

Using assessment tools



This chapter covers

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

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

2.1 Evaluating with Google PageSpeed Insights

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

2.1.1 Appraising website performance

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

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

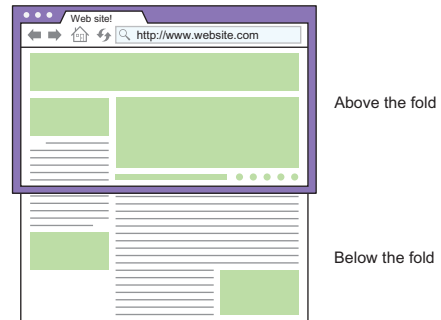


Figure 2.1 Google PageSpeed Insights checks two aspects of page speed: the load time of above-the-fold content, which is what the user sees immediately upon visiting a page, and the load time of the entire page.

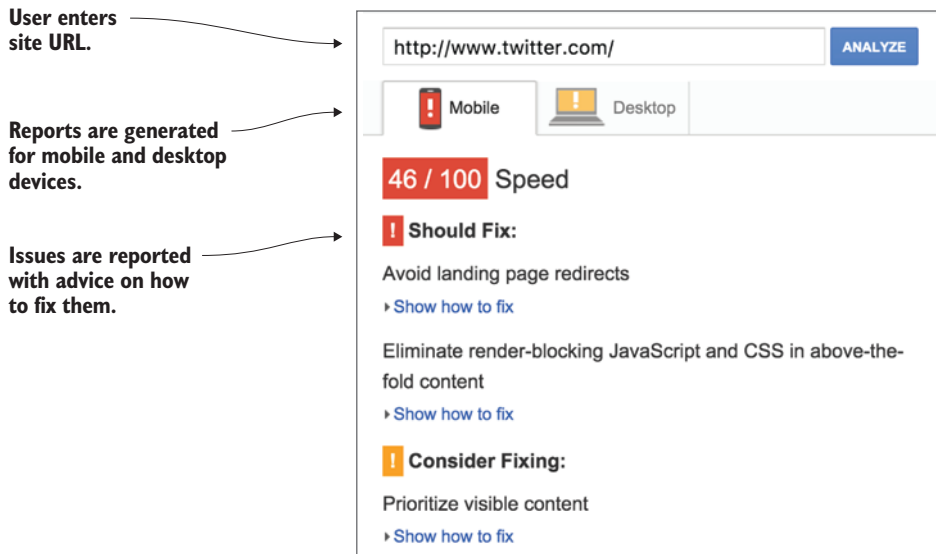


Figure 2.2 Google PageSpeed Insights results for the mobile view of a website. A user enters a URL and gets performance tips grouped by severity for both mobile and desktop states.

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

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

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

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

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

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

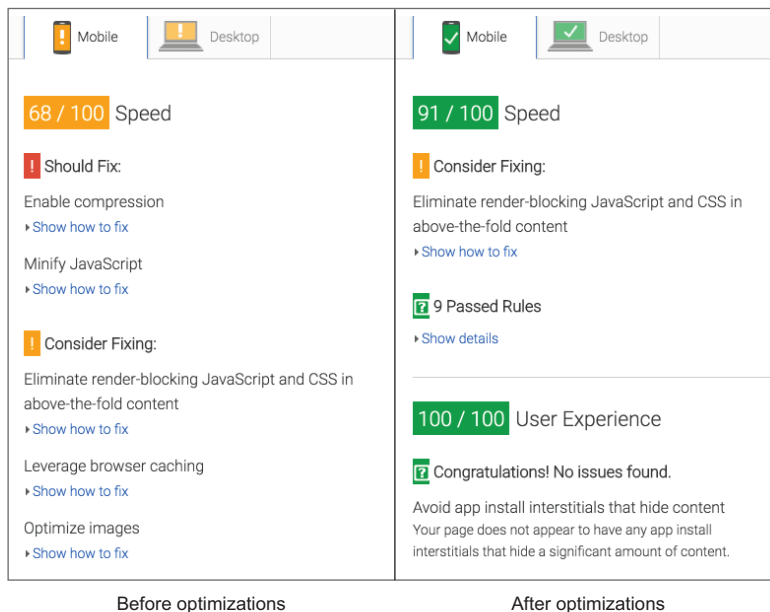


Figure 2.3 The PageSpeed Insights report for the client website from chapter 1 prior to (left) and after (right) your optimizations.

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

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

2.1.2 Using Google Analytics for bulk reporting

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

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

Legal implications

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

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

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

- *Page*—The URL of the page.
- *Pageviews*—The number of views a page has received in the reporting period. The reporting period is usually the preceding month but can be changed to a custom time period.
- *Avg. Page Load Time*—The average number of seconds the page has taken to load.

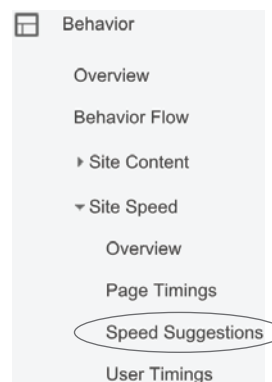
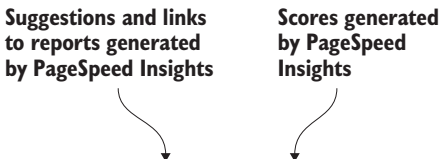


Figure 2.4 PageSpeed Insights reporting information can be accessed in Google Analytics by navigating to the Behavior section on the left menu and clicking the Speed Suggestions link.

