HTML5 Offline Caching

Intro

It's PeepCode! Today we'll be looking at caching files and data in the web browser, also known as HTML5 offline caching. This is, I think, one of the most useful but least understood features of HTML5.

And we're lucky to have had this content prepared by official W3C CSS working group committee member Ben Schwarz.

Since web applications have become more of a part of our daily lives, there's been a real requirement and yearning to provide a snappy experience for the user across devices of different shapes and sizes, in all kinds of places, over any kind of network. This includes making web applications run over slow network connections, or sometimes with no network connection at all.

Today we're going to explore a couple of HTML5 technologies that are available in most browsers right now. Although this is usually referred to as "offline caching", these features are useful for almost any application, even if they are never used offline.

What are all the tools we have for delivering web content to users quickly? First there are server-based performance improvements for generating content as fast as possible. We might buy a beefier server with a faster CPU, or turn on optimizations in the database, or use a cache on the server. Second, there are optimizations for delivering the content to the user quickly. We might compress data before we send it, or send headers telling the browser to cache images for a long time. Third, there is browser caching. We can tell the browser to keep a copy of files so it doesn't need to ask for them at all. Or we can store API-generated data in the browser so it's ready to use.

All three are important. Sometimes developers only think about the server component and consider their job to be done as long as the server is generating pages quickly. But that's a small part of the speed of your web application. Your users don't care if you can generate a dynamic HTML document in 50ms on the server if it takes an hour to deliver it to them over a slow network connection. If you design a good HTML5 offline caching strategy, those assets may rarely need to travel over the network at all.

That being said, client-side caching should be part of an overall strategy for caching content. It can become confusing for you as a developer if some content is cached on the server, some in a proxy cache like Varnish, and some in thousands of browsers around the world. But in the next hour we'll give you the tools to understand and work with client-side caching.

Specifically, we'll look at two kinds of browser caching: application cache and local storage.

HTML5 Offline Caching

Application cache, or "appcache" uses a file-based manifest that describes the assets used on your site. You tell the browser to store a copy of specific files, usually JavaScripts, stylesheets, and images. In a traditional cache, the browser figures out which files to store, but you don't have much control over what is stored or when the cache is cleared. And browsers have different rules, even between their desktop and mobile versions. For example, some mobile browsers won't cache a full copy of jQuery because it exceeds the size specified by the browser's built-in caching rules. So the user must download a full copy of jQuery every time the page is accessed.

Appcache gives you more control and has a higher total file size limit (usually 5 MB to start with). In our real world tests with a frequently used mobile app over actual cellular networks, an offline manifest reduced page load time from 10-15 seconds down to less than 1 second.

Appcache isn't supported by Internet Explorer, but it works in all other modern browsers. It can be used to explicitly cache cross-domain resources such as Google-hosted jQuery (for non-SSL sites).

The second bit of browser-based caching technology is local storage. Local storage is a key-value store built into your browser. An early version used a SQL database but the current version is a simpler key-value database.

A JavaScript API stores data from a single domain (but not subdomains). It will stay around for months or even years. This is useful for storing API-generated data, such as the list of currency exchange rates that we'll use in this screencast. An API-compatible counterpart called sessionStorage works the same way but only for the current session.

In this screencast we will

- Setup the demo application
- Add an appcache manifest to store assets
- Update the cache
- Use the JavaScript API to the appcache
- Store and use data in localStorage
- Use the appcache with the Rails 3 asset pipeline (but it works great with any server technology).

As with all PeepCode screencasts, we'll cover a lot of information in about an hour. Pause or rewind of you need to view a section again. Use the chapter menu in Quicktime to jump to a section.

HTML5 Offline Caching

Let's get started!

Setup & Demo Application

In this chapter we'll look at the demo application and get it running.

The application we'll be working with is called Currency.io. You an find a live copy at http://currency.io. It's a currency converter that uses the features of HTML5 that we'll use in this screencast: AppCache and localStorage. It works primarily for WebKit, specifically iPhone and iPod Touch. The version we'll run works in desktop browsers as well.

If you run it in the iPhone simulator, you'll be asked to install it to your home screen. Launch it from there.

