Instant Loading Web Apps with an Application Shell Architecture

By Addy Osmani

Eng Manager, Web Developer Relations

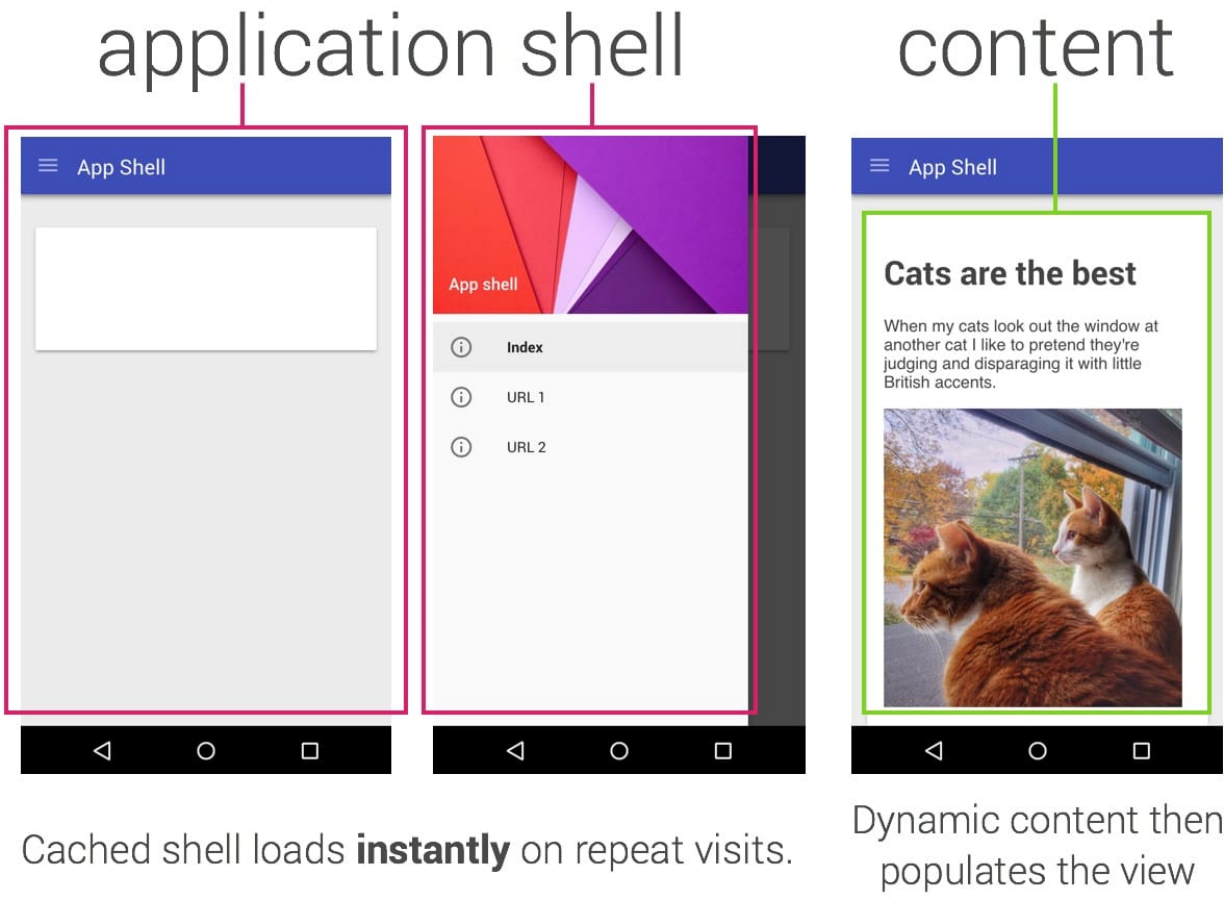
By Matt Gaunt

Matt is a contributor to Web**Fundamentals**

An **application shell** is the minimal HTML, CSS, and JavaScript powering a user interface. The application shell should:

- load fast
- be cached
- dynamically display content

An application shell is the secret to reliably good performance. Think of your app's shell like the bundle of code you'd publish to an app store if you were building a native app. It's the load needed to get off the ground, but might not be the whole story. It keeps your UI local and pulls in content dynamically through an API.



Background

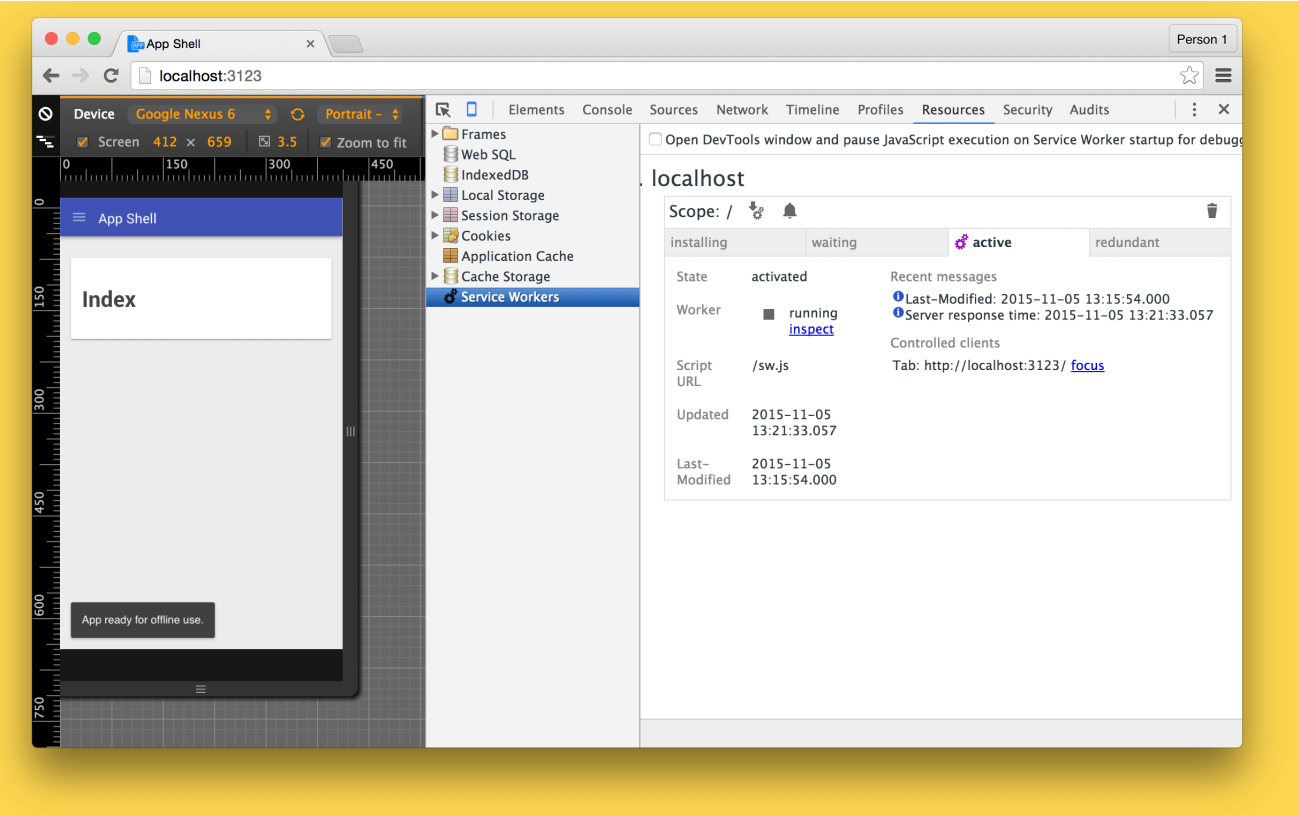
Alex Russell's Progressive Web Apps article describes how a web app can *progressively* change through use and user consent to provide a more native-app-like experience complete with offline support, push notifications and the ability to be added to the home screen. It depends very much on the functionality and performance benefits of service worker and their caching abilities. This allows you to focus on **speed**, giving your web apps the same **instant loading** and regular updates you're used to seeing in native applications.