- **PageSpeed Suggestions**—The number of suggestions PageSpeed Insights has to improve the performance of the associated page URL. Clicking this value directs you to a new window containing a PageSpeed Insights report for that specific URL.
- **PageSpeed Score**—The score given by the PageSpeed Insights report. This score is expressed in a range from 1 to 100, with lower scores indicating room for improvement, and higher scores indicating positive performance characteristics.









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

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

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

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

2.2 Using browser-based assessment tools

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

Similarities in developer tools across browsers

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

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

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

2.3 Inspecting network requests

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

2.3.1 Viewing timing information

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

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

To find out how long a request is taking, you'll look at how it's done in Chrome, which is similar to how this information can be found in most browsers (save for

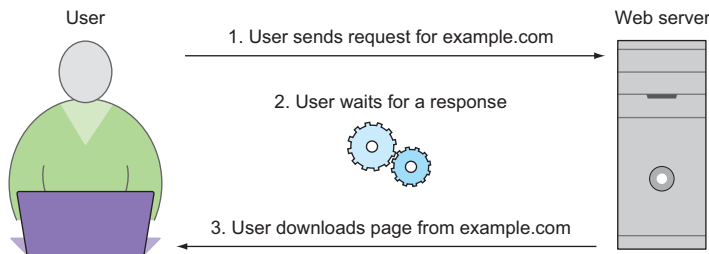


Figure 2.6 The process of a web browser's request to a web server. Latency occurs in each step of the process. The amount of time between the instant the user makes a request to the time the response arrives is known as *Time to First Byte* (TTFB).

Safari, which we'll get to later). To start, open the Network tab in the Chrome Developer Tools and follow these steps:

- 1 Populate the waterfall graph with data if you haven't already. You can get a detailed walk-through of this in chapter 1, but the easiest way is to reload the site while the developer tools are open and the Network tab is active.
- 2 After the waterfall graph is populated, you can click any of the asset entries and view the timing information for it.

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

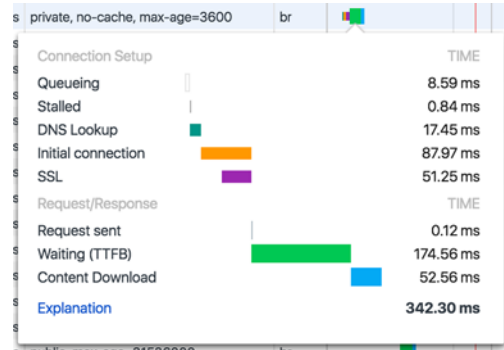


Figure 2.7 Timing information for a site asset. The TTFB in this example is 174.56 ms.

A note on DNS lookups

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

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

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

In the developer tools, you can click the Network tab and go to the optimized client website from chapter 1 at <http://jlwagner.net/webopt/ch01-exercise-post-optimization>. You'll see in figure 2.10 that the Safari version of the Network tab lacks a

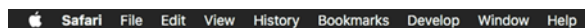


Figure 2.8 The Safari Developer Tools can be used only if the Develop option is visible in the menu bar when the Safari web browser window is in focus. If you don't see this menu, you have to enable the developer tools.

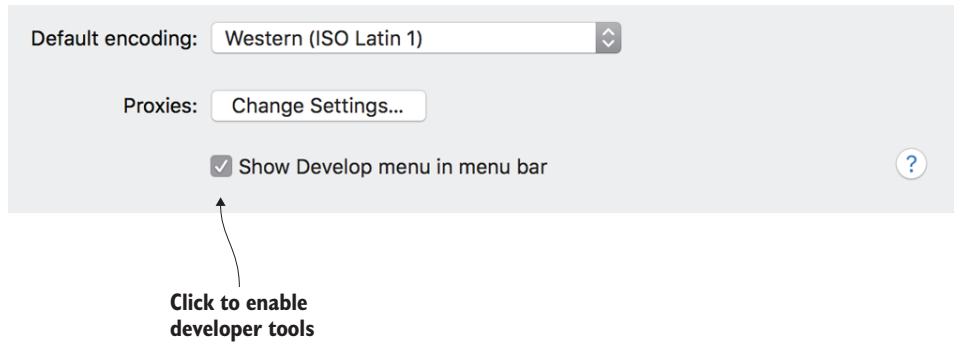


Figure 2.9 You can enable the Safari Developer Tools by choosing Safari > Preferences from the menu bar. In the window that appears, click the Advanced tab and select the check box.

