9 Boosting performance with service workers

This chapter covers

- Understanding what service workers are and what they allow you to do
- Installing a service worker on a simple site
- Caching network requests inside a service worker
- Updating a service worker

As the web has matured, so too has the technology that it relies on. No longer are we tied to our desks while browsing the web. With the advent of mobile devices, people are accessing content on Wi-Fi and data networks of varying degrees of quality and reliability. This introduces challenges in the way we access content, particularly in the case of poor or absent internet connections that may leave the user in a lurch.

Sometimes we go offline—in an airplane without Wi-Fi, or passing through a tunnel in a car or train, for example. It's a fact of life. When this happens, we're somewhat used to being unable to view content on websites. But it doesn't have to be this way, and this is where service workers enter the picture.

In this chapter, you'll learn about service workers: how they work, how to use them to intercept network requests, and how they can be used to cache site assets for times when your device is offline. Beyond the mere convenience of providing an offline experience to your users, you'll also learn of the performance benefits that can come with using service workers, which can make repeat visits to your website even faster than before.

As with any code you write, sometimes changes are necessary. Changing a service worker isn't as straightforward as changing other components of a website, so we'll cover what you need to do when you make changes to one. Let's get started learning about service workers!

9.1 What are service workers?

Service workers are a kind of worker—an evolving standard of scripts that operate in a separate and special scope from ordinary scripts. Workers perform tasks in the background, on a separate processing thread than typical JavaScript code that you'd write and reference with `<script>` tags. Figure 9.1 shows a service worker operating on its own thread.

Because service workers operate on a separate thread, they behave differently than JavaScript loaded via the `<script>` tag. Service workers don't have direct access to the `window` object on the owner page. Although they *can* communicate with the parent page, they must do so indirectly through an intermediary, such as the `postMessage` API.

The problems that workers solve depend on the kind of worker we're talking about. Web workers, for example, allow the browser to perform CPU-intensive tasks without slowing or halting the browser's UI. Service workers, which we cover specifically in this chapter, allow the user to intercept network requests and conditionally store items in a special cache via the `CacheStorage` API. This cache is separate from the native browser cache, and by using it, we can serve content to the user from a `CacheStorage` cache when they're offline. We can also use this special cache to boost the rendering performance of a page.

A theoretical example of a service worker in use is on a popular blog. If the page caches articles via `CacheStorage` as the user reads them, they'd be available for offline

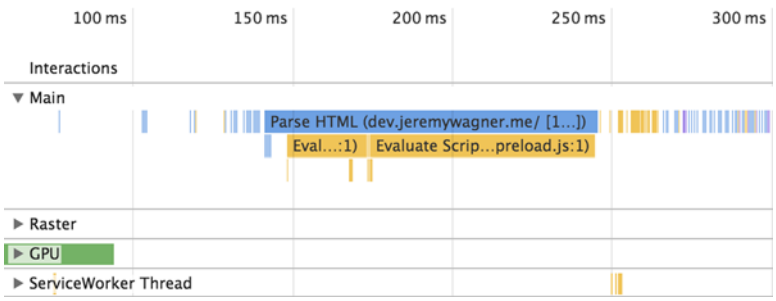

Figure 9.1 A service worker operating on its own thread labeled ServiceWorker Thread can be seen at the bottom in this view of Chrome's Timeline tool.

viewing in the event that a user somehow loses connectivity. This could be useful in various situations, such as when a cellular or Wi-Fi connection is weak, or when a network connection isn't available.

Service workers can help us deal with this problem, not by overcoming the problem of poor or absent connectivity, but by presenting cached content that the user has already seen so that they have *something* to look at, rather than nothing at all. It doesn't fix the inability to access updated content, but rather solves the problem of a broken web-browsing experience.

The service worker interface itself is light, and consists of events that are triggered in specific instances, such as when a service worker is installed, or when a network request is made. These events are listened for with the `addEventListener` method that you learned about in chapter 8. The workhorse of the service worker you'll write in this chapter is the `fetch` event. This is the event you'll use to intercept network requests, and store or request items from a `CacheStorage` cache. Figure 9.2 illustrates this process.

With this event-driven interface, service workers can help create offline experiences for when an internet connection is poor or absent altogether. This is done through an interface that intercepts network requests, and reads from or writes to a `CacheStorage` cache. You'll use `CacheStorage` when you write your first service worker in the next section.

Now that you have a little background on what service workers do at a higher level, there's no time like the present to dive in and learn how to use them. In the next section, you'll do exactly that.

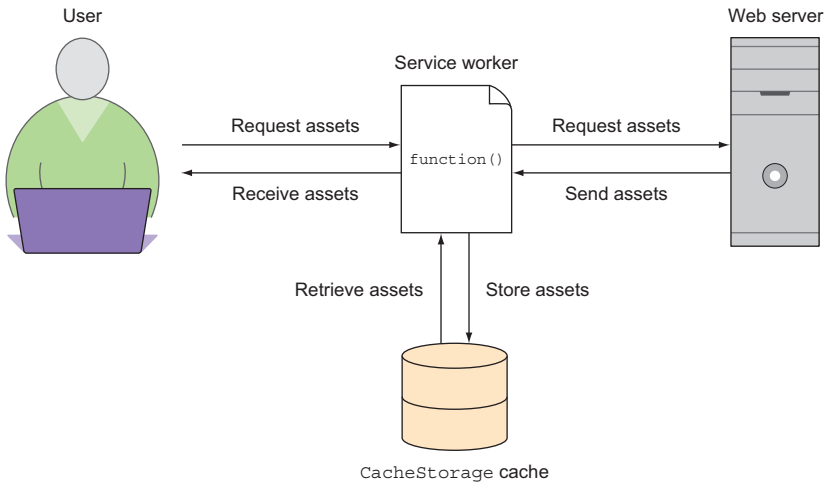

Figure 9.2 A service worker communicating as a proxy between a user and a web server. The user makes requests, which the service worker can intercept. Depending on how the service worker code is written, assets can be retrieved from the service worker's `CacheStorage` cache, or passed through to the web server. The service worker can also write to the cache in specific instances.

