

Progressive Web Apps

Dean Alan Hume

FOREWORD BY Addy Osmani



 MANNING



Progressive Web Apps

by Dean Alan Hume

Chapter 3

Copyright 2018 Manning Publications

brief contents

PART 1	DEFINING PROGRESSIVE WEB APPS	1
	1 ■ Understanding Progressive Web Apps	3
	2 ■ First steps to building a Progressive Web App	15
PART 2	FASTER WEB APPS	29
	3 ■ Caching	31
	4 ■ Intercepting network requests	51
PART 3	ENGAGING WEB APPS	65
	5 ■ Look and feel	67
	6 ■ Push notifications	81
PART 4	RESILIENT WEB APPLICATIONS	97
	7 ■ Offline browsing	99
	8 ■ Building more resilient applications	111
	9 ■ Keeping your data synchronized	120
PART 5	THE FUTURE OF PROGRESSIVE WEB APPS.....	133
	10 ■ Streaming data	135
	11 ■ Progressive Web App Troubleshooting	147
	12 ■ The future is looking good	157

Imagine you're on a train using your mobile phone to browse your favorite website. Every time the train enters an area with an unreliable network, the website takes ages to load—an all-too-familiar scene. This is where Service Worker caching comes to the rescue. Caching ensures that your website loads as efficiently as possible for repeat visitors.

This chapter starts off by looking at the basics of HTTP caching and what happens under the hood when your browser navigates to a URL. We'll also look closely at how you can use Service Worker caching to provide your users with a faster, more reliable website and how it works hand-in-hand with traditional HTTP caching. You'll learn how you can use Service Worker caching in a real-world application, including versioning and precaching resources. Finally, you'll discover one of my favorite Service Worker libraries: Workbox.

3.1 *The basics of HTTP caching*

Modern browsers are clever. They can interpret and understand a variety of HTTP requests and responses and are capable of storing and caching data until it's needed. I like to think of the browser's ability to cache information as the sell-by date on milk. In the same way you might keep milk in your fridge until it reaches the expiry date, browsers can cache information about a website for a set duration of time. After the data has expired, it will go and fetch the updated version. This ensures that web pages load faster and use less bandwidth.

Before we dive into Service Worker caching, let's take a step back and see how traditional HTTP caching works. Web developers have been able to use HTTP

caching since the introduction of HTTP/1.0 around the early 1990s.¹ HTTP caching allows the server to send the correct HTTP headers that will instruct the browser to cache the response for a certain amount of time.

A web server can take advantage of the browser's ability to cache data and use it to improve the repeat request load time. If the user visits the same page twice within one session, there's often no need to serve them a fresh version of the resources if the data hasn't changed. This way, a web server can use the `Expires` header to notify the web client that it can use the current copy of a resource until the specified "Expiry date." In turn, the browser can cache this resource and only check again for a new version when it reaches the expiry date. Figure 3.1 illustrates HTTP caching.

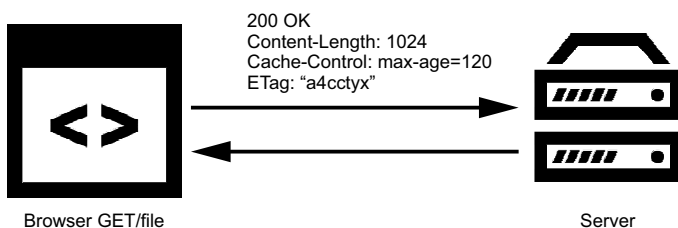


Figure 3.1 When a browser makes an HTTP request for a resource, the server sends an HTTP response containing useful information about the resource.

In figure 3.1, you can see that when a browser makes a request for a resource, the server returns the resource with a collection of HTTP headers. These headers contain useful information that the browser can then use to understand more about the resource. The HTTP response tells the browser what type of resource this is, how long to cache it for, whether it's compressed, and much more.

HTTP caching is a fantastic way to improve the performance of your website, but it isn't without flaws. Using HTTP caching means that you're relying on the server to tell you when to cache a resource and when it expires. If you have content that has dependencies, any updates can cause the expiry dates sent by the server to easily become out of sync and affect your site.

With great power comes great responsibility, and this is quite true for HTTP caching. When you make significant changes to HTML, you're likely to also change the CSS to reflect the new structure and update any JavaScript to accommodate changes to the style and content. If you've ever released changes to a website but haven't quite got your HTTP caching right, I'm sure you've seen the website break because of incorrectly cached resources.

¹ <https://hpbn.co/brief-history-of-http/>

Figure 3.2 shows what my own personal blog looks like when I have files cached incorrectly.

As you can imagine, this can be quite frustrating for both the developer and the user. In figure 3.2, you can see that the CSS styles for the page aren't loading. That's because incorrect caching caused a mismatch.

