№8. LET'S TAKE THIS OFFLINE

show table of contents

❧

DIVING IN

hat is an offline web application? At first glance, it sounds like a contradiction in terms. Web pages are things you download and render. Downloading implies a network connection. How can you download when you're offline? Of course, you can't. But you *can* download when you're online. And that's how HTML5 offline applications work.

At its simplest, an offline web application is a list of URLs — HTML, CSS, JavaScript, images, or any other kind of resource. The home page of the offline web application points to this list, called a manifest file, which is just a text file located elsewhere on the web server. A web browser that implements HTML5 offline applications will read the list of URLs from the manifest file, download the resources, cache them locally, and automatically keep the local copies up to date as they change. When the time comes

that you try to access the web application without a network connection, your web browser will automatically switch over to the local copies instead.

From there, most of the work is up to you, the web developer. There's a flag in the DOM that will tell you whether you're online or offline. There are events that fire when your offline status changes (one minute you're offline and the next minute you're online, or vice-versa). But that's pretty much it. If your application creates data or saves state, it's up to you to store that data locally while you're offline and synchronize it with the remote server once you're back online. In other words, HTML5 can take your web application offline. What you do once you're there is up to you.

OFFLINE SUPPORT

IE	FIREFOX	SAFARI	CHROME	OPERA	IPHONE	ANDROID
·	3.5+	4.0+	5.0+	10.6+	2.1+	2.0+

THE CACHE MANIFEST

An offline web application revolves around a cache manifest file. What's a manifest file? It's a list of all of the resources that your web application might need to access while it's disconnected from the network. In order to bootstrap the process of downloading and caching these resources, you need to point to the manifest file, using a `manifest` attribute on your `<html>` element.

```
<!DOCTYPE html>
<html manifest="/cache.manifest">
<body>
...
</body>
```

```
</html>
```

Your cache manifest file can be located anywhere on your web server, but it must be served with the content type `text/cache-manifest`. If you are running an Apache-based web server, you can probably just put an `AddType` directive in the `.htaccess` file at the root of your web directory:

```
AddType text/cache-manifest .manifest
```

Then make sure that the name of your cache manifest file ends with `.manifest`. If you use a different web server or a different configuration of Apache, consult your server's documentation on controlling the `Content-Type` header.

ASK PROFESSOR MARKUP

Q: My web application spans more than one page. Do I need a `manifest` attribute in each page, or can I just put it in the home page?

A: Every page of your web application needs a `manifest` attribute that points to the cache manifest for the entire application.

OK, so every one of your HTML pages points to your cache manifest file, and your cache manifest file is being served with the proper `Content-Type` header. But what goes *in* the manifest file? This is where things get interesting.

The first line of every cache manifest file is this:

```
CACHE MANIFEST
```

After that, all manifest files are divided into three parts: the “explicit” section, the “fallback” section, and the “online whitelist” section. Each section has a header, on its own line. If the manifest file doesn’t have any section headers, all the listed resources are implicitly in the “explicit” section. Try not to dwell on the terminology, lest your head explode.

Here is a valid manifest file. It lists three resources: a CSS file, a JavaScript file, and a JPEG image.

```
CACHE MANIFEST
/clock.css
/clock.js
/clock-face.jpg
```

This cache manifest file has no section headers, so all the listed resources are in the “explicit” section by default. Resources in the “explicit” section will get downloaded and cached locally, and will be used in place of their online counterparts whenever you are disconnected from the network. Thus, upon loading this cache manifest file, your browser would download `clock.css`, `clock.js`, and `clock-face.jpg` from the root directory of your web server. Then you could unplug your network cable and refresh the page, and all of those resources would be available offline.

ASK PROFESSOR MARKUP

Q: Do I need to list my HTML pages in my cache manifest?
A: Yes and no. If your entire web application is contained in a single page,

just make sure that page points to the cache manifest using the `manifest` attribute. When you navigate to an HTML page with a `manifest` attribute, the page itself is assumed to be part of the web application, so you don't need to list it in the manifest file itself. However, if your web application spans multiple pages, you should list all of the HTML pages in the manifest file, otherwise the browser would not know that there are other HTML pages that need to be downloaded and cached.

NETWORK SECTIONS

Here is a slightly more complicated example. Suppose you want your clock application to track visitors, using a tracking.cgi script that is loaded dynamically from an `<img src>` attribute. Caching this resource would defeat the purpose of tracking, so this resource should never be cached and never be available offline. Here is how you do that:

```
CACHE MANIFEST
NETWORK:
/tracking.cgi
CACHE:
/clock.css
/clock.js
/clock-face.jpg
```

This cache manifest file includes section headers. The line marked `NETWORK:` is the beginning of the "online whitelist" section. Resources in this section are never cached and are not available offline. (Attempting to load them while offline will result in an error.) The line marked `CACHE:` is the beginning of the "explicit" section. The rest of the cache manifest file is the same as the previous example. Each of the three resources listed will be cached and available offline.