To take full advantage of these capabilities we need a new way of thinking about websites: the **application shell architecture**.

Let's dive into how to structure your app using a **service worker augmented application shell architecture**. We'll look at both client and server-side rendering and share an end-to-end sample you can try today.

To emphasize the point, the example below shows the first load of an app using this architecture. Notice the 'App is ready for offline use' toast at the bottom of the screen. If an

update to the shell becomes available later, we can inform the user to refresh for the new version.



What are service workers, again?

A service worker is a script that runs in the background, separate from your web page. It responds to events, including network requests made from pages it serves and push notices from your server. A service worker has an intentionally short lifetime. It wakes up when it gets an event and runs only as long as it needs to process it.

Service workers also have a limited set of APIs when compared to JavaScript in a normal browsing context. This is standard for workers on the web. A Service worker can't access the DOM but can access things like the Cache API, and they can make network requests using the Fetch API. The IndexedDB API and postMessage() are also available to use for data persistence and messaging between the service worker and pages it controls. Push events sent from your server can invoke the Notification API to increase user engagement.

A service worker can intercept network requests made from a page (which triggers a fetch event on the service worker) and return a response retrieved from the network, or retrieved from a local cache, or even constructed programmatically. Effectively, it's a programmable proxy in the browser. The neat part is that, regardless of where the response comes from, it looks to the web page as though there were no service worker involvement.

To learn more about service workers in depth, read an Introduction to Service Workers.

Performance benefits

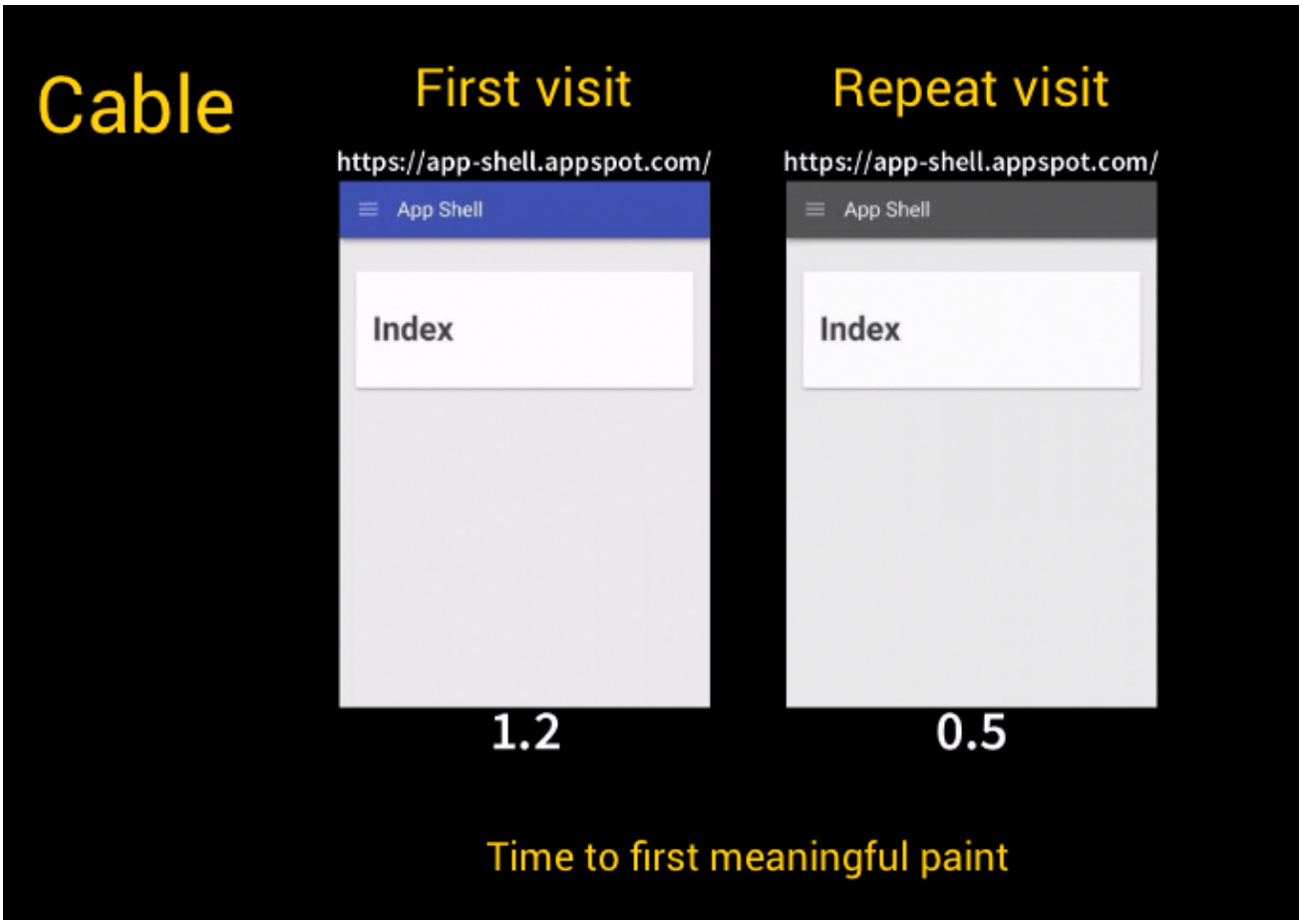
Service workers are powerful for offline caching but they also offer significant performance wins in the form of instant loading for repeat visits to your site or web app. You can cache your application shell so it works offline and populate its content using JavaScript.

