Beyond SPAs: alternative architectures for your PWA

By Jeff Posnick
Web DevRel @ Google

Note: Prefer a video to an article? You can watch the presentation on which this was based instead:

Let's talk about... architecture?

I'm going to cover an important, but potentially misunderstood topic: The architecture that you use for your web app, and specifically, how your architectural decisions come into play when you're building a progressive web app.

"Architecture" can sound vague, and it may not be immediately clear why this matters. Well, one way to think about architecture is to ask yourself the following questions: When a user visits a page on my site, what HTML is loaded? And then, what's loaded when they visit another page?

The answers to those questions are not always straightforward, and once you start thinking about progressive web apps, they can get even more complicated. So my goal is to walk you through one possible architecture that I found effective. Throughout this article, I'll label the decisions that I made as being "my approach" to building a progressive web app.

You're free to use my approach when building your own PWA, but at the same time, there are always other valid alternatives. My hope is that seeing how all the pieces fit together will inspire you, and that you will feel empowered to customize this to suit your needs.

Stack Overflow PWA

To accompany this article I built a Stack Overflow PWA. I spend a lot of time reading and contributing to Stack Overflow, and I wanted to build a web app that would make it easy to browse frequently asked questions for a given topic. It's built on top of the public Stack Exchange API. It's open source, and you can learn more by visiting the GitHub project.

Multi-page Apps (MPAs)

Before I get into specifics, let's define some terms and explain pieces of underlying technology. First, I'm going to be covering what I like to call "Multi Page Apps", or "MPAs".

MPA is a fancy name for the traditional architecture used since this beginning of the web. Each time a user navigates to a new URL, the browser progressively renders HTML specific to that page. There's no attempt to preserve the page's state or the content in between navigations. Each time you visit a new page, you're starting fresh.

This is in contrast to the single-page app (SPA) model for building web apps, in which the browser runs JavaScript code to update the existing page when the user visits a new section. Both SPAs and MPAs are equally valid models to use, but for this post, I wanted to explore PWA concepts within the context of a multi-page app.

Reliably fast

You've heard me (and countless others) use the phrase "progressive web app", or PWA. You might already be familiar with some of the background material, elsewhere on this site.

You can think of a PWA as a web app that provides a first-class user experience, and that truly earns a place on the user's home screen. The acronym "**FIRE**", standing for **F**ast,

Integrated, **R**eliable, and **E**ngaging, sums up all the attributes to think about when building a PWA.

In this article, I'm going to focus on a subset of those attributes: **Fast** and **Reliable**.

Fast: While "fast" means different things in different contexts, I'm going to cover the speed benefits of loading as little as possible from the network.

