Nicolas Bevacqua May 8, 2017

I’ ve been playing around with ServiceWorker a lot recently, so when Chris asked me to write an article about it I couldn’ t have been more thrilled. ServiceWorker is the most impactful modern web technology since Ajax. It’ s an API that lives inside the browser and sits between your web pages and your application servers. Once installed and activated, a ServiceWorker can programmatically determine how to respond to requests for resources from your origin, even when the browser is offline. ServiceWorker can be used to power the so-called “Offline First” web.

ServiceWorker is a progressive technology, and in this article, I’ ll show you how to take a website and make it available offline for humans who are using a modern browser while leaving humans with unsupported browsers unaffected.

Here’ s a silent, 26 second video of a supporting browser (Chrome) going offline and the final demo site still working:

Hey! If you would like to just look at the code, there is a Simple Offline Site (https://github.com/chriscoyier/Simple-Offline-Site) repo we’ ve built for this. You can see the entire thing as a CodePen Project (https://codepen.io/chriscoyier/project/editor/ZPvNRA/) , and it’ s even a full on demo website (https://simpleoffline.website/) .

(#aa-browser-support) Browser Support

Today, ServiceWorker has browser support in Google Chrome, Opera, and in Firefox behind a configuration flag. Microsoft is likely to work on it (https://twitter.com/jacobrossi/status/608291251121618944) soon. There’ s no official word from Apple’ s Safari yet.

Jake Archibald has a page tracking the support of all the ServiceWorker-related technologies (https://jakearchibald.github.io/isserviceworkerready/) .

This browser support data is from Caniuse (http://caniuse.com/#feat=serviceworkers) , which has more detail. A number indicates that browser supports the feature at that version and up.

Desktop

🟩 : 45 🟩 : 44 🟥 : No 🟩 : 17 🟩 : 11.1

Mobile / Tablet

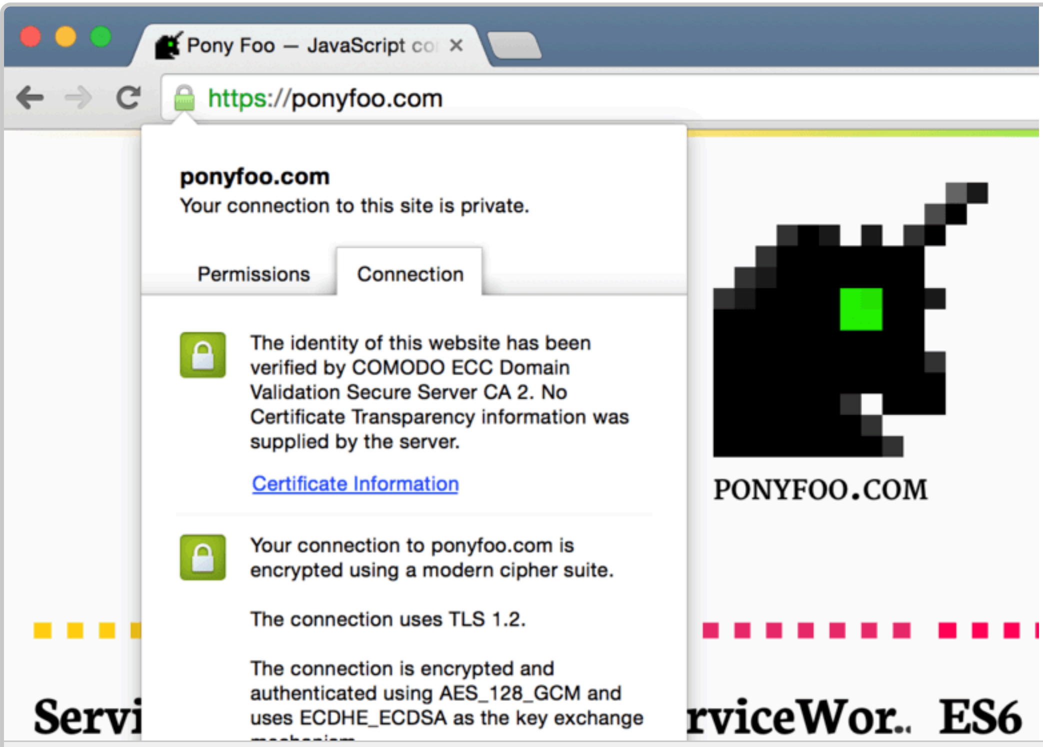
🟩 : 131 🟩 : 132 🟧 : 131 🟩 : 11.3-11.4

Given the fact that you can implement this stuff in a progressive enhancement style (doesn’ t affect unsupported browsers), it’ s a great opportunity to get ahead of the pack. The ones that are supported are going to greatly appreciate it.

