8 Keeping JavaScript lean and fast

This chapter covers

- Affecting the loading behavior of the `<script>` tag
- Replacing jQuery with smaller and faster API-compatible alternatives
- Using native JavaScript methods to replace jQuery functionality
- Animating with the `requestAnimationFrame` method

The world of JavaScript has exploded into a mélange of libraries and frameworks, leaving us with a slew of options for developing websites. In our excitement to use learn and use them, we often forget that the surest path to a fast website is a willingness to embrace minimalism.

This isn't to say these tools don't have a place in the web development landscape. They can be quite useful and can save developers hours of writing code. The goal of this chapter, however, is to promote minimalism in your website's JavaScript for the benefit of your users.

In this chapter, you'll dive into what *you* can do to improve the performance of script loading on your website. You'll also spend time learning about jQuery-compatible libraries that do much of what jQuery does, but with smaller file sizes and better performance. You'll go one step further and investigate how to replace jQuery with in-browser APIs that deliver much of what jQuery provides, but without the overhead. Finally, you'll learn to use the `requestAnimationFrame` method to code high-performance animations. Let's get started!

8.1 Affecting script-loading behavior

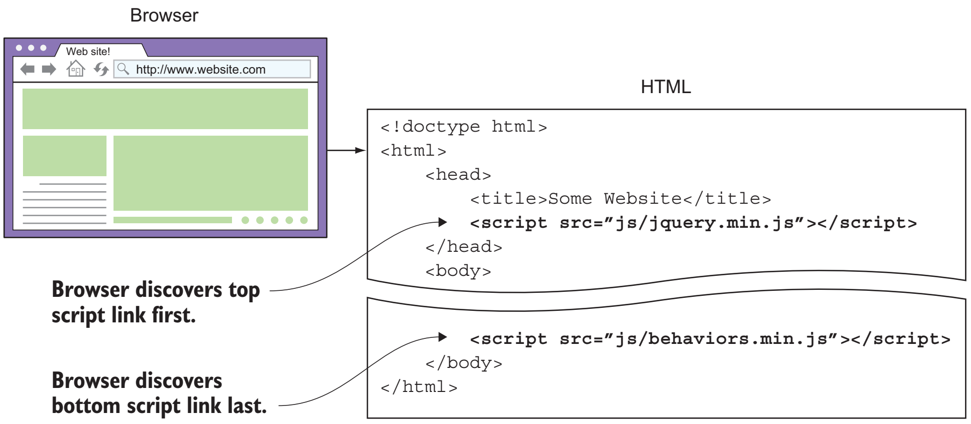
As with the `<link>` tag when loading CSS, the `<script>` tag can hinder the rendering of a page, depending on the tag's placement in the document. You can also modify script-loading behavior via the element's `async` attribute. Let's look at these aspects of script loading, and get a feel for how they can impact performance. You'll start by revisiting the Coyle Appliance Repair website. Download and run it from GitHub with the following commands:

```
git clone https://github.com/webopt/ch8-javascript.git
cd ch8-javascript
npm install
node http.js
```

Let's start by experimenting with the placement of the `<script>` tag.

8.1.1 Placing the <script> element properly

As you may recall from chapters 3 and 4, the placement, and even the presence, of the `<link>` tag can block rendering of a page when loading CSS. The `<script>` tag is responsible for this same kind of behavior as well, but because scripts don't impact the appearance of a page like CSS imports, you have more flexibility in placing `<script>` tags. Figure 8.1 diagrams this behavior.


Browser
Web site!
http://www.website.com
HTML
Browser discovers top script link first.
Browser discovers bottom script link last.

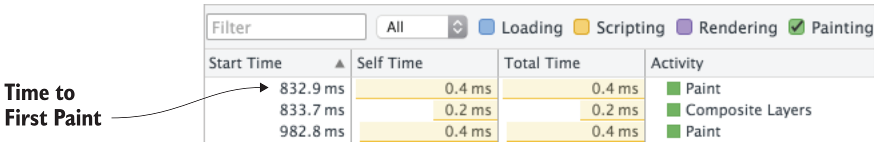
```
<!doctype html>
<html>
    <head>
        <title>Some Website</title>
        <script src="js/jquery.min.js"></script>
    </head>
    <body>

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

Figure 8.1 Browsers read HTML documents from top to bottom. When links to external resources (such as scripts, in this case) are found, the browser stops to parse them. When parsing occurs, rendering is blocked.

This behavior can have an impact on when the browser first paints the page. If the browser detects a `<script>` tag in the `<head>`, for example, it pauses what it's doing to download and parse the script. As this goes on, the browser puts the rendering of the page on the back burner. In your client website's code, the `<script>` tags for jquery.min.js and behaviors.js will be in the document's `<head>` tag, which induce render blocking. You can measure this effect by checking the document's Time to First Paint. The way to measure this was first described in chapter 4, but let's recap the process.

To measure the Time to First Paint for the client's website, select the Regular 3G throttling profile to simulate page loading on a slower connection. Then go to the Timeline tab and head to http://localhost:8080. When the page loads and the timeline populates, go to the bottom of the Timeline pane and switch to the Event Log tab. Once there, filter out all event types except for painting events. The first event to appear in the list is the Time to First Paint, which should look something like figure 8.2.



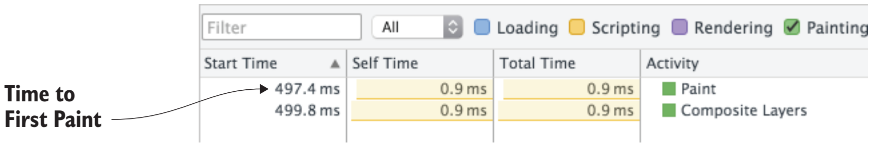
Start Time ▲	Self Time	Total Time	Activity
832.9 ms	0.4 ms	0.4 ms	🟩 Paint
833.7 ms	0.2 ms	0.2 ms	🟩 Composite Layers
982.8 ms	0.4 ms	0.4 ms	🟩 Paint

Figure 8.2 The Time to First Paint in Chrome for the Coyle Appliance Repair website with `<script>` tags in the `<head>` of the document.

The average Time to First Paint for the client's website is roughly 830 ms when the `<script>` tags are in the `<head>`. This seems like a long time for the page to start painting. Experiment and see how that figure changes when you move these scripts to the end of index.html, just before the closing `</body>` tag, as shown in figure 8.3.

In my testing, I was able to achieve an approximate Time to First Paint of 500 ms, which translates to roughly a 40% reduction overall. Well, in this example, anyway. As with most optimizations, your mileage will vary. The size and number of scripts as well as the length of the HTML document can play a role.

The good news about this approach is that it's consistent in nearly all browsers, so it's an easy fix that requires minimal effort. There are other things you can do with the `<script>` tag to influence how scripts load, such as the `async` attribute.



Start Time ▲	Self Time	Total Time	Activity
497.4 ms	0.9 ms	0.9 ms	🟩 Paint
499.8 ms	0.9 ms	0.9 ms	🟩 Composite Layers

Figure 8.3 The Time to First Paint in Chrome for the Coyle Appliance Repair website with the `<script>` tags at the end of the document

8.1.2 Working with asynchronous script loading

Modern browsers support a method for changing the loading behavior of external scripts. This change is via the `<script>` tag's `async` attribute. `async` (short for *asynchronous*) tells the browser to execute a script as soon as it loads, rather than loading each script in order and waiting to execute them in sequence. Figure 8.4 compares these behaviors.

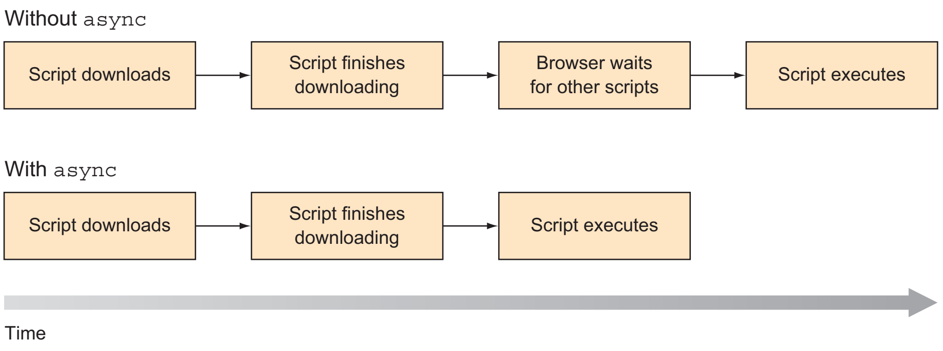


Figure 8.4 A comparison of loading scripts with and without the user of the `async` attribute. The main difference is that scripts loaded with `async` won't wait for other scripts to finish loading before they execute.

`<script>` tags with the `async` attribute behave differently than those without it in that they'll execute immediately on download. They also won't block rendering while they download.

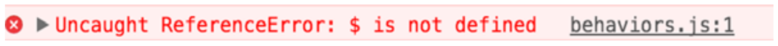
8.1.3 Using async

To use `async`, add it to `<script>` tags that you want to execute asynchronously. In this case, doing this will slash your client website's Time to First Paint by approximately 40%. Try it on the jquery.min.js and behaviors.js scripts, as shown in bold here:

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

Seems easy enough, right? You can reload the page and check whether things still work. Except that they don't. After reloading, you'll see a console error, as shown in figure 8.5.

Well, this is awful, isn't it? What's the point of `async` if it breaks stuff? There is a benefit in using `async`, but things get hairy when scripts are dependent on each other.