Name	Domain	Type	Method	Scheme	Status	Cached	Size	Transferred	Start Time	Latency	Duration
ch01-exercise-post-optimization	jlwagner.net	Document	GET	HTTP	200	No	3.71 KB	4.10 KB	0ms	389.7ms	8.971ms
styles.min.css	jlwagner.net	Stylesheet	GET	HTTP	200	No	15.60 KB	3.13 KB	397.3ms	115.5ms	2.477ms
jquery.min.js	jlwagner.net	Script	GET	HTTP	200	No	84.41 KB	84.82 KB	398.2ms	200.3ms	165.4ms
behaviors.min.js	jlwagner.net	Script	GET	HTTP	200	No	1.66 KB	1.10 KB	398.3ms	195.2ms	0.394ms
bg@2x.jpg	jlwagner.net	Image	GET	HTTP	200	No	28.68 KB	29.03 KB	515.8ms	95.71ms	72.47ms
logo@2x.png	jlwagner.net	Image	GET	HTTP	200	No	24.63 KB	24.98 KB	519.3ms	150.6ms	94.19ms
brothers@2x.jpg	jlwagner.net	Image	GET	HTTP	200	No	29.07 KB	29.42 KB	520.5ms	153.2ms	90.57ms
states@2x.png	jlwagner.net	Image	GET	HTTP	200	No	3.46 KB	3.81 KB	520.7ms	156.9ms	0.385ms
favicon.ico	jlwagner.net	XHR	GET	HTTP	200	No	—	266 B	808.6ms	151.3ms	0.318ms

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

waterfall graph, but it does show a table of timing data. In this case, you have the Latency, Start Time, and Duration columns.

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

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

2.3.2 Viewing HTTP request and response headers

Another useful aspect of the developer tools across all browsers is the ability to inspect HTTP headers that travel with browser requests for content, and the responses from web servers. In figure 2.11, you see the typical request/response diagram, showing

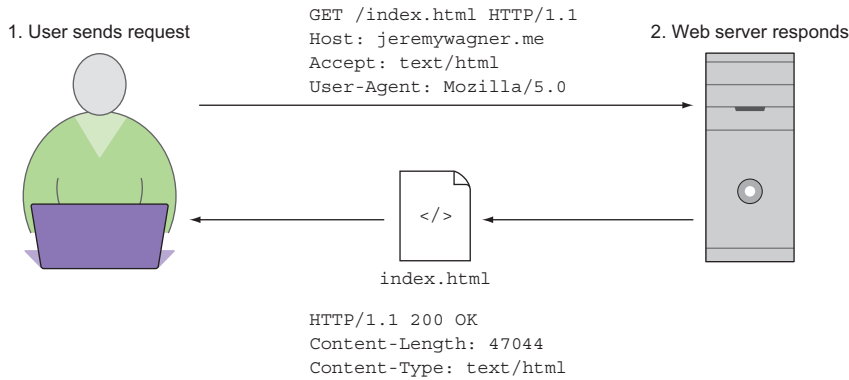


Figure 2.11 HTTP headers are sent by the browser in the initial request *and* by the server in its response. In this figure, a simplified set of headers is shown. The network inspection utilities in the developer tools for every browser allow the user to examine these headers.

sample headers that accompany the request and the response (albeit with much more brevity than in practice).

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

An example of a performance-related header is the Content-Encoding response header. This header tells you whether a resource is compressed by the web server.

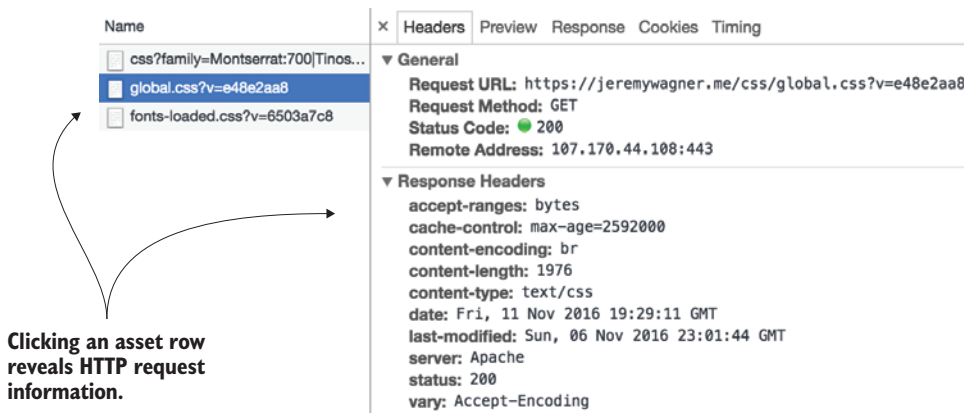


Figure 2.12 Viewing HTTP headers in Chrome's Developer Tools. Accessing HTTP headers for an asset can be done by clicking the asset name. A new pane to the right opens, with the header information contained within the Headers tab.

When you set up your own server, chances are good that you'll know whether compression is enabled. If you're working in an unfamiliar hosting environment and lack certainty, response headers are the place to check. Figure 2.13 shows response headers for `jquery.js` in the optimized client website.

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

Most tools in other browsers present this information in much the same way as Chrome does. Firefox's Developer Tools use the same flow as Chrome's. Microsoft Edge uses the same flow, but instead of opening the informational window on the right-hand side, it requires the user to open it explicitly by clicking a small toggle button, as seen in figure 2.14.

