. You'll replace the jQuery AJAX functionality with a native JavaScript version called the Fetch API.

8.3.5. Making AJAX requests with the Fetch API

In the old days of AJAX requests, you had to use the XMLHttpRequest request object. It was an unwieldy way of making AJAX requests, and different browsers required different approaches. jQuery made AJAX requests a much more convenient task by wrapping its own AJAX functionality around the XMLHttpRequest object. It still works great to this day, but some browsers have implemented a native resource-fetching API called the Fetch API.

8.3.6. Using the Fetch API

The most basic use of the Fetch API is for a GET request of a resource. A good example is interacting with an API that gives access to a database of movies and returns JSON data. This shows such a request using fetch.

Listing 8.17. Fetch API–driven AJAX request with a JSON response

```
fetch("https://api.moviemaniac.com/movies/the-burbs")
    .then(function(response) {
    return response.json();
}).then(function(data) {
    console.log(data);
});
```

In the listing, the fetch method takes a minimum of one argument, which is the URL to the resource. On success, a promise is returned that allows you to work with the JSON data. The raw response object has a json method that you can return to the next promise in the chain. Another promise is returned with the encoded JSON data. The console.log (data); line outputs the data from the response to the console.

This is only a basic use of the Fetch API. The client's website uses the jQuery $.ajax method to send form data using a POST request. Accomplishing this takes a bit more work but uses a bit less code than if you used jQuery's $.ajax function.

To be fair, you've been submitting the form to a mock location that returns a JSON response for illustrative purposes. If you try this approach in your own websites with a back-end script, you'll find that it should work fine. This listing shows the Fetch API at work in place of jQuery.

Listing 8.18. Fetch API–driven AJAX request

When this code runs, test the appointment-scheduling modal and you should see that it works fine. Not bad for a native API, and it's much more attractive than the usual XMLHttpRequest tango that we've done in years past.

None of this is meant to pick on jQuery's $.ajax API. It's an awesome wrapper around XMLHttpRequest, but as browsers pick up more support for the Fetch API, it makes sense to abandon $.ajax. Of course, if you're going to rely on fetch, you need to be able to polyfill those browsers that don't support it yet.

8.3.7. Polyfilling the Fetch API

As could be expected, not every browser supports the Fetch API. At this point, you have a few options:

- You can avoid using fetch altogether and use a standalone implementation of jQuery's $.ajax API, such as this one at https://github.com/ForbesLindesay/ajax.
- You can sniff for the fetch method in the window object. If the method is found, you can use fetch. If not, you can use the standard XMLHttpRequest object. Or use the XMLHttpRequest object no matter what, because it's well supported (albeit a pain to use).
- You can sniff for the fetch method, and if not found, asynchronously load a polyfill.

In this short section, you'll opt for the third method because it's the most optimal. Browsers that support the Fetch API will rely on a native browser method without the overhead

of an external script. Browsers that don't will incur the overhead of the polyfill, but the syntax will be unified.

A decently robust polyfill of the Fetch API can be found at https://github.com/github/fetch. For the sake of simplicity, a minified version of this script is bundled in the repo for the client's website as fetch.min.js in the js folder.

Using a familiar approach to loading polyfills in prior chapters, you can load this script conditionally based on the presence of the fetch object. You'll do this by placing an inline <script> at the bottom of index.html before the closing </body> tag.

Listing 8.19. Conditionally loading the Fetch API polyfill

You might be thinking about dependencies, because behaviors.js depends on fetch. The key here is that the call to fetch isn't executed on page load. Plenty of time is afforded for the polyfill to load before the user has a chance to open the scheduling modal, fill out the form, and hit Submit. It's a soft sort of dependency in that time and logistics work in your favor, and you can see how this plays out in figure 8.12.

Figure 8.12. The loading of the fetch API polyfill and its timing with the user's intentions to fire the scheduling modal

Test this approach in a browser that doesn't support the Fetch API, such as IE, and see how it works. You'll be able to tell whether fetch.min.js has been loaded by examining the network requests for that browser.

With all jQuery methods firmly replaced with native JavaScript, you can now remove the reference to jquery.min.js, and use the async attribute to asynchronously load behaviors.js. Now that your client's website is running optimally, you can move onto learning about animating elements with JavaScript by using requestAnimationFrame.

8.4. Animating with requestAnimationFrame

Animation in the earlier days of JavaScript was less perfect than it is now. You'd usually have to use a timer function such as setTimeout or setInterval in order to achieve the effect. As time has passed, and the capability of browsers has increased, we have a newer and higher-performing method that helps us animate elements with JavaScript.

This section discusses traditional timer-based animations and how to use requestAnimationFrame in their place. From there, you'll see how requestAnimationFrame compares in performance to its timer-based ancestors and CSS transitions, and then put the method to work on the Coyle Appliance Repair website.

8.4.1. requestAnimationFrame at a glance

Animation is different when done in JavaScript as compared to CSS. In CSS, you'd apply a transition property to an element that tells the browser that a specific property (or properties) will change. When that property changes, the browser animates the transition between the start and end points. The underlying logic that runs the animation is all handled by the browser. With JavaScript, you have to perform this work yourself. Let's take a look at how animation has traditionally been achieved in JavaScript and compare that to requestAnimationFrame.

8.4.2. Timer function-driven animations and requestAnimationFrame

When animating in JavaScript, you're changing an element's appearance or position on a screen through the element's style object via a timer function to give the appearance of motion. In the days of old, so to speak, setTimeout and setInterval were the timer functions used to animate elements on an interval, usually by an interval of 1000 ms / 60, which aims to animate effects at roughly 60 frames per second. Typical code for this kind of animation looks something similar to this.

Listing 8.20. Animating with a timer function (`setTimeout`)

```
function draw(){
    document.querySelector(".item").style.width =
        (parseInt(document.style.width) + 2)) + "px";
    setTimeout(draw, 1000 / 60);
}

draw();
```

Updates the left property of the element in 2 pixel increments.

Recursively runs the draw function at 60 FPS.

Drawing is triggered with the initial function call.

Timers themselves aren't expensive, but they're not optimal for animation code. To solve this problem, the requestAnimationFrame method was developed. Using this method is similar to the code in listing 8.20, and is shown here.

Listing 8.21. Animating with `requestAnimationFrame`

```
function draw(){
    document.querySelector(".item").style.width =
        (parseInt(document.style.width) + 2)) + "px";
    requestAnimationFrame(draw);
}

draw();
```

Runs a specific function, but in an unspecified interval.

At first glance, this code seems somewhat lacking, because unlike setTimeout, requestAnimationFrame doesn't allow the user to specify an interval in milliseconds. So how does it even work, then? Simple: The interval is handled within request-AnimationFrame, and it acts according to the refresh rate of the display, which tends to be 60 Hz on most devices; requestAnimationFrame aims for 60 FPS on a typical device. If a device has a different refresh rate, requestAnimationFrame will animate accordingly.

Admittedly, this example has limitations in that it animates the element's left property for an infinite amount of time, but it illustrates the concept. In a bit, I'll show a realistic implementation of this on the client's website, but not before you take a look at the performance of requestAnimationFrame versus its traditional timer-based animations and CSS transitions.

8.4.3. Comparing performance

Earlier I said that requestAnimationFrame boasts better performance than its timer-based ancestors setTimeout and setInterval. To test these methods, I wrote a simple animation for each. The animation is of a box that travels a distance of 256 pixels from left to right,

while doubling in width and height and shifting to 50% opacity. You can try each of these tests for yourself at http://jlwagner.net/webopt/ch08-animation if you're so inclined. Figure 8.13 compares the performance of these two methods using setTimeout, requestAnimationFrame, and CSS transitions as profiled in Google Chrome's timeline profiler.

Figure 8.13. Normalized performance of various animation methods in Chrome's Timeline tool

One thing to remember about both setTimeout and requestAnimationFrame is that because they're dependent on JavaScript, they require more scripting time than CSS transitions, and this is normal. But requestAnimationFrame spends less time both painting and rendering than either of the other two methods.

Now that you know how to use requestAnimationFrame and how it performs, let's fire up the Coyle Appliance Repair website again and animate the scheduling modal.

8.4.4. Implementing requestAnimationFrame

You've had a little bit of downtime since the Coyle Appliance Repair website launched, and so now it might be fun to experiment with requestAnimationFrame. On Coyle, the only animation that occurs is when the scheduling modal is opened. This used a CSS transition in the past, but now you're experimenting with a timer-based animation that you want to transition over to use requestAnimationFrame. You'll need to grab the latest code for that, so type in this command to switch over to a new branch:

```
git checkout -f requestanimationframe
```

I've written a flexible function for the client's website to test setTimeout and requestAnimationFrame for animating the modal in behaviors.js.

Listing 8.22. Animation function using `setTimeout`

Though not as complete as something like jQuery's animate function, this is a much more flexible implementation than what was shown in listing 8.20.

In the invocation illustrated in figure 8.14, you're telling animate to select the .modal element and animate its top property from -150% to 10% over a duration of 500 milliseconds. If you click the Schedule an Appointment button and launch the modal, you'll see that it opens fine with your JavaScript animation code. What kind of work is involved in turning this function over to use requestAnimationFrame, though? Surprisingly little. The next listing shows the animate function's draw method modified to use requestAnimationFrame, with modifications in bold.

Figure 8.14. The `animate` function in use, with arguments labeled

Listing 8.23. Substituting `requestAnimationFrame` in place of `setTimeout`

```
draw = function(){
    if(endTime > +new Date()){
        element.style[property] = progress() + units;
        requestAnimationFrame(draw);
    }
    else{
        element.style[property] = to + units;
        return;
    }
};
```

That's it, for the most part. You remove the call to setTimeout and replace it with a call to requestAnimationFrame. Everything should work as before, only with a higher-performing animation method. If you want to do a little cleanup, you can also remove the interval variable, because requestAnimationFrame doesn't need it.

requestAnimationFrame isn't universally supported, so what can you do to ensure better support? For one, you can create a placeholder that allows you to use requestAnimationFrame first but then fall back to setTimeout when it no longer exists. The next listing shows how to do just that.

Listing 8.24. `requestAnimationFrame` **fallback using** `setTimeout`

If you use this approach, you'll need to update your code to use the custom raf method in place of the requestAnimationFrame method, but this will give your application's animation methods the broadest possible support. With this approach, you get the benefits of requestAnimationFrame when it's available, and you fall back to setTimeout when it's unavailable. Not too shabby.

Next, you'll briefly cover how to use Velocity.js, a simple requestAnimationFrame-driven JavaScript animation library.

8.4.5. Dropping in Velocity.js

This foray into requestAnimationFrame may leave you with more questions than answers. It can be challenging to use this method in place of CSS transitions or jQuery animations, especially if requirements are complex. This short section briefly introduces Velocity.js, which makes animation as convenient as jQuery's animate method.

Velocity.js is an animation library that uses an API similar to jQuery's animate method. You can learn more about it at http://velocityjs.org. The best part about Velocity is that it's jQuery-independent. But if you have a project using jQuery that relies heavily on its animate method, dropping in Velocity.js makes the process of animating the same as with jQuery. For example, consider this jQuery animation code:

```
$(".item").animate({
    opacity: 1,
    left: 8px
}, 500);
```

You can port this animation code to use Velocity.js, like so (changes in bold):

```
$(".item").velocity({
    opacity: 1,
    left: 8px
}, 500);
```

This simple change from animate to velocity will use the Velocity animation engine instead of jQuery's, which gives you silky smooth requestAnimationFrame-powered performance, as well as easing functions to give your animations a sense of natural movement. Unlike jQuery's animate method, it allows you to animate colors, transforms, and scrolling.

If you use Velocity.js *without* jQuery, the syntax does change somewhat. As Velocity loads, it'll check whether jQuery is loaded. If jQuery isn't present, the syntax for animating the same element as shown in the examples changes to the following:

```
Velocity(document.querySelector(".item"), {
    opacity: 1,
    left: 8px
}, {
    duration: 500
});
```

Aside from semantics, this syntax doesn't differ all that much. With or without jQuery, Velocity.js can make your JavaScript-driven animations much more fluid and efficient, without getting into the weeds of writing your own animation code.

Be warned that this library is about 13 KB minified and compressed, so consider using it only if animation features prominently on your website and performance is paramount. Adding 13 KB of overhead to a site that doesn't feature much in the way of animation will contribute to a suboptimal experience for your users, and may be better served by writing your own animation code or using CSS transitions instead.

8.5. Summary

You learned many concepts in this chapter about how to keep your JavaScript lean and fast:

- Depending on its position, the $\langle script \rangle$ tag can block rendering, which delays the display of the page in the browser. Placing $\langle script \rangle$ tags toward the bottom of the document can speed up the rendering of a page.
- The async attribute can provide further performance benefits if you can manage the execution of scripts that use it.
- Managing the execution of interdependent scripts that use async can be challenging. A third-party script-loading library such as Alameda or RequireJS can provide a convenient interface for managing script dependencies, while also providing the benefit of asynchronous script loading and execution.
- Although jQuery is useful, it has a relatively large footprint. A portion of its functionality can be better served by jQuery-compatible alternatives that are smaller in file size, and in some cases, better performing.
- Browsers are providing more jQuery-like functionality as time goes on. You can select elements with querySelector and querySelectorAll, and bind events to them by using addEventListener. You can also manipulate element classes by using classList, get and set attributes on them by using getAttribute and setAttribute, and modify their contents by using innerHTML. For a cheat sheet of jQuery methods and their native browser equivalents, check out appendix B.

- The Fetch API provides a convenient native interface for requesting remote resources via AJAX. It can also be effectively polyfilled for browsers that don't support it.
- The `requestAnimationFrame` API is a newer JavaScript function that you can animate in lieu of `setTimeout` or `setInterval`. It's higher-performing than those older timer-based methods, and renders and paints faster than CSS transitions.

In the next chapter, you'll learn about service workers in JavaScript. You'll see how to use them to serve offline experiences to users with limited or no internet connectivity, and to improve the performance of your site.

Chapter 9. Boosting performance with service workers

`This chapter covers`

- Understanding what service workers are and what they allow you to do
- Installing a service worker on a simple site
- Caching network requests inside a service worker
- Updating a service worker

As the web has matured, so too has the technology that it relies on. No longer are we tied to our desks while browsing the web. With the advent of mobile devices, people are accessing content on Wi-Fi and data networks of varying degrees of quality and reliability. This introduces challenges in the way we access content, particularly in the case of poor or absent internet connections that may leave the user in a lurch.

Sometimes we go offline—in an airplane without Wi-Fi, or passing through a tunnel in a car or train, for example. It's a fact of life. When this happens, we're somewhat used to being unable to view content on websites. But it doesn't have to be this way, and this is where service workers enter the picture.

In this chapter, you'll learn about service workers: how they work, how to use them to intercept network requests, and how they can be used to cache site assets for times when your device is offline. Beyond the mere convenience of providing an offline experience to your users, you'll also learn of the performance benefits that can come with using service workers, which can make repeat visits to your website even faster than before.

As with any code you write, sometimes changes are necessary. Changing a service worker isn't as straightforward as changing other components of a website, so we'll cover what you need to do when you make changes to one. Let's get started learning about service workers!

9.1. What are service workers?

Service workers are a kind of worker—an evolving standard of scripts that operate in a separate and special scope from ordinary scripts. Workers perform tasks in the background, on a separate processing thread than typical JavaScript code that you'd write and reference with <script> tags. Figure 9.1 shows a service worker operating on its own thread.

Figure 9.1. A service worker operating on its own thread labeled ServiceWorker Thread can be seen at the bottom in this view of Chrome's Timeline tool.

