
Table of Contents

| | |
|--|------|
| Introduction | 1.1 |
| Introduction to the App Shell Architecture | 1.2 |
| Introduction to Service Workers | 1.3 |
| Auditing an existing site with Lighthouse | 1.4 |
| Offline Quickstart | 1.5 |
| Working with Promises | 1.6 |
| Working with the Fetch API | 1.7 |
| Caching files with the Service Worker | 1.8 |
| Working with Indexed DB | 1.9 |
| Designing for slow, unreliable connections | 1.10 |
| Working with live data in the service worker | 1.11 |
| Intro to developer tooling (NPM, Gulp) | 1.12 |
| sw-toolbox and sw-precache | 1.13 |
| Intro to Web Push & Notifications | 1.14 |
| Payment integration | 1.15 |
| Discoverability & Analytics | 1.16 |
| Practical Progressive Enhancement | 1.17 |
| Debugging Service Workers in Browsers | 1.18 |

Progressive Web Apps ILT - Text

Curriculum for instructor-led training course for teaching Progressive Web Apps concepts.
Developed by Google Developer Training

Introduction to Progressive Web App Architectures

Progressive Web Apps (PWAs) use modern web capabilities to deliver fast, engaging, and reliable mobile web experiences that are great for users and businesses.

This document describes the architectures and technologies that allow your web app to support offline experiences, background synchronization, and push notifications. This opens the door to functionality that previously required using a native application.

Contents:

[Instant Loading with Service Workers and Application Shells](#)

[Architectural Styles and Patterns](#)

[Migrating an Existing Site to PWA](#)

[What is an Application Shell?](#)

[How to Create an App Shell](#)

[Building Your App Shell](#)

[Push Notifications](#)

[Conclusion](#)

Instant Loading with Service Workers and Application Shells

Progressive Web Apps combine many of the advantages of native apps and the Web. PWAs evolve from pages in browser tabs to immersive apps by taking ordinary HTML and JavaScript and enhancing it to provide a first class native-like experience for the user.

PWAs deliver a speedy experience even when the user is offline or on an unreliable network. There is also the potential to incorporate features previously available only to native applications, such as push notifications. Developing web apps with offline functionality and high performance depends on using service workers in combination with the Cache API .

Service workers: Thanks to the caching and storage APIs available to service workers, PWAs can precache parts of a web app so that it loads instantly the next time a user opens it. Using a service worker gives your web app the ability to intercept and handle network requests, including managing multiple caches, minimizing data traffic, and saving offline user-generated data until online again. This caching allows developers to focus on speed, giving web apps the same instant loading and regular updates seen in native applications. If you are unfamiliar with service workers, read [Introduction To Service Workers](#) to learn more about what they can do, how their lifecycle works, and more.

A service worker performs its functions without the need for an open web page or user interaction. This enables new services such as Push Messaging or capturing user actions while offline and delivering them while online. (This is unlikely to bloat your application because the browser starts and stops the service worker as needed to manage memory.)

Service workers provide services such as:

- Intercepting HTTP/HTTPS requests so your app can decide what gets served from a cache, local data store, or the network.

A service worker cannot access the DOM but it can access the [Cache Storage API](#), make network requests using the [Fetch API](#), and persist data using the [IndexedDB API](#). Besides intercepting network requests, service workers can use `postMessage()` to communicate between the service worker and pages it controls (e.g. to request DOM updates).

- Receiving push messages from your server

The service worker runs independently from the rest of your web app and provides hooks into the underlying operating system. It responds to events from the OS, including push messages.

- Letting the user do work when offline by holding onto a set of tasks until the browser is on the network (that is, background synchronization)

Think of a service worker as being a butler for your application, waking when needed and carrying out tasks for the app. Effectively, the service worker is an efficient background event handler in the browser. A service worker has an intentionally short lifetime. It wakes up when it gets an event and runs only as long as necessary to process it.

The concept of caching is exciting because it allows you to support offline experiences and it gives developers complete control over what exactly that experience is. But, to take full advantage of the service worker and progressively incorporate more and more PWA capabilities also invites a new way of thinking about building web sites by using the *application shell architecture*.

Application shell (app shell): PWAs tend to be architected around an *application shell*.

This contains the local resources that your web app needs to load the skeleton of your user interface so it works offline and populates its content using JavaScript. On repeat visits, the app shell allows you to get meaningful pixels on the screen really fast without the network, even if your content eventually comes from there. This is not a hard requirement but it comes with significant performance gains.

The shell of the functionality is loaded and displayed to the user (and potentially cached offline by the service worker), and then the page content is loaded dynamically as the user navigates around the app. This reliably and instantly loads on your users' screens, similar to what is seen in native applications.

The [What is an Application Shell?](#) section in this document goes into detail about using an app shell, which is the recommended approach to migrate existing single-page apps (SPAs) and structure your PWA. This architecture provides connectivity resilience and it is what makes a PWA feel like a native app to the user, giving it application-like interaction and navigation, and reliable performance.

Service worker caching should be considered a progressive enhancement. If your web app follows the model of conditionally registering a service worker only if it is supported, then you get offline support on browsers with service workers and on browsers that do not support service workers. The offline-specific code is never called and there is no overhead or breakage for older browsers.

Key Concepts

The app shell approach relies on caching the “shell” of your web application using a service worker. Using the ***app shell + dynamic content model*** greatly improves app performance and works really well with service worker caching as a progressive enhancement.

Progressively enhancing your web app means you can gradually add in features like offline caching, push notifications, and add-to-home-screen.

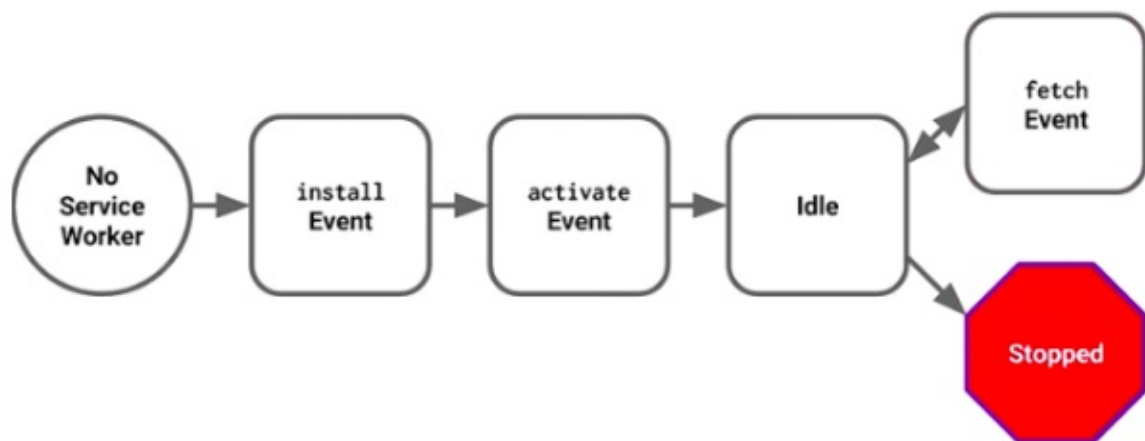
Here is a high-level description of how it works:

1. When the user accesses your website the basic HTML, JavaScript, and CSS display.

On the initial website visit, the page registers the service worker that controls future navigations on the site. Registration creates a new service worker instance and triggers the `install` event that the service worker responds to. Also, the app shell content is added to the cache. Once installed, the service worker controls future navigations on the site.

2. Once the shell content loads, the single-page app requests content to populate the view. The app shell plus dynamic content equals the complete rendered page.

Next, the SPA requests content (for example, via `XMLHttpRequest` or the [Fetch API](#)) and page content is fetched and used to populate the view. The service worker is registered and activated and, if necessary, performs cache cleanup of any out-of-date resources that are no longer needed in your shell. And, once those handlers complete, your service worker enters into an idle state. So, the service worker is running but it is not doing anything until a network request fires off a new event. And, in response to a network request, a `fetch` event handler intercepts the request and responds as you see fit. After a period of idleness your service worker script is stopped automatically but when the next network request is made when you page is loaded again the service worker is started back up and can immediately respond to `fetch` events.



What about browsers that do not support service workers?

The app shell model is great but how does it work in browsers that do not support service workers? Don't worry! Your web app can still be loaded even if a browser is used without service workers. Everything necessary to load your UI (e.g. HTML, CSS, JavaScript) is the same whether or not you use service workers. The service worker simply adds native-like features to your app. Without a service worker, your app continues to operate via HTTPS requests instead of cached assets. In other words, when service workers are not supported, the assets are not cached offline but the content is still fetched over the network and the user still gets a basic experience.

Components

| Component | Description |
|-----------|---|
| app shell | The minimal HTML, CSS, and JavaScript and any other static resources that provide the structure for your page, minus the actual content specific to the page. |

| | |
|-----------------------------|--|
| cache | <p>There are two types of cache: the automatic browser cache and program-controlled caches (Cache API and IndexedDB).</p> <ul style="list-style-type: none"> • The browser cache is a temporary storage location on your computer for files downloaded by your browser to display websites. Files that are cached locally include any documents that make up a website, such as HTML files, CSS style sheets, JavaScript scripts, as well as graphic images and other multimedia content. • The service worker also creates cache independent of the browser cache using the Cache Storage API or IndexedDB. The cache objects hold the same kinds of assets as a browser cache but make them offline accessible. The service worker provides these to enable offline support in browsers. This is referred to as “Cache (Storage) API” in this document. <p>The cache objects are one tool you can use when building your app, but you must use them appropriately for each resource. Several caching strategies are described in Determine the Best Caching Strategy.</p> |
| client-side rendering (CSR) | <p>Client-side rendering means JavaScript running in the browser produces HTML (probably via templating). The benefit is you can update the screen instantly when the user clicks, rather than waiting a few hundred milliseconds at least while the server is contacted to ask what to display. Sites where you mostly navigate and view static content can get away with mostly server-side rendering. Any portion of a page that is animated or highly interactive (a draggable slider, a sortable table, a dropdown menu) almost certainly uses client-side rendering.</p> |
| dynamic content | <p>Dynamic content is all of the data, images, and other resources that your web app needs to function, but exists independently from your app shell. Although the app shell is intended to quickly populate the content of your site, users might expect dynamic content, in which case your app must fetch data specific to the user’s needs. Sometimes an app pulls this data from external, third-party APIs, and sometimes from first-party data that is dynamically generated or frequently updated.</p> |
| Fetch API | <p>You can optionally implement the Fetch API to help the service worker get data. For example, if your web app is for a newspaper, it might make use of a first-party API to fetch recent articles, and a third-party API to fetch the current weather. Both of those types of requests fall into the category of dynamic content.</p> |
| progressive enhancement | <p>An approach to web development that begins with common browser features, and then adds in functionality or enhancements when the user’s browser supports more modern technologies.</p> |
| PWA architecture styles | <p>Any of several approaches to building PWAs based on the back-end technologies available and the performance requirements. The patterns include using an app shell, server-side rendering, client-side rendering, and others. These patterns are listed in Choose an Architecture.</p> |

| | |
|--------------------------------|---|
| server-side rendering (SSR) | SSR means when the browser navigates to a URL fetches the page, it immediately gets back HTML describing the page. SSR is nice because the page loads faster (this can be a server-rendered version of the full page, just the app shell or the content). There's no <i>white page</i> displayed while the browser downloads the rendering code and data and runs the code. If rendering content on the server-side, users can get meaningful text on their screens even if a spotty network connection prevents assets like JavaScript from being fully fetched and parsed. SSR also maintains the idea that pages are documents, and if you ask a server for a document by its URL, the text of the document is returned, rather than a program that generates that text using a complicated API. |
| service worker | A type of web worker that runs alongside your web app but with a life span that is tied to the execution of the app's events. Some of its services include a network proxy written in JavaScript that intercepts HTTP/HTTPS requests made from web pages. It also receives push messages. Additional features are planned in the future. |
| sw-precache | The <code>sw-precache</code> library integrates with your build process and generates code for caching and maintaining all the resources in your app shell. |
| sw-toolbox | The <code>sw-toolbox</code> library is loaded by your service worker at run time and provides pre-written tools for applying common caching strategies to different URL patterns. |
| Universal JavaScript rendering | Universal or Isomorphic JavaScript apps have code that can run on the client-side and the server-side. This means that some of your application view logic can be executed on both the server and the client. This means better performance time to first paint and more stateful web apps. Universal apps come with interesting sets of challenges around routing (ideally, having a single set of routes mapping URI patterns to route handlers), universal data fetching (describing resources for a component independent from the fetching mechanism so they can be rendered entirely on the server or the client) and view rendering. (Views must be renderable on either the client or the server depending on our app needs.) |
| web app manifest | <p>The app shell is deployed from a web app manifest, which is a simple JSON file that controls how the application appears to the user and how it can be launched. (This is typically named <code>manifest.json</code>.) When connecting to a network for the first time, a web browser reads the manifest file, downloads the resources given and stores them locally. Then, if there is no network connection, the browser uses the local cache to render the web app while offline.</p> <p>Note: Do not confuse this with the older <code>.manifest</code> file used by AppCache. PWAs should use the service worker to implement caching and the web app manifest to enable “add to homescreen” and push messaging.</p> |

Architectural Styles and Patterns

Building a PWA does not mean starting from scratch. If you are building a modern single-page app, then you are probably using something similar to an app shell already whether you call it that or not. The details might vary a bit depending upon which libraries or architectures you are using, but the concept itself is framework agnostic. PWA builds on the web architectures you already know.

The prevalent architecture up until recently has been to use **server-side rendering (SSR)**, which is when the browser fetches the page over HTTP/HTTPS and it immediately gets back a complete page with any dynamic data pre-rendered. Server-side rendering is nice because:

- SSR usually provides a quick time to first render and your content is visible to search engines like Google. On the other hand, the consequence of reloading a SSR page is you end up throwing away your entire DOM for each navigation. That means having to pay the cost of parsing, rendering, and laying out the resources on the page each time.
- SSR is a mature technique with a significant amount of tooling to support. Also, SSR pages normally work across a range of browsers without concern over differences in JavaScript implementations or the differing features implemented in each browser.

Sites where you mostly navigate and view static content can get away with using a strictly SSR approach. For sites that are more dynamic, the disadvantage is that SSR throws away the entire DOM whenever you navigate to a new page. And, because of this delay, the app loses its perception of being fast, and users are quickly frustrated and abandon your app.

Client-side rendering (CSR) is when JavaScript runs in the browser and manipulates the DOM. The benefit of CSR is you can update the screen instantly when new data is received from the server, or following user interaction. As with SSR, the consequence of reloading is you end up replacing your entire DOM for each navigation. That means reparsing, rerendering, and laying out the resources on the page each time even if it's only a small portion of the page that changed. Practically every website does some CSR, especially now with the strong trend toward mobile web usage. Any portion of a page that is animated or highly interactive (a draggable slider, a sortable table, a dropdown menu) likely uses client-side rendering.

It is typical to render a page on the server and then *update it dynamically on the client using JavaScript* — or, alternatively, to implement the same features entirely on the client side. Some sites use the same rendering code on the server and client, an approach known as [Universal \(or Isomorphic\) JavaScript](#).

However, the fact is that you do not have to make an “either SSR or CSR” decision. The server is always responsible for getting the data (e.g. from a database) and including it in the initial navigation response, or providing it when the client asks for it after the page has loaded. There is no reason why it cannot do both! Likewise, modern tools such as NodeJS have made it easier than ever to migrate CSR code to the server and use it for SSR, and vice versa.

Whenever possible, the best practice is to combine SSR and CSR so that you first render the page on the server side using data from the server directly. When the client gets the page, the service worker caches everything it needs for the shell (interactive widgets and all). Once the shell is cached, it can query the server for data and re-render the client (the rendering switches to dynamically getting data and displaying fresh updates). In essence, the initial page loads quickly using SSR and after that initial load the client has the option of re-rendering the page with only the parts that must be updated.

This option presumes you can render the same way on the client and server. The reason server-side and client-side rendering is problematic is because they are typically done in different programming environments and in different languages. For websites that blend static, navigable content and app-like interactivity, this can become a huge pain.

Note: If you are using a Universal JavaScript framework, the same templating code might run on both the server and the client but that is not a requirement for using the service worker and app shell model.

In an app shell architecture, a server-side component should be able to treat the content separately from how it is presented on the UI. Content could be added to a HTML layout during a SSR of the page, or it could be served up on its own to be dynamically pulled in. Static content sites such as news outlets can use PWAs and so can dynamic sites such as social media or shopping. What’s important is that the app does something meaningful when offline.

PWA Architectural Patterns

PWAs can be built with any architectural style (SSR, CSR, or a hybrid of the two) but service workers imply some subtle changes in how you build your application architecture.

The following patterns are known styles for building PWAs, listed in recommended order. See the [Table of Known Patterns for Building PWAs](#) for examples of real-world businesses using each pattern.

1. Application shell (SSR) + use JavaScript to fetch content once the app shell is loaded

SSR is optional. Your shell is likely to be highly static, but SSR provides slightly better performance in some cases.

Note: If you are a publisher already considering [Accelerated Mobile Pages \(AMP\)](#), you may be interested in an app shell (SSR) "viewer" + use AMP for leaf nodes (content).

2. Application shell (SSR both shell + content for entry page) + use JavaScript to fetch content for any further routes and do a "take over"

Notes:

- In the future, consider a server-side render of UI with Streams for body content model (even better). See <https://jakearchibald.com/2016/streams-ftw/> to learn more.
 - If you are building a PWA using Polymer leveraging this pattern, then it might be worth exploring SSR of content in the Light DOM
3. Server-side rendering full page (full page caching)
Note: For browsers that do not support service workers, we gracefully degrade to still server-side rendering content (for example, iOS).
 4. Client-side rendering full page (full page caching, potential for JSON payload bootstrapping via server)

Table of Known Patterns for Building PWAs

| Use-case | Patterns | Examples |
|------------|--|--|
| Publishing | Full SSR | https://babe.news/ https://ampbyexample.com https://ampproject.org |
| Publishing | Application Shell | https://app.jalantikus.com/ https://m.geo.tv/ https://app.kompas.com/ https://www.nfl.com/now/ https://www.chromestatus.com |
| Publishing | AppShell + SSR content for entry pages | https://react-hn.appspot.com https://www.polymer-project.org/1.0/ |
| Publishing | Streams for body content / UI | https://wiki-offline.jakearchibald.com/wiki/The_Raccoons |
| Publishing | AppShell + AMP leaf nodes | https://www.washingtonpost.com/pwa/ |
| Social | AppShell | https://web.telegram.org/ |
| E-commerce | Application Shell | https://m.aliexpress.com/ https://kongax.konga.com/ https://m.flipkart.com (mobile/emulate) https://m.airberlin.com/en/pwa https://shop.polymer-project.org/ |
| E-commerce | AppShell + SSR content for entry page | https://selio.com/ (try on mobile/emulate) https://lite.5milesapp.com/ (partial) |
| Conference | AppShell | https://events.google.com/io2016/schedule |

Migrating an Existing Site to PWA

When migrating to a PWA, there are no hard and fast requirements around what to cache. You might, in fact, find it useful to think of your offline strategy as a series of milestones. It is feasible to begin by adding a simple service worker and just caching static assets, such as stylesheets and images, so these can be quickly loaded on repeat visits.

The next step might be caching the full-page HTML or caching the app shell to serve the empty UI first, and then allow the data layer to be pulled in from the server. This is the equivalent of a server sending down a rendered page without any results.

Each milestone allows you to deploy separately, measure the potential performance gains from each step you explore, and progressively roll out a better PWA.

Note: Understanding the network traffic is key to successful migration. You can use the guidelines in [Measure Resource Loading Times](#) to get started using the Network DevTools panel.

Migrating an Existing Site with Server Rendering to PWA

Server-rendered pages can vary in complexity, either being (primarily) static HTML pages or involve more dynamic content. It is useful to think about how you might want to handle dynamic content as a number of different offline caching strategies can be used here. Jake Archibald's [Offline Cookbook](#) is a good reference point once you moved your site over to [HTTPS](#), added a [Web App manifest](#) and can start crafting your service worker story.

Once you decide on a strategy for caching then you must implement it. A SPA architecture is often recommended when using an app shell, but it can take some time to refactor an existing site/app over to this architecture. If refactoring is a daunting task or if using an exclusively SSR approach is your only option for now, then you can still take advantage of service worker caching. But, you might end up treating your UI app shell the same way you would dynamic content.

- A cache-first strategy will not be entirely safe here if your server-rendered content is not entirely static and may change.
- A [cache/network race](#) approach might work as with some combinations of hardware, getting resources from the network can be quicker than going to disk. Just keep in mind that requesting content from the network when the user has some copy of it on their device can be waste potentially costly data.
- A [network-first approach that falls back to the cache](#) might also work. Effectively, provide online users with the most up to date version of the content, but offline users get an older cached version. If a network request succeeds, then ensure the cached version gets updated.

Any of these strategies implements a web app that works offline. However, it is possible for data (any common HTML between `/route1`, `/route2`, `/route3`, etc) to be cached twice. There can be performance and bandwidth hits when going to the network for the full content of the

page as opposed to the app shell approach only fetches content (instead of content + UI). This can be mitigated using proper [HTTP browser caching headers](#).

If you have time for a larger refactor, then try to implement a hybrid approach that relies on server-side rendering for non-service worker controlled navigations. This then upgrades to an SPA-style experience when the service worker is installed. To accomplish this, use a JavaScript framework that supports universal rendering so that the code to render pages is shared between the server and client. React, Ember and Angular are examples of solutions that have universal rendering options.

Additional Migration and Architectural Considerations

- The app shell architecture comes with some challenges because the network request for content is delayed by the app shell loading from the cache, the JavaScript executing, and initiating the fetch. Eventually, Streams is a viable option in this case. Until then, the four known patterns described earlier for building PWAs are valid approaches.
- The app shell should be managed with a cache-first strategy (cache-first, network-fallback). The point is to get reliable performance and to achieve this you must get the cache out of the network. See the [Table of Common Caching Strategies](#) for more information.

What is an Application Shell?

Using the application shell architecture is one way to build PWAs that reliably and instantly load on your users' screens, similar to what you see in native applications. An app shell is the recommended approach to migrating existing single-page apps (SPAs) and structuring your PWA. This architecture provides connectivity resilience and it is what makes a PWA feel like a native app to the user, giving it application-like interaction and navigation, and reliable performance.

An **application shell (or app shell)** refers to the local resources that your web app needs to load the skeleton of your user interface (UI). Think of your app's shell like the bundle of code you would publish to a native app store when building a native app. It is the load needed to get off the ground but might not be the whole story. For example, if you have a native news application, you upload all of the views and fonts and images necessary to render the basic skeleton of the app but not the actual news stories. The news is the dynamic content that is not uploaded to the native app store but is fetched at runtime when the app is opened.

For SPAs with JavaScript-heavy architectures, an application shell is the go-to approach. This approach relies on aggressively caching the “shell” of your web application (typically the basic HTML, JavaScript, and CSS) needed to display your layout and to get the application running. Next, the dynamic content loads for each page using JavaScript. An app shell is useful for getting some initial HTML to the screen fast without a network, even if the content eventually comes in from the network.

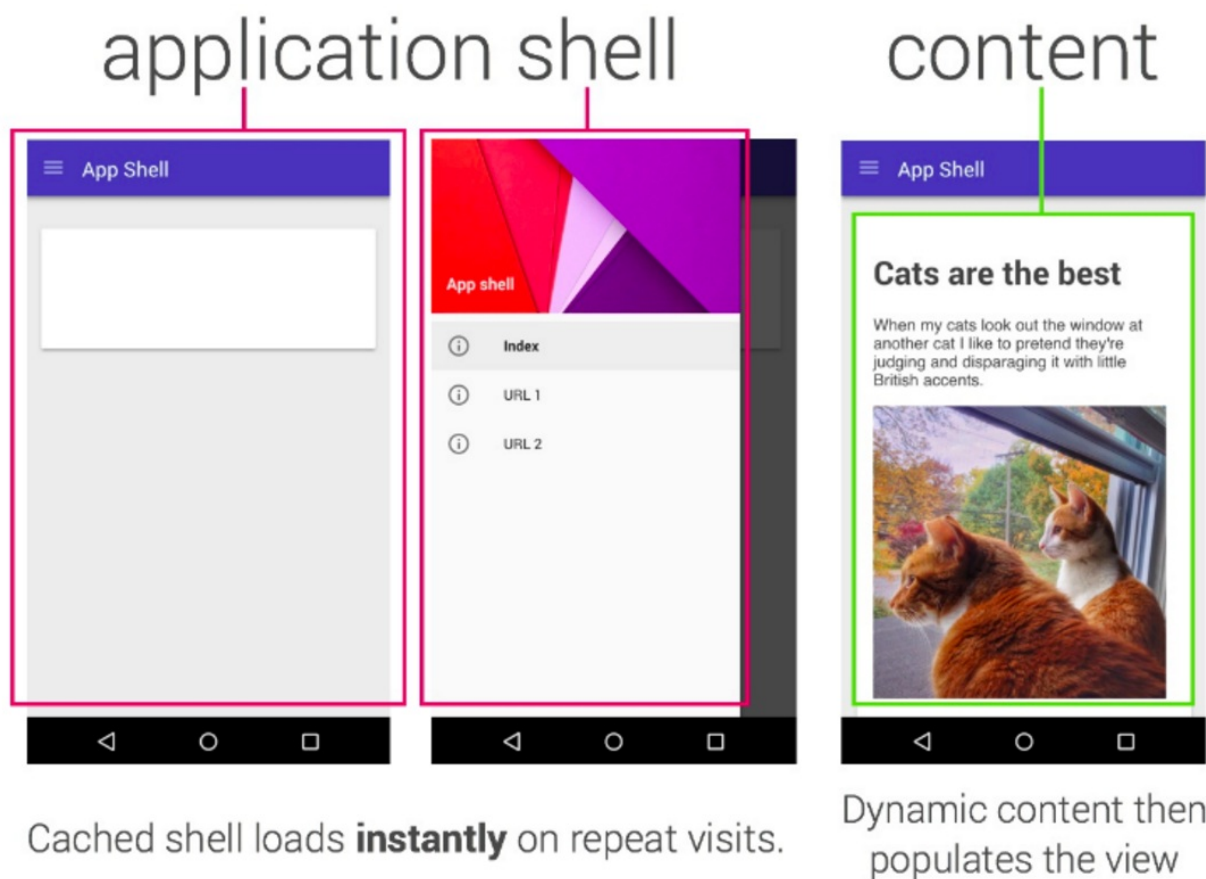
An app shell always includes HTML, usually includes JavaScript and CSS, and might include any other static resources that provide the structure for your page. However, it does not include the actual content specific to the page. In other words, the app shell contains the parts of the page that change infrequently and can be cached so that they can be loaded instantly from the cache on repeat visits. Generally, this includes the pieces of your UI commonly across a few different pages of your site—headers, toolbars, footers and so on—that compose everything other than the primary content of the page. Some static web apps, where the page content does not change at all, consist entirely of an app shell.

The app shell should:

- Load fast
- Use as little data as possible
- Use static assets from a local cache
- Separate content from navigation
- Retrieve and display page-specific content (HTML, JSON, etc.)
- Optionally, cache dynamic content

All resources that are precached are fetched by a service worker that runs in a separate thread. It is important to be judicious in what you retrieve because fetching files that are nonessential (large images that are not shown on every page, for instance) result in browsers downloading more data than is strictly necessary when the service worker is first installed. This can result in delayed loading and consume valuable data, and that often leads to user frustration and abandonment.

The app shell keeps your UI local and pulls in content dynamically through an API but does not sacrifice the linkability and discoverability of the web. The next time the user accesses your app, the latest version displays automatically. There is no need to download new versions before using it.



Building a PWA does not mean starting from scratch. If you are building a modern [single-page app \(SPA\)](#), then you are probably using something similar to an app shell already whether you call it that or not. The details might vary a bit depending upon which libraries or frameworks you are using, but the concept itself is framework agnostic.

To see how Google built an app shell architecture, take a look at [Building the Google I/O 2016 Progressive Web App](#). This real-world app started with a SPA to create a PWA that pre-caches content using a service worker, dynamically loads new pages, gracefully transitions between views, and reuses content after the first load.

When should you use the app shell architecture? It makes the most sense for apps and sites with relatively unchanging navigation but changing content. A number of modern JavaScript frameworks and libraries already encourage splitting your application logic from the content, making this architecture more straightforward to apply. For a certain class of websites that only have static content you can still follow the same model but the site is 100% app shell.

App Shell Features

PWAs use a service worker to cache the app shell and data content so that it always loads fast regardless of the network conditions, even when fully offline, retrieving from cache when appropriate and making live calls when appropriate. For instance, a service worker can redirect HTTP/HTTPS requests to a cache and serve dynamic data from a local database.

But, unlike [the older AppCache standard](#) with its fixed rules, all of these decisions happen in the code that you write. Developers get to decide how network requests from apps are handled.

Benefits

The benefits of an app shell architecture with a service worker include:

- Reliable performance that is consistently fast

Repeat visits are extremely quick. Static assets (e.g. HTML, JavaScript, images and CSS) are immediately cached locally so there is no need to re-fetch the shell (and optionally the content if that is cached too). The UI is cached locally and content is updated dynamically as required.

- Application-like interactions

By adopting the *app shell-plus-content* application model, you can create experiences with application-like navigation and interactions, complete with offline support.

- Economical use of data

Design for minimal data usage and be judicious in what you cache because listing files that are non-essential (large images that are not shown on every page, for instance) result in browsers downloading more data than is strictly necessary. Even though data is relatively cheap in western countries, this is not the case in emerging markets where connectivity is expensive and data is costly.

Example HTML for an App Shell

The example separates the core application infrastructure and UI from the data. It is important to keep the initial load as simple as possible to display just the page's layout as soon as the web app is opened. Some of it comes from your application's index file (inline DOM, styles) and the rest is loaded from external scripts and stylesheets.

All of the UI and infrastructure is cached locally using a service worker so that on subsequent loads, only new or changed data is retrieved, instead of having to load everything.