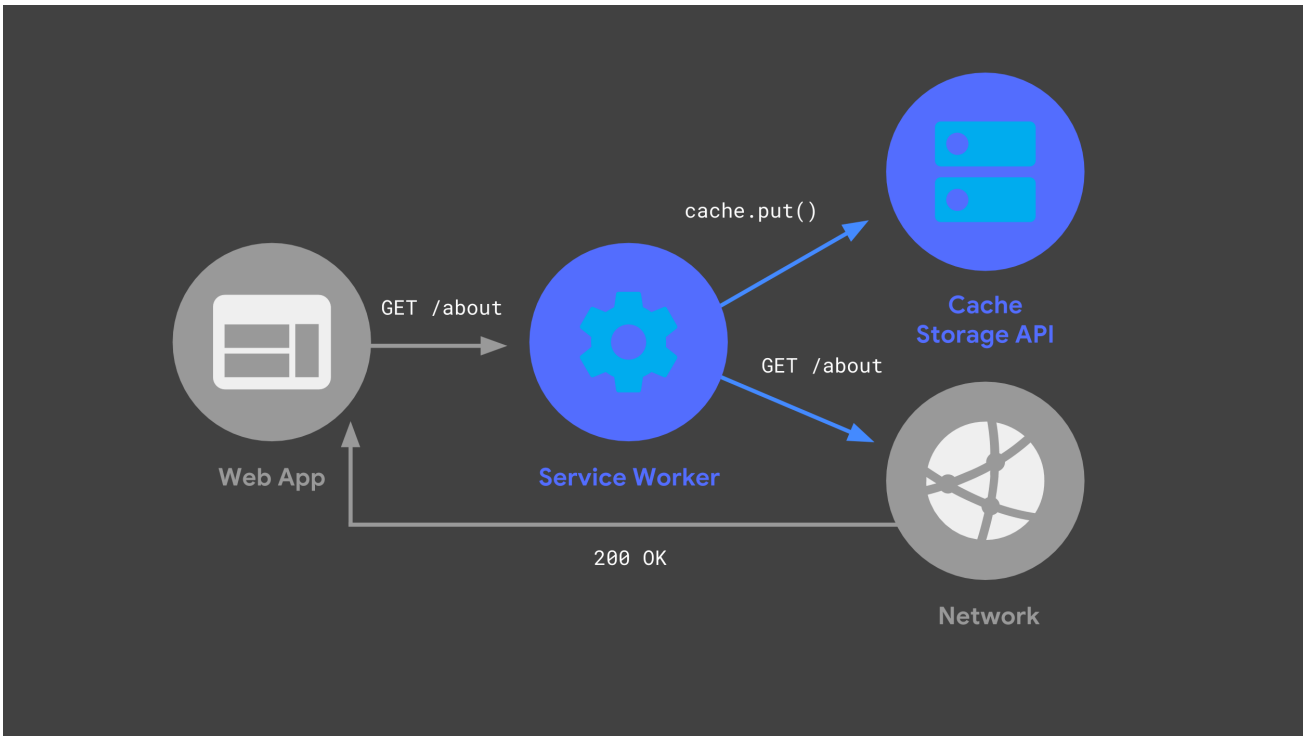
Reliable: But raw speed isn't enough. In order to feel like a PWA, your web app should be reliable. It needs to be resilient enough to always load something, even if it's just a customized error page, regardless of the state of the network.

Reliably fast: And finally, I'm going to rephrase the PWA definition slightly and look at what it means to build something that's reliably fast. It's not good enough to be fast and reliable only when you're on a low-latency network. Being reliably fast means that your web app's speed is consistent, regardless of the underlying network conditions.

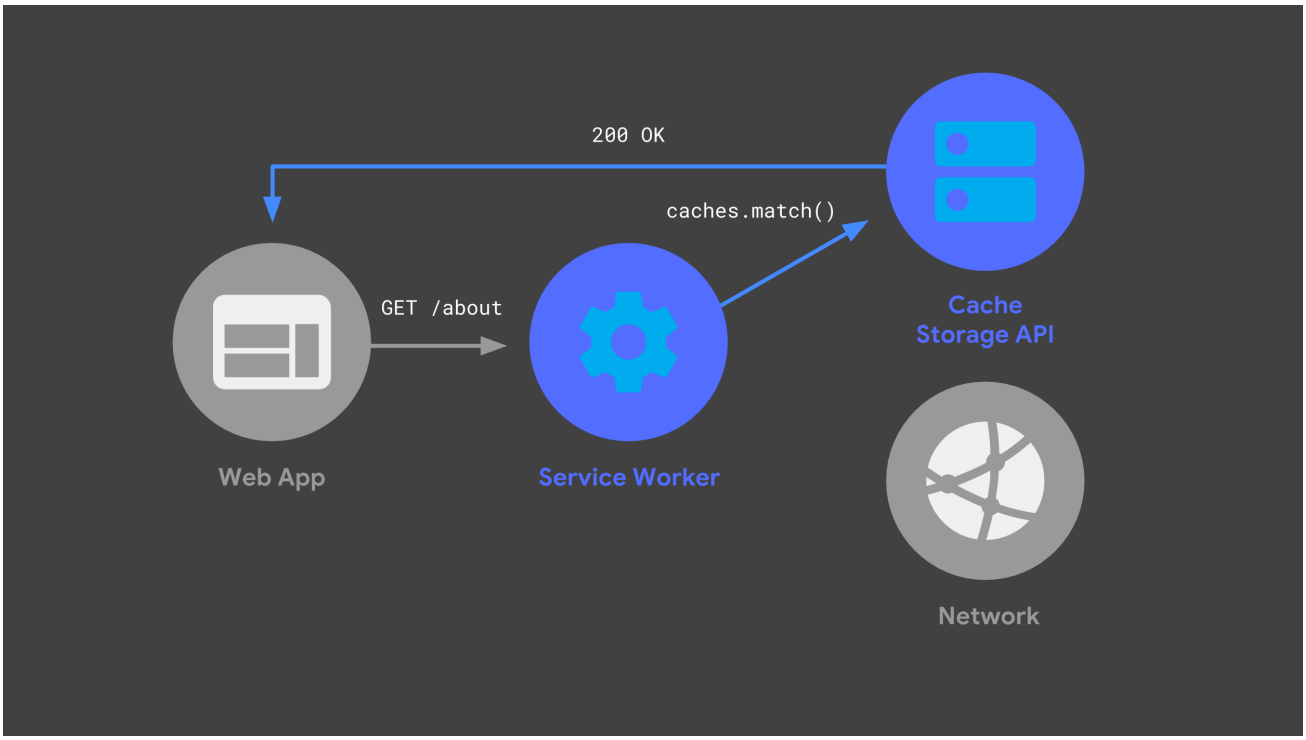
Enabling Technologies: Service Workers + Cache Storage API