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

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

▼ Response Headers

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

Indicates a compressed asset

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

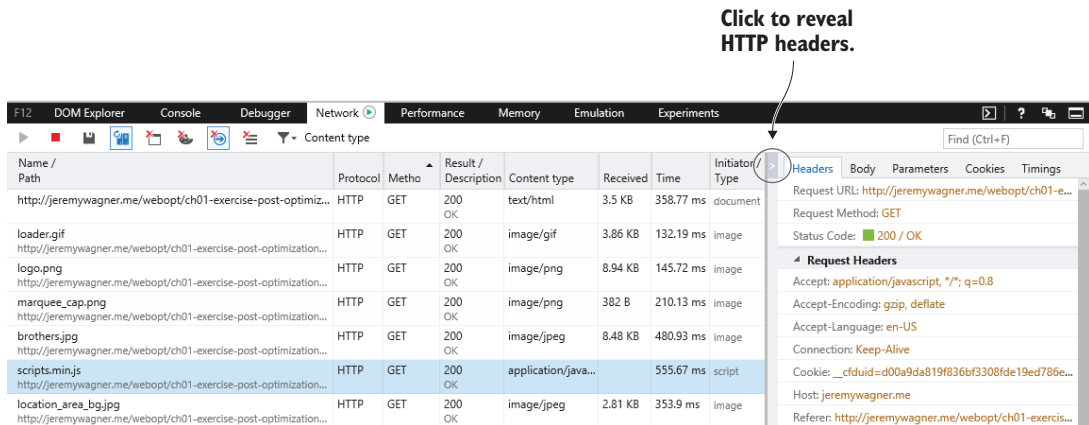


Figure 2.14 Viewing HTTP headers in Microsoft Edge requires the user to click a small toggle button at the far-right side of the window in the Network tab.

2.4 Rendering performance-auditing tools

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

2.4.1 Understanding how browsers render web pages

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

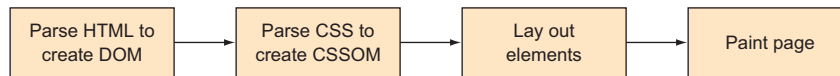


Figure 2.15 The page-rendering process.

In detailed terms, the steps in this process are:

- 1 *Parse HTML to create the Document Object Model (DOM)*—When the HTML is downloaded from the web server, it's parsed by the browser to build the DOM, which is a hierarchical representation of the HTML document's structure.
- 2 *Parse CSS to create the CSS Object Model (CSSOM)*—After the DOM is built, the browser parses the CSS and creates the CSSOM. This is similar to the DOM, except it represents the way that CSS rules are applied to the document.
- 3 *Lay out elements*—The DOM and CSSOM trees are combined to create a render tree. The render tree then goes through the layout process, where CSS rules are applied and elements are laid out on the page to create the UI.
- 4 *Paint page*—After the document has finished the layout process, the cosmetic aspects of the page are applied from the CSS and media in the page. At the end of the painting process, the output is converted into pixels (rasterized) and displayed on the screen.

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

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

2.4.2 Using Google Chrome's Timeline tool

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

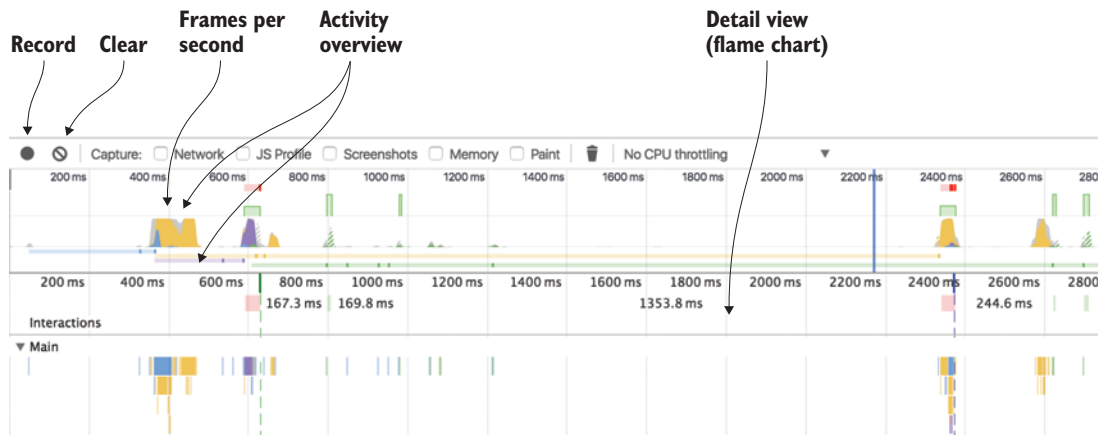


Figure 2.16 The Timeline tool in the populated state.