This application knows when it's online. It also knows when it's offline. It will download currency exchange rates from the server when it can. It will store the entire application on the device (HTML, CSS, and JavaScript) so we don't have to be online. You can take your iPhone around the world and know roughly how much you're spending in your home currency. There is a list of other currencies that we can convert against.

Let's get started with a version application that only works online, and we'll add code and configuration to make it work offline.

In the code download at PeepCode, you'll find a starter demo application. I'm going to take you quickly through the code that's being used here, so you can run the demo application on your local machine.

Open the code in TextMate and you'll discover it's a very simple Sinatra application. It only has two URL routes, one of which serves the "index" HTML file. The other serves JSON-formatted exchange rates for our currency converter.

The rest of the files are served automatically by Sinatra out of the "public" directory.

The demo application uses jQuery to make it easier to understand. The production version you'll find at currency.io did not use jQuery in order to be more lightweight for mobile browsers.

We'll be doing much of our work today in the "javascripts" directory. In particular, we'll implement "offline.js", which I've set up with a jQuery DOMReady event. The application has its own "application.js" file with the logic for converting between currencies. We won't be worried about that code at all today.

We've tried to make it as easy as possible to run this, whether or not you know Ruby. Navigate to the project directory in the Terminal. I'm using Ruby 1.9 but you can use Ruby 1.8 as well.

Run "rake server" and it will install any dependencies and start the server on localhost: 5000. If this doesn't work, email peepcode@topfunky.com with the output of "ruby --version" and "gem --version" and we'll help you get started.

Assuming it worked, you can visit http://localhost:5000 in the browser and you'll see our currency converter.

Next up, we're going to write code to make this currency converter work offline.

Basic Offline

In this chapter we'll look at the Chrome developer console, learn about the appcache mime type, specify a manifest in our HTML document, and build a simple manifest file.

A few years ago we tried to use offline caching in a mobile reporting app but abandoned it after a few weeks because it was too hard to debug. Things are *much* better now, especially with the debugging tools built into the Google Chrome browser.

Press "option command I" to open the developer console. You could also use Safari, Firebug in Firefox, or the Internet Explorer developer tools. But for the remainder of this screencast we'll use Chrome.

Its tools are currently the best for working with the application cache and offline storage. The most important one here is the console tab. It will show informative messages as the page loads. Chrome's console is the most verbose and gives the best feedback to tell you what's happening with the application cache.

The technical details of offline caching are relatively simple, but the mechanism is super sensitive to even the smallest error in your code or omission of a filename. If you've ever taken a driving test and been disqualified for some minor infraction such as...I don't know...running a red light, you know what I mean. Watching the developer console will make you aware of these tiny errors and avoid frustration.

Let's add code to make the application work offline and we'll see how it looks in the console.

We need to write a manifest file that lists the files used by the application and describes how they will be accessed: Do they need to be downloaded and saved locally for offline use? Or are they network resources that should always be requested from the server?

HTML5 Offline Caching

Start by making a list of what assets need to be available offline. Looking into our index.erb view, we'll see some images, some CSS, and some JavaScripts. All these files are necessary to make this application work.

Next, jump back to the application code and add a few important details. We're using Sinatra for simplicity and will only need to write four lines of code.

The first is the MIME type. ".appcache" is the suggested file extension, and it must be set to a MIME type of "text/cache-manifest". If you don't do this, offline caching simply won't work. Chrome is kind enough to tell you that you've set the wrong MIME type, and it will tell you there is a failure. However, on mobile devices this is not the case. So I like to do this step first.

Next, add a route in the Sinatra application to serve the "offline.appcache" manifest file. Use that MIME type as the content type. Render a template with the ERB templating engine.

Add a new file in the "views" directory named "offline.appcache.erb". To get started, we need to use a very simple directive at the start of the file. It must exist and it must be in capitals, CACHE MANIFEST. The second directive tells the browser that we want it to save some files locally: CACHE followed by a colon, and then a new line.

The CACHE is a list of exact files the browser should download for offline use. They must be exact filenames (including the path), not shell globs or directory names.

Looking back at the "index.erb" HTML file, we have a bunch of assets. Let's start minimally with our JavaScripts: dataset.js, jquery, and application.js.

Next, let's tell the browser that a manifest exists by putting it in the HTML file.

Go back to the ERB template and find the HTML element at the top.