```
Uncaught ReferenceError: $ is not defined    behaviors.js:1
```

Figure 8.5 The `async` attribute creates a problem in which behaviors.js fails because it executes before its dependency jquery.min.js is available.

When you use `async` with interdependent scripts, they enter into what's called a race condition, where two scripts can run out of sequence. In this example, a race condition occurs between jquery.min.js and behaviors.js. Because behaviors.js is much smaller than jquery.min.js, it'll always win the race and run first. Because behaviors.js is dependent on jquery.min.js, behaviors.js will always fail due to an unavailable `jQuery` object. This is because jquery.min.js hasn't loaded and executed before behaviors.js does. Figure 8.6 illustrates this race condition.

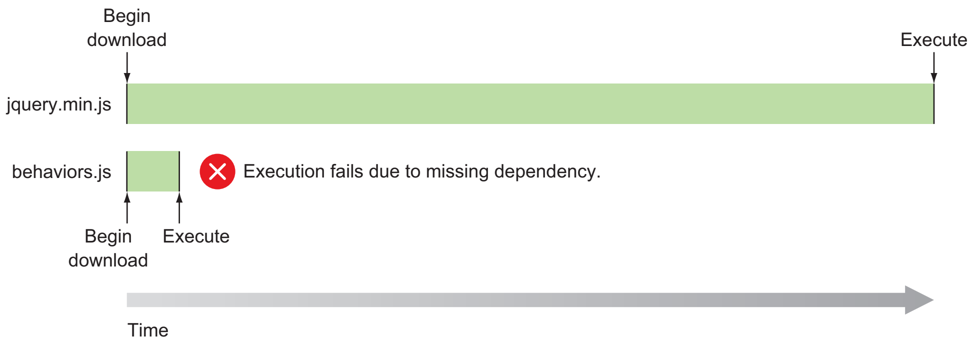


Figure 8.6 A race condition between jquery.min.js and behaviors.js always results in a failure, because behaviors.js loads and executes before its dependency is available.

This doesn't always occur. For example, if none of your scripts have dependencies, you can use `async` freely. It's when scripts have dependencies that things get tricky.

A way of getting around this is to combine your dependent scripts so that those dependencies are wrapped into a single asset. In this case, you can combine jquery.min.js and behaviors.js, in that order. From your command line, you can run this command to combine both scripts into scripts.js:

```
cat jquery.min.js behaviors.js > scripts.js
```

Because `cat` is available only on UNIX-like systems, Windows users will go this route:

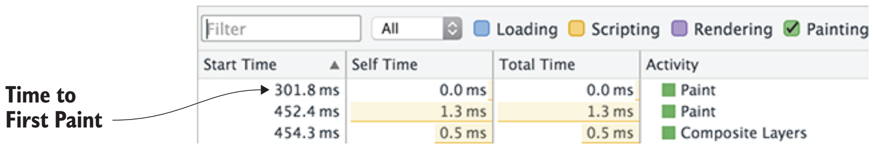
```
type jquery.min.js behaviors.js > scripts.js
```

This command will finish quickly, and when it does, you get rid of both `<script>` tags in index.html and replace them with one `<script>` tag pointing to scripts.js:

```
<script src="js/scripts.js" async></script>
```

When you reload, you'll notice that the page works again, but you're probably thinking, "What's the benefit?" The benefit is quite noticeable. Figure 8.7 shows a further improved Time to First Paint after using `async`.

In my testing, `async` yielded an average Time to First Paint of roughly 300 ms with the scripts bundled, which outperforms placing the independent scripts without



Start Time	Self Time	Total Time	Activity
301.8 ms	0.0 ms	0.0 ms	🟩 Paint
452.4 ms	1.3 ms	1.3 ms	🟩 Paint
454.3 ms	0.5 ms	0.5 ms	🟩 Composite Layers

Figure 8.7 The Time to First Paint value in Chrome for the Coyle Appliance Repair website with scripts bundled and loaded using the `async` attribute

`async` at the bottom of the page by about 200 ms. There's a clear benefit in using `async`, but it doesn't end there. Without `async`, the DOM isn't available until about 1.4 seconds after the page begins to load. With `async`, this figure falls to 300 ms.

If you can manage your dependencies, it pays to use `async`. It's also highly supported, being available in all major browsers, even in IE10 and above. If `async` isn't supported, you can leave your `<script>` tags in the footer and they'll load in older browsers the normal way. Everyone wins! Except only sort of, which we'll discuss shortly.

8.1.4 Using async reliably with multiple scripts

You may take issue with bundling, but it's a good optimization practice for HTTP/1 servers and clients, because it can help alleviate the head-of-line blocking issue inherent to that protocol version.

HTTP/2 connections, however, benefit from assets being served in a more granular fashion as opposed to bundling them. More-granular resources make caching more effective. You can get away with this because HTTP/2 solves the head-of-line blocking problem by being able to serve more concurrent requests than HTTP/1. The particulars of this are explored in chapter 11. The point of this short section is to show you how to load scripts asynchronously while maintaining dependencies. To do this, you'll use a module loader called Alameda.

Alameda is an Asynchronous Module Definition (AMD) module loader written by Mozilla developer James Burke. Despite its capability of supporting complex projects with many dependencies, it's a small script, weighing in at only about 4.6 KB after minification and compression. As far as overhead goes, that's not adding too much to ensure that scripts load asynchronously while respecting their dependencies.

Wait, what are AMD modules?

AMD stands for Asynchronous Module Definition. This specification defines scripts as modules, and provides a mechanism for loading scripts asynchronously with respect to their dependencies on one another.

Using Alameda for this task is easy, and because it's part of the GitHub repository you downloaded at the beginning of this section, you can use it without tracking it down. The GitHub repository for Alameda is at https://github.com/requirejs/alameda. The first thing you should do is remove any `<script>` tags from index.html and add the following `<script>` right before the closing `</body>` tag:

```
<script src="js/alameda.min.js" data-main="js/behaviors" async></script>
```

Here you see three attributes:

- `src` loads the Alameda script.
- `data-main` includes the behaviors.js script. Alameda refers to scripts without their .js extensions, which is the syntax for AMD modules.
- `async` asynchronously loads Alameda, preventing blocking of page rendering.

It's not enough to slap this script on the page and assume everything's going to work. You need to open behaviors.js, add configuration code, and define your jQuery behaviors as an AMD module.

Listing 8.1 Configuring Alameda and defining behaviors.js as an AMD module

```
requirejs.config({
    paths:{
        jquery: "jquery.min"
    }
});

require(["jquery"], function($){
    /* behaviors.js contents truncated for brevity */
});
```

Start of the Alameda configuration

Dependency definitions

Location of the jQuery script relative to behaviors.js

End of the Alameda configuration

AMD module definition.

Contents of behaviors.js wrapped in the module definition

You're doing two things here: you define a configuration that tells Alameda where jquery.min.js lives, and then you wrap the behaviors.js script in a module definition that specifies jQuery as a dependency in the first argument. The dependent code in the second argument then runs when its dependencies are met.

When you reload the page with these changes and check the page's Time to First Paint, you'll notice that you're still hovering around the same mark as when you bundled scripts and used `async`. The big difference, though, is that you're keeping your scripts separate while still respecting the dependency of behaviors.js on jquery.min.js.

Alameda requires a modern browser!

Alameda is an update of RequireJS that requires functionality native to modern browsers to work, such as JavaScript promises. If you need support on a wider array of browsers, you can use RequireJS in place of Alameda. RequireJS and Alameda share a fully compatible API, so you can drop either in place of the other and they should work. Plus, RequireJS is only about 2 KB larger than Alameda when minified and gzipped. Check it out at http://requirejs.org.

Now that you know how to optimize script loading, let's look at alternatives to jQuery that provide a compatible API but offer less overhead and faster execution.

8.2 Using leaner jQuery-compatible alternatives

jQuery burst onto the scene years ago, during a time when accomplishing simple tasks such as selecting DOM elements and binding events required complex code to check for different methods available across browsers. Few methods were unified, and jQuery capitalized on this by providing a consistent API that worked regardless of the browser that used it.

Understandably, jQuery persists because of its utility and convenient syntax. But there are many reasons to consider alternatives that share portions of jQuery's API, but provide a smaller footprint and greater performance.

This section presents alternatives to jQuery. You'll compare their size and performance, and choose one of these options to use on the Coyle Appliance Repair website, as well as cover caveats of these alternatives.

8.2.1 Comparing the alternatives

Many JavaScript libraries are jQuery-compatible. *jQuery-compatible* doesn't mean that every single jQuery method is provided in these alternatives; it means that numerous methods present in jQuery are available in the alternative, and with the same syntax. The idea is that some measure of file size is traded off for less functionality. Some of these libraries also provide better performance.

8.2.2 Exploring the contenders

Here you'll compare three distinct jQuery-compatible libraries: Zepto, Shoestring, and Sprint. Here's a rundown of each:

- *Zepto* is described as a lightweight jQuery-compatible JavaScript library. Of all the jQuery alternatives, it's the most feature-rich out of the box and can be extended to do more. It's the most popular alternative in this field. Find out more at http://zeptojs.com.
- *Shoestring* is written by the Filament Group. It offers less capability than Zepto, but it provides most of the core DOM traversal and manipulation methods that jQuery does, as well as limited support for the `$.ajax` method. Find out more about Shoestring at https://github.com/filamentgroup/shoestring.
- *Sprint* is the lightest and most feature-bare alternative to jQuery, but it's high performing. Although it doesn't provide nearly as much functionality, it's great when you want to start out with very little, but are open to adding a more capable library when the need arises. Find out more about Sprint at https://github.com/bendc/sprint.

When comparing these libraries, you'll consider the size of each and the performance of equivalent methods.

8.2.3 Comparing file size

The biggest motivation for using these alternatives lies in the savings that they offer in file size. As you know from earlier in this book, the best thing you can do to decrease the load time of a website is to limit the amount of data that you send to the user. If you use jQuery in your web projects, these libraries are a perfect place to start cutting the fat. Figure 8.8 compares the file size of these libraries to jQuery. All file sizes assume minification and server compression.

jQuery isn't huge, but its alternatives are far smaller. If you could lighten the load of your jQuery-dependent website by at least 20 KB, wouldn't you? Of course you would. The benefits don't stop there, though. Benefits also exist in performance.

8.2.4 Comparing performance

In this short section, you'll compare the execution times of common jQuery tasks: selecting elements by class name, toggling a class on an element with the `addClass` and `removeClass` methods, and toggling an attribute via the `attr` and `removeAttr` methods.

To measure performance in these cases, I selected a JavaScript library named Benchmark.js, which you can find out more about at https://benchmarkjs.com/. This library allows you to see the number of executions per second that a snippet of JavaScript is capable of.

I won't get into how Benchmark.js works under the hood. It's a highly accurate tool, and if you want to see how I wrote the test scripts, check out the GitHub repo for these tests at https://github.com/webopt/ch8-benchmark. I only want to show how the alternatives selected compare to jQuery for a handful of commonly used methods.

In the element selection test, you select an element of `div.myDiv` on the page by its class name. Figure 8.9 shows how the gallery fares with this simple task.

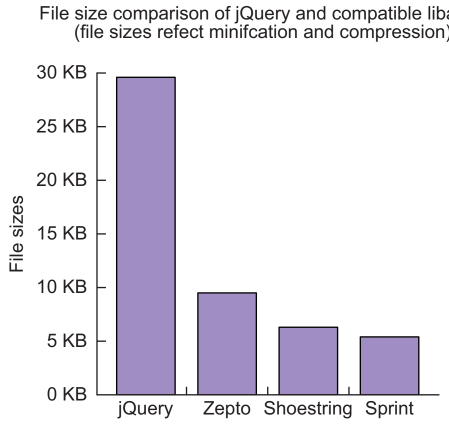


Figure 8.8 A comparison of file sizes of jQuery and its alternatives

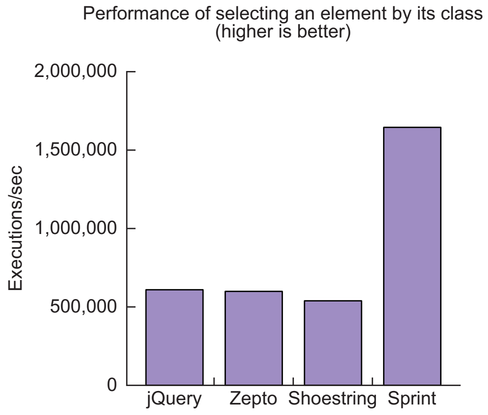


Figure 8.9 Performance of jQuery versus its alternatives when selecting an element by its class

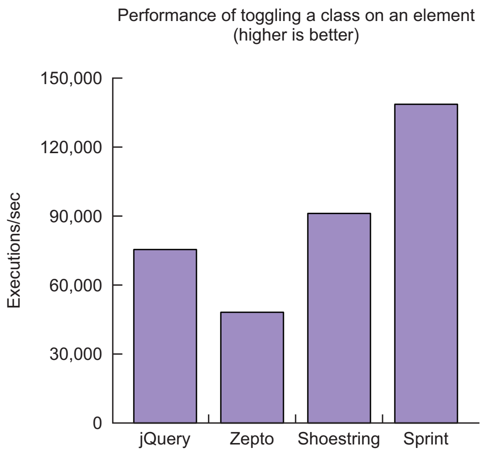


Figure 8.10 Performance of jQuery versus its alternatives when toggling a class on an element

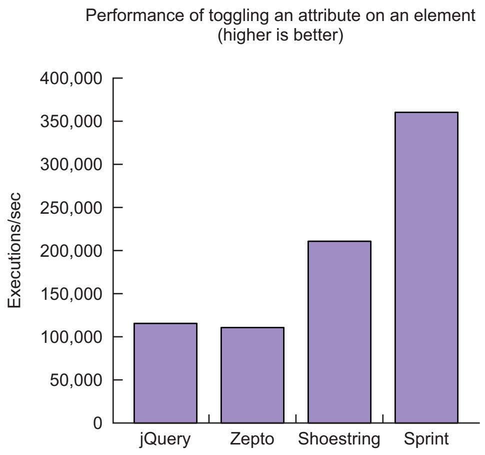


Figure 8.11 Performance of jQuery versus its alternative when toggling an attribute on an element

Sprint is the definite winner here, with jQuery outperforming both Zepto and Shoestring. Figure 8.10 shows how everything stacks up when you toggle a class on an element.

Things are more nuanced here. Sprint is still the clear winner. jQuery comes in third, losing to Shoestring, but still beats out Zepto. You can see how the gallery fares in figure 8.11 when you toggle attributes.

Again, Sprint dominates. Zepto loses, with jQuery in third and Shoestring in second. It should be noted that this is an arbitrary sampling of methods. Each method will compare differently, but some trends do persist. Sprint seems to be the fastest, but it's worth noting that Sprint's API isn't 100% compatible with jQuery's. Zepto covers most jQuery methods, and has plugins to further extend its capability. Although Sprint seems attractive performance-wise, it's not the easiest of the alternatives to retrofit into your jQuery-centric web project.

Now that you have a taste of what these libraries offer in weight and capability, let's go one step further by retrofitting the Coyle Appliance Repair website with one of these alternatives.

8.2.5 *Implementing an alternative*

Using a jQuery alternative is simple for sites that make light use of it: just drop the alternative in jQuery's place. In this short section, you'll do just that. Before you start, you may need to undo any changes you made earlier in the chapter. To do this, type `git reset --hard`.

8.2.6 *Using Zepto*

The alternative that you'll go with for Coyle is Zepto. Though not the highest-performing library of the alternatives, it has support for everything you need, and it's a little less than a third of the size of jQuery. You can take the site's payload down from 122 KB to 102 KB easily.

Another good reason to use Zepto in this case is that it represents the least amount of effort because it's the most compatible with jQuery, whereas libraries such as Shoestring and Sprint require refactoring to work. In environments where time is key (and when is that never true?), you can drop in Zepto in most cases with less effort than the alternatives.

Included in the repository's js folder is a copy of Zepto. To change out jQuery and replace it with Zepto, you need only to update the `src` attribute from js/jquery.min.js to this:

```
js/zepto.min.js
```

When you reload the page, you'll notice that there are no console errors, and all of the functionality on the page should be present, including the jQuery AJAX-driven form submission for the appointment-scheduling modal.

8.2.7 Understanding caveats on using Shoestring or Sprint

"That's it? Nothing else?" is likely what you're thinking. In this case, yes, that's pretty much it. Remember we picked Zepto because it's the most compatible with jQuery out of all the alternatives.

If you drop in Shoestring or Sprint, you'll need to refactor to make them work. The Coyle Appliance Repair website uses jQuery's `$.ajax` method to send an appointment-scheduling email to the site owner, and Shoestring's implementation of the `$.ajax` method isn't fully jQuery-compatible. Because Sprint has no `$.ajax` implementation, it can't make the cut.

It's not just the `$.ajax` method that can be problematic. jQuery alternatives don't support *everything* that jQuery does. Shoestring doesn't support the `toggleClass` method, but Sprint does. Sprint doesn't support the `bind` method, but Shoestring does. Many of these incompatibilities can be refactored by using workarounds or native JavaScript methods.

This is fine, though! The idea is that if you're beginning development of a new website with jQuery in mind, you should start with a minimalist library such as Sprint. If Sprint eventually fails to provide what you need, you can try Shoestring or Zepto. If you get to a point where those options are no longer cutting it, *then* you should go to jQuery.

If you start with minimalism in mind, you can ensure that you're keeping things as lean as possible. This mindset contributes to a faster site for your users. This isn't true of only jQuery, but also all aspects of web development. Always ask, "Do I *need* that hot new library for this site?" Chances are that this may lead you down a different path than what you initially intended.

In the next section, you'll go one step further and remove the need for jQuery and any alternatives altogether, and use native JavaScript to accomplish your goals. This is a significant undertaking if you're used to jQuery's methods, but it will allow you to eliminate all overhead associated with the library and to provide an even faster experience for your client.

8.3 Getting by without jQuery

jQuery and its alternatives are great, but many methods have been implemented (or are being implemented) into browsers that provide much of the same functionality. Tasks such as element selection and event binding that were once a burden to write for cross-browser compatibility now have a unified syntax, thanks to tstandardization efforts.

This section covers how to check for DOM readiness, select elements with `querySelector` and `querySelectorAll`, bind events with `addEventListener`, manipulate classes on elements using `classList`, modify attributes and element contents with `setAttribute` and `innerHTML`, and use the Fetch API to make AJAX calls.

Want to skip ahead?

If you get stuck at any time doing the work in this section, you can skip ahead by typing `git checkout -f native-js` at the command line to see the finished work.

Before you start, you need to undo any work you've done on the client website in your local repository by entering `git reset --hard` at the command line. This reverts all local changes. You'll convert all of the code piece by piece, leaving jQuery in place until everything is replaced by native JavaScript. When done, you'll be able to remove the reference to jquery.min.js, load behaviors.js by using the `async` attribute, and subsequently benefit from faster load times. Let's open behaviors.js in your text editor and get to work!

8.3.1 Checking for the DOM to be ready

If you're familiar with jQuery, you know that you must check for the DOM to be ready before you can execute code. This isn't true of only jQuery, but also DOM-dependent scripts in general. You need to do this because the DOM doesn't fully load before the scripts run, resulting in events not being bound to elements and critical behaviors not functioning. The following listing provides a truncated version of behaviors.js, showing how jQuery checks for the DOM to be ready.

Listing 8.2 jQuery checking for DOM readiness

```
$(function(){                                          <-- Checks for DOM readiness
    /* Content of behaviors.js truncated for brevity. */
});
```

In jQuery, anything encapsulated in `$(function(){});` isn't executed until the document is loaded and ready. To achieve this in native JavaScript, you'll use `addEventListener` (which you'll also use to bind other events later) to check for DOM readiness. The following listing shows this in action.

Listing 8.3 Checking for DOM readiness with `addEventListener`