9.2 Writing your first service worker

In this section, you'll embark on writing your first service worker. First, you'll learn how to check whether a browser supports service workers, and if so, go about installing one. Then, you'll write the guts of your service worker and use it to intercept network requests. Finally, you'll measure the performance benefits that your service worker affords.

The project you'll write a service worker for is a static version of my blog. I've been trying to find new ways to squeeze more performance out of my site, while also allowing readers to read old content if they're offline. First, you'll grab a copy of the code from GitHub and get it running on your computer. To do this, enter the following commands:

```
git clone https://github.com/webopt/ch9-service-workers.git
cd ch9-service-workers
npm install
node http.js
```

> **Service workers require HTTPS!**
>
> For convenience, service workers can run on localhost without HTTPS. But because of the level of access that service workers have in terms of being able to intercept network requests and run in the background, HTTPS is required on a production web server. You have some leeway on localhost, but when you go to production, you'll *need* a valid SSL certificate.

When finished, the site will be running on your local machine at http://localhost:8080. After you verify that the site is running, you'll be ready to write your first service worker!

9.2.1 Installing the service worker

The installation process of a service worker requires little code. You need to check whether the browser supports service workers at all. If the browser supports them, you can continue with the installation. If the browser *doesn't* support service workers, nothing will happen. This ensures that your site will continue to function, even if the user's browser isn't capable of using service workers. Figure 9.3 illustrates this behavior flow.

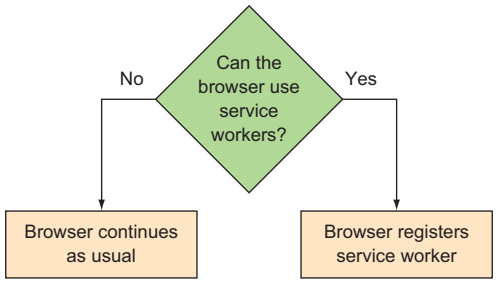

Figure 9.3 The service worker installation process. The code checks for the status of service worker support. If the browser supports it, the service worker is installed. If not, the browser does nothing.

The first part of installing your service worker involves registering it via the sw-install.js script referenced via a `<script>` tag in the footer of each page.

9.2.2 Registering the service worker

To get started with installing your service worker, you'll see a file named sw-install.js in the htdocs folder. Open this file in your text editor and enter the contents of this listing into it.

Listing 9.1 Service worker support detection and installation code

```
if("serviceWorker" in navigator){
    navigator.serviceWorker.register("/sw.js");
}
```

Sniffing out service worker support is easy. In the first line, we use the `in` operator to check for the existence of the `serviceWorker` object within the `navigator` object. If service workers are supported, the script in `/sw.js` is registered via the `serviceWorker` object's `register` method as shown in the second line.

> **A note on service worker scope**
>
> If you're curious as to why the service worker code isn't in the js directory, it's because of scoping. By default, a service worker is scoped to work only in the directory it resides in and its subdirectories. If you want it to work across the entire site, you need to place it in the site's root folder, which is what you'll do in this chapter's example. If you want to place your service worker in a more logical location, you can overcome this issue by setting the `Service-Worker-Allowed` HTTP response header to a value of `/`, which allows the service worker to work across the entire domain.

Don't reload the page and test your changes just yet! For your service worker to *do* anything, you need to write some of your service worker behavior, particularly what should occur when the service worker is first installed.

WRITING THE SERVICE WORKER'S INSTALL EVENT

As I said earlier in this chapter, the service worker interface is light and consists of events that you can attach code to by using the `addEventListener` method. When a service worker is first installed, the `install` event is fired.

When you install your first service worker, you want to immediately cache the global assets for the site. These are items such as the site's CSS, JavaScript, images, and any other asset that's common across all pages and devices. Figure 9.4 depicts this caching process.

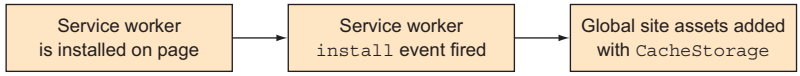

Figure 9.4 The behavior that you want to occur when the service worker's `install` event is fired

To get started writing the installation behavior of your service worker, open sw.js in the htdocs folder. The first thing you want to do is cache the page assets that are necessary for the site to run offline. These are usually the static pieces, such as CSS, JavaScript, and images. The following listing shows how to accomplish this important step.