Add an attribute to the HTML tag: manifest. The value is the path to our manifest file: offline.appcache.

Save, go back to your browser, and reload. Now we have a lot more output. Let's have a look.

It shows that it's "Creating application cache with manifest" and it links to our offline.appcache file. It runs a "checking" event, a "downloading" event, and a couple of progress events which list the files we asked it to cache. Then a "cached" event which shows that it worked successfully.

This is very exciting! We're caching part of this application.

But actually, this isn't enough to make the application work. Refresh and you'll see that there are no stylesheets or images. Remember what I said about it being super sensitive to any errors or omissions?

In the next chapter, we'll fix that and also learn how to refresh the cache with a new list of files.

Invalidation

In this chapter we'll add to the manifest, learn how to deliver updates to the browser, and look at the lifecycle of events triggered when using an application cache.

At the end of the last chapter, we saw that the second load of the page was missing styling and images. How do we fix this?

Any files not mentioned in the manifest will not be downloaded by the browser at all. Once you start using a manifest, the browser expects to find all necessary files mentioned there in some way. But you can't use any kind of wildcard in the CACHE: section. Only exact URL paths to single files.

The easiest way to get everything to work as before is to use the NETWORK directive with a wildcard. Add a line with the word NETWORK: and put a star on the next line. This will tell the browser to treat any assets not mentioned in the CACHE: section as standard, live assets that should be requested from the server on every load of the page.

This applies to all kinds of files served from any domain. If it's not listed in the manifest, it won't be retrieved or served.

Refresh the browser at least twice and you'll see the entire application as expected.

But there's still a problem. This as a brute force technique that won't actually cache all the assets for offline use or for quick page load. It's good for getting it to work, but not to make it work well.

So let's list the assets required to make our application run. I'll go ahead and do that for you. WestCiv also has a bookmarklet called ManifestR that can help you get started (but it requires customization to work best).

Return to the browser and tap the reload button. The first time we'll see all of the additional assets download and we'll see the application cache "updateready" event.

On the second reload, we'll see the new assets and proper display of the page.

What the "updateready" event means is that there is a new cache in store. The application cache has a staging approach. Imagine two boxes. One is the cache that's available in the browser right now and is being rendered to the screen. The second is the temporary store that loads in the background when a new version of the page is available.

The progress event for each of the assets shows that they have been downloaded to the temporary staging cache in the background. You won't see them until the *next* refresh of the page. This works as a safety mechanism to keep the existing store safe. If there's any error in the background staging cache, the good copy of the cache won't be corrupted and can still be used.

Let's look at what happens when we refresh again.

We see two events, "checking" and a "noupdate" event. The cache that we just downloaded a moment ago has new been transferred into the main store. We're looking at it.

Let's cause an error to see how it works a bit better. I'm going to put a bogus file here. This path does not exist and it's going to cause an error. Go back to the browser and reload.

Our browser has tried to cache that resource. But it's received a resource fetch failed 404 error. Reloading my browser continues to show the same error, but the page still displays correctly to the user. My browser is sensibly using the primary application cache. Each time I reload it's creating a new staging area for the update, since there's an error it doesn't replace the primary cache and continues to use that for display.

So this is why you won't see changes on the initial refresh of a page, only on the second refresh.

We've illustrated the error condition. Remove the "bogus" line to put the application back into a working, functional state.

Go back to the browser and reload. It shows that there was a checking event and a no-update event. So my application cache knows that there is no difference from my master cache to my manifest file. How is that?

So far we've been looking at the client. Let's look at what happens on the server.

Go to the server log in the Terminal and clear the screen (Command-K on the Mac). Then go back to the browser and refresh. The browser checks the manifest file. It went through a "checking" event. Back in the Terminal we'll see something very interesting. There are a couple of duplicate calls in here and you may see some debug messages. The

important line is a “GET” for “offline.appcache” and also a “POST” for “/exchange”. This is the list of exchange rate data. Those are the only requests, just two.

There are no extra requests for images or JavaScripts or any other asset listed in the file. Where are they coming from? The answer is the application cache. The browser has a copy and doesn’t ask the server for a copy. It *will* ask for “offline.appcache” every time to see if there is an update, but it won’t ask for assets it already has. If the contents of offline.appcache change at all, it will download all assets again. This is how you trigger a refresh of the cache: make any change to the appcache manifest.