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

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

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

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

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

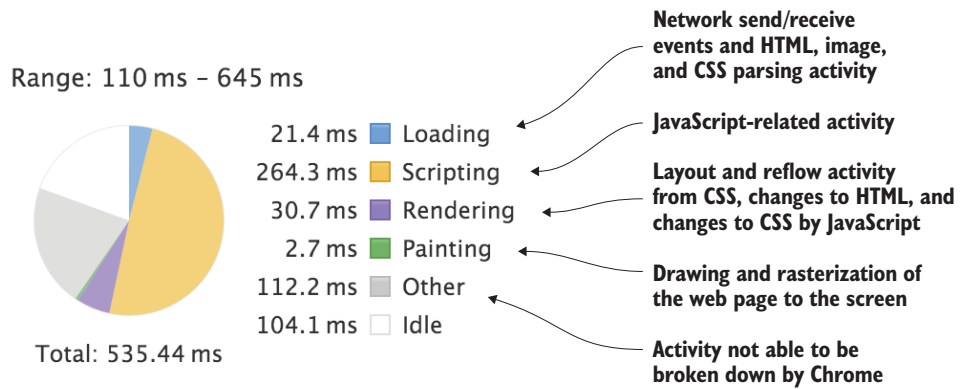


Figure 2.17 The breakdown of session activity as recorded by the Timeline tool

Narrowing things down

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

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

Keeping tabs on visible pages

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

Now onto the flame chart itself. A *flame chart* is a kind of chart used to represent the events that occur in a computer program. With Chrome's Timeline tool, it arranges this data in a call stack. With respect to flame charts, a *call stack* is a hierarchical representation of recorded page activity. Figure 2.18 shows a call stack of the client website's HTML parsing and of the activity that originates from it.

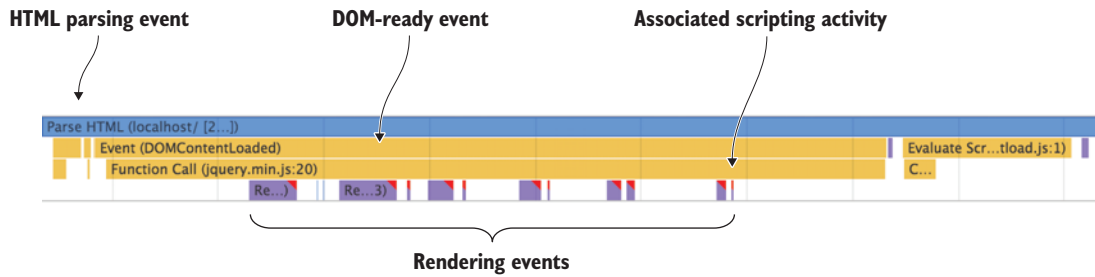


Figure 2.18 An isolated call stack from the flame chart view in the Timeline tool. The top event is a loading event where the HTML was parsed. Underneath it are events originating from it, such as the `DOMContentLoaded` event that fires when the DOM is ready, and scripting and rendering events.

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

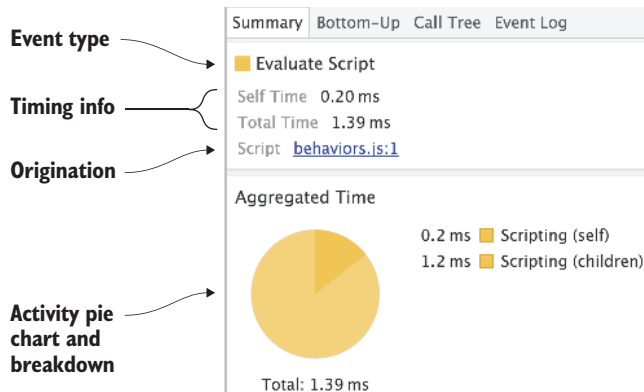


Figure 2.19 The breakdown of a scripting event. You can see information related to the event, such as the amount of CPU time used, the event type, and its origination. This data is also visualized in a pie chart.

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

2.4.3 Identifying problem events: *thy enemy is jank*

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

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

So what causes jank? It occurs when too much CPU time is consumed during a single frame. A *frame* is the amount of work the browser does in one frame per second of display time. When I say *work*, I'm talking about the events described earlier, such as loading, scripting, painting, and rendering.

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

Spotting jank

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

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

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

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

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

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

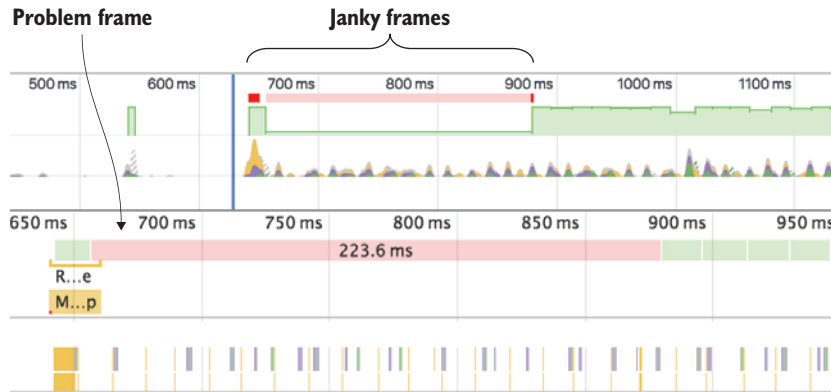


Figure 2.20 A timeline recording of the modal opening on the client website. A range of janky frames is denoted with red markers in the activity overview, and highlighted in red and clickable in the flame chart.

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

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

In this case, the average FPS of this frame is a measly 3. That needs fixing, sure, but how can you figure out what's causing the issue? You know that the modal uses an animation to slide into position, so maybe it's something to do with that. To drill down a bit more, you can click the Event Log tab shown in figure 2.21. When you click the event log, you'll see every single event related to the frame, as seen in figure 2.22.

