

# 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.

<pre>1 p{ 2   font-family: "Arial", "Helvetica", sans-serif; 3   font-size: 0.75rem; 4   font-style: italic; 5 }</pre>	<pre>1 p{ 2   font: italic 0.75rem "Arial", "Helvetica", sans-serif; 3 } 4 5</pre>
Longhand font properties	Shorthand font property (~35% smaller)

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

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:

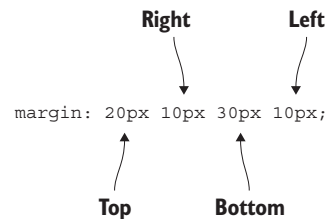
```
git clone https://github.com/webopt/ch3-css.git
cd ch3-css
npm install
node http.js
```

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.

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:



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

- *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.

### Overrides and shorthand properties

When overriding one part of a shorthand property for an element in a specific context, you might be tempted to copy the original shorthand and change only the value you need. This contributes to less-maintainable code and should be avoided. If an element has a property of `margin: 20px;` and you need to override the bottom margin for the same element in a new context, it's best to use the more specific `margin-bottom`. That way, any changes to the original context's `margin` value will be inherited by the new context.

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](http://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.

<pre>div.mainContent div.genericContent div.listContainer ul.genericList{   width: 202px;   margin-right: 12px;   float: left;   display: inline;   list-style: none; }</pre>	<pre>.genericList{   width: 202px;   margin-right: 12px;   float: left;   display: inline;   list-style: none; }</pre>
Too specific	Shallow (~82% smaller)

**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.

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{
  max-width: 1280px;
  width: 100%;

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

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

← The parent selector

Nested child selectors

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.

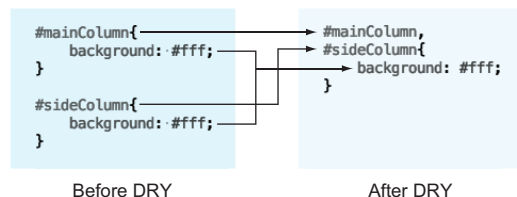
### 3.1.6 *Going DRY*

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

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.



**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.



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 pre-loaded 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

```
{.greyStrip} AND {.phoneNumber} share 5 declarations
- position: absolute
- position: static
- right: 0
- right: auto
- top: auto
```

Duplicate position property with  
conflicting redundant values

Duplicate right property with  
conflicting redundant values

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.

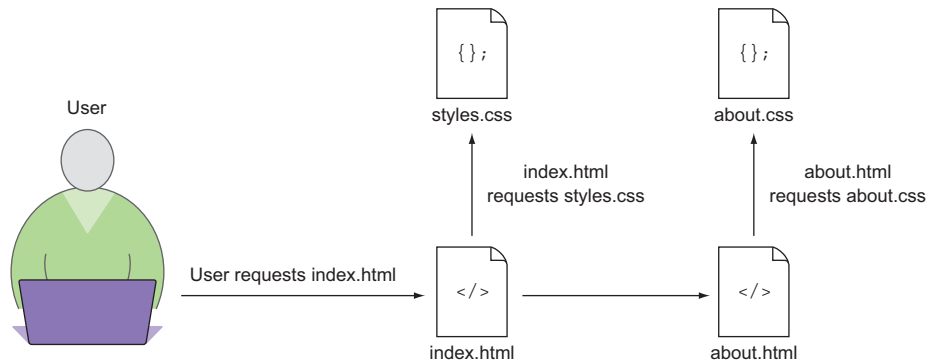
### 3.1.8 Segment CSS

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.

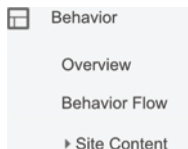
Serving your CSS this way can be a gamble, however, because your users may never navigate to subpages. A portion of your users would be forced to download CSS for pages they'll never see. This slows the initial visit to your site. The safer bet is to spread the weight across a few pages, but intelligently.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

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 navi-



**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.



Click the Behavior Flow option.