FALLBACK SECTIONS

There is one more type of section in a cache manifest file: a fallback section. In a fallback section, you can define substitutions for online resources that, for whatever reason, can't be cached or weren't cached successfully. The HTML5 specification offers this clever example of using a fallback section:

```
CACHE MANIFEST
FALLBACK:
/ /offline.html
NETWORK:
*
```

What does this do? First, consider a site that contains millions of pages, like Wikipedia. You couldn't possibly download the entire site, nor would you want to. But suppose you could make *part* of it available offline. But how would you decide which pages to cache? How about this: every page you ever look at on a hypothetical offline-enabled Wikipedia would be downloaded and cached. That would include every encyclopedia entry that you ever visited, every talk page (where you can have makeshift discussions about a particular encyclopedia entry), and every edit page (which you can actually make changes to the particular entry).

That's what this cache manifest does. Suppose every HTML page (entry, talk page, edit page, history page) on Wikipedia pointed to this cache manifest file. When you visit any page that points to a cache manifest, your browser says "hey, this page is part of an

offline web application, is it one I know about?" If your browser hasn't ever downloaded this particular cache manifest file, it will set up a new offline "appcache" (short for "application cache"), download all the resources listed in the cache manifest, and then add the current page to the appcache. If your browser does know about this cache manifest, it will simply add the current page to the existing appcache. Either way, the page you just visited ends up in the appcache. This is important. It means that you can have an offline web application that "lazily" adds pages as you visit them. You don't need to list every single one of your HTML pages in your cache manifest.

Now look at the fallback section. The fallback section in this cache manifest only has a single line. The first part of the line (before the space) is not a URL. It's really a URL pattern. The single character (`/`) will match any page on your site, not just the home page. When you try to visit a page while you're offline, your browser will look for it in the appcache. If your browser finds the page in the appcache (because you visited it while online, and the page was implicitly added to the appcache at that time), then your browser will display the cached copy of the page. If your browser doesn't find the page in the appcache, instead of displaying an error message, it will display the page `/offline.html`, as specified in the second half of that line in the fallback section.

Finally, let's examine the network section. The network section in this cache manifest also has just a single line, a line that contains just a single character (`*`). This character has special meaning in a network section. It's called the "online whitelist wildcard flag." That's a fancy way of saying that anything that isn't in the appcache can still be downloaded from the original web address, as long as you have an internet connection. This is important for an "open-ended" offline web application. It means that, while you're browsing this hypothetical offline-enabled Wikipedia *online*, your browser will fetch images and videos and other embedded resources normally, even if they are on a different domain. (This is common in large websites, even if they aren't part of an offline web application. HTML pages are generated and served locally, while images and videos are served from a CDN on another domain.) Without this wildcard flag, our hypothetical offline-enabled Wikipedia would behave strangely when you were online — specifically, it wouldn't load any externally-hosted images or videos!

Is this example complete? No. Wikipedia is more than HTML files. It uses common CSS, JavaScript, and images on each page. Each of these resources would need to be listed explicitly in the `CACHE:` section of the manifest file, in order for pages to display and behave properly offline. But the point of the fallback section is that you can have an "open-ended" offline web application that extends beyond the resources you've listed explicitly in the manifest file.

THE FLOW OF EVENTS

So far, I've talked about offline web applications, the cache manifest, and the offline application cache ("appcache") in vague, semi-magical terms. Things are downloaded, browsers make decisions, and everything Just Works. You know better than that, right? I mean, this is web development we're talking about. Nothing ever Just Works.

First, let's talk about the flow of events. Specifically, DOM events. When your browser visits a page that points to a cache manifest, it fires off a series of events on the `window.applicationCache` object. I know this looks complicated, but trust me, this is the simplest version I could come up with that didn't leave out important information.

1. As soon as it notices a `manifest` attribute on the `<html>` element, your browser fires a `checking` event. (All the events listed here are fired on the `window.applicationCache` object.) The `checking` event is always fired, regardless of whether you have previously visited this page or any other page that points to the same cache manifest.
2. If your browser has never seen this cache manifest before...
    - It will fire a `downloading` event, then start to download the resources listed

in the cache manifest.
    - While it's downloading, your browser will periodically fire `progress` events, which contain information on how many files have been downloaded already and how many files are still queued to be downloaded.
    - After all resources listed in the cache manifest have been downloaded successfully, the browser fires one final event, `cached`. This is your signal that the offline web application is fully cached and ready to be used offline. That's it; you're done.