PWAs introduce a high bar for speed and resilience. Fortunately, the web platform offers some building blocks to make that type of performance a reality. I'm referring to service workers and the Cache Storage API.

You can build a service worker that listens for incoming requests, passing some on to the network, and storing a copy of the response for future use, via the Cache Storage API.



The next time the web app makes the same request, its service worker can check its caches and just return the previously cached response.



Avoiding the network whenever possible is a crucial part of offering reliably fast performance.

"Isomorphic" JavaScript

One more concept that I want to cover is what's sometimes referred to as "isomorphic", or "universal" JavaScript. Simply put, it's the idea that the same JavaScript code can be shared between different runtime environments. When I built my PWA, I wanted to share JavaScript code between my back-end server, and the service worker.

There are lots of valid approaches to sharing code in this way, but **my approach** was to use ES modules as the definitive source code. I then transpiled and bundled those modules for the server and the service worker using a combination of Babel and Rollup. In my project, files with an `.mjs` file extension is code that lives in an ES module.

The server

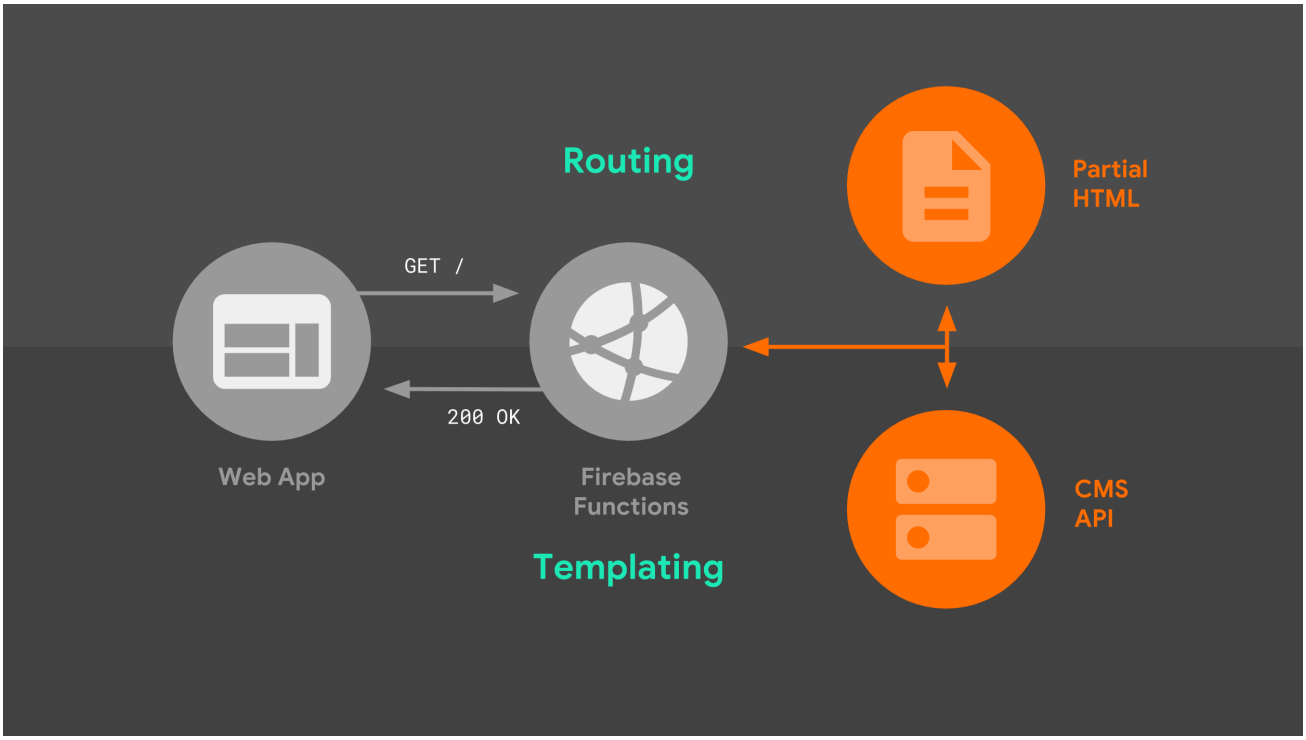
Keeping those concepts and terminology in mind, let's dive into how I actually built my Stack Overflow PWA. I'm going to start by covering our backend server, and explain how that fits