Listing 9.2 Caching assets in the service worker's `install` event

```
var cacheVersion = "v1",
    cachedAssets = [
        "/css/global.css",
        "/js/debounce.js",
        "/js/nav.js",
       "/js/attach-nav.js",
  "/img/global/jeremy.svg",
        "/img/global/icon-github.svg",
        "/img/global/icon-email.svg",
        "/img/global/icon-twitter.svg",
        "/img/global/icon-linked-in.svg"
    ];

self.addEventListener("install", function(event){
    event.waitUntil(caches.open(cacheVersion).then(function(cache){
        return cache.addAll(cachedAssets);
    }).then(function(){
        return self.skipWaiting();
    }));
});

self.addEventListener("activate", function(event){
    return self.clients.claim();
});
```

Version identifier of asset cache.

URIs of assets you want to cache when service worker is installed

Fires when service worker is being installed.

New cache is opened with the identifier name set in the cacheVersion variable.

Cache is populated with cachedAssets array.

Tells service worker to immediately take effect and fire activate event.

Fires when the service worker's skipWaiting method fires.

When the activate event fires, the service worker will begin working immediately.

The installation code is a little tricky at first glance, but easy to grasp once you walk through it. First, you define a cache identifier in the `cacheVersion` string. This allows you to give your cache a name, and you can update it when the cache is changed in future versions of the service worker. The assets you want to cache up front in the service worker are then specified in the `cachedAssets` array.

Next, you write the `install` event code, which is executed as soon as the service worker is installed by sw-install.js. Here, you return a promise for opening a new `caches` object by the identifier you've set in the `cacheVersion` variable, and then add all of your assets specified in the `cachedAssets` array to it. The promise is chained with a `then` call that returns the result of the service worker's `skipWaiting` method. This instructs the service worker to immediately fire the `activate` event after the `install` event is complete. The `activate` event code then executes the service worker's `claim` method, which allows the service worker to begin working immediately.

With this code in place, *now* you can reload the page. When the page reloads, it seems like nothing has happened. So how can you tell whether the service worker is, well, working? You can verify in Chrome by opening the Developer Tools and navigating to the Application tab. Click the Service Workers item in the left pane, and you'll see something similar to figure 9.5.

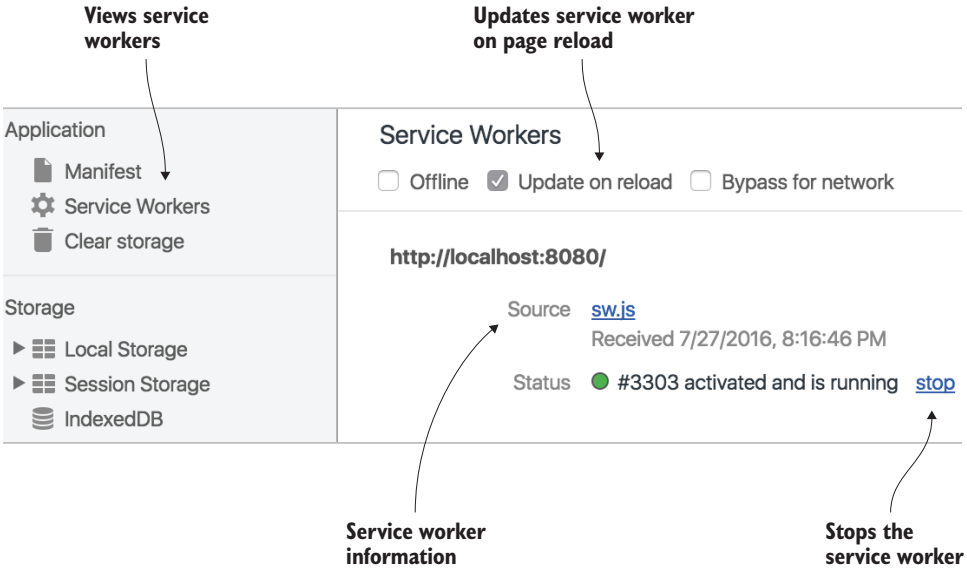

Figure 9.5 The Application tab in Chrome's Developer Tools showing active service workers for the current site. Click the Service Workers item in the left pane to access this panel.

When you open this panel in the Application tab, you'll be able to view the service workers currently running on the page, as well as perform actions such as stop a service worker, unregister it, and more important, force it to update on reload. During your work in this chapter, you should select the Update on Reload check box, which forces updates on page reload. Service workers can be tricky to work with as you develop them, and selecting this option simplifies the process.

Looking at the service worker cache

Now that you've verified that the service worker is installed, how can you tell whether the assets you've specified have been cached? The answer to this burning question is once again in the Application tab. In the left pane, expand the Cache Storage item and click the cache for your site, which is labeled v1, as specified in the `install` event code. You can see this in figure 9.6.

When you look at your v1 cache under the Cache Storage item in the Application tab, you can see all of the items that you've specified in the `cachedAssets` array. With these in your cache, let's see what happens if you go offline. To go offline, you could turn off your network connection on your machine by turning off Wi-Fi or