On repeat visits, this allows you to get **meaningful pixels** on the screen without the network, even if your content eventually comes from there. Think of it as displaying toolbars and cards **immediately**, then loading the rest of your content **progressively**.

To test this architecture on real devices, we've run our application shell sample on WebPageTest.org and shown the results below.

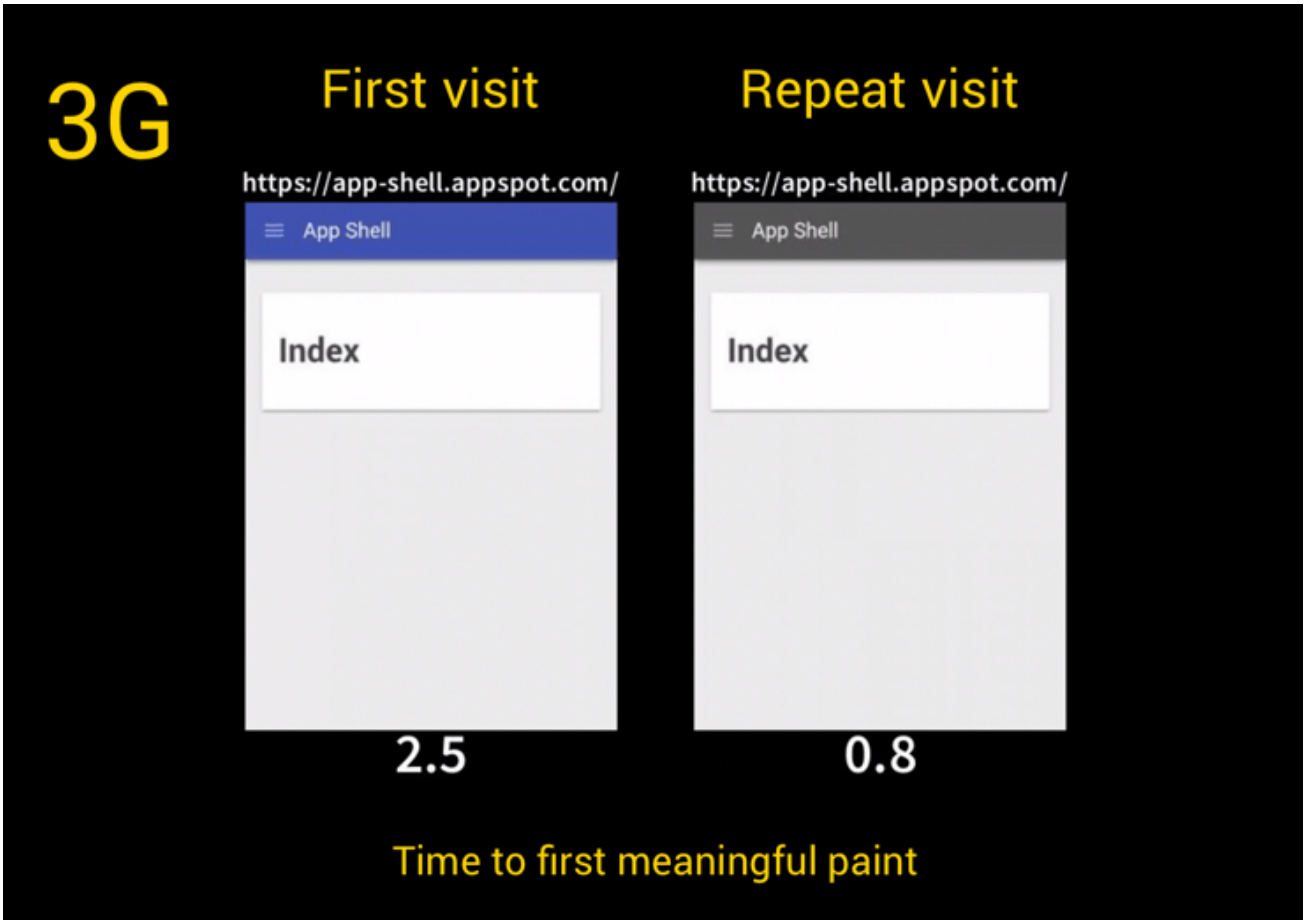
Test 1: Testing on Cable with a Nexus 5 using Chrome Dev

The first view of the app has to fetch all the resources from the network and doesn't achieve a meaningful paint until **1.2 seconds** in. Thanks to service worker caching, our repeat visit achieves meaningful paint and fully finishes loading in **0.5 seconds**.