into the overall architecture.

I was looking for a combination of a dynamic backend along with static hosting, and my approach was to use the Firebase platform.

Firebase Cloud Functions will automatically spin up a Node-based environment when there's an incoming request, and integrate with the popular Express HTTP framework, which I was already familiar with. It also offers out-of-the-box hosting for all of my site's static resources. Let's take a look at how the server handles requests.

When a browser makes a navigation request against our server, it goes through the following flow:



The server routes the request based on the URL, and uses templating logic to create a complete HTML document. I use a combination of data from the Stack Exchange API, as well as partial HTML fragments that the server stores locally. Once our service worker knows how to respond, it can start streaming HTML back to our web app.

There are two pieces of this picture worth exploring in more detail: routing, and templating.

Routing

When it comes to routing, my approach was to use the Express framework's native routing syntax. It's flexible enough to match simple URL prefixes, as well as URLs that include

parameters as part of the path. Here, I create a mapping between route names the underlying Express pattern to match against.

```
const routes = new Map([
  ['about', '/about'],
  ['questions', '/questions/:questionId'], ['index', '/'],
]);

export default routes;
```

I can then reference this mapping directly from the server's code. When there's a match for a given Express pattern, the appropriate handler responds with templating logic specific to the matching route.

```
import routes from './lib/routes.mjs';
app.get(routes.get('index'), async (req, res) => {
  // Templating logic.
});
```

Server-side templating

And what does that templating logic look like? Well, I went with an approach that pieced together partial HTML fragments in sequence, one after another. This model lends itself well to streaming.

The server sends back some initial HTML boilerplate immediately, and the browser is able to render that partial page right away. As the server pieces together the rest of the data sources, it streams them to the browser until the document is complete.

To see what I mean, take a look at the Express code for one of our routes:

```
app.get(routes.get('index'), async (req, res) => {
  res.write(headPartial + navbarPartial);
  const tag = req.query.tag || DEFAULT_TAG;
  const data = await requestData(...);
  res.write(templates.index(tag, data.items));
  res.write(footPartial);
  res.end();
});
```

By using the **response** object's `write()` method, and referencing locally stored partial templates, I'm able to start the response stream immediately, without blocking any external

data source. The browser takes this initial HTML and renders a meaningful interface and loading message right away.

The next portion of our page uses data from the Stack Exchange API. Getting that data means that our server needs to make a network request. The web app can't render anything else until it gets a response back and process it, but at least users aren't staring at a blank screen while they wait.

Once the web app's received the response from the Stack Exchange API, it calls a custom templating function to translate the data from the API into its corresponding HTML.

Templating language

Templating can be a surprisingly contentious topic, and what I went with is just one approach among many. You'll want to substitute your own solution, especially if you have legacy ties to an existing templating framework.

What made sense for my use case was to just rely on JavaScript's template literals, with some logic broken out into helper functions. One of the nice things about building an MPA is that you don't have to keep track of state updates and re-render your HTML, so a basic approach that produced static HTML worked for me.