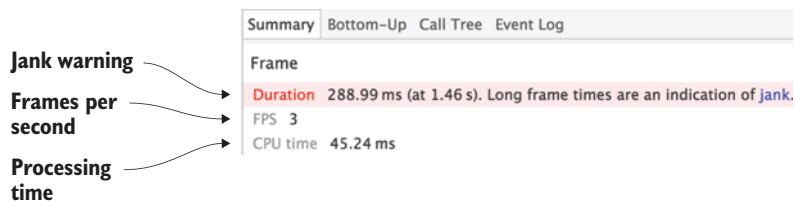


Figure 2.21 The summary view of a janky frame. Note the explicit warning and the low frame rate.

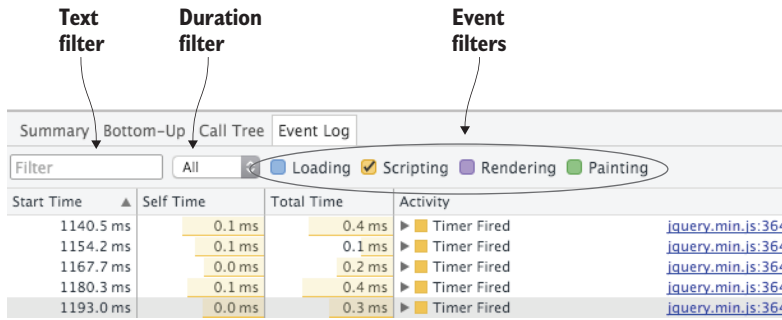


Figure 2.22 The event log filtered by scripting events. The text box can be used to filter events by the contents of their activity, filtered by a specific length of time in the duration drop-down, and/or by type.

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

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

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

If you don't know anything about `CSS transitions`, don't worry. They're covered in detail in chapter 3.

Want to skip ahead?

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

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

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

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

```
transform: translateY(-150%);
```