Before getting started, I should point out a couple of things for you to take into account.

(#aa-secure-connections-only) Secure Connections Only

You should know that there are a few hard requirements when it comes to ServiceWorker. First and foremost, **your site needs to be served over a secure connection**. If you’ re still serving your site over HTTP, it might be a good excuse to implement HTTPS.



ponyfoo.com
Your connection to this site is private.

The identity of this website has been verified by COMODO ECC Domain Validation Secure Server CA 2. No Certificate Transparency information was supplied by the server.

Your connection to ponyfoo.com is encrypted using a modern cipher suite.

The connection uses TLS 1.2.

The connection is encrypted and authenticated using AES_128_GCM and uses ECDHE_ECDSA as the key exchange

HTTPS is required for ServiceWorker

You could use a CDN proxy like CloudFlare to serve traffic securely. Remember to find and fix mixed content warnings as some browsers may warn your customers about your site being unsafe, otherwise.

> I wrote a tutorial that may help you set up CloudFlare for your site (https://ponyfoo.com/articles/securing-your-web-app-in-3-easy-steps)
> You might want to leverage LetsEncrypt.org (https://letsencrypt.org/) to get a free TLS certificate

While the spec for HTTP/2 doesn’ t inherently enforce encrypted connections, browsers intend to implement HTTP/2 and similar technologies *only* over HTTPS. The ServiceWorker specification, on the other hand, recommends browser implementation over HTTPS (http://www.w3.org/TR/service-workers/#security-considerations) . Browsers have also hinted at marking sites served over unencrypted connections as insecure. Search engines penalize unencrypted results.

“HTTPS only” is the browsers way of saying *“this is important, you should do this”* .

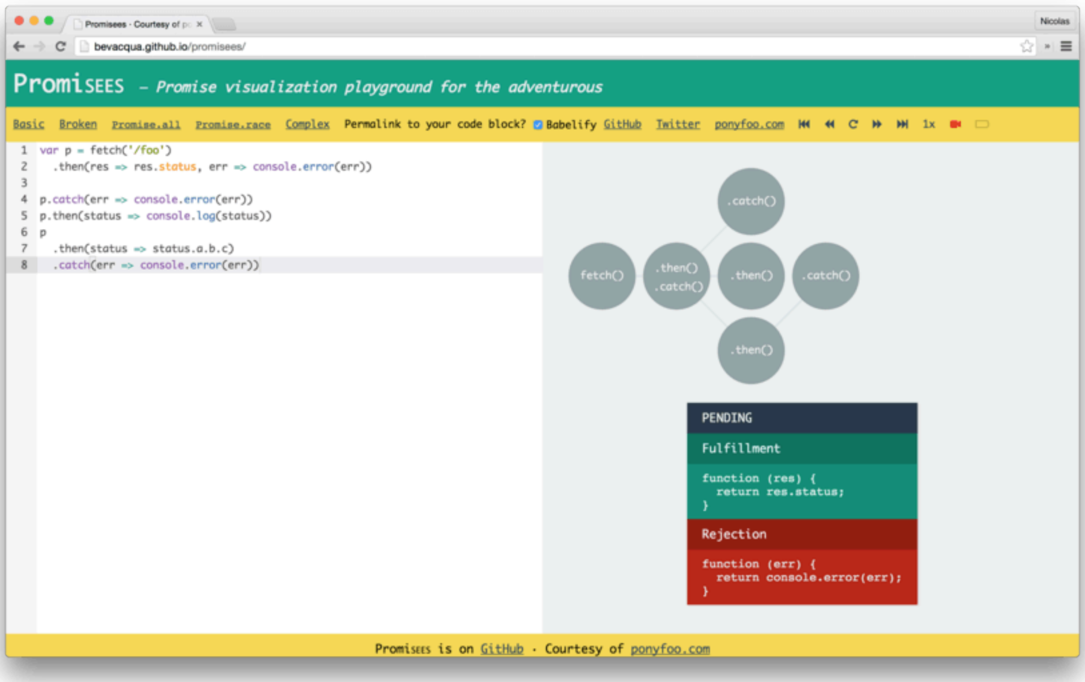
(#aa-a-promise-based-api) A `Promise` Based API

The future of web browser API implementations is `Promise`-heavy. The `fetch` API, for example, sprinkles sweet `Promise`-based sugar on top of `XMLHttpRequest`. ServiceWorker makes occasional use of `fetch`, but there’ s also worker registration, caching, and message passing, all of which are Promise-based.

