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.

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.

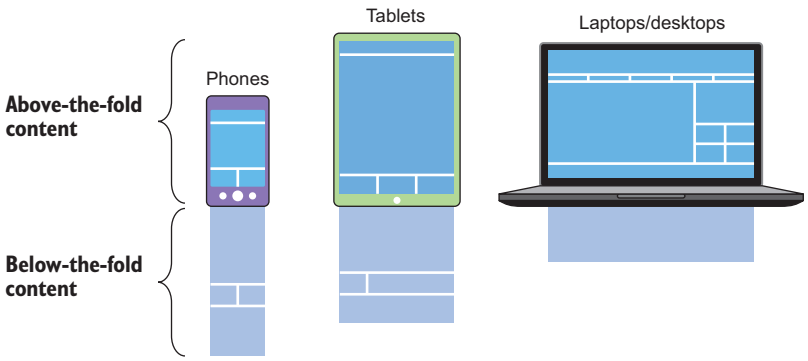


Figure 4.1 A depiction of above- and below-the-fold content on an array of devices. The above-the-fold content begins at the top of a website and ends at the bottom of the screen. Anything that's out of the browser's view is below the fold.

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.

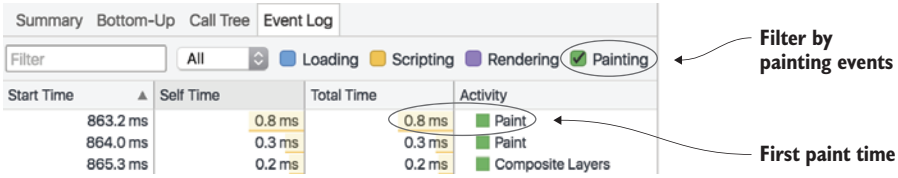


Figure 4.2 Chrome's Timeline profiler when the document's first painting event occurs. The event can be found under the Event Log tab by filtering out all but the painting events.

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.

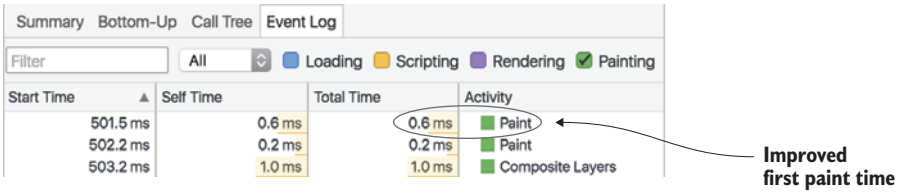


Figure 4.3 Chrome's Timeline profiler showing an improved paint time after the contents of the site's CSS have been inlined into the HTML

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.

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.

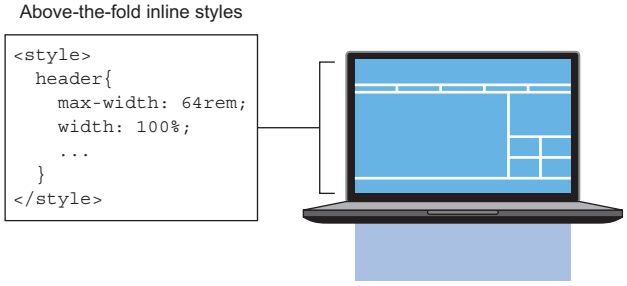


Figure 4.4 Inline styles loaded for above-the-fold content. The CSS for the above-the-fold content is inlined into the HTML for faster parsing, which translates into a faster Time to First Paint.

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.

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.

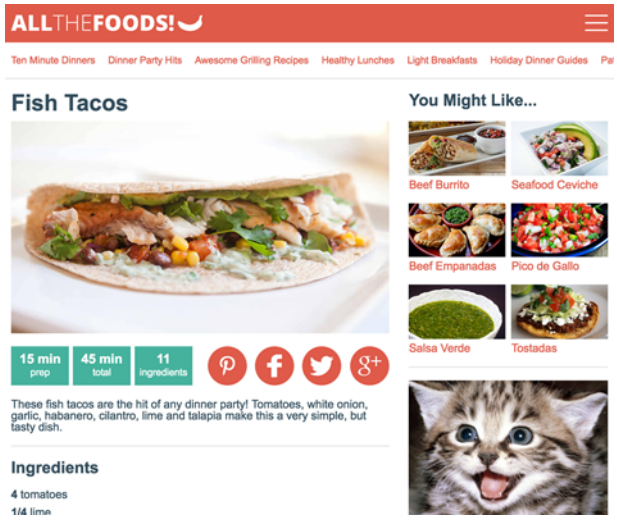


Figure 4.6 The recipe website in Chrome. This is the tablet breakpoint at roughly 750 pixels wide.