Because service workers operate on a separate thread, they behave differently than JavaScript loaded via the <script> tag. Service workers don't have direct access to the window object on the owner page. Although they *can* communicate with the parent page, they must do so indirectly through an intermediary, such as the postMessage API.

The problems that workers solve depend on the kind of worker we're talking about. Web workers, for example, allow the browser to perform CPU-intensive tasks without slowing or halting the browser's UI. Service workers, which we cover specifically in this chapter, allow the user to intercept network requests and conditionally store items in a special cache via the CacheStorage API. This cache is separate from the native browser cache, and by using it, we can serve content to the user from a CacheStorage cache when they're offline. We can also use this special cache to boost the rendering performance of a page.

A theoretical example of a service worker in use is on a popular blog. If the page caches articles via CacheStorage as the user reads them, they'd be available for offline viewing in the event that a user somehow loses connectivity. This could be useful in various situations, such as when a cellular or Wi-Fi connection is weak, or when a network connection isn't available.

Service workers can help us deal with this problem, not by overcoming the problem of poor or absent connectivity, but by presenting cached content that the user has already seen so that they have *something* to look at, rather than nothing at all. It doesn't fix the inability to access updated content, but rather solves the problem of a broken web-browsing experience.

The service worker interface itself is light, and consists of events that are triggered in specific instances, such as when a service worker is installed, or when a network request is made. These events are listened for with the addEventListener method that you learned

about in chapter 8. The workhorse of the service worker you'll write in this chapter is the fetch event. This is the event you'll use to intercept network requests, and store or request items from a CacheStorage cache. Figure 9.2 illustrates this process.

Figure 9.2. A service worker communicating as a proxy between a user and a web server. The user makes requests, which the service worker can intercept. Depending on how the service worker code is written, assets can be retrieved from the service worker's CacheStorage cache, or passed through to the web server. The service worker can also write to the cache in specific instances.

With this event-driven interface, service workers can help create offline experiences for when an internet connection is poor or absent altogether. This is done through an interface that intercepts network requests, and reads from or writes to a CacheStorage cache. You'll use CacheStorage when you write your first service worker in the next section.

Now that you have a little background on what service workers do at a higher level, there's no time like the present to dive in and learn how to use them. In the next section, you'll do exactly that.

9.2. Writing your first service worker

In this section, you'll embark on writing your first service worker. First, you'll learn how to check whether a browser supports service workers, and if so, go about installing one. Then, you'll write the guts of your service worker and use it to intercept network requests. Finally, you'll measure the performance benefits that your service worker affords.

The project you'll write a service worker for is a static version of my blog. I've been trying to find new ways to squeeze more performance out of my site, while also allowing readers to read old content if they're offline. First, you'll grab a copy of the code from GitHub and get it running on your computer. To do this, enter the following commands:

```
git clone https://github.com/webopt/ch9-service-workers.git
cd ch9-service-workers
npm install
node http.js
```

Service workers require HTTPS!

For convenience, service workers can run on localhost without HTTPS. But because of the level of access that service workers have in terms of being able to intercept network requests and run in the background, HTTPS is required on a production web server. You have some leeway on localhost, but when you go to production, you'll *need* a valid SSL certificate.

When finished, the site will be running on your local machine at http://localhost:8080. After you verify that the site is running, you'll be ready to write your first service worker!

9.2.1. Installing the service worker

The installation process of a service worker requires little code. You need to check whether the browser supports service workers at all. If the browser supports them, you can continue with the installation. If the browser *doesn't* support service workers, nothing will happen. This ensures that your site will continue to function, even if the user's browser isn't capable of using service workers. Figure 9.3 illustrates this behavior flow.

Figure 9.3. The service worker installation process. The code checks for the status of service worker support. If the browser supports it, the service worker is installed. If not, the browser does nothing.

The first part of installing your service worker involves registering it via the sw-install.js script referenced via a <script> tag in the footer of each page.

9.2.2. Registering the service worker

To get started with installing your service worker, you'll see a file named sw-install.js in the htdocs folder. Open this file in your text editor and enter the contents of this listing into it.

Listing 9.1. Service worker support detection and installation code

```
if("serviceWorker" in navigator){
    navigator.serviceWorker.register("/sw.js");
}
```

Sniffing out service worker support is easy. In the first line, we use the in operator to check for the existence of the serviceWorker object within the navigator object. If service workers are supported, the script in /sw.js is registered via the serviceWorker object's register method as shown in the second line.

A note on service worker scope

If you're curious as to why the service worker code isn't in the js directory, it's because of scoping. By default, a service worker is scoped to work only in the directory it resides in and its subdirectories. If you want it to work across the entire site, you need to place it in the site's root folder, which is what you'll do in this chapter's example. If you want to place your service worker in a more logical location, you can overcome this issue by setting the Service-Worker-Allowed HTTP response header to a value of /, which allows the service worker to work across the entire domain.

Don't reload the page and test your changes just yet! For your service worker to *do* anything, you need to write some of your service worker behavior, particularly what should occur when the service worker is first installed.

Writing the service worker's install event

As I said earlier in this chapter, the service worker interface is light and consists of events that you can attach code to by using the addEventListener method. When a service worker is first installed, the install event is fired.

When you install your first service worker, you want to immediately cache the global assets for the site. These are items such as the site's CSS, JavaScript, images, and any other asset that's common across all pages and devices. Figure 9.4 depicts this caching process.

Figure 9.4. The behavior that you want to occur when the service worker's install event is fired

To get started writing the installation behavior of your service worker, open sw.js in the htdocs folder. The first thing you want to do is cache the page assets that are necessary for the site to run offline. These are usually the static pieces, such as CSS, JavaScript, and images. The following listing shows how to accomplish this important step.

Listing 9.2. Caching assets in the service worker's `install` **event**

```
                                              Version identifier of asset cache.
    var cacheVersion = "v1",
        cachedAssets = [
            "/css/global.css",
            "/js/debounce.js",
            "/js/nav.js",                        URIs of assets you
            "/js/attach-nav.js",                 want to cache when
        "/img/global/jeremy.svg",                service worker is
            "/img/global/icon-github.svg",       installed
  Fires when    "/img/global/icon-email.svg",
  service       "/img/global/icon-twitter.svg",
  worker is     "/img/global/icon-linked-in.svg"    New cache is opened with the
  being     ];                                       identifier name set in the
  installed.                                          cacheVersion variable.
      self.addEventListener("install", function(event){
          event.waitUntil(caches.open(cacheVersion).then(function(cache){

          return cache.addAll(cachedAssets);                    Cache is populated
      }).then(function(){                                        with cachedAssets
          return self.skipWaiting();      Tells service worker to  array.
      }));                                immediately take effect
    });                                   and fire activate event.

    self.addEventListener("activate", function(event){          Fires when the service
        return self.clients.claim();                            worker's skipWaiting
    });                                                         method fires.
                  When the activate event fires, the service
                  worker will begin working immediately.
```

The installation code is a little tricky at first glance, but easy to grasp once you walk through it. First, you define a cache identifier in the cacheVersion string. This allows you to give your cache a name, and you can update it when the cache is changed in future versions of the service worker. The assets you want to cache up front in the service worker are then specified in the cachedAssets array.

Next, you write the install event code, which is executed as soon as the service worker is installed by sw-install.js. Here, you return a promise for opening a new caches object by the identifier you've set in the cacheVersion variable, and then add all of your assets specified in the cachedAssets array to it. The promise is chained with a then call that returns the result of the service worker's skipWaiting method. This instructs the service worker to immediately fire the activate event after the install event is complete. The activate event code then executes the service worker's claim method, which allows the service worker to begin working immediately.

With this code in place, *now* you can reload the page. When the page reloads, it seems like nothing has happened. So how can you tell whether the service worker is, well, working? You can verify in Chrome by opening the Developer Tools and navigating to the Application tab. Click the Service Workers item in the left pane, and you'll see something similar to figure 9.5.

Figure 9.5. The Application tab in Chrome's Developer Tools showing active service workers for the current site. Click the Service Workers item in the left pane to access this panel.

When you open this panel in the Application tab, you'll be able to view the service workers currently running on the page, as well as perform actions such as stop a service worker, unregister it, and more important, force it to update on reload. During your work in this chapter, you should select the Update on Reload check box, which forces updates on page reload. Service workers can be tricky to work with as you develop them, and selecting this option simplifies the process.

Looking at the service worker cache

Now that you've verified that the service worker is installed, how can you tell whether the assets you've specified have been cached? The answer to this burning question is once again in the Application tab. In the left pane, expand the Cache Storage item and click the cache for your site, which is labeled v1, as specified in the install event code. You can see this in <u>figure 9.6</u>.

Figure 9.6. The v1 cache created by your service worker. You can see that the assets you've specified in the service worker's `cachedAssets` array are present.

Selected cache   Cached assets

When you look at your v1 cache under the Cache Storage item in the Application tab, you can see all of the items that you've specified in the cachedAssets array. With these in your cache, let's see what happens if you go offline. To go offline, you could turn off your network connection on your machine by turning off Wi-Fi or unplugging your network cable, but there's an easier way. In Chrome's Developer Tools, go to the Network panel and locate the Offline check box, shown in [figure 9.7]().

Figure 9.7. Selecting the Offline check box in Chrome's Network panel allows you to simulate what it's like to be offline without having to disable your network connection.

Select the Offline check box and reload the page. You'll notice that, although you've cached all the necessary page assets for offline viewing, you still get a connection error and not the offline version of the site. Why is that?

In this case, it's because you haven't cached the HTML document itself. Even if you *did* do that, though, you'd still need a mechanism that intercepts network requests, and then you'd need to figure out what to *do* with them. Indiscriminately adding every asset to the CacheStorage cache up front *isn't* a viable strategy, because it loads a ton of stuff that the user may end up never needing. You *first* cache global assets common on all pages up

front in the install event, and *then* use the fetch event to intercept and cache assets on an as-needed basis. This ensures that your visitors are efficiently caching everything you know they'll need up front, and then you can programmatically add assets to the cache after they request them.

9.2.3. Intercepting and caching network requests

To control what happens when you're offline, you need a mechanism that sits between you and the server that allows you to cache content for offline viewing. The fetch event allows this functionality via a behavior flow defined in <u>figure 9.8</u>.

Figure 9.8. The behavior of the service worker's `fetch` event. The user makes a request for an asset, and the service worker steps in to intercept it to see whether the asset is already in the cache. If not, the asset is fetched from the network, and the service worker caches it. If it's in the cache, it's pulled from the cache.

You might be wondering why you bothered to cache assets during the install event. You're priming the cache up front with assets that you *know* you'll need. Whether or not you've cached an asset, however, you *still* need to define behavior for a fetch event that deals with the user's requests and caches assets for use later on. For assets that you've cached up front, the service worker will serve them from that cache. For assets that you're less certain about, such as HTML documents, images specific to articles, fonts, and so on, you

want to subject them to a more rigorous check: a check that goes and fetches them from the network, and *then* places them into the cache for later use.

One good reason for this is that the assets you request may not be consistent across all devices, and thus shouldn't be added to the cache up front. A device with a high-density display will download images appropriate for *that* device, and should be cached programmatically according to the device's needs. A less capable device should download and cache assets appropriate to its own limitations.

You need to write your own logic to accomplish this goal. This logic is written in this listing. Add it to your local copy of sw.js.

Listing 9.3. Intercepting and caching additional assets in the `fetch` event

A regular expression of assets you're blacklisting from the service worker cache.

The fetch event listener that lets you intercept network requests.

A regular expression of hosts you'll intercept requests for.

```
self.addEventListener("fetch", function(event){
    var allowedHosts =
        /(localhost|fonts\.googleapis\.com|fonts\.gstatic\.com)/i,
        deniedAssets = /(sw\.js|sw-install\.js)$/i;

    if(allowedHosts.test(event.request.url) === true &&
        deniedAssets.test(event.request.url) === false){
        event.respondWith(
```

Before responding to a request, check that the request URL is from an accepted host, and isn't one of the blacklisted assets.

The event object's respondWith method intercepts the request. Bypassing this allows default browser behavior to occur.

If the item isn't in the service worker cache, use the fetch API to retrieve it.

If the item is in the service worker cache, serve the cached response.

Check if the item is in the service worker cache.

```
            caches.match(event.request).then(function(cachedResponse){
                return cachedResponse ||
                    fetch(event.request).then(function(fetchedResponse){
                        caches.open(cacheVersion).then(function(cache){
                            cache.put(event.request, fetchedResponse.clone());
                        });

                        return fetchedResponse;
                    });
            })
        );
    }
});
```

When the asset is fetched, open the service worker cache with the cacheVersion identifier.

The fetched asset is placed into the service worker cache.

After caching the response, return the original response.

With this code, you can fetch items from the cache that you've primed in the service worker's install event code. If you come across a request for an asset that isn't in the cache, you use the fetch method (covered in chapter 8) to retrieve it from the network. After the asset is downloaded, you place it in the service worker cache. Then that asset

is retrieved from there instead of from the network. Force a reload of the page by using Ctrl-Shift-R (or Cmd-Shift-R on a Mac), and your changes should take effect.

Tip for stubborn service workers

Even when the Update on Reload check box is selected in the Application tab of Chrome's Developer Tools, a service worker can fail to update. This may be because of a lax caching policy that instructs the browser to hold the service worker in the cache. In our example, you have a Cache-Control header value of no-cache, which instructs the browser to revalidate the stored copy with the server for changes. To learn more about Cache-Control and how it works, check out chapter 10.

With the updated service worker running, open the Network tab and check out the asset information in the Size column. Those that have been intercepted by your service worker will read *(from ServiceWorker)*, as shown in figure 9.9.

Figure 9.9. Network requests intercepted by the service worker will be indicated by a value of "(from ServiceWorker)" in the Size column in Chrome's network utility.

Requests intercepted by the service worker

Now that you've verified that items are being cached by the service worker by way of the CacheStorage API, select the Offline check box in the network utility next to the network throttling drop-down, and then reload the page. Rather than receiving a network error, you'll notice that the site is now available offline. Congratulations! You just wrote your first offline web experience! Now let's look at how these changes have affected the performance of the page.

9.2.4. Measuring the performance benefits

When you retrieve assets from the service worker cache, you can achieve better performance than the browser cache. This means that you can further accelerate rendering performance for the user by lowering the amount of time it takes for the browser to begin painting the

page. Figure 9.10 tracks the Time to First Paint of three scenarios: when the browser has nothing in its cache, when the cache is populated, and when the service worker cache is used instead of the browser cache.

Figure 9.10. A comparison of the Time to First Paint performance of various caching scenarios on Chrome's Regular 3G throttling profile. The scenarios are an uncached page, the page when retrieved by the browser cache, and the page when retrieved from the service worker cache.

This was somewhat surprising to me, but the numbers show that when service workers are used to enhance performance, you can see a 50% improvement over the browser's caching behavior. That's a sizeable decrease in rendering time!

This doesn't mean the browser cache is dead. You still need it, because it works so well and because you can fall back to it in browsers that don't support service workers. Even in browsers that *do* support service workers, you can configure your fetch event code to

ignore requests that you *don't* want to intercept, at which point your requests will fall to the browser cache.

Next, you'll take another look at your service worker code, and see how to tweak it to be a little more flexible.

9.2.5. Tweaking network request interception behavior

So you've written your first service worker, and it works pretty well. Except for one thing: If you try to change any of your assets, those changes won't be seen unless you force a reload of the entire page. This is problematic. In particular, it's important that your site's HTML be as up-to-date as possible, so that if you update references to things like CSS or JavaScript, you'll be able to see those changes as well as updates to content.

