High-performance service worker loading

By Jeff Posnick
Web DevRel @ Google

Adding a service worker to your web app can offer significant performance benefits, going beyond what's possible even when following all the traditional browser caching best practices. But there are a few best practices to follow in order to optimize your load times. The following tips will ensure you're getting the best performance out of your service worker implementation.

First, what are navigation requests?

Navigation requests are (tersely) defined in the Fetch specification as: *A navigation request is a request whose destination is "*`document`*".* While technically correct, that definition lacks nuance, and it undersells the importance of navigations on your web app's performance. Colloquially, a navigation request takes place whenever you enter a URL in your browser's location bar, interact with `window.location`, or visit a link from one web page to another. Putting an `<iframe>` on a page will also lead to a navigation request for the `<iframe>`'s `src`.

Note: Single page applications, relying on the History API and in-place DOM modifications, tend to avoid navigation requests when switching from view to view. But the initial request in a browser's session for a single page app is still a navigation.

While your web app might make many other subresource requests in order to display all its contents—for elements like scripts, images, or styles—it's the HTML in the navigation response that's responsible for kicking off all the other requests. Any delays in the response for the initial navigation request will be painfully obvious to your users, as they're left staring at a blank screen for an indeterminate period of time.

Note: HTTP/2 server push adds a wrinkle here, as it allows subresource responses to be returned without additional latency, alongside the navigation response. But any delays in establishing the connection to the remote server will also lead to delays the data being pushed down to the client.

Traditional caching best practices, the kind that rely on HTTP `Cache-Control` headers and not a service worker, require going to the network each navigation, to ensure that all of the subresource URLs are fresh. The holy grail for web performance is to get all the benefits of aggressively cached subresources, *without* requiring a navigation request that's dependent on the network. With a properly configured service worker tailored to your site's specific architecture, that's now possible.

For best performance, bypass the network for navigations

The biggest impact of adding a service worker to your web application comes from responding to navigation requests without waiting on the network. The best-case-scenario for connecting to a web server is likely to take orders of magnitude longer than it would take to read locally cached data. In scenarios where a client's connection is less than ideal—basically, anything on a mobile network—the amount of time it takes to get back the first byte of data from the network can easily outweigh the total time it would take to render the full HTML.

Choosing the right cache-first service worker implementation largely depends on your site's architecture.

Streaming composite responses

If your HTML can naturally be split into smaller pieces, with a static header and footer along with a middle portion that varies depending on the request URL, then handling navigations using a streamed response is ideal. You can compose the response out of individual pieces that are each cached separately. Using streams ensures that the initial portion of the response is exposed to the client as soon as possible, giving it a head start on parsing the HTML and making any additional subresource requests.

The "Stream Your Way to Immediate Responses" article provides a basic overview of this approach, but for real-world examples and demos, Jake Archibald's "2016 - the year of web streams" is the definitive guide.

Note: For some web apps, there's no avoiding the network when responding to a navigation request. Maybe the HTML for each URL on your site depends on data from a content management system, or maybe your site uses varying layouts and doesn't fit into a generic, application shell structure. Service workers still open the door for improvements over the *status quo* for loading your HTML. Using streams, you can respond to navigation requests immediately with a common, cached chunk of HTML—perhaps your site's full `<head>` and some initial `<body>` elements—while still loading the rest of the HTML, specific to a given URL, from the network.

Caching static HTML

If you've got a simple web app that relies entirely on a set of static HTML documents, then you're in luck: your path to avoiding the network is straightforward. You need a service worker that responds to navigations with previously cached HTML, and that also includes non-blocking logic for keeping that HTML up-to-date as your site evolves.

One approach is to use a service worker `fetch` handler that implements a stale-while-revalidate policy for navigation requests, like so:

```
self.addEventListener('fetch', event => {
  if (event.request.mode === 'navigate') {
    // See /web/fundamentals/getting-started/primers/async-functions
    // for an async/await primer.
    event.respondWith(async function() {
      // Optional: Normalize the incoming URL by removing query parameters.
      // Instead of https://example.com/page?key=value,
      // use https://example.com/page when reading and writing to the cache.
      // For static HTML documents, it's unlikely your query parameters will
      // affect the HTML returned. But if you do use query parameters that
      // uniquely determine your HTML, modify this code to retain them.
      const normalizedUrl = new URL(event.request.url);
      normalizedUrl.search = '';

      // Create promises for both the network response,
      // and a copy of the response that can be used in the cache.
      const fetchResponseP = fetch(normalizedUrl);
      const fetchResponseCloneP = fetchResponseP.then(r => r.clone());

      // event.waitUntil() ensures that the service worker is kept alive
      // long enough to complete the cache update.
      event.waitUntil(async function() {
        const cache = await caches.open('my-cache-name');
        await cache.put(normalizedUrl, await fetchResponseCloneP);
      }());

      // Prefer the cached response, falling back to the fetch response.
      return (await caches.match(normalizedUrl)) || fetchResponseP;
    }());
  }
});
```

Another approach is to use a tool like Workbox, which hooks into your web app's build process to generate a service worker that handles caching all of your static resources (not just HTML documents), serving them cache-first, and keeping them up to date.

Using an Application Shell

If you have an existing single page application, then the application shell architecture is straightforward to implement. There's a clear-cut strategy for handling navigation requests without relying on the network: each navigation request, regardless of the specific URL, is fulfilled with a cached copy of a generic "shell" of an HTML document. The shell includes everything needed to bootstrap the single page application, and client-side routing logic can then render the content specific to the request's URL.

Written by hand, the corresponding service worker `fetch` handler would look something like:

```
// Not shown: install and activate handlers to keep app-shell.html
// cached and up to date.
self.addEventListener('fetch', event => {
  if (event.request.mode === 'navigate') {
    // Always respond to navigations with the cached app-shell.html,
    // regardless of the underlying event.request.url value.
    event.respondWith(caches.match('app-shell.html'));
  }
});
```

Workbox can also help here, both by ensuring your `app-shell.html` is cached and kept up to date, as well as providing helpers for responding to navigation requests with the cached shell.

⚠ Performance gotchas

If you can't respond to navigations using cached data, but you need a service worker for other functionality—like providing offline fallback content, or handling push notifications — then you're in an awkward situation. If you don't take specific precautions, you could end up taking a performance hit when you add in your service worker. But by steering clear of these gotchas, you'll be on solid ground.

Never use a "passthrough" fetch handler

If you're using a service worker just for push notifications, you might mistakenly think that the following is either required, or will just be treated as a no-op:

```
// Don't do this!
self.addEventListener('fetch', event => {
  event.respondWith(fetch(event.request));
});
```

This type of "passthrough" fetch handler is insidious, since everything will continue to work in your web application, but you'll end up introducing a small latency hit whenever a network request is made. There's overhead involved in starting up a service worker if it's not already running, and there's also overhead in passing the response from the service worker to the client that made the request.

If your service worker doesn't contain a `fetch` handler at all, some browsers will make note of that and not bother starting up the service worker whenever there's a network request.

Use navigation preload when appropriate

There are scenarios in which you *need* a `fetch` handler to use a caching strategy for certain subresources, but your architecture makes it impossible to respond to navigation requests. Alternatively, you might be okay with using cached data in your navigation response, but you still want to make a network request for fresh data to swap in after the page has loaded.

A feature known as Navigation Preload is relevant for both of those use cases. It can mitigate the delays that a service worker that didn't respond to navigations might otherwise introduce. It can also be used for "out of band" requests for fresh data that could then be used by client-side code after the page has loaded. The "Speed up Service Worker with Navigation Preloads" article has all the details you'd need to configure your service worker accordingly.