I wrote a tutorial that may help you get started with Promises (https://ponyfoo.com/articles/es6-promises-in-depth)

There’ s also an ES6 overview in bullet points (https://ponyfoo.com/articles/es6) on my blog

There’ s also this tool I wrote that helps you visualize promises (http://bevacqua.github.io/promisees/) if you’ re more of a visual learner



```
var p = fetch('/foo')
  .then(res => res.status, err => console.error(err))

p.catch(err => console.error(err))
p.then(status => console.log(status))
p
  .then(status => status.a.b.c)
  .catch(err => console.error(err))
```

PENDING

Fulfillment

```
function (res) {
  return res.status;
}
```

Rejection

```
function (err) {
  return console.error(err);
}
```

Promises - Courtesy of p
bevacqua.github.io/promisees/
Nicolas
PromiSEES – Promise visualization playground for the adventurous
Basic
Broken
Promise.all
Promise.race
Complex
Permalink to your code block?
Babelify
GitHub
Twitter
ponyfoo.com
1x
.catch()
fetch()
.then() .catch()
.then()
.catch()
.then()
Promisees is on GitHub · Courtesy of ponyfoo.com

Whether or not you are a fan of promises, they are here to stay, so you better get used to them.

(#aa-registering-your-first-serviceworker) Registering Your First ServiceWorker

I worked together with Chris on the simplest possible practical demonstration of how to use ServiceWorker. He implemented a simple website (static HTML, CSS, JavaScript, and images) and asked me to add offline support. I felt like that would be a great opportunity to display how easy and unobtrusive it is to add offline capabilities to an existing website.

If you’ d like to skip to the end, take a look at the this commit to the demo site (https://github.com/chriscoyier/Simple-Offline-Site/commit/4928bfe074ec39ce72c27bb9078b8ed50672e938) on GitHub.

The first step is to register the ServiceWorker. Instead of blindly attempting the registration, we feature-detect that ServiceWorker is available.

JavaScript

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

}
```

The following piece of code demonstrates how we would install a ServiceWorker. The JavaScript resource passed to `.register` will be executed in the context of a ServiceWorker. Note how registration returns a `Promise` so that you can track whether or not the ServiceWorker registration was successful. I preceded logging statements with `CLIENT`: to make it visually easier for me to figure out whether a logging statement was coming from a web page or the ServiceWorker script.

JavaScript

```javascript
// ServiceWorker is a progressive technology. Ignore unsupported browsers
if ('serviceWorker' in navigator) {
  console.log('CLIENT: service worker registration in progress.');
  navigator.serviceWorker.register('/service-worker.js').then(function() {
    console.log('CLIENT: service worker registration complete.');
  }, function() {
    console.log('CLIENT: service worker registration failure.');
  });
} else {
  console.log('CLIENT: service worker is not supported.');
}
```

The endpoint to the `service-worker.js` file is quite important. If the script were served from, say, `/js/service-worker.js` then the ServiceWorker would only be able to intercept requests in the `/js/` context, but it’d be blind to resources like `/other`. This is typically an issue because you usually scope your JavaScript files in a `/js/`, `/public/`, `/assets/`, or similar “directory”, whereas you’ll want to serve the ServiceWorker script from the domain root in most cases.

That was, in fact, the only necessary change to your web application code, provided that you had already implemented HTTPS. At this point, supporting browsers will issue a request for `/service-worker.js` and attempt to install the worker.

How should you structure the `service-worker.js` file, then?

(#aa-putting-together-a-serviceworker) Putting Together A ServiceWorker

ServiceWorker is event-driven and **your code should aim to be stateless**. That’s because when a ServiceWorker isn’t being used it’s shut down, losing all state. You have no control over that, so it’s best to avoid any long-term dependence on the in-memory state.

Below, I listed the most notable events you’ll have to handle in a ServiceWorker.

The `install` event fires when a ServiceWorker is first fetched. This is your chance to prime the ServiceWorker cache with the fundamental resources that should be available even while users are offline.

The `fetch` event fires whenever a request originates from your ServiceWorker scope, and you’ll get a chance to intercept the request and respond immediately, without going to the network.

The `activate` event fires after a successful installation. You can use it to phase out older versions of the worker. We’ll look at a basic example where we deleted stale cache entries.

Let’ s go over each event and look at examples of how they could be handled.

(#aa-installing-your-serviceworker) Installing Your ServiceWorker

A version number is useful when updating the worker logic, allowing you to remove outdated cache entries during the activation step (https://ponyfoo.com/articles/getting-started-with-serviceworker?verify=0b80097456ce81cfcd2ffb5157eb395a#phasing-out-older-serviceworker-versions) , as we’ ll see a bit later. We’ ll use the following version number as a prefix when creating cache stores.

