SAMPLE CHAPTER

Progressive Web Apps

Dean Alan Hume

FOREWORD BY Addy Osmani

MANNING

Progressive Web Apps

by Dean Alan Hume

Chapter 4

brief contents

4 Intercepting network requests

Chapter 3 looked into using Service Worker caching to dramatically speed up the performance of your website. Instead of the user making a request to the server, the Service Worker intercepts the request and decides to serve it from cache instead. We also briefly touched on how to use Service Workers to transform the requests and responses made by the client using the `fetch` event.

In this chapter, we'll dive deeper into the `fetch` event and you'll learn more about the many use cases it offers. Service Workers are the key to unlocking the power that lies within your browser. By the end of the chapter, you'll know how to serve lighter, leaner web pages depending on your user's browser or preferences. In this section of the book, we're focusing on the *faster* part of Progressive Web Apps (PWAs), although it's important to also ensure that your web apps are resilient and engaging, too.

4.1 The Fetch API

As web developers, we often need the ability to retrieve data from the server in order to update our applications asynchronously. Traditionally, this data is retrieved using JavaScript and the `XMLHttpRequest` object. Otherwise known as AJAX, this is a developer's dream because it allows you to update a web page without reloading the page by making HTTP requests in the background. In our sample application, Progressive Times, you use this code to retrieve a list of news articles.

If you've ever implemented complex logic to retrieve data from the server, writing code using the `XMLHttpRequest` object can be quite tricky. As you start to add more and more logic and callbacks, it can quickly become messy, as you can see in the following listing.

Listing 4.1 HTTP eequest using the `XMLHTTPRequest` object

```
var request;
if (window.XMLHttpRequest) {
     request = new XMLHttpRequest();
} else if (window.ActiveXObject) {
     try {
          request = new ActiveXObject('Msxml2.XMLHTTP');
     } catch (e) {
            try {
                    request = new ActiveXObject('Microsoft.XMLHTTP');
             } catch (e) {}
     }
}

request.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        doSomething(this.responseText);
    }
  };

// Open, send.
request.open('GET', '/some/url', true);
request.send();
```

The code in listing 4.1 seems like a lot of code to make an HTTP request. The interesting thing is that the `XMLHttpRequest` object was originally created by the developers of Outlook Web Access for Microsoft Exchange Server. After a number of permutations, it eventually became the standard for what we use today to make HTTP requests in JavaScript. The example in the listing fulfills its purpose, but it isn't as clean as it could be. The other problem with the code in the listing is that the more complex your logic becomes, the more complex this code will become. In the past, a number of libraries and techniques were available to make this code simpler and easier to read, with popular libraries such as jQuery and Zepto, including cleaner APIs.

Fortunately, modern browser vendors have realized that this situation needed to be updated, and this is where the Fetch API comes in. The Fetch API is a part of the Service Worker global scope, and you can use it to make HTTP requests inside any Service Worker. Up until now, you've been using the Fetch API inside your Service Worker code, but we haven't dived deeper into it. Let's look at a few code examples in order to get a better understanding of the Fetch API, beginning with the following listing.

Listing 4.2 An HTTP Request Using the Fetch API

```
fetch('/some/url', {
     method: 'GET'
}).then(function(response) {
     // success
}).catch(function(err) {
     // something went wrong
});
```

The URL to access using a GET request

If successful, return the response.

If something went wrong, you can respond appropriately.

The code in listing 4.2 is a basic example of the Fetch API in action. You might also notice that there are no callbacks and events—they've been replaced with the `then()` method. This method is part of ES6's new promises functionality and aims to make your code much more readable and easier for developers to understand. A *promise* represents the eventual result of an asynchronous operation, even if the value won't be known until the operation completes at some point in the future.

Listing 4.2 seems easy enough to understand, but what about a `POST` request using the fetch API? Check out the next listing.