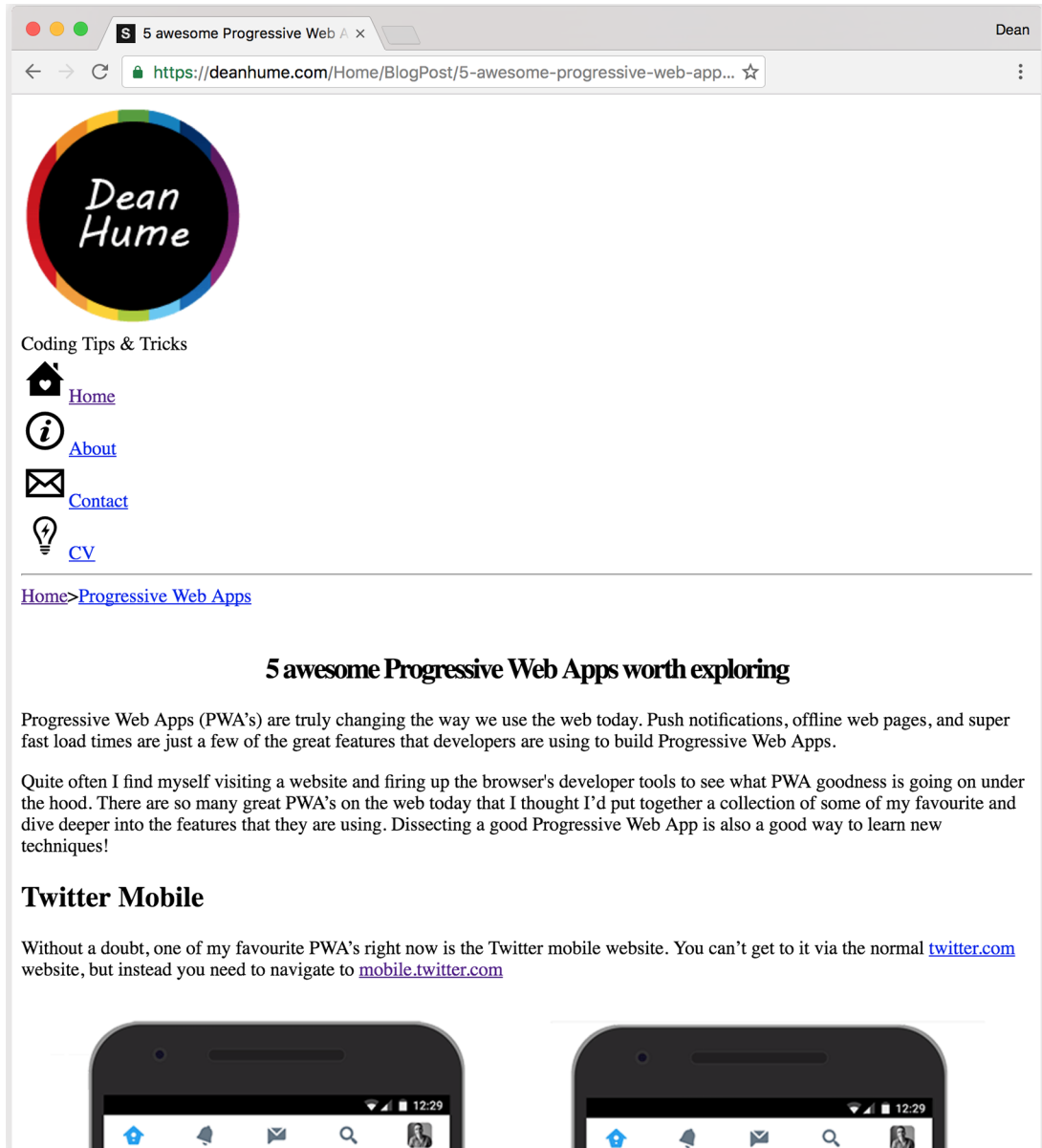


Figure 3.2 When cached files become out of sync, the look and feel of your website can be affected.

3.2 The basics of caching Service Worker caching

You may be wondering why you even need Service Worker caching if you have HTTP caching. How is Service Worker caching different? Well, instead of the server telling the browser how long to cache a resource, you are in complete control. Service Worker caching is extremely powerful because it gives you programmatic control over exactly how you cache your resources. As with all Progressive Web App (PWA) features, Service Worker caching is an enhancement to HTTP caching and works hand-in-hand with it.

The power of Service Workers lies in their ability to intercept HTTP requests. In this chapter, you'll use this ability to intercept HTTP requests and responses to provide users with a lightning fast response directly from cache.

3.2.1 Precaching during Service Worker installation

Using Service Workers, you can tap into any incoming HTTP requests and decide exactly how you want to respond. In your Service Worker, you can write logic to decide what resources you'd like to cache, what conditions need to be met, and how long to cache a resource for. You are in total control.

You may be familiar with figure 3.3—we looked at this briefly in earlier chapters of this book. When the user visits the website for the first time, the Service Worker begins downloading and installing itself. During the installation stage, you can tap into this event and prime the cache with all the critical assets for the web app.

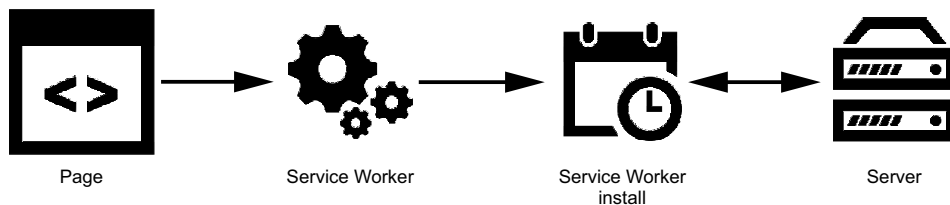


Figure 3.3 During the Service Worker installation step, you can fetch resources and prime the cache for the next visit.

Using this figure as an example, let's see a basic caching example in order to get a better understanding about how this work in reality. The next listing shows a simple HTML page that registers a Service Worker file.

Listing 3.1 Simple HTML page that registers a Service Worker file

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello Caching World!</title>
  
```

```

</head>
<body>
  <!-- Image -->
  
  <!-- JavaScript -->
  <script async src="/js/script.js"></script>
</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>

```

Reference to a "hello" image

Reference to a basic JavaScript file

Check to see if the current browser supports Service Workers.

If error during Service Worker registration, you can catch it and respond appropriately

In listing 3.1, you see a simple web page that references an image and a JavaScript file. The web page isn't anything fancy, but you'll use it to learn how to cache resources using Service Worker caching. The code checks whether your browser supports Service Workers; if so, it will try to register a file called `service-worker.js`, assuming you're playing along at home.

We have our basic page ready. Next you need to create the code that will cache your resources. The code in the following listing goes inside the Service Worker file `service-worker.js`.

Listing 3.2 Code in `service-worker.js`