Assume you are building a simple blog reader. The components of a simple app shell include:

- A link to the manifest file

- Navigation UI and logic
- The code to display posts after they are retrieved from the server (and store them in a local database)
- The code to display comments (also storing them in the database)
- Optionally, the code for posting comments

Your index.html file in your work directory should look something like the following code. This is a subset of the actual contents and is not a complete index file. See [https://\[appshell.appspot.com/\]\(https://app-shell.appspot.com/\)](https://[appshell.appspot.com/](https://app-shell.appspot.com/)) for a real-life look at a very simple app shell.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>App Shell</title>
  <link rel="manifest" href="/manifest.json">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>App Shell</title>
  <link rel="stylesheet" type="text/css" href="styles/inline.css">
</head>
<body>

  <header class="header">
    <h1 class="header__title">App Shell</h1>
  </header>

  <main class="main">
    ...
  </main>

  <div class="dialog-container">
    . . .
  </div>

  <div class="loader">
    <svg viewBox="0 0 32 32" width="32" height="32">
      <circle id="spinner" cx="16" cy="16" r="14" fill="none"></circle>
    </svg>
  </div>

  <script src="app.js" async></script>

</body>
</html>
```

In this example, the key change is to add the link to the manifest file as shown in the bold text. The rest of this file is standard HTML.

Real World Examples

You can see actual offline application shells demonstrated in Jake Archibald's demo of an [offline Wikipedia app](#), [Flipkart Lite](#) (an e-commerce company), and [Voice Memos](#) (a sample web app that records voice memos). For a very basic app shell plus service worker example with minimal options about frameworks or libraries, see app-shell.appspot.com.

Other great examples include [AliExpress](#), one of the world's largest e-commerce sites, [BaBe](#), an Indonesian news aggregator service, [United eXtra](#), a leading retailer in Saudi Arabia, and [The Washington Post](#), America's most widely circulated newspaper.

How to Create an App Shell

So, your next step could be to add a service worker to your existing web app. Using a service worker is one of the things that turns a single-page app into an app shell. (See the [App Shell Features](#) section earlier in this document for a description of the service worker.)

However, there are many situations that can affect your strategy for structuring your PWA and its app shell. It is important to understand the network traffic so you know what to actually precache and what to request as far as dynamic content. These decisions cannot be arbitrary.

You can use [Chrome Developer Tools](#) to help analyze network traffic patterns. It is also important to ask yourself: "What are people trying to achieve when they visit my site?"

Exercise

- What kind of app are you considering?
- What are its main features (for example, displaying blog posts, showing products and maintaining a shopping cart, and so on)?
- What data does the app get from the server (for example, product types and prices)?
- What should the user be able to do when off-line?
- How does your current non-PWA app display data? (for example, by getting it from a database and generating a HTML page on the server?)
 - Provide the data to a single-page app via HTTP/HTTPS (e.g. using `REST`)?
 - Provide the data to a single-page app via another mechanism (e.g. `Socket.io`)?

By using service workers, the appropriate [architectural styles](#), APIs, and the appropriate [caching strategies](#), you gain these benefits:

- Optimize load time for initial and return visitors
- Power your web app while offline
- Offer substantial performance benefits while online

Using Libraries to Code Service Workers

Development with the service worker is not necessarily a trivial process. It is, by design, a low-level API and there can be a fair bit of code involved. While you could write your own service worker code, there are some libraries provided that automate many of the details for you while also following the best practices and avoiding common gotchas.

The `sw-toolbox` and `sw-precache` libraries go together hand-in-hand and are built on top of the service worker primitives, like the Cache and Fetch APIs. The libraries abstract low level complexities and make it easier for developers to work with service workers.

The `sw-toolbox` Library

`sw-toolbox` is loaded by your service worker at run time and provides pre-written tools for applying common caching strategies to different URL patterns. Specifically, it provides common caching patterns and an expressive approach to using those strategies for runtime requests. ([Caching Strategies Supported by sw-toolbox](#) describes this in more.)

Setting Up `sw-toolbox` for Common Caching Strategies

You can install `sw-toolbox` through `Bower`, `npm` or direct from `GitHub` :

```
bower install --save sw-toolbox

npm install --save sw-toolbox

git clone https://github.com/GoogleChrome/sw-toolbox.git
```

To load `sw-toolbox`, use `importScripts` in your service worker file. For example:

```
importScripts('js/sw-toolbox/sw-toolbox.js');
// Update path to match your setup
```

A full code example is shown later in the [Using sw-precache to Create the App Shell](#) section.

More usage information is available in the [Tutorial](#) on Github.

Additional Caching Solutions with `sw-toolbox`

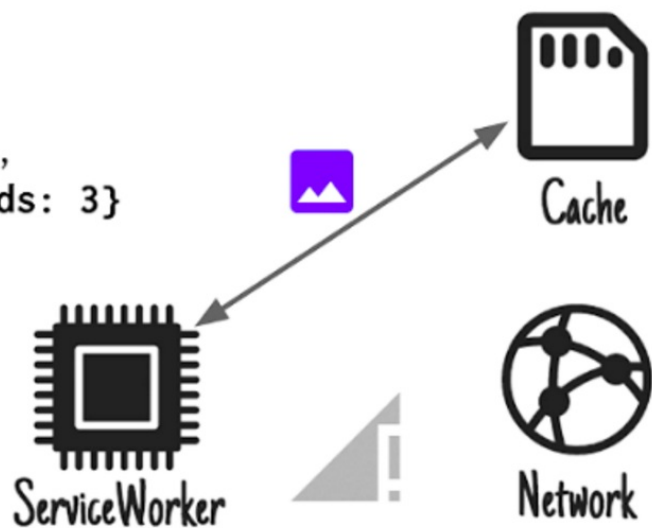
Besides applying common caching strategies, the `sw-toolbox` library is useful for solving a couple of additional problems that arise while fetching your content, making service worker caching even more useful in real world scenarios:

- **“Lie-fi”** is when the device is connected but the network connection is extremely unreliable or slow and the network request drags on and on before eventually failing. Users end up wasting precious seconds just waiting for the inevitable.

While your app shell should always be cached first, there might be some cases where you app uses the “network first” caching strategy to request the dynamic content used to populate your shell. You can avoid Lie-fi in those cases by using `sw-toolbox` to set an explicit network timeout.

The following example uses the `networkFirst` caching strategy to set the timeout to three seconds when fetching an image across the network. If, after those three seconds there is no response from the network, then the app automatically falls back to the cached content.

```
toolbox.router.get(
  '/path/to/image',
  toolbox.networkFirst,
  {networkTimeoutSeconds: 3}
);
```



- **Cache expiration** - As users go from page to page on your site you are probably caching the page-specific content such as the images associated with each page the user visits at run time. This ensures that the full page loads instantly (not just the app shell) on a repeat visit.

But, if you keep adding to dynamic caches indefinitely then your app consumes an ever increasing amount of storage. So `sw-toolbox` actually manages cache expiration for you, saving you the trouble of implementing it yourself.

The following example configures `sw-toolbox` to use a dedicated cache for images with a maximum cache size of 6. Once the cache is full (as it is now) new images cause the least recently used images to be evicted. In addition to the *least recently used* expiration option, `sw-toolbox` also gives you a time-based expiration option where you can automatically expire everything once it reaches a certain age.

```
toolbox.router.get(
  '/path/to/images/.*',
  toolbox.cacheFirst,
  {cache: {
    name: 'images',
    maxEntries: 6
  }}
);
```



Cache

The `sw-precache` Library

`sw-precache` integrates with your build process and automatically generates the service worker code that takes care of caching and maintaining all the resources in your app shell. The `sw-precache` module hooks into your existing node-based build process (e.g. `Gulp` or `Grunt`) and generates a list of versioned resources, along with the service worker code needed to precache them. Your site can start working offline and load faster even while online by virtue of caching.

Because `sw-precache` is a build-time code generation tool, it has direct access to all your local resources, and can efficiently calculate the hash of each to keep track of when things change. It uses those changes to trigger the appropriate service worker lifecycle events and re-downloads only modified resources, meaning that updates are small and efficient, without requiring the developer to manage versioning.

The service worker code generated by `sw-precache` caches and serves the resources that you configure as part of your build process. For mostly static sites, you can have it precache every image, HTML, JavaScript, and CSS file that makes up your site. Everything works

offline, and loads fast on subsequent visits without any extra effort. For sites with lots of dynamic content, or many large images that are not always needed, precaching a subset of your site often makes the most sense.

You can combine `sw-precache` with one of the service worker "recipes" or techniques outlined in the [Offline Cookbook](#) to provide a robust offline experience with sensible fallbacks. For example, when a large, uncached image is requested offline, serve up a smaller, cached placeholder image instead. You can also use wildcards to precache all of the resources that match a given pattern. There is no list of files or URLs that require manual maintenance.

A code example is shown in the [Using sw-precache to Create the App Shell](#) section. There is also a lot more information on the [GitHub project page](#), including a demo project with `gulpfile.js` and `Gruntfile.js` samples, and a [script](#) you can use to register the generated service worker. To see it in action, look at the [app-shell-demo on Github](#).

Caching Strategies Supported by `sw-toolbox`

The best caching strategy for your dynamic content is not always clear-cut and there are many situations that can affect your strategy. For example, when using video or large files, or you do not know the amount of storage on your customer devices, then that forces you to evaluate different strategies.

The gold standard for caching is to use a cache-first strategy for your app shell. If you implement the `sw-precache` API, then the details of caching are handled automatically for you.

Use the following table to determine which caching strategy is most appropriate for the dynamic resources that populate your app shell.

Table of Common Caching Strategies

| Strategy | The service worker ... | Best strategy for ... | Corresponding <code>sw-toolbox</code> handler |
|----------------------|-----------------------------------|---|---|
| Cache first, Network | Loads the local (cached) HTML and | When dealing with remote resources that | <code>toolbox.cacheFirst</code> |

| | instead. | | |
|----------------------------------|---|--|-----------------------------------|
| Network first, Cache fallback | Checks the network first for a response and, if successful, returns current data to the page. If the network request fails, then the service worker returns the cached entry instead. | When data must be as fresh as possible, such as a real-time API response, but you still want to display something as a fallback when the network is unavailable. | <code>toolbox.networkFirst</code> |
| Cache/network race | Fires the same request to the network and the cache simultaneously. In most cases, the cached data loads first and that is returned directly to the page. Meanwhile, the network response updates the previously cached entry. The cache updates keep the cached data relatively fresh. The updates occur in the background and do not block rendering of the cached content. | When content is updated frequently, such as for articles, social media timelines, and game leaderboards. It can also be useful when chasing performance on devices with slow disk access where getting resources from the network might be quicker than pulling data from cache. | <code>toolbox.fastest</code> |
| Network only | Only checks the network. There is no going to the cache for data. If the network fails, then the request fails. | When only fresh data can be displayed on your site. | <code>toolbox.networkOnly</code> |
| Cache only | The data is cached during the <code>install</code> event so you can depend on the data being there. | When displaying static data on your site. | <code>toolbox.cacheOnly</code> |

While you can implement these strategies yourself manually, using `sw-toolbox` is recommended for caching your app's dynamic content. The last column in the table shows the `sw-toolbox` library that provides a canonical implementation of each strategy. If you do

| | | | |
|--|---------------------------------|------------|--|
| | depend on the data being there. | your site. | |
|--|---------------------------------|------------|--|

While you can implement these strategies yourself manually, using `sw-toolbox` is recommended for caching your app's dynamic content. The last column in the table shows the `sw-toolbox` library that provides a canonical implementation of each strategy. If you do implement additional caching logic, put the code in a separate JavaScript file and include it using the `importScripts()` method.

Note that you do not have to choose just one strategy. The `sw-toolbox` routing syntax allows you apply different strategies to different URL patterns. For example:

```
toolbox.router.get('/images', toolbox.cacheFirst);
toolbox.router.get('/api', toolbox.networkFirst);
toolbox.router.get('/profile', toolbox.fastest);
```

For more information about caching strategies, see Jake Archibald's [Offline Cookbook](#).

Exercise: Determine the Best Caching Strategy for Your App

Use the following table to identify which caching strategy provides the right balance between speed and data freshness for each of your data sources. Use the [Table of Common Caching Strategies](#) to fill in the last column. An example is provided after the table.

| Kind of data | When data changes... | Caching Strategy |
|--------------|--|------------------|
| | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | |
| | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | |
| | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | |
| | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | |

Example

| Kind of data | When data changes... | Caching Strategy |
|----------------------|---|---|
| User name | <input checked="" type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | Cache first, Network fallback |
| Product description | <input checked="" type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | Cache first, Network fallback |
| Product price | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input checked="" type="checkbox"/> User should see new value when possible <input type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | Network first, Cache fallback or Cache/network race |
| Product availability | <input type="checkbox"/> Almost never changes <input type="checkbox"/> User can see old value <input type="checkbox"/> User should see new value when possible <input checked="" type="checkbox"/> User must always see latest value <input type="checkbox"/> Secure information - do not cache | Network only |

Regardless of which caching strategy you choose, you can use `sw-precache` to handle the implementation for you. All of the standard caching strategies, along with control over advanced options like maximum cache size and age, are supported via the automatic

inclusion of the `sw-toolbox` library.

Remember that `sw-precache` integrates with your build process, but `sw-toolbox` is loaded by your service worker at run time. `sw-toolbox` is the answer for dynamic, or runtime caching within your web apps.

Considerations

- Service worker caching should be considered a progressive enhancement. If your web app follows the model of conditionally registering a service worker only if it is supported (determined by `if('serviceWorker' in navigator)`), then you get offline support on browsers with service workers and on browsers that do not support service workers. The offline-specific code is never called and there is no overhead or breakage for older browsers.

[Registering a Service Worker](#) shows an example of this.

- All resources that are precached are fetched by a service worker running in a separate thread as soon as the service worker is installed. You should be judicious in what you cache, because listing files that are non essential (large images that are not shown on every page, for instance) result in browsers downloading more data than is strictly necessary.
- Precaching does not make sense for all architectures (described in the [PWA Architectural Styles and Patterns](#) section and also outlined in the [Offline Cookbook](#)),. Several [caching strategies](#) are described later in this document that can be used in conjunction with the `sw-precache` library to provide the best experience for your users. If you do implement additional caching logic, put the code in a separate JavaScript file and include it using the `importScripts()` method.
- The `sw-precache` library uses a [cache-first](#) strategy, which results in a copy of any cached content being returned without consulting the network. A useful pattern to adopt with this strategy is to display an alert to your users when new content is available, and give them an opportunity to reload the page to pick up that new content (which the service worker adds to the cache, and makes available at the next page load). The code for listening for a service worker update lives in the JavaScript for the page that registers the service worker. To see an example, go to this [Github repository](#).

The Key to Designing UIs for PWAs

The following guidelines to great PWA user experience are taken from [Designing Great UIs for Progressive Web Apps](#).

1. Always test on real-world hardware

When you start each new project, find an old and decrepit mobile device and set up [remote debugging with Chrome](#). Test every change to ensure you start fast and stay fast.

2. Get user experience inspiration from native apps

It is possible to take a poor mobile website and slap on service worker caching to improve performance. This is worth doing, but it falls well short of providing the full benefit of a PWA.

- Start by forgetting everything you know about conventional web design, and instead imagine designing a native app. Pay attention to detail because native apps have set a precedent for users expectations around touch interactions and information hierarchy that are important to match to avoid creating a jarring experience.
- Try related native apps on iOS and Android and browse sites like [Dribbble](#) for design inspiration. Spend some time browsing the [Material Design Specification](#) to level up your familiarity with common UI components and their interaction.

3. Use these recommendations to ensure you avoid common errors.

The following checklist is an abridged version of the original one in [Designing Great UIs for Progressive Web Apps](#) by Owen Campbell-Moore that includes more information and great examples.

- Screen transitions shouldn't feel slow due to blocking on the network
- Tappable areas should give touch feedback
- Touching an element while scrolling shouldn't trigger touch feedback
- Content shouldn't jump as the page loads
- Pressing back from a detail page should retain scroll position on the previous list page
- Buttons and 'non-content' shouldn't be selectable
- Ensure inputs aren't obscured by keyboard
- Provide an easy way to share content
- Use system fonts
- Avoid overly "web-like" design (use links sparingly and instead carefully place "buttons" and tappable regions)
- Touch interactions should be implemented very well, or not at all

Building Your App Shell

Structure your app for a clear distinction between the page shell and the dynamic content. In general, your app should load the simplest shell possible but include enough meaningful page content with the initial download. By now you have analyzed your app and the [architectural styles](#), APIs, and [caching strategies](#) and determined the right balance between speed and data freshness for each of your data sources.

Prerequisites

- Make sure your site is served using HTTPS

Service worker functionality is [only available](#) on pages that are accessed via HTTPS.
(`http://localhost` also works well to facilitate testing.)

- Create a web app manifest
- Edit the `index.html` to tell the browser where to find the manifest
- Register the service worker
- Incorporate `sw-precache` into your node-based build script

Use a Web App Manifest File

In essence, the manifest provides the ability to create user experiences that are more comparable to that of a native application. The web app manifest contains metadata provided by the web developer that can be used when a web app is added to a user's homescreen on Android. This includes things like a high-resolution icon, the web app's name, splash screen colors, and other properties.

It is a simple JSON file that provides developers with:

- A centralized place to put metadata about a web site, such as fields for the application name, display mode information such as background color and font size, links to icons, and so on.
- A way to declare a default orientation for their web application, and provide the ability to set the display mode for the application (e.g., in full screen).

The following manifest file is for the simple app shell at <https://app-shell.appspot.com/>.

```
{
  "short_name": "App shell",
  "name": "App shell",
  "start_url": "/index.html",
  "icons": [{
    "src": "images/icon-128x128.png",
    "sizes": "128x128",
    "type": "image/png"
  }, {
    "src": "images/apple-touch-icon.png",
    "sizes": "152x152",
    "type": "image/png"
  }, {
    "src": "images/ms-touch-icon-144x144-precomposed.png",
    "sizes": "144x144",
    "type": "image/png"
  }, {
    "src": "images/chrome-touch-icon-192x192.png",
    "sizes": "192x192",
    "type": "image/png"
  }, {
    "src": "images/chrome-splashscreen-icon-384x384.png",
    "sizes": "384x384",
    "type": "image/png"
  }],
  "display": "standalone",
  "orientation": "portrait",
  "background_color": "#3E4EB8",
  "theme_color": "#2E3AA1"
}
```

To include the manifest file in your app, include a link tag in your index.html to tell the browser where to find your manifest file:

```
<!-- Add to your index.html -->

<!-- Web Application Manifest -->
<link rel="manifest" href="manifest.json">
```

Tip:

- To read the W3C draft specification, see the [W3C Web App Manifest](#).
- To automatically generate a manifest from existing HTML, try the [ManifeStation](#) website.
- To test the validity of a web manifest according to the rules from the W3C specification, try the [Web Manifest Validator](#).

Registering a Service Worker

Service worker caching should be considered a progressive enhancement. If you follow the model of conditionally registering a service worker only when supported by the browser (determined by `if('serviceWorker' in navigator)`), you get offline support on browsers with service workers and on browsers that do not support service workers, the offline-specific code is never called. There's no overhead/breakage for older browsers.

Here's a high-level overview of the steps required to make your app work offline.

1. Register the service worker if it is supported by the browser
 - i. Verify that the browser supports service workers.
 - ii. If it does, register the service worker JavaScript file in the browser.
 - iii. Create a JavaScript file containing the service worker.

For example, add the following code to your app:

```
if ('serviceWorker' in navigator) {  
  navigator.serviceWorker  
    .register('./service-worker.js')  
    .then(function() { console.log('Service Worker Registered'); });  
}
```

2. Cache the site assets
3. Serve the app shell from the cache

The above code snippet checks to see if the browser supports service workers, and if it does, calls the register method that returns a [Promise](#). After the registration is completed, the browser resolves the Promise and calls the function in the `.then()` clause. (Note: this happens asynchronously.)

This is a very simple service worker registration snippet. From the main page, in addition to registering your service worker, you also have the opportunity to listen for service worker lifecycle events. For example, you could display a message to your users saying "Hey, something's been updated in the background, please refresh this page."

You can see a more complete, sophisticated implementation of this sort of service worker lifecycle management code in [Github](#).

Caching the Application Shell

You can manually hand code an app shell or use the `sw-precache` service worker library to automatically generate it and minimize the amount of boilerplate code you must write.

Note: The examples are provided for general information and illustrative purposes only. The actual resources used, such as jQuery, may be different for your application.

Caching the App Shell Manually

```
var cacheName = 'shell-content';
var filesToCache = [
  '/css/bootstrap.css',
  '/css/main.css',
  '/js/bootstrap.min.js',
  '/js/jquery.min.js',
  '/offline.html',
  '/',
];

self.addEventListener('install', function(e) {
  console.log('[ServiceWorker] Install');
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      console.log('[ServiceWorker] Caching app shell');
      return cache.addAll(filesToCache);
    })
  );
});
```

Using `sw-precache` to Cache the App Shell

The [sw-precache Library](#) section earlier in this document describes this API in detail. This section describes how you can run the `sw-precache` API as a command-line tool or as part of your build process.

Important: Every time you make changes to local files and are ready to deploy a new version of your site, re-run this step. To ensure this is done, include the task that generates your service worker code in your list of tasks that are automatically run as part of your deployment process.

Using `sw-precache` From the Command Line

To test the result of using `sw-precache` without changing your build system for every version of the experiment, you can run the `sw-precache` API at from the command line.

First, create a `sw-precache-config.json` file with our `sw-precache` configuration. In this example `staticFileGlobs` indicates the path to each file that we want to precache and `stripPrefix` tells `sw-precache` what part of each file path to remove.

```
{
  "staticFileGlobs": [
    "app/index.html",
    "app/js/main.js",
    "app/css/main.css",
    "app/img/**/*.svg",
    "app/img/**/*.png",
    "app/img/**/*.jpg",
    "app/img/**/*.gif"
  ],
  "stripPrefix": "app/"
}
```

Once the `sw-precache` configuration is ready then run it with the following command:

```
$ sw-precache --config=path/to/sw-precache-config.json --verbose
```

Using `sw-precache` From Gulp

The following code example uses the `gulp` command to build a project. It first creates a `gulp` task that uses the `sw-precache` module to generate a `service-worker.js` file. The following code is added to the `gulp` file:

```
/*jshint node:true*/
(function() {
  'use strict';

  var gulp = require('gulp');
  var path = require('path');
  var swPrecache = require('sw-precache');

  var paths = {
    src: 'app/'
  };

  gulp.task('generate-service-worker', function(callback) {
    swPrecache.write(path.join(paths.src, 'service-worker.js'), {

      //1
      staticFileGlobs: [
        paths.src + 'index.html',
        paths.src + 'js/main.js',
        paths.src + 'css/main.css',
        paths.src + 'img/**/*.{svg,png,jpg,gif}'
      ],
      // 2
      importScripts: [
        paths.src + '/js/sw-toolbox.js',
        paths.src + '/js/toolbox-scripts.js'
      ],
      // 3
      stripPrefix: paths.src
    }, callback);
  });
})();
```

What Happens Next?

When you run `gulp` you should see output similar to the following:

```
$ gulp generate-service-worker
[11:56:22] Using gulpfile ~/gulpfile.js
[11:56:22] Starting 'generate-service-worker'...
Total precache size is about 75.87 kB for 11 resources.
[11:56:22] Finished 'generate-service-worker' after 49 ms
$
```

This process generates a new `service-worker.js` file in the app directory of your project. All resources that are precached are fetched by a service worker running in a separate thread as soon as the service worker is installed.

Remember to rerun the API each time any of your app shell resources change to pick up the latest versions.

Push Notifications

Once the first interaction with a user is complete, re-engaging on the web can be tricky. Push notifications address this challenge on native apps, and now the [Push API](#) is available on the web as well. This allows developers to reconnect with users even if the browser is not running. Over 10 billion push notifications are sent every day in Chrome, and it is growing quickly. Push Notifications can provide great benefits for users if applied in a timely, relevant and precise manner.

How Do Push Notifications Work?

Push notifications enable an app that is not running in the foreground to alert users that it has information for them. For example, the notification could be a meeting reminder, information from a website, a message from an app, or new data on a remote server. A push notification originates on a remote server that you manage, and is *pushed* to your app on a user's device.

Even if the user is not actively using your app, upon receiving a push notification the user can tap it to launch the associated app and see the details. Users can also ignore the notification, in which case the app is not activated.

Each browser manages push notifications through their own system, called a “push service.” When the user grants permission for Push on your site, the app subscribes to the browser's internal push service. This creates a special subscription object that contains the “endpoint URL” of the push service, which is different for each browser, and a public key. Your application server sends push messages to this URL, encrypted with the public key, and the push service sends it to the right client.

Users engage most with apps that have fresh content and updated information. Push Notifications (from services sending data using the Push Protocol) can deliver the latest information on news, weather, travel alerts, hotel booking information, e-commerce events, and more.

What you get:

- System level notifications, like native apps
- Fresh content that engages users
- Works even when page is closed

- Helps to avoid abandoned shopping carts

Service workers are the driving force behind push notifications. Push notifications with PWAs is described in complete detail in [Web Push Notifications Codelab](#).

What Happens When the App Shell Gets a Push Message?

What happens when push notifications are enabled when using an app shell but the user is offline? The [Push API](#) allows the server to push the message even while the app is not active.

The app shell gets a token that is used to register for push notifications. The app shell keeps the token for decoding messages later (or unregistering from the notifications). Because the server is not sending any data to the client right now, it just sends a simple message called a *tickle* that wakes up the service worker and triggers the `push` event in the service worker. Inside the `push` event handler the service worker can retrieve the data from the push message, perform some logic based on the data, and display the notification.

Once the user sees the notification they can ignore it until later, dismiss it, or action it. The user normally tap on the notification to choose. These actions raise events in the service worker that you can handle any way you choose. For example, if the user dismisses the notification the app might log this to our analytics. If they click the notification, the app could take them to the specific part of the app that was referenced in the notification.

Conclusion

Using the architectures and technologies in this document means you now have a key to unlock faster performance, push notifications, and offline operation. The *app shell + service worker* model is the one of the best ways to structure your web apps if you want reliable and instant load times. This model also allows you to progressively enhance your web app to support additional offline experiences, background synchronization, and push notifications..

Where does all of this leave you as a developer who wants to use PWA architectures?

If you are starting from scratch and want inspiration or just want to see a finished real-world example, look at Jeff Posnick's `iFixit` API demo (client):

- [Source code on GitHub](#)
- [Deployed example](#)

If you are building a modern single-page app and want to add a service worker to your existing web app, then get started by looking at Jake Archibald's Offline Wikipedia demo.

- [Source code on GitHub](#)
- [Deployed example](#)

Introduction to Service Worker

Contents:

[What Is a Service Worker?](#)

[What Can Service Workers Do?](#)

[Service Worker Lifecycle](#)

[Service Worker Events](#)

What Is a Service Worker?

A service worker is a client side, programmable, proxy. It also gives you fine control over the requests it handles. For example: you can control the caching behavior of requests for your site's HTML and treat them differently than requests for your site's images.

Service workers are a type of [web worker](#), an object that executes a script separately from the main browser thread. Because workers run separately from the main thread, service workers run independent of the application they are associated with and can receive messages when not active (either because your application is in the background or not open, or the browser is closed).

The primary use for service workers is as a caching agent to handle network requests and store content for offline use.

The service worker becomes idle when not in use, and restarts when it's next needed. If there is information that you need to persist and reuse across restarts, then service workers can work with [IndexedDB](#) databases.

Service workers are promise based. At a high level, a promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation. If you need a refresher about promises, then see the [Promise Codelab](#) for a good starting point.

Service workers also depend on two APIs to work effectively: [Fetch](#) (a standard way to retrieve content from the network) and [Cache](#) (a persistent content storage for application data. This cache is persistent and independent from the browser cache or network status).

Because of the power of a service worker to rewrite responses to the client and to prevent man-in-the-middle attacks, service workers are only available on secure origins served through TLS and using the HTTPS protocol. We will test service workers using localhost, however, which is exempt from this policy.

Services like [Letsencrypt](#) allow you to procure SSL certificates for free to install on your server.

What Can Service Workers Do?

Service workers enable applications to control network requests, cache those requests to improve performance and provide offline access to cached content.

This is just the tip of the iceberg. We will explore some things you can do with a service worker.

Improve Performance of Your Application/Site

Caching by the browser content will make content load faster under most network conditions. Two specific types of caching behavior are suitable for use in your service worker:

Precache Assets During Installation

If you have assets (HTML, CSS, JavaScript, images, media) that are shared across your application you can cache them when you first install the service worker in the client's browser, during the install process. This technique is at the core of the [Application Shell Architecture](#).

Note that using this technique does not preclude regular additional caching during fetch events or the way the browser caches the resources on your site.

Provide a Fallback for Offline Access

Using the Fetch API inside a Service Worker we can fetch requests and then modify the response with content other than the object requested. We can use this technique to provide alternative images in case the images are not available in cache and the network is unreachable.

Act As the Base for Advanced Features

Service workers are designed to work as the starting point for features that make web applications work like native apps. Some of these features are:

- **Channel Messaging API**: Allows web workers and service workers to communicate with each other and with the host application. Examples of this API include new content notification and updates that require user interaction
- **Notifications API**: A way to integrate push notifications to the Operating System's native notification system
- **Push API**: An API that enables push services to send push messages to an application. Servers can send messages at any time, even when the application or the browser is not running. Push messages are delivered to a Service Worker which can use the information in the message to update local state or display a notification to the user
- **Background Sync** lets you defer actions until the user has stable connectivity. This is useful for ensuring that whatever the user wants to send, is actually sent. This API also allows servers to push periodic updates to the app so the app can update when it's next online

Service Worker Lifecycle

Every service worker goes through three steps in its lifecycle:

- Registration
- Installation
- Activation

Registration and Scope

To **install** a service worker, you need to **register** it in your main JavaScript code.

Registration tells the browser where your service worker is located, and to start installing it in the background. For example, you could include a `<script>` element in your site's index.html file with the following code:

```
// Check for browser support of serviceWorker
if (!('serviceWorker' in navigator)) {
  console.log('Service worker not supported');
  return;
}
navigator.serviceWorker.register('/service-worker.js')
.then(function(registration) {
  // Successful registration
  console.log('Registration successful, scope is:', registration.scope);
})
.catch(function(error) {
  // Failed registration, service worker won't be installed
  console.log('Service worker registration failed, error:', error);
});
```

This code starts by checking for browser support by examining `navigator.serviceWorker`. The service worker is then registered with `navigator.serviceWorker.register`, which returns a [promise](#) that resolves when the service worker has been successfully registered. The **scope** of the service worker is then logged with `registration.scope`.

The **scope** of the service worker determines from which path the service worker will intercept requests. The default scope is the path to the service worker file, and extends to all lower directories. So if **service-worker.js** is located in the root directory, the service worker will control requests from all files at this domain.

You can also set an arbitrary scope by passing in an additional parameter when registering. The code below shows an example.