Listing 4.3 An HTTP POST request using the Fetch API

```
fetch('/some/url', {
        method: 'POST',
        headers: {
          'auth': '1234'
        },
         body: JSON.stringify({
        name: 'dean',
        login: 'dean123',
      })
    })
    .then(function (data) {
      console.log('Request success: ', data);
    })
    .catch(function (error)
      console.log('Request failure: ', error);
    });
```

The URL to access using a POST request

Headers can be included in the request.

The body of the POST request

If successful, return the response.

If something went wrong, you can respond appropriately.

Say you wanted to send some user details to the server and needed to do so using a `POST` request. In listing 4.3, you change the method to `POST` and add a `body` parameter in the `fetch` options. Not only does using promises make your code cleaner, it also allows you to chain code together to share logic across `fetch` requests.

The Fetch API is currently available in all browsers that support Service Workers, but if you intend to use this API on browsers that aren't supported, you may want to consider using a polyfill. A *polyfill* is a piece of code that provides you with the functionality you expect from a modern browser. For example, if the latest version of Internet Explorer has some functionality you need, but it doesn't exist in an older version, you can use a polyfill to provide similar functionality for the older browser. Think of it as a wrapper around an API that's used to keep the API landscape flattened. A polyfill written by the team at GitHub (https://github.com/github/fetch) will ensure that older browsers are able to make requests using the Fetch API. Include it in your web page and you'll be able to start writing code using this API.

4.2 The fetch event

A Service Worker's ability to intercept any outgoing HTTP requests is what makes it so powerful. Every HTTP request that falls within this service worker's scope will trigger

this event—for example, HTML pages, scripts, images, CSS, and so on. This gives you as a developer total control over how you want to handle the way the browser responds to any of these `fetch`es.

In chapter 1, we looked at a basic example of the `fetch` event in action. Remember the unicorn (shown in the next listing)?

Listing 4.4 The `fetch` event inside a Service Worker

```
self.addEventListener('fetch', function(event) {
  if (/\.jpg$/.test(event.request.url)) {
    event.respondWith(
      fetch('/images/unicorn.jpg'));
  }
});
```

Add an event listener to the fetch event.

Check to see whether the HTTP request URL requests a file ending in .jpg.

Try to fetch an image of a unicorn and respond with it instead.

In listing 4.4, you're listening out for the `fetch` event, and if the HTTP request is for a JPEG file, you're intercepting it and forcing it to return a picture of a unicorn instead of its original intended URL. The code here will do this for each and every HTTP request made for a JPEG file from the website. For any other file types, it will ignore them and move on.

Although the code in listing 4.4 is a fun example, it doesn't show you what Service Workers are capable of. Let's take this a step further and see how to return your own custom HTTP response, as shown in the following listing.

Listing 4.5 Creating a custom HTTP response inside a Service Worker

```
self.addEventListener('fetch', function(event) {
  if (/\.jpg$/.test(event.request.url)) {
    event.respondWith(
    new Response('<p>This is a response that comes from your service
     worker!</p>', {
       headers: { 'Content-Type': 'text/html' }
      });
    );
  }
});
```

Add an event listener to the fetch event.

Check to see whether the HTTP request URL requests a file ending in .jpg.

Build a custom Response and respond accordingly.

In listing 4.5, the code intercepts any HTTP requests by listening for the fetch event to be triggered. Next it determines if the incoming request is for a JPEG file, and if it is, it will respond with a custom HTTP response. Using Service Workers, you can build your own custom HTTP responses, including editing their headers. This functionality makes Service Workers extremely powerful—which is why you can understand that they need to serve requests over HTTPS. Imagine the malicious things a hacker could get up to with this at their fingertips.

4.2.1 The Service Worker lifecycle

Right at the beginning of the book in chapter 1, you learned about the Service Worker lifecycle and the role it plays when building PWAs. Let's look closely at that diagram again in figure 4.1.