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.

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

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

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.

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.

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.

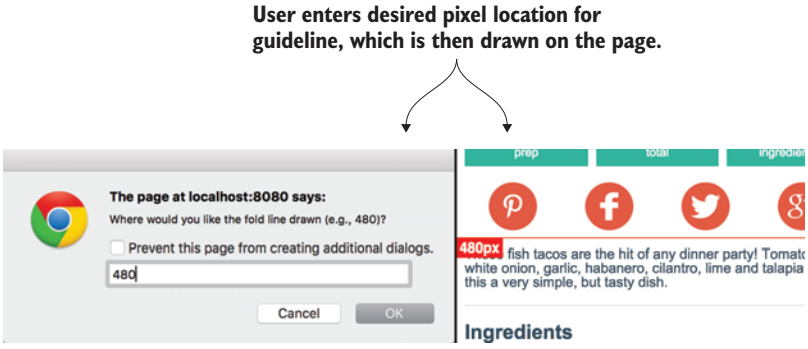


Figure 4.8 The VisualFold! bookmarklet in action. The user enters a number in a dialog box (left) indicating the desired location of a guideline to be drawn on the page (right). This assists the user in locating the fold. By resizing the window, the user can see how the content flows with respect to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

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/filamentgroup/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.

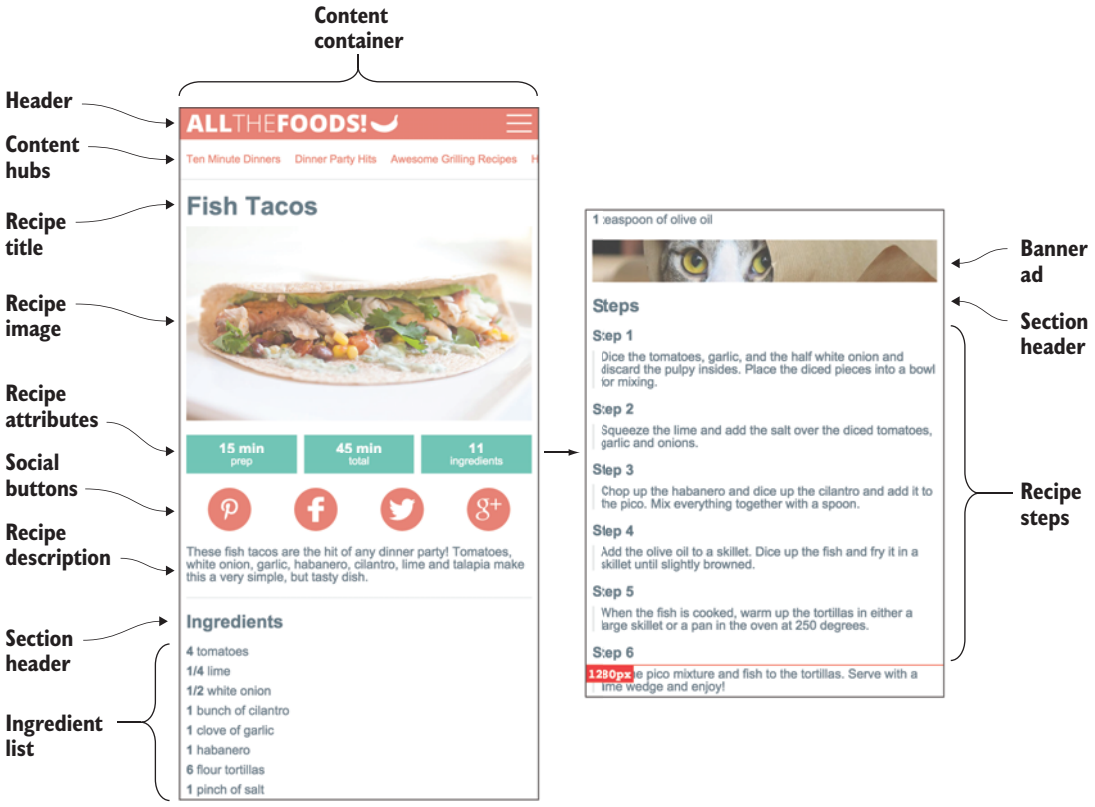


Figure 4.9 The mobile breakpoint of the page with labels of the critical components