```
navigator.serviceWorker.register('/service-worker.js', {
  scope: '/app/'
});
```

In this case we are setting the scope of the service worker to `/app/`, which means the service worker will control requests from pages like `/app/`, `/app/lower/` and `/app/lower/lower`, but not from pages like `/app` or `/`, which are higher.

You can attempt to register a service worker every time the page loads, and the browser will only complete the registration if the service worker is new or has been updated.

Installation and Activation

Once the browser registers a service worker, **installation** can be attempted. This will occur if the service worker is considered to be new by the browser, either because this is the first service worker encountered for this page, or because there is a byte difference between

the current service worker and the previously installed one. An example of an installation event looks like this:

```
// Listen for install event, set callback
self.addEventListener('install', function(event) {
  // Perform installation
});
```

If this is the first encounter of a service worker for this page, then installation will be attempted and the service worker will transition into the **activation** stage upon success. We can use activation event to clean up stale data from existing caches for the application.

```
self.addEventListener('activate', function(event) {
  // Perform activation or update
});
```

Once activated, the service worker will control all pages that load within its scope, and intercept corresponding network requests.

However the pages in your app that are open will not be under the service worker's scope since the service worker was not loaded when the pages opened. To put currently open pages under service worker control you must reload the page or pages. Until then, requests from this page will bypass the service worker and operate like they normally would.

Service workers maintain control as long as there are pages open that are dependant on that specific version. This ensures that only one version of the service worker is running at any given time. Old service workers will become redundant and be deleted once all pages using it are closed. This will activate the new service worker and allow it to take over.

Refreshing the page is not sufficient to transfer control to a new service worker, because there won't be a time when the old service worker is not in use.

Service Worker Events

Service workers are event driven. Both the installation and activation processes fire off corresponding **install** and **activate** events to which the service workers can respond. There are also **message** events, where the service worker can receive information from other scripts, and functional events such as **fetch**, **push**, and **sync**.

To examine service workers, navigate to the Service Worker section in your browser's developer tools; The process is different in each browser that supports service workers. For information about using your browser's dev tools to check the status of service workers, see

[Debugging Service Workers in Browsers.](#)

Lighthouse PWA analysis tool

Contents:

Introduction

Lighthouse

[Running Lighthouse as a Chrome extension](#)

[OPTIONAL: Running Lighthouse from the Command Line](#)

Introduction

How do I tell if I have all of my Progressive Web App (PWA) features in order? [Lighthouse](#) is an open-source tool from Google that audits a web app for PWA features. It provides a set of metrics to help guide you in building a fully PWA with a full application-like experience for your users.

We will discuss what Lighthouse is and how you can use it from both as a Chrome extension and, optionally, from the command line .

Lighthouse

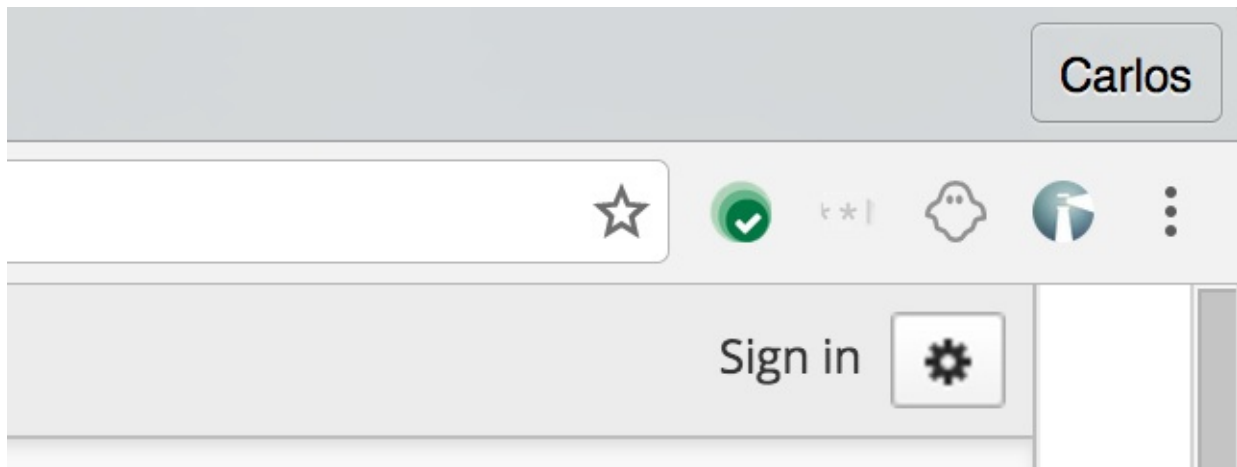
Lighthouse is a tool that tests and provides feedback on aspects of your progressive web application. It audits whether your app can load in offline or flaky network conditions, checks your experience is relatively fast, tests whether your application is served from a secure origin, and performs a number of other general web dev best practice audits like accessibility.

Lighthouse is available as a Chrome extension for Chrome 52 and later and a command line tool. We've chosen to concentrate on the Chrome extension as it provides a more friendly user interface than the command line; the extension is also free of dependencies, other than the Chrome browser.

Running Lighthouse as a Chrome extension

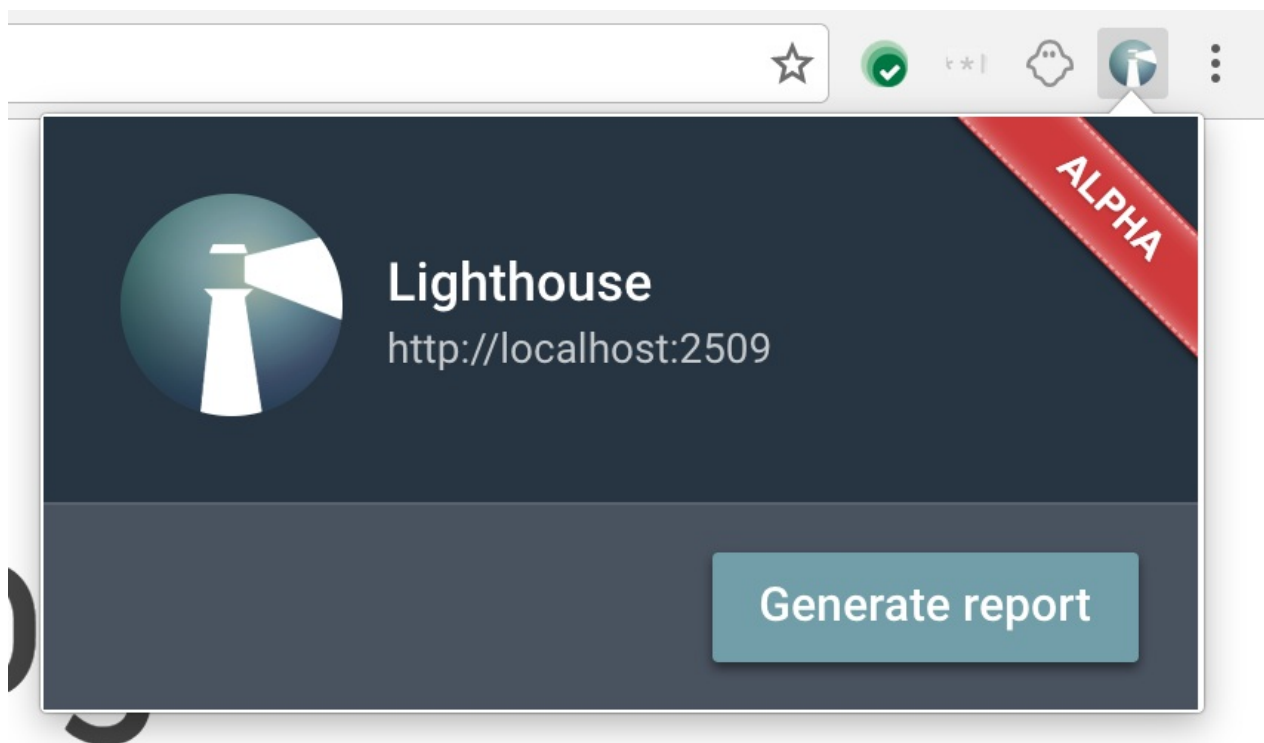
Download the Lighthouse Chrome extension from the [Chrome Web Store](#).

When installed it'll place an icon in your taskbar.



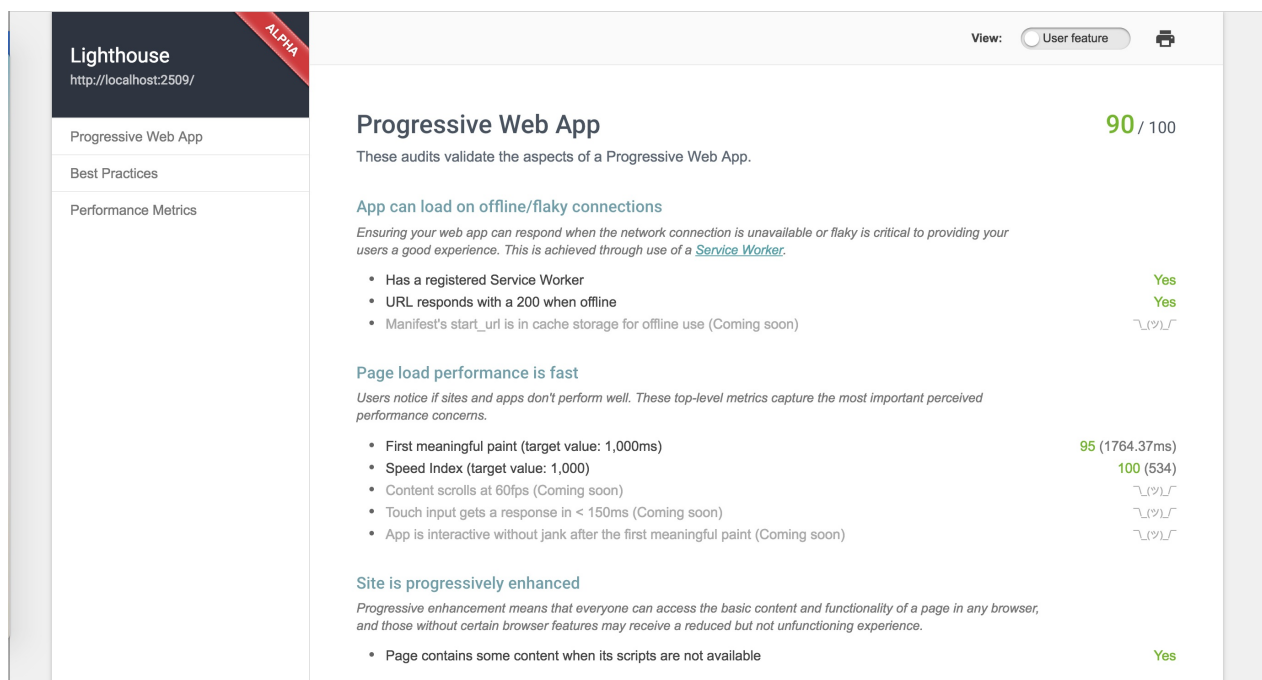
Navigate to airhorner.com, click on the Lighthouse icon and then click on **Generate report**

As an option you can start a local web server pointing to your application and generate a report from a local copy of your application.



Lighthouse will run the reports and generate an HTML page with the results. You can print this page and, if your Operating System allows it, save it as PDF to share with others.

An example page is shown below.



A copy of the report will be saved to your hard drive for further review.

OPTIONAL: Running Lighthouse from the Command Line

If you want to run Lighthouse from the command line (for example: to integrate it with a build system) it is available as a Node module that you can install from the terminal.

If you haven't already, download Node from nodejs.org and select the version that best suits your environment and operating system.

Install Lighthouse's Node module from the terminal:

```
$ npm install -g GoogleChrome/lighthouse
```

Configure Chrome to run Lighthouse. The command looks in your node_modules directory for Lighthouse and then calls an NPM script that launches Chrome with special flags Lighthouse needs to run its tests.

```
$ npm explore -g lighthouse -- npm run chrome
```

Run Lighthouse on a demo Progressive Web App:

```
$ lighthouse https://airhorner.com/
```

Check Lighthouse flags and options:

```
$ lighthouse --help
```


Offline Quickstart

Contents:

[Why Build Offline Support?](#)

[How Do I Take My App Offline?](#)

[References](#)

Why Build Offline Support?

Increasingly, the majority of growth in internet traffic comes from mobile-first and (in some cases), mobile-only connections in regions where internet connectivity is sparse, expensive or just unreliable.

Most of the internet growth comes from mobile first and sometimes from mobile-only connections in regions where internet connectivity is expensive or unreliable. Even in a best-case scenario, mobile connectivity might not be as reliable as we want it to be.

As application developers, we want to ensure a good user experience, preventing network outages from affecting your application. This is particularly true when the user's device appears to be connected to the network but the network is extremely unreliable or slow and nothing is coming through.

We now have a way to build offline support using [service workers](#). Service workers provide an in-browser, programmable network proxy, so that users can always get to something on your website.

Service workers are a browser feature that provides many new services to your web application, including caching files under program control, intercepting network requests so you can build intelligent caches, and receiving push messages in the background. Each web application can have a single service worker associated with it. The service worker runs independently of the web app and can even be called when your app isn't running (e.g. to wake it up and deliver a message.) A service worker executes scripts that you write to manage the cache, process push messages, and more.

Some benefits of implementing service workers include:

- Offline access

- Improved performance
- Access to advanced (browser independent) features

Offline Access

Service workers use the [Cache API](#) to cache your application's static assets and then it fetches the remaining content from the network. Your service worker script can implement more complex caching strategies, including storing data in a local database. This enables new services such as capturing user actions while offline and delivering them while online.

Service worker's approach was driven by the problems the community had with AppCache, namely that a purely declarative approach to caching proved to be too inflexible. Service worker provides a mechanism to create your own caching strategies. For an example of working with App Cache and the challenges developers face, see Jake Archibald's [Application Cache is a Douchebag](#) in A List Apart.

Unlike Application Cache, service workers make all behavior explicit without defaults to guess. If a behavior is not written into your service worker, then the behavior does not happen. There are no default behaviors to fall back on. By explicitly coding behaviors in a service worker, the task of writing and debugging is made easier because all the code the service worker uses is written in the script, there is no defaults to guess or implicit behavior to figure out.

Improved Performance

Caching data locally results in speed and cost benefits for mobile users (many pay according to their usage). The content is cached in the user's browser and, when you configure your service worker, to retrieve data from the cache without going to the network, you provide a faster experience for everyone.

The service worker stores data in an object store using the Cache Storage API or indexedDB. The Cache Storage API is controlled programmatically and it is independent of the browser's cache. The [cache objects](#) hold the same kinds of assets as a browser cache but make them offline accessible. The service worker provides these to enable offline support in browsers.

The cache objects are one tool you can use when building your app, but you must use them appropriately for each resource. Several caching strategies are described in [The Offline Cookbook](#), especially [Serving Suggestions: Responding to requests](#)

Access to Browser Independent Features

Service workers are the foundation for browser independent features for web applications. Because a service worker's lifecycle is independent of the web app's lifecycle, the service worker can receive notifications from the server even when the web app isn't running. Examples include receiving push notifications, syncing data in the background, and geofencing. Not all browsers that support service workers support the advanced features, but users can still access your website - it doesn't break in older browsers that don't support that feature. To see if your target browser supports a given feature, check [Is ServiceWorker Ready](#).

How Do I Take My App Offline?

The core of an offline experience is the service worker. It allows users in supported browsers to cache your application's content and store it locally so when the user next visits the app the content is accessed from the local cache and displays faster to the user.

The top level structure of the sample application looks like this:

```
offline-quickstart/solution-code
| -app
| ---css
| ---images
| ----touch
| ---js
| ---index.html
```

You can download the sample repository from: <https://github.com/caraya/offline-quickstart-lab/>

To take your application offline requires these steps that are discussed in the following sections:

- [Create a Service Worker Script](#)
- [Register the Service Worker](#)

Create a Service Worker Script

In the root directory of your application create a `service-worker.js` file and copy the following code into it:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('static-cache-v1')
      .then(function(cache) {
        return cache.addAll([
          'offline/index.html',
          'offline/main.js',
          'offline/main.css'
        ]);
      })
  );
});

self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request)
    .then(function(response) {
      if (response) {
        return response;
      }
      return fetch(event.request);
    })
  ).catch(function() {
    console.log('resource not available');
  });
});
```

This is the most basic service worker-enabled script possible. It receives two events from the service worker:

- Install Event

The install event is called after the service worker has loaded the script. We'll use it to cache a list of static assets. The service worker exposes a [cache management API](#). This script retrieves the files (`index.html` , `main.css`, and `main.js`) and stores them in a cache file named `static-cache-v1` .

- Fetch Event

The service worker also acts as a network proxy, intercepting HTTP requests between your app and the browser and emitting a `fetch` event per request. This code listens for fetch events to deliver files from the cache.

The fetch handler takes the following steps:

1. Search the caches in the user's browser for a file matching the request. If such a file is found, then return the matching item from the cache as the response.
2. If the file is not in the cache, then attempt to retrieve it from the network.

3. If the network request (2) fails, then log the error message to the console.

Register the Service Worker

Open your `index.html` file and add the following code to the bottom of the file, right before your closing `</body>` tag:

```
<script>
if ('serviceWorker' in navigator) { //1
  navigator.serviceWorker.register('service-worker.js', {scope: './'}) //2
  .then(function(registration) {
    console.log('service worker successfully registered'); // 3
  })
  .catch(function(error) {
    console.log('registration failed ' + error); // 4
  });
} else {
  window.alert('service workers not supported in this browser'); //5
}
</script>
```

This script performs the following tasks

1. Checks if the browser supports service worker by testing if `window.navigator` implements the `serviceWorker` attribute. If it's not implemented, then the browser does not support service worker and we jump to step 5.
2. Register the script by passing a name and a URL scope. The scope tells the service worker what URL paths to intercept.
3. The service worker returns a [Promise](#) that resolves once it has registered the script. For now, log the success message to console.
4. If the registration fails, then so does the promise. The `.catch` clause logs the error to the console and exits. This indicates there was a problem registering the service worker. It **does not** mean service workers are not supported.
5. If the browser doesn't support service workers then we log an appropriate message to the console.

We don't need to place this script on every page. Placing it on `index.html` (or whatever you name as the entry point to your application) is enough to register the service worker for the entire application.

Summary

We built a service worker that caches a set of resources the first time a user accesses the site. The service worker also intercepts HTTP (and HTTPS) requests for the application's files and attempts to serve them from the cache before trying the network.

This document describes how to build a minimal implementation of a service worker, and it does not take into account updates to the cached files, user-controlled caching (e.g. marking content for offline reading), providing offline alternatives, and so on. More advanced uses of service workers are provided throughout the training.

References

General Information

- [Is ServiceWorker Ready?](#)
- [Mozilla Developers Network](#)
- [Introduction to service workers](#)
- [Fetch Event](#)
- [Push Notifications and Service Workers](#)
- [Your first push notifications web app](#)

Sample Applications

- [IO 2016](#)
- [Trained to Thrill](#)
- [Chrome Dev Summit](#)
- [Push Demo](#)

Promises Textbook

Contents:

Introduction and supported browsers

What is a promise

[Promise Terminology](#)

[Promise chains, then and catch](#)

[Not all promise-related functions have to return a promise](#)

Other uses for Promises

[Running Promises simultaneously: Promise.all](#)

[Using the first promise to finish: Promise.race](#)

Caveats

[Immutable results](#)

Introduction and supported browsers

In this document you will learn about promises, a new way to handle asynchronous code in Javascript. Promises have been around for a while in the form of libraries, such as:

- [Q](#)
- [when](#)
- [WinJS](#)
- [RSVP.js](#)

The promise libraries listed above and promises that are part of the ES2015 JavaScript specification (also referred to as ES6) are all [Promises/A+](#) compatible.

Promises are supported natively in all browsers except Internet Explorer and Opera Mini as shown in the matrix below, taken from [caniuse.com](https://caniuse.com/promises)

Promises - OTHER Global 74.46% + 0.07% = 74.53%

A promise represents the eventual result of an asynchronous operation.

Current aligned Usage relative Show all

| IE | Edge * | Firefox | Chrome | Safari | Opera | iOS Safari * | Opera Mini * | Android Browser * | Chrome for Android |
|----|--------|---------|--------|--------|-------|--------------|--------------|-------------------|--------------------|
| | | | 29 | | | | | | |
| | | | 49 | | | | | 4.3 | |
| | | | 50 | | | | | 4.4 | |
| 8 | 13 | 47 | 51 | | | 9.2 | | 4.4.4 | |
| 11 | 14 | 48 | 52 | 9.1 | 39 | 9.3 | all | 51 | 51 |
| | | 49 | 53 | 10 | 40 | | | | |
| | | 50 | 54 | TP | 41 | | | | |
| | | 51 | 55 | | | | | | |

Notes Known issues (0) Resources (8) Feedback

No notes

What is a promise

If you've done any work with JavaScript you've probably seen code like this using callbacks and nested function calls:

```
function isUserTooYoung(id, callback) {
  openDatabase(db => {
    getCollection(db, 'users', col => {
      find(col, {'id': id}, result => {
        result.filter(user => {
          callback(user.age < cutoffAge);
        });
      });
    });
  });
}
```

The more callbacks that you use in a callback chain the harder it is to read and analyze its behavior. Fortunately ES2015 introduces promises as a way to handle asynchronous operations. The above example becomes much simpler using promises:


```
function isUserTooYoung(id) {
  return openDatabase()
    .then(db => getCollection(db, 'users'))
    .then(col => find(col, {'id': id}))
    .then(user => user.age < cutoffAge);
}
```

Note: `.then()` takes a function. The expression `user => {` is an ES2015 shorthand for `function (user) {`.

Formally speaking, a promise represents the eventual result of an asynchronous operation. It is a placeholder that will eventually hold the successful result or the reason for failure. Most new web specifications return Promises from their asynchronous methods and your code can do the same.

Promise Terminology

When working with promises you may hear different terminology to that you may be used to when working with callbacks or other synchronous code

```
function loadImage(url) {
  // wrap image loading in a Promise
  return new Promise((resolve, reject) => {
    // A new Promise is "pending"
    var image = new Image();
    image.src = url;

    image.onload = () => {
      // Resolving a Promise changes its state to "fulfilled"
      resolve(image);
    };

    image.onerror = () => {
      // Rejecting a promise changes its state to "rejected"
      reject(new Error('Could not load image at ' + url));
    };
  });
}
```

A Promise is in one of these states:

- Pending: the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet
- Fulfilled: The operation resolves and the promise has a value
- Rejected: The operation failed and the promise will never be fulfilled. A failed promise

has a reason indicating why it failed

You may also hear the term **settled**: it represents a promise that has been acted upon, either fulfilled or rejected.

Promise chains, then and catch

Remember our earlier example that calls several actions in a row:

```
function isUserTooYoung(id) {  
  return openDatabase()  
    .then(db => getCollection(db, 'users'))  
    .then(col => find(col, {'id': id}))  
    .then(user => user.age < cutoffAge);  
}
```

Once we have an initial promise, we can attach additional functions (using `.then()` and `.catch()`) to create a *promise chain*. In a promise chain, the output of one function serves as input for the next.

Calling `.then()` unwraps the value from the current promise. It returns a promise that can be passed to follow on functions or a value that can be acted upon or returned and used as a parameter in follow on functions in the promise chain. This allows you to chain any number of actions using `.then()`.

In the following example, we load an image using `loadImage()` and apply a series of conversions using `then()`. If at any point we get an error (either the original promise or any of the subsequent steps rejects) we jump to the `catch()` statement and end the process.

Only the last `then()` statement will attach the image to the DOM. Until then we return the same image

```
function processImage(imageName, domNode) {
  return loadImage(imageName)
    .then(image => {
      // returns an image for the next step
      return scaleToFit(150, 225, image);
    })
    .then(image => {
      // returns another image for the next step
      return watermark('Google Chrome', image);
    })
    .then(image => {
      // returns the image for the final display
      return grayscale(image);
    })
    .then(image => {
      // Attach the image to the DOM after all
      // processing has been completed
      showImage(image);
    })
    .catch(error => {
      console.log('we had a problem in running processImage, + error')
    });
}
```

Every promise includes two methods to chain actions: `then()` and `catch()` :

then()

This schedules a function to be called when the promise is fulfilled. When the promise is fulfilled, `.then()` extracts the promise's value, calls the function, and wraps the returned value in a new promise.

Think of `then()` as the try portion of a try/catch block.

catch()

Promises also provide a mechanism to simplify error handling. When a promise rejects (or throws an exception), it locates the first `.catch()` call following the error and passes control to its function. (Think of the whole chain as being wrapped in an implicit `try { } block`.)

Most developers place a single catch at the end of the chain to handle any error or exception that arises.

Even though we have been calling `then` with a single function, it actually accepts two functions: `then(success, error)` -- you can supply the error handler as the second function.

In some places you may see code like:

```
p.then(val => console.log("fulfilled:", val))
  .then(null, err => console.log("rejected:", err));
```

Calling `then(null, function(err) { })` is equivalent to using `catch()`, so we can rewrite the previous snippet as:

```
p.then(val => console.log("fulfilled:", val))
  .catch(err => console.log("rejected:", err));
```

Using `catch` is preferred as it's easier to see what the promise is doing.

Not all promise-related functions have to return a promise

The following functions will work as parts of the image processing chain and they do not return a promise directly but they do return the image passed to them so we can give it to the next function in the chain.

```
function scaleToFit(width, height, image) {
  image.width = width;
  image.height = height;
  console.log('Scaling image to ' + width + ' x ' + height);
  return image;
}

function watermark(text, image) {
  var container = document.createElement('p');
  container.innerHTML = text;
  document.body.appendChild(container);
  console.log('Watermarking image with ' + text);
  return image;
}

function grayscale(image) {
  image.classList.add('grayscale');
  console.log('Converting image ' + image.src + ' to grayscale');
  return image;
}

function showImage(image) {
  var imageEl = new Image();
  imageEl.src = image;
  document.body.appendChild(imageEl);
}
```

Other uses for Promises

Running Promises simultaneously: Promise.all

There are times when we need to run multiple steps of the chain in parallel either because we need to make sure that all tasks complete before proceeding or the order in which tasks complete is less important than ensuring all tasks complete successfully (or fail appropriately).

In the example below, request1 and request2 return promises. We want both of them to load before proceeding. If either request rejects then Promise.all rejects with the value of the rejected promise. If both requests fulfill, Promise.all fulfilled as well.

```
var request1 = fetch('/users.json');
var request2 = fetch('/articles.json');

Promise.all([request1, request2]) // Array of promises to complete
  .then(results => {
    console.log('all data has loaded');
  })
  .catch(error => {
    console.log('one or more requests have failed: ' + error);
  });
```

Using the first promise to finish: Promise.race

`Promise.race()` takes an array of promises and returns a promise with the value (resolve or reject) of the first settled promise.

We can use a promise race to create a timer to use when we want our task to take no more than the specified time.

We first create a delay function to set up the time we will race against.

```
function delay(ms) {
  return new Promise(function (resolve, reject) {
    setTimeout(resolve, ms);
  });
}
```

We then set up a race between a function we want to run and a delay. For example: we want to load a resource and make sure that it will load in less than the time we set in the delay function, 2000 milliseconds in the example below.

```
Promise.race([
  fetch('http://example.com/file.txt'),
  delay(2000).then(() => {
    throw new Error('Timed out')
  });
])
  .then(text => {
    console.log('text downloaded successfully');
  })
  .catch(reason => {
    console.log('timeout triggered')
  });
```

If the fetch promise fulfills first then the race resolves and the then portion of the promise is executed.

If the delay promise resolves first then we throw an error; this will automatically reject Promise.race and the catch portion of the promise is executed.

This is commonly used when trying to load a resource under a specified time.

Caveats

Immutable results

When a promise is settled (either fulfilled or rejected) it will remain settled and the success value or reason for error will remain unchanged.

If we want a different response from a promise that has already fulfilled we need to create a new promise object rather than use an existing one.

Fetch API Concepts

Contents:

[What is Fetch?](#)

[Making a Request](#)

[Reading the Response Object](#)

[Making HEAD and POST Requests](#)

[Further Reading](#)

What is Fetch?

The [Fetch API](#) is a simple interface for fetching resources. It is an update to the older [XMLHttpRequest](#) that requires you to implement additional logic (e.g. handling redirects).

Note: fetch supports the [Cross Origin Resource Sharing \(CORS\)](#). Testing generally requires [running a local server](#). Also note that fetch does not require HTTPS, but service workers do. So utilizing fetch in a service worker will require HTTPS. Local servers are exempt from this. You can check for browser support of fetch in the window interface. For example:

```
if (!('fetch' in window)) {  
  console.log('Fetch API not found, try including the polyfill');  
  return;  
}  
// We can safely use fetch from now on
```

There is a [polyfill](#) for [browsers that are not currently supported](#).

The [fetch\(\)](#) method takes the absolute path to a resource as input. The method returns a [promise](#) that resolves to the [Response](#) of that request.

Making a Request

Let's look at a simple example of fetching a JSON file:

```
fetch('/examples/example.json')
  .then(function(response) {
    // Do stuff with the response
  })
  .catch(function(error) {
    console.log('Fetch failed', error);
  });
```

We pass the absolute path for the resource we want to retrieve as a parameter to `fetch`. In this case this is *example.json*. The `fetch` call returns a promise that will resolve to a response object.

When the promise resolves the response is passed to `then()`. This is where the response could be used. If the request does not complete `catch()` takes over, and is passed the corresponding error.

Response objects represent the response to a request. They contain the requested resource and useful properties and methods. For example, `response.ok`, `response.status`, and `response.statusText` can all be used to evaluate the status of the response.