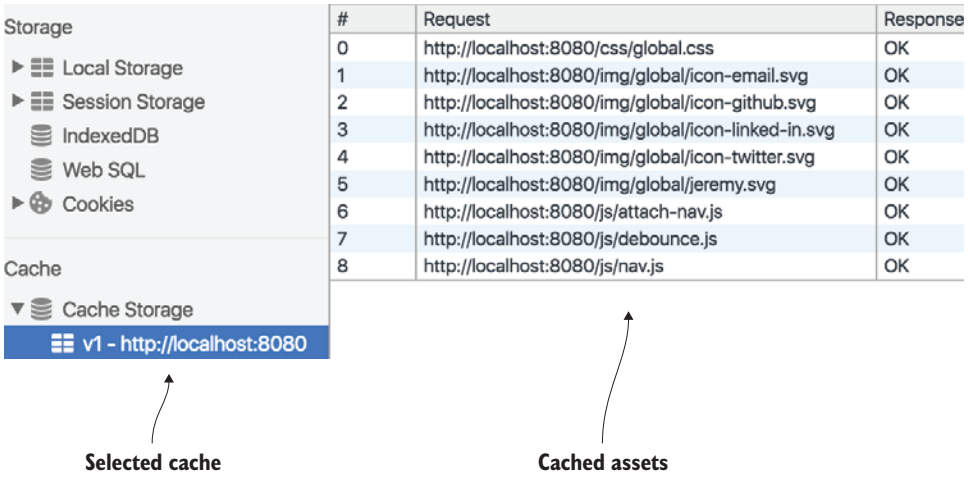

Figure 9.6 The v1 cache created by your service worker. You can see that the assets you've specified in the service worker's `cachedAssets` array are present.

unplugging your network cable, but there's an easier way. In Chrome's Developer Tools, go to the Network panel and locate the Offline check box, shown in figure 9.7.

Select the Offline check box and reload the page. You'll notice that, although you've cached all the necessary page assets for offline viewing, you still get a connection error and not the offline version of the site. Why is that?

Figure 9.7 Selecting the Offline check box in Chrome's Network panel allows you to simulate what it's like to be offline without having to disable your network connection.

In this case, it's because you haven't cached the HTML document itself. Even if you *did* do that, though, you'd still need a mechanism that intercepts network requests, and then you'd need to figure out what to *do* with them. Indiscriminately adding every asset to the `CacheStorage` cache up front *isn't* a viable strategy, because it loads a ton of stuff that the user may end up never needing. You *first* cache global assets common on all pages up front in the `install` event, and *then* use the `fetch` event to intercept and cache assets on an as-needed basis. This ensures that your visitors are efficiently caching everything you know they'll need up front, and then you can programmatically add assets to the cache after they request them.

9.2.3 Intercepting and caching network requests

To control what happens when you're offline, you need a mechanism that sits between you and the server that allows you to cache content for offline viewing. The `fetch` event allows this functionality via a behavior flow defined in figure 9.8.

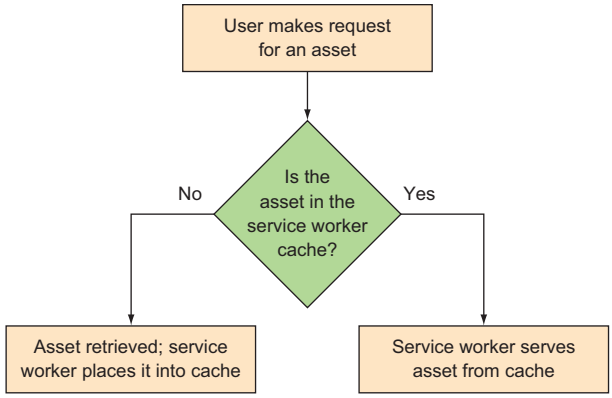

Figure 9.8 The behavior of the service worker's `fetch` event. The user makes a request for an asset, and the service worker steps in to intercept it to see whether the asset is already in the cache. If not, the asset is fetched from the network, and the service worker caches it. If it's in the cache, it's pulled from the cache.

You might be wondering why you bothered to cache assets during the `install` event. You're priming the cache up front with assets that you *know* you'll need. Whether or not you've cached an asset, however, you *still* need to define behavior for a `fetch` event that deals with the user's requests and caches assets for use later on. For assets that you've cached up front, the service worker will serve them from that cache. For assets that you're less certain about, such as HTML documents, images specific to articles, fonts, and so on, you want to subject them to a more rigorous check: a check that goes and fetches them from the network, and *then* places them into the cache for later use.

One good reason for this is that the assets you request may not be consistent across all devices, and thus shouldn't be added to the cache up front. A device with a high-density display will download images appropriate for *that* device, and should be cached programmatically according to the device's needs. A less capable device should download and cache assets appropriate to its own limitations.

You need to write your own logic to accomplish this goal. This logic is written in this listing. Add it to your local copy of sw.js.

Listing 9.3 Intercepting and caching additional assets in the `fetch` event

A regular expression of assets you're blacklisting from the service worker cache.

The fetch event listener that lets you intercept network requests.

A regular expression of hosts you'll intercept requests for.

```
self.addEventListener("fetch", function(event){
    var allowedHosts =
        /(localhost|fonts\.googleapis\.com|fonts\.gstatic\.com)/i,
        deniedAssets = /(sw\.js|sw-install\.js)$/i;

    if(allowedHosts.test(event.request.url) === true &&
       deniedAssets.test(event.request.url) === false){
        event.respondWith(
```

Before responding to a request, check that the request URL is from an accepted host, and isn't one of the blacklisted assets.

The event object's respondWith method intercepts the request. Bypassing this allows default browser behavior to occur.

If the item isn't in the service worker cache, use the fetch API to retrieve it.

If the item is in the service worker cache, serve the cached response.