```
document.addEventListener("readystatechange", function(){
    /* Content of behaviors.js truncated for brevity. */
};
```

Listens for readystatechange, which waits for the DOM to be ready.

That's it! The `addEventListener` method is available in IE9 and above, so this has a high level of compatibility.

Getting deeper support

If you need to support IE versions prior to 9, you can use the `document.onreadystatechange` method to monitor for DOM readiness. This method works in newer browsers as well.

Next, you'll investigate how to use the `querySelector` and `querySelectorAll` methods to select elements on a page, as well as how to take the `addEventListener` method further by binding events to these elements.

8.3.2 Selecting elements and binding events

The lion's share of jQuery's usefulness is in its ability to select elements and bind events to them. When it comes to selecting elements natively, the `querySelector` and `querySelectorAll` methods are the go-to solution. Like jQuery's core `$` method, these two methods accept a CSS selector string as an argument. That string is used to return a node in the DOM that you can work with. The difference between the two is that `querySelector` returns the first element that matches the expression, whereas `querySelectorAll` returns *all* elements that match.

Both methods have strong support across browsers, including IE9 and above, with partial support in IE8. This listing compares these two methods to their jQuery equivalents.

Listing 8.4 `querySelector` and `querySelectorAll` vs. jQuery's core `$` method

```
/* Selecting one element. */
var element = document.querySelector("div.item");
var jqElement = $("div.item").eq(0);

/* Selecting a set of elements */
var elements = document.querySelectorAll("div.item");
var jqElements = $("div.item");
```

The first line in listing 8.4 selects the first matching div.item element `querySelector` and the second line selects the first matching div.item element via jQuery. Line 3 selects all matching div.item elements via `querySelectorAll` and the final line selects all matching div.item elements via jQuery.

When element(s) are returned with either of these methods, you can then use the `addEventListener` method to attach events to those elements. The next listing shows a simple use of `addEventListener` to bind a click event on an item returned with `querySelector`.

Listing 8.5 Binding a click event on an item with `addEventListener`

```
document.querySelector("#schedule").addEventListener("click", function(){
    /* Code to execute on click. */
});
```

Using a combination of these methods will help you eliminate most of the jQuery-dependent code in behaviors.js. This code fires the appointment-scheduling modal.

Listing 8.6 jQuery-centric appointment scheduling modal launch code

```
// Scheduling modal open
$("#schedule").bind("click", function(){
    $("body").addClass("locked");
    openModal();
});
```

jQuery element selection and click event binding

The part of the code you want to focus on here is the first line, which selects the appointment-scheduling button element (`#schedule`) and the `bind` method that binds a click event to that element. Using a combination of `querySelector` and `addEventListener`, you can convert this to what you see next.

Listing 8.7 Appointment-scheduling modal event binding using native JavaScript

```
// Scheduling modal open
document.querySelector("#schedule").addEventListener("click", function(){
    $("body").addClass("locked");
    openModal();
});
```

Native element selection and click event binding

When you reload, you'll still be able to trigger the modal by clicking the scheduling button. Although the code within the event handler is still driven by jQuery, you're getting closer to your goal of removing jQuery altogether.

In your text editor, switch out the remaining `bind` events to use `addEventListener`, as in listing 8.7. There should be three remaining calls to `bind` that you'll be able to replace.

Next, you'll replace calls to jQuery's `addClass` and `removeClass` methods with the native JavaScript `classList` method.

8.3.3 Using classList to manipulate classes on elements

The client website's JavaScript makes extensive use of jQuery's `addClass` and `removeClass` methods to add and remove classes. A native method called `classList` gives you this same functionality. This listing shows this method compared to its jQuery counterparts.

Listing 8.8 `classList` vs. jQuery's `removeClass` and `addClass` methods

```
/* Adding Classes */
$(".modal").addClass("show");                            <- jQuery's addClass method
document.querySelector(".modal").classList.add("show");  <- Adding a class with classList's add method

/* Removing Classes */
$(".modal").removeClass("show");                         <- jQuery's removeClass method
document.querySelector(".modal").classList.remove("show");  <- Removing a class with classList's remove method

/* Toggling Classes */
$(".modal").toggleClass("show")                          <- jQuery's toggleClass method
document.querySelector(".modal").classList.toggle("show");  <- Toggling a class with classList's toggle method
```

Although your client's website doesn't use the `toggleClass` method, it's important to note that `classList` has a toggle method. Unfortunately, this method isn't supported well in IE. Support for the `classList` method otherwise is good overall, with IE10 and above supporting it. This listing shows the `openModal` function that opens the scheduling modal.

Listing 8.9 The jQuery-dependent `openModal` function

```
function openModal(){
    window.scroll(0, 0);
    $(".pageFade").removeClass("hide");   <- Removing hide class on the .pageFade element
    $(".modal").addClass("open");         <- Adding open class on the .modal element
}
```

Achieving the same result with the `classList` method is a bit more involved. You need to convert the jQuery element selection code to use the `querySelector` method instead. This listing shows how to transform the code from listing 8.9 into fully jQuery-independent code.

Listing 8.10 The jQuery-independent `openModal` function

```
function openModal(){
    window.scroll(0, 0);
    document.querySelector(".pageFade").classList.remove("hide");  <- Removing the hide class on the .pageFade element
    document.querySelector(".modal").classList.add("open");        <- Adding the open class on the .modal element
}
```

Next, you need to search for all uses of the `removeClass` and `addClass` methods in behaviors.js, and update them to use `classList`. When you do, be sure to update the jQuery `$` selection methods to use the `querySelector` method.

> **What if classList isn't supported?**
>
> It's possible that you may need to support IE9 or below. If this is the case, you can use the `className` property instead. This property doesn't have methods for adding, removing, or toggling classes. Instead, it's a string that you can use to assign whatever classes you need to the targeted element—not as convenient as `classList`, but it works in a pinch.

After you've changed all the code to use the `classList` method, it's time to replace the jQuery attribute and content modification methods with their native JavaScript counterparts.

8.3.4 Reading and modifying element attributes and content

Another piece of functionality that the client's website relies on jQuery to accomplish is the reading and modifying of attributes of elements, as well as modifying element contents. These behaviors can be replaced by native JavaScript methods easily. The next listing compares jQuery's attribute manipulation methods to their native JavaScript equivalents.

Listing 8.11 Modifying attributes with jQuery vs. native JavaScript

```
/* Reading attributes */
var jqAttr = $("link").attr("media");                          <- Reading an attribute using jQuery
var attr = document.querySelector("link").getAttribute("media");   <- Reading an attribute with native JavaScript

/* Setting attributes */
$("link").attr("media", "print");                              <- Setting an attribute using jQuery
document.querySelector("link").setAttribute("media", "print");     <- Setting an attribute with native JavaScript

/* Removing attributes */
$("link").removeAttr("media");                                 <- Removing an attribute using jQuery
document.querySelector("link").removeAttribute("media");           <- Removing an attribute with native JavaScript
```

You'll also need to be able to know how to read and/or modify the contents of an element, because the client's website also uses this method. Here's how jQuery reads the contents of an element as compared to JavaScript's `innerHTML` property.

Listing 8.12 jQuery's `html` method vs. JavaScript's `innerHTML` property

```
/* Reading the content of an element */
var jqContents = $(".item").html();
var contents = document.querySelector(".item").innerHTML;

/* Changing the contents of an element */
$(".item").html("Hello world!");
document.querySelector(".item").innerHTML = "Hello world!";
```

Reading element contents with jQuery

Reading element contents with innerHTML property

Setting element contents with jQuery

Setting element contents with innerHTML property

The syntax between jQuery's `html` method and the native `innerHTML` property is different, in that the `html` method is a function, whereas `innerHTML` is a property that you assign a value to.

There aren't a whole lot of places in the client website's JavaScript that need to modify attributes or set content on elements, but it does happen during a key moment: when the user submits an appointment request, and the confirmation modal appears. This is in a `success` callback in a jQuery `ajax` call.

Listing 8.13 Attribute and element content modification via jQuery