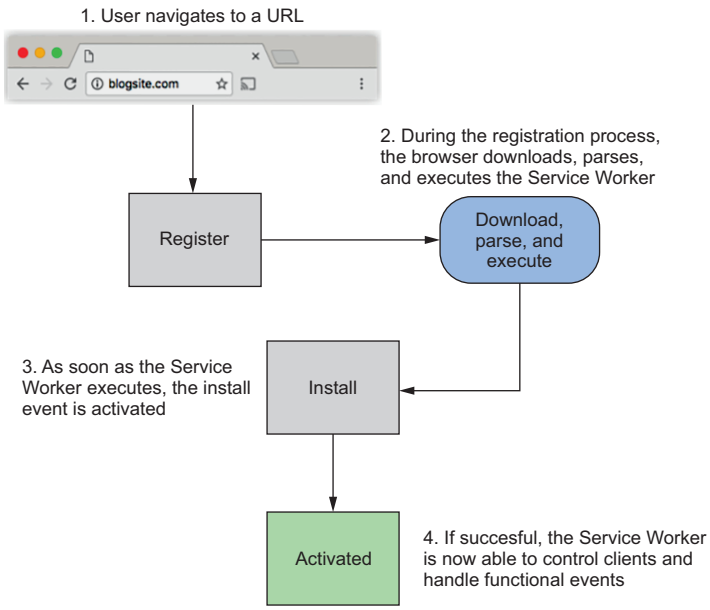

Figure 4.1 Lifecycle of a Service Worker

Looking at figure 4.1, you'll remember that when a user visits your website for the first time, they don't have an active Service Worker controlling the page. Only once the Service Worker has been installed and they refresh the page, or navigate to another part of the site, does the Service Worker become active and start intercepting requests.

To explain this more clearly, imagine a Single Page Application (SPA) or a web page with AJAX interactions that might take place after a page has been loaded. When you register and install a Service Worker using the method you've been using in the book up until now, any HTTP requests that take place after the page has loaded will be missed. Only when the user reloads the page will the Service Worker become active and start intercepting requests. This isn't ideal because ultimately you want the Service Worker to start working its magic as soon as possible and include these requests that are made while the Service Worker isn't active.

If you want your Service Worker to start working immediately instead of waiting for the user to navigate to another part of your site or reload the page, there's a sneaky little trick that you can use to activate your Service Worker immediately, shown in the following listing.

Listing 4.6 Install the current Service Worker without waiting for reload

```
self.addEventListener('install', function(event) {
  event.waitUntil(self.skipWaiting());
});
```

The code in listing 4.6 sits inside the `install` event of your Service Worker. By using the `skipWaiting()` function, you're ultimately triggering the `activate` event and telling the Service Worker to start working immediately without waiting for the user to navigate or reload the page.

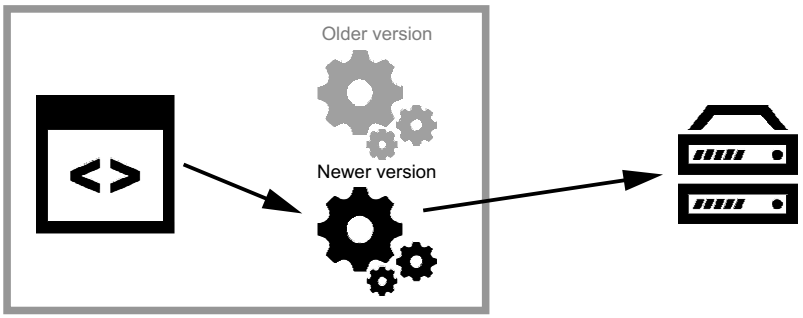

Figure 4.2 `self.skipWaiting()` causes your Service Worker to kick out the current active worker and activate itself as soon as it enters the waiting phase.

The `skipWaiting()` function forces the waiting Service Worker to become the active Service Worker. The `self.skipWaiting()` function can also be used with the `self.clients.claim()` function to ensure that updates to the underlying Service Worker take effect immediately.