We can explore the items that are stored in our application cache using the Chrome developer tools. Click on “resources” to show a new section. Scroll to the bottom and find Application Cache, “localhost” section.

This lists all the files we’re caching. We’ve got the manifest and some assets. But what’s this “Master” asset? It wasn’t listed in our manifest.

That’s the root of our application. Because we’re linking to the application cache manifest from an HTML document, it realizes that this HTML document should be cached, too. It automatically put this into our cache file whether we like it or not.

This might affect your choice to use an application cache. For example, if you tried to use application cache on a traditional blog, visitors wouldn’t see the newest articles, only the article from one visit ago. The mechanism pretty much assumes that you’ll use JavaScript to update the page with fresh information *after* the HTML document has loaded.

Next, let’s use the JavaScript API to display a progress bar while a new version of the application loads in the background.

Progress & Debugging

In this chapter we’ll add a progress bar to the screen and control it with JavaScript. This will illustrate the workflow of updating a cached document during development.

Now that our application is loading all the necessary assets from our application cache and we verified what’s in there, I think it’d be nice to add something to show the user that the application is being updated in the background. This is similar to a Mac OS X updater, where you see that an update is being downloaded and have a chance to reload the application. The only difference is that we’ll download it automatically.

HTML5 Offline Caching

Go back to the index.erb HTML template. Scroll down to the section with ID “rates”. We’ll notice it says “loading” in the markup, but if we check the interface, it doesn’t actually show that “loading” text.

The application’s JavaScript uses this section to show the currencies being converted. It’s a good place to display a progress bar; at the top of the screen.

I’m going to use a brand new HTML5 element: “progress”. The “progress” element’s important attributes are a minimum value of 0, and a maximum value of 100. The current value is 0.

If I go back to the browser and reload, you’ll notice that nothing happens. This is a symptom of the application cache. It’s caching everything and doesn’t show any changes because there hasn’t been a change to the manifest file.

This can be a point of confusion for many developers, so there are a few techniques that are available. One option is to version your manifest file. Put a comment in the file that includes a version number that you can increment to force clients to download a new version of the application. Comments start with a hash mark.

Refresh the browser, and we’ll see the application cache was updated, but the display looks exactly the same. Refresh the second time to view the updated page. There is a nice little progress bar with absolutely nothing in it. Let’s work with it in the console.

I’ve got DOM handle on the ID “rates”. I’ll use plain JavaScript to find the child nodes and the node at index 1, or the second item which is my progress bar element.

Modify the value of the progress element by setting a “value”. You’ll see a progress bar that is 45% complete. I can change that up to 100%.

Let’s move on to style it and work out how we can fill it from the application cache API. But this will be frustrating if we have to refresh twice every time. So let’s think about how to work with this in development.

The easiest option is to remove the application manifest all together. Or, use your web framework’s view templating language to only show the manifest attribute if you’re in production mode. It will save you a lot of time and stress.

But you’ll notice the application cache is still being used. Very Frustrating. Let’s use Chrome again to help with this.

Visit chrome://appcache-internals or go to about:about. In here you’ll find lots of interesting things about the Chrome browser, but near the top is appcache-internals.

The current version of this shows the contents of the cache and allows us to remove it altogether.

Go back to the application URL and tap reload. You'll notice that the application cache is no longer being used. So we're free to make changes to our application and see results immediately.

I've prepared some styles for this progress bar element that will make it look a bit better. I'll go to my application.css file, locate the right place to put it, and I'll put in some styles. First, I'm giving it a width of 25% of its parent, a height of about 10 pixels, a margin that's automatically set to the left and right, and I'll make it display in block style.

Refresh the browser to see it.

But we probably want to hide this progress bar initially. We want a script to show it when it is actually doing something. So I'll go back to the stylesheet and update it to "display: none". You'll see it disappear.

In the next chapter, we'll use JavaScript to update the progress bar.

JavaScript API

In this chapter we'll display and update the progress bar as the appcache is being updated.

To show our progress bar and display the correct progress as files are downloaded, we're going to need to use JavaScript. Application cache has a JavaScript API that will allow us to use events to track the status of what the cache is doing at any given time. And it will enable us to fill the progress bar.