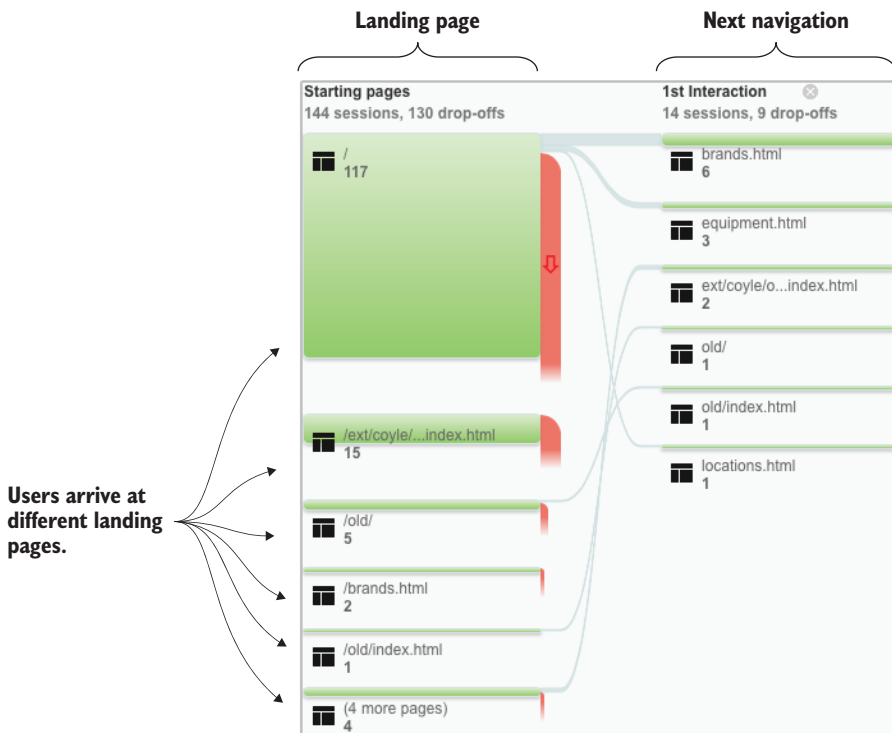
**Figure 3.6** The Behavior section of the left-hand menu in Google Analytics. The visitor flows can be seen by clicking the Behavior Flow link in the Behavior submenu.

gating to the Behavior section in the left-hand menu, and selecting the Behavior Flow option in the submenu, as illustrated in figure 3.6.

After clicking this option, the right-hand pane populates. In figure 3.7, 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.

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.



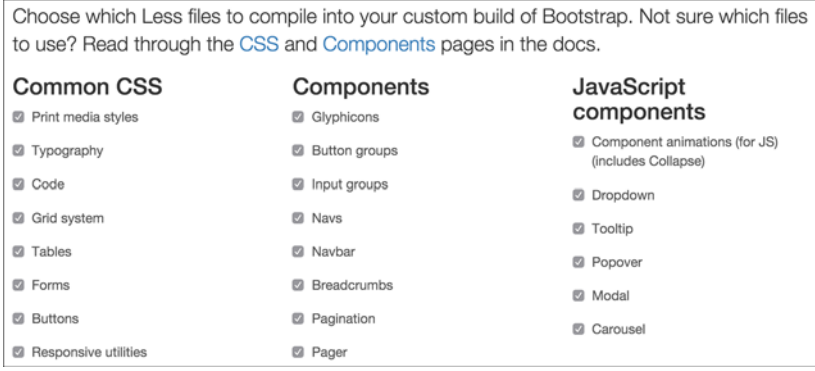
**Figure 3.7** The visitor flow chart in Google Analytics. Starting at the left, you see where users enter the site. In this case, you see that the vast majority of users are entering on the site's main page. Few visitors click through to the subpages.

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.



Choose which Less files to compile into your custom build of Bootstrap. Not sure which files to use? Read through the [CSS](#) and [Components](#) pages in the docs.