So here's an example of how I'm templating the dynamic HTML portion of my web app's index. As with my routes, the templating logic is stored in an ES module that can be imported into both the server and the service worker.

```
export function index(tag, items) {
  const title = `<h3>Top "${escape(tag)}" Questions</h3>`;
  const form = `<form method="GET">...</form>`;
  const questionCards = items.map((item) => questionCard({
    id: item.question_id,
    title: item.title,
  })).join('');
  const questions = `<div id="questions">${questionCards}</div>`;
  return title + form + questions;
}
```

Warning: Whenever you're taking user-provided input and converting it to HTML, it's crucial that you take care to properly escape potentially dangerous character sequences. If you're using an existing templating solution rather than rolling your own, that might already be taken care of for you.

These template functions are pure JavaScript, and it's useful to break out the logic into smaller, helper functions when appropriate. Here, I pass each of the items returned in the API response into one such function, which creates a standard HTML element with all of the appropriate attributes set.

```
function questionCard({id, title}) {
  return `<a class="card"
             href="/questions/${id}"
             data-cache-url="${questionUrl(id)}">${title}</a>`;
}
```

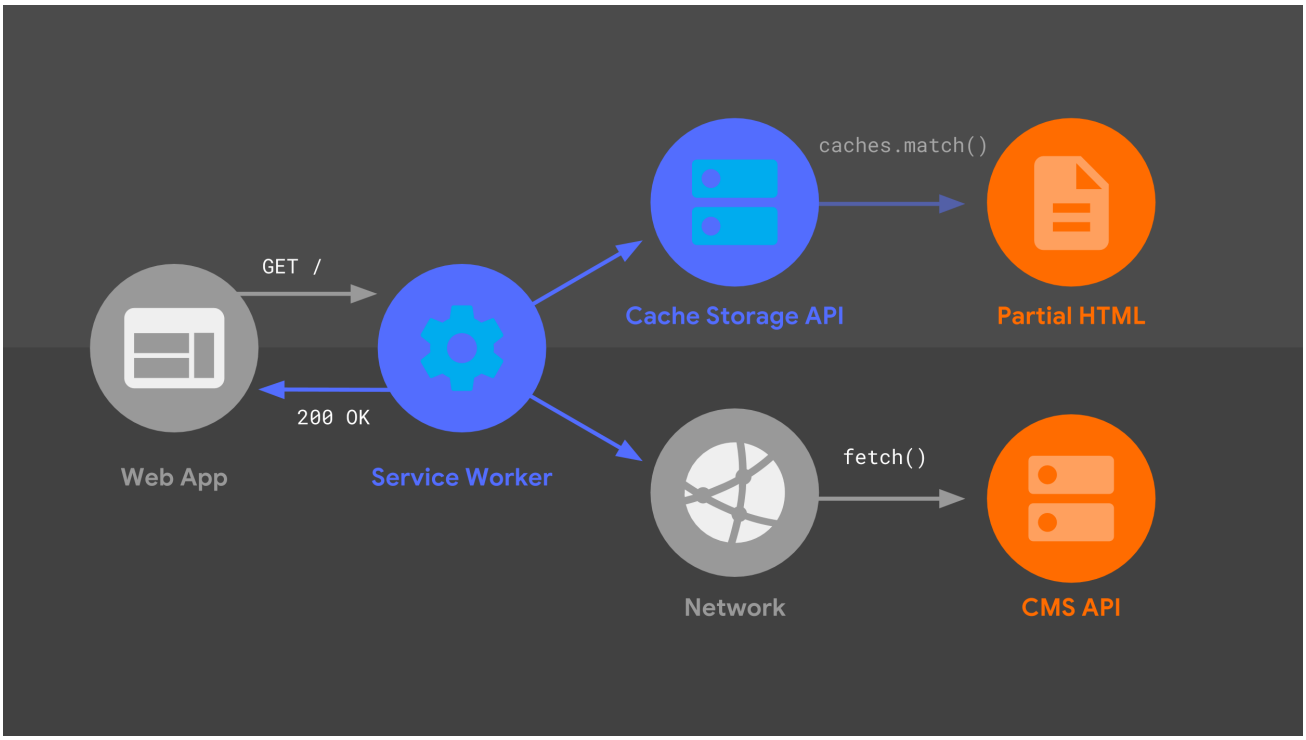
Of particular note is a data attribute that I add to each link, `data-cache-url`, set to the Stack Exchange API URL that I need in order to display the corresponding question. Keep that in mind. I'll revisit it later.