The code in the next listing can be combined with the `skipWaiting()` function in order to ensure that your Service Worker activates itself immediately.

Listing 4.7 Activate a Service Worker immediately

```
self.addEventListener('activate', function(event) {
  event.waitUntil(self.clients.claim());
});
```

The code in listings 4.6 and 4.7 can be used together to kick-start the activation of your Service Worker. If your site has complex AJAX requests taking place once the page has loaded, these functions are perfect. If your site serves mostly static pages without HTTP requests taking place once the page has loaded, you may not need to use these functions.

4.3 Fetch in action

As we've seen in this chapter, Service Workers offer almost unlimited control of the network. Intercepting HTTP requests, editing HTTP responses, and crafting your own responses are a small part of what you can do by tapping into the `fetch` event.

Up until now, most of the code samples we've looked at haven't been real-world examples. In the next section, we're going to dive in to two useful techniques you can use to make your website faster, more engaging, and resilient.

4.3.1 An example using WebP images

Images play an important role on the web today. Imagine a world without images on our web pages. High-quality images can make a website stand out, but unfortunately they come with a price. Due to their large file sizes, they're bulky to download and result in slow page load times. If you've ever been on a device with a poor network connection, you'll know how frustrating this experience can be.

You may be familiar with the image format WebP. Developed by the team at Google, WebP files are 26% smaller than PNG images and around 25–34% smaller than JPEG images. That's a pretty decent savings, and the best thing about them is that the image quality isn't noticeably affected when choosing this format, as you can see in figure 4.3.

Figure 4.3 WebP images are significantly smaller in file size compared to their original format with little noticeable difference to the quality of the image.

Figure 4.3 shows a WebP image next to its equivalent JPEG image with negligible difference to image quality. By default, WebP images are supported in Chrome, Opera, and Android, but unfortunately not by Safari, Firefox, or Internet Explorer.

Browsers that support WebP images notify you of that fact by passing through an `accept: image/webp` header with each HTTP request. Given that you have Service Workers at your disposal, this seems like a perfect opportunity to start intercepting requests and returning lighter, leaner images to browsers that can render them.

The basic web page in the following listing references an image of Brooklyn Bridge in New York.

Listing 4.8 A basic HTML web page including a JPEG image

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Brooklyn Bridge - New York City</title>
 </head>
 <body>
<h1>Brooklyn Bridge</h1>
<img src="./images/brooklyn.jpg" alt="Brooklyn Bridge - New York">
      <script>
          // Register the service worker
            if ('serviceWorker' in navigator) {
            navigator.serviceWorker.register('./service-
     worker.js').then(function(registration) {
 // Registration was successful
console.log('ServiceWorker registration successful with scope: ',
     registration.scope);
            }).catch(function(err) {
              // registration failed :(
                   console.log('ServiceWorker registration failed: ', err);
                });
            }
     </script>
 </body>
</html>
```

That image is in JPEG format and comes in at 137 KB. If you convert it to WebP and store it on the server, you can choose to return this for browsers that support it and fall back to the original for those that don't.

The next listing shows code in your Service Worker that you can use to start intercepting the HTTP request for this image.

Listing 4.9 Service Worker Code to Return WebP Images if the Browser Supports It

```
"use strict";

// Listen to fetch events
self.addEventListener('fetch', function(event) {

  if (/\.jpg$|.png$/.test(event.request.url)) {

    var supportsWebp = false;
    if (event.request.headers.has('accept')) {
      supportsWebp = event.request.headers
```

Check whether the incoming request is for an image of type JPEG or PNG.

Inspect the accept header for WebP support.