```

var cacheName = 'helloWorld';

self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(cacheName)
      .then(cache => cache.addAll([
        '/js/script.js',
        '/images/hello.png'
      ]))
  );
});

```

Name of the cache

Tap into the Service Worker install event

Open a cache using the cache name we specified

Add the JavaScript and image into the cache

In chapter 1, we looked at the Service Worker lifecycle and the different stages it goes through before it becomes active. One of these stages is the install event, which happens when the browser installs and registers the Service Worker. This is the perfect time to add anything into cache that you think might be used at a later stage.

For example, if you know that a specific JavaScript file might be used throughout the site, you can decide to cache it during installation. That would mean that any other pages referencing this JavaScript file will easily be able to retrieve it from cache at a later stage.

The code in listing 3.2 taps into the `install` event and adds the JavaScript file and the hello image during this stage. It also references a variable called `cacheName`. This is a string value that I've set to name the cache. You can name each cache differently and you can even have multiple different copies of the cache because each new string makes it unique. This will come in handy later in the chapter when we look at versioning and cache busting.

In listing 3.2, you can see that once the cache has been opened, you can then begin to add resources into it. Next you call `cache.addAll()` and pass in your array of files. The `event.waitUntil()` method uses a JavaScript promise to know how long installation takes and whether it succeeded.

If all the files are successfully cached, the Service Worker will be installed. If any of the files fails to download, the `install` step will fail. This is important because it means you need to rely on all the assets being present on the server and you need to be careful with the list of files that you decide to cache in the `install` step. Defining a long list of files will increase the chances that one file may fail to cache, leading to your Service Worker not being installed.

Now that your cache is primed and ready to go, you're able to start reading assets from it. You need to add the code in the next listing to your Service Worker in order to start listening to the `fetch` event.

Listing 3.3 Code to add to Service Worker to start listening to the `fetch` event

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        if (response) {
          return response;
        }
        return fetch(event.request); #E
      })
  );
});
```

Annotations for Listing 3.3:

- Add an event listener to the fetch event**: Points to `self.addEventListener('fetch', ...)`
- Check whether incoming request URL matches anything that exists in the current cache**: Points to `caches.match(event.request)`
- If there's a response and it isn't undefined/null, then return it**: Points to `return response;`
- Else continue as normal and fetch the resource as intended**: Points to `return fetch(event.request);`

The code in listing 3.3 is the final piece of our Service Worker masterpiece. You start off by adding an event listener for the `fetch` event. Next, you check if the incoming URL matches anything that might exist in your current cache using the `caches.match()` function. If it does, return that cached resource, but if the resource doesn't exist in cache, continue as normal and fetch the requested resource.

If you open a browser that supports Service Workers and navigate to this newly created page, you should notice something similar to figure 3.4.



Figure 3.4 The sample code produces a basic web page with an image and a JavaScript file.

The requested resources should now be available in the Service Worker cache. When I refresh the page, the Service Worker will intercept the HTTP request and load the appropriate resources instantly from cache instead of making a network request to the server. In a few lines of code inside a Service Worker, you’ve made a site that loads directly from cache and responds instantly for repeat visits.

NOTE Service workers only work on secure origins such as HTTPS. But when you’re developing Service Workers on your local machine, you can use <http://localhost>. Service Workers have been built this way in order to ensure safety when deployed to live, and also for flexibility, to make it easier for developers to work on their local machine.

Some modern browsers can see what’s inside the Service Worker cache using the developer tools built into the browser. For example, if you open Google Chrome’s Developer Tools and navigate to the Application tab, you’ll see something similar to figure 3.5.

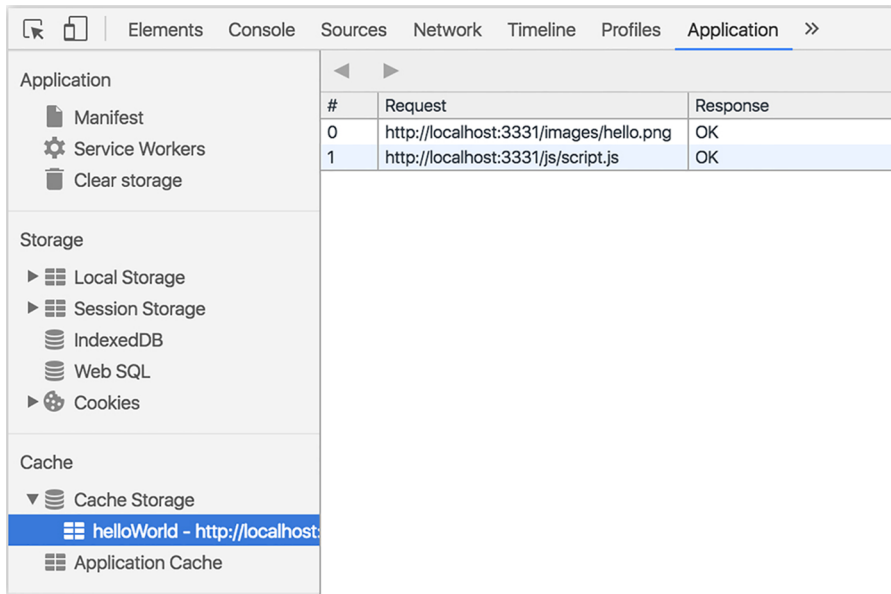


Figure 3.5 Google Chrome's Developer tools are helpful when you want to see what's stored in cache.

Figure 3.5 shows the cache entries for both the `scripts.js` and `hello.png` files stored in the cache named `helloWorld`. Now that the resources have been stored in cache, any future requests for those resources will be instantly fetched from cache.

3.2.2 Intercept and cache

Listing 3.2 showed how you can cache important resources during the installation of a Service Worker, which is known as precaching. This example works well when you know exactly the resources that you want to cache, but what about resources that might be dynamic or that you might not know about? For example, your website might be a sports news website that needs constant updating during a match; you won't know about those files during Service Worker installation.