Jumping back to my route handler, once templating is complete, I stream the final portion of my page's HTML to the browser, and end the stream. This is the cue to the browser that the progressive rendering is complete.

```
app.get(routes.get('index'), async (req, res) => {
  res.write(headPartial + navbarPartial);
  const tag = req.query.tag || DEFAULT_TAG;
  const data = await requestData(...);
  res.write(templates.index(tag, data.items));
  res.write(footPartial);
  res.end();
});
```

So that's a brief tour of my server setup. Users who visit my web app for the first time will always get a response from the server, but when a visitor returns to my web app, my service worker will start responding. Let's dive in there.

The service worker



This diagram should look familiar—many of the same pieces I've previously covered are here in a slightly different arrangement. Let's walk through the request flow, taking the service worker into account.

Our service worker handles an incoming navigation request for a given URL, and just like my server did, it uses a combination of routing and templating logic to figure out how to respond.

The approach is the same as before, but with different low-level primitives, like `fetch()` and the Cache Storage API. I use those data sources to construct the HTML response, which the service worker passes back to the web app.

Workbox

Rather than starting from scratch with low-level primitives, I'm going to build my service worker on top of a set of high-level libraries called Workbox. It provides a solid foundation for any service worker's caching, routing, and response generation logic.

Routing

Just as with my server-side code, my service worker needs to know how to match an incoming request with the appropriate response logic.

My approach was to translate each Express route into a corresponding regular expression, making use of a helpful library called `regexparam`. Once that translation is performed, I can take advantage of Workbox's built-in support for regular expression routing.

After importing the module that has the regular expressions, I register each regular expression with Workbox's router. Inside each route I'm able to provide custom templating logic to generate a response. Templating in the service worker is a bit more involved than it was in my backend server, but Workbox helps with a lot of the heavy lifting.

```
import regExpRoutes from './regexp-routes.mjs';

workbox.routing.registerRoute(regExpRoutes.get('index'),
  // Templating logic.
);
```

Static asset caching

One key part of the templating story is making sure that my partial HTML templates are locally available via the Cache Storage API, and are kept up to date when I deploy changes to the web app. Cache maintenance can be error prone when done by hand, so I turn to Workbox to handle precaching as part of my build process.

I tell Workbox which URLs to precache using a configuration file, pointing to the directory that contains all of my local assets along with a set of patterns to match. This file is automatically read by the Workbox's CLI, which is run each time I rebuild the site.

```
module.exports = {
  globDirectory: 'build',
  globPatterns: ['**/*.{html,js,svg}'],
  // Other options...
};
```

Workbox takes a snapshot of each file's contents, and automatically injects that list of URLs and revisions into my final service worker file. Workbox now has everything it needs to make the precached files always available, and kept up to date. The result is a `service-worker.js` file that contains something similar to the following:

```
workbox.precaching.precacheAndRoute([
  {
    url: 'partials/about.html',
    revision: '518747aad9d7e'
  }, {
    url: 'partials/foot.html',
```

```
    revision: '69bf746a9ecc6'
  },
  // etc.
]);
```

For folks who use a more complex build process, Workbox has both a webpack plugin and a generic node module, in addition to its command line interface.

Streaming

Next, I want the service worker to stream that precached partial HTML back to the web app immediately. This is a crucial part of being "reliably fast"—I always get something meaningful on the screen right away. Fortunately, using the Streams API within our service worker makes that possible.

Now, you might have heard about the Streams API before. My colleague Jake Archibald has been singing its praises for years. He made the bold prediction that 2016 would be the year of web streams. And the Streams API is just as awesome today as it was two years ago, but with a crucial difference.

While only Chrome supported Streams back then, the Streams API is more widely supported now. The overall story is positive, and with appropriate fallback code, there's nothing stopping you from using streams in your service worker today.