```
JavaScript
var version = 'v1::';
```

You can use `addEventListener` to register an event handler for the `install` event. Using `event.waitUntil` blocks the installation process on the provided `p` promise. If the promise is rejected because, for instance, one of the resources failed to be downloaded, the service worker won’ t be installed. Here, you can leverage the promise returned from opening a cache with `caches.open(name)` and then mapping that into `cache.addAll(resources)`, which downloads and stores responses for the provided resources.

```
JavaScript
self.addEventListener("install", function(event) {
  console.log('WORKER: install event in progress.');
  event.waitUntil(
    /* The caches built-in is a promise-based API that helps you cache responses,
       as well as finding and deleting them.
    */
    caches
      /* You can open a cache by name, and this method returns a promise. We use
         a versioned cache name here so that we can remove old cache entries in
         one fell swoop later, when phasing out an older service worker.
      */
      .open(version + 'fundamentals')
      .then(function(cache) {
        /* After the cache is opened, we can fill it with the offline fundamentals.
           The method below will add all resources we've indicated to the cache,
           after making HTTP requests for each of them.
        */
        return cache.addAll([
          '/',
          '/css/global.css',
          '/js/global.js'
        ]);
      })
      .then(function() {
        console.log('WORKER: install completed');
      })
  );
});
```

Once the install step succeeds, the `activate` event fires. This helps us phase out an older ServiceWorker (#phasing-out-older-serviceworker-versions) , and we’ ll look at it later. For now, let’ s focus on the `fetch` event, which is a bit more interesting.

(#aa-intercepting-fetch-requests) Intercepting Fetch Requests

The `fetch` event fires whenever a page controlled by this service worker requests a resource. This isn’t limited to `fetch` or even `XMLHttpRequest`. Instead, it comprehends even the request for the HTML page on first load, as well as JS and CSS resources, fonts, any images, etc. Note also that requests made against other origins will also be caught by the `fetch` handler of the ServiceWorker. For instance, requests made against `i.imgur.com` – the CDN for a popular image hosting site – would also be caught by our service worker as long as the request originated on one of the clients (e.g browser tabs) controlled by the worker.

Just like `install`, we can block the `fetch` event by passing a promise to `event.respondWith(p)`, and when the promise fulfills the worker will respond with that instead of the default action of going to the network. We can use `caches.match` to look for cached responses, and return those responses instead of going to the network.

As described in the comments, here we’re using an “eventually fresh” caching pattern where we return whatever is stored on the cache but always try to fetch a resource again from the network regardless, to keep the cache updated. If the response we served to the user is stale, they’ll get a fresh response the next time they request the resource. If the network request fails, it’ll try to recover by attempting to serve a hardcoded `Response`.

```javascript
self.addEventListener("fetch", function(event) {
  console.log('WORKER: fetch event in progress.');

  /* We should only cache GET requests, and deal with the rest of method in the
     client-side, by handling failed POST,PUT,PATCH,etc. requests.
  */
  if (event.request.method !== 'GET') {
    /* If we don't block the event as shown below, then the request will go to
       the network as usual.
    */
    console.log('WORKER: fetch event ignored.', event.request.method, event.request.url);
    return;
  }
  /* Similar to event.waitUntil in that it blocks the fetch event on a promise.
     Fulfillment result will be used as the response, and rejection will end in a
     HTTP response indicating failure.
  */
  event.respondWith(
    caches
      /* This method returns a promise that resolves to a cache entry matching
         the request. Once the promise is settled, we can then provide a response
         to the fetch request.
      */
      .match(event.request)
      .then(function(cached) {
        /* Even if the response is in our cache, we go to the network as well.
           This pattern is known for producing "eventually fresh" responses,
           where we return cached responses immediately, and meanwhile pull
           a network response and store that in the cache.
           Read more:
           https://ponyfoo.com/articles/progressive-networking-serviceworker
        */
        var networked = fetch(event.request)
          // We handle the network request with success and failure scenarios.
          .then(fetchedFromNetwork, unableToResolve)
          // We should catch errors on the fetchedFromNetwork handler as well.
          .catch(unableToResolve);