That's not to say that service workers are inherently problematic, or that using CacheStorage over the browser cache is a bad idea. If you want to provide an offline experience, it's the only real way to do so. When you intercept and change the way network requests are fulfilled, however, you're writing behavior that supersedes the browser's own built-in cache. You must be mindful of how you choose to fulfill these requests.

How you fulfill these requests depends on the nature of your website. In the case of the blog example in this chapter, the strategy is basic: assets that don't change often (such as images, scripts, and CSS), you don't worry about right now. When you *do* need to change them, there's a mechanism for doing so that's covered in the next section.

For HTML, you adopt a different strategy in your service worker's fetch event code that will allow you to get the latest page content every time when you're online. You can still accommodate offline viewing for your user as a part of a fallback strategy.

Your current service worker fetch event code is straightforward: if a request for an asset matches something that's already in the service worker cache, you serve the asset from the cache. If the request doesn't match anything in the cache, you grab the latest copy from the network, and then add it to the cache.

This is an excellent strategy for performance, but it can negatively impact content freshness. You don't want to abandon this strategy altogether, because some assets rarely ever change. You want to adopt a new approach for HTML documents. Figure 9.11 shows how to adopt a two-pronged approach to intercepting network requests in the service worker's fetch event.

Figure 9.11. A two-pronged approach for intercepting a network request in a service worker's fetch event. If the requested asset is an HTML document, you always fetch it from the network and place it in the cache, and serve it from the service worker cache only if you're offline. If the resource isn't an HTML document, you always serve from the cache and retrieve it from the network if it's not in the service worker cache.

With this flow, you're maintaining the performance advantages that service workers give us for assets such as images, CSS, and JavaScript, but giving priority to the freshness of HTML content. This gives you the ability to update site assets by changing their URLs, which you can later point to in the cache that you populate in the service worker's install event code.

To implement this new flow, you need to update the service worker's fetch event code. First, you need to add a new regular expression to check whether the incoming request is an HTML document. Changes are in bold.

Listing 9.4. Adding a regular expression to check for HTML requests

```
var allowedHosts = /(localhost|fonts\.googleapis\.com|fonts\.gstatic\.com)/i,
    deniedAssets = /(sw\.js|sw-install\.js)$/i,
    htmlDocument = /(\/|\.html)$/i;
```
A regular expression that checks if a URL is an HTML document.

This regular expression will be used later to check whether the request URL is for an HTML document, and will be what your network interception request behavior hinges on. From there, you'll create a new condition with this regular expression. If the regular expression test passes for the current request, and you're dealing with an HTML document, you'll handle the request in your pattern of network first/serve from cache for offline. If the test fails, you'll use your initial pattern of cache first/populate cache from the network.

Listing 9.5. Handling HTML requests with a network first/cache for offline pattern

event.respondWith intercepts the request and responds with the encapsulated code.

Checks if the current request is for an HTML document

```
if(allowedHosts.test(event.request.url) === true &&
   deniedAssets.test(event.request.url) === false){
   if(htmlDocument.test(event.request.url) === true){
      event.respondWith(
         fetch(event.request).then(function(response){
            caches.open(cacheVersion).then(function(cache){
               cache.put(event.request, response.clone());
            });
```
Opens the cache and places the network response into it for offline use later

Fetches the HTML document immediately from the network

```
            return response;
         }).catch(function(){
            return caches.match(event.request);
         })
```
Responds with the asset retrieved from the network

If a later request fails, the last version obtained is served from the cache.

```
      );
   }
   else{
      /* Handle non-HTML requests as before */
   }
}
```
Other assets are handled as previously defined.

After you reload the page and try the new code, go ahead and modify index.html. You'll notice that updates to it should be reflected immediately.

This method slows rendering performance of the page *slightly* as compared to your earlier fetch event code, because the document needs to be fetched from the network rather than read from the service worker cache. Testing under the Regular 3G throttling profile in Chrome shows an average Time to First Paint of 120 ms. Although slower than the average time of 90 ms that your earlier service worker code yielded, it's still faster than the browser cache's average Time to First Paint of approximately 175 ms.

The results you get will depend on your specific project. Keep in mind that you don't *need* to intercept *every* network request, nor *should* you. Remember that network requests in your fetch event code are intercepted only if you pass a response object to the event object's respondWith method. If a request isn't passed to this method, the browser's default behavior will kick in. The server worker specification doesn't prescribe any method for handling requests; it only provides an interface for you to do so. *You* dictate the logic with respect to whether a request is intercepted. In this chapter's example, you've already done quite a bit of this by creating regular expressions to filter out requests that you don't want to intercept.

Your service worker and CDN-hosted assets

CDN-hosted assets are another aspect of request interception, and so is caching them with CacheStorage. Generally speaking, you'll be able to save CDN-hosted assets to your service worker cache without trouble. CDN hosts configure their servers to serve assets with an Access-Control-Allow-Origin: * header, which allows any origin to access those resources without restriction. If you're having trouble caching a CDN asset, check for the presence of this header. All properly configured CDNs supply this header, so adding special logic in your service worker to work with these assets is something you won't need to worry about. For further information on CDNs and how they work, check out [chapter 10](#).

Back to your service worker: even though you're sacrificing performance a bit by hitting the network for HTML requests, the net effect over the browser cache is positive. Better yet, this method still lets you serve content to users when they're offline. It's the best of both worlds.

Of course, if you modify the site's CSS, JavaScript, or images, those will still be served from the service worker cache, and updates won't be reflected. There's a good approach to dealing with those assets, and next we cover how to update your service worker cache to include changes to your site assets.

9.3. Updating your service worker

So far, you've written a service worker that caches site assets such as CSS, JavaScript, and images, and always fetches a fresh copy of HTML files from the server. Let's imagine that you've pushed this service worker code to production, and it's working great.

Unfortunately, you've hit a snag in that you have new CSS that you need to make sure your users see, but the old CSS file cached by the service worker is stubbornly persistent, and updates only if you force a reload of the entire page. This is problematic, because it's not a good solution to tell your users, "Just force a reload of the page to see the new

styles!" You need to be able to usher in the changed CSS and automatically put it in the user's service worker cache.

In this short section, you'll learn how to version files on your site so that the service worker picks those files up instead. Because you want to keep your caches lean out of respect for your user's device storage quota, I'll then show you how to clear out the old cache. Let's begin!

9.3.1. Versioning your files

You'll recall that you wrote your service worker fetch event code to always serve files such as CSS, JavaScript, and images from the service worker cache, but to always prefer fetching HTML from the network if the user is online. One good reason for this is so that you can force updating of other asset types by versioning the references to them in the HTML file. Because the HTML will always be fetched from the server, you can ensure that any new references to assets will be downloaded by the user.

In regards to caching, versioning occurs when you take a file and modify the reference to it. Take global.css, for example, which is included in index.html via the <link> tag:

<link rel="stylesheet" href="/css/global.css" type="text/css">

This approach is familiar to you by now. It's a useful one-liner that instructs the browser to download global.css. What happens if you make a change to global.css, though? Your service worker will never pick up on it, because it's already been cached. In fact, depending on the caching policy for that file, even the native browser cache may never pick up new changes.

This is where the concept of versioning comes in. By adding a query string to the filename as shown here in bold, you can differentiate the asset from its previous version:

<link rel="stylesheet" href="/css/global.css?v=1" type="text/css">

Even though the asset's filename is the same, the query string is enough of a differentiator for the browser to trigger it to download the file again, and treat it differently than the asset that doesn't have the query string.

Query strings in browser caches

The query string trick isn't handy only when trying to bust the service worker cache; it also works for the browser's native cache. Chapter 10 covers this trick in more depth, as well as automating this process to make repeated changes a lot less tedious.

To test this out, make a small but noticeable change in global.css—something like changing the background-color of the <body> element. If you open index.html and change the <link> tag reference to global.css to add the query string as shown previously, you'll notice that the new styles kick in immediately. Success! Or so you think? Maybe you should check out the Cache Storage section under the Application tab in Chrome's Developer Tools and look at your v1 cache. You'll see something like figure 9.12.

Figure 9.12. An orphaned cache entry after updating the style sheet reference.

global.css?v=1 **is in the cache, whereas the unused global.css entry remains.**

Although leaving this orphaned entry in the cache isn't going to kill anyone's user experience, it's not a good idea to leave it and move on. Think of it like littering. Is one candy bar wrapper tossed on the ground going to kill the world? Obviously not, but it's a bad thing to do, and you should always clean up after yourself.

Think of orphaned cache entries such as these as being like candy bar wrappers and bottles alongside the highway. Over time, your service worker cache will become bloated and take up unnecessary space on the user's device. In the next section, you'll learn how to clean up after yourself like a proper person.

9.3.2. Cleaning up old caches

Now that you've found out how to bypass a stubborn service worker cache, you need to learn how to remove outdated caches from it. The first thing you need to do is bump up the cacheVersion variable from v1 to v2, and replace your reference to global.css to read global.css?v=1 in your cachedAssets array. These changes are in bold.

Listing 9.6. Updating the cache name and the assets to cache

```
var cacheVersion = "v2",              ◁— New cache name
    cachedAssets = [
        "/css/global.css?v=1",         ◁——┐ Updated reference
        "/js/debounce.js",                 │ to the new CSS file
        "/js/nav.js",
        "/js/attach-nav.js",
        "/img/global/jeremy.svg",
        "/img/global/icon-github.svg",
        "/img/global/icon-email.svg",
        "/img/global/icon-twitter.svg",
        "/img/global/icon-linked-in.svg"
    ];
```

These changes alone are enough for the new cache to take effect, but it's not enough for the old v1 cache to be removed. You'll take care of that by rewriting the entire activate event code, which you can see in the following listing.

Listing 9.7. Removing old caches in the `activate` event

With this new activate event code, your service worker will process everything in your new cache. If any of the caches in the service worker don't go by any name specified in the cacheWhitelist variable, they'll be removed. After you run this code, go to the Cache Storage section in the left pane of Chrome's Application tab in the Developer Tools. You should be able to see that the only cache left is the new v2 cache, as shown in figure 9.13.

Figure 9.13. Your new v2 cache. If you click this, you'll be able to see the updated cache contents, particularly the `global.css?v=1` entry.

At this point, you'd proceed to update every reference to global.css to global.css?v=1. If you fail to do this, navigating to a subsequent page would store a separate cache entry for the old URI. This isn't a step you need to complete as part of the work in this chapter; it's more of a caveat for when you implement service worker changes on your own site.

You're are the end of your work in this chapter. Let's quickly recap what you've learned before moving on to the next chapter.

Going further with service workers

Covering all capabilities of service workers is outside the performance-oriented scope of this chapter. In fact, service workers are capable of more than creating offline experiences and boosting site performance. Although the patterns in this chapter are useful for content-driven sites such as blogs and the like, a few resources out there can help you take service workers a little (or a lot) further:

- Jake Archibald, a developer advocate for Google, has written a great article titled "The Offline Cookbook" available at https://jakearchibald.com/2014/offline-cookbook. It's a resource for patterns that you can use in your service worker. Some patterns are performance-oriented, some favor flexibility for offline experiences, and quite a few others fall in between. If you're wondering where to begin writing a service worker that makes the most sense for your website, this is a great place to start.
- Mozilla has created its own service workers cookbook at https://serviceworke.rs. It covers a broad spectrum of possibilities with the technology, including how to use service workers to send push notifications to mobile devices (for real!).

Throughout this chapter, you've made heavy use of the CacheStorage object, particularly methods such as caches.match and caches.open, which match items in a local cache and open caches by name, respectively. Although it works great with service workers, this API is a standalone piece of functionality with many methods available for use. To learn more about CacheStorage, check out the Mozilla Developer Network reference at http://mng.bz/NVXR.

In the first section of this chapter, recall that I said that communication between a service worker and its parent page isn't possible unless you use the postMessage API. Learn about this technology at Google Chrome's GitHub site at http://mng.bz/De31.

9.4. Summary

As you've witnessed in this chapter, service workers can be a tool used to enhance the performance of a website. Specifically, you learned the following key concepts:

- Service workers are a kind of JavaScript worker that operates on a thread separate from the main processing thread on which all other scripting activity occurs.
- Installing a service worker is easily done in browsers that support service workers. You check for the serviceWorker member in the navigator object. If a browser doesn't support the technology, the page experience will continue on without the service worker functionality.
- Because progressive enhancement is necessary to provide a level of functionality to all users, it's important that your site doesn't explicitly *depend* on service workers. Service workers are an *enhancement*, and should not be a *requirement* for a site to work.
- Service workers require HTTPS to be used. Although you can develop and use a service worker over HTTP on localhost in development, make sure you have a valid SSL certificate for when you push your service worker to a production server.
- Using CacheStorage in tandem with the service worker's fetch event, you have a vast amount of power and flexibility in intercepting and caching network requests. The marriage of these two pieces of functionality allows you to enhance page performance by serving items directly from the service worker cache, as well as fall back to an offline experience when a user's internet connection is absent or intermittent.
- Although the service worker cache operates similarly to the browser cache, it's a separate entity. If you don't intercept and send network requests to the fetch event's event.respondWith method, the responses to those requests will be handled according to the browser's default behavior.
- Service workers can provide a performance boost when it comes to rendering. In the case of my blog, it provided nearly a 50% boost in rendering speed over the browser cache!
- Aggressive caching of HTML documents can create a scenario where it becomes difficult to update HTML content on a page, as well as the assets referenced on the page, such as CSS, JavaScript files, and images. For content-driven sites such as a blog, it makes more sense to fetch these assets from the network and *then* cache them in case the user goes offline later.
- Sometimes site assets change, and you need to invalidate your service worker's cache. If this happens, you can invalidate a service worker cache by adopting a new name for the cache, and dropping it into a white list. From here, you can use the service worker's activate event to eliminate all caches that aren't a part of your white list, ensuring easy updates and cleanup when assets are changed on your site.

In the next chapter, you'll explore methods you can use to fine-tune the delivery of assets on your website, ranging from configuring your site's browser caching policy, providing resource hints, working with CDNs, and more.

Chapter 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.

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.

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.

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.

Effects of compression levels on
TTFB and load times of jQuery
(lower times are better)

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){          ⟵──┐ Apply compression
        return true;                   ⟵──┐            based on logic that
    }                                     Compress      you define
}));                                      everything
```

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.

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

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.

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.

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

accept-encoding: gzip, deflate, sdch, br

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!

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

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.

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.

Brotli encoding token

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 II. 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.

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.

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.

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

Time to first byte for gzip vs. Brotli
compression of jQuery
(lower is better)

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 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.

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.

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.

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.


```
First visit:
uncached ————→  14 requests  |  298 KB transferred  |  Finish: 4.04 s  |  DOMContentLoaded: 1.88 s  |  Load: 4.04 s

Subsequent  ————→  14 requests  |  2.9 KB transferred  |  Finish: 628 ms  |  DOMContentLoaded: 550 ms  |  Load: 657 ms
visits: cached
```

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 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.

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

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

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

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 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- e-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.

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.

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.

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 used

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"),                    Imports the mime
    compression = require("compression"),            module that you'll use
    path = require("path"),                           to look up file types for
    mime = require("mime"),                     <──── requested assets.
    app = express(),
    pubDir = "./htdocs";

// Run static server                                 setHeaders callback lets
app.use(compression());                              you specify behavior
app.use(express.static(path.join(__dirname, pubDir), {    that will run just before
    setHeaders: function(res, path){          <────   the response is sent.
        var fileType = mime.lookup(path);        <────
                                                     File type of the requested
        switch(fileType){                            asset is determined by the
            case "text/html":                        mime module.
                res.setHeader("Cache-Control",
                              "private, no-cache, max-age=" + (60*60));

                break;

            case "text/javascript":                  JavaScript and CSS files set
            case "application/javascript":            Cache-Control: public,
            case "text/css":                          max-age=2592000.
                res.setHeader("Cache-Control",
                              "public, max-age=" + (60*60*24*30));
                break;

            case "image/png":                         PNG, JPEG, and SVG images set
            case "image/jpeg":                        Cache-Control: public, max-
            case "image/svg+xml":                     age=31536000.
                res.setHeader("Cache-Control",
                              "public, max-age=" + (60*60*24*365));
                break;
        }
    }
}));
app.listen(8080);
```

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

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

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. 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.