Check if the item is in the service worker cache.

```
caches.match(event.request).then(function(cachedResponse){
    return cachedResponse ||
    fetch(event.request).then(function(fetchedResponse){
        caches.open(cacheVersion).then(function(cache){
            cache.put(event.request, fetchedResponse.clone());
        });

        return fetchedResponse;
    });
})
);
}
});
```

When the asset is fetched, open the service worker cache with the cacheVersion identifier.

The fetched asset is placed into the service worker cache.

After caching the response, return the original response.

With this code, you can fetch items from the cache that you've primed in the service worker's `install` event code. If you come across a request for an asset that isn't in the cache, you use the `fetch` method (covered in chapter 8) to retrieve it from the network. After the asset is downloaded, you place it in the service worker cache. Then that asset is retrieved from there instead of from the network. Force a reload of the page by using Ctrl-Shift-R (or Cmd-Shift-R on a Mac), and your changes should take effect.

Tip for stubborn service workers

Even when the Update on Reload check box is selected in the Application tab of Chrome's Developer Tools, a service worker can fail to update. This may be because of a lax caching policy that instructs the browser to hold the service worker in the cache. In our example, you have a `Cache-Control` header value of `no-cache`, which instructs the browser to revalidate the stored copy with the server for changes. To learn more about `Cache-Control` and how it works, check out chapter 10.

With the updated service worker running, open the Network tab and check out the asset information in the Size column. Those that have been intercepted by your service worker will read *(from ServiceWorker)*, as shown in figure 9.9.

Name	Method	Status	Domain	Size
localhost	GET	200	localhost	(from ServiceWorker)
css?family=Fjord+One|Montserrat:4...	GET	200	fonts.googleapis.com	(from ServiceWorker)
global.css	GET	200	localhost	(from ServiceWorker)

Requests intercepted by the service worker

Figure 9.9 Network requests intercepted by the service worker will be indicated by a value of "(from ServiceWorker)" in the Size column in Chrome's network utility.

Now that you've verified that items are being cached by the service worker by way of the `CacheStorage` API, select the Offline check box in the network utility next to the network throttling drop-down, and then reload the page. Rather than receiving a network error, you'll notice that the site is now available offline. Congratulations! You just wrote your first offline web experience! Now let's look at how these changes have affected the performance of the page.

9.2.4 Measuring the performance benefits

When you retrieve assets from the service worker cache, you can achieve better performance than the browser cache. This means that you can further accelerate rendering performance for the user by lowering the amount of time it takes for the browser to begin painting the page. Figure 9.10 tracks the Time to First Paint of three scenarios: when the browser has nothing in its cache, when the cache is populated, and when the service worker cache is used instead of the browser cache.

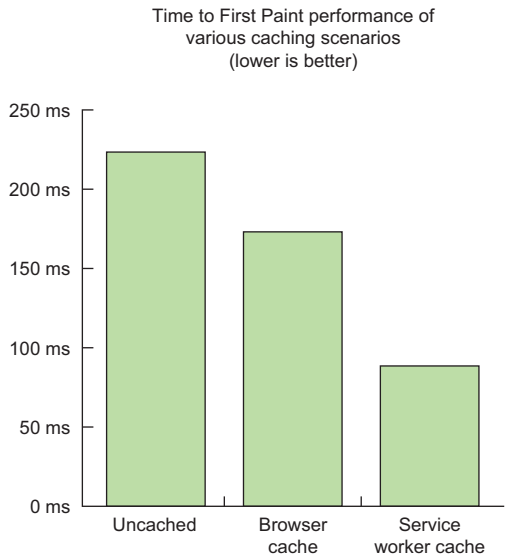

Figure 9.10 A comparison of the Time to First Paint performance of various caching scenarios on Chrome's Regular 3G throttling profile. The scenarios are an uncached page, the page when retrieved by the browser cache, and the page when retrieved from the service worker cache.

This was somewhat surprising to me, but the numbers show that when service workers are used to enhance performance, you can see a 50% improvement over the browser's caching behavior. That's a sizeable decrease in rendering time!

This doesn't mean the browser cache is dead. You still need it, because it works so well and because you can fall back to it in browsers that don't support service workers. Even in browsers that *do* support service workers, you can configure your `fetch` event code to ignore requests that you *don't* want to intercept, at which point your requests will fall to the browser cache.

Next, you'll take another look at your service worker code, and see how to tweak it to be a little more flexible.

9.2.5 Tweaking network request interception behavior

So you've written your first service worker, and it works pretty well. Except for one thing: If you try to change any of your assets, those changes won't be seen unless you force a reload of the entire page. This is problematic. In particular, it's important that your site's HTML be as up-to-date as possible, so that if you update references to things like CSS or JavaScript, you'll be able to see those changes as well as updates to content.

That's not to say that service workers are inherently problematic, or that using `CacheStorage` over the browser cache is a bad idea. If you want to provide an offline experience, it's the only real way to do so. When you intercept and change the way network requests are fulfilled, however, you're writing behavior that supersedes the browser's own built-in cache. You must be mindful of how you choose to fulfill these requests.

How you fulfill these requests depends on the nature of your website. In the case of the blog example in this chapter, the strategy is basic: assets that don't change often (such as images, scripts, and CSS), you don't worry about right now. When you *do* need to change them, there's a mechanism for doing so that's covered in the next section.

For HTML, you adopt a different strategy in your service worker's `fetch` event code that will allow you to get the latest page content every time when you're online. You can still accommodate offline viewing for your user as a part of a fallback strategy.