Common CSS	Components	JavaScript components
<input checked="" type="checkbox"/> Print media styles	<input checked="" type="checkbox"/> Glyphicons	<input checked="" type="checkbox"/> Component animations (for JS) (includes Collapse)
<input checked="" type="checkbox"/> Typography	<input checked="" type="checkbox"/> Button groups	<input checked="" type="checkbox"/> Dropdown
<input checked="" type="checkbox"/> Code	<input checked="" type="checkbox"/> Input groups	<input checked="" type="checkbox"/> Tooltip
<input checked="" type="checkbox"/> Grid system	<input checked="" type="checkbox"/> Navs	<input checked="" type="checkbox"/> Popover
<input checked="" type="checkbox"/> Tables	<input checked="" type="checkbox"/> Navbar	<input checked="" type="checkbox"/> Modal
<input checked="" type="checkbox"/> Forms	<input checked="" type="checkbox"/> Breadcrumbs	<input checked="" type="checkbox"/> Carousel
<input checked="" type="checkbox"/> Buttons	<input checked="" type="checkbox"/> Pagination	
<input checked="" type="checkbox"/> Responsive utilities	<input checked="" type="checkbox"/> Pager	

**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.

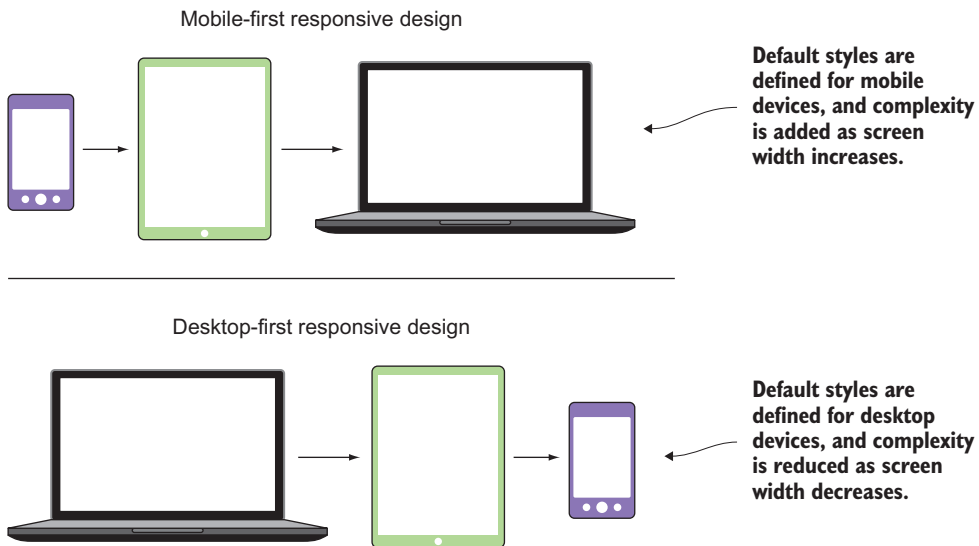
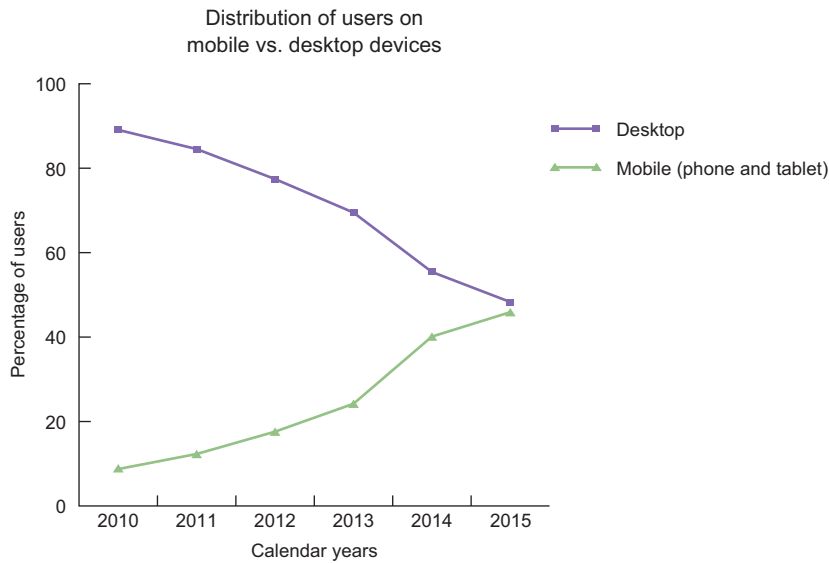


Figure 3.9 Mobile-first versus desktop-first responsive design flows

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).