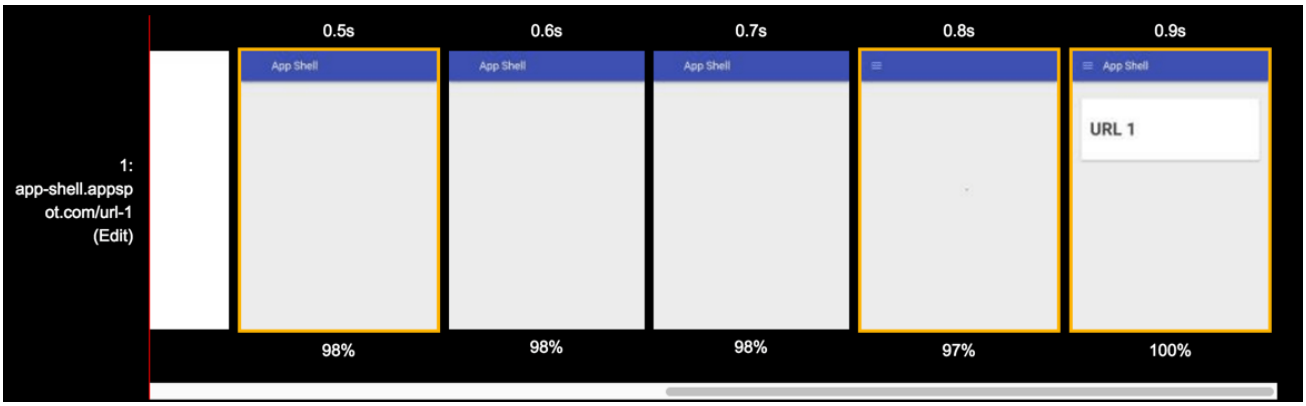
Test 2: Testing on 3G with a Nexus 5 using Chrome Dev

We can also test our sample with a slightly slower 3G connection. This time it takes **2.5 seconds** on first visit for our first meaningful paint. It takes **7.1 seconds** to fully load the page. With service worker caching, our repeat visit achieves meaningful paint and fully finishes loading in **0.8 seconds**.



Other views tell a similar story. Compare the **3 seconds** it takes to achieve first meaningful paint in the application shell:

to the **0.9 seconds** it takes when the same page is loaded from our service worker cache. Over 2 seconds of time is saved for our end users.



Similar and reliable performance wins are possible for your own applications using the application shell architecture.

Does service worker require us to rethink how we structure apps?

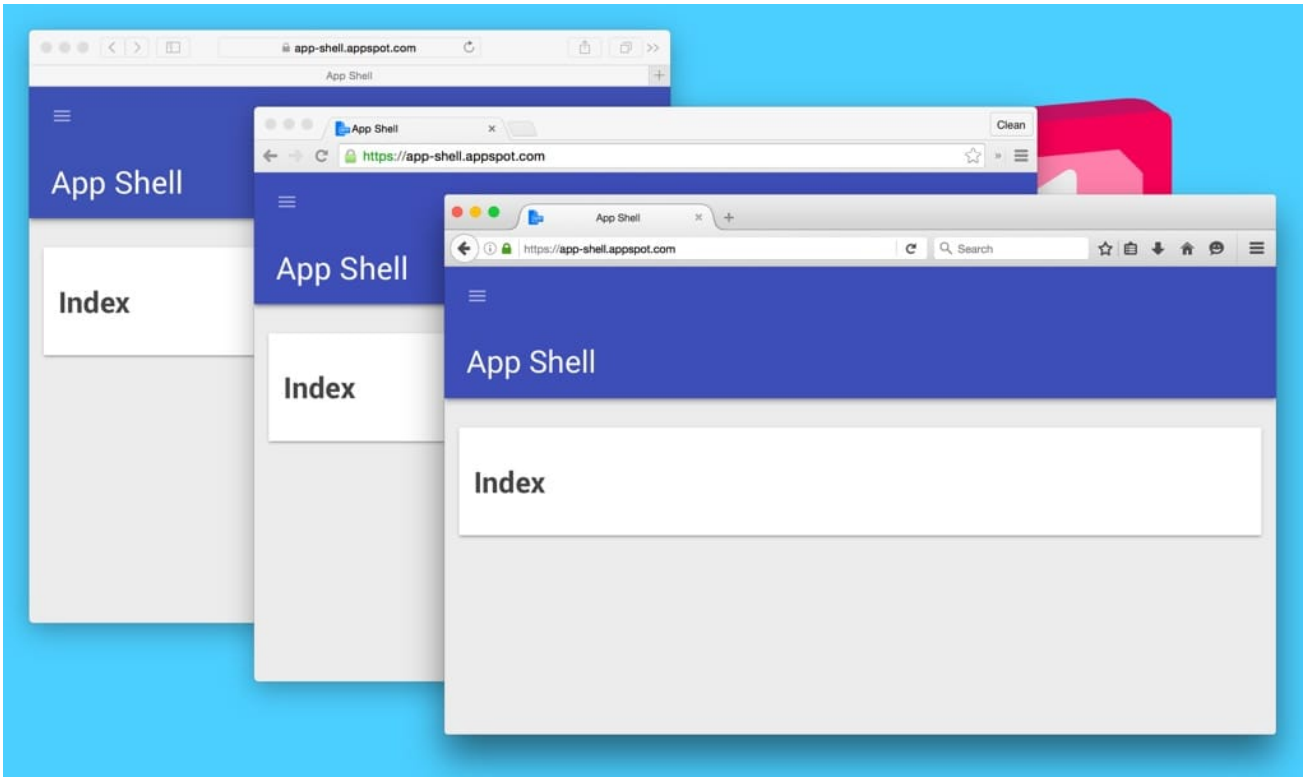
Service workers imply some subtle changes in application architecture. Rather than squashing all of your application into an HTML string, it can be beneficial to do things AJAX-style. This is where you have a shell (that is always cached and can always boot up without the network) and content that is refreshed regularly and managed separately.

The implications of this split are large. On the first visit you can render content on the server and install the service worker on the client. On subsequent visits you need only request data.

What about progressive enhancement?

While service worker isn't currently supported by all browsers, the application content shell architecture uses progressive enhancement to ensure everyone can access the content. For example, take our sample project.

Below you can see the full version rendered in Chrome, Firefox Nightly and Safari. On the very left you can see the Safari version where the content is rendered on the server *without* a service worker. On the right we see the Chrome and Firefox Nightly versions powered by service worker.