Your current service worker `fetch` event code is straightforward: if a request for an asset matches something that's already in the service worker cache, you serve the asset from the cache. If the request doesn't match anything in the cache, you grab the latest copy from the network, and then add it to the cache.

This is an excellent strategy for performance, but it can negatively impact content freshness. You don't want to abandon this strategy altogether, because some assets rarely ever change. You want to adopt a new approach for HTML documents. Figure 9.11 shows how to adopt a two-pronged approach to intercepting network requests in the service worker's `fetch` event.

With this flow, you're maintaining the performance advantages that service workers give us for assets such as images, CSS, and JavaScript, but giving priority to the freshness of HTML content. This gives you the ability to update site assets by changing their URLs, which you can later point to in the cache that you populate in the service worker's `install` event code.

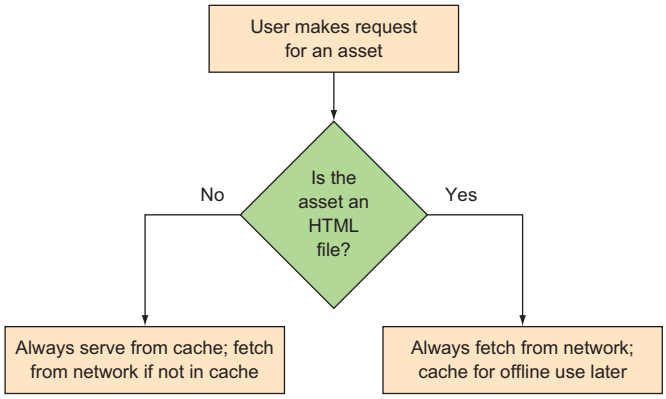

Figure 9.11 A two-pronged approach for intercepting a network request in a service worker's fetch event. If the requested asset is an HTML document, you always fetch it from the network and place it in the cache, and serve it from the service worker cache only if you're offline. If the resource isn't an HTML document, you always serve from the cache and retrieve it from the network if it's not in the service worker cache.

To implement this new flow, you need to update the service worker's `fetch` event code. First, you need to add a new regular expression to check whether the incoming request is an HTML document. Changes are in bold.

Listing 9.4 Adding a regular expression to check for HTML requests

```
var allowedHosts = /(localhost|fonts\.googleapis\.com|fonts\.gstatic\.com)/i,
    deniedAssets = /(sw\.js|sw-install\.js)$/i,
    htmlDocument = /(\/|\.html)$/i;
```

A regular expression that checks if a URL is an HTML document.

This regular expression will be used later to check whether the request URL is for an HTML document, and will be what your network interception request behavior hinges on. From there, you'll create a new condition with this regular expression. If the regular expression test passes for the current request, and you're dealing with an HTML document, you'll handle the request in your pattern of network first/serve from cache for offline. If the test fails, you'll use your initial pattern of cache first/populate cache from the network.

Listing 9.5 Handling HTML requests with a network first/cache for offline pattern

Checks if the current request is for an HTML document

event.respondWith intercepts the request and responds with the encapsulated code.

```
if(allowedHosts.test(event.request.url) === true &&
   deniedAssets.test(event.request.url) === false){
    if(htmlDocument.test(event.request.url) === true){
        event.respondWith(
            fetch(event.request).then(function(response){
                caches.open(cacheVersion).then(function(cache){
                    cache.put(event.request, response.clone());
                });
                return response;
            }).catch(function(){
                return caches.match(event.request);
            })
        );
    }
    else{
        /* Handle non-HTML requests as before */
    }
}
```

Opens the cache and places the network response into it for offline use later

Fetches the HTML document immediately from the network

Responds with the asset retrieved from the network

If a later request fails, the last version obtained is served from the cache.

Other assets are handled as previously defined.

After you reload the page and try the new code, go ahead and modify index.html. You'll notice that updates to it should be reflected immediately.

This method slows rendering performance of the page *slightly* as compared to your earlier `fetch` event code, because the document needs to be fetched from the network rather than read from the service worker cache. Testing under the Regular 3G throttling

profile in Chrome shows an average Time to First Paint of 120 ms. Although slower than the average time of 90 ms that your earlier service worker code yielded, it's still faster than the browser cache's average Time to First Paint of approximately 175 ms.

The results you get will depend on your specific project. Keep in mind that you don't *need* to intercept *every* network request, nor *should* you. Remember that network requests in your `fetch` event code are intercepted only if you pass a response object to the `event` object's `respondWith` method. If a request isn't passed to this method, the browser's default behavior will kick in. The server worker specification doesn't prescribe any method for handling requests; it only provides an interface for you to do so. *You* dictate the logic with respect to whether a request is intercepted. In this chapter's example, you've already done quite a bit of this by creating regular expressions to filter out requests that you don't want to intercept.

> **Your service worker and CDN-hosted assets**
>
> CDN-hosted assets are another aspect of request interception, and so is caching them with `CacheStorage`. Generally speaking, you'll be able to save CDN-hosted assets to your service worker cache without trouble. CDN hosts configure their servers to serve assets with an `Access-Control-Allow-Origin: *` header, which allows any origin to access those resources without restriction. If you're having trouble caching a CDN asset, check for the presence of this header. All properly configured CDNs supply this header, so adding special logic in your service worker to work with these assets is something you won't need to worry about. For further information on CDNs and how they work, check out chapter 10.