3. On the other hand, if you have previously visited this page or any other page that points to the same cache manifest, then your browser already knows about this cache manifest. It may already have some resources in the appcache. It may have the entire working offline web application in the appcache. So now the question is, has the cache manifest changed since the last time your browser checked it?

    - If the answer is no, the cache manifest has not changed, your browser will immediately fire a `noupdate` event. That's it; you're done.
    - If the answer is yes, the cache manifest *has* changed, your browser will fire a `downloading` event and start re-downloading every single resource listed in the cache manifest.
    - While it's downloading, your browser will periodically fire `progress` events, which contain information on how many files have been downloaded already and how many files are still queued to be downloaded.
    - After all resources listed in the cache manifest have been re-downloaded successfully, the browser fires one final event, `updateready`. This is your signal that the new version of your offline web application is fully cached and ready to be used offline. *The new version is not yet in use.* To "hot-swap" to the new version without forcing the user to reload the page, you can manually call the `window.applicationCache.swapCache()` function.

If, at any point in this process, something goes horribly wrong, your browser will fire an `error` event and stop. Here is a hopelessly abbreviated list of things that could go wrong:

- The cache manifest returned an HTTP error 404 (Page Not Found) or 410 (Permanently Gone).
- The cache manifest was found and hadn't changed, but the HTML page that pointed to the manifest failed to download properly.
- The cache manifest changed while the update was being run.
- The cache manifest was found and had changed, but the browser failed to download one of the resources listed in the cache manifest.

THE FINE ART OF DEBUGGING, A.K.A. "KILL ME! KILL ME NOW!"

I want to call out two important points here. The first is something you just read, but I bet it didn't really sink in, so here it is again: if even a single resource listed in your cache manifest file fails to download properly, the entire process of caching your offline web application will fail. Your browser will fire the `error` event, but there is no indication of what the actual problem was. This can make debugging offline web applications even more frustrating than usual.

The second important point is something that is not, technically speaking, an error, but it will look like a serious browser bug until you realize what's going on. It has to do with exactly how your browser checks whether a cache manifest file has changed. This is a three-phase process. This is boring but important, so pay attention.

1. Via normal HTTP semantics, your browser will check whether the cache manifest has expired. Just like any other file being served over HTTP, your web server will typically include meta-information about the file in the HTTP response headers. Some of these HTTP headers (`Expires` and `Cache-Control`) tell your browser how it is allowed to cache the file without ever asking the server whether it has changed. This kind of caching has nothing to do with offline web applications. It

happens for pretty much every HTML page, stylesheet, script, image, or other resource on the web.

2. If the cache manifest has expired (according to its HTTP headers), then your browser will ask the server whether there is a new version, and if so, the browser will download it. To do this, your browser issues an HTTP request that includes that last-modified date of the cache manifest, which your web server included in the HTTP response headers the last time your browser downloaded the manifest file. If the web server determines that the manifest file hasn't changed since that date, it will simply return a `304 (Not Modified)` status. Again, none of this is specific to offline web applications. This happens for essentially every kind of resource on the web.
3. If the web server thinks the manifest file has changed since that date, it will return an HTTP `200 (OK)` status code, followed by the contents of the new file, along with new `Cache-Control` headers and a new last-modified date, so that steps 1 and 2 will work properly the next time. (HTTP is cool; web servers are always planning for the future. If your web server absolutely must send you a file, it does everything it can to ensure that it doesn't need to send it twice for no reason.) Once it's downloaded the new cache manifest file, your browser will check the contents against the copy it downloaded last time. If the contents of the cache manifest file are the same as they were last time, your browser won't re-download any of the resources listed in the manifest.

Any one of these steps can trip you up while you're developing and testing your offline web application. For example, say you deploy one version of your cache manifest file, then 10 minutes later, you realize you need to add another resource to it. No problem, right? Just add another line and redeploy. Bzzt. Here's what will happen: you reload the page, your browser notices the `manifest` attribute, it fires the `checking` event, and then... nothing. Your browser stubbornly insists that the cache manifest file has not changed. Why? Because, by default, your web server is probably configured to tell browsers to cache static files for a few hours (via HTTP semantics, using `Cache-Control` headers). That means your browser will never get past step 1 of that three-phase process. Sure, the web server knows that the file has changed, but your browser never even gets around to asking the web server. Why? Because the last time your

browser downloaded the cache manifest, the web server told it to cache the resource for a few hours (via HTTP semantics, using `Cache-Control` headers). And now, 10 minutes later, that's exactly what your browser is doing.

To be clear, this is not a bug, it's a feature. Everything is working exactly the way it's supposed to. If web servers didn't have a way to tell browsers (and intermediate proxies) to cache things, the web would collapse overnight. But that's no comfort to you after you spend a few hours trying to figure out why your browser won't notice your updated cache manifest. (And even better, if you wait long enough, it will mysteriously starts working again! Because the HTTP cache expired! Just like it's supposed to! Kill me! Kill me now!)