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

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.

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.

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

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

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.

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.

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.

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 figure 4.12.

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.

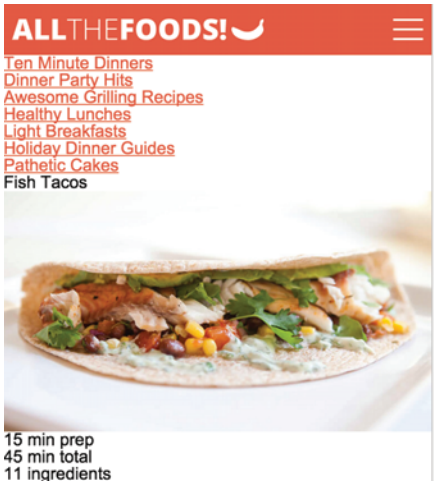


Figure 4.12 The critical CSS after all of the header styles have been inlined into index.html

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.

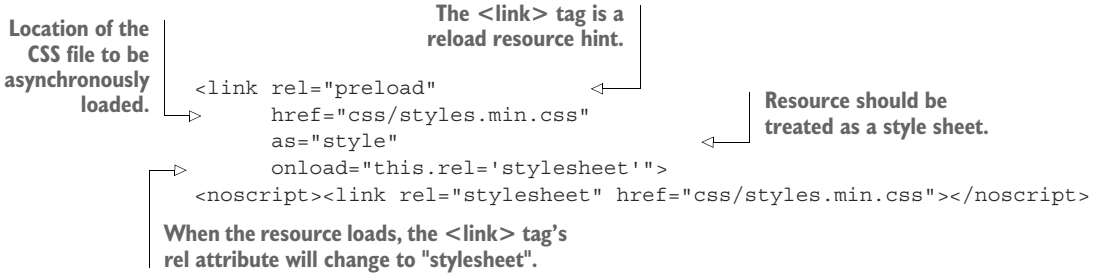
4.3.3 Loading below-the-fold CSS

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




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

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.