Because Service Workers can intercept HTTP requests, this is the perfect opportunity to make the HTTP request and then store the response in cache. This means that instead you request the resource and then cache it immediately. That way, as the next HTTP request is made for the same resource, you can instantly fetch it out of the Service Worker cache, as shown in figure 3.6.

The next listing updates the code you previously used to include a new resource.

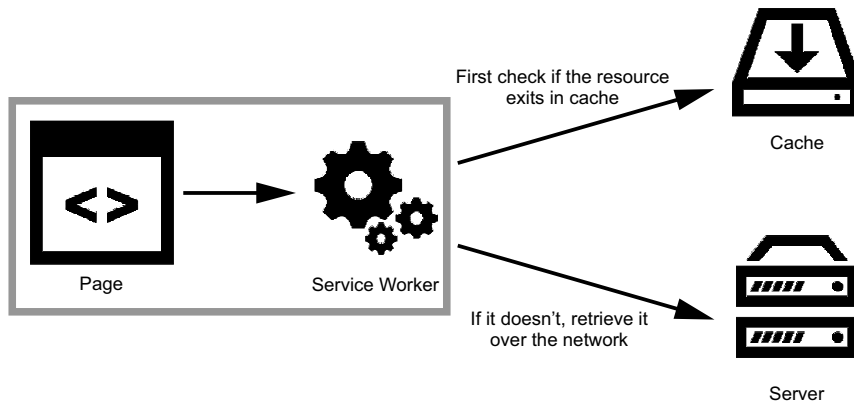


Figure 3.6 For any HTTP requests made, you can then check whether the resource already exists in cache, and if not we retrieve it via the network.

Listing 3.4 A basic web page to display Google fonts

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello Caching World!</title>
    <link href="https://fonts.googleapis.com/css?family=Lato"
      rel="stylesheet">
    <style>
      #body{ font-family: 'Lato', sans-serif; }
    </style>
  </head>
  <body>
    <h1>Hello Service Worker Cache!</h1>
    <!-- JavaScript -->
    <script async src="/js/script.js"></script>
    <script>
      if ('serviceWorker' in navigator) {
        navigator.serviceWorker.register('/service-
          worker.js').then(function(registration) {
            console.log('ServiceWorker registration successful with scope: ',
              registration.scope);
          }).catch(function(err) {
            console.log('ServiceWorker registration failed: ', err);
          });
      }
    </script>
  </body>
</html>

```

Add a reference to web fonts.

JavaScript file that provides functionality for the current page

First check whether the browser supports service workers.

If there is an error during the service worker registration, you can catch it and respond appropriately.

In listing 3.4, the code hasn't changed much compared to listing 3.1, except that you've added a reference to web fonts in the HEAD tag. Because this is an extra resource that may be likely to change, you can cache the resource once the HTTP

request has been made. You'll also notice that the JavaScript code used to register the Service Worker hasn't changed. In fact, with a few exceptions, this code is a pretty standard way of registering your Service Worker. You'll be using this boilerplate code to register a Service Worker repeatedly throughout the book.

Now that the page is complete, you're ready to start adding some code to the Service Worker file. The next listing shows the code you'll be using.

Listing 3.5 Adding code to the Service Worker file

```

var cacheName = 'helloWorld';

self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        if (response) {
          return response;
        }

        var requestToCache = event.request.clone();

        return fetch(requestToCache).then(
          function(response) {
            if (!response || response.status !== 200) {
              return response;
            }

            var responseToCache = response.clone();

            caches.open(cacheName)
              .then(function(cache) {
                cache.put(requestToCache, responseToCache);
              });