        /* We return the cached response immediately if there is one, and fall
           back to waiting on the network as usual.
        */
```

```
          console.log('WORKER: fetch event', cached ? '(cached)' : '(network)', event.request.url);
          return cached || networked;

          function fetchedFromNetwork(response) {
            /* We copy the response before replying to the network request.
               This is the response that will be stored on the ServiceWorker cache.
            */
            var cacheCopy = response.clone();

            console.log('WORKER: fetch response from network.', event.request.url);

            caches
              // We open a cache to store the response for this request.
              .open(version + 'pages')
              .then(function add(cache) {
                /* We store the response for this request. It'll later become
                   available to caches.match(event.request) calls, when looking
                   for cached responses.
                */
                cache.put(event.request, cacheCopy);
              })
              .then(function() {
                console.log('WORKER: fetch response stored in cache.', event.request.url);
              });

            // Return the response so that the promise is settled in fulfillment.
            return response;
          }

          /* When this method is called, it means we were unable to produce a response
             from either the cache or the network. This is our opportunity to produce
             a meaningful response even when all else fails. It's the last chance, so
             you probably want to display a "Service Unavailable" view or a generic
             error response.
          */
          function unableToResolve () {
            /* There's a couple of things we can do here.
               - Test the Accept header and then return one of the `offlineFundamentals`
                 e.g: `return caches.match('/some/cached/image.png')`
               - You should also consider the origin. It's easier to decide what
                 "unavailable" means for requests against your origins than for requests
                 against a third party, such as an ad provider
               - Generate a Response programmaticaly, as shown below, and return that
            */

            console.log('WORKER: fetch request failed in both cache and network.');

            /* Here we're creating a response programmatically. The first parameter is the
               response body, and the second one defines the options for the response.
            */
            return new Response('<h1>Service Unavailable</h1>', {
              status: 503,
              statusText: 'Service Unavailable',
              headers: new Headers({
                'Content-Type': 'text/html'
              })
            });
          }
        })
    );
  });
```

There’ s several more strategies, some of which I discuss in an article about ServiceWorker strategies on my blog (https://ponyfoo.com/articles/serviceworker-revolution) .

As promised, let’ s look at the code you can use to phase out older versions of your ServiceWorker script.

(#aa-phasing-out-older-serviceworker-versions) Phasing Out Older ServiceWorker Versions

The `activate` event fires after a service worker has been successfully installed. It is most useful when phasing out an older version of a service worker, as at this point you know that the new worker was installed correctly. In this example, we delete old caches that don’ t match the `version` for the worker we just finished installing.

JavaScript

```
self.addEventListener("activate", function(event) {
  /* Just like with the install event, event.waitUntil blocks activate on a promise.
     Activation will fail unless the promise is fulfilled.
  */
  console.log('WORKER: activate event in progress.');

  event.waitUntil(
    caches
      /* This method returns a promise which will resolve to an array of available
         cache keys.
      */
      .keys()
      .then(function (keys) {
        // We return a promise that settles when all outdated caches are deleted.
        return Promise.all(
          keys
            .filter(function (key) {
              // Filter by keys that don't start with the latest version prefix.
              return !key.startsWith(version);
            })
            .map(function (key) {
              /* Return a promise that's fulfilled
                 when each outdated cache is deleted.
              */
              return caches.delete(key);
            })
        );
      })
      .then(function() {
        console.log('WORKER: activate completed.');
      })
  );
});
```

Hey! Reminder: there is a Simple Offline Site repo (https://github.com/chriscoyier/Simple-Offline-Site) we’ ve built for this. You can see the entire thing as a CodePen Project (https://codepen.io/chriscoyier/project/editor/ZPvNRA/) , and it’ s even a full on demo website (https://simpleoffline.website/) .

%d