Stream Your Way to Immediate Responses

By Jeff Posnick
Web DevRel @ Google

Anyone who's used service workers could tell you that they're asynchronous all the way down. They rely exclusively on event-based interfaces, like `FetchEvent`, and use promises to signal when asynchronous operations are complete.

Asynchronicity is equally important, albeit less visible to the developer, when it comes to responses provided by a service worker's fetch event handler. Streaming responses are the gold standard here: they allow the page that made the original request to start working with the response as soon as the first chunk of data is available, and potentially use parsers that are optimized for streaming to progressively display the content.

When writing your own `fetch` event handler, it's common to just pass the `respondWith()` method a `Response` (or a promise for a `Response`) that you get via `fetch()` or `caches.match()`, and call it a day. The good news is that the `Response`s created by both of those methods are already streamable! The bad news is that "manually" constructed `Response`s aren't streamable, at least until now. That's where the Streams API enters the picture.

Streams?

A stream is a data source that can be created and manipulated incrementally, and provides an interface for reading or writing asynchronous chunks of data, only a subset of which might be available in memory at any given time. For now, we're interested in `ReadableStream`s, which can be used to construct a `Response` object that's passed to `fetchEvent.respondWith()`:

```
self.addEventListener('fetch', event => {
  var stream = new ReadableStream({
    start(controller) {
      if (/* there's more data */) {
        controller.enqueue(/* your data here */);
      } else {
        controller.close();
      }
    });
```

```
  });

  var response = new Response(stream, {
    headers: {'content-type': /* your content-type here */}
  });

  event.respondWith(response);
});
```

The page whose request triggered the **`fetch`** event will get a streaming response back as soon as **`event.respondWith()`** is called, and it will keep reading from that stream as long as the service worker continues **`enqueue()`**ing additional data. The response flowing from the service worker to the page is truly asynchronous, and we have complete control over filling the stream!

Real-world uses

You've probably noticed that the previous example had some placeholder **`/* your data here */`** comments, and was light on actual implementation details. So what would a real-world example look like?

Jake Archibald (not surprisingly!) has a great example of using streams to stitch together an HTML response from multiple cached HTML snippets, along with "live" data streamed via **`fetch()`**—in this case, content for his blog.

The advantage using a streaming response, as Jake explains, is that the browser can parse and render the HTML as it streams in, including the initial bit that's loaded quickly from the cache, without having to wait for the entire blog content fetch to complete. This takes full advantage of the browser's progressive HTML rendering capabilities. Other resources that can also be progressively rendered, like some image and video formats, can also benefit from this approach.

Streams? Or app shells?

The existing best practices around using service workers to power your web apps focus on an App Shell + dynamic content model. That approach relies on aggressively caching the "shell" of your web application—the minimal HTML, JavaScript, and CSS needed to display your structure and layout—and then loading the dynamic content needed for each specific page via a client-side request.

Streams bring with them an alternative to the App Shell model, one in which there's a fuller HTML response streamed to the browser when a user navigates to a new page. The streamed response can make use of cached resources—so it can still provide the initial chunk of HTML quickly, even while offline!—but they end up looking more like traditional, server-rendered response bodies. For example, if your web app is powered by a content management system that server-renders HTML by stitching together partial templates, that model translates directly into using streaming responses, with the templating logic replicated in the service worker instead of your server. As the following video demonstrates, for that use case, the speed advantage that streamed responses offer can be striking:

One important advantage of streaming the entire HTML response, explaining why it's the fastest alternative in the video, is that HTML rendered during the initial navigation request can take full advantage of the browser's streaming HTML parser. Chunks of HTML that are inserted into a document after the page has loaded (as is common in the App Shell model) can't take advantage of this optimization.

So if you're in the planning stages of your service worker implementation, which model should you adopt: streamed responses that are progressively rendered, or a lightweight shell coupled with a client-side request for dynamic content? The answer is, not surprisingly, that it depends: on whether you have an existing implementation that relies on a CMS and partial templates (advantage: stream); on whether you expect single, large HTML payloads that would benefit from progressive rendering (advantage: stream); on whether your web app is best modeled as a single-page application (advantage: App Shell); and on whether you need a model that's currently supported across multiple browsers' stable releases (advantage: App Shell).

We're still in the very early days of service worker-powered streaming responses, and we look forward to seeing the different models mature and especially to seeing more tooling developed to automate common use cases.

Diving deeper into streams

If you're constructing your own readable streams, simply calling `controller.enqueue()` indiscriminately might not be either sufficient or efficient. Jake goes into some detail about how the `start()`, `pull()`, and `cancel()` methods can be used in tandem to create a data stream that's tailored to your use case.

For those who want even more detail, the Streams specification has you covered.

Compatibility

Support for constructing a `Response` object inside a service worker using a `ReadableStream` as its source was added in Chrome 52.

Firefox's service worker implementation does not yet support responses backed by `ReadableStream`s, but there is a relevant tracking bug for Streams API support that you can follow.

Progress on unprefixed Streams API support in Edge, along with overall service worker support, can be tracked at Microsoft's Platform status page.

Last updated July 2, 2018.