Go to the browser console and type "window.applicationCache". Yes, we're now at *three* slightly different terms: manifest, appcache, and applicationCache. Did you expect consistency from a web browser API?

A DOMApplicationCache object is returned. Opening it shows a bunch of events, some of which we've seen already: oncache, downloading, error, uploading, obsolete, progress, updateready, and status.

Here's a brief description of each:

- The *checking* event is the first one used, to check if there is an update to the cache manifest. All the events thereafter are actually optional.

- The *noupdate* event, means the manifest wasn't updated.
- *Downloading* says files and assets have begun to be downloaded.
- *Progress* events mean a file has been completed.
- The *cached* event means the download of the cache was successful,
- The *updateready* event means the cache was successfully moved from the temporary staging area into the primary application cache.

To show our progress bar, we'll use the *downloading* event because it means that we've started to download some new files. To fill the bar we'll use the *progress* event.

Going back to our application code, we're finally going to use the offline.js file. Add it to "index.erb". Verify that it works with a console debug message.

Refresh the browser and there it is, it's logging.

Next, I want to check that the current browser actually supports application cache. I'll check for specific features instead of using a compatibility framework like Modernizr. I just want to know if the browser supports applicationCache, so I'll use this conditional. The "!!" turns any existing object into a boolean "true", and turns a null or undefined value into "false".

If the browser does support applicationCache, I'll add a callback to the applicationCache object. Although we have access to jQuery, I'm using straight JavaScript to add an event listener, which is to say, I'm defining a callback function that will be called when the "downloading" event occurs. The third argument deals with a browser implementation detail. Just use the standard "false" here.

Now tell the progress element to display in block fashion.

Now let's add an event to fill the progress bar as the manifest downloads, and another to hide it once it's complete. I'll be using the applicationCache object several times, so let's set a local variable to make it more concise.

The first event we're going to add is the *progress* event. The anonymous function receives an argument with the progress event. I'll do some math and assign it to a "percentage" variable. The *progress* event has a property named *loaded* and another named *total*. Divide them, then multiply by 100 to calculate a value in the range 0 to 100.

Then set that value as the value of our progress bar.

That gives us a handle on the start of the sync and gives us progress as files are received. Finding an event on which to determine that the sync has *concluded* is more vague.

The final event could be one of many events. It could be any of the *cached, updateready, error,* or *obsolete* events. We should hide the progress bar after any of them, so let's create a function that will hide the progress bar. I'll pass the function to each of the aforementioned events.

I'll paste the code here. We're defining a *complete* function and passing it to the *cached*, *updateready*, *error*, and *obsolete* events. That's looking pretty good.

If we test this in the browser, everything will happen so fast that we won't ever see the progress bar. So let's simulate slow network traffic with Sinatra. Add a surplus route that sleeps for one second. I'll add this to the manifest to slow it down. Go to the offline.appcache.erb file and list the "sleep" route. Duplicate several times, how about eight?

We're in development but we want to see the effect of the appcache. So add the manifest back to the HTML document.

Go to the browser. Before loading the page, clear the cache at chrome://appcache-internals. Now reload and see what happens.

There's our progress bar. The events are slightly delayed, but we saw the progress bar and it showed that an update occurred.

Status

In this chapter we'll continue using JavaScript, but this time we'll show the user's online/offline status.

Since the very beginning of this screencast, you may have noticed this online area of the screen. To be honest it's actually always been faked. If you look at our marker it's hard coded to show that we're online.

We have an element with the ID "network-status". It has a class "online", and if you open our application stylesheet, we've got the class "online" that prepends a green tick or a red cross.

Let's change this indicator based on the network status. We'll take the "network-status" ID and hook into JavaScript events to add the "online" or "offline" class to it.

HTML5 Offline Caching

Let's start with offline. Write a function assigned to a variable because I know I'll use it at least twice.

There's window event named "offline". We're going to send it the "offline" function and set the third "false" argument to not use event bubbling.

But the "offline" event only fires if we move from online to offline. It doesn't fire when the page loads initially.

So let's check another object to make sure we're setup correctly from the start.

Again, since this is a web browser, it's only expected that we find the online/offline status from a completely different object with a different API. The "navigator" object has an "onLine" property (note the capital L). Call the "offline" function directly if we're not onLine.