Evaluating the success of responses is particularly important when using `fetch` because bad responses (like 404's) still resolve. The only time a `fetch` promise will reject is if the request was unable to complete. So the previous code segment would fall back to `catch()` if there was no network connection but not if the response was bad (like a 404). The previous code can be updated to validate responses:

```
fetch('/examples/example.json')
  .then(function(response) {
    if (!response.ok) {
      throw Error(response.statusText);
    }
    // Do stuff with the response
  })
  .catch(function(error) {
    console.log('Fetch failed', error);
  });
```

Reading the Response Object

Response objects also have methods. For example, `Response.json()` reads the response and returns a promise that resolves to JSON. Adding this step to the current example updates the code to:


```
fetch('/examples/example.json')
  .then(function(response) {
    if (!response.ok) {
      throw Error(response.statusText);
    }
    // Read the response as json.
    return response.json();
  })
  .then(function(responseAsJson) {
    // Do stuff with the JSON
    console.log(responseAsJson);
  })
  .catch(function(error) {
    console.log('Fetch failed', error);
  });
```

This code will be cleaner and easier to understand if it's abstracted into functions:

```
function readResponseAsJSON(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response.json();
}

function logResult(result) {
  console.log(result);
}

function logError(error) {
  console.log('Fetch failed', error);
}

fetch('/examples/example.json') // Step 1
  .then(responseAsJSON) // Step 2
  .then(logResult) // Step 3
  .catch(logError);
```

(This is [promise chaining](#).)

To summarize what is happening:

Step 1. Fetch is called on a resource, *example.json*. Fetch returns a promise that will either resolve to a Response object or reject with an error. If it rejects, `catch()` takes over and the error is passed to `logError()`. If it resolves, the Response object is passed to `readResponseAsJSON()` in the first `then()` block and step 2 begins.

Step 2. `readResponseAsJSON()` checks the validity of the response (i.e. is it a 200?). If the response is bad an error is thrown immediately and `catch()` takes over. This is particularly important because without this check bad responses would be passed down the line and could break later code that might rely on a having a valid response. If the response is good, it is read with `response.json()` which returns yet another promise that will resolve to a JSON object. If this operation fails for some reason, the promise will reject and `catch()` will take over. If it's successful, the JSON object will be passed to `logResult()` in the next `then()` block, and step 3 begins.

Step 3. Finally, the JSON object from the original request to *example.json* is logged.

Example: Fetching Images

Let's look at an example of fetching an image and appending it to a web page. Similarly to `Response.json()`, `Response.blob()` reads a response into a [Blob](#). The [URL object's](#) `createObjectURL()` method can then be used to generate a data URL representing the Blob. That URL can be used as the `src` attribute in an image tag. This can be achieved by adding the following functions to the previous code:

```
function readResponseAsBlob(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response.blob();
}

function showImage(responseAsBlob) {
  const myImage = document.querySelector('img');
  const myImageUrl = URL.createObjectURL(responseAsBlob);
  myImage.src = myImageUrl;
}
```

And updating the fetch chain to:

```
fetch('/images/kittens.jpg')
  .then(readResponseAsBlob)
  .then(showImage)
  .catch(logError);
```

Unlike the JSON example, this code fetches an image file instead of a JSON file. The file is also read as a Blob rather than as JSON. An image URL is constructed from the Blob and set as the `src` attribute of an image tag.

Example: Fetching Text

Let's look at another example, this time fetching text and inserting it into the page. To do this the following functions are used:

```
function readResponseAsText(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response.text();
}

function updatePage(responseAsText) {
  var container = document.querySelector('#container');
  container.textContent = responseAsText;
}
```

And the the fetch chain is updated to:

```
fetch('/pages/offline.txt')
  .then(readResponseAsText)
  .then(updatePage)
  .catch(logError);
```

In this example the fetched resource is read as text. The text is then inserted into a container element on the page.

You can see a [comprehensive list](#) of methods and properties of the Response object. For completeness, note that the methods we have used are actually methods of [Body](#), a Fetch API [mixin](#) that is implemented in the Response object.

Making HEAD and POST Requests

`Fetch()` can also receive a second optional parameter, `init`, that allows you to create custom settings for the request, such as the [request method](#), cache mode, credentials, [and more](#).

By default fetch uses the GET [method](#), which retrieves a specific resource. But other request methods can be used.

Example: HEAD Requests

HEAD requests are just like GET requests except the body of the response is empty. This kind of request can be used when all that's wanted is metadata about a file, but not all of the file's data needs to be transported.

To call an API with a HEAD request, set the method in the *init* parameter. For example:

```
fetch('/examples/example.txt', {
  method: 'HEAD'
})
```

This will make a HEAD request for *example.txt*.

Example: POST Requests

Rather than requesting resources, it's also possible to send data to an API. This might be the case, for example, if a user needs to be logged into an account. The data can be sent with POST requests. To call an API with POST, the method is again specified with the *init* parameter. This is also where the body of the request is set, which would represent the data to be sent. For example

```
fetch('someurl/login', {
  method: 'POST',
  body: 'username=john&password=12345'
})
```

Will send a username and password, as a string, to *someurl/login*.

Example: Advanced Use Case

Let's look at a final cumulative example. Consider the scenario in which a requested resource may be unavailable. Rather than break the user experience we can use fetch to request an alternative resource and deliver that instead of the original. The following code achieves this:

```
fetch('examples/example.json')
.then(function(response) {
  if (!response.ok) {
    fetch('examples/alternative.txt')
    .then(readResponseAsText)
    .then(updatePage);
  } else {
    readResponseAsJSON(response)
    .then(logResult);
  }
})
.catch(logError);
```

First, a HEAD request is made for *example.json*. If the response headers indicate that the file is not available for any reason, *alternative.txt* is fetched, read as text, and added to the page. We could test this example by altering the “examples/example.json” URL to make it incorrect. For example we could change it to “examples/non-existent.json”.

Further Reading

- MDN Documentation

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

<https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch>

- Google Developer Intro

<https://developers.google.com/web/updates/2015/03/introduction-to-fetch?hl=en>

- Blog posts

<https://davidwalsh.name/fetch>

<https://jakearchibald.com/2015/thats-so-fetch/>

Caching files with the service worker

The cache is intended to enable an offline user-experience as well as provide faster loading times for your app. This can include a custom offline page (or if you want finer control, several pages for different response error codes) which we can respond with if the response from the network errors out.

Contents:

Using the Cache API in the service worker

- [When to store resources](#)

- [Serving files from the cache](#)

- [Removing outdated caches](#)

The Cache API

- [Check for support](#)

- [Create the Cache](#)

- [Working with Data](#)

More resources

Using the Cache API in the service worker

Service Worker comes with a [Cache API](#), letting you create stores of responses keyed by request. While this API was intended for service workers it is actually exposed on the window, and can be accessed from anywhere in your scripts. The entry point is `caches`.

You are responsible for implementing how your script (service worker) handles updates to the cache. All updates to items in the cache must be explicitly requested; items will not expire and must be deleted.

When to store resources

In this section, we outline a few common patterns for caching resources. These are caching files *on install*, *on user interaction*, and *on network response*. There are a few patterns we don't cover here. See Jake Archibald's [Offline Cookbook](#) for a more complete list.

On install - caching the application shell

We can cache the HTML, CSS, JS, and any static files that make up the application shell in the “install” event of the service worker:

```
self.addEventListener('install', function(e) {
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      return cache.addAll(
        [
          '/css/bootstrap.css',
          '/css/main.css',
          '/js/bootstrap.min.js',
          '/js/jquery.min.js',
          '/offline.html'
        ]
      );
    })
  );
});
```

This event listener triggers when the service worker is first installed. It is important to note that while this event is happening, any previous version of your service worker is still running and serving pages, so the things you do here mustn't disrupt that.

`event.waitUntil` takes a promise to define the length & success of the install. If the promise rejects, the installation is considered a failure and this ServiceWorker will be abandoned (if an older version is running, it'll be left intact). `caches.open` and `cache.addAll` return promises. If any of the resources fail to fetch, the `cache.addAll` call rejects.

On user interaction

If the whole site can't be taken offline, you may allow the user to select the content they want available offline (e.g. a video on something like YouTube, an article on Wikipedia, a particular gallery on Flickr).

Give the user a "Read later" or "Save for offline" button. When it's clicked, fetch what you need from the network and put it in the cache.

```
document.querySelector('.cache-article').addEventListener('click', function(event) {
  event.preventDefault();

  var id = this.dataset.articleId;
  caches.open('mysite-article-' + id).then(function(cache) {
    fetch('/get-article-urls?id=' + id).then(function(response) {
      // /get-article-urls returns a JSON-encoded array of
      // resource URLs that a given article depends on
      return response.json();
    }).then(function(urls) {
      cache.addAll(urls);
    });
  });
});
```

The caches API is available from pages as well as service workers, meaning you don't need to involve the service worker to add things to the cache.

On network response

This approach works best for resources that frequently update such as a user's inbox or article contents. This is also useful for non-essential content such as avatars, but care is needed.

If a request doesn't match anything in the cache, get it from the network, send it to the page and add it to the cache at the same time.

If you do this for a range of URLs, such as avatars, you'll need to be careful you don't bloat the storage of your origin — if the user needs to reclaim disk space you don't want to be the prime candidate. Make sure you get rid of items in the cache you don't need any more.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return cache.match(event.request).then(function (response) {
        return response || fetch(event.request).then(function(response) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

To allow for efficient memory usage, you can only read a response/request's body once. In the code above, `.clone()` is used to create additional copies that can be read separately.

Serving files from the cache

To serve content from the cache and make your app available offline you need to intercept network requests and respond with files stored in the cache. There are several approaches to this: *cache only*, *network only*, *cache falling back to network*, *network falling back to cache*, and *cache then network*. There are a few approaches we don't cover here. See Jake Archibald's [Offline Cookbook](#) for a full list.

Cache only

You don't often need to handle this case specifically. [Cache falling back to network](#) is more often the appropriate approach.

This approach is good for any static assets that are part of your app's main code (part of that "version" of your app). You should have cached these in the install event, so you can depend on them being there.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request));
});
```

If a match isn't found in the cache, the response will look like a connection error.

Network only

This is the correct approach for things that can't be performed offline, such as analytics pings and non-GET requests. Again, you don't often need to handle this case specifically and the [cache falling back to network](#) approach will often be more appropriate.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch(event.request));
});
```

Alternatively, simply don't call `event.respondWith`, which will result in default browser behaviour.

Cache falling back to the network

If you're making your app offline-first, this is how you'll handle the majority of requests. Other patterns will be exceptions based on the incoming request.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      return response || fetch(event.request);
    })
  );
});
```

This gives you the "Cache only" behavior for things in the cache and the "Network only" behaviour for anything not-cached (which includes all non-GET requests, as they cannot be cached).

Network falling back to the cache

This is a good approach for resources that update frequently, that are not part of the "version" of the site (e.g. articles, avatars, social media timelines, game leader boards). Handling network requests this way means the online users get the most up-to-date content, but offline users get an older cached version.

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get content from the cache. This can take an extremely long time and is a frustrating user experience. See the next approach, [Cache then network](#), for a better solution.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

Here we first send the request to the network using `fetch()`, and only if it fails do we look for a response in the cache.

Cache then network

This is a good approach for resources that update frequently, that are not part of the "version" of the site (e.g. articles, avatars, social media timelines, game leader boards). This approach will get content on screen as fast as possible, but still display up-to-date content once it arrives.

This requires the page to make two requests, one to the cache, one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

Sometimes you can just replace the current data when new data arrives (e.g. game leaderboard), but that can be disruptive with larger pieces of content. Basically, don't "disappear" something the user may be reading or interacting with.

Twitter adds the new content above the old content & adjusts the scroll position so the user is uninterrupted. This is possible because Twitter mostly retains a mostly-linear order to content.

Here is the code in the page:

```
var networkDataReceived = false;

startSpinner();

// fetch fresh data
var networkUpdate = fetch('/data.json').then(function(response) {
  return response.json();
}).then(function(data) {
  networkDataReceived = true;
  updatePage(data);
});

// fetch cached data
caches.match('/data.json').then(function(response) {
  if (!response) throw Error("No data");
  return response.json();
}).then(function(data) {
  // don't overwrite newer network data
  if (!networkDataReceived) {
    updatePage(data);
  }
}).catch(function() {
  // we didn't get cached data, the network is our last hope:
  return networkUpdate;
}).catch(showErrorMessage).then(stopSpinner());
```

We are sending a request to the network and the cache. The cache will most likely respond first and, if the network data has not already been received, we update the page with the data in the response. When the network responds we update the page again with the latest information.

Here is the code in the service worker:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return fetch(event.request).then(function(response) {
        cache.put(event.request, response.clone());
        return response;
      });
    })
  );
});
```

This caches the network responses as they are fetched.

Generic fallback

If you fail to serve something from the cache and/or network you may want to provide a generic fallback. This technique is ideal for secondary imagery such as avatars, failed POST requests, "Unavailable while offline" page.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
      // However, in reality you'd have many different
      // fallbacks, depending on URL & headers.
      // Eg, a fallback silhouette image for avatars.
    })
  );
});
```

The item you fallback to is likely to be an install dependency.

Removing outdated caches

Once a new ServiceWorker has installed & a previous version isn't being used, the new one activates, and you get an `activate` event. Because the old version is out of the way, it's a good time to delete unused caches.

```
self.addEventListener('activate', function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.filter(function(cacheName) {
          // Return true if you want to remove this cache,
          // but remember that caches are shared across
          // the whole origin
        }).map(function(cacheName) {
          return caches.delete(cacheName);
        })
      );
    });
  );
});
```

During activation, other events such as fetch are put into a queue, so a long activation could potentially block page loads. Keep your activation as lean as possible, only use it for things you couldn't do while the old version was active.

The Cache API

Here we cover the Cache API properties and methods.

Check for support

We can check if the browser supports the Cache API like this:

```
if ('caches' in window) {
  // has support
}
```

Create the Cache

An origin can have multiple, named Cache objects. To create a cache or open a connection to an existing cache we use the [caches.open](#) method.

```
caches.open(cacheName)
```

This will return a promise that resolves to the cache object. `Caches.open` accepts a string which will be the name of the cache.

Working with Data

The Cache API comes with several methods that allow us to create and manipulate data in the cache. These can be categorized into general groups: create, match, and delete.

Create Data

There are three methods we can use to add data to the cache. These are `add`, `addAll`, and `put`. In practice, we will call these methods on the cache object returned from

`caches.open()`. For example:

```
caches.open('example-cache').then(function(cache) {  
  cache.add('/example-file.html');  
});
```

`Caches.open` returns the 'example-cache' Cache object which is passed to the callback in `.then`. We call the `add` method on this object to add the file to that cache.

`cache.add(request)` - The add method takes a URL, retrieves it, and adds the resulting response object to the given cache. The key for that object will be the request, so we can retrieve this response object again later by this request.

`cache.addAll(requests)` - This method is the same as add except it takes an array of URLs and adds them to the cache. If any of the files fail to be added to the cache, the whole operation will fail and none of the files will be added.

`cache.put(request, response)` - This method takes both the request and response object and adds them to the cache. This allows you to manually insert the response object. Often, you will just want to `fetch()` one or more requests, then add the result straight to your cache. In such cases you are better off just using `cache.add/cache.addAll`, as they are shorthand functions for one or more of these operations:

```
fetch(url).then(function (response) {  
  return cache.put(url, response);  
})
```

Match Data

There are a couple methods to search for specific content in the cache: `match` and `matchAll`. These can be called on the `caches` object to search through all of the existing caches, or on a specific cache returned from `caches.open()`.

`caches.match(request, options)` - This method returns a Promise that resolves to the response object associated with the first matching request in the cache or caches. It will return *undefined* if no match is found. The first parameter is the request, and the second is an optional list of options to refine the search. Here are the options as defined by MDN:

- `ignoreSearch` : A Boolean that specifies whether to ignore the query string in the url. For example, if set to true the `?value=bar` part of `http://foo.com/?value=bar` would be ignored when performing a match. It defaults to false.
- `ignoreMethod` : A Boolean that, when set to true, prevents matching operations from validating the Request http method (normally only GET and HEAD are allowed.) It defaults to false.
- `ignoreVary` : A Boolean that when set to true tells the matching operation not to perform VARY header matching — i.e. if the URL matches you will get a match regardless of whether the Response object has a VARY header. It defaults to false.
- `cacheName` : A DOMString that represents a specific cache to search within. Note that this option is ignored by `Cache.match()`.

`caches.matchAll(request, options)` - This method is the same as `match` except that returns all of the matching responses from the cache, instead of just the first. For example, if your app has cached some images contained in an image folder, we could return all images and perform some operation on them like this:

```
caches.open('example-cache').then(function(cache) {
  cache.matchAll('/images/').then(function(response) {
    response.forEach(function(element, index, array) {
      cache.delete(element);
    });
  });
});
```

Delete Data

We can delete items in the cache with `cache.delete(request, options)`. This method finds the item in the cache matching the request, deletes it, and returns a Promise that resolves to true. If it doesn't find the item, it resolves to false. It also has the same optional options parameter available to it as the `match` method.

Retrieve Keys

Finally, we can get a list of cache keys using `cache.keys(request, options)`. This will return a Promise that resolves to an array of cache keys. These will be returned in the same order they were inserted into the cache. Both parameters are optional. If nothing is passed, `cache.keys` will return all of the requests in the cache. If a request is passed, it will return all of the matching requests from the cache. The options are the same as those in the previous methods.

The keys method can also be called on the caches entry point to return the keys for the caches themselves. This allows you to purge outdated caches in one go.

More resources

Cache API

<https://developer.mozilla.org/en-US/docs/Web/API/Cache>

Using service workers

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers

The Offline Cookbook

<https://jakearchibald.com/2014/offline-cookbook/>

IndexedDB Tutorial

This tutorial guides you through the basics of the [IndexedDB API](#). We are using Jake Archibald's [IndexedDB Promised](#) library, which is very similar to the IndexedDB API, but uses promises rather than events. This simplifies the API while maintaining its structure, so anything you learn using this library can be applied to the IndexedDB API directly.

Contents:

Overview

What is IndexedDB?

IndexedDB terms

1. Setting up

2. Check for IndexedDB support

3. Opening a database

4. Working with object stores

4.1 Creating object stores

4.2 Defining primary keys

4.3 Defining indexes

5. Working with data

5.1 Creating data

5.2 Reading data

5.3 Updating data

5.4 Deleting data

6. Getting all the data

6.1 The getAll method

6.2 Using cursors

6.3 Working with ranges and indexes

7. Database Versioning Best Practices (optional)

8. Further reading

9. Appendix

Overview

What you will learn

- How to use IndexedDB to store content in the browser

What you should know

- Basic JavaScript and HTML
- [JavaScript Promises](#)
- How to clone GitHub repos and checkout branches from the command line

What you will need

- Computer with terminal/shell access
- Connection to the internet
- [Chrome](#)

What is IndexedDB?

IndexedDB is a large-scale, no-SQL storage system. It allows you to store just about anything in the user's browser. In addition to the usual search, get, and put actions, IndexedDB also supports transactions. Here is the definition on the Mozilla Developer Network (MDN):

“IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high performance searches of this data. While DOM Storage is useful for storing smaller amounts of data, it is less useful for storing larger amounts of structured data. IndexedDB provides a solution.”

Each IndexedDB database is unique to an origin (typically, this is the site domain or subdomain), meaning it cannot access or be accessed by any other origin. [Data storage limits](#) are usually quite large if they exist at all, but different browsers handle limits and data eviction differently. See the resources in the

[Further reading](#) section for more information.

IndexedDB terms

Database - This is the highest level of IndexedDB. It contains the object stores, which in turn contain the data you would like to persist. You can create multiple databases with whatever names you choose, but generally there should be one database per app.

Object store - An object store is an individual bucket to store data. You can think of object stores as similar to tables in traditional relational databases. Typically, there is one object store for each 'type' (not JavaScript data type) of data you are storing. For example, given an app that persists blog posts and user profiles, you can imagine two object stores. Unlike tables in traditional databases, actual JavaScript data types of data within the store do not need to be consistent (e.g. if there are three people in the 'people' object store, their age properties could be 53, 'twenty-five', and *unknown*).

Index - An Index is a kind of object store for organizing data in another object store (called the reference object store) by an individual property of the data. The index is used to retrieve records in the object store by this property. For example, if you're storing people, you may want to fetch them later by their name, age, or favorite animal.

Operation - An interaction with the database.

Transaction - A transaction is wrapper around an operation, or group of operations, to ensure database integrity. If one of the actions within a transaction fail, none of them are applied and the database returns to the state it was in before the transaction began. All read or write operations in IndexedDB must be part of a transaction. This allows for atomic read-modify-write operations without worrying about other threads acting on the database at the same time.

Cursor - A mechanism for iterating over multiple records in database.

1. Setting up

Clone the starter repository, change directories into it, and check out the starter code:

```
$ git clone sso://devrel/google-developer-training/pwa/ilt/indexed-db-vanilla-js-lab
$ cd indexed-db-vanilla-js-lab
$ git checkout 01-setting-up
```

This repository contains:

- **README.md**
- **index.html** is the main HTML page

- **js/main.js** is the main JavaScript file that we write our code in
- **js/idb.js** is the IndexedDB Promised library

2. Check for IndexedDB support

Replace TODO 1 in **main.js** with the following code:

main.js

```
if (!('indexedDB' in window)) {  
  console.log('IndexedDB not supported by this browser!');  
}
```

Explanation

IndexedDB is gaining traction with the major browsers, but still [isn't fully supported](#). It is important that you check for support before using it. The easiest way is to check the window object.

3. Opening a database

Replace TODO 2 in **main.js** with the following code:

main.js

```
var dbPromise = idb.open('test-db1', 1);  
  
// TODO 3 - change line above to add an object store
```

Open index.html in Chrome and open DevTools by right-clicking the page and clicking **Inspect**. Go to the **Application** tab and expand **IndexedDB**. You should see the “test-db1” database. You may need to refresh the page to see the changes display in DevTools.

Explanation

With IndexedDB you can create multiple databases with any names you choose. In general, there is just one database per app. To open a database, we use:

```
idb.open(name, version, upgradeCallback)
```

This method returns a promise that resolves to a database object. When using `idb.open`, you provide a name, version number, and an optional callback to set up the database. The callback function only triggers if a database with the same name does not already exist in the browser or if the version is greater than the existing database version.

If your database ever gets into a bad state, you can run

`indexedDB.deleteDatabase('databaseName')` in the console to remove the database and start fresh.

Solution code

```
$ git reset --hard
$ git checkout 03-open-db
```

4. Working with object stores

4.1 Creating object stores

To complete TODO 3, change the `dbPromise` code to this:

main.js

```
var dbPromise = idb.open('test-db1', 2, function(upgradeDb) {
  console.log('making a new object store');
  if (!upgradeDb.objectStoreNames.contains('firstOS')) {
    upgradeDb.createObjectStore('firstOS');
  }

  // TODO 4 - add object stores with defined primary keys

});
```

Now when you check IndexedDB in Chrome DevTools you should see the `test-db1` database containing the `firstOS` object store. You may have to refresh the browser once or twice for the object store to display in DevTools.

Notice we have changed the version number to 2 so the callback function in `idb.open` executes.

Explanation

A web app typically has one database containing one or more object stores. Object stores can be thought of as similar to tables in SQL databases and should contain objects of the same “type”. For example, for a site persisting user profiles and notes, we can imagine a 'people' object store containing 'person' objects and a 'notes' object store. A well structured IndexedDB database contains one object store for each type of data you need to persist.

To ensure database integrity, object stores can only be created and removed in the callback function in `idb.open`. The callback receives an instance of `UpgradeDB`, a special object in the IDB Promised library that is used to create object stores. We call the `createObjectStore` method on `UpgradeDB` to create the object store:

```
upgradeDb.createObjectStore('storeName', options);
```

This method takes the name of the object store as well as a parameter object that lets us define various configuration properties for the object store.

In our code, we include the callback function in `idb.open` in order to create the object store. The browser throws an error if we try to create an object store that already exists in the database so, to avoid this, we wrap the `createObjectStore` method in an `if` statement. Inside the `if` block, we call `createObjectStore` on the `UpgradeDB` object to create an object store named 'firstOS'.

Solution code

```
$ git reset --hard
$ git checkout 04-1-create-store
```

4.2 Defining primary keys

Replace TODO 4 with the following code:

```
console.log('Creating object stores');
if (!upgradeDb.objectStoreNames.contains('people')) {
  var peopleOS = upgradeDb.createObjectStore('people', {keyPath: 'email'});
}
if (!upgradeDb.objectStoreNames.contains('notes')) {
  var notesOS = upgradeDb.createObjectStore('notes', {autoIncrement: true});
}
if (!upgradeDb.objectStoreNames.contains('logs')) {
  var logsOS = upgradeDb.createObjectStore('logs', {keyPath: 'id', autoIncrement: true});
}
if (!upgradeDb.objectStoreNames.contains('store')) {
  var storeOS = upgradeDb.createObjectStore('store', {keyPath: 'name'});
}

// TODO 5 - create indexes on the object stores
```

Remember to change the database version to 3 in the second parameter of `idb.open` or the callback won't execute and the object stores will not be created:

```
var dbPromise = idb.open('test-db1', 3, function(upgradeDb) {
```

Refresh the page in the browser. In DevTools, click on each object store. You should see the key paths specified at the top of the “Key” column.

Explanation

In an object store, each item must have a way of uniquely identifying itself: the primary key. When you define object stores, you have the opportunity to define how data is uniquely identified. The primary key becomes the way to identify a particular object in the store.

There are two main ways of defining primary keys. One way is to define a `keyPath`, which is a property that must exist in each object and contain a unique value. For example, in the case of a `people` object store we might choose the `email` property as the key path. Another option is to use a key generator, such as `autoIncrement`. The key generator creates a unique value for every object added to the object store. By default, if we don't specify a key, IndexedDB creates a key and stores it separate from the data.

Choosing which method to use to define the key depends on your data. If your data has a property that is always unique, you can make it the `keyPath` to enforce this uniqueness. Otherwise, using an auto incrementing value makes sense.

In the code above, we create a `people` object store with the `email` property as the primary key, a `notes` object store with a generated key, and a `logs` object store with a generated key assigned to an `id` property.

Solution code

```
$ git reset --hard
$ git checkout 04-2-primary-keys
```

4.3 Defining indexes

Replace TODO 5 in **main.js** with the following code to create indexes on the 'people', 'notes', and 'store' object stores:

main.js

```
if (upgradeDb.objectStoreNames.contains('people')) {
  var store = upgradeDb.transaction.objectStore('people');
  store.createIndex('gender', 'gender', {unique: false});
  store.createIndex('ssn', 'ssn', {unique: true});
}
if (upgradeDb.objectStoreNames.contains('notes')) {
  var store = upgradeDb.transaction.objectStore('notes');
  store.createIndex('title', 'title', {unique: false});
}
if (upgradeDb.objectStoreNames.contains('store')) {
  var store = upgradeDb.transaction.objectStore('store');
  store.createIndex('price', 'price');
}
```

Remember to change the database version number to 4:

```
var dbPromise = idb.open('test-db1', 4, function(upgradeDb) {
```

Refresh the page and check DevTools. You should see the indexes appear under their respective object stores.

Explanation

Indexes are a kind of object store used to retrieve data from the reference object store by a particular property. An index lives inside the reference object store and contains the same data, but uses the specified property as its key path instead of the reference store's primary key. Indexes must be made when you create your object stores and can also be used to define a unique constraint on your data.

To create an index, call the [createIndex](#) method on an object store instance:


```
objectStore.createIndex('name of index', 'path', options);
```

This method creates and returns an index object. `createIndex` takes the name of the new index as the first argument, and the second argument refers to the property on the data you wish to index. The final argument allows you to define two possible options that determine how the index operates: `unique` and `multiEntry`. If `unique` is set to true, the index does not allow duplicate values for a single key. `multiEntry` determines how `createIndex` behaves when the indexed property is an array. If it's set to true, `createIndex` adds an entry in the index for each array element. Otherwise, it adds a single entry containing the array.

In our code, the `people` and `notes` object stores have indexes. To create the indexes, we first open the object store on a transaction object (more on transactions in the next section) and assign it to a variable so we can call `createIndex` on it.

Indexes are updated every time you add, edit, or delete data. More indexes mean more work for IndexedDB.

Solution code

```
$ git reset --hard
$ git checkout 04-3-indexes
```

5. Working with data

In this section, we describe how to create, read, update, and delete data. These operations are all asynchronous; the IndexedDB Promised library uses promises where the IndexedDB API uses requests. This simplifies the API: Instead of listening for events triggered by the request, we can use `.then` on the database object returned from `idb.open` to start interactions with the database.

All data operations in IndexedDB are done in a transaction. Each operation has this form:

1. Get database object
2. Open transaction on database
3. Open object store on transaction
4. Perform operation on object store

A transaction can be thought of as a safe wrapper around an operation or group of operations. If one of the actions within a transaction fail, all of the actions are rolled back. Transactions are specific to one or more object stores, which we define when we open the

transaction. They can be read-only or read and write. This signifies whether the operations inside the transaction are reading data or making a change to the database.

5.1 Creating data

Replace TODO 6 in **main.js** with the following code to add items to the ``store`` object store:

```
var name = document.getElementById('name').value;
var price = document.getElementById('price').value;
var desc = document.getElementById('desc').value;
console.log('About to add ' + name + '/' + price + '/' + desc);

dbPromise.then(function(db) {
  //Get a transaction
  var tx = db.transaction(['store'], 'readwrite');
  //Open the objectStore on the transaction object
  var store = tx.objectStore('store');
  //Define an item
  var item = {
    name: name,
    price: price,
    description: desc,
    created: new Date().getTime()
  };
  //Perform the add
  store.add(item);
  return tx.complete;
}).then(function() {
  console.log('added item to the store os!');
});
```

Optional: Replace TODO 7 in **main.js** with the code to add `notes` objects to the `notes` object store. The code for this is very similar to the code above. Some of the function is already written.

Try filling out the **Add Item** and **Add Note** forms with some dummy information and then clicking the corresponding buttons to add the object to the object store. In DevTools, open the object stores to see the objects you added. If the data doesn't display in DevTools, try refreshing the page.

Explanation

To create data, we call the `add` method on the object store and simply pass in the data we want to add. Add has an optional second argument which allows you to define the primary key for the individual object on creation, but this should only be used if you have not specified the key path in `createObjectStore`. Here is a simple example:

```
someObjectStore.add(data, optionalKey);
```

The data parameter can be data of any type - a string, number, object, array, etc. The only restriction is if the object store has a defined keypath, the data must contain this property and the value must be unique. The add method returns a promise that resolves once the object has been added to the store.

In our code, after getting the name, price, and description values from the form, we begin the process of working with the database. This process follows the four steps outlined at the beginning of this section.

First, we get the database object. We call `.then` on `dbPromise`, which resolves to the database object, and pass this object to the callback function. Because `dbPromise (idb.open)` is a promise, we can safely assume that when `.then` executes, the database is open and all object stores and indexes are ready for use.

The next step is to open a transaction by calling the `transaction` method on the database object. This method takes a list of names of object stores and indexes, which defines the scope of the transaction (in our demo it is just the `store` object store). The `transaction` method also has an optional second argument for the mode, which can be read-only or read and write. This option is read-only by default.

We can then open the `store` object store on this transaction and assign it to the `store` variable. Now, when we call `store.add`, the add happens within the transaction.

Finally, we return `tx.complete` to be sure that the `add` operation was actually carried out. The `complete` method is a promise that resolves when the transaction completes and rejects if the transaction errors. Because `add` happens within a transaction, it may be rolled back if one of the actions within the transaction fails. We only need to do this for write operations (`add`, `put`, and `delete`). For read operations, we can simply return the value we want to get.

Solution code

```
$ git reset --hard
$ git checkout 05-1-creating-data
```

5.2 Reading data

Replace TODO 8 in the `getItem` function in **main.js** with the following code to get items from the 'store' object store:

main.js

```
var key = document.getElementById('getname').value;
if (key === '') {return;}
dbPromise.then(function(db) {
  var tx = db.transaction(['store'], 'readonly');
  var store = tx.objectStore('store');
  return store.get(key);
}).then(function(val) {
  console.dir(val);

  // TODO 10 - put object values into Update Item fields on page

});
```

Optional: Replace TODO 9 in the `getNote` function in **main.js** with the code to get items from the 'notes' object store.

Try getting an object from the database by entering its primary key into the **Get Item** or the **Get Note** form and clicking the corresponding button. If you try to get an object that doesn't exist, the success handler still runs, but the result is `undefined`.

Explanation

To read data, call the `get` method on the object store. The `get` method takes the primary key of the object you want to retrieve from the store. Here is a basic example:

```
someObjectStore.get(primaryKey);
```

As with the `add`, `get` returns a promise and must happen within a transaction.

In our code, we start the operation by getting the database object and creating a transaction. Note that this time it is a read-only transaction because we are not writing anything to the database inside the transaction (i.e. using `put`, `add`, or `delete`). We then open the object store on the transaction and assign the resulting object store object to the `store` variable. Finally, we return the result of `store.get` and log this object.

Solution code

```
$ git reset --hard
$ git checkout 05-2-reading-data
```

5.3 Updating data

Replace TODO 10 in **main.js** with the code to populate the **Update Item** fields with the object values when **Get Item** is clicked:

main.js

```
document.getElementById('updateName').value = val.name;
document.getElementById('updatePrice').value = val.price;
document.getElementById('updateDesc').value = val.description;
document.getElementById('created').value = val.created;
```

The `getItem` function now populates the **Update Item** form with the retrieved values.

Replace TODO 11 in `updateItem()` in **main.js** with the following code to update the item:

main.js

```
var name = document.getElementById('updateName').value;
var price = document.getElementById('updatePrice').value;
var desc = document.getElementById('updateDesc').value;
var created = document.getElementById('created').value;
console.log('About to update ' + name + '/' + price + '/' + desc);
dbPromise.then(function(db) {
  var tx = db.transaction(['store'], 'readwrite');
  var store = tx.objectStore('store');
  var item = {
    name: name,
    price: price,
    description: desc,
    created: created
  };
  //Perform the update
  store.put(item);
  return tx.complete;
}).then(function() {
  console.log('item updated!');
});
```

Try getting an item with the **Get Item** form and changing the object values in the **Update Item** form. Click **Update Item** and refresh the page to see your changes in DevTools.

Explanation

To update data, we call the `put` method on the object store. The `put` method is very similar to the `add` method and can be used instead of `add` to create data in the object store. Like `add`, `put` takes the data and an optional primary key:

```
someObjectStore.put(data, optionalKey);
```

Again, this method returns a promise and happens inside a transaction. As with `add`, we must be careful to check `transaction.complete` if we want to be sure that the operation has actually happened.

In our code, the database interaction in `updateItem()` has the same structure as the previous examples: get the database object, create a transaction, open an object store on the transaction, perform the operation on the object store.

Solution code

```
$ git reset --hard
$ git checkout 05-3-updating-data
```

5.4 Deleting data

Replace TODO 12 in the `deleteItem` function in **main.js** with the following code:

main.js

```
var key = document.getElementById('deleteName').value;
if (key === '') {return;}
dbPromise.then(function(db) {
  var tx = db.transaction(['store'], 'readwrite');
  var store = tx.objectStore('store');
  store.delete(key);
  return tx.complete;
}).then(function() {
  console.log('Item deleted');
});
```

Enter an object key in the **Delete Item** form and click **Delete Item** to remove the object from the `store` object store. Refresh the page to see the changes in DevTools.

Explanation

To delete data, we call the `delete` method on the object store.

```
someObjectStore.delete(primaryKey);
```

This takes the `primaryKey` of the object you want to delete.

The structure of the database interaction is the same as for the other operations. Note that we again check that the whole transaction has completed by returning the `tx.complete` method, to be sure that the delete happened.

Solution code

```
$ git reset --hard
$ git checkout 05-4-deleting-data
```

6. Getting all the data

So far we have only retrieved objects from the store one at a time. There are a couple methods at our disposal if we would like to retrieve all of the data from an object store or index.

6.1 The getAll method

Replace TODO 13 in the `getAll` function in **main.js** with the following code:

main.js

```
dbPromise.then(function(db) {
  var tx = db.transaction(['store'], 'readonly');
  var store = tx.objectStore('store');
  return store.getAll();
}).then(function(items) {
  console.log('Items by name:', items);
});
```

Click **Get All** and check the console to see all of the objects in the `store` object store.

Explanation

The `getAll` method is the simplest way to retrieve all of the data from an object store.

```
someObjectStore.getAll(optionalConstraint);
```

This method returns an array containing all the objects in the object store matching the specified key or key range (see [Working with ranges and indexes](#)), or all objects in the store if no parameter is given. The objects in the array are arranged by the primary key or index key.

Solution code

```
$ git reset --hard
$ git checkout 06-1-get-all
```

6.2 Using cursors

Replace TODO 14 in the `getItems` function in `main.js` with the code to iterate through all the items in the 'store' object store and insert them into the HTML:

```
var s = '';
s += '<div class="mdl-grid">';
dbPromise.then(function(db) {
  var tx = db.transaction(['store'], 'readonly');
  var store = tx.objectStore('store');
  return store.openCursor();
}).then(function showItems(cursor) {
  if (!cursor) {return;}
  console.log('Cursored at:', cursor.value.name);

  s += '<div class="mdl-cell mdl-cell--3-col">';
  s += '<h5>Key ' + cursor.key + '</h5><p>';
  for (var field in cursor.value) {
    s += field + '=' + cursor.value[field] + '<br/>';
  }
  s += '</p></div>';

  return cursor.continue().then(showItems);
}).then(function() {
  s += '</div>';
  console.log('Done cursoring');
  document.getElementById('results').innerHTML = s;
});
```

Click **Show All** to see a list of the objects in the store displayed on the page. You can use the **Add Item** form to add data to the object store.

Explanation

A `cursor` selects each object in an object store or index one by one, allowing you to do something with the data as it is selected.

We create the `cursor` by calling the `openCursor` method on the object store:

```
someObjectStore.openCursor(optionalKeyRange, optionalDirection);
```

This method returns a promise for a cursor object representing the first object in the object store or `undefined` if there is no object. To move on to the next object in the object store, we call `cursor.continue`. This returns `undefined` if there isn't another object for the cursor to select. The `keyrange` option in the `openCursor` method limits `cursor.continue` to a subset of the objects in the store. The `direction` option can be `next` or `prev` specifying forward or backward traversal through the data.

In our code, we call the `openCursor` method on the object store and pass the `cursor` object to the callback function in `.then`. This time we give the callback function a name, `showItems`, so we can call it from inside the function and make a loop. The line `if (!cursor) return;` breaks the loop if `cursor.continue` returns `undefined` (i.e. runs out of items to select).

The cursor object contains a `key` property that represents the primary key for the item. It also contains a `value` property that represents the data.

Solution code

```
$ git reset --hard
$ git checkout 06-2-cursors
```

6.3 Working with ranges and indexes

Replace TODO 15 in the `searchItems` function in `main.js` with the code to iterate through the objects in a specified range:

```

var lower = document.getElementById('lower').value;
var upper = document.getElementById('upper').value;
if (lower == '' && upper == '') {return;}
var range;
if (lower != '' && upper != '') {
    range = IDBKeyRange.bound(lower, upper);
} else if (lower == '') {
    range = IDBKeyRange.upperBound(upper);
} else {
    range = IDBKeyRange.lowerBound(lower);
}
var s = '';
dbPromise.then(function(db) {
    var tx = db.transaction(['store'], 'readonly');
    var store = tx.objectStore('store');
    var index = store.index('price');
    return index.openCursor(range);
}).then(function showRange(cursor) {
    if (!cursor) {return;}
    console.log('Cursored at:', cursor.value.name);

    s += '<h5>Key ' + cursor.key + '</h5><p>';
    for (var field in cursor.value) {
        s += field + '=' + cursor.value[field] + '<br/>';
    }
    s += '</p>';

    return cursor.continue().then(showRange);
}).then(function() {
    console.log('Done cursoring');
    if (s === '') {s = '<p>No results.</p>';}
    document.getElementById('results').innerHTML = s;
});

```

Explanation

Indexes allow us to fetch the data in an object store by a property other than the primary key. We can create an index on any property (which becomes the keypath for the index), specify a range on that property, and get the data within the range using the `getAll` method or a `cursor`.

We define the range using the `IDBKeyRange` object. This object has four methods which are used to define the limits of the range: `upperBound`, `lowerBound`, `bound` (which means both), and `only`. As expected, the `upperBound` and `lowerBound` methods specify the upper and lower limits of the range.

```
IDBKeyRange.lowerBound(indexKey);
```

Or

```
IDBKeyRange.upperBound(indexKey);
```

They each take the index's keypath value of the item you want to specify as the upper or lower limit. The bound method is used to specify both an upper and lower limit, and takes the lower limit as the first argument.

```
IDBKeyRange.bound(lowerIndexKey, upperIndexKey);
```

The range for all of these functions is inclusive by default, but can be specified as exclusive by passing `false` in the second argument (or the third in the case of `bound`). An inclusive range includes the data at the limits of the range. An exclusive range does not.

The code we added to our file first gets the values for the limits and does a quick validation to check if the limits exist. Then it decides which method to use to limit the range based on the values. In the database interaction, we open the `price` index on the object store. This allows us to search for the items by price. We open a cursor on the index and pass in the range. The cursor returns a promise representing the first object in the range or `undefined` if there is no data within the range. `cursor.continue` returns the next object and so on until it reaches the end of the range.

Solution code

```
$ git reset --hard  
$ git checkout 06-3-ranges
```

7. Database Versioning Best Practices (optional)

Replace TODO 16 with the following code:

main.js

```
var dbPromise2 = idb.open('test-db2', 2, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      upgradeDb.createObjectStore('store', {keyPath: 'name'});
    case 1:
      var store0S = upgradeDb.transaction.objectStore('store');
      store0S.createIndex('price', 'price');

      // TODO 17 - add a case 2
  }
});
```

Optional: Create a case 2 that creates a description index on the on the 'store' object store. Remember to change the database version to 3.

Refresh the page and check DevTools. You should see a `test-db2` database containing a `store` object store and `price` index.

Explanation

The `UpgradeDB` object in `idb.open` gets a special `oldVersion` method which returns the version number of the database existing in the browser. We can pass this version number into a `switch` statement to execute blocks of code inside the upgrade callback based on the existing database version number.

In the above code, we have set the newest version of the database at 2. When this code first executes and the database doesn't yet exist in the browser, `upgradeDb.oldVersion` is 0 and the switch statement starts at case 0. In our example, this results in a `store` object store being added to the database.

Usually in `switch` statements there is a `break` after each case, but we are deliberately not doing that here. This way, if the existing database is a few versions behind (or if it doesn't exist), the code continues through the rest of the `case` blocks until it has executed all the latest changes.

Solution code

```
$ git reset --hard
$ git checkout 07-versioning
```

8. Further reading

IndexedDB Documentation

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB

<https://www.w3.org/TR/IndexedDB/>

Data storage limits

<http://www.html5rocks.com/en/tutorials/offline/quota-research/>

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria

9. Appendix

Comparison of IndexedDB API and IndexedDB Promised library

The IndexedDB Promised library sits on top of the IndexedDB API, translating its requests into promises. The overall structure is the same between the library and the API and, in general, the actual syntax for the database operations is the same and they act the same way. But there are a few divergences arising from the differences between requests and promises which we describe here.

All database interactions in the IndexedDB API are requests and have associated 'onsuccess' and 'onerror' event handlers. These are equivalent to the .then and .catch promise functions. The indexedDB.open method also gets a special event handler, onupgradeneeded, which is used to create the object stores and indexes. This is equivalent to the upgrade callback in idb.open in the Promised library. In fact, if you look through the Promised library, you will find the upgrade callback is just a convenient wrapper for the onupgradeneeded event handler.

Let's look at an example of the IndexedDB API. In this example we open a database, add an object store, and add an item to the store:

```
var db;

var openRequest = indexedDB.open('test_db', 1);

openRequest.onupgradeneeded = function(e) {
  var db = e.target.result;
  console.log('running onupgradeneeded');
  if (!db.objectStoreNames.contains('store')) {
    var storeOS = db.createObjectStore('store',
      {keyPath: 'name'});
  }
};

openRequest.onsuccess = function(e) {
  console.log('running onsuccess');
  db = e.target.result;
  addItem();
};

openRequest.onerror = function(e) {
  console.log('onerror!');
  console.dir(e);
};

function addItem() {
  var transaction = db.transaction(['store'], 'readwrite');
  var store = transaction.objectStore('store');
  var item = {
    name: 'banana',
    price: '$2.99',
    description: 'It is a purple banana!',
    created: new Date().getTime()
  };

  var request = store.add(item);

  request.onerror = function(e) {
    console.log('Error', e.target.error.name);
  };
  request.onsuccess = function(e) {
    console.log('Woot! Did it');
  };
}
```

This code does something very similar to previous examples in this tutorial except that it doesn't use the Promised library. We can see that the structure of the database interaction hasn't changed. Object stores are created on the database object in the upgrade event handler, and items are added to the object store in the same transaction sequence we've seen before. The difference is that this is all done with requests and event handlers rather than promises and promise chains.

Here is a quick reference of the differences between the IndexedDB API and the IndexedDB Promised library.

| | IndexedDB Promised | IndexedDB API |
|------------------|---|--|
| Open database | <code>idb.open(name, version, upgradeCallback)</code> | <code>indexedDB.open(name, version)</code> |
| Upgrade database | Inside <code>upgradeCallback</code> | <code>request.onupgradeneeded</code> |
| Success | <code>.then</code> | <code>request.onsuccess</code> |
| Error | <code>.catch</code> | <code>request.onerror</code> |

Designing for slow, unreliable connections

Working with Live Data in a Service Worker

Contents:

[Why IndexedDB and Cache Storage API?](#)

[Browser Support](#)

[How Much Can You Store](#)

[Getting Started](#)

[Separating Data From Application State](#)

[The Code](#)

[IndexedDB](#)

[Service Worker](#)

[IDB in a Service Worker](#)

Why IndexedDB and Cache Storage API?

The main advantages of using the [Cache Storage API](#) and [IndexedDB API](#) is that they are both supported via service workers and on the window object for most major browsers.

We've chosen to continue using Jake Archibald's [IndexedDB Promised](#) in this tutorial because we used it the tutorial for IndexedDB and it provides a promise abstraction without obscuring the amount of work you must accomplish to get work done with IndexedDB.

Once you become comfortable with IndexedDB, there are other IndexedDB wrapper libraries you can implement to get full IndexedDB support and additional SQL-like functionality.


It's important to understand the complexities inherent in the IndexedDB API (transactions, schema versioning, etc.) so that you can deal with them effectively. Some libraries hide this complexity from developers.

Both Cache and IndexedDB are asynchronous. By default, IndexedDB is event based and the Cache API is Promise based.








Debugging support for IndexedDB is available in [Chrome](#) (Application tab), Opera, [Firefox](#) (Storage Inspector) and Safari (Storage tab). Most of these are similar to the [service worker debugging tools](#)

Browser Support

The first place to check if a browser supports a specific feature is [caniuse.com](#). The table below shows the [support for IndexedDB](#):

| | | | | | | | | | | | |
|---|-----------------|---------|--------|------------------|-------|------------------|-------------------|----------------|--------------------|------------------------|------------------|
| IndexedDB  - REC | | | | | | | | | | | |
| Method of storing data client-side, allows indexed database queries. | | | | | | | | | | | |
| <div>Current aligned</div> <div>Usage relative</div> <div>Show all</div> | | | | | | | | | | | |
| IE | Edge * | Firefox | Chrome | Safari | Opera | iOS Safari * | Android Browser * | Opera Mobile * | Chrome for Android | UC Browser for Android | Samsung Internet |
| | | | 29 | | | | | | | | |
| | | | 49 | | | | 4.3 | | | | |
| | | | 50 | | | | 4.4 | | | | |
| 8 | ¹ 13 | 47 | 51 | | | ² 9.2 | 4.4.4 | | | | |
| ¹ 11 | ¹ 14 | 48 | 52 | ² 9.1 | 39 | ² 9.3 | 51 | 37 | 51 | 9.9 | 4 |
| | | 49 | 53 | 10 | 40 | | | | | | |
| | | 50 | 54 | TP | 41 | | | | | | |
| | | 51 | 55 | | | | | | | | |
| <div>Notes</div> <div>Known issues (3)</div> <div>Resources (9)</div> <div>Feedback</div> | | | | | | | | | | | |
| ¹ Partial support in IE 10 & 11 refers to a number of subfeatures not being supported . | | | | | | | | | | | |
| ² Partial support in Safari & iOS 8 & 9 refers to seriously buggy behavior as well as complete lack of support in WebViews. | | | | | | | | | | | |

Nolan Lawson offers [more detailed testing results](#) for both web workers and service workers support across browsers. To see the results of tests in service workers, scroll to the corresponding section because web workers are listed first.

| Service Workers | | | | | | | |
|-----------------|--|---|--|--|---|---|--|
| |  Chrome 51 more |  Firefox 47 more |  IE 11 more |  Edge 13 more |  Safari 9 more |  iOS Safari 9 more |  Android Chrome 39 more |
| Network | | | | | | | |
| fetch | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| WebSockets | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| XMLHttpRequest | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Storage | | | | | | | |
| Cache | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| IndexedDB | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| LocalStorage | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SessionStorage | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| WebSQL | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

Safari has [fixed several IndexedDB bugs](#) in their latest Tech Preview builds. That said, some developers have run into stability issues with Safari 10's IndexedDB implementation.

IndexedDB wrappers like [Dexie.js](#) and [localForage](#) work in Safari without extra effort (just not using IndexedDB directly). IndexedDB Promised relies on the browser's IndexedDB implementation so your application might not work as intended in desktop and iOS Safari.

Please test your app to make sure it works on your target browser. File bugs with your browser's vendor so that browser implementors and library maintainers can investigate.

How Much Can You Store

Now that we have decided how we'll store our content on the user's browser we must figure out how much data to store. There is no uniform way to measure it and different browsers have different size limitations and provide warnings to users at different points. The following table shows some browser size limits and any additional notes.

| Browser | Limitation | Notes |
|-------------------------|------------|---|
| Chrome and Opera | No limit | Storage is per origin not per API (local storage, session storage, service worker cache and IndexedDB all share the same space) |
| Firefox | No limit. | Prompts after 50 MB of data is stored |
| Mobile Safari | 50MB. | |
| Desktop Safari | No limit. | Prompts after 5MB of data is stored |
| Internet Explorer (10+) | 250MB. | Prompts after 10MB of data is stored |

Getting Started

Separating Data From Application State

Start by deciding what to store in the cache and what to add to an indexedDB database.

Follow these simple rules:

- Store resources addressed through a URL in a cache
- Store all dynamic resources in an IndexedDB database

For our sample application we'll apply the rules as follows:

- We will create an application shell and cache it when we install the service worker
- We will cache all the application assets using a cache first strategy
- We will store all the dynamic data in an IndexedDB using the service worker

The Code

We've broken the code in functions and events for both IndexedDB and Service Worker.

We'll explain them each separately and then together. Let's begin with the IndexedDB code.

IndexedDB

We created individual functions for each phase of the process to make it easier to reason through the code. Begin by defining constants and the data that is used in the example functions.

`DBNAME` represents the name of the database and `DBVERSION` represents the current version of the database. Keeping these constructs as separate values makes it easier to upgrade the database later on.

The `records` variable stores the values we'll add to the database in this demo.

```
const DBNAME = 'users';
const DBVERSION = 1;

var records = [
  {name: 'carlos',
    url: 'http://placekitten.com/g/200/200'},
  {name: 'dexter',
    url: 'http://placekitten.com/g/200/200'},
  {name: 'isabel',
    url: 'http://placekitten.com/g/200/200'},
  {name: 'mark',
    url: 'http://placekitten.com/g/200/200'}
];
```

`installDB` handles installation and, if necessary, upgrading the database. If we've never accessed the database or if the version of the database is bigger than the current version then apply all the changes to the database.

Pay special attention to the switch statements and the case clauses. The case clauses do not have a break statement. This is done on purpose. If there is more than one `case` clause we want all the case clauses to execute and update the database.

If we add a break statement, then any subsequent case clauses are not executed and the database is not fully updated.

```
function installDB() {
  var dbPromise = idb.open(DBNAME, DBVERSION, upgradeDB => {
    // Note: we don't use 'break' in this switch statement,
    // we want to fall through and get all the results.
    switch (upgradeDB.oldVersion) {
      case 0:
        upgradeDB.createObjectStore(DBNAME, {keyPath: 'name'});
      }
    })
  .then(db => console.log('DB opened!', db))
  .catch(error => {
    console.log('TRY AGAIN', error);
  });
}
```

The `addUser` function takes the records array and creates a [map](#) which we use to add each member of the records array into the database.

Begin by opening the database. This creates a promise that is used throughout the function. The first step is to create the transaction and then run the transaction on the object store. Then, create a map and store each individual records in the database.

```
function addUsers() {
  var dbPromise = idb.open(DBNAME, DBVERSION);

  dbPromise.then(db => {
    var tx = db.transaction(['users'], 'readwrite');
    var store = tx.objectStore('users');
    records.map(record => {
      store.put({
        name: record.name,
        url: record.url,
      });
    });
    return tx.complete;
  })
  .then(() => {
    console.log('added users to the users os!');
  })
  .catch(error => {
    console.log('Could not add users to database', error);
  });
}
```

The `showUsers` function provides a simple way to display the content of the database. For this example we will display all the database records to the console.

First, open the database and create a promise. Then return a transaction on the `users` database and the `users` object store. Log all records in the database to the console.

```
function showUsers() {
  var dbPromise = idb.open(DBNAME, DBVERSION);

  dbPromise.then(db => {
    return db.transaction('users')
      .objectStore('users').getAll()
      .then(allUsers => console.log(allUsers));
  })
  .catch(error => {
    console.log('Could not access users in database', error);
  });
}
```

Service Worker

The service worker code should look familiar to you. It is almost the same code we have been working with.

The first event is the install that caches the skeleton of our application as the shell to provide something quickly to the user while the rest of the content loads.

```
importScripts('scripts/idb.js');
importScripts('idb.js');

const CACHENAME = 'static-cache';
const CACHEVERSION = 1;
const EXPECTEDCACHENAMES = [CACHENAME + '-v' + CACHEVERSION];

var staticAssets = [
  'index.html',
  'styles/main.css'
];

function cacheStaticAssets(event) {
  event.waitUntil(
    caches.open(CACHENAME + '-v' + CACHEVERSION)
      .then(cache => {
        return cache.addAll(staticAssets);
      })
      .catch(error => {
        console.log('Could not cache static assets', error);
      })
  );
}
```

Activation events provide a way to clean up the service worker cache when a new version of the service worker is activated.

The service worker searches the existing caches and deletes the caches that don't match our expected caches (the ones where name and version match our variables).

```
function updateCaches(event) {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cacheNames => {
          if (EXPECTEDCACHENAMES.indexOf(CACHENAME) === -1) {
            console.log('Deleting out of date cache:', CACHENAME);
            return caches.delete(CACHENAME);
          }
        })
      );
    })
  );
}

self.addEventListener('activate', event => {
  updateCaches(event);
});
```

The service worker attempts to match the request with data from the cache and, if it's not able to, it makes a network request for the resource.


```
function respondCacheFirst(event, altUrl) {
  event.respondWith(caches.match(altUrl || event.request)
    .then(response => {
      if (response) {
        return response;
      }
      return fetch(altUrl || event.request);
    })
    .catch(() => {
      console.log('resource not available');
    })
  );
}

self.addEventListener('fetch', function(event) {
  if (event.request.url.includes('users')) {
    respondCacheFirst(event, 'index.html');
  } else {
    respondCacheFirst(event, false);
  }
});
```

IndexedDB in a Service orker

To make IndexedDB and service worker work together complete the following tasks:

- Import `indexeddb-promised` (promise-based IDB library) and `database.js` our IDB functions
- Insert the functions from `idb.js` into the service worker so it runs both APIs

Loading scripts into the service worker is done using `importScripts` to load the two required libraries into the service worker's scope synchronously. Once this is done then use the functions in `database.js` as local functions.

We have also created constants for the name and version of the database to make it easier to update the name and version for when we need to update it.

```
importScripts('scripts/idb.js');
importScripts('idb.js');

const CACHENAME = 'static-cache';
const CACHEVERSION = 1;
const EXPECTEDCACHENAMES = [CACHENAME + '-v' + CACHEVERSION];

var staticAssets = [
  'index.html',
  'styles/main.css'
];
```

For the installation event the code moves the caching code outside the event itself. In the event we run `installDB()` to install the IndexedDB database, `addUsers()` to populate the database, and `cacheStaticAssets()` to cache the assets needed for the application shell.

If you must add files to the application shell, then add the path to the new assets to the `staticAssets` variable defined earlier. This includes the files the next time the service worker is updated.

```
function cacheStaticAssets(event) {
  event.waitUntil(
    caches.open(CACHENAME + '-v' + CACHEVERSION)
      .then(cache => {
        return cache.addAll(staticAssets);
      })
      .catch(error => {
        console.log('Could not cache static assets', error);
      })
  );
}

self.addEventListener('install', event => {
  installDB();
  addUsers();
  cacheStaticAssets(event);
});
```

For the activate event the code splits the caching functionality into a separate `updateCaches` function that then calls inside the activate event.

```
function updateCaches(event) {
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return Promise.all(
        cacheNames.map(cacheNames => {
          if (EXPECTEDCACHENAMES.indexOf(CACHENAME) === -1) {
            console.log('Deleting out of date cache:', CACHENAME);
            return caches.delete(CACHENAME);
          }
        })
      );
    })
  );
}

self.addEventListener('activate', event => {
  updateCaches(event);
});
```

For our fetch event the code splits the response code into a `respondCacheFirst` function where it responds using a cache-first strategy. If the resource is in the cache it is served from there. Otherwise, it requests the resource on the network.

The event itself requests either the result of the event or `index.html` from the cache and then we run `showUsers()`. If the resource is not available in the cache, then fetch it from the network.

```
function respondCacheFirst(event, altUrl) {
  event.respondWith(caches.match(altUrl || event.request)
    .then(response => {
      if (response) {
        return response;
      }
      return fetch(altUrl || event.request);
    })
    .catch(() => {
      console.log('resource not available');
    })
  );
}

self.addEventListener('fetch', event => {
  if (event.request.url.includes('users')) {
    respondCacheFirst(event, 'index.html');
    showUsers();
  } else {
    respondCacheFirst(event, false);
  }
});
```

Tooling and Automation

Contents:

[Introduction](#)

[Setting up your project](#)

[Making Gulp do things](#)

Introduction

In this session we will discuss tools and automation to make your development workflow easier. We will concentrate on the Gulp task runner and tasks that will make creating a service worker and testing for performance easier. Because of scope and time constraints we will not discuss how to build general purpose workflows but provide a good starting point.

Why an automated build system?

Modern web development is full of repetitive tasks like concatenating and minimizing CSS and JavaScript, compressing images, converting SASS to CSS and many others. Gulp and related tools automate these tasks making it easier to configure and run these repetitive tasks so developers no longer need to complete them manually.

What is Gulp?

[Gulp](#) is a streaming task runner. It creates a stream of files to work with and then pipes the stream to different tasks that modify the input files and, finally, to a destination where Gulp will write the modified files.

Gulp provides basic tooling and infrastructure for you to build your projects. You can read more in the Tooling and Automation chapter, but you don't have to be a Gulp expert for this project.

For more information about Gulp check the Gulp [Getting Started guide](#). To get an idea of what Gulp can do check the [list of Gulp recipes](#) on Github

Setting up your project

For this tutorial we'll assume that you're starting a brand new project. If you're working on your own project not all instructions will be applicable as is.

Directory Structure

Create your project directory and navigate into it

```
$ mkdir my-project  
$ cd my-project
```

Once inside `my-project` make directories for images, Javascript and CSS. Make sure that you run

```
$ mkdir img  
$ mkdir js  
$ mkdir css
```

With these commands we've created the directory structure for my-app. If need to create additional directories you can follow the same process. Make sure there are no spaces in the directory names.

NPM Init

Before we can install Gulp and other plugins we need to initialize our application as a Node.js application. We do this by running:

```
$ npm init
```

This command starts the NPM initialization. You will be asked a series of questions, shown below. It's ok to leave everything blank for now; we can go back and edit it later. When you're done answering these questions NPM will use your answers to generate a `package.json` file. Later we'll use this file to store informations about the plugins we want to work with.