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.

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.

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.

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

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.

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

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.

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

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.

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.

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.

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 <u>figure 10.20</u>.

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

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 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.

Chapter 11. Looking to the future with HTTP/2

`This chapter covers`

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

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

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

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

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

11.1. Understanding why we need HTTP/2

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

11.1.1. Understanding the problem with HTTP/1

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

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

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

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

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

Head-of-line blocking

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

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

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

Another rather hacky way around this request limit is to use a technique called *domain sharding*. This technique gets around the maximum simultaneous request limit of six per domain by spreading requests across domains. With two domains serving content, twelve requests can be fulfilled at once. With three, up to eighteen requests could be accommodated

simultaneously. Although this technique *is* effective, it requires a significant investment of resources, both temporal and financial, to implement. It's not an option for every organization.

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

Uncompressed headers

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

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

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

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

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

Nonsecure web sites

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

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

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

11.1.2. Solving common HTTP/1 problems via HTTP/2

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

No more head-of-line blocking

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

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

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

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

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

Header compression

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

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

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

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

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

HTTPS is guaranteed

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

SSL performance overhead

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

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

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

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

11.1.3. Writing a simple HTTP/2 server in Node

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

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

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

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

This downloads all the source code and installs the Node packages you need for the HTTP/2 server you're going to write, including the spdy package. Once everything is good to go,

open your text editor, and create a new file named http2.js in the root folder of the website. In this file, enter what you see here.

Listing 11.1. Importing modules needed for the HTTP/2 server

```
                                  Imports the filesystem
                                  module for reading files.
 Imports the SPDY                                              Imports the path module
 module for HTTP/2      var fs = require("fs"),                for normalizing paths.
 functionality.             path = require("path"),
                            http2 = require("spdy"),
                            mime = require("mime"),
                            pubDir = path.join(__dirname, "/htdocs");      Sets the root
                                                                           directory to
                     Imports the MIME module for                          serve files from.
                     determining content types.
```

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

Listing 11.2. Setting up SSL certificates on the server

```
 Creates an instance                                          The key and certificate
 of the HTTP/2 server.                                        files needed for SSL

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

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

Listing 11.3. Writing the HTTP/2 server behavior


```
The content                                                    The request handler.
type of the      }, function(request, response){  ←┘
  asset.               var filename = path.join(pubDir, request.url),  ←┐ The filesystem
            └▷             contentType = mime.lookup(filename);           path to the asset.

                   if((filename.indexOf(pubDir) === 0) &&
                       fs.existsSync(filename) &&              Checks if the
                       fs.statSync(filename).isFile()){        asset exists.
                       Sends a 200 response to the client.
                   response.writeHead(200, {  ←
                           "content-type": contentType,     ← Sets the Content-Type
                           "cache-control": "max-age=3600"      response header.

Caches assets ┌─▷      });
    for 1 hour. │
                       var fileStream = fs.createReadStream(filename);  │ Sends the
                       fileStream.pipe(response);                        │ asset to
                       fileStream.on("finish", response.end);           │ the user.
                   }
                   else{
                       response.writeHead(404);      │ If the asset isn't found, a
                       response.end();               │ 404 response is sent.
                   }
               });
                                                  │ Starts the server
               server.listen(8443);             ←┘ on port 8443
```

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

```
node http2.js
```

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

Making an exception

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

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

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

11.1.4. Observing the benefits

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

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

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

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

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

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

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

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

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

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

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

11.2. Exploring how optimization techniques change for HTTP/2

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

- Techniques that *reduce* the size of assets are things you should *still* do on HTTP/2. These are techniques such as minification, server compression, and image optimization. Reducing the size of an asset contributes to lower load times, always and forever.

- Techniques that *combine* files are things you should *stop* doing on HTTP/2. Although useful in alleviating latency in HTTP/1 client/server interactions, requests are much cheaper in HTTP/2, and combining files can have an adverse effect on your caching effectiveness.

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

11.2.1. Asset granularity and caching effectiveness

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

Why *is* this, exactly? The answer is in caching. As you learned in chapter 10, caching helps reduce the payload of a page on visits subsequent to the first. The problem isn't in caching itself, however, because a properly configured caching policy will operate whether or not we concatenate files. The problem is that when you concatenate files, you're reducing the efficiency of your caching when assets change. This is true of both protocols, but when using HTTP/1, you were willing to forego a certain level of efficiency to minimize load times for users visiting a site for the first time.

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

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

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

11.2.2. Identifying performance antipatterns for HTTP/2

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

Bundling CSS and JavaScript

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

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

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

Image sprites

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

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

Asset inlining

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

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

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

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

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

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

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

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

11.3. Sending assets preemptively with Server Push

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

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

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

11.3.1. Understanding Server Push and how it works

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

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

Imagine that a user goes to the Weekly Timber website (which runs on an HTTP/2 server for the purpose of this example) and requests index.html. Predictably, the server receives the request for index.html and constructs a response for it. But let's also imagine that the owner of the web server has configured the server to *also* respond with a copy of styles.min.css, which is the site's style sheet. This reduces the amount of time that the user has to wait for the styles to download, because the server doesn't have to wait for the client to request styles.min.css. The server sends it in parallel along with the response for index.html. [Figure 11.12](#) shows this process in action.

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

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

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

11.3.2. Using Server Push

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

How Server Push is typically invoked

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

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

If this looks familiar to you, it's because the preload resource hint HTTP header covered in chapter 10 takes on the same format. That said, *don't* confuse this with the <link> tag

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

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

Informing the browser of other pushed content types

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

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

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

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

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

Writing Server Push behavior in Node

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

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

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

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

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

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

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

Pushed asset

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

Profiling HTTP/2 Server Push on the command line

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

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

11.3.3. Measuring Server Push performance

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

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

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

When the CSS was pushed to the user, the page began painting about 19% faster than when not pushed. On my broadband connection, this translated to about an 80 ms increase in rendering speed. This is nothing to sneeze at, especially when you consider that devices on slower mobile networks will realize a proportionately larger increase in rendering speed. Considering that it takes little effort on most servers to use, it's practically a gimme for the performance-minded web developer.

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

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

A note on server push and browser caches

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

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

11.4. Optimizing for both HTTP/1 and HTTP/2

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

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

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

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

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

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

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

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

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

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

11.4.2. Segmenting your users

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

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

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

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

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

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

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

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

When I import the data for Weekly Timber, I can see that around 18% of the site's visitors are using browsers that don't support HTTP/2. This is a conservative estimate too, because partial support in this case doesn't imply that every user in that segment can use HTTP/2, either.

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

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

11.4.3. Serving assets according to browser capability

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

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

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

If you want to skip ahead to the finished code, you can do so by typing git checkout -f protocol-detection-complete in your terminal.

Detecting the protocol version

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

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

Listing 11.6. Detecting the HTTP version

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

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

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

Adding the HTTP/1 class

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

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

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

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

Listing 11.8. Isolating other requests that don't require modification

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

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

When you've finished, run the server with Node and pull up the site in a modern browser. You'll see that the response is unchanged. But if you pull up the website in a browser that doesn't support HTTP/2, you'll notice that the `<html>` tag has the http1 class added to it, as shown in <u>figure 11.19</u>.

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

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

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

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

Replacing multiple scripts with concatenated ones for HTTP/1 users

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

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

Listing 11.9. Scripts on the Weekly Timber site

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

Candidate scripts for concatenation on HTTP/l connections

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

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

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

Concatenated version of all candidate scripts in listing ll.9

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

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

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

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

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

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

```
                    for(var i in scripts){                      Creates a new <script>
  Removes all           scripts[i].remove();                     element referencing the
  scripts that      }                                            concatenated scripts.
  aren't CDN-
     hosted.        jQueryScript.parentNode.
                    insertBefore(concatenatedScript, jQueryScript.nextSibling);    ←──

                    var newDocument = "<!doctype html>" +
                                 window.document.documentElement.outerHTML;
                    response.end(newDocument);
     });                                                Adds the new <script> element
                                                        pointing to the concatenated script.
```

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

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

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

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

Considerations

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

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

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

Listing 11.12. Serving assets by protocol in PHP


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

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

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

11.5. Summary

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

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

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

Chapter 12. Automating optimization with gulp

- Understanding how gulp works and why you should use it
- Structuring your project for use with gulp
- Installing gulp plugins
- Understanding how gulp tasks work
- Writing tasks for your project
- Testing your gulp-based build system on a client's website

The worst part of optimizing your website for performance is the sheer repetitiveness of it. Minify this CSS, uglify that JavaScript, optimize those images, and so on. The prospect of doing all of this important (yet mind-numbing) work puts a damper on your enthusiasm for the job.

Thankfully, there's a tool out there that automates all of these tedious tasks for you, and its name is gulp (http://gulpjs.com). gulp is a Node-based build system that makes your workflow much more efficient and saves you time.

In this chapter, you'll learn about gulp and how it works. You'll create a folder structure that works best for using gulp with your front-end development project. Once this structure is defined, you'll install the gulp plugins required for automating your optimization tasks.

Speaking of tasks, you'll learn about the anatomy of a gulp task, and then write tasks to help you automate optimization techniques you've learned throughout this book. These include things like minifying HTML, compiling and minifying your CSS from LESS files, uglifying JavaScript, and optimizing images. You'll also write tasks that watch your project's files for changes, and automatically run relevant tasks whenever files are changed or added to your project. Then you'll cap it off by writing a build task for compiling your project for deployment.

Once these tasks are written and defined, you'll fire everything up and try it out on the Weekly Timber website so you can see how it all works. Finally, we'll end this chapter and the entire book by highlighting other gulp plugins that exist in the gulp ecosystem, and where you can find even more plugins to automate all sorts of tasks (even though they might not have anything to do with improving your website's performance!). Let's get started with gulp!

12.1. Introducing gulp

When Node went mainstream, it became the conduit through which all sorts of useful tools were created by and for web developers. Before long, it was used to create complex tools such as unit-testing software, package managers, and even build systems. gulp is a Node-based build system. In this section, you'll learn why you should consider gulp, as well as how gulp works.

Warning: This chapter assumes you're using gulp 4

As it turns out, writing a technical book can pose interesting challenges. At the time of this writing, the release of gulp 4 was pending, and gulp 3 was the "latest" release. When you read this chapter, that may yet be true. This may change how you need to install the gulp package with npm. Don't worry, though! I'll guide you through these choppy waters when the time comes.

12.1.1. Why should I use a build system?

gulp bills itself as a streaming build system. It automates tasks for you that you'd otherwise have to do yourself. When you use a build system, you're free to focus on being productive.

"But why should I use a build system? The way I do stuff now works good enough for me!" This is the kind of thing I said to myself when tools like this started to become more common, and for the most part, the way I was doing my work *was* fine.

Except that it was extremely repetitive. I use LESS in many of my projects, and whenever I'd make changes to my LESS files, I'd go through a process like the one in <u>figure 12.1</u>.

Figure 12.1. An unautomated workflow for compiling LESS into CSS

Does this work? Sure! Does it drive you crazy the 500th time you do it? It should! It doesn't take long, but think of how much wasted time adds up when you repeat this process ad infinitum: You make a change. You save the file. You switch over to the terminal and run the command to compile your CSS. You switch over to the browser and reload the page. Repeat until your hands have turned into gnarled little claws at the tender age of 30. Okay, that's too

dramatic, but it *is* repetitive. With a build system, you can change your workflow as shown in figure 12.2.

Figure 12.2. An automated workflow for compiling LESS into CSS. The only tasks the developer has to perform are making and saving changes, while the build system builds the CSS and reloads the page for us.

This new workflow frees you to focus on being more productive. Instead of having to constantly rerun commands in a terminal, you launch the build system once and focus on editing your CSS and seeing the changes appear in your browser window as you make them.

Of course, you're not limited to using a build system to compile LESS files to CSS. You can use it to minify your CSS in the process, as well as minify other assets, optimize images, and generally do whatever the build system's plugin ecosystem allows you to do. In the case of gulp, this is a virtually limitless number of tasks you can automate, as the gulp ecosystem has approximately 2,500 plugins that you can download and use at the time of this writing.

So now that you know some of the benefits that a build system such as gulp can lend to your workflow, you'll probably want to know how gulp works. Let's find out!

12.1.2. How gulp works

As I said before, gulp bills itself as a *streaming build system*, but what does that even mean? Streams are points in gulp's build process where data is transformed. Multiple streams can be chained to create tasks. Let's start by talking about how streams work.

How streams work

Streams are an old concept of I/O, and in gulp, they allow you to transform the input of data via plugins, and then pipe that transformed input as output. This process of a single point in a stream as it relates to compiling LESS files is shown in figure 12.3.

Figure 12.3. The concept of a stream. In this example, the input is composed of LESS files that are piped into the stream, which then compiles the LESS into CSS and pipes that completed output into a CSS file.

This is the simplest representation of data I/O as it works in gulp. Some input data, usually a file on disk, is piped into a stream that's transformed by a plugin of some sort. The transformed data is then output by the stream, which can then either be written to disk or passed into further streams prior to that step. Figure 12.4 shows a chain of streams connected to transform data multiple times.

Figure 12.4. An example of data being piped in and out of multiple streams. The first stream compiles the LESS file into CSS, which is then piped into another stream that minifies it.

Chaining streams lets us take the same input and transform it multiple times. You can chain as many streams as necessary, and when finished, pass it to a handler that will write the output to a file on the disk. In the preceding example, you take the input of a LESS file and pass it to a stream that compiles it into CSS. Then you take that output and pipe it as input into yet another stream that will minify it.

Now that you understand streams, let's talk about the bigger picture of where they belong, which leads us to gulp tasks.

How tasks work

Streams—any number of them—are the building blocks of what's called a *task*. In gulp, a task accomplishes a specific thing (or set of things) that begins with reading the data from the disk. The outline of a simple task with a single stream is shown in figure 12.5.

Figure 12.5. The outline of a task. The task is identified by its name, buildCSS, and begins with a LESS source file named main.less that resides on the disk. This file is piped into a stream that compiles main.less into a CSS file that is outputted from the stream and saved to the disk as styles.css.

That's all a task is. It's a wrapper for streams that begin with input from the filesystem, and ends with stream output that's written back to the filesystem in another location.

A single task is usually relegated to a single concern. In this case, the buildCSS task shown in figure 12.5 deals with the CSS-related aspect of the project. For minifying your HTML, you'd write a separate task. The same goes for other things such as optimizing images and uglifying your JavaScript.

As many tasks as necessary can be defined for a project. When the code for these tasks is assembled, this creates the project's build system, also known as a *gulpfile*. Before you can embark on writing your gulpfile, however, you need to create a folder structure for your project, and then install gulp and your plugins.

12.2. Laying down the foundations

Before you can get started writing the gulpfile for the project, you need to do a couple of things. You need to set up a folder structure for your project, and then you install all the plugins you need. Let's start by getting your folders in order.

12.2.1. Structuring your project's folders

When you begin a new project, the first thing you want to do is set up a folder structure for it. This is true even when you use a build system, but it does change things a bit.

As I said in the previous section, tasks begin with input data from a source file, and end with output data that's written to the disk. Because of how this works, you're going to edit your files in a source directory, and use the build system to compile everything to a distribution directory. Figure 12.6 shows this process.

Figure 12.6. The build system processes files from a source folder (src in this example)

and processes them and writes the output to the distribution folder (named dist)

To get started, create a new folder on your computer. Name it anything you like. Then go into it and make a folder named src. This folder is where you'll edit your files. You likely don't have a project of your own to work on, so you'll populate this folder with the Weekly Timber website files. To do this, you use the git command to pull the website down from a remote repository:

git clone https://github.com/webopt/ch12-weekly-timber.git ./src

This gives your build system something to build. After this command finishes, you need to create a folder in the root of the project named dist. When finished, you'll have a folder structure that should look something like this:

```
/
src
  img
  js
  less
dist
```

When your folder structure is set up like this, you're ready to go. Note that your projects don't necessarily *need* to follow this exact structure. The general idea is to separate the folder that you do your work in from the folder that the build system outputs files to. Next, you'll install gulp and the plugins you need for gulp to do its job.

12.2.2. Installing gulp and its plugins

Before you can *do* anything with gulp, you need to globally install the gulp command-line interface on your system. This will enable you to process gulpfiles with the gulp command. Use this command for the installation:

```
npm install -g gulp-cli
```

You'll never need to run this command again, as the gulp program will then be globally accessible on your system. Then, you initialize your project directory with npm:

```
npm init
```

When you execute this command, you'll be asked your project's name, version number, and other things. What you enter here isn't terribly important as far as the work in this chapter is concerned, so enter the values you feel are necessary, and omit those that you're unsure of or don't care about. They're mostly relevant for projects of your own, or npm modules that you intend to publish on npmjs.com.

What this command does do that's convenient, however, is create a file named package.json that keeps track of all the modules that you install for your project. This makes the project portable so that you won't need to distribute the modules that you install for it using npm. When you use the --save flag with npm's install command, it will write the module information to package.json. Now you can commence with installing gulp itself!

Installing gulp itself

Earlier in this chapter, I said that, depending on the status of gulp, version 3 may be the latest release but that this won't be the case in the future. On the off chance that gulp already has updated as you read this, you need to make sure that you're getting gulp 4 instead of gulp 3. Let's first check for the latest version of gulp available in the remote package repository by using npm:

```
npm show gulp version
```

When this command finishes, you'll get the version number of the package. If you receive a response starting with 4 (for example, 4.0.0 or something similar), then you're golden, and you can install the gulp package normally by using npm:

```
npm install gulp --save
```

If you receive a response starting with a 3 (for example, 3.9.1), then things get a little trickier. You'll need to use npm to install the gulp package from GitHub, and point to the repository's 4.0 branch. This is easily done with the following command:

```
npm install gulpjs/gulp#4.0 --save
```

This syntax may seem unfamiliar, but what you're doing here is pointing to a GitHub user (gulpjs), a repository (gulp), and then a specific branch (#4.0). This allows you to install version 4 of the gulp package from GitHub without it being available yet via npm. Depending on when you read this chapter, the #4.0 branch of this repository may cease to exist after gulp 4 is tagged as the latest version. In this case, you'd simply install the gulp package with the npm install gulp command as usual.

Now you can get started with installing the plugins you'll need for your project. You'll categorize these plugin installations by the functions they provide, and describe what each one will do.

Essential plugins

This category of plugins is what you'll need for gulp to work and do the basic stuff for you. To install these plugins, type in the following command:

```
npm install gulp-util del gulp-livereload gulp-ext-replace --save
```

These plugins fulfill the purposes shown in table 12.1.

Table 12.1. Essential gulp plugins

Plugin name	Purpose
gulp-util	Used by some plugins to output information to the terminal, such as errors and diagnostic information.
del	Deletes files and folders. Useful for when we want to perform "clean" builds that involve deleting the distribution folder and building from scratch.
gulp-livereload	Automatically reloads the browser when you change files. This involves installing the LiveReload plugin for your browser, which we'll cover when we finish writing the build system.
gulp-ext-replace	Allows us to specify a different file extension for the destination output than what exists in the source input. When we convert our PNG and JPEG files to WebP by using imagemin-webp, you'll need this to save files with a .webp extension.

HTML minification plugin

One optimization task you can automate is the minification of HTML. This requires only one plugin named gulp-htmlmin that you install like so:

```
npm install gulp-htmlmin --save
```

When this finishes installing, the HTML minification plugin will be ready for use by your gulpfile. This plugin takes all of the whitespace and unnecessary characters out of our HTML for you, which results in fewer bytes transferred to the client. Fewer bytes means faster page load times.

CSS-related plugins

The Weekly Timber site uses LESS as the precompiler of choice for building CSS, as well as PostCSS and a set of PostCSS-centric plugins. To install these plugins, you enter the following:

```
npm install gulp-less gulp-postcss autoprefixer autorem cssnano --save
```

describes these plugins.

Table 12.2. CSS-related gulp plugins

Plugin name	Purpose
gulp-less	Compiles LESS into CSS that the browser can understand. If you're a SASS user, don't despair! You can use the gulp-sass plugin if your project depends on SASS (which is mentioned in section 12.4.) Weekly Timber uses LESS, so you'll go with this plugin for this chapter.
gulp-postcss	A library that transforms CSS. PostCSS accomplishes *tons* of tasks via plugins in the PostCSS ecosystem. Learn more at http://postcss.org.
autoprefixer	PostCSS plugin that automatically adds vendor prefixes to CSS for you. Useful for backward compatibility without using LESS/SASS mixins, or awful copy/paste workflows. Write CSS, and autoprefixer will take care of the vendor prefixing minutiae.
autorem	Another PostCSS plugin (written by me!) that converts px units into rem units. Useful for making your pages more accessible without the pain of having to manually convert every single px unit to rem.
cssnano	A PostCSS plugin that minifies and makes many focused optimizations to your CSS that results in a lower file size. Learn more about cssnano at http://cssnano.co.

JavaScript-related plugins

Your JavaScript requires two plugins for optimization purposes. To install them, enter the following command:

npm install gulp-uglify gulp-concat --save

Table 12.3 describes the functionality of these plugins.

Table 12.3. JavaScript-related gulp plugins

Plugin name	Purpose
gulp-uglify	Uglifies JavaScript files. If you're not familiar with uglification, it's like minification in that it removes all unnecessary whitespace from a JavaScript file, but also shortens code, while preserving functionality, to yield even smaller file sizes.

Plugin name	Purpose
gulp-concat	Concatenates JavaScript files. While concatenation is a no-no with HTTP/2 connections, you can easily generate a concatenated version of site scripts that can be conditionally used for HTTP/1 connections.

Image-processing plugins

You may recall in chapter 6 that the largest asset type tends to be images. So it stands to reason that you'll want to find a way to automate the optimization of images. It turns out that gulp gives you a lot to work with to achieve this. You'll need to install these plugins:

```
npm install gulp-imagemin imagemin-webp imagemin-jpeg-recompress
    imagemin-pngquant imagemin-gifsicle imagemin-svgo --save
```

Table 12.4 describes the functionality of these plugins.

Table 12.4. Plugins related to image optimization

Plugin name	Purpose
gulp-imagemin	Provides the base imagemin functionality. You'll recall from chapter 6 that this was the Node module you used to optimize images. You can use this gulp extension to automate that behavior for us.
imagemin-webp	Allows you to convert images into WebP, which are usually smaller than their PNG and JPG counterparts. WebP support in Chromium-derived browsers means that a large segment of users can use them.
imagemin-jpegrecompress	An imagemin plugin used for optimizing JPEG images.
imagemin-pngquant	An imagemin plugin used for optimizing PNG images.
imagemin-gifsicle	An imagemin plugin used for optimizing GIF images.
imagemin-svgo	An imagemin plugin used for optimizing SVG images.

With all of these plugins installed, you're now ready to write your gulpfile, which you'll tackle in the next section.

12.3. Writing gulp tasks

gulp tasks are basic in their composition. They consist of many chained parts that pipe data from one stream to another. In this section, you'll learn the anatomy of a gulp task and then you'll go about building your gulpfile.

12.3.1. The anatomy of a gulp task

gulp tasks are terse expressions of the objectives you want to achieve as a part of a build process. Think of tasks as wrappers around the functionality you seek. Each task is encapsulated by the gulp.task method. This method generally takes one argument, which is a pointer to a function that performs the task. The following code shows the shell of a task:

```
function minifyHTML() {
    // Task code
}

gulp.task(minifyHTML);
```

Here you can see a function named minifyHTML. You can set up your task code within it, and subsequently bind it to the task method, which will define it. There are other aspects to using the task method, but this is its simplest use. We'll cover other scenarios, such as running tasks in series or in parallel, later.

Reading source files

As I said earlier in this chapter, streams in gulp require an input source. The vehicle for providing input to a stream is gulp.src. This method takes an argument that accepts a string (or array of strings) for files that you want to read as input. The following is an example of the gulp.src method:

```
function minifyHTML() {
    return gulp.src("src/*.html");
}
```

Here you use the gulp.src method to read all HTML files that are in the src folder (the location of which is relative to the location of the gulpfile). The file pattern of "src/*.html" you use here is what's known as a *file glob*. If you've worked with files in a terminal for any length of time, you've had some exposure to this concept. Here are a few example patterns to get you up to speed if you're not too familiar with globbing:

- `img/*` matches everything in the img folder.
- `img/**` matches everything in the img folder *and* its subfolders.
- `img/*.png` matches all PNG images in the img folder.
- `img/**/*.png` matches all PNG images in the img folder *and* its subfolders.
- `img/**/*.{png, jpg}` matches all PNG *and* JPEG images in the img folder *and* its subfolders.
- `!img/**/*.svg` *excludes* all SVG images in the img folder *and* its subfolders.

Most of the work you do in Node will use patterns that are largely similar to these, but file globbing is much more powerful than only these patterns. For a primer on globbing, check out https://github.com/isaacs/node-glob#glob-primer.

Moving data through a stream

Once input is read from the disk via gulp.src, you need a mechanism that helps you ferry that data to plugins. This is done by using the pipe method. In the following example, you use the pipe method to move data along from the source to the htmlmin plugin, which minifies HTML:

```
function minifyHTML(){
    return gulp.src("src/*.html")
        .pipe(htmlmin());
}
```

This example is more abstract, because it doesn't show where or how the htmlmin() instance is created, but we'll cover that soon. The important piece here is the pipe method. When operating on a stream, you use pipe and chain it after gulp.src. The pipe method takes an argument for a function that you want to pass data to. In this case, you're taking the data you've read with gulp.src, and then you pipe that data to htmlmin(). When htmlmin() finishes its job, it will return the minified HTML data that you can then pipe again to another point in the stream, such as writing the minified output to files on the disk.

Writing data to the disk

The final part of a task's journey is to take the data you've transformed from a source file, and write it to files on the disk. To do this, you use pipe and pass gulp.dest to it.

gulp.dest is a method that takes an argument specifying what the destination of the transformed data should be. Rather than a glob, this is a string that identifies a specific folder or file to write to on the disk. The following is an example of the pipe method ferrying minified HTML output to gulp.dest:

```
function minifyHTML(){
```

```
    return gulp.src("src/*.html")
        .pipe(htmlmin())
        .pipe(gulp.dest("dist"));
}
```

The sample task is complete: it reads HTML files from the disk by using gulp.src, then pipes the data to htmlmin(), and then pipes the minified HTML to the dist folder via gulp.dest. See how simple gulp tasks are? Most are usually short, requiring little to no configuration for most tasks. Armed with your new knowledge of gulp's methods, you're ready to write a gulpfile.

12.3.2. Writing the core tasks

gulp works via the gulp command, which you installed on your system when you installed the gulp-cli Node package earlier in this chapter. When executed, gulp looks for a file in the current working directory named gulpfile.js. If no gulpfile is found, nothing happens and gulp will exit. If a gulpfile *is* found, however, gulp will run any task you've specified within it.

Want to skip ahead?

If you're stuck and want to skip ahead, or you'd like to grab the finished gulp boilerplate and start using it right away, clone the GitHub repository that contains the finished build system by typing git clone https://github.com/webopt/ch12-gulp.git in your terminal window.

In this section, you'll begin writing your gulpfile by importing modules, and then working your way through each of the core tasks. Once finished, you'll have the meat of the gulpfile written.

Importing modules

Let's start by creating a new file named gulpfile.js in the root folder of your project. Before you can do anything with gulp and the myriad plugins you've installed for it, you need to import them into your gulpfile. Using your text editor, enter the contents of this listing into your gulpfile.

Listing 12.1. Importing all modules needed for the gulpfile

```
          var gulp = require("gulp"),
              util = require("gulp-util"),
              del = require("del"),                         Foundational modules
HTML          livereload = require("gulp-livereload"),      needed for the build
minification  extReplace = require("gulp-ext-replace"),     system to work
module        htmlmin = require("gulp-htmlmin"),
              less = require("gulp-less"),
              postcss = require("gulp-postcss"),            Modules required for
              autoprefixer = require("autoprefixer"),       building LESS, as well
              autorem = require("autorem"),                 as PostCSS plugins
              cssnano = require("cssnano"),
              uglify = require("gulp-uglify"),
Plugins required  concat = require("gulp-concat"),
for uglifying and imagemin = require("gulp-imagemin"),
concatenating     jpegRecompress = require("imagemin-jpeg-recompress"),
JavaScript        pngQuant = require("imagemin-pngquant"),
              svgo = require("imagemin-svgo"),
              gifsicle = require("imagemin-gifsicle"),      Modules required for
              webp = require("imagemin-webp");              image optimization
```

This code imports all of the modules necessary for gulp to work its magic. When finished with this step, you can now create your very first gulp task.

Exploring the general structure of a task

Most of the tasks you'll write in this section will follow a predictable pattern. Some may differ slightly, but the basic structure should start to look familiar as you proceed. Figure 12.7 shows the general structure that your tasks will follow.

Figure 12.7. The general structure of gulp tasks you'll write for this chapter's gulpfile

```
┌──────────────────┐      ┌──────────────────┐      ┌──────────────────┐
│ Read source files│ ───▶ │Run optimization  │ ───▶ │  Write output    │
│    from disk     │      │task              │      │    to disk       │
│                  │      │(e.g., minify HTML)│      │                  │
└──────────────────┘      └──────────────────┘      └──────────────────┘
```

Tasks written in this chapter will almost always start by reading source files from the disk. From there, you'll pipe data to the plugin relevant to the task's goal. These are behaviors such as minifying HTML, optimizing images, and so forth. After this completes, you'll write the files to the disk by using the gulp.dest method.

Not every single task is going to follow this pattern in lockstep. For example, the utility tasks you use to make clean builds and watch files for changes will be markedly different. We'll tackle those and explain them as needed.

Ready to start writing gulp tasks? Let's start by writing the HTML minification task.

Minifying HTML

Minification is one of the foundational optimization methods in the web developer's toolbox. Although minifying HTML doesn't save gobs of bandwidth in all scenarios, it's super easy to do, and gulp makes it even easier.

In listing 12.1, you imported the gulp-htmlmin plugin into the htmlmin variable. With this, you can write the HTML minification task. Enter the contents of this listing into the gulpfile.

Listing 12.2. The HTML minification task

There's a bit here to process, but most of it is common to the rest of the tasks that you'll write. First, you read from HTML files on the disk by using gulp.src. From there, the data is piped to the gulp-htmlmin plugin, which minifies your HTML. You've passed two options to it: removeComments, which removes all comments in the HTML, and collapseWhitespace, which safely removes all of the whitespace in the file without impacting the integrity of the content. A full list of options is available at https://github.com/kangax/html-minifier#options-quick-reference.

After minification has completed, you write the changes to the dist directory via gulp.dest, and then pipe the stream to the livereload module, which signals to a listening LiveReload instance to reload the browser page. (We'll talk later in this section about how to set up your browser to listen for changes.) Finally, you bind the minifyHTML function to gulp's task method, which sets up the HTML minification task. If you haven't already, save your gulpfile. At the command line in the same directory as the gulpfile, run this command:

gulp minifyHTML

This runs the minifyHTML task that you wrote. When it runs, you should see output similar to this:

```
[13:47:33] Using gulpfile /var/www/ch12-gulp/gulpfile.js
[13:47:33] Starting 'minifyHTML'...
[13:47:33] Finished 'minifyHTML' after 64 ms
```

Your output will vary a little, but it should look pretty much the same. When the task finishes, you'll notice that all of the HTML from the src directory has been minified and saved to the dist folder.

Congratulations! You wrote your first task. The rest of the tasks are mostly similar in terms of effort, but with varying degrees of complexity. Let's tackle the CSS-related task next.

Building LESS files and using PostCSS

The next task is more involved than the HTML minification one you wrote. You'll be using the gulp-less plugin to compile LESS files from the src/less folder, transforming/optimizing the compiled CSS by using the gulp-postcss plugin, and then writing it to the dist/css folder. The modules involved in this task are gulp-less, gulp-postcss, autoprefixer, autorem, and cssnano. To continue, enter the contents of the following listing into your gulpfile.

Listing 12.3. The LESS compilation/CSS optimization task

```
Reports errors          ┌─▷      util.log(err);
if they occur.          │        this.emit("end");        ◁─── Ends the error-handling process.
                        │    }))
                                                          ┌─ Automatically adds vendor prefixes
Pipes the compiled      ┌─▷   .pipe(postcss([            │  to relevant CSS properties.
CSS into PostCSS.       │        autoprefixer({      ◁───┘
                        │  ┌─▷      browsers: ["last 4 versions"]
                        │  │    }),                        ┌─ autorem translates px
Adds prefixes for the four  │    autorem(),           ◁───┘  units into rem units.
most recent browser versions.│
                            │
cssnano minifies and    ┌─▷      cssnano()
optimize your CSS.      │    }))
                             .pipe(gulp.dest(dest))
                             .pipe(livereload());
                                                          ┌─ Binds the buildCSS
    }                                                     │  task function to gulp.
                                                      ◁───┘
    gulp.task(buildCSS);
```

This task, though more complex than the HTML minification one you wrote before, is simple. All of the styles for Weekly Timber are written in LESS, and the main file is the main.less file, which you read from the disk and pipe into the gulp-less plugin instance. This compiles the LESS into CSS. Additionally, an error handler is used to catch errors if they occur, and logs them to the console via the gulp-util plugin's log method.

From here, it's more involved. We use three PostCSS plugins in this project, all of which are passed into the gulp-postcss plugin instance: autoprefixer, autorem, and cssnano. These plugins add vendor prefixes to your CSS automatically, convert px units to rem units, and minify/optimize your CSS, respectively. When this all finishes, the minified CSS is written to the dist/css directory. When you finish this task, you test it out by running the task like so:

```
gulp buildCSS
```

If you go to the dist/css folder, you'll see the optimized CSS file as main.css. You've finished writing your CSS optimization task! Next, you'll take on your JavaScript tasks.

Uglifying and concatenating scripts

The JavaScript optimization tasks for your build system are twofold: uglifying your JavaScript to reduce its size and then concatenating them. You'll provide a concatenated and unconcatenated set of scripts for flexibility's sake in the event that you'd have the ability to provide optimal asset delivery for both HTTP/1 and HTTP/2. Let's start by first writing the uglify task by entering the contents of the following listing into your gulpfile.

Listing 12.4. The JavaScript uglification task

```
The task function
                 ⤷  function uglifyJS(){
                         var src = "src/js/**/*.js",          The destination folder to
The source                  dest = "dist/js";                 write the uglified scripts to
file glob
                         return gulp.src(src)                 Source files are piped
                             .pipe(uglify())                  into the uglify plugin.
                             .pipe(gulp.dest(dest))
                             .pipe(livereload());
Binds the
uglifyJS task            }
function to gulp.
                 ⤷  gulp.task(uglifyJS);
```

This task looks for JavaScript files recursively inside of the src/js folder and feeds them
into the gulp-uglify module instance. When finished, it will write the uglified scripts
into the dist/js directory.

Next, you should write the concatenation task that bundles scripts. This one is as simple
as the uglifyJS task. Enter the contents of the following listing into your gulpfile.

Listing 12.5. The script concatenation task

```
                                The task function.                        The source file glob.
The destination for
the concatenated     function concatJS(){
script.                  var src = ["dist/**/*.js", "!dist/js/scripts.js"],
                         dest = "dist/js",
                         concatScript = "scripts.js";          The destination name of
                                                               the concatenated script.
Script data is piped into
the gulp-concat plugin.  return gulp.src(src)
                             .pipe(concat(concatScript))
                             .pipe(gulp.dest(dest))
                             .pipe(livereload());
                     }
                                                               Binds the concatJS task
                     gulp.task(concatJS);                      function to gulp.
```

The concatJS task will become interesting to use later, because it's dependent upon files
being processed by the uglifyJS task first. You'll use a special function when you later
define the watch and build tasks that will ensure that the uglifyJS task is run before the
concatJS task, because they are dependent on one another.

Another point of interest lies in the src file glob in that you want to *exclude* scripts.js,
which will be the file containing the concatenated scripts. If you don't make this exclusion,
the concat task will recursively bundle scripts.js every time it's run. This is obviously
not an optimal outcome, so you want to avoid it.

From here, this task proceeds similarly to the ones you've written before: You pipe the data read from the src file glob, process it with gulp-concat, and output it to the dist/js directory as scripts.js. Now you're ready to move onto your image-processing tasks.

Performing image optimization

As you recall from [chapter 6](#), you can save space if you're willing to optimize images. In most cases, you can do this without any noticeable drop in visual quality. Because image optimization can be incredibly tedious when done manually, a gulp plugin instance of imagemin called gulp-imagemin provides all of the functionality that you learned in [chapter 6](#).

In this section, you'll write two imagemin-related tasks: The main image-processing task that optimizes your PNGs, JPEGs, and SVGs, and a separate task that converts PNGs and JPEGs to WebP images. Let's start by writing the main image-optimization task that processes your standard image types by entering the contents of this listing into the gulpfile.

Listing 12.6. Optimizing your PNGs, JEPGs, and SVGs with `imagemin`

This task is more on the complicated side than some others, but is still relatively straightforward. You're reading all PNG, JPEG, SVG, and GIF files from the src/img directory and passing them to the gulp-imagemin plugin instance. imagemin has its own plugin defaults that it will go with if no plugins are supplied, but because an abundance of imagemin plugins are out there, I've selected some that I've found perform a bit better than the default.

When it comes to optimizing images with imagemin, it turns out that there is an obscene number of plugins available for every image format you can imagine. A list of plugins can be found at https://www.npmjs.com/browse/keyword/imageminplugin. Each has a wealth of options you can use to squeeze out every last drop of performance you can.

In this task, you rely on the imagemin-jpeg-recompress, imagemin-pngquant, imagemin-svgo, and imagemin-gifsicle plugins to optimize your images. When the images are optimized, they'll be written to the dist/img folder.

This takes care of your common image formats, but what if you want to leverage the WebP format? Turns out there's a plugin for that, and you've installed it: the imagemin-webp plugin. You can use this plugin to convert your existing PNG and JPEG image to WebP. To add this task to your build system, add this listing to the gulpfile.

Listing 12.7. The WebP conversion task

To try both these tasks, run the following command:

```
gulp imageminMain imageminWebP
```

With these tasks done, look in your dist/img folder and you'll see not only optimized images, but also WebP versions of them. Congratulations! You've written a task that converts all of your images for you on the fly, and it's also the last of the core tasks in the gulpfile.

In the next section, you'll write the build task that builds everything in the src directory for you, and the watch task, which watches your files for changes.

12.3.3. Writing the utility tasks

So you've written all of the core tasks for your build system. The meat on the bone, so to speak. These are the tasks that perform the heavy lifting for you: minification, uglification, image optimization, and building CSS—all of the important things you need.

Of course, you haven't really *automated* anything. These tasks, although useful, still require you to run *ad hoc* commands in your terminal to do anything. What you need are two more tasks:

- A task that watches files for changes, and when changes occur, runs tasks automatically and reloads the browser page for you
- A task that performs a clean build of all the site functions to the dist folder when the project is complete and ready for production

Let's start by writing the watch task!

Writing the watch task

As with other tasks, the watch task is defined via the gulp.task method. Inside it, though, you use a new method called gulp.watch. This method takes two arguments: a file glob pattern that specifies the files that are to be watched, and an array of one or more tasks that should run when changes in files are detected. The following listing shows the entirety of the watch task, which you define as the default task.

Listing 12.8. The `watch` task

This task is somewhat more linear than the ones you've written before. The first thing to note is that this task is bound to a label of default, which is a reserved label in gulp. Any task with this label doesn't need to be explicitly called by the gulp command. It's executed when the user enters gulp into the terminal in the same directory as the gulpfile that defines it.

Next, you tell the gulp-livereload plugin instance to start a server that listens for file changes. When a browser that's configured with a LiveReload plugin receives a signal from this server that a file has changed, it reloads the page.

The way you configure LiveReload for your browser depends on which browser you use. Chrome has a LiveReload extension that you can install by visiting the Chrome Web Store at https://chrome.google.com/webstore and searching for the LiveReload extension. When this plugin is installed, you'll see a small toolbar icon next to the address bar, as shown in figure 12.8.

Figure 12.8. The LiveReload extension icon in the Chrome toolbar. Clicking this icon enables the LiveReload listener that receives signals from the local LiveReload server to reload when files change.

LiveReload is also available for Firefox, Opera, and Safari. Search your browser's extension repository, or go to livereload.com for more information on alternative setup methods for unsupported browsers.

With your watch task written and the LiveReload extension installed for your browser, you can launch the watch task by entering the gulp command in your terminal. You'll see output in your terminal window that looks like this:

```
[22:36:46] Using gulpfile /private/var/www/ch12-gulp/gulpfile.js
[22:36:46] Starting 'default'...
```

Instead of being returned to the command line, the task listens for changes to files specified in the watch task function. To test this, you can run an http.js web server as you have in chapters past in the root folder to serve content from dist, and enable the LiveReload browser extension for that page. To do this, you can take code from any of the example web servers from earlier in the book, or clone the repo from https://github.com/webopt/ch12-gulp.git. Then in your text editor, modify files in the src directory and you'll see that tasks run automatically as you make changes. When a task finishes, piped calls to the gulp-livereload plugin instance in each task will signal the browser to reload the page.

You may take issue with this task blocking your ability to do anything else in the terminal. You could run this in the background, but depending on your terminal, you may not see any program output, and that's not ideal for development in case you run into errors. Open another terminal window if you need one, and if you need to quit gulp, press Ctrl+C and the program will stop.

A couple of new methods you'll notice are series and parallel. Both accept any number of tasks that you want to run, but the difference ends there. series runs the specified tasks one after the other, whereas parallel runs all of the specified tasks at the same time. You'll notice that you run the imagemin-related tasks in parallel when changes occur, and you run the uglifyJS and concatJS tasks in series because the concatJS task is dependent on the uglifyJS task.

You now have a fully automated workflow. Changes occur as you make them, and the browser automatically reloads the page for you to display your changes. This not only generates optimized web pages for you, but also increases your efficiency as a developer. *Now* you're cooking with gas. All that's left is to define two remaining tasks for performing builds, and you'll be set.

Writing the build task

The build task is by far the most succinct of any you've written so far. It's a small piece of code that accepts a name for the task, and specifies a set of tasks that you want to run in series. The build task looks like this:

```
gulp.task("build", gulp.parallel(minifyHTML, buildCSS, uglifyJS,
    imageminMain, imageminWebP, gulp.series(uglifyJS, concatJS)));
```

Writing the clean task

Sometimes you'll need to destroy the dist folder prior to performing a build. This could be because you have assets that you've created in src at one point but then removed, and

so still have some files in dist from previous builds that are orphaned. That's when you need to invoke a task to clean out the dist folder to perform *clean builds*.

Earlier, you installed a plugin by using npm named del. This isn't a gulp plugin *per se*, but rather a Node module that removes folders for you. Because of the nature of gulp, you can write any valid Node code and run it. The only caveat is that any code you write needs to return a Vinyl file object. If you're interested in going down this road, you'll need to learn a bit about Vinyl, which you can do at https://github.com/gulpjs/vinyl. For the purposes of this chapter, however, we'll eschew any exploration into Vinyl and continue.

Our clean task is another short and sweet one:

```
function clean(){
    return del(["dist"]);
}

gulp.task(clean);
```

The del module takes one argument, which is an array of one or more directories to delete. So now when you want to generate clean, pristine builds, you need to enter only two commands in your terminal window:

```
gulp clean
gulp build
```

This gives you a spotless build in the dist folder that's now production-ready. With this, you're fully automated and ready for any new web project that comes your way. Before bringing this book to a close, however, let's talk about the gulp plugin ecosystem, and point out a few other plugins that may be of use to you and your organization.

12.4. Going a little further with gulp plugins

Although you can execute any valid Node code inside a gulp task, it's clear that much of gulp's convenience and functionality is provided by the many gulp plugins that are available. I've shown you only a small handful of plugins that are available, and many more are out there for your consideration. You can peruse and use any of the 2,500 plugins available by going to http://gulpjs.com/plugins. This section highlights a few that caught my eye:

- *gulp-changed* is a plugin that allows you to process only files that have changed since the last build. This can be particularly useful for tasks that have a tendency to run for a long time, such as those that perform image-optimization. By processing only changed files, you can reduce build time, especially as you work.
- *gulp-nunjucks* is a plugin for Mozilla's Nunjucks templating engine. You can use it to do things as simple as separating your HTML into reusable pieces that you can programmatically import as partials (think something

similar to PHP's `include` and `require` functions.) Or, you can go full-on nuts with it and use it for templating and inserting content into HTML files by using a Handlebars-like syntax. This plugin is useful for developers who want to serve static site files but want to have the flexibility of some CMS-like features. Learn more about Nunjucks at https://mozilla.github.io/nunjucks.

- *gulp-inline* is a plugin that automatically inlines files for you. Although not a recommended practice for HTTP/2-capable servers, plenty of HTTP/1 clients and servers are out there yet that benefit from this useful (albeit hacky) performance improvement. This plugin allows you to maintain editability and modularity for assets destined for inlining, but handles the mundane part of that process for you.
- *gulp-spritesmith* is a plugin that generates image sprites from separate image files, and generates CSS for them. Although image sprites are an HTTP/2 no-no (because it's really concatenation, but for images), the practice provides performance benefits for HTTP/1.
- *gulp-sass* is a plugin that generates CSS from SASS files. We used LESS in this example, and perhaps you're not the biggest fan of it and prefer SASS instead. That's totally fine, and this plugin will accommodate your wishes. gulp-sass uses a syntax that's similar to gulp-less, so once you're familiar with one, you'll be familiar with the other.
- *gulp-uncss* is a wrapper for the `uncss` tool we used in chapter 3. It will remove the unused CSS from your project, only this time in an automated fashion!

There's likely a plugin for any tool that you can think of. Covering every single useful gulp plugin could be a book in itself, so we can't cover every one, obviously.

What if you can't find a gulp plugin for your task? If you know how to write the task by using JavaScript in Node, you can wrap it in a gulp.task and run it anyway. gulp doesn't limit you to plugins. If you want to help the community and write a plugin for a task that you feel is useful, go for it! Guidelines for writing gulp plugins are available in the gulp docs at http://mng.bz/109I. Now that you've had a chance to check out some other useful plugins in the gulp ecosystem, it's that time. We've hit the end of this chapter, as well as this book. Let's cap it all off with a summary of the things you've learned.

12.5. Summary

This chapter represents a milestone in your ability to apply your optimization knowledge to your projects. You can now automate common optimization tasks that otherwise would have taken you significant time to perform manually. As a part of this effort, these are the ideas and concepts that you picked up along the way:

- gulp is a streaming build system. Streams are a way for us to read data from a source on the disk, process and transform it, and then write the result back to the disk again. These streams are the foundations of gulp tasks.
- Folder structures help you organize your projects so that you can be more productive. With gulp, a proper folder structure can ensure that you separate your source files from the files that you deploy to a production server. This allows you to maintain editability while achieving the highest possible level of optimization.
- gulp doesn't explicitly rely on plugins to fulfill common tasks, but they add expediency in completing them. Knowing how to install plugins for your project enables you access to an entire ecosystem of tools that can boost your productivity.

- Writing gulp tasks takes little effort, and they're usually short. You can achieve a variety of goals with them, such as building CSS, minifying HTML, uglifying JavaScript, optimizing images, and anything else that you can think of. Beyond these foundational tasks, you can also write tasks that watch files for changes and automatically reload the browser for you whenever a file is changed.
- You can write utility tasks that build project files for you. These build tasks can help you to create a clean build of your site that you can take to production.
- gulp's plugin ecosystem is expansive, with over 2,500 plugins for your perusal. Whatever tasks you're seeking to accomplish, there's an extremely good chance a gulp plugin can help!

Your time spent with this book has taken you across many subjects. You've learned many ways to tune your site for higher performance, from winnowing down your CSS, writing leaner JavaScript, optimizing the delivery of your images and fonts, and more.

Increasing the performance of your website isn't merely a pursuit of convenience; it's also crucial to the user experience. By making your website better performing, you're making your website easier to access. When your site is easier to access, the user will stick around to see what it is you have to offer. Whatever the goal is, be it a larger readership or more sales for your e-commerce website, a faster website can only help you achieve that goal.

Wherever your goals take you, know that this book is only the starting point in your quest for a higher-performing website. The topic is so broad and shifting in focus that no book can cover the entire breadth of it, but some aspects of the discipline never really change. Reduce the footprint of your website as much as humanly possible, use the latest technologies (HTTP/2, for example), and rely on techniques that can give the perception of higher performance.

Good luck to you. May your websites always be lean, your network latency always be low, your rendering always be fast, and your goals always be within your reach.

Appendix A. Tools reference

Many tools are used throughout this book, and this appendix collects them all in one handy spot. Use this reference when you've completed this book but still need a reference for the tools you've used.

Note

Browser-based developer tools aren't listed in this appendix. You can invoke these tools in most browsers by pressing F12 on Windows systems, or Cmd-Alt-I on Mac systems. Tools are listed in the order they appear in this book.

A.1. Web-based tools

This section collects all of the web-based tools used throughout the book:

- TinyPNG (http://tinypng.com) An image optimizer on the web. Shrinks PNGs and JPEGs via a user-friendly interface.
- PageSpeed Insights (https://developers.google.com/speed/pagespeed/insights) Analyzes a URL and gives advice on how to improve page performance.
- Google Analytics (https://www.google.com/analytics) Provides data on your site visitors.
- Jank Invaders (http://jakearchibald.github.io/jank-invaders) Not so much a tool per se, but rather a game that helps you identify what jank looks like. Great for training the eye to catch sluggish animations.
- Mobile-Friendly Test (https://www.google.com/webmasters/tools/mobile-friendly) Analyzes a URL and reports whether its design is mobile-friendly.
- mydevice.io (http://mydevice.io) A comprehensive listing of devices, their screen resolutions, and pixel densities.
- VisualFold! (http://jlwagner.net/visualfold) A bookmarklet (by me!) that places guides on the page at locations you specify.
- Grumpicon (http://grumpicon.com) Generates PNG from SVG sprites (and a whole lot more).
- Can I Use (http://caniuse.com) A comprehensive listing of browser features and their level of support.

A.2. Node.js-based tools

This section collects all the packages that require Node.js. You can install any of these tools by using the npm install <package-name> syntax. You can also get info on any of these tools by searching for its name at https://www.npmjs.com.

A.2.1. Web servers and related middleware

- `express` (http://expressjs.com)
    - A small web server framework. Used throughout the book to spin up servers on localhost for example code.
    - `compression` (http://expressjs.com)
    - Provides gzip compression for Express-based web servers.
    - `shrink-ray` (https://www.npmjs.com/package/shrink-ray)
    - A fork of the `compression` Express middleware that supports Brotli compression.
    - `mime` (https://github.com/broofa/node-mime)
    - A module for detecting content types of files on the local filesystem.
    - `spdy` (https://github.com/indutny/node-spdy)
    - An HTTP/2-enabled web server module.

A.2.2. Image processors and optimizers

- `svg-sprite` (https://github.com/jkphl/svg-sprite) Generates SVG sprites on the command line.
- `imagemin` (https://github.com/imagemin/imagemin) An image-optimization library.
- `imagemin-jpeg-recompress` (https://github.com/imagemin/imagemin-jpeg-recompress) An `imagemin` plugin for reducing the size of JPEGs.
- `imagemin-optipng` (https://github.com/imagemin/imagemin-optipng) An `imagemin` plugin for reducing the size of PNGs.
- `svgo` (https://github.com/svg/svgo) A command-line utility for reducing the size of SVGs.
- `imagemin-webp` (https://github.com/imagemin/imagemin-webp) An `imagemin` plugin for converting images to WebP format.
- `imagemin-svgo` (https://github.com/imagemin/imagemin-svgo) `imagemin` wrapper for `svgo`.
- `imagemin-pngquant` (https://github.com/imagemin/imagemin-pngquant) Yet another `imagemin` plugin for reducing the size of PNGs.
- `imagemin-gifsicle` (https://github.com/imagemin/imagemin-gifsicle) An `imagemin` plugin for reducing the size of GIFs.

A.2.3. Minifiers/reducers

- `html-minify` (https://github.com/yize/html-minify) Minifies HTML files on the command line.
- `minifier` (https://github.com/fizker/minifier) Minifies CSS and JavaScript files on the command line.
- `uncss` (https://github.com/giakki/uncss) Removes unused rules from a CSS file by analyzing a website.

A.2.4. Font conversion tools

- `tt2eot` (https://github.com/fontello/ttf2eot) Converts TrueType fonts to Embedded OpenType.
- `tt2woff` (https://github.com/fontello/ttf2woff) Converts TrueType fonts to WOFF.

- `tt2woff2` (https://github.com/nfroidure/ttf2woff2) Converts TrueType fonts to WOFF2.

A.2.5. gulp and gulp plugins

- `gulp` (http://gulpjs.com) A streaming JavaScript task runner.
- `gulp-cli` (https://github.com/gulpjs/gulp-cli) The command-line interface for gulp.
- `gulp-util` (https://github.com/gulpjs/gulp-util) Utilities for gulp plugins.
- `gulp-changed` (https://github.com/sindresorhus/gulp-changed) A gulp plugin that checks for changed files.
- `del` (https://github.com/sindresorhus/del) Deletes files and folders.
- `gulp-livereload` (https://github.com/vohof/gulp-livereload) Automatically reloads the browser when files change on the disk.
- `gulp-ext-replace` (https://github.com/tjeastmond/gulp-ext-replace) Changes a files extension.
- `gulp-htmlmin` (https://github.com/jonschlinkert/gulp-htmlmin) Minifies HTML files.
- `gulp-less` (https://github.com/plus3network/gulp-less) A gulp plugin for compiling LESS files.
- `gulp-postcss` (https://github.com/postcss/gulp-postcss) A gulp plugin wrapper for PostCSS functionality.
- `gulp-uglify` (https://github.com/terinjokes/gulp-uglify) Uglifies JavaScript files. Uglification not only minifies JavaScript, but also reduces variable and function names to save the greatest amount of space possible.
- `gulp-concat` (https://github.com/contra/gulp-concat) Bundles multiple files into a single file.
- `gulp-imagemin` (https://github.com/sindresorhus/gulp-imagemin) A gulp plugin wrapper for `imagemin`.

A.2.6. PostCSS and PostCSS plugins

- PostCSS (https://github.com/postcss/postcss) A Node program that transforms CSS.
- Autoprefixer (https://github.com/postcss/autoprefixer) Automatically adds vendor prefixes for CSS properties.
- cssnano (http://cssnano.co) A CSS optimizer. It doesn't just minify; it reduces CSS via many focused optimizations.
- autorem (https://github.com/malchata/node-autorem) A PostCSS plugin (by me!) that changes `px` units in CSS to `rem` units.

A.3. Other tools

These tools don't belong under any other classification, but bear mentioning here:

- `csscss` (https://zmoazeni.github.io/csscss) A Ruby-based command-line tool that identifies redundancies in CSS.
- loadCSS (https://github.com/filamentgroup/loadcss) A library by the Filament Group for loading CSS without blocking rendering.
- Picturefill (https://scottjehl.github.io/picturefill) A polyfill for the `<picture>` element and the `srcset` and `sizes` attributes by Scott Jehl of the Filament Group.
- Modernizr (https://modernizr.com) A JavaScript feature-detection library. Can be customized to detect as many (or as few) features as necessary.

- fontTools (https://github.com/behdad/fonttools) A Python-based library of font utilities. Contains the font subsetting tool `pyftsubset`.
- Font Face Observer (https://github.com/bramstein/fontfaceobserver) A library by Bram Stein that controls the loading and display of fonts, similar in functionality to the browser-based Font Loading API.
- Alameda (https://github.com/requirejs/alameda) A compact AMD module/script loader built to use JavaScript promises.
- RequireJS (http://requirejs.org) An older (albeit more compatible) version of Alameda.
- Zepto (http://zeptojs.com) A lightweight jQuery-compatible alternative. This library is the most feature rich of all jQuery alternatives.
- Shoestring (https://github.com/filamentgroup/shoestring) An even lighter jQuery-compatible alternative by the Filament Group.
- Sprint (https://github.com/bendc/sprint) Yet another lightweight jQuery-compatible alternative that's *very* fast.
- `$.ajax` Standalone Implementation (https://github.com/ForbesLindesay/ajax) A standalone implementation of jQuery's `$.ajax` method.
- Fetch API (https://github.com/github/fetch) A polyfill for the Fetch API.
- Velocity.js (http://velocityjs.org) A `requestAnimationFrame`-driven implementation of jQuery's `animate` method. Provides fast animation in a familiar API.

Appendix B. Native equivalents of common jQuery functionality

In <u>chapter 8</u>, I discussed the importance of embracing minimalism in your website's JavaScript. A way of getting there is to drop jQuery altogether and use what's available in the browser. This appendix highlights some common jQuery functions and then shows you how to accomplish the same tasks by native means. This isn't a complete reference. Such a thing would be far beyond the scope of an appendix. This is to get you started.

B.1. Selecting elements

jQuery's core `$` method selects DOM elements by using a CSS selector string like so:

`$("div");`

This selects all `<div>` elements on a page. Generally speaking, whatever valid CSS selector that works in the `$` method will work with `document.querySelector` and `document.querySelectorAll` (excluding any custom selectors specific to jQuery). The difference between them is `document.querySelector` returns *only* the first matching element, whereas `document.querySelectorAll` returns *all* matched elements in an array object, even if only one element is returned. This listing shows examples, with return values annotated.

Listing B.1. Using `querySelector` and `querySelectorAll`

Although these two methods are useful, other methods are more widely supported and are faster when it comes to selecting elements. They're shown in <u>table B.1</u>.

Table B.1. jQuery versus native element selection methods

Selector	jQuery code	Native equivalent
ID	$("#element");	document.getElementById("element");
Tag	$("div");	document.getElementsByTagName("div");

Selector	jQuery code	Native equivalent
Class name	$(".element");	document.getElementsByClassName("element");

These core element selection methods are supported in virtually all browsers, and are great for selecting elements when their presence in the DOM is known and predictable. More-complex selections should be made using the aforementioned querySelector methods.

B.2. Checking DOM readiness

This is covered in chapter 8, but I'll recap it here for reference purposes. Before you can do anything in jQuery, you must check whether the DOM is ready. Otherwise, jQuery code will fail to execute properly. The common way to check for DOM readiness in jQuery is shown here:

```
$(document).ready(function(){
    // Code goes here
});
```

You can use an equivalent shorthand method if you want to save a few bytes:

```
$(function(){
    // Code goes here
});
```

If you prefer to use plain old JavaScript instead of jQuery, you can check for DOM readiness by listening for the DOMContentLoaded event via addEventListener, like so:

```
document.addEventListener("DOMContentLoaded", function(){
    // Code goes here
});
```

This is supported since IE9. In older browsers, though, you need to go a different route:

```
document.onreadystatechange = function(){
    if(document.readyState === "interactive"){
        // Code goes here
    }
};
```

If you're going to go without jQuery and you don't know which DOM readiness method to use, go with document.onreadystatechange. It has wide support and works much the same way as listening for the DOMContentLoaded event with addEventListener.

B.3. Binding events

Next to element selection, jQuery's biggest strength is its event-binding syntax. This little section shows common jQuery event-binding methods, and how to achieve similar functionality without jQuery.

This section *isn't* an event reference!

This section isn't an exhaustive reference of all events you can use in either jQuery or in native JavaScript APIs. For a reference of available events to plug into addEvent-Listener, check out this event reference from MDN: https://developer.mozilla.org/en-US/docs/Web/Events.

B.3.1. Simple event binding

jQuery listens for events on elements by using the bind method (which has been deprecated since jQuery v3 in favor of on, which is shown later in this section). Here's a simple example of executing some code when an element is clicked:

```
$(".click-me").bind("click", function(){
    // Click event code goes here
});
```

jQuery also has shorthand methods that look a bit cleaner than using bind. The following code accomplishes the same task:

```
$(".click-me").click(function(){
    // Click event code goes here
});
```

You can achieve the same thing by using querySelector and addEventListener like so:

```
document.querySelector(".click-me").addEventListener("click", function(){
    // Click event code goes here
});
```

Most of the time, you should be able to take the same event names used in jQuery's bind syntax and plug them into addEventListener, but don't assume that you can for anything but the most basic events. Check out the Mozilla Developer Network's web event reference for a list of events you can use.

B.3.2. Triggering events programmatically

Sometimes you'll have event code bound to an element that you want to trigger programmatically within your JavaScript code. Let's assume that you still have the same click event code bound to the .click-me element, and you want to run the click event code bound to it on demand. Using jQuery's trigger method, you can do that:

```
$(".click-me").trigger("click");
```

This statement will run the click event code bound to the .click-me element—nice if you need to run event code attached to an element on demand. You can accomplish the same task without jQuery via the dispatchEvent method.

Listing B.2. Triggering events programmatically without jQuery


```
var clickEvent = new Event("click");
document.querySelector(".click-me").dispatchEvent(clickEvent);
```

This syntax isn't as compact as jQuery's, but it works. You *could* shorten this by creating a helper function that eliminates the tediousness of creating new Event objects.

Listing B.3. An event-triggering helper function

```
function trigger(selector, eventType)
    document.querySelector(selector).dispatchEvent(new Event(eventType));
}
trigger(".click-me", "click");
```

The triggering function in the listing selects the element with a given selector, and triggers a specified event. It's not often that you'd need to trigger events outside the context in which they were bound to elements, but it's possible to achieve this functionality without jQuery.

B.3.3. Targeting elements that don't exist yet

jQuery can bind to elements that don't exist by using the on method. This is useful when you need to impart functionality to elements that don't exist now but *may* exist in the future. Here's an example of this method executing code on <li> elements within a <ul> element:

```
$(".list").on("mouseover", ".list-item", function(){
    // Mouseover code goes here
});
```

With this code, any .list-item elements added to the .list element in the future will still execute the code bound to the mouseover element. You can imagine all sorts of scenarios where this would be useful, and you can do the same thing without jQuery via the following code.

Listing B.4. Binding behavior to elements that don't exist yet without jQuery

```
document.querySelector(".list").addEventListener("mouseover", function(event){    ◄─┐
    if(event.target.className === "list-item"){    ◄─┐
        // Mouseover code goes here                                              The event is bound to the
    }                          Checks the targeted element                       target element's parent.
});                            for the desired class
```

Again, not as compact as jQuery, but functional. Of course, if you're targeting child elements with something other than a class, you may need to poke around in the event.target object for other methods to target child elements by. For example, you can use the event.target.id property to target elements by ID, or the event.target.tagName property to target elements by their tag name. It's not as convenient or compact as jQuery's syntax, but it works.

B.3.4. Removing event bindings

jQuery can remove bindings from an element by using the unbind and off methods like so:

```
$(".click-me").unbind("click");
$(".list").off("mouseover", ".list-item");
```

Like bind, unbind has been deprecated since version 3 of jQuery, so using off is preferable going forward. In either case, you can use removeEventListener in regular JavaScript to remove an event binding on an element:

```
$(".click-me").removeEventListener("click", boundFunctionReference);
```

When you remove an event binding with removeEventListener, you have to provide the function that you bound to it in the first place. In this case, boundFunction-Reference is a placeholder for the function you'd have bound to an element with addEventListener.

B.4. Iterating over a set of elements

jQuery gives you a super helpful method for iterating over a set of matched elements in the form of the each method. You can run it on any set of matched elements:

```javascript
$("ul > li").each(function(){
    $(this); // The current element in the iteration
});
```

Doing this without jQuery is easy. Just use a for loop as shown next.

Listing B.5. Iterating over a set of elements without jQuery

```javascript
var listElements = document.querySelectorAll("ul > li");
for(var i = 0; i < listElements.length; i++){ /
    listElements[i];
}
```

Selects all immediate children
Loops over all items in listElements
The current element in the iteration

Another way of looping over a set of matched elements also uses the for construct, but in a different way:

```javascript
for(var i in listElements){
    listElements[i]; // The current member in the iteration
}
```

Beware of this syntax, however: it loops over not only all elements in the set, but also object members such as the length property. Most of the time, you won't want to use this syntax, but you may conceive of scenarios where it's desirable.

B.5. Manipulating classes on elements

jQuery allows you to manipulate classes on elements by using the addClass, removeClass, and toggleClass methods.

Listing B.6. Manipulating element classes with jQuery

```javascript
$(".item").addClass("new-class");
$(".item").removeClass("new-class");
$(".item").toggleClass("new-class");
```

Adds the new-class class
Toggles the new-class class
Removes the new-class class

A native class-manipulation API called classList provides much of this functionality. The following are classList-driven equivalents of the jQuery methods shown previously.

Listing B.7. Manipulating element classes without jQuery

```
var item = document.querySelector(".item");     | Adds the new-class class
item.classList.add("new-class");           <----
item.classList.remove("new-class");        <------ Removes the new-class class
item.classList.toggle("new-class");        <--
                                               | Toggles the new-class class
```

You can also supply a condition as a second argument to the toggle method. If the condition evaluates to true, the class is added. If the condition evaluates to false, the class is removed.

Listing B.8. Conditionally toggling classes using classList

```
var enabled = true;
item.classList.toggle("enabled", enabled);     <------ Enabled class is added.
enabled = false;
item.classList.toggle("enabled", enabled);     <------ Enabled class is removed.
```

classList doesn't have universal support, however, and all versions of IE since IE10 only partially support it. For example, the second argument for the toggle method shown in the preceding code isn't supported in *any* version of IE. In this case, you can always manipulate the selected element's className property. You can easily add a class to an element just by concatenating a string to that property:

item.className += " new-class";

Removing/toggling classes is more involved. It usually involves using a regular expression or expanding a className property into an array and manipulating it that way. If you need to do more than simple class addition in the absence of classList, consider using a polyfill. One is available at https://github.com/eligrey/classList.js that weighs in at a little over 2 KB minified. Server compression can reduce the weight of this polyfill even further.

Sometimes you might need to check whether an element has a particular class. jQuery features the hasClass method, which affords you this ability:

$(".item").hasClass("item"); // Returns true

The easiest way to do this is to use the classList contains method.

Listing B.9. Checking for an existing class with classList.contains

```
document.querySelector(".item").classList.contains("item");      ◄──── Returns true
```

For browsers without classList support, use the polyfill noted previously. Otherwise, you could write your own code to check for the existence of a class.

B.6. Accessing and modifying styles

jQuery allows you to access and modify element styles by way of the css method. You can get or set a single CSS property in jQuery.

Listing B.10. Setting styles with jQuery

```
$(".item").css("font-size");                     Gets the item's current font size
$(".item").css("font-size", "1.5rem");      ◄──── Sets the item's font size to 1.5rem
```

You can also set multiple CSS properties on an element with the CSS method:

```
$(".item").css({
    color: "#f00",
    border: "1px solid #0f0",
    fontSize: "24px"
});
```

Note

When setting CSS properties on elements, remember that properties with dashes aren't expressed the same way when used in the context of an object. font-size becomes fontSize, border-bottom becomes borderBottom, and so on. The hyphen character isn't a legal character in variable names because it's a language operator. This convention also exists in native JavaScript, not only jQuery, so be wary.

Getting/setting styles without jQuery is more involved. If you want to retrieve a CSS property set on an element, you need to use the getComputedStyle method.

Listing B.11. Getting an element's style without jQuery

```
var item = document.querySelector(".item");
getComputedStyle(item).fontSize;          ⟵—— Returns the element's font size
```

Setting styles requires using the style object:

```
item.style.fontSize = "24px";
```

What if you want to set multiple styles? You *could* set them all at once via the HTML style attribute:

```
item.setAttribute("style", "font-size: 24px; border-bottom: 1px solid #0f0;");
```

This may seem a bit unsightly for some, but it's high-performing. If you don't care, and still want something a bit nicer looking, you could create a helper function like the one shown in the following listing that uses a similar syntax to the way jQuery's css method sets multiple CSS rules.

Listing B.12. A helper function for setting multiple CSS properties without jQuery

```
function setCSS(element, props){                      Iterates over the CSS
    for(var CSSProperty in props){      ⟵—┘          properties in the props object
        element.style[CSSProperty] = props[CSSProperty];  ⟵—┐
    }                                                           Sets the associated
}                          An element is passed              property by using
                           to the helper function.           the object's key
setCSS(document.querySelector(".item"), {   ⟵—┘
    fontSize: "24px",                         An object of CSS properties
    border: "1px solid #0f0",                 and their associated values is
    borderRadius: "8px"                       passed to the helper function.
});
```

You can *read* properties set by the style object, but they'll be populated only if they've been previously set. If they haven't been, you'll get an empty string. In this case, use getComputedStyle as shown previously.

B.7. Getting and setting attributes

jQuery's attr method allows you get and set attribute values like so.

Listing B.13. Setting attributes with jQuery

```
$(".item").attr("style");
$(".item").attr("style", "color: #0f0;");        ◁—— Sets the style attribute
```

Gets the style attribute's
current value

Setting them in plain old JavaScript couldn't be much simpler.

Listing B.14. Setting attributes without jQuery

```
var item = document.querySelector(".item");
item.getAttribute("style");                  ◁
item.setAttribute("style", "color: #0f0;");     ◁—— Sets the style attribute
```

Gets the style attribute's
current value

If you want to set multiple attributes in one go, you can use something similar to the code shown in listing B.8:

```
function setAttrs(element, attrs){
    for(var attr in attrs){
        element.setAttribute(attr, attrs[attr]);
    }
}

setAttrs(document.querySelector(".item"), {
    style: "color: #333;",
    id: "uniqueItem"
});
```

The setAttribute and getAttribute methods have nearly ubiquitous support, so they can be used without too much concern over compatibility.

B.8. Getting and setting element contents

jQuery has two methods for getting and setting element contents: html and text. The difference between the two is that html retrieves element contents with markup, and when used to set element contents, will treat markup literally. text strips out markup, and when used to set element contents, will treat text literally and encode any markup-related characters as HTML entities. The following listing shows how jQuery gets and sets element contents with these methods.

Listing B.15. Getting and setting element contents with jQuery

These methods have similar equivalents in native JavaScript: innerHTML and innerText, and they're used as shown in the following listing.

Listing B.16. Getting and setting element contents without jQuery

A caveat: innerHTML is considered a standard property, but innerText isn't (even though a lot of browsers support it). innerText is aware of styling in that if elements with another element are hidden by CSS, innerText won't include the hidden element's contents in its return value. If this is problematic for you, you can use the text-Content property instead.

Listing B.17. Setting an element's text content

innerHTML is the gold standard of getting or setting element contents if you want to include HTML, but if you're in doubt over which to use for getting or setting element text, default to textContent. It's well supported, with the exception of IE8 and below; innerText is supported in IE6 and above.

B.9. Replacing elements

jQuery has a replaceWith method that, unlike html, allows you to replace the entire element itself with whatever content you'd like, rather than just its inner contents:

$(".list").replaceWith("<p>I don't like lists.</p>");

With this code, the .list element is replaced with a new <p> element. As it turns out, an outerHTML element has been supported in browsers for a while that does the same thing:

```
document.querySelector(".list").outerHTML = "<p>I don't like lists.</p>";
```

Doesn't get much easier than that. outerHTML has been supported since IE4. Yes, you read correctly: Internet Explorer 4. Other browsers have supported it either since their inception, or many moons ago, so use it with confidence. An outerText property works like innerText, but replaces an element with whatever text you provide:

```
document.querySelector(".list").outerText = "<p>I don't like lists.</p>";
```

This replaces an element entirely with the supplied text, and will encode HTML characters so that they appear literally, with no interpretation made by the browser. Unlike outerHTML, however, outerText isn't a standard property. It's supported in every browser except for Firefox, so use it with care.

B.10. Hiding and showing elements

This one is super easy. jQuery has two methods for hiding and showing elements, named hide and show, respectively. They work like so:

```
$(".item").hide();
$(".item").show();
```

You can achieve the same functionality by using the element's style object.

Listing B.18. Hiding and showing elements with the style object

```
document.querySelector(".item").style.display = "none";    ◁──┐ Hides the element
document.querySelector(".item").style.display = "block";   ◁──── Shows the element
```

Of course, you should bear in mind that using a display value of block may not always be the best idea. Maybe you're toggling an element that uses a display type of flex, inline-flex, inline, or inline-block rather than block. In cases like these, it's best to have a global utility class that hides elements like this:

```
.hide{
    display: none;
}
```

Then you can just use a classList method to add or remove this class. When the hide class is added to an element, it will hide it. When it's removed, the element's original display property value will kick in. This prevents unintended layout problems.

B.11. Removing elements

Sometimes an element just needs to go. jQuery gives you a nice method for it named remove. It works like this:

```
$(".item").remove();
```

This removes every element with a class of item from the DOM. A method in native JavaScript goes by the same name and works similarly. Try this on for size:

```
document.querySelector(".item").remove();
```

The problem here is that querySelector returns only the first item that matches the query. You can use querySelectorAll, but the return value for it is an object array. Therefore, if you want to remove all items that match the query, you need to iterate over the set of matched elements as shown in the following listing.

Listing B.19. Removing multiple elements from the DOM without jQuery

```
var items = document.querySelectorAll(".item");       <—  All elements with
for(var i = 0; i < items.length; i++){     <—           a class of item
    items[i].remove();      <—                          are selected.
}
            Current item        Collection
            in iteration        of elements
            removed from        iterated over
            the DOM.            using a for loop.
```

The same code must be used with any element selection method that returns an object array (for example, getElementsByTagName, getElementsByClassName), not just querySelectorAll. Things are much more straightforward if you need to use only getElementById or querySelector, as you can call the remove method directly on the selected element rather than having to iterate over a set of them.

B.12. Going further

You can achieve a lot more via native JavaScript means in lieu of jQuery than what's documented here. A great place to check out is the You Might Not Need jQuery website at http://youmightnotneedjquery.com. It has a slew of code snippets of common (and not so common) jQuery behaviors and their native equivalents. It also allows you to specify the level of browser compatibility you need. If something is missing here, give that site a look. Failing that, look to Google! Someone out there has figured it out, and it's out there!

Index

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

T

U

X

Z

List of Figures

Chapter 1. Understanding web performance

Chapter 2. Using assessment tools

Chapter 3. Optimizing CSS

Chapter 4. Understanding critical CSS

Chapter 5. Making images responsive

Chapter 6. Going further with images

Chapter 7. Faster fonts

Chapter 8. Keeping JavaScript lean and fast

Chapter 9. Boosting performance with service workers

Figure 9.1. A service worker operating on its own thread labeled ServiceWorker Thread can be seen at the bottom in this view of Chrome's Timeline tool.

Figure 9.2. A service worker communicating as a proxy between a user and a web server. The user makes requests, which the service worker can intercept. Depending on how the service worker code is written, assets can be retrieved from the service worker's CacheStorage cache, or passed through to the web server. The service worker can also write to the cache in specific instances.

Figure 9.3. The service worker installation process. The code checks for the status of service worker support. If the browser supports it, the service worker is installed. If not, the browser does nothing.

Figure 9.4. The behavior that you want to occur when the service worker's install event is fired

Figure 9.5. The Application tab in Chrome's Developer Tools showing active service workers for the current site. Click the Service Workers item in the left pane to access this panel.

Figure 9.6. The v1 cache created by your service worker. You can see that the assets you've specified in the service worker's cachedAssets array are present.

Figure 9.7. Selecting the Offline check box in Chrome's Network panel allows you to simulate what it's like to be offline without having to disable your network connection.

Figure 9.8. The behavior of the service worker's fetch event. The user makes a request for an asset, and the service worker steps in to intercept it to see whether the asset is already in the cache. If not, the asset is fetched from the network, and the service worker caches it. If it's in the cache, it's pulled from the cache.

Figure 9.9. Network requests intercepted by the service worker will be indicated by a value of "(from ServiceWorker)" in the Size column in Chrome's network utility.

Figure 9.10. A comparison of the Time to First Paint performance of various caching scenarios on Chrome's Regular 3G throttling profile. The scenarios are an uncached page, the page when retrieved by the browser cache, and the page when retrieved from the service worker cache.

Figure 9.11. A two-pronged approach for intercepting a network request in a service worker's fetch event. If the requested asset is an HTML document, you always fetch it from the network and place it in the cache, and serve it from the service worker cache only if you're offline. If the resource isn't an HTML document, you always serve from the cache and retrieve it from the network if it's not in the service worker cache.

Chapter 10. Fine-tuning asset delivery

Chapter 11. Looking to the future with HTTP/2

downloading until the largest file (masthead.jpg) in the first batch finishes downloading. This problem can cause delays in load times.

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 12. Automating optimization with gulp

List of Tables

Chapter 7. Faster fonts

Chapter 10. Fine-tuning asset delivery

Chapter 12. Automating optimization with gulp

Appendix B. Native equivalents of common jQuery functionality

List of Listings

Chapter 8. Keeping JavaScript lean and fast

Chapter 12. Automating optimization with gulp

Appendix B. Native equivalents of common jQuery functionality