The complement happens with the "online" event, back on the "window" object. Use an anonymous function since it's only used once. Set and remove the appropriate CSS classes.

Let's test it out in our browser.

Clear the console with Command-K. Query the onLine property with "navigator.onLine".

The browser says we're online. I'll turn off my WiFi to try offline. The browser now knows that we're not online. If I inspect the navigator.onLine property, it confirms; we're not online.

Try it out by disabling your WiFi and refreshing the browser. It should display the proper status from start to finish.

API Data Storage

In this chapter, you'll learn how to store arbitrary data for offline use, or even just faster retrieval at any time.

So far we've had the luxury of working online. Or at least, I'm serving the application from the local machine, so the server is always available. So this is by no means a real world example of what we would see if we were actually offline. If I kill my web server, previously running in the background, and head back to my browser and refresh, I still get the interface. View the console and you'll see the application cache threw an error on

the GET to the manifest. The application and assets are being served from the AppCache.

But if I type in some numbers you'll see that no calculation is being made. All "conversions" are one to one. What's happening? In our application code we have a file named dataset.js. This gives us a default list of currencies. We set everything against the US dollar and then do a conversion from there. We also have our default conversion which is from Japanese Yen to Australian Dollars.

In our application code, we attempt to do an Ajax request for the live conversion rates. When offline, this Ajax request does not succeed and we don't get any currency data.

To solve that, let's store a local copy of the conversion rate data whenever we're online. We can use that as a backup data source whenever we're offline.

LocalStorage is a key-value database that saves data indefinitely. The developer console conveniently has a section for localStorage. We'll see it here once we start setting values.

LocalStorage is a JavaScript API and works like the properties on JavaScript objects. We can set an item by giving it a key and a value. I can retrieve its value by using the getItem method. Alternately I can access it with square brackets. Dot notation also works.

If I try to store an object to localStorage, it silently returns the object as if it was successful.

But it only stored the string "[object Object]". Not too useful! Local storage will only store a string.

Here's the trick: every browser that supports localStorage will also have a native JSON parser and emitter. We can use it to generate a string from any object and store that.

Try this. Call "JSON.stringify" on an object and store it in localStorage. I get a serialized JSON copy of my object.

Conversely, I can parse the string to build the native JavaScript object again.

This is perfect for storing any data in local storage. Make a storage object and store a string. Parse it on the way out. Why this wasn't included as part of the built-in API, we'll never know. But all the functionality is there for us to use.

This will make a perfect database for our currency data. Let's do that in the next chapter.

Network

In this chapter we'll use the NETWORK directive in the manifest, and we'll store and retrieve application data with localStorage.

We've built a manifest that lists static assets the browser should cache. But most JavaScript-driven applications need to access live data, too. This is the kind of data we'll save in localStorage, but the manifest must also be edited to let those requests through when the site is live.

We briefly saw the NETWORK: directive. The NETWORK section lists URLs that the browser should not cache, but should retrieve every time the browser is online. We used it with a wildcard to serve all routes that are not explicitly cached.

Let's use it more traditionally here to list URLs used by Ajax API calls.

In application.js there is an updateCurrencies function. It runs a POST to "/exchange". Add this to the offline.appcache manifest to explicitly note that you'll need internet access for this route. While I'm doing that I'll also clean up these "sleep" URLs from the previous chapter.

Add the NETWORK: section and list "/exchange". If we're offline and "/exchange" can't be reached, it's up to our custom application to figure out what to do. The browser won't help.

There is also an optional directive named FALLBACK. If the markup mentions a URL path, it should cache an alternate resource and use that instead when offline. The best example is an avatar image. If someone tries to access http://gravatar.com while offline, give them /default_avatar.png. But do you really want to show a page full of the default avatar? That wouldn't be very useful. It seems more useful to intelligently cache the most recent 50 avatars visited by the user and show it if the browser is offline. Otherwise the usefulness of the application would degrade drastically when offline.

FALLBACK also works only for GET requests.

In our case, our Ajax will POST to /exchange, and the NETWORK directive handles that properly. But we will have to save the data manually.

Let's update the application code. At the moment, the updateCurrencies method is being run even though we're not online. Add a check as before and return immediately if we're not online.