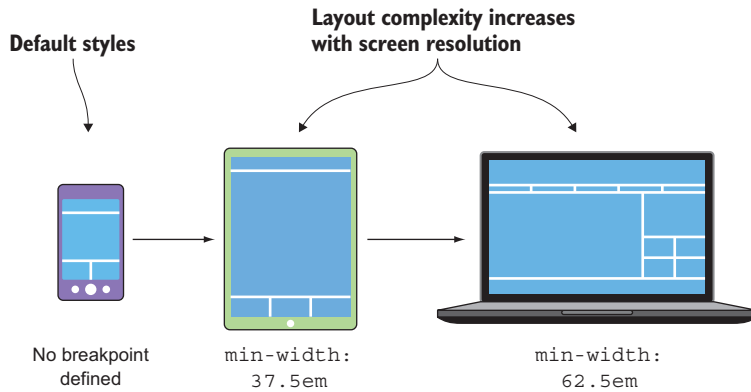
The advantage of mobile-first CSS is that you're serving CSS intended for the devices that are most likely to consume it. Because mobile devices often have less processing power and memory than desktop devices, the mobile device shouldn't have to apply desktop styles, interpret media queries, and then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

You'll notice in figure 3.11 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:

$$px / \text{default font size} = em$$



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

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

```

/* CSS resets should go here. */
html{
    font-size: 16px;
}
/* Mobile styles should go here. */

@media screen and (min-width 37.5em){ /* 600px / 16px */
    /* Tablet styles should go here */
}

@media screen and (min-width 62.5em){ /* 1000px / 16px */
    /* Desktop style should go here. */
}

```

Default mobile styles go first.

CSS resets go at the top of the document first.

Default font size set for the `<html>` element.

Tablet-specific styles

Desktop-specific styles



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 `<meta>` 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.



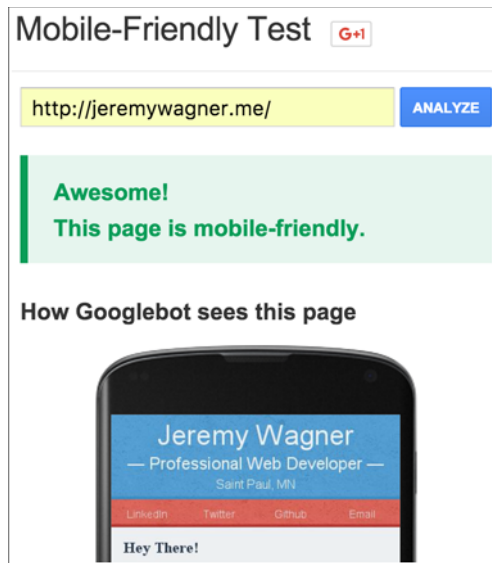
**Figure 3.12** A responsive site on a mobile device without the `<meta>` viewport tag (left) and the same site with it (right). Even though the site pictured is a mobile-first responsive site, it won't display in the proper breakpoint without this crucial tag in place, and the user will be forced to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

- *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



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

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.

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

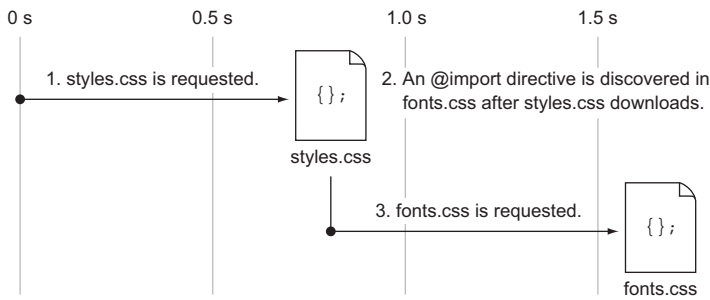
Beyond writing terse and mobile-first responsive CSS, it's vital that you tune your CSS to be as high performing as possible so that users will have a fast and smooth experience. Achieving this starts with applying a set of techniques designed to improve loading and rendering times.

#### 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.



**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`.

*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.