            return response;
          }
        );
      })
  );
});

```

Name of the cache → `var cacheName = 'helloWorld';`

← **Add an event listener for the fetch event to intercept requests.** `self.addEventListener('fetch', function(event) {`

← **Does the current request match anything you might have in cache?** `caches.match(event.request)`

← **If it does, return it at this point and continue no further.** `return response;`

← **Clone the request—a request is a stream and can only be consumed once.** `var requestToCache = event.request.clone();`

← **If request fails or server responds with an error code, return that error immediately** `if (!response || response.status !== 200) {`

← **Try to make the original HTTP request as intended.** `return fetch(requestToCache).then(`

← **Again clone the response because you need to add it into cache and because it's used for the final return response.** `var responseToCache = response.clone();`

← **Open helloWorld cache.** `caches.open(cacheName)`

← **Add response into cache.** `cache.put(requestToCache, responseToCache);`

← `return response;`

← `);`

← `});`

Listing 3.5 seems like a lot of code. Let's break it down and explain each section. The code starts off by tapping into the fetch event by adding an event listener. The first thing you want to do is check whether the requested resource already exists in cache. If it does, you can return it at this point and go no further.

But if the requested resource doesn't already exist in the cache, you make the request as originally intended. Before the code goes any further you need to clone the request because a request is a stream that can only be consumed once. Because you're consuming this once by cache and then again when you make the HTTP request for it, you need to clone the response at this point. You then need to check

the HTTP response and ensure that the server returned a successful response and that nothing went wrong. You don't want to cache an errored result.

If the response was successful, you're clone the response again. You're probably wondering why you need to clone the response again, but remember that a response is a *stream that can only be consumed once*. Because you want the browser to consume the response as well as the cache consuming the response, you need to clone it so you have two streams.

Finally, the code then uses this response and adds it to the cache so you can use it again next time. If the user then refreshes the page or visits another page on the site that requires these resources, it will be fetched from cache instantly instead of via the network.

In figure 3.7, notice that there are new entries in the cache for the three resources on the page. In the coding example covered earlier, you were able to dynamically add a resource into cache as each successful HTTP response was returned. This technique is perfect for when you might want to cache resources but aren't quite sure how often they may change or exactly where they might be coming from.

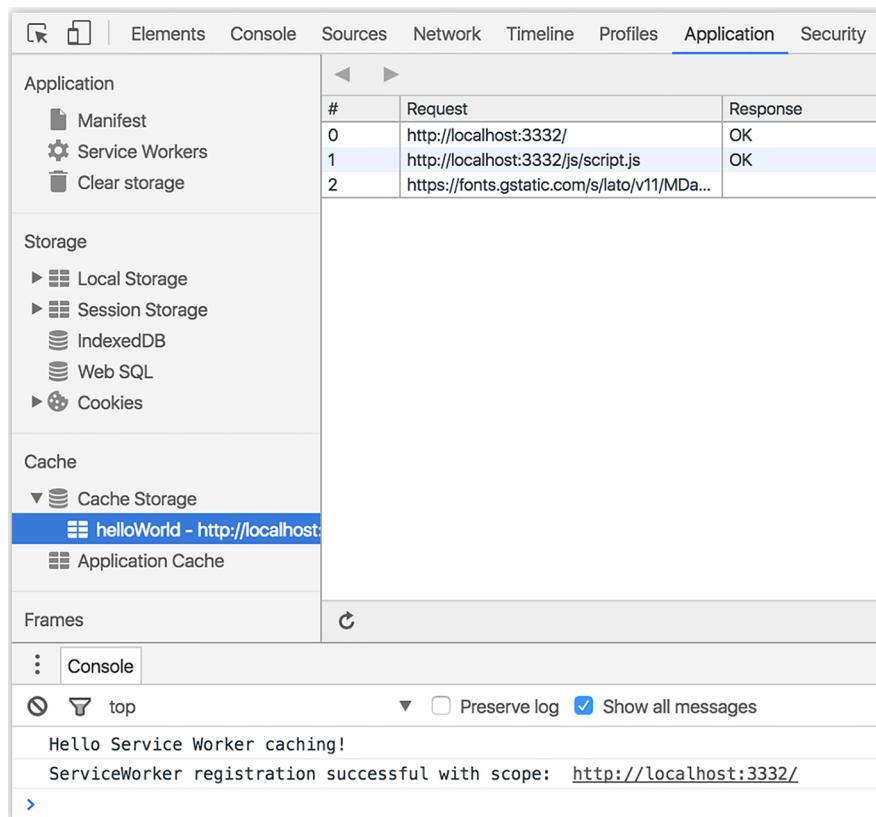


Figure 3.7 Using Google Chrome's Developer tools you see that the web fonts were retrieved from the network and then added to cache in order to ensure faster repeat requests.

Service Workers give a developer total control over the code and allow you to easily build custom caching solutions that fit your needs. In fact, the two caching techniques covered earlier can be combined to produce even faster load times. The control is in your hands.

For example, let's say you were building a new web application that used the App Shell Architecture. You might want to precache the shell using the code in listing 3.2. Then any further HTTP requests that are made can be cached using the intercept and cache technique. Or perhaps you want to cache parts of an existing site that you know don't change often. By intercepting and caching these resources, you'll provide your users with improved performance in a few lines of code. Depending on your situation, Service Worker caching can be adapted to suit your needs and make an instant difference to the experience your users receive.

3.2.3 *Putting it all together*

The code examples we've run through so far have been helpful, but it isn't easy to imagine them on their own. In chapter 1, we talked about the many different ways that you could use Service Workers to build amazing web apps. One of those concepts was a newspaper web app, which we can use to play with everything you've learned about Service Worker caching in a real-world scenario. I'm going to call our sample application *Progressive Times*. The web app is a news site where people will regularly visit and read multiple pages, so it makes sense to cache future pages ahead of time so they load instantly. You could even save the content so that a user could browse while offline.

The sample web application contains a collection of funny news facts from around the world (figure 3.8). Believe it or not, all the stories in this news site are true and came from credible news sources. The web app contains most of the basic elements of a website that you can imagine, such as CSS, JavaScript, and images. To keep the sample code basic, I've also used a flat JSON file for each article; in real life, this would point to a back-end endpoint to retrieve the data in a similar format. On its own, this web app is not that impressive, but when you start to use the power of Service Workers, you can take it to the next level.

The web application uses the App Shell Architecture to dynamically fetch the contents of each article and inject the data onto the page, as shown in figure 3.8.

Using the App Shell Architecture also means you can use precaching to ensure that the web app loads instantly for repeat visits. You can also assume that a visitor will tap a link and follow through to the full contents of a news article. If you cached this when the Service Worker was installed, it would mean that the next page would load significantly faster for them.

Let's put everything you learned this far in the chapter together and see how to add a Service Worker to the *Progressive Times* app that will precache important resources and cache any other requests as they are made, as shown in the next listing.

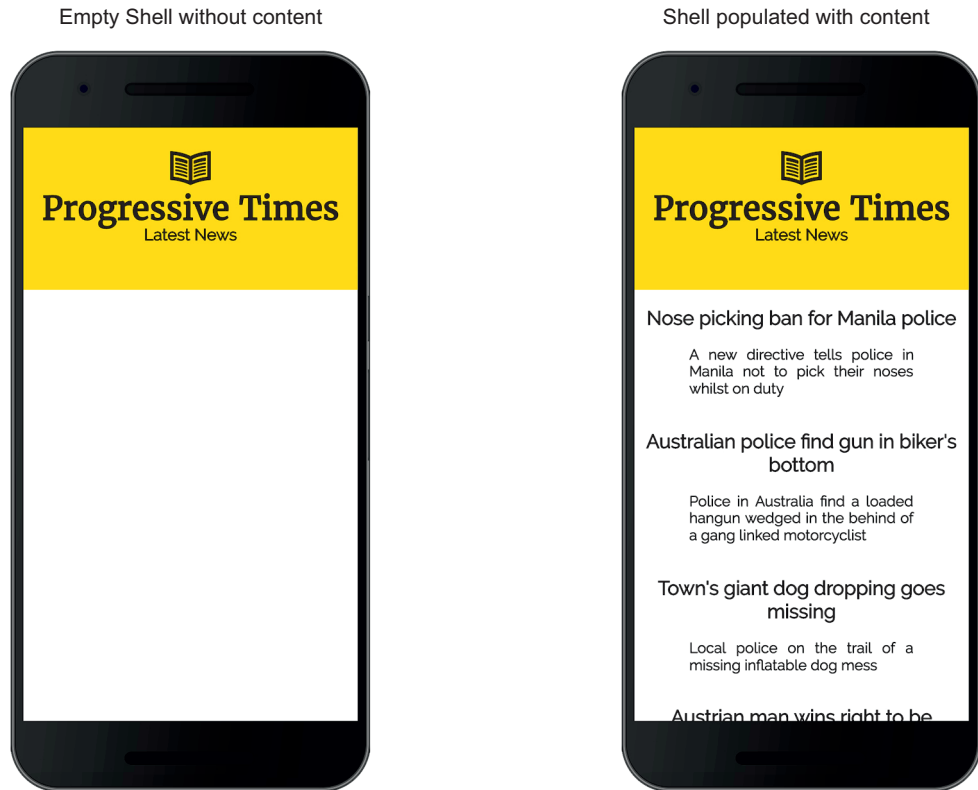


Figure 3.8 The Progressive Times sample application uses the App Shell Architecture.

Listing 3.6 Service Worker code to precache and Cache resources during runtime

