Jake Archibald wrote... who?

Service Worker - first draft published

Posted 08 May 2014 with an unbecoming amount of excitement

The first draft of the service worker spec was published today! It's been a collaborative effort between Google, Samsung, Mozilla and others, and implementations for Chrome and Firefox are being actively developed. Anyone interesting in the web competing with native apps should be excited by this.

Update: Is ServiceWorker ready? - track the implementation status across the main browsers.

So, what is it?

If text isn't your thing, you can see/hear me spout a lot of this stuff in a 15 minute intro video, or this full-fat 50 minute Google I/O talk. No? Ok, on with the text...

The service worker is like a shared worker, but whereas pages control a shared worker, a service worker controls pages.

Being able to run JavaScript before a page exists opens up many possibilities, and the first feature we're adding is interception and modification of navigation and resource requests. This lets you tweak the serving of content, all the way up to treating network-connectivity as an enhancement. It's like having a proxy server running on the client.

Browsers don't yet support all of the features in this post, but there's some stuff you can play around with. Is ServiceWorker ready tracks the implementation status & the flags you'll need to activiate in order to use them.

Making something work offline-first

Trained To Thrill is a silly little demo app that pulls pictures of trains from Flickr (although Flickr's search is super broken right now, so sometimes it displays nothing).

I built it in a way that uses a service worker to get to first render with cached trains without hitting the network. If there's no cached trains or the browser doesn't support service workers (which is all of them at the moment), it works without the offline enhancement.

The service worker code is pretty simple, as is the in-page code that races the network and the cache.

Registering a service worker

```
if ('serviceWorker' in navigator) {
```

```
  navigator.serviceWorker
    .register('/my-blog/sw.js', {
      scope: '/my-blog/',
    })
    .then(function (sw) {
      // registration worked!
    })
    .catch(function () {
      // registration failed :(
    });
}
```

If registration worked, `/my-blog/sw.js` will begin installing for URLs that start `/my-blog/`. The scope is optional, defaulting to the whole origin.

Registration will fail if the URLs are on a different origin to the page, the script fails to download & parse, or the origin is not HTTPS.

Wait, service workers are HTTPS-only?

HTTP is wide-open to man-in-the-middle attacks. The router(s) & ISP you're connected to could have freely modified the content of this page. It could alter the views I'm expressing here, or add login boxes that attempt to trick you. Even this paragraph could have been added by a third party to throw you off the scent. But it wasn't. OR WAS IT? You don't know. Even when you're somewhere 'safe', caching tricks can make the 'hacks' live longer. We should be terrified that the majority of trusted-content websites (such as news sites) are served over HTTP.

To avoid giving attackers more power, service workers are HTTPS-only. This may be relaxed in future, but will come with heavy restrictions to prevent a MITM taking over HTTP sites.

Thankfully, popular static hosting sites such as github.io, Amazon S3, and Google Drive all support HTTPS, so if you want to build something to test/demo service workers you can do it there. Browser developer tools will let you use service worker via HTTP for development servers.

Messing around with the network

`/my-blog/sw.js` will run in a worker context. It doesn't have DOM access and will run on a different thread to JavaScript in pages. Within it, you can listen for the "fetch" event:

```
self.addEventListener('fetch', function (event) {
  console.log(event.request);
});
```

This event will be fired when you navigate to a page within `/my-blog/*`, but also for any request originating from those pages, even if it's to another origin. The request object will tell you about the url, method, headers, body (with some security-based restrictions).

It won't fire for the page that registered it, not yet anyway. Pages continue using the service worker they're born with, which includes "no service worker". You can change this behaviour, which I'll come to later. By default, if you refresh the page, it'll use the new service worker.

Back to the "fetch" event, like other events you can prevent the default and do something else.

```
self.addEventListener('fetch', function (event) {
  event.respondWith(new Response('This came from the service worker!'));
});
```

Now every navigation within `/blog/*` will display "This came from the service worker!". `respondWith` takes a `Response` or a promise for a `Response`.

The service worker API uses promises for anything async, if you're unfamiliar with promises, now's a good time to learn!

Making a site work offline

The service worker comes with a few toys to make responding without a network connection easier. Firstly, the service worker has a life-cycle:

Download
Install
Activate

You can use the install event to prepare your service worker:

```
self.addEventListener('install', function (event) {
  event.waitUntil(
    caches.open('static-v1').then(function (cache) {
      return cache.addAll([
        '/my-blog/',
        '/my-blog/fallback.html',
        '//mycdn.com/style.css',
        '//mycdn.com/script.js',
      ]);
    }),
  );
});
```

Service worker introduces a new storage API, a place to store responses keyed by request, similar to the browser cache. You can have as many caches as you want.

`waitUntil` takes a promise, this defines the length of the install process. Pages won't be controlled by this service

worker until it fulfils. If the promise rejects, that indicates install failure and this service worker won't be responsible for further events.

`cache.addAll([requestsOrURLs...])` is an atomic operation and returns a promise. If any of the requests fail, the cache isn't modified, and the promise rejects (failing installation).

Lets use that cache:

```
self.addEventListener('fetch', function (event) {
  event.respondWith(caches.match(event.request));
});
```

Here we're hijacking all fetches and responding with whatever matches the incoming request in the caches. You can specify a particular cache if you want, but I'm happy with any match.

`caches.match(requestOrURL)` returns a promise for a `Response`, just what `respondWith` needs. Matching is done in a similar way to HTTP, it matches on url+method and obeys Vary headers. Unlike the browser cache, the service worker cache ignores freshness, things stay in the cache until *you* remove or overwrite them.