```
success: function(data){
    $("#status").html(data.message);
    document.querySelector(".statusModal").classList.add("show");
    document.querySelector(".modal").classList.remove("open");

    if(data.status === true){
        $("#okayButton").attr("data-status", "success");
        $("#headerStatus").html("Thank You!");
    }
    else{
        $("#okayButton").attr("data-status", "failure");
        $("#headerStatus").html("Error");
    }
}
```

Message text is placed into the status text area.

data-status attribute is set on okayButton.

Header is updated to reflect success or failure.

In the listing, the message text from the appointment-scheduling emailer is placed into the status text area. The `data-status` attribute is set on the okay button, which is used to determine what the button does in the context of success or failure. The header of the status modal is updated to reflect the success or failure of appointment submission.

With some modifications, you can take what you've learned so far and turn this into something like this listing, which runs without jQuery.

Listing 8.14 Attribute and element content modification via native JavaScript

```
success: function(data){
    document.querySelector("#status").innerHTML = data.message;
    document.querySelector(".statusModal").classList.add("show");
    document.querySelector(".modal").classList.remove("open");

    if(data.status === true){
        document.querySelector("#okayButton")
            .setAttribute("data-status", "success");
        document.querySelector("#headerStatus")
            .innerHTML = "Thank You!";
    }
    else{
        document.querySelector("#okayButton")
            .setAttribute("data-status", "failure");
        document.querySelector("#headerStatus")
            .innerHTML = "Error";
    }
}
```

Assigns message and status text from appointment scheduler.

Sets status of scheduling operation on the data-status attribute.

You need to update one more spot that uses jQuery's `attr` method to read an attribute. This occurs when the user clicks the okay button in the status modal. Here are the relevant lines of this code.

Listing 8.15 Getting an attribute via jQuery's `attr` method

```
$("#okayButton").bind("click", function(e){
    if($(this).attr("data-status") === "failure"){
```

The jQuery click binding

data-status attribute's value

This spot is a little tricky because jQuery's `$(this)` object is used inside the click binding's code to refer to the `#okayButton` element. When you convert this to the `addEventListener` syntax that you used earlier, this code will break. You need to use the event object (assigned to `e` in the function call) to replace the reference to the `$(this)` object, and use the `getAttribute` method to retrieve the value of the `data-status` attribute instead. Here is a working replacement of both methods.

Listing 8.16 Getting an attribute via the `getAttribute` method

```
document.querySelector("#okayButton").addEventListener("click", function(e){
    if(e.target.getAttribute("data-status") === "failure"){
```

The top line is the converted click binding with the event object in the function call. In the last line, the `e.target` attribute refers to the element that the click binding was attached to, with the `getAttribute` method retrieving the data-status attribute's value.

In the absence of jQuery's `$(this)` object, you can use the event object's `target` method to refer to the element that the event was bound to inside the event code itself. It's weird to get used to if you've been used to jQuery, but it'll quickly become second nature.

Before you can wrap things up and remove jQuery from the project, you'll replace the last bit of jQuery-dependent functionality left, which is the `$.ajax` call used to send an AJAX request to the server to schedule an appointment. You'll replace the jQuery AJAX functionality with a native JavaScript version called the Fetch API.

8.3.5 Making AJAX requests with the Fetch API

In the old days of AJAX requests, you had to use the `XMLHttpRequest` request object. It was an unwieldy way of making AJAX requests, and different browsers required different approaches. jQuery made AJAX requests a much more convenient task by wrapping its own AJAX functionality around the `XMLHttpRequest` object. It still works great to this day, but some browsers have implemented a native resource-fetching API called the Fetch API.

8.3.6 Using the Fetch API

The most basic use of the Fetch API is for a `GET` request of a resource. A good example is interacting with an API that gives access to a database of movies and returns JSON data. This shows such a request using `fetch`.

Listing 8.17 Fetch API–driven AJAX request with a JSON response

```
fetch("https://api.moviemaniac.com/movies/the-burbs")
    .then(function(response){
    return response.json();
}).then(function(data){
    console.log(data);
});
```

In the listing, the `fetch` method takes a minimum of one argument, which is the URL to the resource. On success, a promise is returned that allows you to work with the JSON data. The raw response object has a `json` method that you can return to the next promise in the chain. Another promise is returned with the encoded JSON data. The `console.log (data);` line outputs the data from the response to the console.

This is only a basic use of the Fetch API. The client's website uses the jQuery `$.ajax` method to send form data using a `POST` request. Accomplishing this takes a bit more work but uses a bit less code than if you used jQuery's `$.ajax` function.

To be fair, you've been submitting the form to a mock location that returns a JSON response for illustrative purposes. If you try this approach in your own websites with a back-end script, you'll find that it should work fine. This listing shows the Fetch API at work in place of jQuery.

Listing 8.18 Fetch API–driven AJAX request

```
fetch("js/response.json", {
    method: "post",
    body: new FormData(document.querySelector("#appointmentForm"))
}).then(function(response){
    return response.json();
}).then(function(data){
    document.querySelector("#status").innerHTML = data.message;
    document.querySelector(".statusModal").classList.add("show");
    document.querySelector(".modal").classList.remove("open");

    if(data.status === true){
        document.querySelector("#okayButton")
            .setAttribute("data-status", "success");
        document.querySelector("#headerStatus").innerHTML = "Thank You!";
    }
    else{
        document.querySelector("#okayButton").setAttribute("data-status",
        ➥ "failure");
        document.querySelector("#headerStatus").innerHTML = "Error";
    }
});
```

Fetch API sends a request to the appointment scheduler.

Request method

Body of the request encapsulated via a FormData object

Promise fulfilled by the scheduler's response

Last promise in the chain is returned, and the post-submission code runs.

JSON response is returned to the next promise in the chain.

When this code runs, test the appointment-scheduling modal and you should see that it works fine. Not bad for a native API, and it's much more attractive than the usual `XMLHttpRequest` tango that we've done in years past.

None of this is meant to pick on jQuery's `$.ajax` API. It's an awesome wrapper around `XMLHttpRequest`, but as browsers pick up more support for the Fetch API, it makes sense to abandon `$.ajax`. Of course, if you're going to rely on `fetch`, you need to be able to polyfill those browsers that don't support it yet.

8.3.7 Polyfilling the Fetch API

As could be expected, not every browser supports the Fetch API. At this point, you have a few options:

- You can avoid using `fetch` altogether and use a standalone implementation of jQuery's `$.ajax` API, such as this one at https://github.com/ForbesLindesay/ajax.
- You can sniff for the `fetch` method in the `window` object. If the method is found, you can use `fetch`. If not, you can use the standard `XMLHttpRequest` object. Or use the `XMLHttpRequest` object no matter what, because it's well supported (albeit a pain to use).
- You can sniff for the `fetch` method, and if not found, asynchronously load a polyfill.

In this short section, you'll opt for the third method because it's the most optimal. Browsers that support the Fetch API will rely on a native browser method without the overhead of an external script. Browsers that don't will incur the overhead of the polyfill, but the syntax will be unified.

A decently robust polyfill of the Fetch API can be found at https://github.com/github/fetch. For the sake of simplicity, a minified version of this script is bundled in the repo for the client's website as fetch.min.js in the js folder.

Using a familiar approach to loading polyfills in prior chapters, you can load this script conditionally based on the presence of the `fetch` object. You'll do this by placing an inline `<script>` at the bottom of index.html before the closing `</body>` tag.

Listing 8.19 Conditionally loading the Fetch API polyfill