Back to your service worker: even though you're sacrificing performance a bit by hitting the network for HTML requests, the net effect over the browser cache is positive. Better yet, this method still lets you serve content to users when they're offline. It's the best of both worlds.

Of course, if you modify the site's CSS, JavaScript, or images, those will still be served from the service worker cache, and updates won't be reflected. There's a good approach to dealing with those assets, and next we cover how to update your service worker cache to include changes to your site assets.

9.3 Updating your service worker

So far, you've written a service worker that caches site assets such as CSS, JavaScript, and images, and always fetches a fresh copy of HTML files from the server. Let's imagine that you've pushed this service worker code to production, and it's working great.

Unfortunately, you've hit a snag in that you have new CSS that you need to make sure your users see, but the old CSS file cached by the service worker is stubbornly persistent, and updates only if you force a reload of the entire page. This is problematic, because it's not a good solution to tell your users, "Just force a reload of the page to see the new styles!" You need to be able to usher in the changed CSS and automatically put it in the user's service worker cache.

In this short section, you'll learn how to version files on your site so that the service worker picks those files up instead. Because you want to keep your caches lean out of respect for your user's device storage quota, I'll then show you how to clear out the old cache. Let's begin!

9.3.1 Versioning your files

You'll recall that you wrote your service worker `fetch` event code to always serve files such as CSS, JavaScript, and images from the service worker cache, but to always prefer fetching HTML from the network if the user is online. One good reason for this is so that you can force updating of other asset types by versioning the references to them in the HTML file. Because the HTML will always be fetched from the server, you can ensure that any new references to assets will be downloaded by the user.

In regards to caching, versioning occurs when you take a file and modify the reference to it. Take global.css, for example, which is included in index.html via the `<link>` tag:

```
<link rel="stylesheet" href="/css/global.css" type="text/css">
```

This approach is familiar to you by now. It's a useful one-liner that instructs the browser to download global.css. What happens if you make a change to global.css, though? Your service worker will never pick up on it, because it's already been cached. In fact, depending on the caching policy for that file, even the native browser cache may never pick up new changes.

This is where the concept of versioning comes in. By adding a query string to the filename as shown here in bold, you can differentiate the asset from its previous version:

```
<link rel="stylesheet" href="/css/global.css?v=1" type="text/css">
```

Even though the asset's filename is the same, the query string is enough of a differentiator for the browser to trigger it to download the file again, and treat it differently than the asset that doesn't have the query string.

> **Query strings in browser caches**
>
> The query string trick isn't handy only when trying to bust the service worker cache; it also works for the browser's native cache. Chapter 10 covers this trick in more depth, as well as automating this process to make repeated changes a lot less tedious.

To test this out, make a small but noticeable change in global.css—something like changing the `background-color` of the `<body>` element. If you open index.html and change the `<link>` tag reference to global.css to add the query string as shown previously, you'll notice that the new styles kick in immediately. Success! Or so you think? Maybe you should check out the Cache Storage section under the Application tab in Chrome's Developer Tools and look at your v1 cache. You'll see something like figure 9.12.

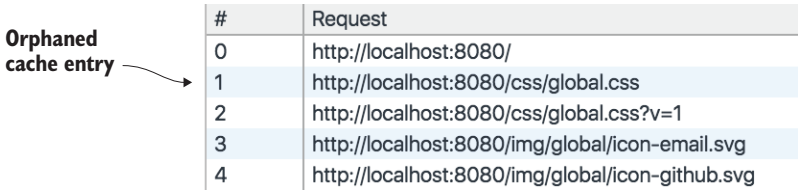

Figure 9.12 An orphaned cache entry after updating the style sheet reference. `global.css?v=1` is in the cache, whereas the unused global.css entry remains.

Although leaving this orphaned entry in the cache isn't going to kill anyone's user experience, it's not a good idea to leave it and move on. Think of it like littering. Is one candy bar wrapper tossed on the ground going to kill the world? Obviously not, but it's a bad thing to do, and you should always clean up after yourself.

Think of orphaned cache entries such as these as being like candy bar wrappers and bottles alongside the highway. Over time, your service worker cache will become bloated and take up unnecessary space on the user's device. In the next section, you'll learn how to clean up after yourself like a proper person.

9.3.2 Cleaning up old caches

Now that you've found out how to bypass a stubborn service worker cache, you need to learn how to remove outdated caches from it. The first thing you need to do is bump up the `cacheVersion` variable from `v1` to `v2`, and replace your reference to global.css to read `global.css?v=1` in your `cachedAssets` array. These changes are in bold.

Listing 9.6 Updating the cache name and the assets to cache

```
var cacheVersion = "v2",                    <-- New cache name
    cachedAssets = [
        "/css/global.css?v=1",              <-- Updated reference to the new CSS file
        "/js/debounce.js",
        "/js/nav.js",
        "/js/attach-nav.js",
        "/img/global/jeremy.svg",
        "/img/global/icon-github.svg",
        "/img/global/icon-email.svg",
        "/img/global/icon-twitter.svg",
        "/img/global/icon-linked-in.svg"
    ];
```

These changes alone are enough for the new cache to take effect, but it's not enough for the old v1 cache to be removed. You'll take care of that by rewriting the entire `activate` event code, which you can see in the following listing.