```
name: (project)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/caraya/code/images/package.json:
```

Installing Gulp

Now we can finally gulp

There are two steps to get Gulp running for this application.

- Install `gulp-cli` globally so we can use gulp without having to declare the full path to the application. This wrapper will also allow different versions of Gulp in different projects
- Install `gulp` in your project. This is the full version of Gulp and it is specific to your project

Install `gulp-cli` Globally

Make sure you're still in your `my-app` directory. Then, to install Gulp globally, type:

```
npm i -g gulp-cli
```

Install Gulp in your project directory

To install gulp in your project directory:

```
npm install --save-dev gulp
```

`--save-dev` will write the name of the plugin (gulp in this case) and the version of the plugin (latest since we didn't specify one) to the package.json file

Making Gulp do things

Installing Plugins

Tasks in Node.js require a plugin to be installed and configured before the task will run successfully. The installation process is the same for plugins to be used in tasks than it is to install gulp in your project directory. For example, to install aPlugin as part of our project dependencies the command is:

```
$ npm install --save-dev aPlugin
```

The `--save-dev` parameter tells NPM to write the package name and version (latest in this case since we didn't specify one) to the `package.json` file. If we push this to a Version Control Repository where we would not normally store our `node_modules` directory we can install the packages by running `npm install`

Defining tasks

Gulp exposes the following methods as its API:

- `gulp.task`: used to define a task for Gulp to run
- `gulp.src`: provides a list matching the glob parameter
- `gulp.dest`: destination of a task. Re-emits all data passed to it so you continue the pipe for your task. Folders that don't exist will be created.
- `gulp.watch`: watches a given file or folder and performs the indicated task when its target changes.

At its simplest a Gulp task can just log output to console, like the hello task shown below:

```
gulp.task('hello', function() {  
  console.log('Hello World');  
});
```

A more common kind of Gulp task takes the source you want to work from using `gulp.src` (1), processes through one or more commands using Node's pipe functionality (2) and finally outputs the result to the destination folder, creating it if doesn't exists (3)

```
gulp.task('task-name', function() {  
  return gulp.src('source-files') // 1  
    .pipe(aGulpPlugin()) // 2  
    .pipe(gulp.dest('destination')) // 3  
}
```


The structure of the Gulpfile

This is a minimal Gulpfile read in order to execute the indicated commands. It illustrates the main sections of a Gulpfile and it'll help understand what to do in your own work.

```
var gulp = require('gulp'); //1
var path = require('path');
var swPrecache = require('sw-precache');

var paths = { //2
  src: 'app/'
};

gulp.task('task-name', function() { // 3
  return gulp.src('source-files')
    .pipe(aGulpPlugin())
    .pipe(gulp.dest('destination'))
});
```

In the first section we `require` the plugins we will use and assign them to variables. This tells Gulp what plugins we want to use and it makes the available to our code at the same time.

The second block contains any variables that we'll use in our tasks.

The last section contains one or more tasks that will actually accomplish our goals. In this case it's a generic task to demonstrate the structure of the gulpfile.

Some examples

copy-content

The quickest way to copy files in Gulp is to create a copy task. This task will match all the files with the indicated extensions inside the app folder and all its children and copy them to a dist folder for further processing and distribution.

```
gulp.task('copy-content', function() {
  gulp.src('app/**/*.{html,css,jpg,png,gif,svg}')
    .pipe(gulp.dest('./dist'));
});
```

generate-service-worker

Sw-precache programmatically builds with the give root directory and list of file extensions to work with. This Gulp task will create a service worker with all the files of the specified formats and importing the indicated scripts.

```
gulp.task('generate-service-worker', function(callback) {
  swPrecache.write(path.join(paths.src, 'service-worker.js'), {
    staticFileGlobs: [
      paths.src + 'index.html',
      paths.src + 'js/main.js',
      paths.src + 'css/main.css',
      paths.src + 'img/**/*.{svg,png,jpg,gif}'
    ],
    importScripts: [
      './js/sw-toolbox.js',
      './js/toolbox-scripts.js'
    ],
    stripPrefix: paths.src
  }, callback);
});
```

psi:mobile

This task will test the specified site against [Pagespeed Insight](#) using a mobile profile.

```
gulp.task('psi:mobile', function() {
  //var key = '';
  var site = 'CHANGE_TO_YOUR_SITE';
  return psi(site, {
    // key: key
    nokey: 'true',
    strategy: 'mobile',
    format: 'cli'
  }).then(function(data) {
    console.log('Speed score: ' + data.ruleGroups.SPEED.score);
    console.log('Usability score: ' + data.ruleGroups.USABILITY.score);
    console.log(data.pageStats);
  });
});
```

SASS

This task shows how to pipe the output of one plugin into the input of a follow on task.

```
gulp.task('sass', function() {  
  return gulp.src('./sass/**/*.scss')  
    .pipe(sourcemaps.init())  
    .pipe(sass().on('error', sass.logError))  
    .pipe(autoprefixer())  
    .pipe(sourcemaps.write())  
    .pipe(gulp.dest('./css'));  
});
```

Using Code Generation for Easier Caching (sw-precache, sw-toolbox)

Contents:

[Introduction](#)

[Getting started](#)

[Creating routes with sw-toolbox](#)

[Running sw-precache from command line](#)

[Using sw-precache and sw-toolbox in a Gulp build file](#)

Introduction

In this workshop we'll cover sw-precache and sw-toolbox, two packages created by Google to automate the creation of service workers and to make creation of custom caching routes easier. We'll explore the tools and how to use sw-precache from the command line, how to build routes using sw-toolbox and how to integrate both tools into a Gulp-based workflow.

What you will learn

By the end of this workshop you should be able to:

- Generate service workers using sw-precache and sw-toolbook.
- Create sw-toolbox routes matching popular caching strategies.
- Integrate sw-precache into your Gulp build process.
- Add the service worker to your index page.

What you should know

- Create semantic HTML, and CSS
- Be familiar with Javascript, ES6 Promises and arrow functions
- Be comfortable with your operating system's command line

- Be able to run Node and NPM from the command line
- Be comfortable building and running Gulp build files

What you should have installed in your system

- Node (and NPM)
- Gulp installed globally

Getting started

Before we can build content with sw-precache and sw-toolbox we need to create the project the tasks will run under.

As discussed in Tooling and Automation, we first create the directory where the application will live, in this case `my-app`.

Creating routes with sw-toolbox

Sw-toolbox simplifies the creation and customization of caching rules and strategies. At its simplest sw-toolbox uses the concepts of routes and methods (representing HTTP verbs) that match content in your application. For example to cache `/my-app/index.html` using the network first caching strategy you'd use:

```
toolbox.router.get('/my-app/index.html', global.toolbox.networkFirst);
```

To get all images with an extension of png or jpg regardless of directory depth, you can use:

```
toolbox.router.get('img/**/*.{png,jpg}', global.toolbox.cacheFirst);
```

Express-like Routing

If you're familiar with Express.js you can create routes with a syntax similar to its routing syntax. Combined with the ability to cache content from different origins it gives you flexibility in configuring your caches.

In the example below we listen for all files and if the origin of the file ends with `googleapis.com` we use the cache first strategy.

```
toolbox.router.get('/:(.*)', global.toolbox.cacheFirst, {  
  origin: /\.googleapis\.com$/  
});
```

The following example matches any content that ends with fly (like butterfly or dragonfly) using the cache first strategy

```
toolbox.router.get('/*.*fly$/', global.toolbox.cacheFirst);
```

Regular Expression Routing

If you're more comfortable working with [regular expressions](#) you can use them to define routes. Pass a [RegExp](#) object as the first parameter of the route. This RegExp will be matched against the full URL (request and path) to determine if the route applies to the request. This matching makes for easier cross origin routing since the origin and the path are matched at the same time without having to specify an origin like we did with Express style routes.

This route will handle all requests that end with index.html

```
toolbox.router.get(/index.html$/, function(request) {  
  return new Response('Handled a request for ' + request.url);  
});
```

While this route will handle all requests that begin with <https://api.flickr.com/>:

```
toolbox.router.post(/^https://api.flickr.com\/$/, global.toolbox.networkFirst);
```

Cache Control

sw-precache also gives us the ability to control the cache and its characteristics. The Express route below handles requests ending with googleapis.com using the cache first strategy. In addition to handling the origin we customize the cache itself.

- We give it a name (googleapis)
- We give it a maximum size of 10 items (indicated by the maxEntries parameter)
- We set the content to expire 86400 seconds (24 hours)

```
toolbox.router.get('/:(.*)', global.toolbox.cacheFirst, {
  cache: {
    name: 'googleapis',
    maxEntries: 10,
    maxAgeSeconds: 86400
  },
  origin: /\.\googleapis\.com$/
});
```

Running sw-precache from command line

There may be times when you want to test the result of using sw-precache and don't want to have to change your build system for every version of the experiment. sw-precache allows you to use the tool from the command line.

We will first create a `sw-precache-config.json` file with our sw-precache configuration. In this example `staticFileGlobs` indicates the path to each file we want to precache and `stripPrefix` tells sw-precache what part of each file path to remove.

```
{
  "staticFileGlobs": [
    "app/index.html",
    "app/js/main.js",
    "app/css/main.css",
    "app/img/**/*.{svg,png,jpg,gif}"
  ],
  "stripPrefix": "app/"
}
```

Once the sw-precache configuration is ready we run it with the following command:

```
$ sw-precache --config=path/to/sw-precache-config.json --verbose
```

Using sw-precache and sw-toolbox in a Gulp build file

Rather than create a service worker manually we'll use sw-precache and sw-toolbox to build a production-ready service worker. sw-precache automatically builds a service worker script and import scripts as needed. sw-toolbox provides abstractions for caching strategies.

sw-precache is available as a command line tool but, rather than use command line tools that are platform dependent, we will use sw-precache as a Gulp plugin and manually build our service worker additions using sw-toolbox.

For this tutorial we'll break the process into the following steps

- Integrating sw-precache into our Gulp build system
- Creating our routes with sw-toolbox
- Experimenting with the results

Integrating sw-precache into a Gulp build system

Because Gulp must know what plugins to use we register each plugin to a variable by assigning a require statement for each of the plugins we use. In this case we add `path` and `swPrecache` to the list of plugins at the top of the build file:

```
var path = require('path');
var swPrecache = require('sw-precache');
```

Create an array of variables to reflect the paths used in our application. Right now the code we only uses only a `src` child to represent the root directory of the application.

```
var paths = {
  src: 'app/'
};
```

The service-worker task takes the static elements we want to cache for our application shell. We tell the gulp task what files we want to cache in the `staticFileGlobs` section (marked as **1**) and what files to import in the `importScripts` section (marked as **2**.) The scripts imported at this point are automatically cached.

Strip the root prefix (**3**) to turn the element into a relative path.

If we do not strip the source prefix we would be working with absolute paths. Absolute paths are written different in Windows (`c:\directory`) and Macintosh/Linux (`/directory`).


```

gulp.task('generate-service-worker', function(callback) {
  swPrecache.write(path.join(paths.src, 'service-worker.js'), {
    //1
    staticFileGlobs: [
      paths.src + 'index.html',
      paths.src + 'js/main.js',
      paths.src + 'css/main.css',
      paths.src + 'img/**/*.{svg,png,jpg,gif}'
    ],
    // 2
    importScripts: [
      paths.src + '/js/sw-toolbox.js',
      paths.src + '/js/toolbox-scripts.js'
    ],
    // 3
    stripPrefix: paths.src
  }, callback);
});

```

Creating our routes with sw-toolbox

The generated service worker imported two scripts, sw-toolbox and toolbox-scripts. The first file contains sw-toolbox functionality and the second file (toolbox-scripts) contains the custom routes we will use in our application.

It's important to consider all of the caching strategies and find the right balance between speed and data freshness for each of your data sources. Use the following table to determine which caching strategy is most appropriate for the dynamic resources that populate your app shell.

Table of Common Caching Strategies

| Strategy | The service worker ... | Best strategy for | Corresponding sw-toolbox handler |
|-------------------------------|--|---|----------------------------------|
| Cache first, Network fallback | Loads the local (cached) HTML and JavaScript first, if possible, bypassing the network. If cached content is not available, then the service worker returns a response from the network instead. | When dealing with remote resources that are very unlikely to change, such as static images. | toolbox.cacheFirst |
| | | When data must | |

| | | | |
|----------------------------------|---|--|----------------------|
| Network first, Cache fallback | Checks the network first for a response and, if successful, returns current data to the page. If the network request fails, then the service worker returns the cached entry instead. | be as fresh as possible, such as a real-time API response, but you still want to display something as a fallback when the network is unavailable. | toolbox.networkFirst |
| Cache/network race | Fires the same request to the network and the cache simultaneously. In most cases, the cached data loads first and that is returned directly to the page. Meanwhile, the network response updates the previously cached entry. The cache updates keep the cached data relatively fresh. The updates occur in the background and do not block rendering of the cached content. | When content is updated frequently, such as for articles, social media timelines, and game leaderboards. It can also be useful when chasing performance on devices with slow disk access where getting resources from the network might be quicker than pulling data from cache. | toolbox.fastest |
| Network only | Only checks the network. There is no going to the cache for data. If the network fails, then the request fails. | When only fresh data can be displayed on your site. | toolbox.networkOnly |
| Cache only | The data is cached during the install event so you can depend on the data being there. | When displaying static data on your site. | toolbox.cacheOnly |

The script below implements some sw-toolbox strategies to cache different parts of an application. The numbers in the examples refer to the comments in the code

Example 1 uses a cache first strategy to fetch content from the `googleapis.com` domain. It will store up to 20 matches in the googleapis cache.

Example 2 matches all PNG and JPG images (those files that end with png or jpg) and stores them in the `images-cache` cache. When more than 50 items are stored in the cache the oldest items will be removed.

Example 3 works with local videos in mp4 and a network only strategy. We don't want large files to bloat the cache so we will only play the videos when we are online.

Example 4 matches all files from Youtube and Vimeo and uses the network only strategy. When working with external video sources we not only have to worry about cache size but also about potential copyright issues.

Example 5 presents our default route. If the request did not match any prior routes it will match this one and run with a cache first strategy.

```
(function(global) {
  'use strict';

  // 1
  global.toolbox.router.get('/(.*)', global.toolbox.cacheFirst, {
    cache: {
      name: 'googleapis',
      maxEntries: 20,
    },
    origin: /\.googleapis\.com$/
  });

  // 2
  global.toolbox.router.get(/\.(?:png|gif|jpg)$/ , global.toolbox.cacheFirst, {
    cache: {
      name: imagesCacheName,
      maxEntries: 50
    }
  });

  // SPECIAL CASES:
  // 3
  self.addEventListener('fetch', function(event) {
    if (event.request.headers.get('accept').includes('video/mp4')) {
      // respond with a network only request for the requested object
      event.respondWith(global.toolbox.networkOnly(event.request));
    }
    // you can add additional synchronous checks based on event.request.
  });

  // 4
  global.toolbox.router.get('(.+)', global.toolbox.networkOnly, {
    origin: /\.(:youtube|vimeo)\.com$/
  });

  // 5
  global.toolbox.router.get('/*', global.toolbox.cacheFirst);
})(self);
```

Experimenting with the results

```
$ git clone https://github.com/caraya/clean-blog-demo
$ cd clean-blog-demo
```

In the code above change make the following changes:

Experiment 1: Clone the repository, change to the directory and open `js/toolbox-scripts.js`

Experiment 2: In example 2, change the caching strategy to **toolbox.networkFirst**. In dev tools throttle the network so it simulates offline. What happens if you haven't visited the page before? What happens if you have?

Experiment 3: In example 5, change the caching strategy to **toolbox.cacheFirst** and add a limit of 5 items to the cache (hint: look at examples 1 and 2 for an idea of how to limit the size of the cache.)

Introduction to Push Notifications

This tutorial describes the basic concepts involved in creating Push Notifications.

Contents:

[What Are Push Notifications?](#)

[Push Notification Terms](#)

[Understanding Push Notifications on the Web](#)

[Notification API](#)

[Designing with the Future in Mind](#)

[Push API](#)

[Best Practices](#)

[More Resources](#)

What Are Push Notifications?

A notification is a message that pops up on the user's device. Notifications can be triggered locally by an open application, or they can be "pushed" from the server to the user even when the app is not running. They allow your users to opt-in to timely updates and allow you to effectively reengage users with customized content.

Push Notifications are assembled using two APIs: the [Notification API](#) and the [Push API](#). The Notification API allows the app to display system notifications to the user. The Push API allows a service worker to handle Push Messages from a server, even while the app is not active.

The Notification and Push API's are built on top of the [Service Worker API](#), which responds to push message events in the background and relays them to your application. All notifications must be created with a service worker.

Service workers require secure origins so testing Push Notifications requires [running a local server](#).

Push Notification Terms

Notification - a message displayed to the user outside of the app's normal UI (i.e. the browser)

Push Message - a message sent from the server to the client

Push Notification - a Notification created in response to a Push Message

Notification API - an interface used to configure and display Notifications to the user

Push API - an interface that gives service workers the ability to receive Push Messages

Web Push - an informal term referring to the process or components involved in the process of pushing messages from a server to a client on the web

Push Service - a system for routing Push Messages from a server to a client. Each browser implements its own push service.

Web Push Protocol - describes how an application server or user agent interacts with a push service

Understanding Push Notifications on the Web

Notifications and Push Messaging allow your app to extend beyond the browser, and give you a new way to interact with users.

Notifications can remain on the user's device past the initial interaction with a page. For example, when your app raises a notification the user might interact with it much later (notifications could live for weeks on the user's device). Leaving the web page open for the entire time the notification is alive would be a massive waste of resources. However, because notifications are paired with a service worker, it can listen for notification interactions in the background without using resources. Only when the user interacts with the notification, by clicking or closing it, does the service worker wake up for a brief time to handle the interaction.

Notifications are an incredibly powerful way to engage with the user. They can do simple things such as alert the user to an important event, displaying a small icon and a small piece of text which the user can then click on to open up your site. You can also integrate action buttons in the notification so that the user can interact with your site or application without needing to go back to your web page.

Notification API

The [Notification API](#) allows us to display notifications to the user. It is incredibly powerful and simple to use. When possible, it uses the same mechanisms a native app would use, giving a completely native look and feel.

The API is split into two core areas. The *Invocation API* controls how to make your notification appear, including styling and vibration. The *Interaction API* controls what happens when the user engages with the notification.

Request Permission

Before we can create a notification we need to get permission from the user. Below is the code to prompt the user to allow notifications. This goes in the app's main JavaScript file.

main.js

```
Notification.requestPermission(function(status) {  
    console.log('Notification permission status:', status);  
});
```

We call the [requestPermission](#) method on the global Notification object. This displays a pop-up message from the browser requesting permission to allow notifications. The user's response is stored along with your app, so calling this again returns the user's last choice. Once the user grants permission, the app can display notifications.

Display a Notification

We can show a notification from the app's main script with the [showNotification](#) method. Here is an example:

main.js

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    navigator.serviceWorker.getRegistration().then(function(reg) {
      reg.showNotification('Hello world!');
    });
  }
}
```

Notice the `showNotification` method is called on the service worker registration object. This creates the notification on the active service worker, so that events triggered by interactions with the notification are heard by the service worker.

Add Notification Options

The `showNotification` method has an optional second argument for configuring the notification. Below is an example demonstrating some of the available options. A complete explanation of each option is in the [showNotification reference on MDN](#).

main.js

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    navigator.serviceWorker.getRegistration().then(function(reg) {
      var options = {
        body: 'Here is a notification body!',
        icon: 'images/example.png',
        vibrate: [100, 50, 100],
        data: {
          dateOfArrival: Date.now(),
          primaryKey: 1
        }
      };
      reg.showNotification('Hello world!', options);
    });
  }
}
```

- The “body” option adds a main description to the notification and should give the user enough information to decide how to act on it.
- The “icon” option attaches an image to make the notification more visually appealing, but also more relevant to the user. For example, if it is a message from their friend you might include an image of the user’s avatar.
- The “vibrate” option specifies a vibration pattern for a phone receiving the notification. In

our example, a phone would vibrate for 100 milliseconds, pause for 50 milliseconds, and then vibrate again for 100 milliseconds.

- The “data” option attaches custom data to the notification, so that the service worker can retrieve it when the user interacts with the notification. For instance, adding a unique “id” or “key” option to the data allows us to determine which notification was clicked when the service worker handles the click event.

Here is a [useful tool](#) that allows you to experiment with all of the different Notification options.

Add Actions to the Notification

Simple notifications display information to the user and handle basic interactions when they are clicked. This is a massive step forward for the web, but it's a little bit basic. We can add contextually relevant actions to the notification so the user can quickly interact with our site or service without opening a page. For example:

main.js

```
function displayNotification() {
  if (Notification.permission == 'granted') {
    navigator.serviceWorker.getRegistration().then(function(reg) {
      var options = {
        body: 'Here is a notification body!',
        icon: 'images/example.png',
        vibrate: [100, 50, 100],
        data: {
          dateOfArrival: Date.now(),
          primaryKey: 1
        },
        actions: [
          {action: 'explore', title: 'Explore this new world',
            icon: 'images/checkmark.png'},
          {action: 'close', title: 'Close notification',
            icon: 'images/xmark.png'},
        ]
      };
      reg.showNotification('Hello world!', options);
    });
  }
}
```

To create a notification with a set of custom actions, we add an actions array inside the notification options object. This array contains a set of objects that define the action buttons to show to the user.

Actions can have an identifier string, a title containing text to be shown to the user, and an icon containing the location of an image to be displayed next to the action.

Listen for Events

Displaying a notification was the first step. Now we need to handle user interactions in the service worker. Once the user has seen your notification they can either *dismiss* it or *action* it, normally via a tap on the notification.

The “notificationclose” Event

If the user dismisses the notification via a direct action on the notification (such as a swipe in Android) then a “notificationclose” event is raised inside the service worker. Note, if the user dismisses all notifications then, to save resources, an event is not raised in the service worker.

This event is important because it allows you to understand how the user is interacting with your notifications. You might, for instance, log to your analytics that the user closed the notification. Or, you might use the event to synchronise your database and avoid renotifying the user of the same event.

Here is an example of a ‘notificationclose’ event listener in the service worker:

serviceworker.js

```
self.addEventListener('notificationclose', function(e) {
  var notification = e.notification;
  var primaryKey = notification.data.primaryKey;

  console.log('Closed notification: ' + primaryKey);
});
```

We can access the [notification object](#) from the event object. From there we can get the data and decide how to respond. In the example, we are getting the primaryKey property defined earlier and logging it to the console.

The “notificationclick” event

The most important thing to do is handle when the user clicks on the notification. The click triggers a “notificationclick” event inside your service worker.

Let’s look at the code to handle the click event in the service worker.

serviceworker.js

```
self.addEventListener('notificationclick', function(e) {
  var notification = e.notification;
  var primaryKey = notification.data.primaryKey;
  var action = e.action;

  if (action === 'close') {
    notification.close();
  } else {
    clients.openWindow('http://www.google.com');
    notification.close();
  }
});
```

We can determine what action button the user pressed by inspecting the action property on the event object.

When the user clicks on the notification the usual expectation is to be taken directly to the area of your site that provides them with more information about what was presented in the notification. You can open a new window by calling the `clients.openWindow` function in your “notificationclick” handler and passing in the URL where you want the user to navigate.

Notice we check for the “close” action first and handle the “explore” action in an `else` block. This is a best-practice as not every platform supports action buttons, and not every platform displays all your actions. Additionally, centralizing everything in a `notificationclick` event allows us to provide a default experience that works everywhere.

Designing with the Future in Mind

We have been using Chrome and Android for our examples so far, but what can you do in other browsers? The [notification spec](#) is constantly evolving with the authors and browser vendors constantly adding new features and increasing the possibilities of what you can do with the API. Note that:

- Not all browsers implement the Notification API to the same level
- Not all operating systems support the same features for notifications

We need to build our sites and apps defensively, yet progressively so that our experiences work well everywhere. Let's look at what we can do to create a smooth experience.

Check for Support

The web is not yet at the point where we can build apps that depend on web notifications. When possible, design for a lack of notification support and layer on notifications.

The simplest thing to do is detect if the ability to send notifications is available and, if it is, enable that part of our experience:

main.js

```
if('Notification' in window && navigator.serviceWorker) {  
  // Display the UI to let the user toggle notifications  
}
```

Here are some things you can do when the user's browser doesn't support the Notification API. For instance, you can:

- Offer a simple inline “notification” on your web page. This works well when the user has the page open.
- Integrate with another service such as an SMS provider or email provider to provide timely alerts to the user.

Check for Permission

Always check for permission to use the Notification API. It is important to keep checking that permission has been granted because the status may change:

main.js

```
if(Notification.permission === "granted") {  
  /* do our magic */  
}  
else if (Notification.permission === "blocked") {  
  /* the user has previously denied push. Can't reprompt. */  
}  
else {  
  /* show a prompt to the user */  
}
```

Cross-Platform Differences

The action buttons and images differ significantly across platforms. For example, native notifications on Mac OS X only display two actions, and those actions are not directly visible to the user. On Android, the buttons are visible.

You can check the maximum number of action buttons that can be displayed by calling `Notification.maxActions`. Do this at the time you create notifications so you can adapt your notifications. You can also check this in the `notificationclick` handler in the service worker so that you can determine what course of action to take.

A good practice is to assume that the system cannot support any actions other than the notification click. This means that you must design your notification to handle the default click and have it perform the default use-case for your notification. Then you can layer on some customization for each action.

Decide if the context of each action requires buttons to be grouped together. If you have a binary choice such as “Accept” and “Decline”, but can only display one button, you might decide to not display buttons.

Finally, treat every attribute of the notification other than “title” and “body” as optional and at the discretion of the browser and the operating system to use. For example, don't rely on images being present in the notification. If you are using the image to display contextual information (such as a photo of a person) then ensure that you have this data visible in the title or the body so the user can quickly ascertain the importance of the notification if the image is not visible.

Action buttons can have images, however not every system can display them (for example on the Mac). So ensure that button labels are clear and concise.

Many systems can't vibrate, or they won't vibrate if the user has their device volume muted so don't rely on vibrations to notify the user.

Push API

We have learned how to create a notification and display it to the user directly from a web page. This is great if you want to create notifications when the page is open, but what if the page isn't open? How can you create a notification that alerts the user of some important information?

Native apps have been able to do this for a long time using a technology called “Push Messaging”. Now, we have the ability to do the same on the web through the Push API.

Push and Notification are different, but complementary functions: a push is the action of the server supplying message information to a service worker; a notification is the action of the service worker sending the information to a user.

Push messaging allows developers to engage users by providing timely and customized content outside the context of the web page. It is one of the most critical API's to come to the web, giving users the ability to engage with web experiences even when the browser is closed, without the need for a native app install.

This section describes how to set up a service so that you can send notifications to the user, in three parts:

- Manage the user subscription for push notifications from the client
- Send your first message without data
- Send your first message with attached data.

There are many moving parts to Web Push that involve client-side management and also server management. We are primarily going to focus on the client-side aspects of Web Push as it relates to Push Notifications (i.e. the Push API). We'll leave the server-side details to commercial services that we will provide links to.

How Web Push Works

Let's walk through a quick overview of how the Web Push works.

Each browser manages push notifications through their own system, called a "push service". When the user grants permission for Push on your site, you subscribe the app to the browser's internal push service. This creates a special subscription object that contains the "endpoint URL" of the push service, which is different for each browser, and a public key. You send your push messages to this URL, encrypted with the public key, and the push service sends it to the right client.

How does the push service know which client to send the message to? The endpoint URL contains a unique identifier. The identifier is used to route the message that you send to the correct device, and ultimately when processed by the browser, it identifies which service worker should handle the request.

The identifier is opaque. As a developer you can't determine any personal data from it and it is not stable so it can't be used to track users.

Because each browser has its own messaging service, the user must have confidence that the operator of the messaging service cannot intercept or manipulate the push messages. HTTPS is one part of the solution. It ensures that the communication channel between your

server and the message router is secure, and from the message router to the user is also secure.

However, HTTPS doesn't ensure that the message router itself is secure. We must be sure that the data sent from your server to the client is not tampered with or directly inspected by any third party. You must encrypt the message payload on your server.

Below is a summary of the process of sending and receiving a Push Message and then displaying a Push Notification.

On the server:

1. Generate the data that we want to send to the user
2. Encrypt the data with the user public key
3. Send the data to the endpoint URL with a payload of encrypted data.

The message is routed to the user's device. This wakes up the browser that finds the correct service worker and invokes a "push" event. Now, on the client:

1. Receive the "push message" event
2. Perform some custom logic with the push message
3. Show a notification

That completes the path from server push to user notification. Let's dive into each part.

Handling the Push Event

Let's see how the service workers handles Push Messages. The service worker both receives the Push Message and creates the notification.

When a browser that supports push messages receives the message, it sends a "push" event to the service worker. We can create a "push" event listener in the service worker to handle the message:

serviceworker.js

```
self.addEventListener('push', function(e) {
  var options = {
    body: 'This notification was generated from a push!',
    icon: 'images/example.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: '2'
    },
    actions: [
      {action: 'explore', title: 'Explore this new world',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'Close',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification('Hello world!', options)
  );
});
```

This code is very similar to what we have covered before in this tutorial, the difference being that this is happening inside the service worker in response to a push event, instead of in the app's main script.

Another important difference is that the `showNotification` method is wrapped in an [e.waitUntil method](#). This extends the lifetime of the push event until the `showNotification` Promise resolves.

Subscribing to Push Notifications

Before we can send a push message we must first subscribe to a push service.

The subscription is a critical piece of the process to send push messages. It tells us, the developer, where we should send our push messages. Remember, each browser will provide their own push message infrastructure. The subscription also details *who* we should be sending the messages to (the endpoint) and how we should encrypt the data so that it is delivered securely to the user (the public key).

It is your job to take this subscription object and store it somewhere on your system. For instance, you might store it in a database attached to a user object. In our examples, we are going to log results out to a console.

First, we need to check if we already have a subscription object and update the UI accordingly.

main.js

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.ready.then(function(reg) {
    reg.pushManager.getSubscription().then(function(sub) {
      if (sub === undefined) {
        // Update UI to ask user to register for Push
        console.log('Not subscribed to push service!');
      } else {
        // We have a subscription, update the database
        console.log('Subscription object: ', sub);
      }
    });
  });
}
```

We should perform this check whenever the user accesses our app because it is possible for subscription objects to change during their lifetime. We need to make sure that it is synchronised with our server. If there is no subscription object we should update our UI to ask the user if they would like receive notifications.