Next, store the currencies in localStorage after we've received data from a successful Ajax request. Use the localStorage object. Save to the "currencies" key. Serialize the

currencies object with JSON.stringify. Then assign it to window.currencies for use elsewhere.

The rest of the work happens in dataset.js. We'll use localStorage by default rather than using the more temporary "window" variable.

Start with a variable named "currencies" for the defaults. Then set the "window.currencies" property from the value in localStorage. We've also got this property "fromTo" which stores the currencies being converted between. We'll extract that from localStorage as well so they're saved between sessions.

On a previous line, set the defaults. If localStorage.currencies is empty, use the serialized version of the currencies.

Do the same for fromTo. Serialize it and set it in localStorage.

Now try it out. Enable your WiFi connection and restart the web server. To be thorough, I'll remove the current application cache.

Refresh the browser and watch the manifest download. We should see currencies both in the web interface and in the localStorage database in the developer console.

Now try it offline. Kill the web server. Then reload the browser. We'll notice the manifest hasn't been able to download, but we're still able to convert currencies and we get meaningful conversion values. The currencies have been stored in localStorage and we're able to use the application completely offline.

Gotchas

Using browser caching can improve the speed of your application, but it can also cause you, the developer, frustration. Here are a few tips to help you.

First, there's no support for the appcache manifest in Internet Explorer. If you go to caniuse.com, and search for AppCache, you'll find offline web applications. It shows the current status across the board of browsers, including some mobile browsers. It's also supported in IOS 5.

But you'll notice that it's not even scheduled for IE 10. This might not matter, because IE will just ignore the manifest and your application should work normally, just slower and never offline.

Second, HTTPS connections. If your application is not served over HTTPS, you can cache resources from other sites, such as avatars, images, JavaScripts, etc. But if you're

serving over HTTPS, most browsers will respect the same origin header, which means all your resources must be served from the same domain. The only browser with an exception is Chrome. Chrome is the best for developing and debugging AppCache, but it might cause you to think that all browsers will work the same way. They won't. Be sure to test on all browsers that you intend to support.

Third, cache control headers. If an asset is served with a Cache-Control header of "no-cache", it will be respected by the browser. You won't be able to cache it with application cache at all. It's a bit frustrating as a developer but I think this one is really for the better.

The fourth one is quite simple, Firefox warnings. If you use Firefox to visit a site that uses application cache, you'll see this warning. The website wants to store data on your computer for offline use. That's a pretty scary message and it's definitely not a good user experience. I don't really know if they're going to keep up with this trend. Certainly no mobile browser has anything like this. Neither Chrome nor Safari have it. Hopefully it goes away in future versions of Firefox and we can use application cache transparently.

Finally, browser support for localStorage. On caniuse.com, search for localStorage, or sessionStorage or web storage, and you'll see this table. They once fell under HTML5 but they're in their own separate specifications now.

The browser support for web storage is fairly good and is supported by most browsers.

A useful API not mentioned is "clear", which will purge the entire contents of localStorage for the given domain that you have control of.

Offline Caching with Rails 3

Let's wrap up this screencast by looking at browser caching issues in Rails 3. Important points are defining the appcache mime type, using the asset pipeline in the manifest, and testing it out in production.

Rails 3.1 and later include the new "asset pipeline" that generates unique filenames for all JavaScript, CSS, image, and other assets in production. This makes it a bit tough to write an appcache manifest that will cache those files. Or will it? Let's find out.

I'll start by generating a basic Rails 3 application. I'll do it without active record, and without an automatic run of Bundler. I like to run Bundler such that it installs dependencies locally in a "tmp" directory inside the application. I'll do that here. And I'll edit the application.

HTML5 Offline Caching

The first task is to register the ".appcache" file type with the matching mime type. Go to "config/initializers/mime_types.rb". I'm using our PeepOpen fuzzy file finder to navigate.

It gives you an example in the comments. Edit it so "appcache" files are served with the "text/cache-manifest" content type. Save the file.

Next I want to serve a dynamically generated template for the appcache file. Create a controller for the appcache. You don't need any generator. Just create a file in "app/controllers" named "appcache_controller.rb".