```
var cacheName = 'latestNews-v1';

// Cache our known resources during install
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(cacheName)
      .then(cache => cache.addAll([
        './js/main.js',
        './js/article.js',
        './images/newspaper.svg',
        './css/site.css',
        './data/latest.json',
        './data/data-1.json',
        './article.html',
        './index.html'
      ]))
  );
});
```

Open the cache and store an array of resources to cache during install time.


```
// Cache any new resources as they are fetched
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request, { ignoreSearch: true })
    .then(function(response) {
      if (response) {
        return response;
      }
      var requestToCache = event.request.clone();
      return fetch(requestToCache).then(
        function(response) {
          if(!response || response.status !== 200) {
            return response;
          }
          var responseToCache = response.clone();
          caches.open(cacheName)
            .then(function(cache) {
              cache.put(requestToCache, responseToCache);
            });
          return response;
        }
      );
    })
  );
});
```

Listen for the fetch event.

Ignore any querystring parameters so you don't get any cache misses.

If you found a successful match, return it at this point and go no further.

If you didn't find anything in cache, make the request

Store it in cache so we won't need to make that request again

The code in listing 3.6 is a combination of precaching during install time and storing in cache as you fetch a resource. The web app is using an App Shell Architecture, which means you can take advantage of Service Worker caching to request only the data needed to populate the page. You've already successfully stored the assets for the shell, so all that's left is the dynamic news content from the server.

If you'd like to see this web page in action, it's available on GitHub and can be easily accessed at bit.ly/chapter-pwa-3. In fact, I've added all the code samples that you'll use throughout this book to that GitHub repo.

Each chapter has a readme file that explains what you need to do to start building and experimenting with the sample code in each chapter. About 90% of the chapters are front-end code, so all you need to do is fire up your localhost and get started. It's also worth noting that you need to be running the code on <http://localhost> environment and not on `file://` environment.

3.3 *Performance comparison: before and after caching*

At this point, I hope I've managed to convince you how great Service Worker caching is. Not yet!? Okay, well, hopefully the performance improvements you'll gain when using caching will change your mind.

Using our Progressive Times sample application, we can compare the difference with and without Service Worker caching. One of my favorite ways to test the real-world performance of a website is to use a tool called WebPagetest.org, shown in figure 3.9.

The screenshot shows the WebPagetest website in a browser window. The page has a dark blue header with the WebPagetest logo and navigation links: HOME, TEST RESULT, TEST HISTORY, FORUMS, DOCUMENTATION, and ABOUT. A banner for 'Fremont Business Internet Services' is visible. The main content area is titled 'Test a website's performance' and features three tabs: Analytical Review, Visual Comparison, and Traceroute. The 'Analytical Review' tab is active, showing a form to 'Enter a Website URL'. Below this, there are dropdown menus for 'Test Location' (set to Singapore - EC2) and 'Browser' (set to Chrome). An 'Advanced Settings' section is expanded, showing options for 'Connection' (Mobile 3G - Fast), 'Number of Tests to Run' (1), 'Repeat View' (First View and Repeat View), 'Capture Video' (checked), 'Keep Test Private' (unchecked), and a 'Label' field. A yellow 'START TEST' button is on the right. The Akamai logo is also present.

Figure 3.9 [WebPagetest.org](https://www.webpagetest.org) is a free tool you can use to test your websites using real devices from around the world.

[WebPagetest.org](https://www.webpagetest.org) is a great tool. Enter the URL of your website, and it allows you to profile your website from any location around the world using a real-world device and a wide range of browsers. The tests run on real devices and provide you with a helpful breakdown and profile of the performance of your website. Best of all, it's open source and completely free to use.

If I run our sample application through [WebPagetest.org](https://www.webpagetest.org), it produces something similar to figure 3.7.

To test how our sample web application performed on a real-world device, I used WebPagetest with a 2G mobile connection from an endpoint in Singapore. If you've ever tried to access a website over a slow network connection, you'll know how annoying it can be while you wait for the site to finish loading. As web developers, it's important that we test our websites as our users would use them, and that includes

using slower mobile connection speeds and low-end devices, too. Once WebpPagetest completed profiling the web app, it produced the results shown in figure 3.10.

	Load Time	First Byte	Start Render	<u>Speed Index</u>	Document Complete		
					Time	Requests	Bytes In
First View	12.258s	3.752s	7.481s	8030	12.258s	13	91 KB
Repeat View	0.578s	3.498s	0.476s	492	0.578s	0	0 KB

Figure 3.10 [WebPagetest.org](https://webpagetest.org) produces useful information about the performance of your web application by using a real device.

In the first view, the page took around 12 seconds to load. This isn't ideal, but not unexpected over a slow 2G connection. But if you look at the repeat view, the site loaded in less than 0.5 seconds and made zero HTTP requests to the server. The sample application used the App Shell Architecture, and if you remember the layout, you'll know that any future requests will be served as quickly because the resources needed have already been cached. If used correctly, Service Worker caching significantly improves the overall speed of your application and enhances the browsing experience regardless of the device or connection used.

3.4 *Diving deeper into Service Worker caching*

In this chapter, we've started to look at how Service Worker caching can be used to improve the performance of your web application. As we progress through the rest of this chapter, we'll look closely at how you can version your files in order to ensure that there are no cache mismatches, as well as to avoid some of the common gotchas you might encounter while using Service Worker caching.

3.4.1 *Versioning your files*

There will be a point in time where your Service Worker cache will need updating. If you make changes to your web application, be sure users receive the newer version of files instead of older versions. As you can imagine, serving older files by mistake would cause havoc on a site.

The great thing about Service Workers is that each time you make any changes to the Service Worker file itself, it automatically triggers the Service Worker update flow. In chapter 1, we looked at the Service Worker lifecycle. Remember that when a user navigates to your site, the browser tries to re-download the Service Worker in the background. If there's even a byte's difference in the Service Worker file compared to what it currently has, it considers it new.

This useful functionality gives you the perfect opportunity to update your cache with new files. You can use two approaches when updating the cache. First, you can update the name of the cache that you use to store against. Referring back to the code

in listing 3.2, you can see the `cacheName` variable with a value `'helloWorld'`. If you updated this value to `'helloWorld-2'`, that would automatically create a new cache and start serving your files from that cache. The original cache would be orphaned and no longer used.

The second option, which I personally feel is the more bulletproof one, is to version your files. This technique is known as *cache busting* and has been around for many years. When a static file gets cached, it can be stored for long periods of time before it ends up expiring. That can be an annoyance in the event that you make an update to a site, but because the cached version of the file is stored in your visitors' browsers, they may be unable to see the changes made. Cache busting solves this problem by using a unique file version identifier to tell the browser that a new version of the file is available.

For example, if you were to add a reference to a JavaScript file in the HTML, you might want to append a hashed string onto the end of the filename, similar to this:

```
<script type="text/javascript" src="/js/main-xtvbas65.js"></script>
```

The idea behind cache busting is that you create a completely new filename each time you make changes to the file in order to ensure that the browser fetches the freshest content possible. Imagine the following scenario in our newspaper web app. Let's say you have a file called `main.js` and store it in cache exactly as it is. Depending on how your Service Worker is set up, it will retrieve this version of the file from cache every time. If you make a change to the `main.js` file with new code, the Service Worker will still intercept and return the older cached version even though you want to serve the newer version of the file. But if you rename the file to, say, `main.v2.js` and update your code to point to this new version, you can ensure that the browser will get the fresh version every time. That way, your newspaper will always return the freshest results to your users.

There are many different approaches to implementing this solution, and all of them may depend on your coding environment. Some developers prefer to generate these hashed filenames during build time, and others may do this using code and generate the filenames on the fly. Whichever approach you use, this technique is a tried-and-tested way to ensure that you always serve the correct files.

3.4.2 Dealing with extra query parameters

When a Service Worker checks for a cached response, it uses a request URL as the key. By default, the request URL must exactly match the URL used to store the cached response, including any query parameters in the search portion of the URL.

If you make any HTTP requests for files appended with query strings that sometimes change, this might end up causing you a few issues. For example, if you make a request for a URL that previously matched, you may find that it misses because the query string differs slightly. To ignore query strings when you check the cache, use the `ignoreSearch` attribute and set the value to `true`, as shown in the following listing.

Listing 3.7 Service Worker code to ignore query string parameters