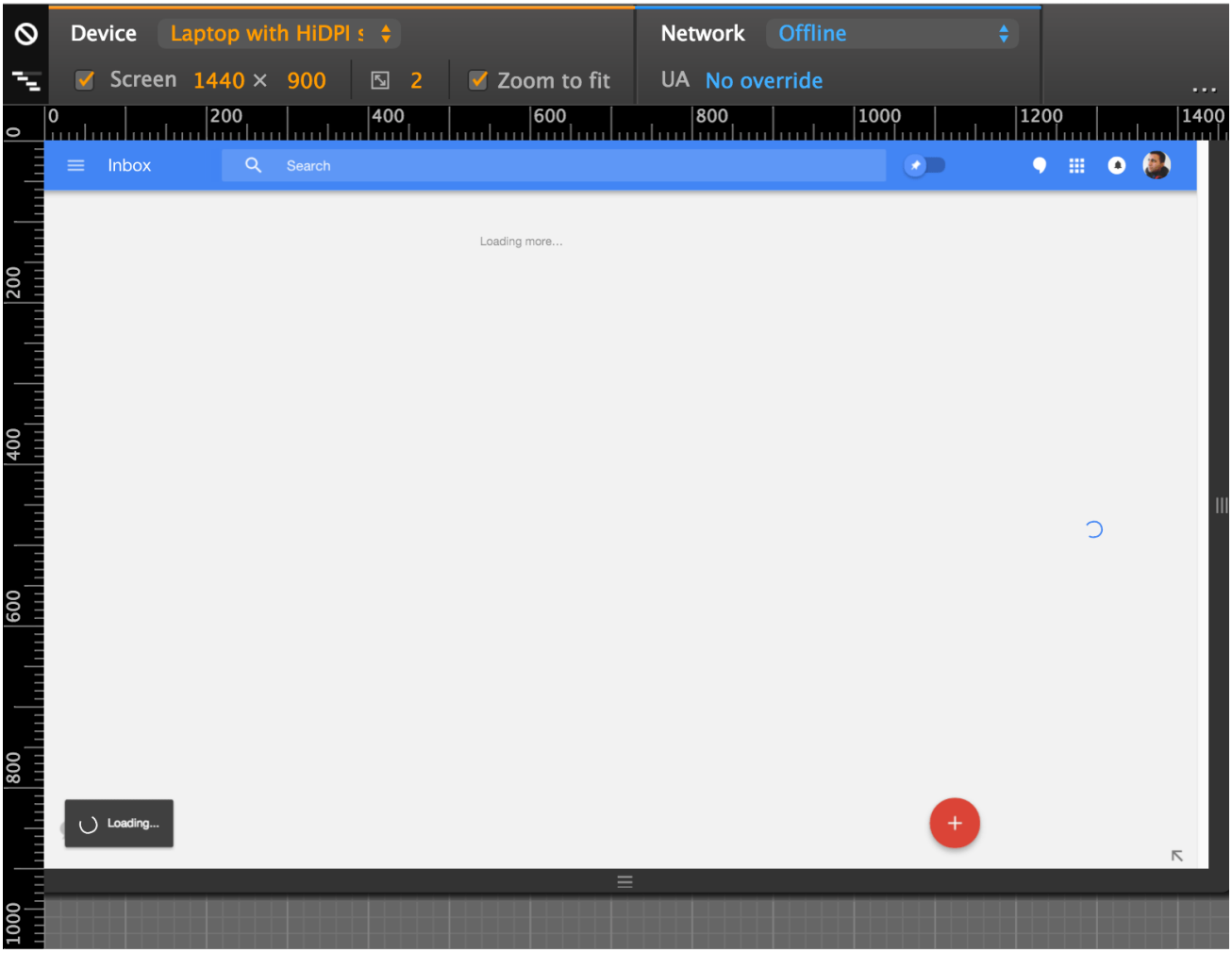


When does it make sense to use this architecture?

The application shell architecture makes the most sense for apps and sites that are dynamic. If your site is small and static, you probably don't need an application shell and can simply cache the whole site in a service worker `oninstall` step. Use the approach that makes the most sense for your project. A number of JavaScript frameworks already encourage splitting your application logic from the content, making this pattern more straight-forward to apply.

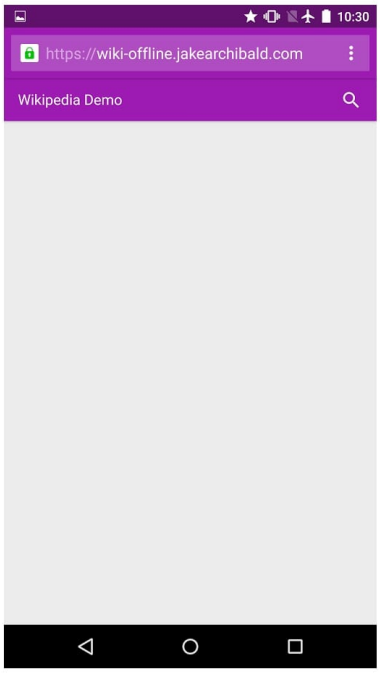
Are there any production apps using this pattern yet?

The application shell architecture is possible with just a few changes to your overall application's UI and has worked well for large-scale sites such as Google's I/O 2015 Progressive Web App and Google's Inbox.

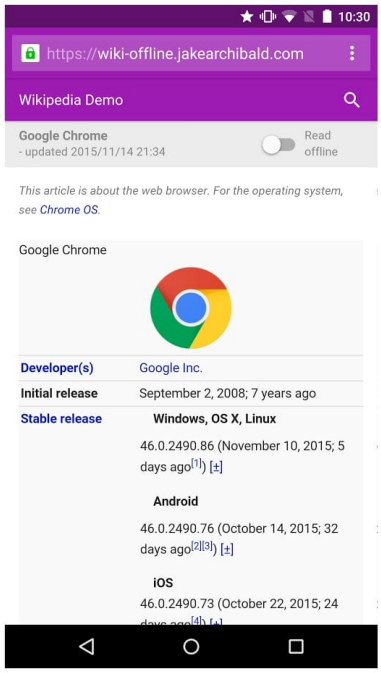


Offline application shells are a major performance win and are also demonstrated well in Jake Archibald's offline Wikipedia app and Flipkart Lite's progressive web app.