Well... there might be one thing stopping you, and that's wrapping your head around how the Streams API actually works. It exposes a very powerful set of primitives, and developers who are comfortable using it can create complex data flows, like the following:

```
const stream = new ReadableStream({
  pull(controller) {
    return sources[0].then((r) => r.read())
    .then((result) => {
      if (result.done) {
        sources.shift();
        if (sources.length === 0) return controller.close();
        return this.pull(controller);
      } else {
        controller.enqueue(result.value);
      }
    })
  }
});
```

But understanding the full implications of this code might not be for everyone. Rather than parse through this logic, let's talk about my approach to service worker streaming.

I'm using a brand new, high-level wrapper, `workbox-streams`. With it, I can pass it in a mix of streaming sources, both from caches and runtime data that might come from the network. Workbox takes care of coordinating the individual sources and stitching them together into a single, streaming response.

Additionally, Workbox automatically detects whether the Streams API is supported, and when it's not, it creates an equivalent, non-streaming response. This means that you don't have to worry about writing fallbacks, as streams inch closer to 100% browser support.

Runtime caching

Let's check out how my service worker deals with runtime data, from the Stack Exchange API. I'm making use of Workbox's built-in support for a stale-while-revalidate caching strategy, along with expiration to ensure that the web app's storage doesn't grow unbounded.

I set up two strategies in Workbox to handle the different sources that will make up the streaming response. In a few function calls and configuration, Workbox lets us do what would otherwise take hundreds of lines of handwritten code.

```
const cacheStrategy = workbox.strategies.cacheFirst({
  cacheName: workbox.core.cacheNames.precache,
});

const apiStrategy = workbox.strategies.staleWhileRevalidate({
  cacheName: API_CACHE_NAME,
  plugins: [
    new workbox.expiration.Plugin({maxEntries: 50}),
  ],
});
```

The first strategy reads data that's been precached, like our partial HTML templates.

The other strategy implements the stale-while-revalidate caching logic, along with least-recently-used cache expiration once we reach 50 entries.

Now that I have those strategies in place, all that's left is to tell Workbox how to use them to construct a complete, streaming response. I pass in an array of sources as functions, and each of those functions will be executed immediately. Workbox takes the result from each

source and streams it to the web app, in sequence, only delaying if the next function in the array hasn't completed yet.

```
workbox.streams.strategy([
  () => cacheStrategy.makeRequest({request: '/head.html'}),
  () => cacheStrategy.makeRequest({request: '/navbar.html'}),
  async ({event, url}) => {
    const tag = url.searchParams.get('tag') || DEFAULT_TAG;
    const listResponse = await apiStrategy.makeRequest(...);
    const data = await listResponse.json();
    return templates.index(tag, data.items);
  },
  () => cacheStrategy.makeRequest({request: '/foot.html'}),
]);
```

The first two sources are precached partial templates read directly from the Cache Storage API, so they'll always be available immediately. This ensures that our service worker implementation will be reliably fast in responding to requests, just like my server-side code.

Our next source function fetches data from the Stack Exchange API, and processes the response into the HTML that the web app expects.

The stale-while-revalidate strategy means that if I have a previously cached response for this API call, I'll be able to stream it to the page immediately, while updating the cache entry "in the background" for the next time it's requested.

Finally, I stream a cached copy of my footer and close the final HTML tags, to complete the response.

Sharing code keeps things in sync

You'll notice that certain bits of the service worker code look familiar. The partial HTML and templating logic used by my service worker is identical to what my server-side handler uses. This code sharing ensures that users get a consistent experience, whether they're visiting my web app for the first time or returning to a page rendered by the service worker. That's the beauty of isomorphic JavaScript.

Dynamic, progressive enhancements

I've walked through both the server and service worker for my PWA, but there's one last bit of logic to cover: there's a small amount of JavaScript that runs on each of my pages, after they're fully streamed in.

This code progressively enhances the user experience, but isn't crucial—the web app will still work if it's not run.

Page metadata

My app uses client-side JavaScipt for to update a page's metadata based on the API response. Because I use the same initial bit of cached HTML for each page, the web app ends up with generic tags in my document's head. But through coordination between my templating and client-side code, I can update the window's title using page-specific metadata.

As part of the templating code, my approach is to include a script tag containing the properly escaped string.

```
const metadataScript = `<script>
  self._title = '${escape(item.title)}';
</script>`;
```

Then, once my page has loaded, I read that string and update the document title.

```
if (self._title) {
  document.title = unescape(self._title);
}
```

If there are other pieces of page-specific metadata you want to update in your own web app, you can follow the same approach.