```
self.addEventListener('fetch', function(event) {  
  event.respondWith(  
    caches.match(event.request, {  
      ignoreSearch: true  
    }).then(function(response) {  
      return response || fetch(event.request);  
    })  
  );  
});
```

The code in listing 3.7 uses the `ignoreSearch` option to ignore the search portion of the URL in both the request argument and cached requests. You can extend this further by using other ignore options such as `ignoreMethod` and `ignoreVary`. For example, the `ignoreMethod` value will ignore the method of the request argument, so a POST request can match a GET entry in the cache. The `ignoreVary` value will ignore the vary header in cached responses.

3.4.3 How much memory do you need?

Whenever I talk to developers about Service Worker caching, the questions that regularly arise involve memory and storage space. How much space does the Service Worker use to cache? How will this memory usage affect my device?

The honest answer is that it depends on your device and storage conditions. Like all browser storage, the browser is free to throw it away if the device comes under storage pressure. That's not necessarily a problem because the data can then be fetched again from the network as needed. In chapter 7, we'll look at another type of storage called *persistent storage* that can be used to store cached data on a more permanent basis.

Right now, older browsers are still able to store cached responses in their memory, and the space they use isn't different from the space that the Service Worker uses to cache resources. The only difference is that Service Worker caching puts you in the driving seat and allows you to programmatically create, update, and delete cached entries, allowing you to access resources without a network connection.

3.4.4 Taking caching to the next level: Workbox

If you find yourself regularly writing code in your Service Workers that caches resources, you might find Workbox (<https://workboxjs.org/>) helpful. Written by the team at Google, it's a library of helpers to get you started creating your own Service Workers in no time, with built-in handlers to cover the most common network strategies. In a few lines of code, you can decide whether you want to serve specific resources solely from cache, serve resources from cache and then fall back, or perhaps only return resources from the network and never cache. This library gives you total control over your caching strategy. See figure 3.11.

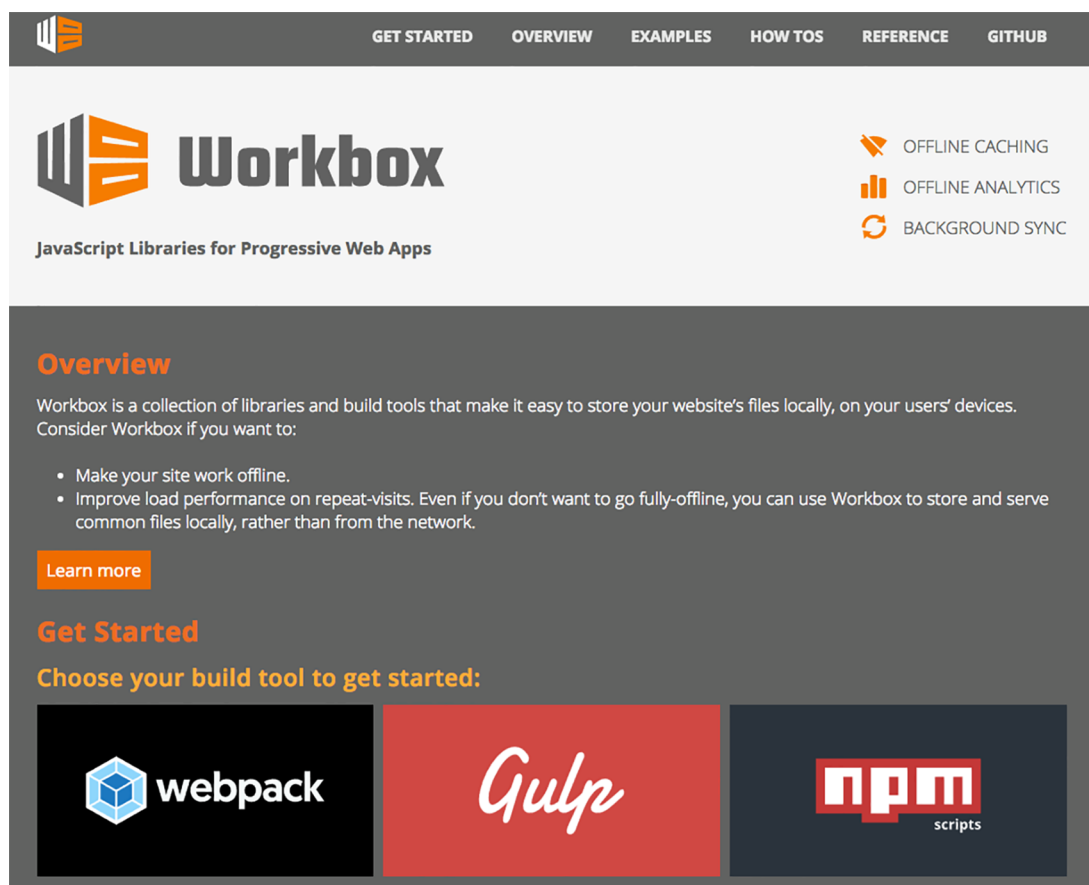


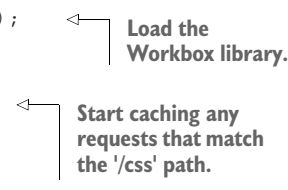
Figure 3.11 Workbox provides a library of helpers for use in creating your own Service Workers.

Workbox provides you with a quick and easy way to reuse common network caching strategies instead of rewriting them again and again. For example, say you wanted to ensure that you always retrieve your CSS files from the cache but only fall back to the network if a resource wasn't available. Using Workbox, you register your Service Worker the same way you have throughout this chapter. Then you import the library into your Service Worker file and start defining routes that you want to cache.

In listing 3.8, the code starts off by importing the Workbox library using the `importScripts` function. Service Workers have access to a global function, called `importScripts()`, which lets them import scripts in the same domain into their scope. This is a handy way to load another script into an existing script. It keeps the code clean and means you only load the file when it's needed.

Listing 3.8 Using Workbox

```
importScripts('workbox-sw.prod.v1.1.0.js');  
const workboxSW = new self.WorkboxSW();  
  
workboxSW.router.registerRoute(  
  'https://test.org/css/(.*)',  
  workboxSW.strategies.cacheFirst()  
);
```



Once the script has been imported, you can start defining routes you want to cache. In listing 3.8, you’re defining a route for anything that matches the ‘/css/’ path and always serving it with a cache first approach. This means that the resources will always be served from cache and will fall back to the network if they don’t exist. Workbox also provides a number of other built-in caching strategies,² such as cache only, network only, network first, cache first, or fastest, which tries to find the fastest response from either cache or the network. Each of these strategies can be applied to different scenarios, and you can even mix and match them with different routes to achieve the best effect.

Workbox also provides you with functionality to precache resources. In the same way that you precached resources during installation of the Service Worker in listing 3.2, you can achieve this with a few lines of code using Workbox.

Whenever I approach a new project, without a doubt my favorite library to use is Workbox. It simplifies your code and provides you with tried-and-tested caching strategies that you can implement in a few lines of code. In fact, the Twitter PWA we dissected in chapter 2 uses Workbox to make the code simpler to understand and relies on these tried-and-tested caching approaches.

3.5 Summary

HTTP caching is a fantastic way to improve the performance of your website, but it isn’t without flaws.

Service Worker caching is extremely powerful because it gives you programmatic control over exactly how you cache your resources. When used hand-in-hand with HTTP caching, you get the best of both worlds.

Used correctly, Service Worker caching is a massive performance enhancement and bandwidth saver.

You can use a number of different approaches to cache resources, and each of them can be adapted to suit the needs of your users.

WebPagetest is a great tool for testing the performance of your web apps using real-world devices.

Workbox is a handy library that provides you with tried-and-tested caching techniques.

² www.recode.net/2016/6/8/11883518/app-boom-over-snapchat-uber

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.”

—Michał 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



9 781617 129458



5 3 9 9 9



\$39.99 / Can \$52.99 [INCLUDING eBook]