- 2 Add this property on the next line:

```
transition: transform .5s;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

with this code:

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

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

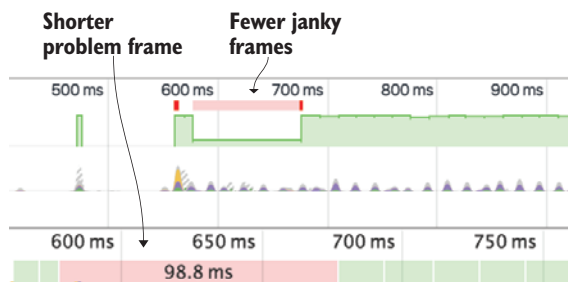


Figure 2.23 Modal animation performance after CSS transitions have been implemented. Janky frames still exist, but much less so than before, resulting in an overall improved experience.

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

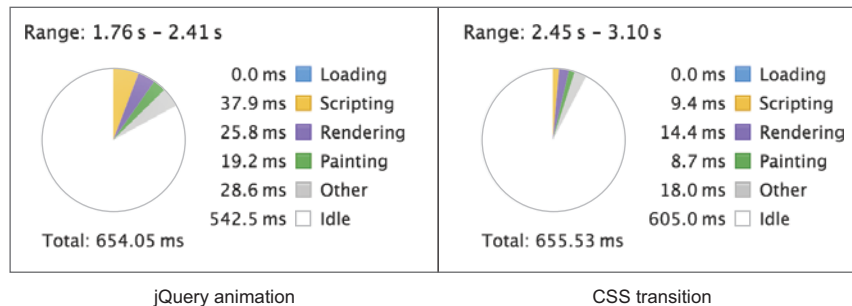


Figure 2.24 CPU usage summary of jQuery animations (left) compared to CSS transitions (right)

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

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

2.4.4 Marking points in the timeline with JavaScript

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

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

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

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

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

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

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

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

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

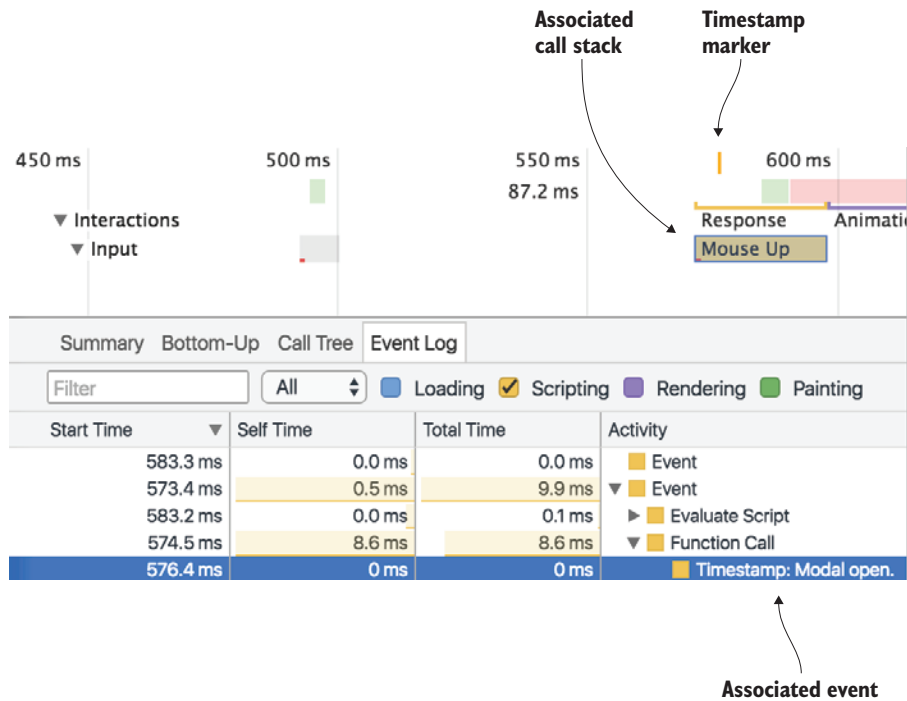


Figure 2.25 A marker added to the timeline. The associated call stack is selected, and the timestamp event call is shown in the event log.

2.4.5 Rendering profilers in other browsers

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

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

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

A big plus is that all browsers covered in this chapter support the `console.timeStamp` method for marking the timeline. So no matter what tool you use, you can label



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

a point in the session to help you find the activity you're looking for. Next, you'll learn how to use the console object in Chrome to benchmark snippets of JavaScript code.

2.5 Benchmarking JavaScript in Chrome

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

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

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

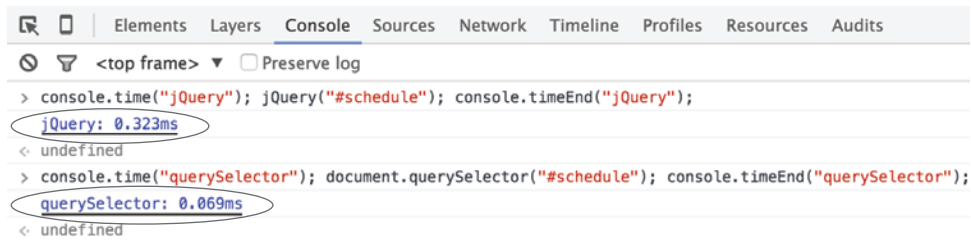


Figure 2.27 The results of two benchmarks you’ve run of jQuery’s DOM selection versus that of the native `document.querySelector` method. Results are circled.

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

- 1 `console.time("jQuery"); jQuery("#schedule"); console.timeEnd("jQuery");`
- 2 `console.time("querySelector"); document.querySelector("#schedule"); console.timeEnd("querySelector");`

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

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

Benchmarking tip

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

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

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

2.6 Simulating and monitoring devices

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

2.6.1 Simulating devices in the desktop web browser

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

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

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

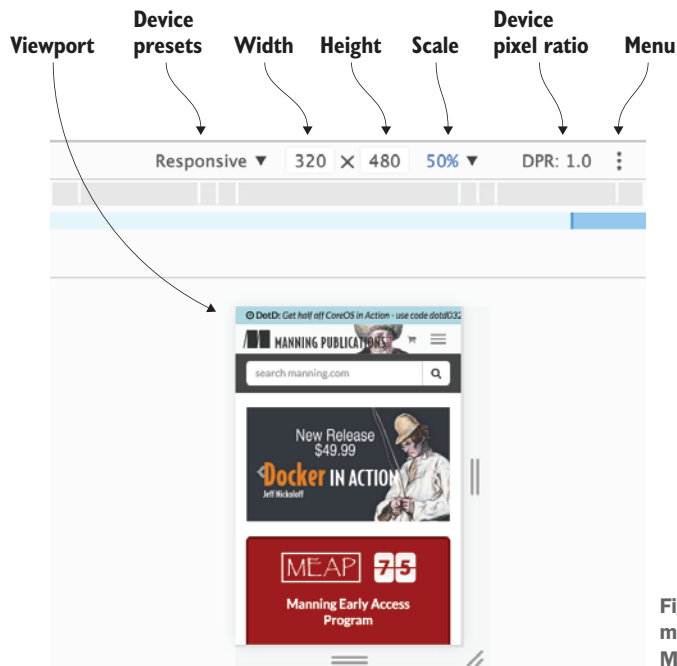


Figure 2.28 The device simulation mode in Chrome viewing the Manning Publications website

Other web browsers have similar utilities. Safari has an iOS-centric device simulation utility called Responsive Design Mode. You invoke this mode from the Develop menu by clicking Enter Responsive Design Mode, or by hitting Alt-Cmd-R. This utility is similar to Chrome's in capability, but with a different UI. Firefox's Responsive Design Mode is similar to Chrome's, but with fewer options overall. Edge is similar as well, and focuses on simulating Microsoft-centric mobile devices and Internet Explorer.