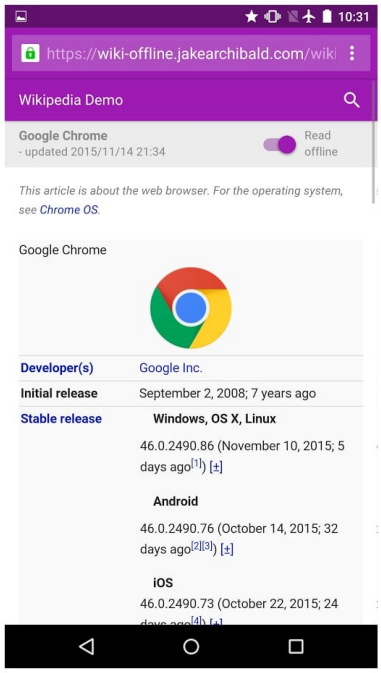
Offline Wikipedia webapp



Cached application shell



Content dynamically loaded in

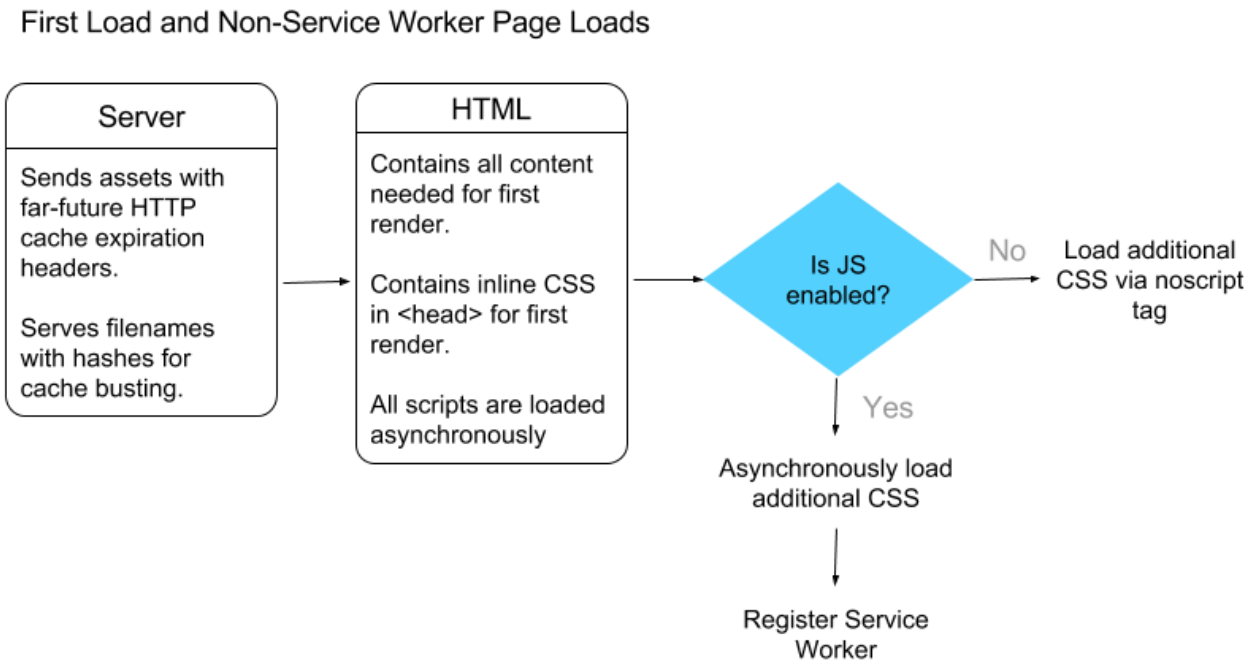


Content can also be cached for offline viewing

Explaining the architecture

During the first load experience, your goal is to get meaningful content to the user's screen as quickly as possible.

First load and loading other pages



In general the application shell architecture will:

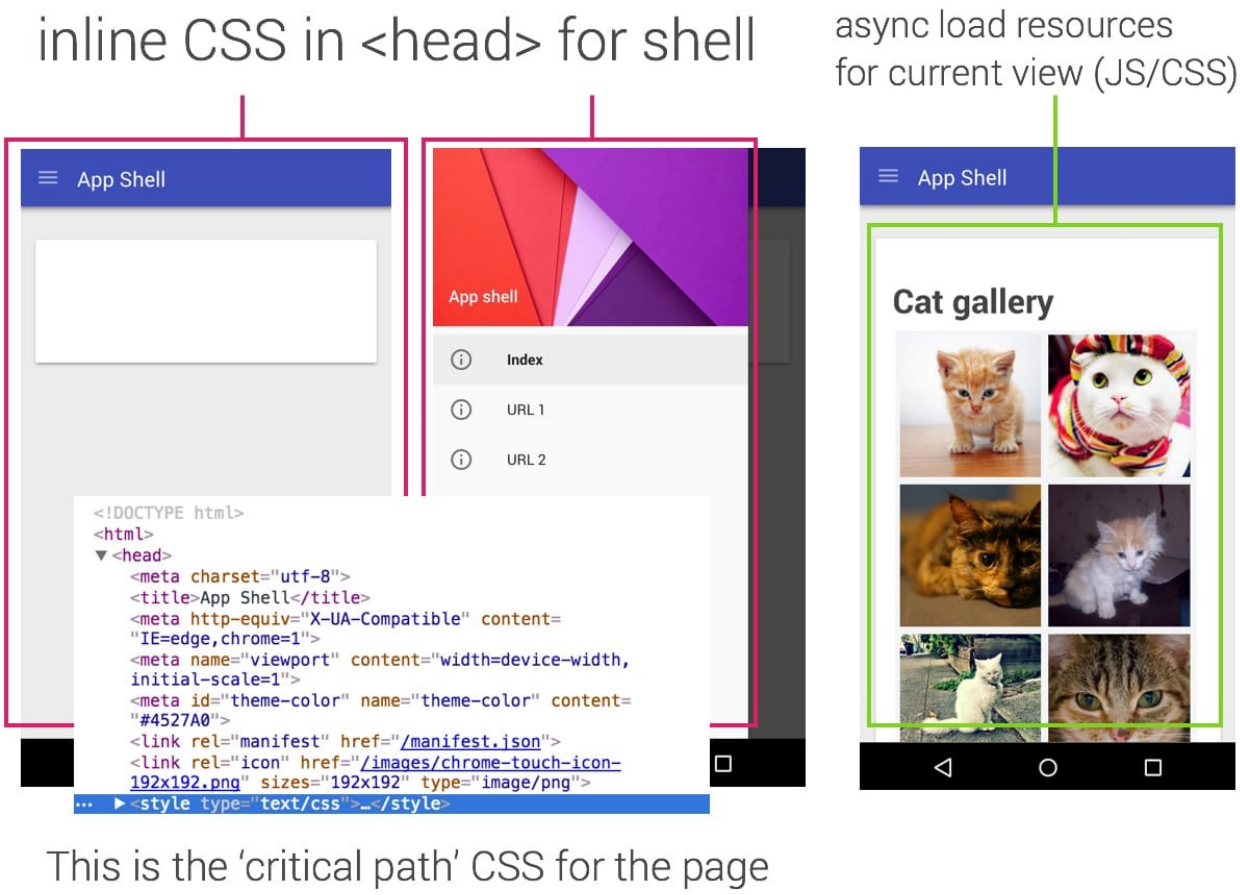
- Prioritize the initial load, but let service worker cache the application shell so repeat visits do not require the shell to be re-fetched from the network.
- Lazy-load or background load everything else. One good option is to use read-through caching for dynamic content.
- Use service worker tools, such as sw-precache, for example to reliably cache and update the service worker that manages your static content. (More about sw-precache later.)