```
        .get('accept')
        .includes('webp');
    }

    if (supportsWebp) {                                  Does the browser
      var req = event.request.clone();                   support WebP?

      var returnUrl = req.url.substr(0, req.url.lastIndexOf(".")) + ".webp";

      event.respondWith(
        fetch(returnUrl, {
          mode: 'no-cors'
        })
      );
    }
  }
});
```

Build the return URL.

There's a lot of code going on in listing 4.9. Let's step back and break it down further. In the first few lines, you're adding an event listener to listen out for any `fetch` events that take place. For each HTTP request that takes place, you check to see whether the current request is for a JPEG or PNG image. If you know the current request is for an image, you can then determine the best content to return based on the HTTP headers that are passed through. In this case, you're inspecting each header and looking for the `image/webp` mime type. Once you know the header values, you can determine whether the browser supports WebP images and return the corresponding WebP image.

Once the Service Worker has activated and is ready, any requests for a JPEG or PNG image will be returned as its WebP equivalent for any browsers that support it. If the browser doesn't support WebP images, it won't advertise the support in the HTTP request header, and the Service Worker will ignore the request and work as normal.

The WebP equivalent comes in at 87 KB, and compared to its JPEG equivalent, you've managed to save 59 KB—around 37% of the original file size. For users on a mobile device, this could add up to a big bandwidth saver across your site.

Service Workers open up a world of endless possibilities, and this example could be extended to include other image formats, and even caching. You could easily add support for Internet Explorer's improved image format called JPEGXR. There's no reason why you can't reward your users with fast web pages right now.

4.3.2 An example using the Save-Data header

I was recently travelling abroad when I urgently needed to get some information from my airline's website. I was on a sketchy 2G connection that took forever to load the page and eventually I gave up completely. I was also paying a fortune for this daily service from my mobile provider back home—so frustrating!

4G-network coverage is rapidly accelerating worldwide, but there's still a long way to go. 3G networks were only launched in late 2007 in countries such as Bangladesh, Brazil, China, India, Nigeria, Pakistan, and Russia—where almost 50% of the global

population is located.[1] Although mobile coverage is growing, it's crazy to think that a 500 MB data plan can cost around 17 hours' worth of minimum wage work in India.[2]