Recovering from failure

Unfortunately, the promise returned by `caches.match` will resolve with null if no match is found, meaning you'll get what looks like a network failure. However, **by the power of promises** you can fix this!

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

Here we've caught the error, and attempted something else. `event.default()` returns a response-promise for the original request. You could also call `fetch(requestOrURL)` to get a response-promise for any URL.

Of course, if the item isn't in the cache and the user has no network connection, you'll still get a network failure. However, **by the power of promises** etc etc...

```
self.addEventListener('fetch', function (event) {
```

```
  event.respondWith(
    caches
      .match(event.request)
      .then(function (response) {
        return response || event.default();
      })
      .catch(function () {
        return caches.match('/my-blog/fallback.html');
      }),
  );
});
```

We can rely on `/my-blog/fallback.html` being in the cache, because we depended on it in our installation step.

This is a really simple example, but it gives you the tools to handle requests however you want. Maybe you want to try the network before going to a cache, maybe you want to add certain things to the cache as you respond via the network (which Trained To Thrill example does). I dunno, do what you want.

You can even respond to a local URL with a response from another origin. This isn't a security risk, as the only person you can fool is yourself. Response visibility is judged by *it's* origin, rather than the original URL. If you give a local XHR request an opaque response (as in, from another origin with no CORS headers), it'll fail with a security error, as it would if it requested the cross-origin URL directly.

This is a low-level API. We don't want to make shortcuts until we know where people want to go. AppCache made this mistake, and we're left with a terse easy-to-make manifest that doesn't do what we want and is near-impossible to debug. Although service workers require more code for similar things, you know what to expect because you're telling it what to do, and if it doesn't do what you expect you can investigate it like you would any other piece of JavaScript.

Updating service workers

If you just need to update a cache or two, you can do that as part of background synchronisation, which is another spec in development. However, sometimes you want to change logic, or carefully control when new caches will be used.

Your service worker is checked for updates on each page navigation, although you can use HTTP Cache-Control headers to reduce this frequency (with a maximum of a day to avoid the immortal AppCache problem). If the worker is byte-different, it's considered to be a new version. It'll be loaded and its `install` event is fired:

```
self.addEventListener('install', function (event) {
```

```
  event.waitUntil(
    caches.open('static-v2').then(function (cache) {
      return cache.addAll([
        '/my-blog/',
        '/my-blog/fallback.html',
        '//mycdn.com/style-v2.css',
        '//mycdn.com/script-v2.js',
      ]);
    }),
  );
});
```

While this happens, the previous version is still responsible for fetches. The new version is installing in the background. Note that I'm calling my new cache 'static-v2', so the previous cache isn't disturbed.

Once install completes, this new version will remain in-waiting until all pages using the current version unload. If you only have one tab open, a refresh is enough. If you can't wait that long, you can call `event.replace()` in your install event, but you'll need to be aware you're now in control of pages loaded using *some previous version*.

When no pages are using the current version, the new worker activates and becomes responsible for fetches. You also get an activate event:

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

  event.waitUntil(
    caches.keys(function (cacheNames) {
      return Promise.all(
        cacheNames.map(function (cacheName) {
          if (cacheWhitelist.indexOf(cacheName) == -1) {
            return caches.delete(cacheName);
          }
        }),
      );
    }),
  );
});
```

This is a good time to do stuff that would have broken the previous version while it was still running. Eg getting rid of old caches, schema migrations, moving stuff around in the filesystem API. Promises past into `waitUntil` will block other events until completion, so you know all the migrations & clean-up have completed when you get your first "fetch" event.

This lets you perform updates without disrupting the user, something Android native apps don't do awfully well at.

Performance

Memory

Service workers have been built to be fully async. APIs such as synchronous XHR and localStorage are banished from this place. A single service worker instance will be able to handle many connections in parallel, just as node.js can.

The service worker isn't constantly needed, so the browser is free to terminate it and spin it up next time an event needs to be triggered. High memory systems may choose to keep the worker running longer to save on startup cost for the next event, low memory systems may terminate it after each event is complete.

That means you can't retain state in the global scope:

```
var hitCounter = 0;

this.addEventListener('fetch', function (event) {
  hitCounter++;
  event.respondWith(new Response('Hit number ' + hitCounter));
});
```

This may respond with hit numbers `1, 2, 3, 4, 1, 2, 1, 2, 1, 1` etc. If you want to retain state, you'll have to save that data in browser storage such as IndexedDB.

Latency

Running JS per request may have a performance impact. We have ideas to solve that, eg declarative routes such as `addRoute(urlRe, sources...)`, where sources is a preference-ordered list of where the browser should look for a response. However, we don't want to add workarounds to performance issues until we have performance issues to work around. Without an implementation we don't know the size and shape of those issues, or if we have any at all.

This is just the start

The above is a brief look at some of the API. For full details, see the GitHub repo. If you spot bugs, file issues. If you have questions, ask away in the comments.

I expect the API to shift around as bugs are spotted and various browser vendors spot better ways of doing particular things, I'll keep this post updated as that happens.

Excitingly, the service worker context may be used by other specifications. This is great as many features the web is missing need to run code independent of pages. Things like:

Background synchronisation (specification on GitHub)
Reacting to a push message (latest draft uses service workers)
Reacting to a particular time & date
Entering a geo-fence

These will be developed independently to the core service worker spec, but if we can nail this, we get many of the

features that make native apps a more attractive option, and that makes me happy.

Further reading

The spec

Is ServiceWorker ready? - track the implementation status across the main browsers

JavaScript promises, there and back again - guide to promises

ES7 async functions - use promises to make async code even easier

The browser cache is vary broken - some research that went into the ServiceWorker cache

View this page on GitHub

Comments powered by Disqus