Offline UX

The other progressive enhancement I've added is used to bring attention to our offline capabilities. I've built a reliable PWA, and I want users to know that when they're offline, they can still load previously visited pages.

First, I use the Cache Storage API to get a list of all the previously cached API requests, and I translate that into a list of URLs.

Remember those special data attributes I talked about, each containing the URL for the API request needed to display a question? I can cross-reference those data attributes against the list of cached URLs, and create an array of all the question links that don't match.

When the browser enters an offline state, I loop through the list of uncached links, and dim out the ones that won't work. Keep in mind that this is just a visual hint to the user about

what they should expect from those pages—I'm not actually disabling the links, or preventing the user from navigating.

```
const apiCache = await caches.open(API_CACHE_NAME);
const cachedRequests = await apiCache.keys();
const cachedUrls = cachedRequests.map((request) => request.url);

const cards = document.querySelectorAll('.card');
const uncachedCards = [...cards].filter((card) => {
  return !cachedUrls.includes(card.dataset.cacheUrl);
});

const offlineHandler = () => {
  for (const uncachedCard of uncachedCards) {
    uncachedCard.style.opacity = '0.3';
  }
};

const onlineHandler = () => {
  for (const uncachedCard of uncachedCards) {
    uncachedCard.style.opacity = '1.0';
  }
};

window.addEventListener('online', onlineHandler);
window.addEventListener('offline', offlineHandler);
```

Common pitfalls

I've now gone through a tour of my approach to building a multi-page PWA. There are many factors that you'll have to consider when coming up with your own approach, and you may end up making different choices than I did. That flexibility is one of the great things about building for the web.

There are a few common pitfalls that you may encounter when making your own architectural decisions, and I want to save you some pain.

Don't cache full HTML

I recommend against storing complete HTML documents in your cache. For one thing, it's a waste of space. If your web app uses the same basic HTML structure for each of its pages, you'll end up storing copies of the same markup again and again.

More importantly, if you deploy a change to your site's shared HTML structure, every one of those previously cached pages is still stuck with your old layout. Imagine the frustration of a returning visitor seeing a mix of old and new pages.

Server / service worker drift

The other pitfall to avoid involves your server and service worker getting out of sync. *My approach** was to use isomorphic JavaScript, so that the same code was run in both places. Depending on your existing server architecture, that's not always possible.

Whatever architectural decisions you make, you should have some strategy for running the equivalent routing and templating code in your server and your service worker.

Worst case scenarios

Inconsistent layout / design

What happens when you ignore those pitfalls? Well, all sorts of failures are possible, but the worst case scenario is that a returning user visits a cached page with a very stale layout—perhaps one with out of date header text, or that uses CSS class names that are no longer valid.

Worst case scenario: Broken routing

Alternatively, a user might come across a URL that's handled by your server, but not your service worker. An site full of zombie layouts and dead ends is not a reliable PWA.

Tips for success

But you're not in this alone! The following tips can help you avoid those pitfalls:

Use templating and routing libraries that have multi-language implementations

Try to use templating and routing libraries that have JavaScript implementations. Now, I know that not every developer out there has the luxury of migrating off your current web server and templating language.

But a number of popular templating and routing frameworks have implementations in multiple languages. If you can find one that works with JavaScript as well as your current server's language, you're one step closer to keeping your service worker and server in sync.

Prefer sequential, rather than nested, templates

Next, I recommend using a series of sequential templates that can be streamed in one after another. It's okay if later portions of your page use more complicated templating logic, as long as you can stream in the initial part of your HTML as quickly as possible.

Cache both static and dynamic content in your service worker

For best performance, you should precache all of your site's critical static resources. You should also set up runtime caching logic to handle dynamic content, like API requests. Using Workbox means that you can build on top of well-tested, production-ready strategies instead of implementing it all from scratch.

Only block on the network when absolutely necessary

And related to that, you should only block on the network when it's not possible to stream a response from the cache. Displaying a cached API response immediately can often lead to a better user experience than waiting for fresh data.

Resources

- SO PWA live demo
- SO PWA GitHub project
- Workbox
- Streams API specification

Last updated July 2, 2018.