Fortunately, browser vendors such as Google Chrome, Opera, and Yandex have realized the pain that many users face. With the latest versions of these browsers, users can opt-in to a feature that will save them data. Once this feature is enabled, the browser will add a new header to each HTTP request. Developers can look out for this header and return the appropriate content to save users data. For example, if a user has opted-in to save data, you could return lighter images, smaller videos, or even different markup. It's a simple concept, but effective.

This sounds like a perfect situation to use a Service Worker. In the next section, you'll build code that will intercept whether or not a user has opted-in to save data and return a lighter version of your PWA.

Remember the PWA you built in chapter 3? Called Progressive Times, it contains a collection of funny news facts from around the world (figure 4.4).

Figure 4.4 The Progressive Times sample application is a basic app we'll revisit throughout the book.

[1] https://gsmaintelligence.com/research/2014/12/mobile-broadband-reach-expanding-globally/453
[2] http://blog.jana.com/2015/05/21/the-data-trap-affordable-smartphones-expensive-data

In the Progressive Times app, you're using web fonts to improve the look and feel of the app.

These fonts are downloaded from a third-party service and come in at around 30 KB. Web fonts do enhance the look and feel of a web page, but if users are trying to save data and money at the same time, web fonts seem unnecessary. There's no reason why your PWA can't cater to users regardless of their network connection.

Whether you're on a desktop or mobile device, enabling this feature is a relatively straightforward process. If you're on a mobile device, you can enable this under the Settings in your menu, as shown in figure 4.5.

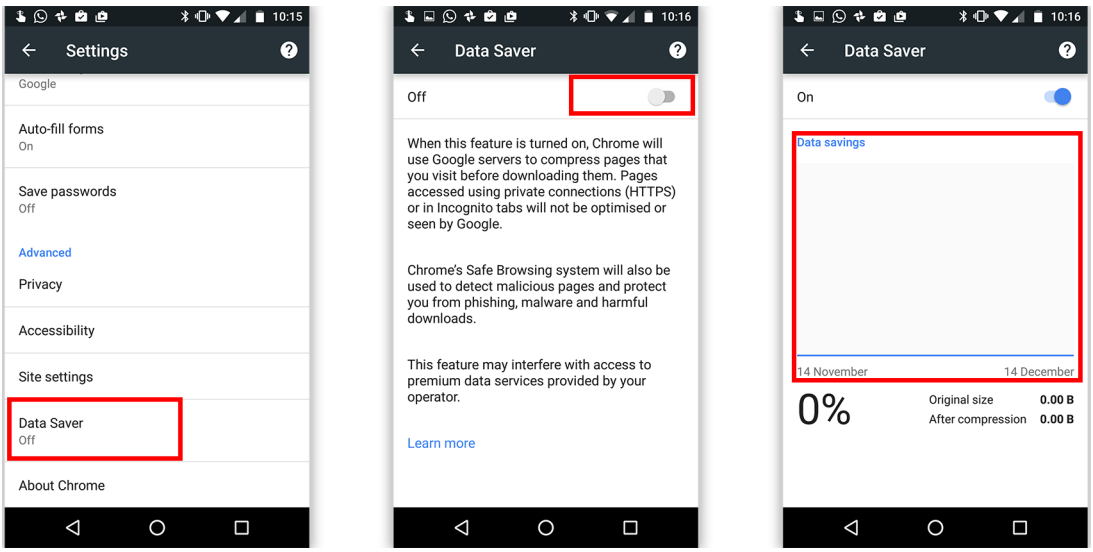

Figure 4.5 You can enable the Save-Data feature on your mobile device or on a phone. Note the highlighted areas in red.

Once the setting is enabled, each HTTP request to the server will include the `Save-Data` header. If you view this using your developer tools, it might look a little something like figure 4.5.

Once the Save-Data feature has been enabled, you can use a few different techniques to return data to the user. Because each HTTP request will go to the server, you could decide to serve different content based on the `Save-Data` header directly from server-side code. But with a few lines of JavaScript and using the power of Service Workers, you could easily intercept the HTTP requests and serve lighter content accordingly. If you're developing a front-end application that's API-driven and don't have access to the server, this is a perfect option.

Service Workers allow you to intercept outgoing HTTP requests, inspect them, and act on this information. Using the Fetch API, you can easily implement a solution to detect the `Save-Data` header and serve lighter content.

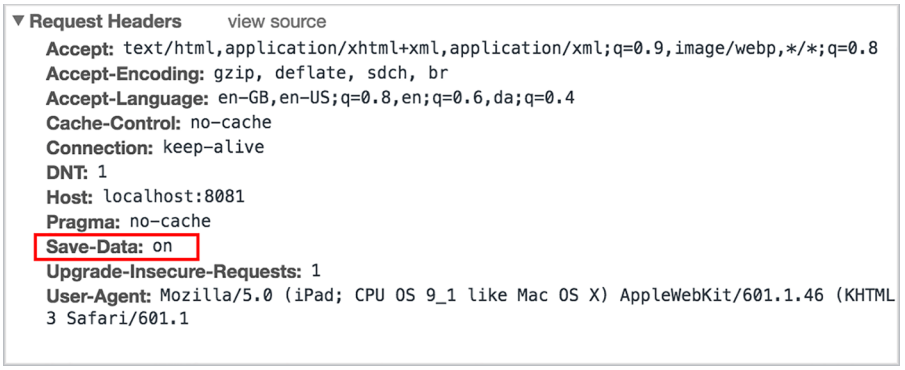

Figure 4.6 With the Save-Data feature enabled, each HTTP request will include this in the header.

You'll get started by creating a JavaScript file called service-worker.js and adding the code in listing 4.10 to it.

Listing 4.10 Service Worker code to check for the save-data HTTP header