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

2.6.2 Debugging websites remotely on Android devices

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

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

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



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

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

- 1 *Enable the developer options on the Android device*—This entails choosing Settings > About Device and tapping the build number field seven times (seriously).
- 2 *Enable USB debugging*—On the Android device, choose Settings > Developer Options and then select the USB Debugging check box.
- 3 *Allow device authorization*—In Chrome on your desktop, go to the URL <chrome://inspect#devices> and ensure that the Discover USB Devices check box is selected. This enables you to receive an authorization request inside Chrome on the attached device. Tap OK to accept it.
- 4 *Inspect the web page open on the device*—After a device appears in the device list as shown in figure 2.29, click the Inspect link underneath the device in the list.

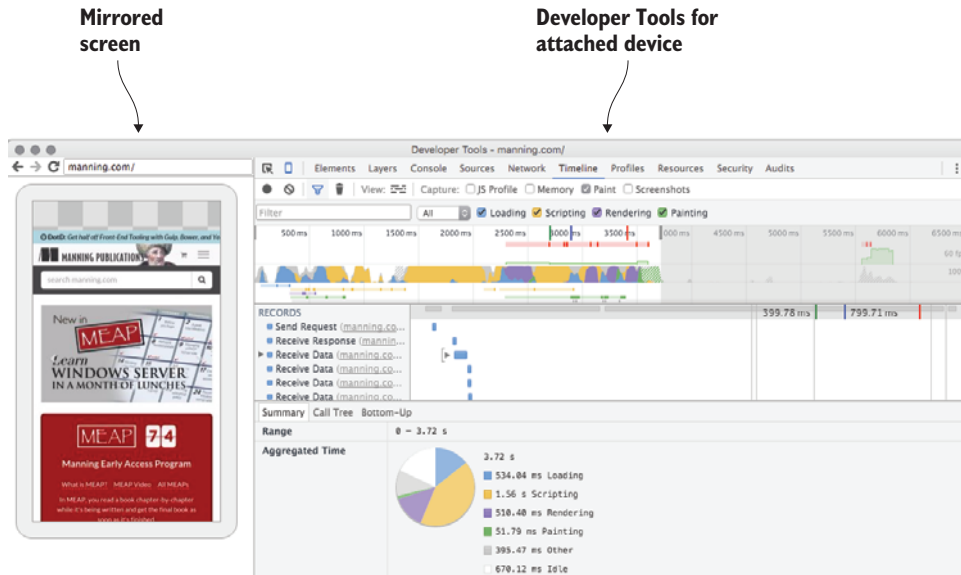


Figure 2.30 The Developer Tools profiling rendering activity of a page on an Android phone. In this view, the device's display is mirrored on the host machine, and the Developer Tools are focused on the device's current page rather than a session active on the desktop.

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

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

Quick tip

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

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

2.6.3 Debugging websites remotely on iOS devices

You can also debug pages on iOS devices, and it's simpler than remote debugging websites on Android phones. First, connect your iOS device to your Mac with a USB cable,

and instead of using Chrome, you'll use Safari on both the desktop and the mobile device. After you have Safari open on both, go to www.manning.com on the attached device and follow along:

- 1 *Authorize your Mac to access your device*—In Safari on your Mac, go to the Develop menu and you'll see the name of your iOS device (for example, Jeremy's iPhone). Underneath that menu, you'll see the Use for Development option. Click this option and you'll see a prompt on your iOS device to trust the computer that it's connected to. Tap the Trust option to allow your Mac to communicate with the device.
- 2 *Inspect the web page open on the device*—After authorizing your Mac, go back to the Develop menu, choose your device, and in that submenu you'll see a list of the web pages that are open in Safari on the attached device. Choose the device that's focused on the Manning Publications page.

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

2.7 Creating custom network throttling profiles

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

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

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

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

This screen displays the following set of fields:

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

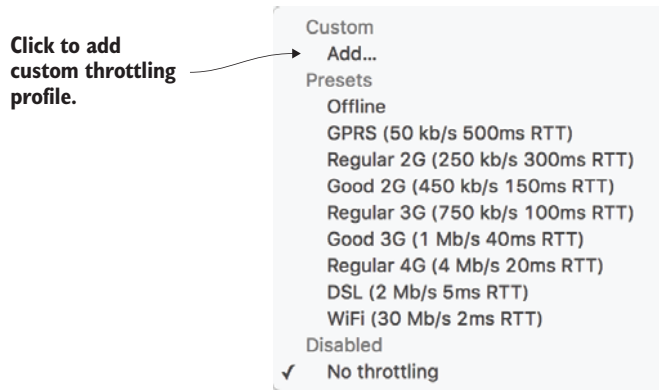


Figure 2.31 The throttling profiles that ship with Chrome, with the option to add custom profiles

Profile Name	Download	Upload	Latency
US Average	1523	1523	55
	optional	optional	optional

Add Cancel

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

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

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

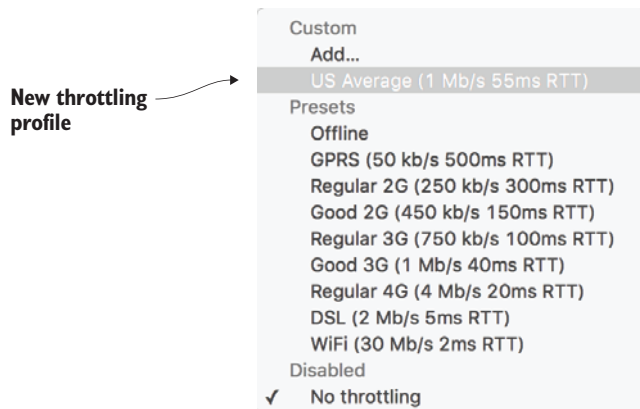


Figure 2.33 Your new custom network throttling profile is now in the list.

2.8 Summary

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

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

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