To achieve this:

- **Server** will send HTML content that the client can render and use far-future HTTP cache expiration headers to account for browsers without service worker support. It will serve filenames using hashes to enable both 'versioning' and easy updates for later in the application lifecycle.
- **Page(s)** will include inline CSS styles in a `<style>` tag within the document `<head>` to provide a fast first paint of the application shell. Each page will asynchronously load the JavaScript necessary for the current view. Because CSS cannot be asynchronously loaded, we can request styles using JavaScript as it IS asynchronous rather than parser-driven and synchronous. We can also take advantage of `requestAnimationFrame()` to avoid cases where we might get a fast cache hit and end up with styles accidentally becoming part of the critical rendering path. `requestAnimationFrame()` forces the first frame to be painted before the styles to be

loaded. Another option is to use projects such as Filament Group's loadCSS to request CSS asynchronously using JavaScript.

- **Service worker** will store a cached entry of the application shell so that on repeat visits, the shell can be loaded entirely from the service worker cache unless an update is available on the network.



A practical implementation

We've written a fully working sample using the application shell architecture, vanilla ES2015 JavaScript for the client, and Express.js for the server. There is of course nothing stopping you from using your own stack for either the client or the server portions (e.g PHP, Ruby, Python).

Service worker lifecycle

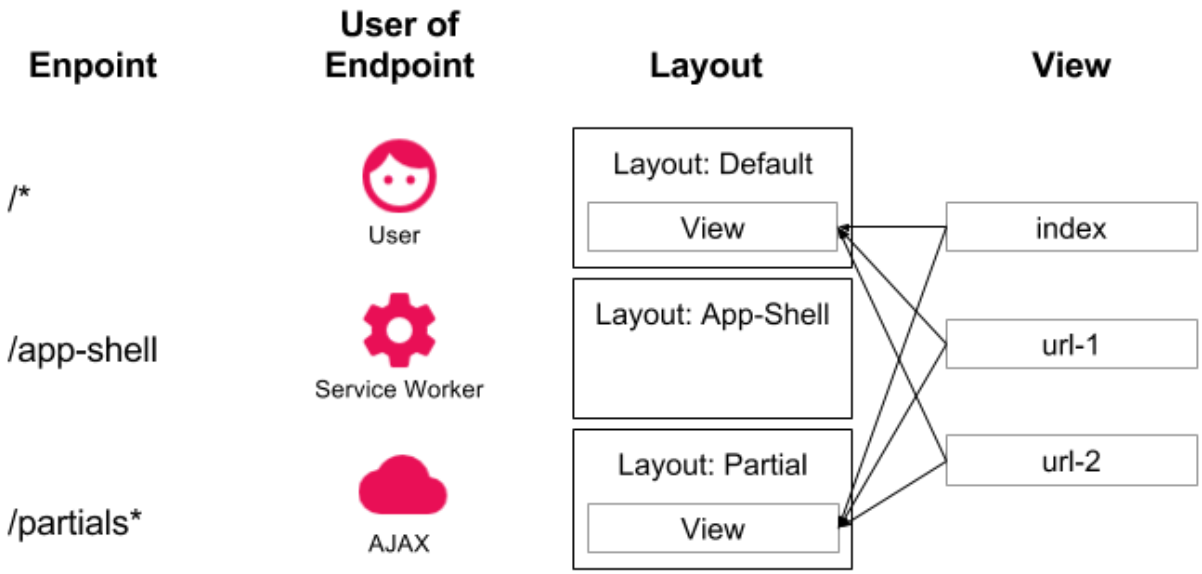
For our application shell project, we use sw-precache which offers the following service worker lifecycle:

Event	Action
Install	Cache the application shell and other single page app resources.
Activate	Clear out old caches.
Fetch	Serve up a single page web app for URL's and use the cache for assets and predefined partials. Use network for other requests.

Server bits

In this architecture, a server side component (in our case, written in Express) should be able to treat content and presentation separately. Content could be added to an HTML layout that results in a static render of the page, or it could be served separately and dynamically loaded.

Understandably, your server-side setup may drastically differ from the one we use for our demo app. This pattern of web apps is achievable by most server setups, though it **does** require some rearchitecting. We've found that the following model works quite well:



- Endpoints are defined for three parts of your application: the user facing URL's (index/wildcard), the application shell (service worker) and your HTML partials.
- Each endpoint has a controller that pulls in a handlebars layout which in turn can pull in handlebar partials and views. Simply put, partials are views that are chunks of HTML that are copied into the final page. Note: JavaScript frameworks that do more advanced data synchronization are often way easier to port to an Application Shell architecture. They tend to use data-binding and sync rather than partials.