Listing 9.7 Removing old caches in the `activate` event

```
self.addEventListener("activate", function(event){
    var cacheWhitelist = ["v2"];

    event.waitUntil(
        caches.keys().then(function(keyList){
            return Promise.all([
                keyList.map(function(key){
                    if(cacheWhitelist.indexOf(key) === -1){
                        return caches.delete(key);
                    }
                }), self.clients.claim()
            ]);
        })
    );
});
```

A white list of the caches you want to keep.

A promise that gives access to all available service worker caches

The cache names are iterated over in the keyList array.

A promise that waits for multiple conditions to be fulfilled

Checks if the current cache key isn't in the white list

If the cache key isn't in the white list, it's deleted.

Allows the service worker to begin working on the page immediately

With this new `activate` event code, your service worker will process everything in your new cache. If any of the caches in the service worker don't go by any name specified in the `cacheWhitelist` variable, they'll be removed. After you run this code, go to the Cache Storage section in the left pane of Chrome's Application tab in the Developer Tools. You should be able to see that the only cache left is the new v2 cache, as shown in figure 9.13.

At this point, you'd proceed to update every reference to global.css to `global.css?v=1`. If you fail to do this, navigating to a subsequent page would store a separate cache entry for the old URI. This isn't a step you need to complete as part of the work in this chapter; it's more of a caveat for when you implement service worker changes on your own site.

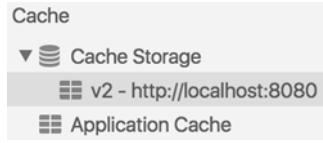

Figure 9.13 Your new v2 cache. If you click this, you'll be able to see the updated cache contents, particularly the `global.css?v=1` entry.

You're are the end of your work in this chapter. Let's quickly recap what you've learned before moving on to the next chapter.

Going further with service workers

Covering all capabilities of service workers is outside the performance-oriented scope of this chapter. In fact, service workers are capable of more than creating offline experiences and boosting site performance. Although the patterns in this chapter are useful for content-driven sites such as blogs and the like, a few resources out there can help you take service workers a little (or a lot) further:

(continued)

- Jake Archibald, a developer advocate for Google, has written a great article titled "The Offline Cookbook" available at https://jakearchibald.com/2014/offline-cookbook. It's a resource for patterns that you can use in your service worker. Some patterns are performance-oriented, some favor flexibility for offline experiences, and quite a few others fall in between. If you're wondering where to begin writing a service worker that makes the most sense for your website, this is a great place to start.
- Mozilla has created its own service workers cookbook at https://serviceworke.rs. It covers a broad spectrum of possibilities with the technology, including how to use service workers to send push notifications to mobile devices (for real!).

Throughout this chapter, you've made heavy use of the `CacheStorage` object, particularly methods such as `caches.match` and `caches.open`, which match items in a local cache and open caches by name, respectively. Although it works great with service workers, this API is a standalone piece of functionality with many methods available for use. To learn more about `CacheStorage`, check out the Mozilla Developer Network reference at http://mng.bz/NVXR.

In the first section of this chapter, recall that I said that communication between a service worker and its parent page isn't possible unless you use the `postMessage` API. Learn about this technology at Google Chrome's GitHub site at http://mng.bz/De31.

9.4 Summary

As you've witnessed in this chapter, service workers can be a tool used to enhance the performance of a website. Specifically, you learned the following key concepts:

- Service workers are a kind of JavaScript worker that operates on a thread separate from the main processing thread on which all other scripting activity occurs.
- Installing a service worker is easily done in browsers that support service workers. You check for the `serviceWorker` member in the `navigator` object. If a browser doesn't support the technology, the page experience will continue on without the service worker functionality.
- Because progressive enhancement is necessary to provide a level of functionality to all users, it's important that your site doesn't explicitly *depend* on service workers. Service workers are an *enhancement*, and should not be a *requirement* for a site to work.
- Service workers require HTTPS to be used. Although you can develop and use a service worker over HTTP on localhost in development, make sure you have a valid SSL certificate for when you push your service worker to a production server.

- Using `CacheStorage` in tandem with the service worker's `fetch` event, you have a vast amount of power and flexibility in intercepting and caching network requests. The marriage of these two pieces of functionality allows you to enhance page performance by serving items directly from the service worker cache, as well as fall back to an offline experience when a user's internet connection is absent or intermittent.
- Although the service worker cache operates similarly to the browser cache, it's a separate entity. If you don't intercept and send network requests to the `fetch` event's `event.respondWith` method, the responses to those requests will be handled according to the browser's default behavior.
- Service workers can provide a performance boost when it comes to rendering. In the case of my blog, it provided nearly a 50% boost in rendering speed over the browser cache!
- Aggressive caching of HTML documents can create a scenario where it becomes difficult to update HTML content on a page, as well as the assets referenced on the page, such as CSS, JavaScript files, and images. For content-driven sites such as a blog, it makes more sense to fetch these assets from the network and *then* cache them in case the user goes offline later.
- Sometimes site assets change, and you need to invalidate your service worker's cache. If this happens, you can invalidate a service worker cache by adopting a new name for the cache, and dropping it into a white list. From here, you can use the service worker's `activate` event to eliminate all caches that aren't a part of your white list, ensuring easy updates and cleanup when assets are changed on your site.

In the next chapter, you'll explore methods you can use to fine-tune the delivery of assets on your website, ranging from configuring your site's browser caching policy, providing resource hints, working with CDNs, and more.