So here's one thing you should absolutely do: reconfigure your web server so that your cache manifest file is not cacheable by HTTP semantics. If you're running an Apache-based web server, these two lines in your `.htaccess` file will do the trick:

```
ExpiresActive On
ExpiresDefault "access"
```

That will actually disable caching for every file in that directory and all subdirectories. That's probably not what you want in production, so you should either qualify this with a `<Files>` directive so it only affects your cache manifest file, or create a subdirectory that contains nothing but this `.htaccess` file and your cache manifest file. As usual, configuration details vary by web server, so consult your server's documentation for how to control HTTP caching headers.

Once you've disabled HTTP caching on the cache manifest file itself, you'll still have times where you've changed one of the resources in the appcache, but it's still at the same URL on your web server. Here, step 2 of the three-phase process will screw you. If your cache manifest file hasn't changed, the browser will never notice that one of the previously cached resources has changed. Consider the following example:

```
CACHE MANIFEST
```

```
# rev 42
clock.js
clock.css
```

If you change `clock.css` and redeploy it, you won't see the changes, because the cache manifest file itself hasn't changed. Every time you make a change to one of the resources in your offline web application, you'll need to change the cache manifest file itself. This can be as simple as changing a single character. The easiest way I've found to accomplish this is to include a comment line with a revision number. Change the revision number in the comment, then the web server will return the newly changed cache manifest file, your browser will notice that the contents of the file have changed, and it will kick off the process to re-download all the resources listed in the manifest.

```
CACHE MANIFEST
# rev 43
clock.js
clock.css
```

LET'S BUILD ONE!

Remember the Halma game that we introduced in the canvas chapter and later improved by saving state with persistent local storage? Let's take our Halma game offline.

To do that, we need a manifest that lists all the resources the game needs. Well, there's the main HTML page, a single JavaScript file that contains all the game code, and… that's it. There are no images, because all the drawing is done programmatically via the canvas API. All the necessary CSS styles are in a `<style>` element at the top of the

HTML page. So this is our cache manifest:

```
CACHE MANIFEST
halma.html
../halma-localstorage.js
```

A word about paths. I've created an `offline/` subdirectory in the `examples/` directory, and this cache manifest file lives inside the subdirectory. Because the HTML page will need one minor addition to work offline (more on that in a minute), I've created a separate copy of the HTML file, which also lives in the `offline/` subdirectory. But because there are no changes to the JavaScript code itself since we added local storage support, I'm literally reusing the same `.js` file, which lives in the parent directory (`examples/`). Altogether, the files look like this:

```
/examples/localstorage-halma.html
/examples/halma-localstorage.js
/examples/offline/halma.manifest
/examples/offline/halma.html
```

In the cache manifest file (`/examples/offline/halma.manifest`), we want to reference two files. First, the offline version of the HTML file (`/examples/offline/halma.html`). Since these two files are in the same directory, it is listed in the manifest file without any path prefix. Second, the JavaScript file which lives in the parent directory (`/examples/halma-localstorage.js`). This is listed in the manifest file using relative URL notation: `../halma-localstorage.js`. This is just like you might use a relative URL in an `<img src>` attribute. As you'll see in the next example, you can also use absolute paths (that start at the root of the current domain) or even absolute URLs (that point to resources on other domains).

Now, in the HTML file, we need to add the `manifest` attribute that points to the cache manifest file.

```
<!doctype html>
```

```
<html lang="en" manifest="halma.manifest">
```

And that's it! When an offline-capable browser first loads the offline-enabled HTML page, it will download the linked cache manifest file and start downloading all the referenced resources and storing them in the offline application cache. From then on, the offline application algorithm takes over whenever you revisit the page. You can play the game offline, and since it remembers its state locally, you can leave and come back as often as you like.

❧

FURTHER READING

Standards:

- Offline web applications in the HTML5 specification

Browser vendor documentation:

- Offline resources in Firefox
- HTML5 offline application cache, part of the Safari client-side storage and offline applications programming guide

Tutorials and demos:

- Gmail for mobile HTML5 series: using appcache to launch offline - part 1
- Gmail for mobile HTML5 series: using appcache to launch offline - part 2
- Gmail for mobile HTML5 series: using appcache to launch offline - part 3
- Debugging HTML5 offline application cache
- an HTML5 offline image editor and uploader application

- 20 Things I Learned About Browsers and the Web, an advanced demo that uses the application cache and other HTML5 techniques

HTML5 Offline Application Cache Tools:

- Cache Manifest Validator, an online validation service
- Manifesto, a validation bookmarklet

This has been "Let's Take This Offline." The full table of contents has more if you'd like to keep reading.