- The user is initially served a static page with content. This page registers a service worker, if it's supported, which caches the application shell and everything it depends on (CSS, JS etc).
- The app shell will then act as a single page web app, using javascript to XHR in the content for a specific URL. The XHR calls are made to a /partials* endpoint which returns the small chunk of HTML, CSS and JS needed to display that content. Note: There are many ways to approach this and XHR is just one of them. Some applications will inline their data (maybe using JSON) for initial render and therefore aren't "static" in the flattened HTML sense.
- Browsers **without** service worker support should always be served a fall-back experience. In our demo, we fall back to basic static server-side rendering, but this is only one of many options. The service worker aspect provides you with new opportunities for enhancing the performance of your Single-page Application style app using the cached application shell.

File versioning

One question that arises is how to handle file versioning and updating. This is application specific and the options are:

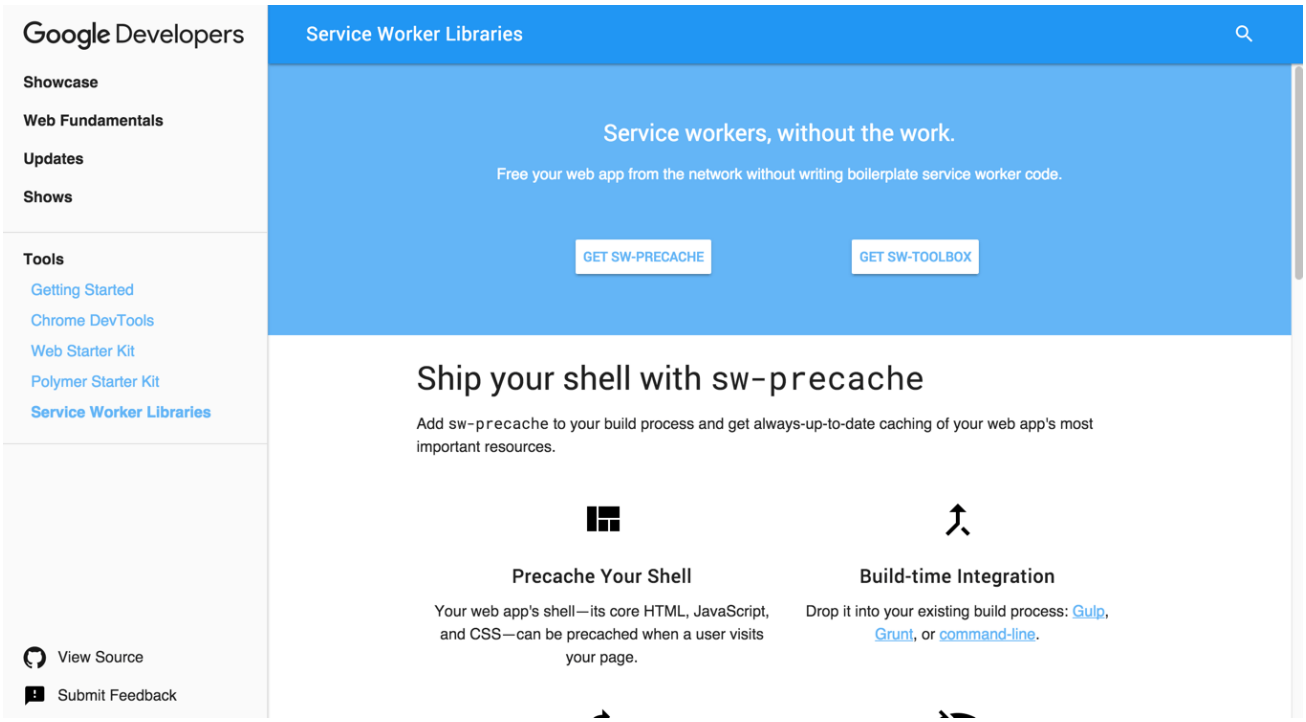
- Network first and use the cached version otherwise.
- Network only and fail if offline.
- Cache the old version and update later.

For the application shell itself, a cache-first approach should be taken for your service worker setup. If you aren't caching the application shell, you haven't properly adopted the architecture.

Note: The application shell sample does not (at the time of writing) use file versioning for the assets referenced in the static render, often used for cache busting. **We hope to add this in the near future.** The service worker is otherwise versioned by sw-precache (covered in the Tooling section).

Tooling

We maintain a number of different service worker helper libraries that make the process of precaching your application's shell or handling common caching patterns easier to setup.



Use sw-precache for your application shell

Using sw-precache to cache the application shell should handle the concerns around file revisions, the install/activate questions, and the fetch scenario for the app shell. Drop sw-precache into your application's build process and use configurable wildcards to pick up your static resources. Rather than manually hand-crafting your service worker script, let sw-precache generate one that manages your cache in a safe and efficient, using a cache-first fetch handler.

Initial visits to your app trigger precaching of the complete set of needed resources. This is similar to the experience of installing a native app from an app store. When users return to your app, only updated resources are downloaded. In our demo, we inform users when a new shell is available with the message, "App updates. Refresh for the new version." This pattern is a low-friction way of letting users know they can refresh for the latest version.

Use sw-toolbox for runtime caching

Use sw-toolbox for runtime caching with varying strategies depending on the resource:

- cacheFirst for images, along with a dedicated named cache that has a custom expiration policy of N maxEntries.
- networkFirst or fastest for API requests, depending on the desired content freshness. Fastest might be fine, but if there's a specific API feed that's updated frequently, use networkFirst.

Conclusion

Application shell architectures comes with several benefits but only makes sense for some classes of applications. The model is still young and it will be worth evaluating the effort and overall performance benefits of this architecture.

In our experiments, we took advantage of template sharing between the client and server to minimise the work of building two application layers. This ensures progressive enhancement is still a first-class citizen.

If you're already considering using service workers in your app, take a look at the architecture and evaluate if it makes sense for your own projects.

With thanks to our reviewers: Jeff Posnick, Paul Lewis, Alex Russell, Seth Thompson, Rob Dodson, Taylor Savage and Joe Medley.

Last updated July 2, 2018.