```
"use strict";

this.addEventListener('fetch', function (event) {

  if(event.request.headers.get('save-data')){
    // We want to save data, so restrict icons and fonts
    if (event.request.url.includes('fonts.googleapis.com')) {
        // return nothing
        event.respondWith(new Response('', {status: 417, statusText: 'Ignore
     fonts to save data.' }));
    }
  }
});
```

Based on the examples we've looked at already, the code in listing 4.10 should look familiar. In the first few lines, you're adding an event listener to listen out for any `fetch` events that take place. For each request that takes place, you're inspecting the header and checking to see if the `Save-Data` header has been enabled. If it has been, you then check to see if the current HTTP request is for a web font from the domain fonts.googleapis.com. Because you're looking to save your users unnecessary data, you return a custom HTTP response with a 417-status code and your own custom status text. HTTP status codes provide users with specific information from the server;[3] in the case of a 417 status code, it's "The server cannot meet the requirements of the Expect request-header field."

[3] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Using this simple technique and a few lines of code, you were able to reduce the overall download size of the page and ensure that the user saved on any unnecessary data. You could extend this technique further and return images of a lower quality, or other larger downloads on your site.

You can see any of the code in this chapter in action on GitHub at http://bit.ly/chapter-pwa-4.

4.4 Summary

The Fetch API is a new browser API that aims to make your code cleaner and easier to read.

The fetch event allows you to intercept any outgoing HTTP requests to and from your browser. This functionality is extremely powerful and allows you to alter responses or even create your own custom HTTP responses without even hitting the server.

WebP images are 26% smaller in file size than PNG images and around 25–34% smaller in file size than JPEG images.

Using Service Workers, you can tap into the fetch event and intercept if the browser supports WebP images. Using this technique, you can serve smaller images to your users and speed up your page load times.

Some modern browsers have an opt-in to a feature that allows users to save data. If the feature is enabled, the browser adds a new header to each HTTP request. Using Service Workers you can tap into the fetch event and decide if you want to return a lighter version of your site.

WEB DEVELOPMENT

Progressive Web Apps

Dean Alan Hume

Offline websites that work. Near-instant load times. Smooth transitions between high/low/no bandwidth. Fantasy, right? Not with progressive web applications. PWAs use modern browser features like push notifications, smart caching, and Service Workers to manage data, minimize server usage, and allow for unstable connections, giving you better control and happier customers. Better still, all you need to build PWAs are JavaScript, HTML, and the easy-to-master techniques you'll find in this book.

Progressive Web Apps teaches you PWA design and the skills you need to build fast, reliable websites. There are lots of ways you can use PWA techniques, and this practical tutorial presents interesting, standalone examples so you can jump to the parts that interest you most. You'll discover how Web Service Workers vastly improve site loading, how to effectively use push notifications, and how to create sites with a no-compromise offline mode.

Inside, you'll find

- Improved caching with Service Workers
- Using manifest files and HTML markup
- Push notifications
- Offline-first web designs
- Techniques for data synchronization

Written for readers with experience developing websites using HTML, CSS, and JavaScript.

Dean Hume is a coder, author, and Google Developer Expert. He's passionate about web performance and user experience.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/progressive-web-apps

"Takes a practical, example-driven approach to learning how PWAs can help you build fast, engaging sites."
—From the Foreword by Addy Osmani, Google

"A pioneering work that will take your web app offline and onto the fast lane."
—Michal Paszkiewicz
Transport for London

"The very best resource for understanding and implementing progressive web applications."
—Evan Wallace
Berkley Insurance Australia

"Thorough, methodical coverage for novice users, with handy insights and many 'aha' moments for advanced users."
—Dev Paliwal, Synapse

ISBN-13: 978-1-61729-458-7
ISBN-10: 1-61729-458-6
53999
9 781617 294587

MANNING $39.99 / Can $52.99 [INCLUDING eBOOK]