```
<script>
    (function(document, window){
        if(!window.fetch){
            var fetchScript = document.createElement("script");
            fetchScript.src = "js/fetch.min.js";
            fetchScript.async = "async";
            document.body.appendChild(fetchScript);
        }
    })(document, window);
</script>
```

Creates a `<script>` element that'll load the fetch API polyfill.

Checks if the fetch method is unavailable.

Sets script source.

Loads script asynchronously.

Appends `<script>` element, initiating script load.

You might be thinking about dependencies, because behaviors.js depends on `fetch`. The key here is that the call to `fetch` isn't executed on page load. Plenty of time is afforded for the polyfill to load before the user has a chance to open the scheduling modal, fill out the form, and hit Submit. It's a soft sort of dependency in that time and logistics work in your favor, and you can see how this plays out in figure 8.12.

Test this approach in a browser that doesn't support the Fetch API, such as IE, and see how it works. You'll be able to tell whether fetch.min.js has been loaded by examining the network requests for that browser.

With all jQuery methods firmly replaced with native JavaScript, you can now remove the reference to jquery.min.js, and use the `async` attribute to asynchronously load behaviors.js. Now that your client's website is running optimally, you can move onto learning about animating elements with JavaScript by using `requestAnimationFrame`.

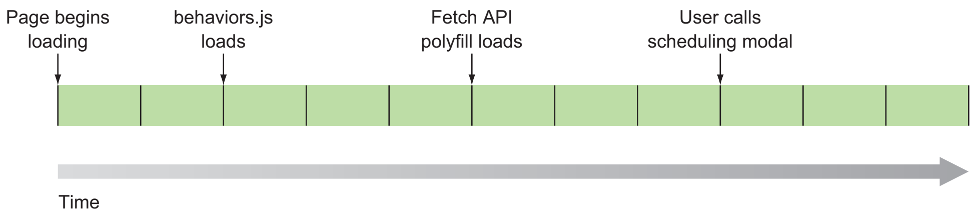


Figure 8.12 The loading of the fetch API polyfill and its timing with the user's intentions to fire the scheduling modal

8.4 Animating with requestAnimationFrame

Animation in the earlier days of JavaScript was less perfect than it is now. You'd usually have to use a timer function such as `setTimeout` or `setInterval` in order to achieve the effect. As time has passed, and the capability of browsers has increased, we have a newer and higher-performing method that helps us animate elements with JavaScript.

This section discusses traditional timer-based animations and how to use `requestAnimationFrame` in their place. From there, you'll see how `requestAnimationFrame` compares in performance to its timer-based ancestors and CSS transitions, and then put the method to work on the Coyle Appliance Repair website.

8.4.1 requestAnimationFrame at a glance

Animation is different when done in JavaScript as compared to CSS. In CSS, you'd apply a `transition` property to an element that tells the browser that a specific property (or properties) will change. When that property changes, the browser animates the transition between the start and end points. The underlying logic that runs the animation is all handled by the browser. With JavaScript, you have to perform this work yourself. Let's take a look at how animation has traditionally been achieved in JavaScript and compare that to `requestAnimationFrame`.

8.4.2 Timer function-driven animations and requestAnimationFrame

When animating in JavaScript, you're changing an element's appearance or position on a screen through the element's `style` object via a timer function to give the appearance of motion. In the days of old, so to speak, `setTimeout` and `setInterval` were the timer functions used to animate elements on an interval, usually by an interval of 1000 ms / 60, which aims to animate effects at roughly 60 frames per second. Typical code for this kind of animation looks something similar to this.

Listing 8.20 Animating with a timer function (`setTimeout`)

```
function draw(){
    document.querySelector(".item").style.width =
        (parseInt(document.style.width) + 2)) + "px";
    setTimeout(draw, 1000 / 60);
}

draw();
```

Updates the left property of the element in 2 pixel increments.

Recursively runs the draw function at 60 FPS.

Drawing is triggered with the initial function call.

Timers themselves aren't expensive, but they're not optimal for animation code. To solve this problem, the `requestAnimationFrame` method was developed. Using this method is similar to the code in listing 8.20, and is shown here.

Listing 8.21 Animating with `requestAnimationFrame`

```
function draw(){
    document.querySelector(".item").style.width =
        (parseInt(document.style.width) + 2)) + "px";
    requestAnimationFrame(draw);
}

draw();
```

Runs a specific function, but in an unspecified interval.

At first glance, this code seems somewhat lacking, because unlike `setTimeout`, `requestAnimationFrame` doesn't allow the user to specify an interval in milliseconds. So how does it even work, then? Simple: The interval is handled within `requestAnimationFrame`, and it acts according to the refresh rate of the display, which tends to be 60 Hz on most devices; `requestAnimationFrame` aims for 60 FPS on a typical device. If a device has a different refresh rate, `requestAnimationFrame` will animate accordingly.

Admittedly, this example has limitations in that it animates the element's `left` property for an infinite amount of time, but it illustrates the concept. In a bit, I'll show a realistic implementation of this on the client's website, but not before you take a look at the performance of `requestAnimationFrame` versus its traditional timer-based animations and CSS transitions.

8.4.3 Comparing performance

Earlier I said that `requestAnimationFrame` boasts better performance than its timer-based ancestors `setTimeout` and `setInterval`. To test these methods, I wrote a simple animation for each. The animation is of a box that travels a distance of 256 pixels from left to right, while doubling in width and height and shifting to 50% opacity. You can try each of these tests for yourself at http://jlwagner.net/webopt/ch08-animation if you're so inclined. Figure 8.13 compares the performance of these two methods

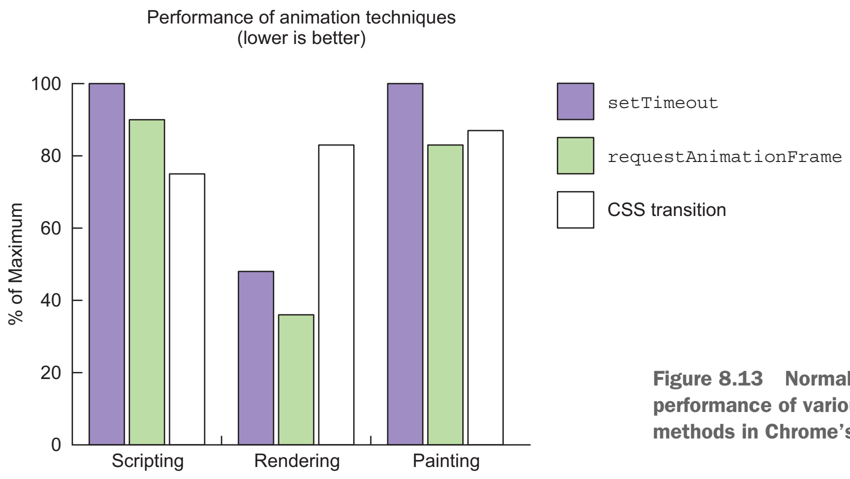


Figure 8.13 Normalized performance of various animation methods in Chrome's Timeline tool

using `setTimeout`, `requestAnimationFrame`, and CSS transitions as profiled in Google Chrome's timeline profiler.

One thing to remember about both `setTimeout` and `requestAnimationFrame` is that because they're dependent on JavaScript, they require more scripting time than CSS transitions, and this is normal. But `requestAnimationFrame` spends less time both painting and rendering than either of the other two methods.

Now that you know how to use `requestAnimationFrame` and how it performs, let's fire up the Coyle Appliance Repair website again and animate the scheduling modal.

8.4.4 Implementing requestAnimationFrame

You've had a little bit of downtime since the Coyle Appliance Repair website launched, and so now it might be fun to experiment with `requestAnimationFrame`. On Coyle, the only animation that occurs is when the scheduling modal is opened. This used a CSS transition in the past, but now you're experimenting with a timer-based animation that you want to transition over to use `requestAnimationFrame`. You'll need to grab the latest code for that, so type in this command to switch over to a new branch:

```
git checkout -f requestanimationframe
```

I've written a flexible function for the client's website to test `setTimeout` and `requestAnimationFrame` for animating the modal in behaviors.js.

Listing 8.22 Animation function using `setTimeout`

```
function animate(selector, duration, property, from, to, units){
    var element = document.querySelector(selector),
        endTime = Number(new Date()) + duration,
        interval = (1000 / 60),
        progress = function(){
            var progress = Math.abs(((endTime - +new Date()) / duration) - 1);
            return (progress * (to - from)) + from;
        },
        draw = function(){
            if(endTime > +new Date()){
                element.style[property] = progress() + units;
                setTimeout(draw, interval);
            }
            else{
                element.style[property] = to + units;
                return;
            }
        };

    draw();
}
```

Interval of the effect, calculated at 60 FPS

Selects element from the DOM and stores it in a variable

Calculates animation's ending time

Draws the animation

Increments animation and recursively calls the draw function

Checks if duration of the effect has expired

Sets element to final position if duration period has elapsed.

Kicks off initial call to begin drawing the animation

Though not as complete as something like jQuery's `animate` function, this is a much more flexible implementation than what was shown in listing 8.20.

In the invocation illustrated in figure 8.14, you're telling `animate` to select the `.modal` element and animate its `top` property from `-150%` to `10%` over a duration of `500` milliseconds. If you click the Schedule an Appointment button and launch the modal, you'll see that it opens fine with your JavaScript animation code. What kind of work is involved in turning this function over to use `requestAnimationFrame`, though? Surprisingly little. The next listing shows the `animate` function's `draw` method modified to use `requestAnimationFrame`, with modifications in bold.

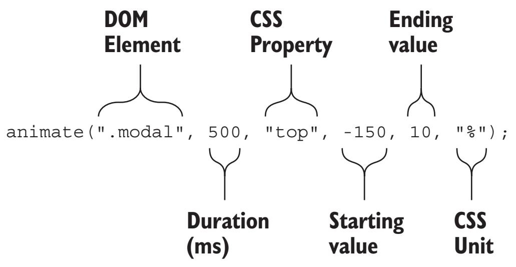


Figure 8.14 The `animate` function in use, with arguments labeled

Listing 8.23 Substituting `requestAnimationFrame` in place of `setTimeout`

```
draw = function(){
    if(endTime > +new Date()){
        element.style[property] = progress() + units;
        requestAnimationFrame(draw);
    }
    else{
        element.style[property] = to + units;
        return;
    }
};
```

That's it, for the most part. You remove the call to `setTimeout` and replace it with a call to `requestAnimationFrame`. Everything should work as before, only with a higher-performing animation method. If you want to do a little cleanup, you can also remove the `interval` variable, because `requestAnimationFrame` doesn't need it.

`requestAnimationFrame` isn't universally supported, so what can you do to ensure better support? For one, you can create a placeholder that allows you to use `requestAnimationFrame` first but then fall back to `setTimeout` when it no longer exists. The next listing shows how to do just that.

Listing 8.24 `requestAnimationFrame` fallback using `setTimeout`

```
window.raf = (function(){                    <- Defines fallback method.
    return window.requestAnimationFrame || function(callback){
        var interval = 1000 / 60;
        window.setTimeout(callback, interval);
    };
})();
```

Calculates a 60 FPS interval.

Returns whichever method is available first.

If setTimeout is relied upon, the callback and interval are set.

If you use this approach, you'll need to update your code to use the custom `raf` method in place of the `requestAnimationFrame` method, but this will give your application's animation methods the broadest possible support. With this approach, you get the benefits of `requestAnimationFrame` when it's available, and you fall back to `setTimeout` when it's unavailable. Not too shabby.

Next, you'll briefly cover how to use Velocity.js, a simple `requestAnimationFrame`-driven JavaScript animation library.

8.4.5 Dropping in Velocity.js

This foray into `requestAnimationFrame` may leave you with more questions than answers. It can be challenging to use this method in place of CSS transitions or jQuery animations, especially if requirements are complex. This short section briefly introduces Velocity.js, which makes animation as convenient as jQuery's `animate` method.

Velocity.js is an animation library that uses an API similar to jQuery's `animate` method. You can learn more about it at http://velocityjs.org. The best part about Velocity is that it's jQuery-independent. But if you have a project using jQuery that relies heavily on its `animate` method, dropping in Velocity.js makes the process of animating the same as with jQuery. For example, consider this jQuery animation code:

```
$(".item").animate({
    opacity: 1,
    left: 8px
}, 500);
```

You can port this animation code to use Velocity.js, like so (changes in bold):

```
$(".item").velocity({
    opacity: 1,
    left: 8px
}, 500);
```

This simple change from `animate` to `velocity` will use the Velocity animation engine instead of jQuery's, which gives you silky smooth `requestAnimationFrame`-powered performance, as well as easing functions to give your animations a sense of natural movement. Unlike jQuery's `animate` method, it allows you to animate colors, transforms, and scrolling.

If you use Velocity.js *without* jQuery, the syntax does change somewhat. As Velocity loads, it'll check whether jQuery is loaded. If jQuery isn't present, the syntax for animating the same element as shown in the examples changes to the following:

```
Velocity(document.querySelector(".item"), {
    opacity: 1,
    left: 8px
}, {
    duration: 500
});
```

Aside from semantics, this syntax doesn't differ all that much. With or without jQuery, Velocity.js can make your JavaScript-driven animations much more fluid and efficient, without getting into the weeds of writing your own animation code.

Be warned that this library is about 13 KB minified and compressed, so consider using it only if animation features prominently on your website and performance is paramount. Adding 13 KB of overhead to a site that doesn't feature much in the way of animation will contribute to a suboptimal experience for your users, and may be better served by writing your own animation code or using CSS transitions instead.

8.5 Summary

You learned many concepts in this chapter about how to keep your JavaScript lean and fast:

- Depending on its position, the `<script>` tag can block rendering, which delays the display of the page in the browser. Placing `<script>` tags toward the bottom of the document can speed up the rendering of a page.
- The `async` attribute can provide further performance benefits if you can manage the execution of scripts that use it.
- Managing the execution of interdependent scripts that use `async` can be challenging. A third-party script-loading library such as Alameda or RequireJS can provide a convenient interface for managing script dependencies, while also providing the benefit of asynchronous script loading and execution.
- Although jQuery is useful, it has a relatively large footprint. A portion of its functionality can be better served by jQuery-compatible alternatives that are smaller in file size, and in some cases, better performing.
- Browsers are providing more jQuery-like functionality as time goes on. You can select elements with `querySelector` and `querySelectorAll`, and bind events to them by using `addEventListener`. You can also manipulate element classes by using `classList`, get and set attributes on them by using `getAttribute` and `setAttribute`, and modify their contents by using `innerHTML`. For a cheat sheet of jQuery methods and their native browser equivalents, check out appendix B.
- The Fetch API provides a convenient native interface for requesting remote resources via AJAX. It can also be effectively polyfilled for browsers that don't support it.
- The `requestAnimationFrame` API is a newer JavaScript function that you can animate in lieu of `setTimeout` or `setInterval`. It's higher-performing than those older timer-based methods, and renders and paints faster than CSS transitions.

In the next chapter, you'll learn about service workers in JavaScript. You'll see how to use them to serve offline experiences to users with limited or no internet connectivity, and to improve the performance of your site.