While we're here, make a directory for the templates: "app/views/appcache". And the production template: "app/views/appcache/production.appcache.erb". We'll use ERB templates and Rails helpers to populate the filenames in the manifest.

But before we do that, add the appcache manifest to the Rails routes in "config/routes.rb". I'll put this at the root of the URL path without any directory. I'm designing so "/production.appcache" will serve a file from the AppcacheController using the previously defined "appcache" mime type. Save that.

Now open AppcacheController. Define a standard Rails controller that inherits from ApplicationController. For extra speed you could even inherit directly from ActionController::Base (this controller won't need authentication or the other filters that are commonly added to ApplicationController).

We'll implement everything we need in the action template, so specify "layout false"; no layout should ever be used for the actions in this controller. That's all the code we'll need here. Rails will automatically render the production.appcache.erb template without any other methods in this controller.

Now go to the production.appcache.erb template we created a minute ago. This is our manifest. You know what goes here: A CACHE: section and a NETWORK: section with the wildcard. But with Rails 3, production assets have special filenames that uniquely identify them. We can't just write "application.js" because in production, it will have a name like "application-12345.js" that changes every time the file is edited.

For that, there's the "asset_path" helper. And because we're using an ERB template, we can use any helper methods that Rails provides.

Don't list the initial directory of "javascripts" or "stylesheets", just the name of the "application.js" file and the "application.css" file. Images work, too. I'm going to put "mountain-goat.jpg" in the HTML document associated with this manifest. This

photograph was taken in Yosemite National Park by PeepCode's own designer, Paula Lavalle.

I've copied that to "app/assets/images" in the final project, but don't put "images" in the argument to "asset_path". You only need to use a path if there is a subdirectory inside "app/assets/images".

I want to try this out in production mode, but the Rails 3 defaults require a few extra steps: turn on static assets and generate those assets.

Go to "config/environments/production.rb" and find the "serve_static_assets" line. It will start out as "false". Switch that to "true".

Now we need to generate those assets with the unique filenames. There's a rake task that will do that. Run "rake assets:precompile". Look at the contents of "public/assets" to see what it generated. Be aware that these two steps will make it difficult to develop. You'll need to delete "public/assets" before working in development mode again. But we'll run in production mode to see how this works.

Start "rails server" with "-e production". Go to "localhost:3000" and view the "production.appcache" that was generated from our template. Sure enough, it includes the assets we requested, but it shows the production version with the fingerprinted filename.

There's a useful side benefit of this. When I used offline caching with Rails a few years ago, I had to figure out a way to get the browser to refresh the cache when assets were edited. Usually that meant putting a comment in the manifest with some kind of version number.

Now, the fingerprint in the filename changes when the contents of the file change. So the manifest will automatically change when the files do. And your browser clients will get a new copy of the cache when it changes. Conversely, they won't need to download the files again if nothing has changed. Pretty cool.

However, the HTML of the documents that are automatically cached with every manifest will *not* be versioned by Rails. That's a good reason to keep your HTML files as minimal as possible if they are serving a JavaScript-powered application. Put client-side JavaScript templates in JavaScript or JST files instead of in the HTML markup. You might make a change to an HTML file and never see it on the client because the client is working with a cached copy of the HTML.

There's one more tip that will help you in development on Rails 3. Let's skip ahead to where we have a MountainGoatsController, a route for that resource, and an

index.html.erb view. In the layout for this controller, I want to list the “manifest” attribute only in production. This makes it easy for me to develop without the appcache, but to deploy to production with the appcache.

For that, use a custom Rails helper method. Call it from the layout. I’ll name it “manifest_attribute”. Define it in “application_helper.rb”. If we’re in production mode, return the string “manifest=‘production.appcache’”. Otherwise, return blank.

Because we’re generating a string that will be inside an HTML tag, you’ll need to call “.html_safe” on the resulting string. Without this, Rails will try to escape the content and it won’t work correctly.

So now you have three tips for using appcache with a Rails 3 application. Define a mime type, use the asset_path helper, and write a manifest attribute helper.

Client-side caching can make a huge difference in the performance of your applications. It’s perfect for fully JavaScript-powered applications, both mobile and desktop. It can be useful for other kinds of applications, too. And localStorage is a perfect compliment to these, making offline applications possible, and giving you options for improving the performance of any application that uses data on the client.