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");
```

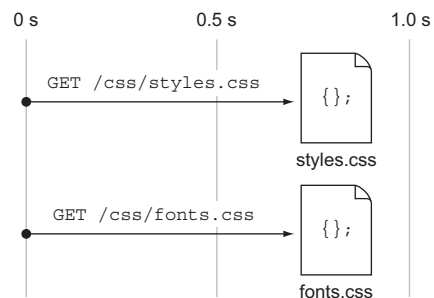
This contributes to a poor performance pattern; requests are serialized one after the other. This increases the overall loading and rendering time for a page. Ideally, you should seek to bundle as many files of the same type as you possibly can. But some CSS includes in your website can come from third 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 figure 3.15.

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



**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.

### 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.

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.

### 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 web-site without any CSS applied to it. Figure 3.16 shows this unsettling effect as the CSS is loaded too late in 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.

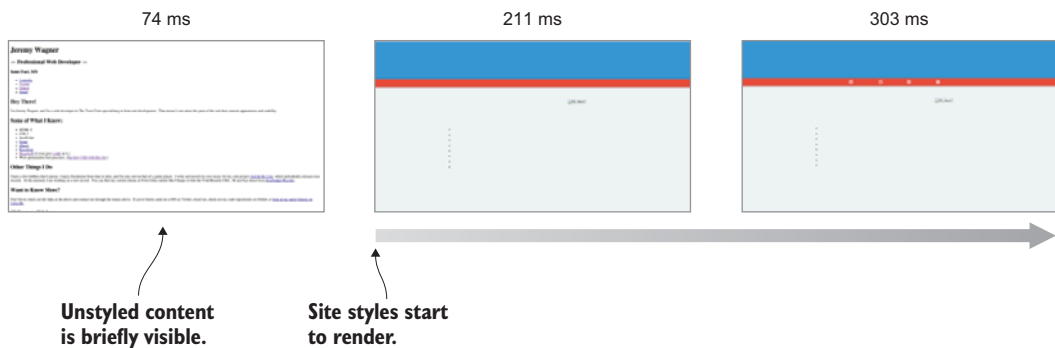
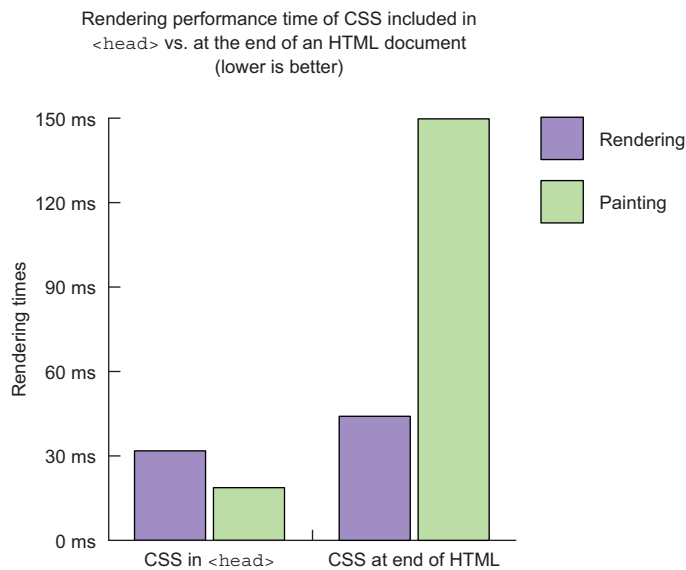


Figure 3.16 A rendering timeline in Chrome showing the Flash of Unstyled Content effect at left. The document eventually renders as intended, but with a brief display of the unstyled content. In this case, the effect is due to a <link> tag referencing a style sheet being placed at the end of the document.

### 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

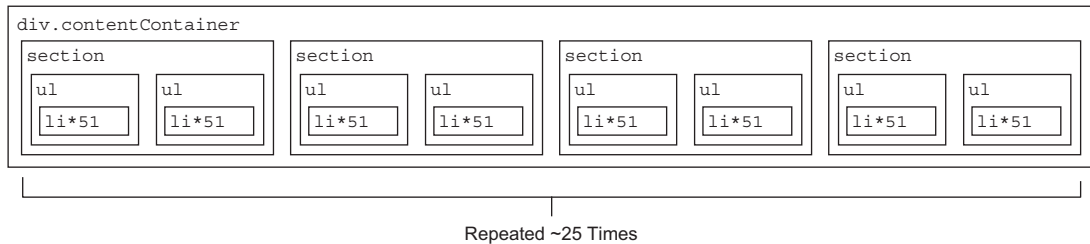
The payoff is big, and all because of a small HTML adjustment. If you're working on a site and you see `<link>` tags outside the `<head>`, relocate them to the `<head>` of the document.

### 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.

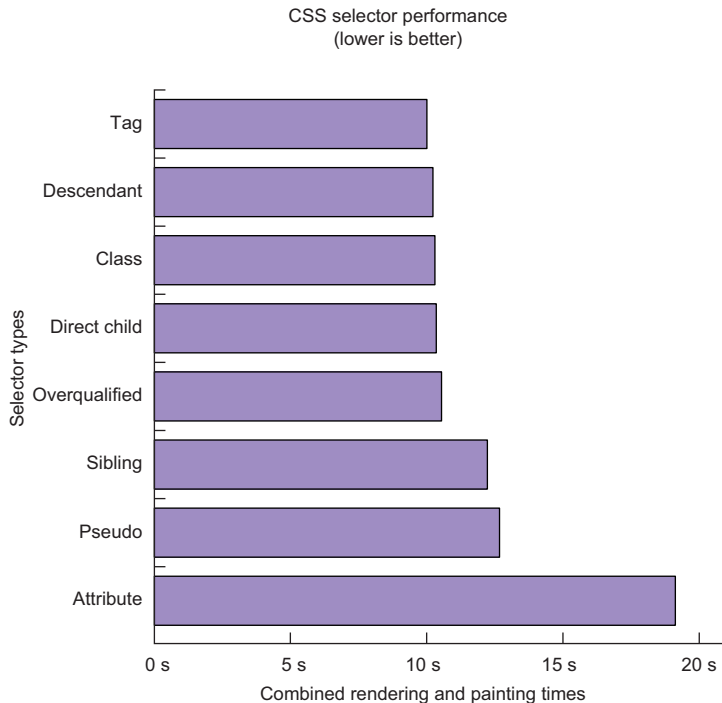
With the tests run and the results recorded, let's examine the results.

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

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

### 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**