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

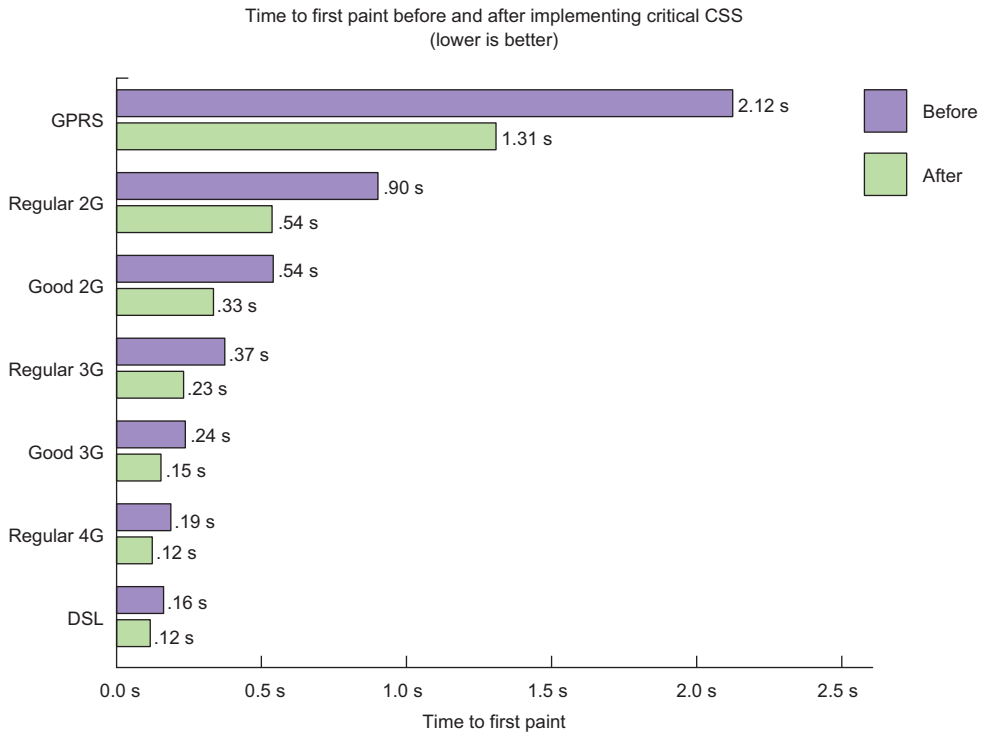


Figure 4.13 Time to first paint performance in Google Chrome before and after implementing critical CSS

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.

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.

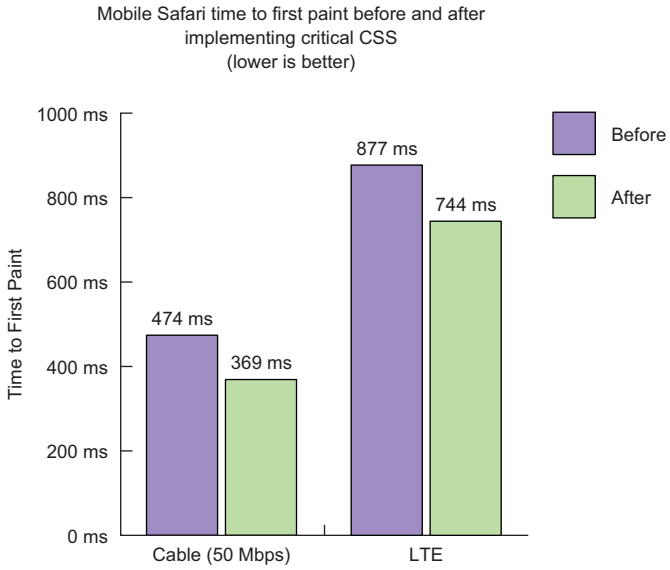


Figure 4.14 Time to First Paint in Mobile Safari on an iPhone 6S over a remote shared host before and after prioritizing critical CSS

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"));
    ?>
</script>
```

The preload resource hint polyfills are also inlined 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 figure 4.15, the focus is on modularity.

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.

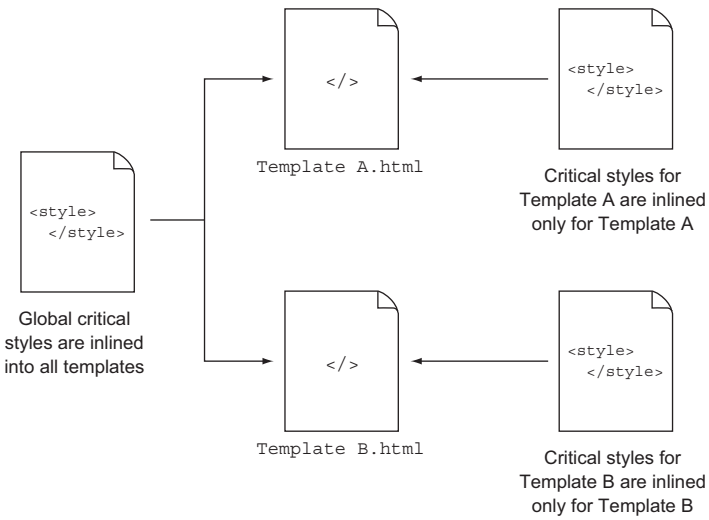


Figure 4.15 A modularized approach to critical CSS. Template A and Template B have their own critical CSS that's inlined only for those pages, but both inline globally common critical styles.

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.