```

.list{
    display: flex;
    justify-content: space-between;
    flex-flow: row wrap;
    margin: 0 auto;
    width: 100%;
}

.item{
    flex-basis: 24.25%;
    list-style: none;
    border: .0625rem solid #000;
    margin: 0 0 1rem;
}

```

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.

### Learn more about flexbox

This section isn't intended to be an exhaustive resource on flexbox, but rather to cover the performance benefits it can provide. For a quick primer on this layout engine, check out this excellent article by Chris Coyier: <https://css-tricks.com/snippets/css/a-guide-to-flexbox>.

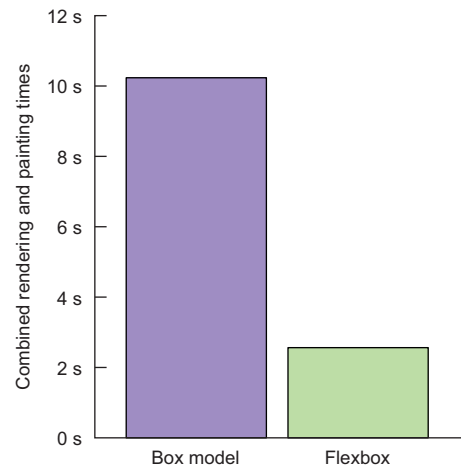
With the environment for the tests defined, let's check out the results!

### 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.

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.



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

## 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.

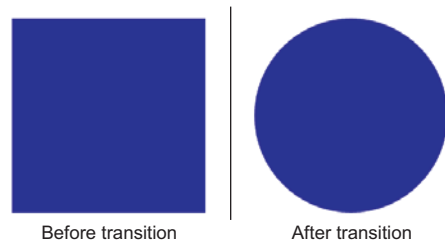
### 3.4.1 Using CSS transitions

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).

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.



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

#### Listing 3.9 Simple CSS hover state transition

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

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

← The transition property

← Trigger the 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 `5rem`s to `24rem`s. 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.

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

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

Transition type	jQuery animate()	CSS transition	Performance gain
	5.10 MB	2.32 MB	+54.51%
	2011.53 ms	1572.02 ms	+22%
	44.4	41.1	+8%

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.