Assume the user enabled notifications. Now we can subscribe to the push service.

main.js

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('sw.js').then(function(reg) {
    console.log('Service Worker Registered!', reg);
    reg.pushManager.subscribe({
      userVisibleOnly: true
    }).then(function(sub) {
      console.log('Endpoint URL: ', sub.endpoint);
    }).catch(function(e) {
      if (Notification.permission === 'denied') {
        console.warn('Permission for notifications was denied');
      } else {
        console.error('Unable to subscribe to push', e);
      }
    });
  });
}).catch(function(err) {
  console.log('Service Worker registration failed: ', err);
});
} else {
  console.log('This browser does not support service workers');
}
```

Here we call the [subscribe method](#) on the [pushManager](#) and log the subscription object to the console.

Notice we are passing a flag named `userVisibleOnly` to the subscribe method. By setting this to `true`, the browser will ensure that every incoming message has a matching notification.

In the current implementation of Chrome, whenever we receive a push message and we don't have our site visible in the browser we must display a notification, we can't do it silently without the user knowing. If we don't display a notification the browser will automatically create one to let the user know that your web page is doing work in the background. If the user doesn't accept the permission request or there's another error, the Promise rejects.

We add a catch clause to handle this and now we can check the permission property on the Notification global object to understand why we can't display notifications.

The Web Push Protocol

Let's look at how to send a push message to the browser using the [Web Push Protocol](#).

The Web Push protocol is the formal standard for sending push messages destined for the browser. It describes the structure and flow of how you create your Push message, encrypt it, and how you send it to a Push messaging platform. The protocol isolates you from the details of which messaging platform and browser the user has.

The actual Web Push protocol is complex but we don't need to understand the details. The browser automatically takes care of subscribing the user with the push service. Our job as developers is to take the subscription token, extract the URL, and send our message there.

Sending a Push Message Using Firebase Cloud Messaging

Chrome doesn't yet use the standard Web Push Protocol. It currently uses [Firebase Cloud Messaging](#) (FCM) for this, though the eventual goal is for Chrome and FCM to support the Web Push Protocol. FCM is the successor of Google Cloud Messaging (GCM) and supports the same functionality as GCM and more.

To use Firebase Cloud Messaging, you need to set up a project on [Firebase](#). Here's how:

1. In the [Firebase console](#), select **Create New Project**.
2. Supply a project name and click **Create Project**.

3. Select the gear icon next to your project name at top left, and select **Project Settings**.
4. Select the **Cloud Messaging** tab. You can find your server key and sender ID in this page. Save these values.

For Chrome to route FCM messages to the correct service worker, it needs to know the Sender ID. Supply this by adding a `gcm_sender_id` property to the `manifest.json` file. For example, the manifest could look like this:

```
{
  "name": "Push Notifications codelab",
  "gcm_sender_id": "370072803732"
}
```

To get FCM to push a notification to your web client, we need to send FCM a request that includes the following:

- The public Server key that you created earlier. FCM matches this with the Sender ID of the project in Firebase and in your `manifest.json`.
- A `Content-Type` header: `application/json`.
- An array of subscription IDs. The subscription ID is the last part of the subscription endpoint URL, after the last forward slash (/)

A production site or app normally sets up a service to interact with FCM from your server. There is some sample code for doing just that in [Push Notifications on the Open Web](#).

We can test push messaging in our app using [cURL](#). We can send an empty message, called a “tickle”, to the push service, then the push service sends a message to the browser. If the notification displays, then we have done everything correctly and our app is ready to push messages from the server.

Sending a Message Using cURL

The cURL command that sends a request to FCM to issue a push message looks like this:

```
curl --header "Authorization: key=SERVER KEY GOES HERE" --header "Content-Type: application/json" https://android.googleapis.com/gcm/send -d "{\"registration_ids\": [\"SUBSCRIPTION ID GOES HERE\"]}"
```

For example:

```
curl --header "Authorization: key=AIZA9bGxkYD_az3GCzhTRY4bqtnCFboLISiezHgLLc3oxgBb-VYBkoknwmXGMzV0ocKIFYHHXq0kiGjiilLNJJ6sJk-QqFv6ag_gdF6ZOHixClnyZtyc0w80I1UgF3EJosiBYVwy3SFv\" --header "Content-Type: application/json" https://android.googleapis.com/gcm/send -d "{\"registration_ids\": [\"fmQw197hny0:APA91bGxkYD_az3GCzhTRY4bqtnCFboLISiezHgLLc3oxgBb-VYBkoknwmXGMzV0ocKIFYHHXq0kiGjiilLNJJ6sJk-QqFv6ag_gdF6ZOHixClnyZtyc0w80I1UgF3EJosiBYVwy3SFv\"]}"
```

If a message sends successfully, a notification displays on the screen. In the command line you should see something like this:

```
{"multicast_id":7007152374450936118,"success":1,"failure":0,"canonical_ids":0,"results":[{"message_id":"0:1457450438558507%1fbab450f9fd7ecd"}]}
```

Working with Data Payloads

As you have seen, it is relatively easy to get a push message to the user. However, so far the notifications we have sent have been empty. Chrome and Firefox support the ability to deliver data to your service worker via the push message.

Receiving Data in the Service Worker

Let's first look at what changes are needed in the service worker to pull the data out of the Push Message.

serviceworker.js

```
self.addEventListener('push', function(e) {
  if (e.data) {
    var data = e.data.json();
    var title = data.title;
    console.log(data);
  } else {
    var title = 'Push message no payload';
  }

  var options = {
    body: 'Sent from a server',
    icon: 'images/notification-flat.png',
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: 1
    },
    actions: [
      {action: 'explore', title: 'Explore this new world',
        icon: 'images/checkmark.png'},
      {action: 'close', title: 'I don't want any of this',
        icon: 'images/xmark.png'},
    ]
  };
  e.waitUntil(
    self.registration.showNotification(title, options)
  );
});
```

When we receive a push notification with a payload, the data is available directly on the event object. This data can be of any type, and you can access the data as a JSON result, a BLOG, a typed array or raw text.

In this example, we're getting the data payload as JSON and extracting the title from it.

Sending from the Server

In this section, we cover how to send a push message from the server.

We use the encryption key information from the subscription object to encrypt the payload on the server before we send it. The details are relatively complex, and as with anything related to encryption it's easier to use an actively developed library than to write your own code.

We are using Mozilla's [web-push library](#) for Node.js. This handles both encryption and the web push protocol, so that sending a push message from a Node.js server is simple:

```
webpush.sendNotification(endpoint, options)
```

The first argument is the endpoint URL from the subscription object. The second argument is an options object that contains the keys and payload. See the example below.

While we definitely recommend using a library, this is a new feature and there are many popular languages that don't yet have any libraries. Here is a list of some available [web-push libraries](#) for various languages. If you do need to implement encryption manually, use Peter Beverloo's [encryption verifier](#).

We now have all the client side components in place, so let's create a simple server side script using Node.js that imports the web-push library and then uses our subscription object to send a message to the client.

To install web-push in the app from the command window we run:

```
$ npm install web-push
```

The node script looks like this:

node/main.js

```
var webPush = require('web-push');

var subscription = JSON.parse('{ "endpoint": "https://android.googleapis.com/gcm/send/f1LsxxkKphfQ:APA91bFUx7ja4BK4JVrNgVjpg1cs9lGSGI6IMNL4mQ3Xe6mDGxvt_C_gItKYJI9CAx5i_Ss6cmDxdWZoLyhS2RJhkcv7LeE6hki0sK6oBzbyifvKCdUYU7ADIRBiYNxIVpLIYeZ8kq_A", "keys": { "p256dh": "BLC4xRzKlKORKWlbdgFaBrrPK3ydWAHo4M0gs0i1oEKgPpWC5cW80CzVrOQRv-1npXRWk8udnW3oYhIO4475rds=", "auth": "5I2Bu2oKdyy9CwL8QVF0NQ==" } }');

webPush.sendNotification(subscription.endpoint, {
  userPublicKey: subscription.keys.p256dh,
  userAuth: subscription.keys.auth,
  payload: JSON.stringify({
    'title': 'Push Notification'
  })
})
.then(r => console.log('Success', r))
.catch(e => console.log('Error', e));
```

This example parses the subscription object and passes it into the `sendNotification` method. The first argument is the endpoint where the message will be sent. The second argument is an object containing the `userPublicKey`, `userAuth`, and `payload`.

Identifying Your Service with VAPID Auth

The Web Push Protocol has been designed to respect the user's privacy by keeping users anonymous and not requiring strong authentication between your app and the push service. This presents some challenges: if the push service can't identify the app sending the messages, how can the service contact the developer if there are problems?

In Chrome, this problem is moot because Chrome requires us to have a project on Firebase or the Google Developer Console to push messages.

For Mozilla Firefox, the answer is to have the publisher optionally identify themselves without an explicit subscription to the push service. How do we prevent everyone from saying that they're something popular like Facebook? [The Voluntary Application Server Identification for Web Push \(VAPID\) protocol](#) was drafted to answer that question.

To authenticate your app with VAPID you create a special token called a JSON Web Token (JWT).

The JWT contains three properties called a "claim". The claim has:

- An Audience attribute, the name of your site that is used to ensure the push message is going to the correct site
- A Subscriber property, who the push service can contact if something fails
- An Expiration time value, how long this token is valid for (max of 24 hours)

You sign the claim with a private key to create a signature that you attach to your HTTPS request to the push server. Because the signature contains your private key, you need to also attach your Public Key along with the request. When you send these two valid headers the server can verify that it was your service creating the requests and not someone else.

Here's what a cURL request using VAPID could look like:

```
curl "https://updates.push.services.mozilla.com/wpush/v1/gAAAAABXmk...dyR" --request POST --header "TTL: 60" --header "Content-Length: 0" --header "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhb...7mc6xi57kWTkj-mYxw" --header "Crypto-Key: p256ecdsa=BDd3_hVL9fZi9Ybo2UUz...cF3foIXRf6_LFTeZaNndIo"
```

We've added two new headers to the request. An Authorization header that is the [HMAC signature](#) of our JWT token, and the Crypto-key which is our public key that is used to determine if the JWT is valid.

This process can be quite cumbersome but the web-push NPM module takes care of the details. Here is an example of using VAPID in a node script with the web-push library:

node/main.js

```
var webPush = require('web-push');

var serviceKeys = webPush.generateVAPIDKeys();

var subscription = JSON.parse('{ "endpoint": "https://android.googleapis.com/gcm/send/f1LsxxkPhfQ:APA91bFUx7ja4BK4JVrNgVjpg1cs9lGSGI6IMNL4mQ3Xe6mDGxvt_C_gItKYJI9CAx5i_Ss6cmDxdWZoLyhS2RJhkc7LeE6hki0sK6oBzbyifvKCdUYU7ADIRBiYNxIVpLIYeZ8kq_A",
"keys": { "p256dh": "BLC4xRzKlKORKWlbdgFaBrrPK3ydWAHo4M0gs0i1oEKgPpWC5cw80CzVr0QRv-1npXRWk8udnW3oYhIO4475rds=", "auth": "5I2Bu2oKdyy9CwL8QVF0NQ==" } }');

webPush.setGCMAPIKey('AIZA5yD1JcZ8WM1vTtH6Y0tXq_Pnuw4jgj_92yg');

webPush.sendNotification(subscription.endpoint, {
  userPublicKey: subscription.keys.p256dh,
  userAuth: subscription.keys.auth,
  vapid: {
    subject: 'nsearle@google.com',
    publicKey: serviceKeys.publicKey,
    privateKey: serviceKeys.privateKey
  },
  payload: JSON.stringify({
    'title': 'First push message!',
    'body': 'From a server!',
    'primaryKey': '-push-notification'
  })
})
.then(function(r) {
  console.log('Pushed message successfully!', r);
})
.catch(function(e) {
  console.log('Error', e);
});
```

Note that in this example we are generating the VAPID keys with

`webPush.generateVAPIDKeys()` every time we push a message. This process is very resource intensive. In a real-world app do this once and store the keys somewhere safe.

We add the VAPID object to the options parameter in the `sendNotification` method. It contains the subject (your email address) and the generated Public and Private keys.

Best Practices

While it has been relatively simple to get notifications up and running, making an experience that users really value is trickier. There are many edge cases to consider when building an experience that works well.

This lesson runs through many of the best practices of implementing push notifications.

Using Notifications Wisely

Notifications done correctly will delight your users. Notifications should be timely, precise, and relevant. By following these three rules, you can keep your users happy and increase their return visits.

Timely - The notification should display at the right time. Use notifications primarily for time-sensitive events, especially if these synchronous events involve other people.

For instance, an incoming chat is a real-time and synchronous form of communication: Another user actively waiting on your response. Calendar events are another good example of when to use a notification to grab the user's attention, because the event is imminent and often involves other people.

Precise - Offer enough information so that the user can make a decision without clicking through to the web page.

In particular, you should:

- Keep it short,
- Make the title and content specific
- Keep important information on the top and to the left
- Make the desired action the most prominent

Because users often give notifications a quick glance, you can make their lives easier with a well-chosen title, description, and icon. If possible, make the icon match the context of the notification so users can identify it without reading.

Relevant - Make notifications relevant to the user's needs. If the user receives too many unimportant notifications, they might turn them all off. So keep it personal. If it's a chat notification, tell them who it's from.

Avoid notifying the user of information that is not directed specifically at them, or information that is not truly time-sensitive. For instance, the asynchronous and undirected updates flowing through a social network generally do not warrant a real-time interruption.

Don't create a notification if the relevant new information is currently on screen. Instead, use the UI of the application itself to notify the user of new information directly in context. For instance, a chat application should not create system notifications while the user is actively chatting with another user.

Whatever you do, don't use notifications for advertising either third-party ads or ads for your native app.

Design Notifications According to Best Principles

This section provides best practices to make your notifications timely, precise, and relevant.

To show notifications we must prompt the user to give permission. But when is the best time to do that?

We can look at people's experience with geolocation and its prompts. Geolocation is a great API, however many sites immediately prompt the user for their location the instant that the page loads. This is a poor time to ask. The user has no context for how to make an informed decision about allowing access to this powerful piece of data, and users frequently deny this request. Acceptance rates for this API can be as low as six percent.

However, when the user is presented with the prompt after an action such as clicking on a locator icon, the acceptance rate skyrockets.

The same applies to the Push Notifications API. If you ask the user for permission to send push notifications when they first land on your site, they might dismiss it. Once they have denied permission, they can't be asked again. Case studies show that when a user has context when the prompt is shown, they are more likely to grant permission.

There are some interaction patterns that are good times to ask for permission to show notifications:

- When the user is configuring their communication settings, you can offer push notifications as one of the options.
- After the user completes a critical action that needs to deliver timely and relevant updates to the user. For example, the user might have purchased an item from your site, you can offer to notify the user of delivery updates.
- When the user returns to your site they are likely to be a satisfied user and more understanding of the value of your service.

Another pattern that works well is to offer a very subtle promotion area on the screen that asks the user if they would like to enable notifications. Be careful not to distract too much from your site's main content. Clearly explain the benefits of what notifications offer the users.

Managing the Number of Notifications

One issue is how to manage lots of notifications in your site. It is not unreasonable for a site to send the user lots of important and relevant updates. However, if you don't build them correctly, they can become unmanageable for the user.

A simple technique is to group messages that are contextually relevant into one notification. For example, if you are building a social app, group notifications by sender and show one per person. If you have an auction site, group notifications by the item being bid on.

The notification object includes a `tag` attribute that is the grouping key. When creating a notification with a tag and there is already a notification with the same tag visible to the user, the system automatically replaces it without creating a new notification. For example:

serviceworker.js

```
registration.showNotification('New message', {body: 'New Message!', tag: 'id1' });
```

Not giving a second cue is intentional, to avoid annoying the user with continued beeps, whistles and vibrations. To override this and continue to notify the user, set the `renotify` attribute to true in the notification options object:

serviceworker.js

```
registration.showNotification('2 new messages', {  
  body: '2 new Messages!',  
  tag: 'id1',  
  renotify: true  
});
```

When to Show Notifications

If the user is already using your application there is no need to display a notification. You can manage this logic on the server, but it is easier to do it in the push handler inside your service worker:

serviceworker.js

```
self.addEventListener('push', function(e) {
  clients.matchAll().then(function(c) {
    if (c.length == 0) {
      // Show notification
      e.waitUntil(
        self.registration.showNotification('Push notification')
      );
    } else {
      // Send a message to the page to update the UI
      console.log('Application is already open!');
    }
  });
});
```

The `clients` global in the service worker lists all of the active push clients on this machine.

If there are no clients active, the user must be in another app. We should show a notification in this case.

If there *are* active clients it means that the user has your site open in one or more windows. The best practice is to relay the message to each of those windows.

Hiding Notifications on Page Focus

When a user clicks on a notification we want to close all the other notifications that have been raised by your site. In most cases you will be sending the user to the same page that has easy access to the other data that is held in the notifications.

We can clear all notifications by iterating over the notifications returned from the `getNotifications` method on our service worker registration and closing each:

serviceworker.js

```
self.addEventListener('notificationclick', function(e) {
  // do your notification magic

  // close all notifications
  self.registration.getNotifications().then(function(notifications) {
    notifications.forEach(function(notification) {
      notification.close();
    });
  });
});
```

If you don't want to clear all of the notifications, you can filter based on the tag by passing it into `getNotifications` :

serviceworker.js

```
self.addEventListener('notificationclick', function(e) {  
  // do your notification magic  
  
  // close all notifications with tag of 'id1'  
  var options = {tag: 'id1'};  
  self.registration.getNotifications(options).then(function(notifications) {  
    notifications.forEach(function(notification) {  
      notification.close();  
    });  
  });  
});
```

You could also filter out the notifications directly inside the promise returned from `getNotifications` . For example, there might be some custom data attached to the notification that you could use as your filter-criteria.

Notifications and Tabs

Window management on the web can often be difficult. Think about when you would want to open a new window, or just navigate to the current open tab.

When the user clicks on the notification, you can get a list of all the open clients. You can decide which one to reuse.

serviceworker.js

```
self.addEventListener('notificationclick', function(e) {
  clients.matchAll().then(function(clis) {
    var client = clis.find(function(c) {
      c.visibilityState === 'visible';
    });
    if (client !== undefined) {
      client.navigate('some_url');
      client.focus();
    } else {
      // there are no visible windows. Open one.
      clients.openWindow('some_url');
      notification.close();
    }
  });
});
```

The code above looks for the first window with `visibilityState` set to `visible`. If one is found then it navigates that client to the correct URL and focuses the window. If a window that suits our needs is not found, then it opens a new window.

Managing Notifications at the Server

So far, we've been assuming the user is around to see your notifications. But consider the following scenario:

1. The user's mobile device is offline
2. Your site sends user's mobile device a message for something time sensitive, such as breaking news or a calendar reminder
3. The user turns the mobile device on a day later. It now receives the push message.

That scenario is a poor experience for the user. The notification is neither timely or relevant. Our site shouldn't display the notification because it's out of date.

You can use the `time_to_live` (TTL) parameter, supported in both HTTP and XMPP requests, to specify the maximum lifespan of a message. The value of this parameter must be a duration from 0 to 2,419,200 seconds, and it corresponds to the maximum period of time for which FCM stores and tries to deliver the message. Requests that don't contain this field default to the maximum period of 4 weeks. If the user doesn't get the message after the TTL, it is not delivered.

Another advantage of specifying the lifespan of a message is that GCM never throttles messages with a `time_to_live` value of 0 seconds. In other words, GCM guarantees best effort for messages that must be delivered "now or never." Keep in mind that a

`time_to_live` value of 0 means messages that can't be delivered immediately are discarded. However, because such messages are never stored, this provides the best latency for sending notifications.

Here is an example of a JSON-formatted request that includes TTL:

```
{
  "collapse_key" : "demo",
  "delay_while_idle" : true,
  "to" : "xyz",
  "data" : {
    "key1" : "value1",
    "key2" : "value2",
  },
  "time_to_live" : 3
},
```

Managing Redundant Notifications

What should you do if the user can get the same notification in multiple places, such as in a chat app?

Consider the following:

1. The user's mobile device is unavailable,
2. The site sends a message to the user's phone announcing a new email,
3. The user checks email on desktop and reads the new email.
4. Now when the user turns their phone on, the push message is received but there is no new email to show (and no visible notification on the site).

We don't want display redundant notifications that have been removed elsewhere, but currently you do have to display a notification to the user.

There are a number of options available to solve this:

1. **Show the old notification**, even if it's no longer relevant. This looks like a small glitch of the clients being out of sync
2. **Handle the push message without triggering a notification**. Chrome allows sites to *very occasionally* handle a push message without triggering a notification. If this case occurs extremely rarely it may be OK to do nothing.
3. Ignore the message from the server and **replace the notification with a fallback** to be displayed if no other is available. For example, rather than display the information from an email the user has already read you could say "We've updated your inbox."

More Resources

Your first push notifications

<https://developers.google.com/web/fundamentals/getting-started/push-notifications/?hl=en>

When to use push notifications

<https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/?hl=en>

Simple Push Demo

<https://gauntface.github.io/simple-push-demo/>

Notification generator

<https://tests.peter.sh/notification-generator/#actions=8>

Messaging concepts and options

<https://developers.google.com/cloud-messaging/concept-options#ttl>

Web-push documentation

<https://www.npmjs.com/package/web-push>

Web Push Libraries

<https://github.com/web-push-libs>

Encryption

<https://developers.google.com/web/updates/2016/03/web-push-encryption?hl=en>

Firebase Cloud Messaging

<https://firebase.google.com/docs/cloud-messaging/>

<https://firebase.google.com/docs/cloud-messaging/chrome/client>

Introduction to Payment Request API

Contents:

[About Web Payments](#)

[Introduction to the Payment Request API](#)

[How Payment Request Processing Works](#)

[Using the Payment Request API](#)

[Resources](#)

The PaymentRequest API improves mobile web checkout (shopping cart) and accepts credit cards electronically (and eventually a number of other payment services and solutions in the wild) using a payment request API. The PaymentRequest API allows merchants to easily collect payment information with minimal integration. The API assumes that browsers facilitate the payment flow between merchants and mobile device users.

Note: The [PaymentRequest API](#) is very new and still subject to developmental changes. Google tracks updates on [this page](#). Please keep checking back. Also on this page is [a shim](#) that you can embed on your site to paper over API differences for two major Chrome versions.

About Web Payments

For mobile device users, making purchases on the web, particularly on mobile devices, can be a frustrating experience. Every web site has its own flow and its own validation rules, and most require us to manually type in the same set of information over and over again. Likewise, it is difficult and time consuming for developers to create good checkout flows that support various payment schemes.

For businesses, checkout can be a complicated process to develop and complete. That's why it is worthwhile investing in capabilities such as the [Payment Request API](#) and enhanced autofill to assist your users with the task of accurately filling in forms.

Mobile users are likely to abandon online purchase forms that are user-intensive, difficult to use, slow to load and refresh, and require multiple steps to complete. This is because two primary components of online payments – security and convenience – often work at cross-

purposes, where more of one typically means less of the other.

Any system that improves or solves one or more of those problems is a welcome change. We've found that forms and payments are completed 25% more when autofill is available, increasing odds for conversion. We started solving the problem already with [Autofill](#), but now we're talking about a more comprehensive solution called the Payment Request API.

Introduction to the Payment Request API

Starting with Chrome 53 for Android, the Payment Request API is a new open-web approach:

- For developers to minimize the need to fill out checkout forms and improve user's payment experience from the ground up. This API follows the recommendations in the [W3C Payment Request API specification](#) published by the [Web Payments Working Group](#).
- For users to check out, make a payment, or fill in forms with minimal use of the mobile device keyboard.

The API uses securely cached data to facilitate payment interaction between the user and the merchant's site. The API also gets the data necessary to process transactions as quickly as possible.

The Payment Request API is a standards-based way to enable checkout on the web that:

- Provides a native user interface for users to select or add a payment method, a shipping address, a shipping option, and contact information in an easy, fast, and secure way.
- Provides standardized (JavaScript) APIs for developers to obtain user's payment preferences in a consistent format.
- Brings secure, tokenized payments to the web (browser as middleman) using secure origin, HTTPS.
- Always returns a payment credential that a merchant can use to get paid (credit card, push payment, token, etc).
- Is designed so that additional functionality can be added in, depending on your particular product requirements (shipping information, email, and phone number collection).

Goals of the Payment Request API

The Payment Request API is an open and cross-browser standard that replaces traditional checkout flows by allowing merchants to request and accept any payment in a single API call. The API allows the web page to exchange information with the browser while the user is providing input, before approving or denying a payment request.

It vastly improves user workflow during the purchase process, providing a more consistent user experience and enabling web merchants to easily leverage disparate payment methods. The Payment Request API is neither a new payment method, nor does it integrate directly with payment processors. Rather, it is a process layer whose goals are to:

- Allow the browser act as intermediary among merchants, users, and payment methods
- Standardize the payment communication flow as much as possible
- Seamlessly support different secure payment methods
- Eventually work on any browser, device, or platform, including mobile devices and otherwise (initial support is available only for Android Chrome but other third-party solutions will be supported in the future)

Best of all, the browser acts as an intermediary, storing all the information necessary for a fast checkout so users can just confirm and pay with a single tap or click.

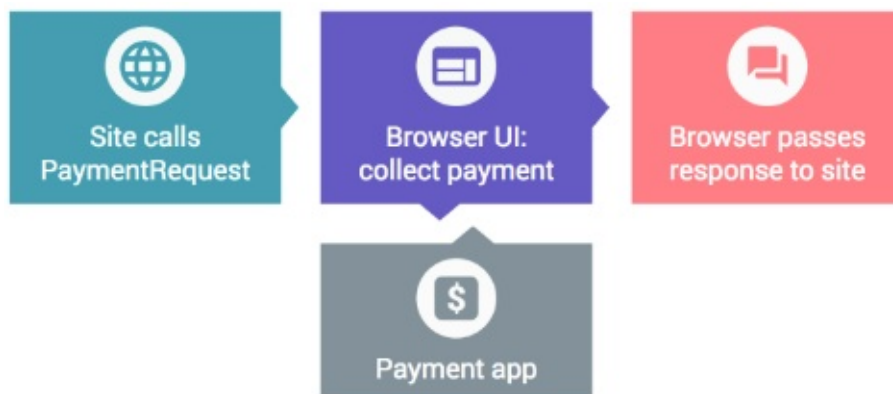
Demos

Payment Request demos are available at these URLs:

- Demo of Web Payment Request API: [g.co/payment request api](https://g.co/paymentrequestapi)
- Demo: <https://emerald-eon.appspot.com/>
- Simple demos and sample code:
<https://googlechrome.github.io/samples/paymentrequest/>

How Payment Request Processing Works

Using the Payment Request API, the transaction process is made as seamless as possible for both users and merchants.



The process begins when the merchant site creates a new `PaymentRequest` and passes to the browser all the information required to make the purchase: the amount to be charged, what currency they expect payment in, and what payment methods are accepted by the site. The browser determines compatibility between the accepted payment methods for the site and the methods the user has installed on the target device.

The browser then presents the payments UI to the user, who selects a payment method and authorizes the transaction. A payment method can be as straightforward as a credit card that is already stored by the browser, or as esoteric as a third-party system (for example, Android Pay or PayPal) written specifically to deliver payments to the site (this functionality is coming soon). After the user authorizes the transaction, all the necessary payment details are sent directly back to the site. For example, for a credit card payment, the site gets a card number, a cardholder name, an expiration date, and a CVC.

`PaymentRequest` can also be extended to return additional information, such as shipping addresses and options, payer email, and payer phone. This allows you to get all the information you need to finalize a payment without ever showing the user a checkout form.

From the user's perspective, all the previously tedious interaction—request, authorization, payment, and result—now takes place in a single step; from the web site's perspective, it requires only a single JavaScript API call. From the payment method's perspective, there is no process change whatsoever.

Using the Payment Request API

For a high-level introduction and FAQ for the Payment Request API:

<https://developers.google.com/web/updates/2016/07/payment-request>

For a step-by-step tutorial that describes implementing the Payment Request API, see the Integration Guide at: <https://developers.google.com/web/fundamentals/primers/payment-request/>

Resources

To learn more about Payment Request API, see these documents and resources:

- [Official specification](#)
- [Payment Request API integration guide](#)
- [Demo](#)
- [Simple demos and sample code](#)

Google Analytics

Contents:

[What is Google Analytics?](#)

[Creating an Account](#)

[Add analytics to your site](#)

[How analytics.js works](#)

[The Google Analytics dashboard](#)

[Debugging](#)

[Custom events](#)

[Offline analytics](#)

[Additional resources](#)

What is Google Analytics?

Google Analytics is a service that collects, processes, and reports data about an application's use patterns and performance. Adding Google Analytics to a web application enables the collection of data like visitor traffic, user agent, user's location, etc. This data is sent to Google Analytics servers where it is processed. The processed data is then reported to the developer and/or application owner. This information is accessible from the Google Analytics web interface and through a [reporting API](#).

Why use it?

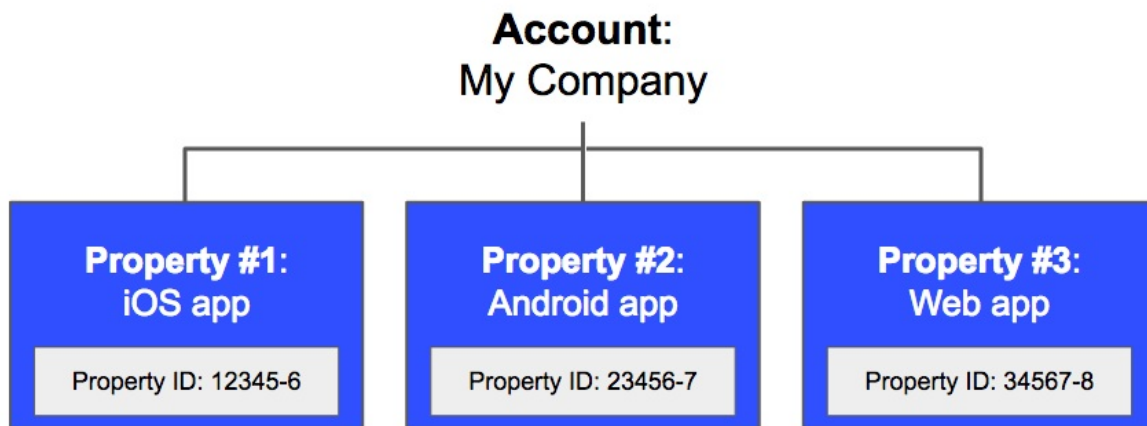
Using analytics tools gives developers valuable information on their application such as:

- User's geographic location, user agent, screen resolution, and language
- How long users spend on pages, how often they visit pages, and the order that pages are viewed
- What times users are visiting the site and from where they arrived at the site

Google Analytics is free, relatively simple to integrate, and customizable.

Creating an Account

Google Analytics requires creating a [Google Analytics account](#). An account has [properties](#), that represent individual collections of data. These properties have tracking ID's (also called property ID's) that identify them to Google Analytics. For example, an account might represent a company. One property in that account might represent the company's web site, while another property might represent the company's iOS app.



If you only have one app, the simplest scenario is to create a single Google Analytics account, and add a single property to that account. That property can represent your app.

A Google Analytics account can be created from the [Google Analytics home page](#). From that page, you need to sign into your [Google/Gmail account](#), and then select "Sign up".

Note: If you already have Google Analytics as part of your Google/Gmail account, select the Admin tab. Under "account", select your current Google Analytics account and choose "create new account". A single Google/Gmail account can have multiple Google Analytics accounts.

The sign up screen looks like this:

New Account

What would you like to track? _____

| | |
|---------|------------|
| Website | Mobile app |
|---------|------------|

Setting up your account _____

Account Name

Accounts are the top-most level of organization and contain one or more tracking IDs.

Setting up your property _____

Website Name

Website URL

| | |
|-----------|-----------------------------------|
| http:// ▾ | Example: http://www.mywebsite.com |
|-----------|-----------------------------------|

Industry Category

Reporting Time Zone

| | |
|-----------------|----------------------------|
| United States ▾ | (GMT-08:00) Pacific Time ▾ |
|-----------------|----------------------------|

What would you like to track?

Websites and mobile apps implement Google Analytics differently. This document assumes a web app is being used. For mobile apps, see [analytics for mobile applications](#).

Setting up your account

This is where you can set the name for your account, for example “PWA Training” or “Company X”.

Setting up your property

A property must be associated with a website (for web apps). The website name can be whatever you want, for example “GA Code Lab Site” or “My New App”. The website URL should be the URL where your app is hosted.

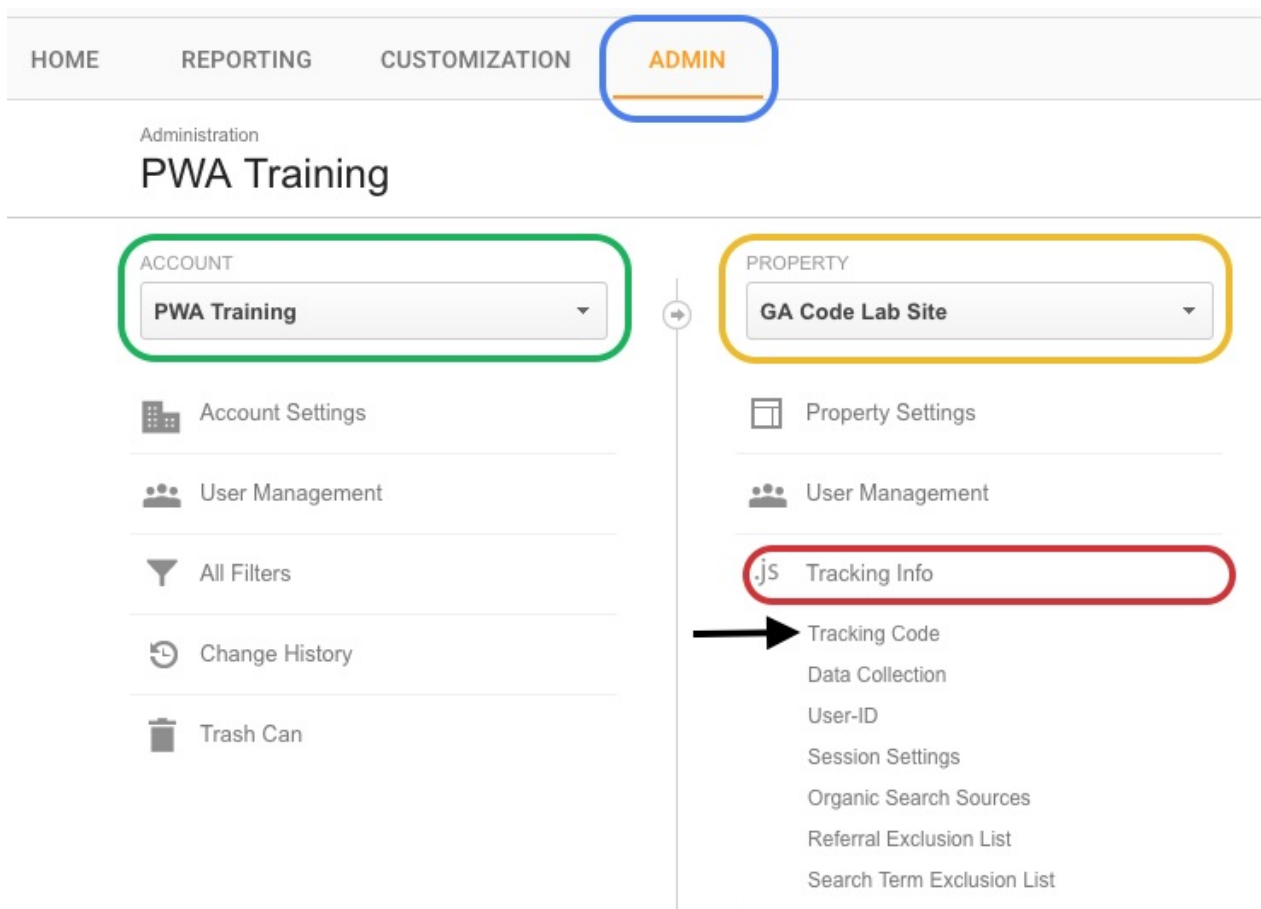
You can set an industry category to get benchmarking information later (in other words, to compare your app with other apps in the same industry). You can set your timezone here as well. You may also see data sharing options, but these are not required.

Once you have filled in your information, choose “Get Tracking ID” and agree to the terms and conditions to finish creating your account and its first property. This will take you to the tracking code page, where you get the tracking ID and tracking snippet for your app.

Add analytics to your site

Once you have created an account, you need to add the tracking snippet to your app. You can find the tracking snippet with the following steps:

1. Select the Admin tab
2. Under “account”, select your account (for example “PWA Training”) from the drop down list.
3. Then under “properties”, select your property (for example “GA Code Lab Site”) from the down list.
4. Now choose “tracking info”, followed by “tracking code”.



Your tracking ID looks like **UA-XXXXXXXX-Y** and your tracking code snippet looks like:

```

<script>
  (function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){(i[r].q=
i[r].q||[]).push(arguments)};i[r].l=1*new Date();a=s.createElement(o),m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)})(window,document,'scr
ipt','https://www.google-analytics.com/analytics.js','ga');

  ga('create', 'UA-XXXXXXX-Y', 'auto');
  ga('send', 'pageview');

</script>

```

Your tracking ID is embedded into your tracking snippet. This snippet needs to be embedded into every page that you want to track.

When a page with the snippet loads, the tracking snippet script is executed. The IIFE in the script does two things

1. Creates another `<script>` tag that starts asynchronously downloading *analytics.js*, the library that does all of the analytics work.
2. Initializes a global `ga` function, called the command queue. This function allows “commands” to be scheduled and run once the *analytics.js* library has loaded.

The next lines add two commands to the queue. The first creates a new [tracker object](#). Tracker objects track and store data. When the new tracker is created, the analytics library gets the user’s IP address, user agent, and other page information, and stores it in the tracker. From this info Google Analytics can extract:

- User’s geographic location
- User’s browser and operating system (OS)
- Screen size
- If Flash or Java is installed
- The referring site

The second command sends a “[hit](#)”. This sends the tracker’s data to Google Analytics. Sending a hit is also used to note a user interaction with your app. The user interaction is specified by the hit type, in this case a “pageview”. Since the tracker was created with your tracking ID, this data is sent to your account and property.

Note: The snippet code is complicated. The following snippet is a clearer alternative:

```

<script>
window.ga=window.ga||function(){(ga.q=ga.q||[]).push(arguments)};ga.l=+new Date;
ga('create', 'UA-XXXXX-Y', 'auto');
ga('send', 'pageview');
</script>
<script async src='https://www.google-analytics.com/analytics.js'></script>

```

The difference is that this new code relies on the `async` tag, which is not supported in some older browsers. You can [learn more about the tracking snippet](#).

The code so far provides the basic functionality of Google Analytics. A tracker is created and a pageview hit is sent every time the page is visited. In addition to the data gathered by tracker creation, the pageview event allows Google Analytics to infer:

- The total time the user spends on the site (usually requires a [hitCallback](#))
- The time spent on each page and the order the pages are visited
- Which internal links are clicked (based on the URL of the next pageview)

For more information

[analytics.js documentation](#)

[Reporting API](#)

How analytics.js works

The main interface for using *analytics.js* is the `ga` command queue. The command queue stores commands (in order) until *analytics.js* has loaded. Once *analytics.js* has loaded, the commands are executed sequentially. All commands called after *analytics.js* has loaded execute immediately. This functionality ensures that analytics can begin independent of the loading time of *analytics.js*.

Commands are added by calling `ga()`. The first argument passed is the command itself, which is a method of the *analytics.js* library. The remaining arguments are parameters for that method. For example:

```
ga('create', 'UA-XXXXX-Y', 'auto');
```

In this code, `'create'` is the command. `'UA-XXXXX-Y'` and `'auto'` are parameters for the `'create'` method. The `ga()` signature is flexible. The above code could also be written as:

```
ga('create', {  
  trackingId: 'UA-XXXXX-Y',  
  cookieDomain: 'auto'  
});
```

Unknown commands are ignored, and will not throw an error.

For more information

All commands - [command queue reference](#)

Creating tracker objects

[Tracker objects](#) track and store data. Trackers are created with the `'create'` command. For example:

```
ga('create', 'UA-XXXXX-Y', 'auto');
```

The second argument, `'UA-XXXXX-Y'`, is the [tracking ID](#). This is the same as the property ID that corresponds to the specific property in your Google Analytics account. The third argument, `'auto'`, specifies [how cookies are stored](#).

When a tracker is created it will automatically gather information about the current browsing context (such as page title and URL) and device (such as screen resolution and viewport size). This information is stored on the tracker.

You can optionally name trackers by passing in an additional [name](#) argument:

```
ga('create', 'UA-XXXXX-Y', 'auto', 'myTrackerName');
```

This is needed if you plan to [work with multiple trackers](#). If you don't name the tracker, the "default" tracker will be created (and a default name of "t0" will be used).

It is possible to set any field of the tracker at creation time by passing additional arguments. For example:

```
ga('create', 'UA-XXXXX-Y', 'auto', 'myTrackerName', {
  userId: '12345'
});
```

Or alternatively:

```
ga('create', {
  trackingId: 'UA-XXXXX-Y',
  cookieDomain: 'auto',
  name: 'myTrackerName',
  userId: '12345'
});
```

For more information

[Create command documentation](#)

[Worstyle="border:1px solid black" king with mul thtiple trackers](#)

Getting and setting tracker data

To get and set data on an existing tracker, use a [ready callback](#) function. You can add a [ready callback](#) function to the command queue:

```
ga(function(tracker) {...});
```

This executes the supplied function and enables reference to the tracker. If you create a default (unnamed) tracker before the ready callback command, the default tracker will be implicitly passed to the callback:

```
ga('create', 'UA-XXXXX-Y', 'auto');
ga(function(tracker) {
  // Logs the tracker created above to the console.
  console.log(tracker);
});
```

Note: If you are using multiple trackers and/or naming them, you will need to access them with [ga Object methods](#)

Use the [‘get’](#) method with a [field](#) specified to get data from the tracker:

```
ga('create', 'UA-XXXXX-Y', 'auto');
ga(function(tracker) {
  // Logs the tracker's name.
  // (Note: default trackers are given the name "t0")
  console.log(tracker.get('name'));
  // Logs the client ID for the current user.
  console.log(tracker.get('clientId'));
  // Logs the URL of the referring site (if available).
  console.log(tracker.get('referrer'));
});
```

Use the [set](#) method to update data fields on the tracker:

```
ga('set', 'page', '/about');
```

Or alternatively:

```
ga('set', {  
  page: '/about',  
});
```

These examples set the 'page' field to "/about". You can also update named trackers:

```
ga('myTrackerName.set', 'page', '/about');
```

Note: Tracker objects do not update themselves. If a user changes the size of the window, or if code running on the page updates the URL (such as in a single page app), tracker objects do not automatically capture this information. In order for the tracker object to reflect these changes, you must manually update it.

For more information

[Accessing trackers](#)

[ga Object methods](#)

Sending data to Google Analytics

Data is sent with the **'send'** command. For example:

```
ga('create', 'UA-XXXXX-Y', 'auto');  
ga('send', 'pageview');
```

In this code the tracker is created and then the tracker sends the data that is stored on itself. The data that trackers sends to Google Analytics is called a [hit](#) and must have a type; such as 'pageview'. The hit is an HTTP request consisting of field:value pairs encoded into a query string.

If you have your browser's developers tools open when you load a page that uses **analytics.js**, you can see the hits being sent in the network tab. Look for requests sent to `google-analytics.com/collect`.

The send command can receive optional parameters to override the current tracker fields:

```
ga('[myTrackerName.]send', [hitType], [...fields], [fieldsObject]);
```

The first parameter includes an optional tracker name. If this is omitted the default tracker is used. The optional parameters are not stored in the corresponding tracker fields - that would require the 'set' command. All send commands must specify a hitType, if one is not already set.

The simplest way to use the send command is to pass all fields using the fieldsObject parameter. For example:

```
ga('send', {
  hitType: 'event',
  eventCategory: 'Video',
  eventAction: 'play',
  eventLabel: 'cats.mp4'
});
```

But some hit types allow sending fields directly as fields parameters. For example:

```
ga('send', 'event', 'Video', 'play', 'cats.mp4');
```

For more information

[The send command](#)

Knowing when data has been sent

The `hitCallback` field can be used to execute a function once a hit has been successfully sent. For example:

```
ga('send', 'event', 'Signup Form', 'submit', {
  hitCallback: function() {
    form.submit();
  }
});
```

This is particularly important in cases when you need to record a hit that would normally take the user away from the page. For example if a user submits a form, the page's JavaScript may be unloaded before `ga('send', ...)` can run. The solution is to intercept the event, send the hit, and then finish the event with the hit callback. Here is an example:


```
function createFunctionWithTimeout(callback, opt_timeout) {
  var called = false;
  function fn() {
    if (!called) {
      called = true;
      callback();
    }
  }
  setTimeout(fn, opt_timeout || 1000);
  return fn;
}

// Gets a reference to the form element, assuming
// it contains the id attribute "signup-form".
var form = document.getElementById('signup-form');

// Adds a listener for the "submit" event.
form.addEventListener('submit', function(event) {

  // Prevents the browser from submitting the form
  // and thus unloading the current page.
  event.preventDefault();

  // Sends the event to Google Analytics and
  // resubmits the form once the hit is done.
  ga('send', 'event', 'Signup Form', 'submit', {
    hitCallback: createFunctionWithTimeout(function() {
      form.submit();
    })
  });
});
```

Note that a timeout **must** be used to ensure that even if the hit is not sent successfully, the user action will still complete.

Note that if the user's browser supports `navigator.sendBeacon` then 'beacon' can be specified as the transport mechanism. This avoids the need for a hit callback. See the [documentation](#) for more info.

For more information

[All of the fields that the tracker sends when sending a hit](#)

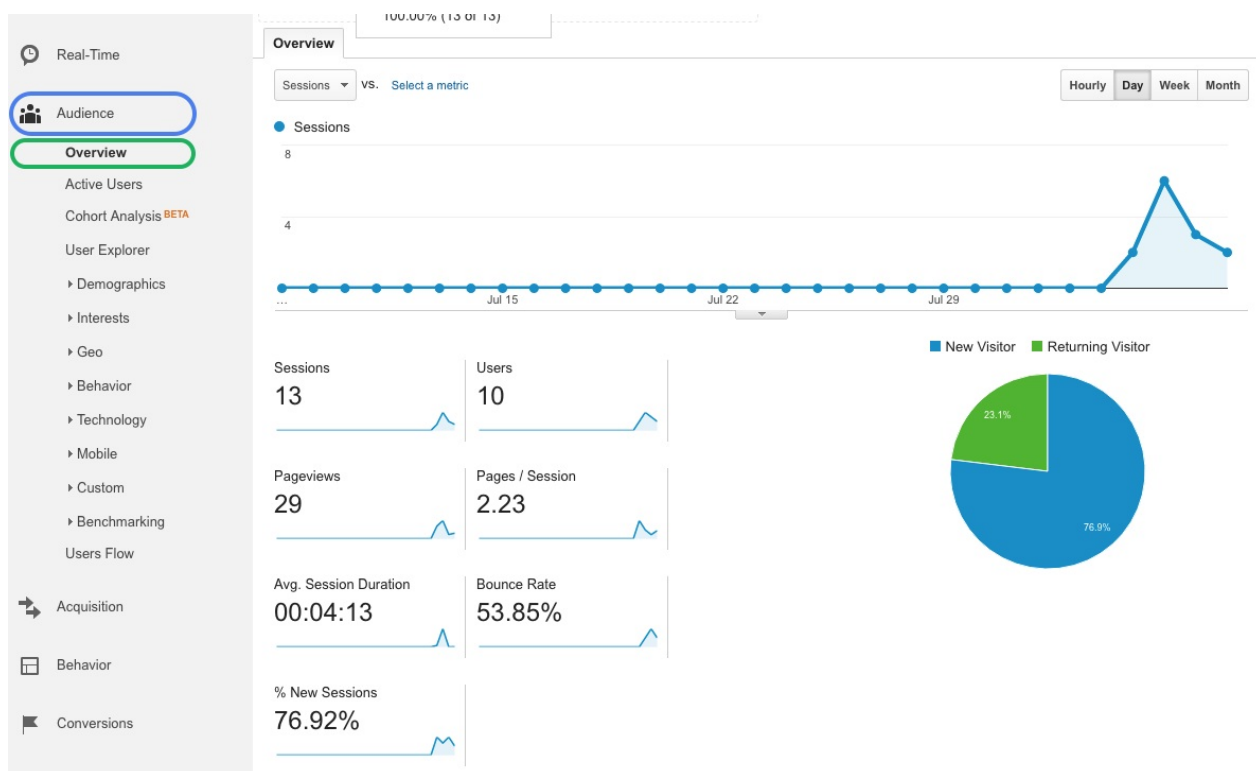
[analytics.js documentation](#)

[Sending hits documentation](#)

The Google Analytics dashboard

All of the data that is sent to Google Analytics can be viewed in the reporting tab of the Google Analytics dashboard (the Google Analytics web interface). For example, overview data is available by selecting Audience and then Overview (shown below).

Here you can see general information such as pageview records, bounce rate, ratio of new and returning visitor, and other statistics.



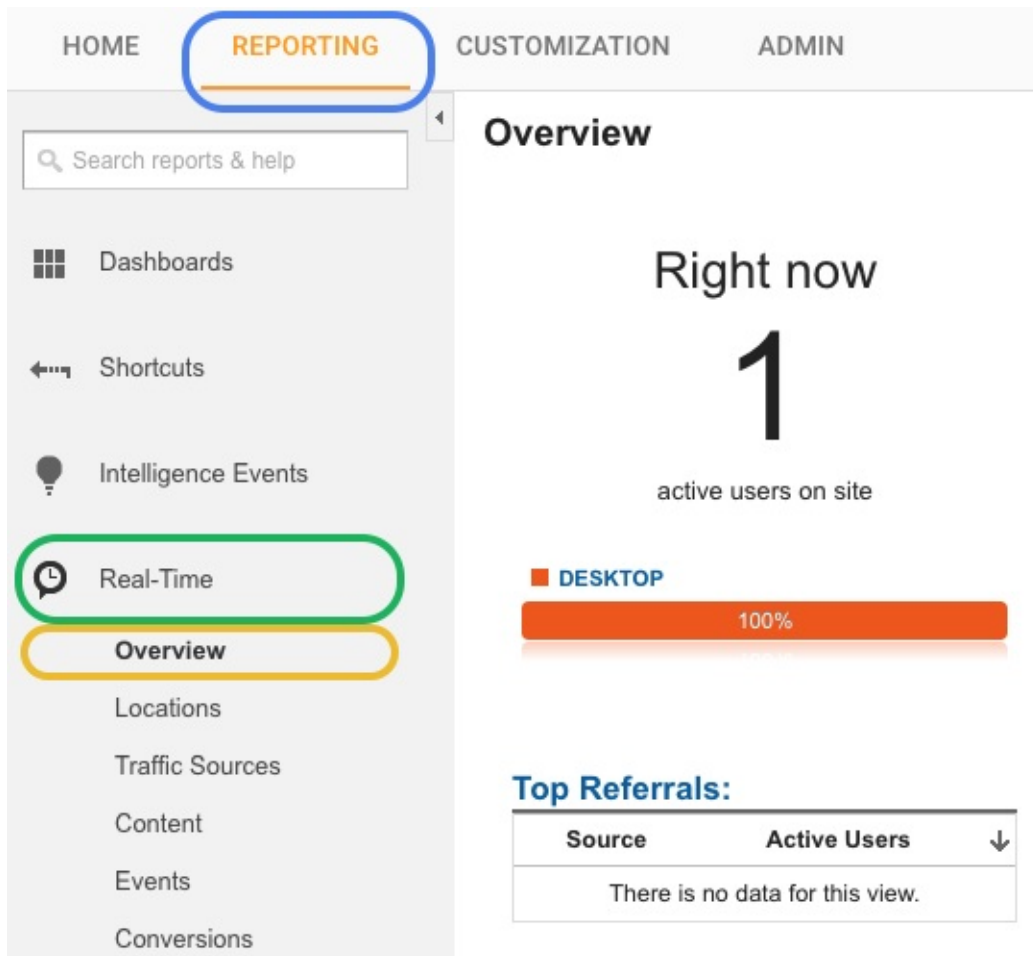
You can also see specific information like visitors' language, country, city, browser, operating system, service provider, screen resolution, and device.

| Demographics | City | Sessions | % Sessions |
|-------------------|------------------|----------|------------|
| Language | 1. Mountain View | 6 | 46.15% |
| Country | 2. (not set) | 3 | 23.08% |
| City | 3. San Mateo | 1 | 7.69% |
| System | 4. Gig Harbor | 1 | 7.69% |
| Browser | 5. Lakewood | 1 | 7.69% |
| Operating System | 6. Vancouver | 1 | 7.69% |
| Service Provider | | | |
| Mobile | | | |
| Operating System | | | |
| Service Provider | | | |
| Screen Resolution | | | |

[view full report](#)

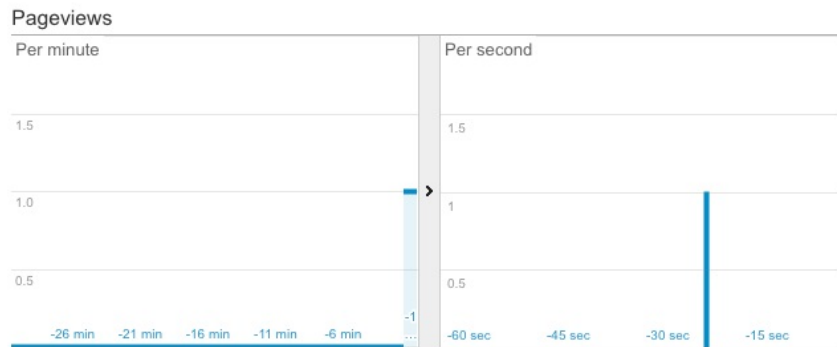
Real time analytics

It's also possible to view analytics information in real time. From the same Reporting tab, select Real-Time and Overview:



If you are visiting your app in another tab or window, you should see yourself being tracked. The screen should look similar to this:

Overview

Create Shortcut **BETA** 

Top Referrals:

| Source | Active Users | ↓ |
|---------------------------------|--------------|---|
| There is no data for this view. | | |

Top Social Traffic:

| Source | Active Users | ↓ |
|---------------------------------|--------------|---|
| There is no data for this view. | | |

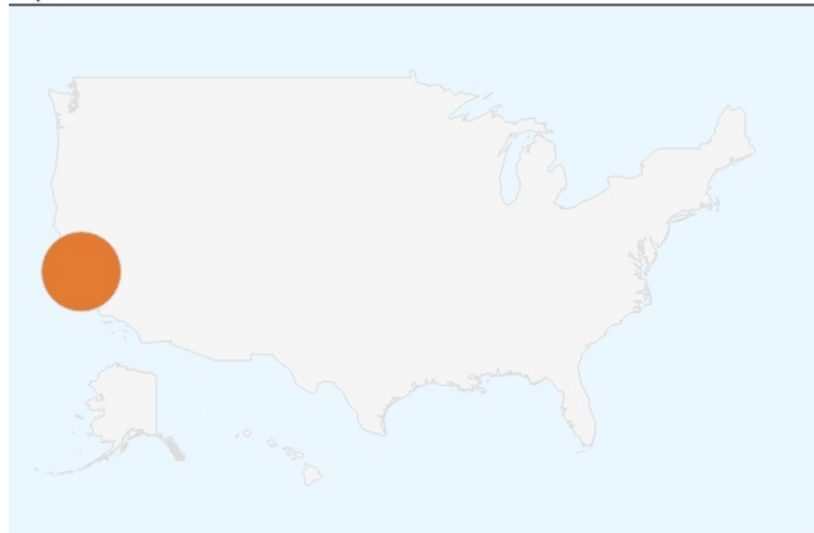
Top Keywords:

| Keyword | Active Users | ↓ |
|---------------------------------|--------------|---|
| There is no data for this view. | | |

Top Active Pages:

| Active Page | Active Users | ↓ |
|-------------|--------------|---------|
| 1. / | 1 | 100.00% |

Top Locations:



These are only the basic aspects of the Google Analytics dashboard. There are an extensive amount of features and functionality. You can [learn about the Google Analytics dashboard in depth](#).

For more information

[Learn about Google Analytics for business](#)

Debugging

Checking the dashboard is not an efficient method for testing. Google Analytics offers the *analytics.js* library with a debug mode: *analytics_debug.js*. Using this version will log detailed messages to the console that breakdown each hit sent. It also logs warnings and errors for

your tracking code. To use this version replace `analytics.js` with `analytics_debug.js` (in all instances of your tracking snippet) .

The debug version should not be used in production as it is a much larger file.

Enabling trace debugging will output more verbose information to the console. This can be enabled by adding the following code to the tracking snippet before any `ga()` calls:

```
window.ga_debug = {trace: true};
```

Note: There is also a [Chrome debugger extension](#) that can be used alternatively.

For more information

[Chrome debugger extension](#)

[Debugging documentation](#)

Custom events

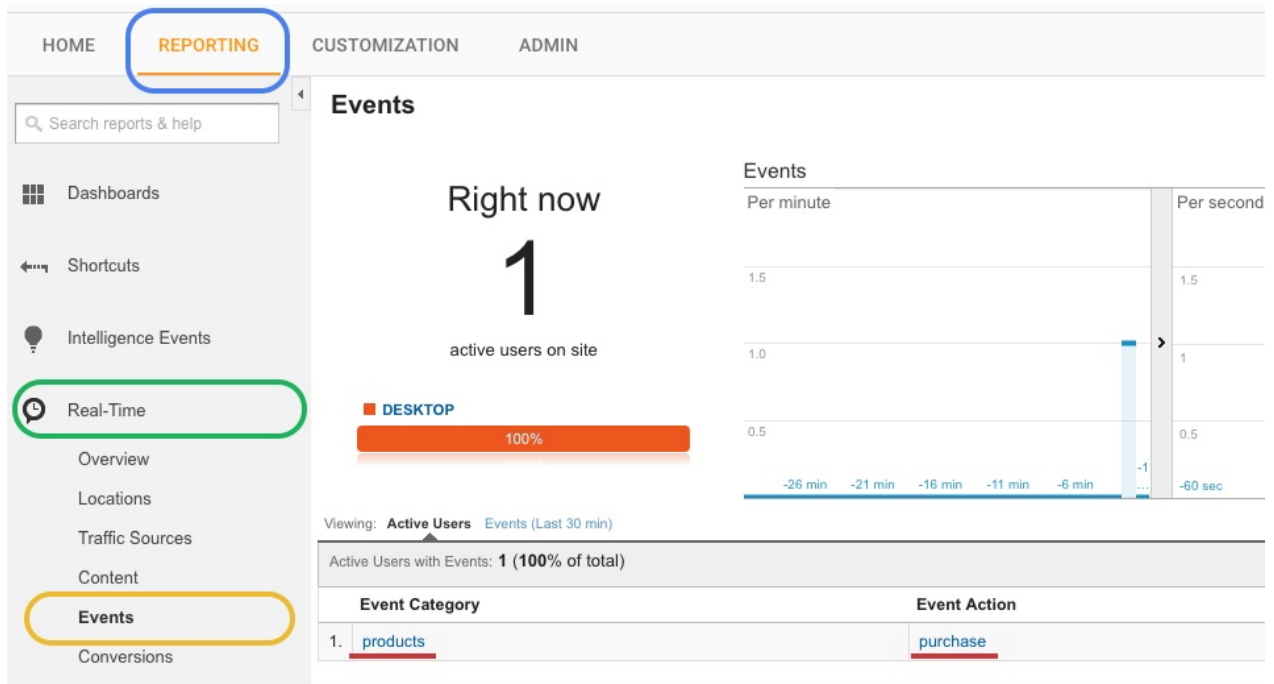
Google Analytics supports custom events that allow fine grain analysis of user behavior.

For example, the following code will send a custom event:

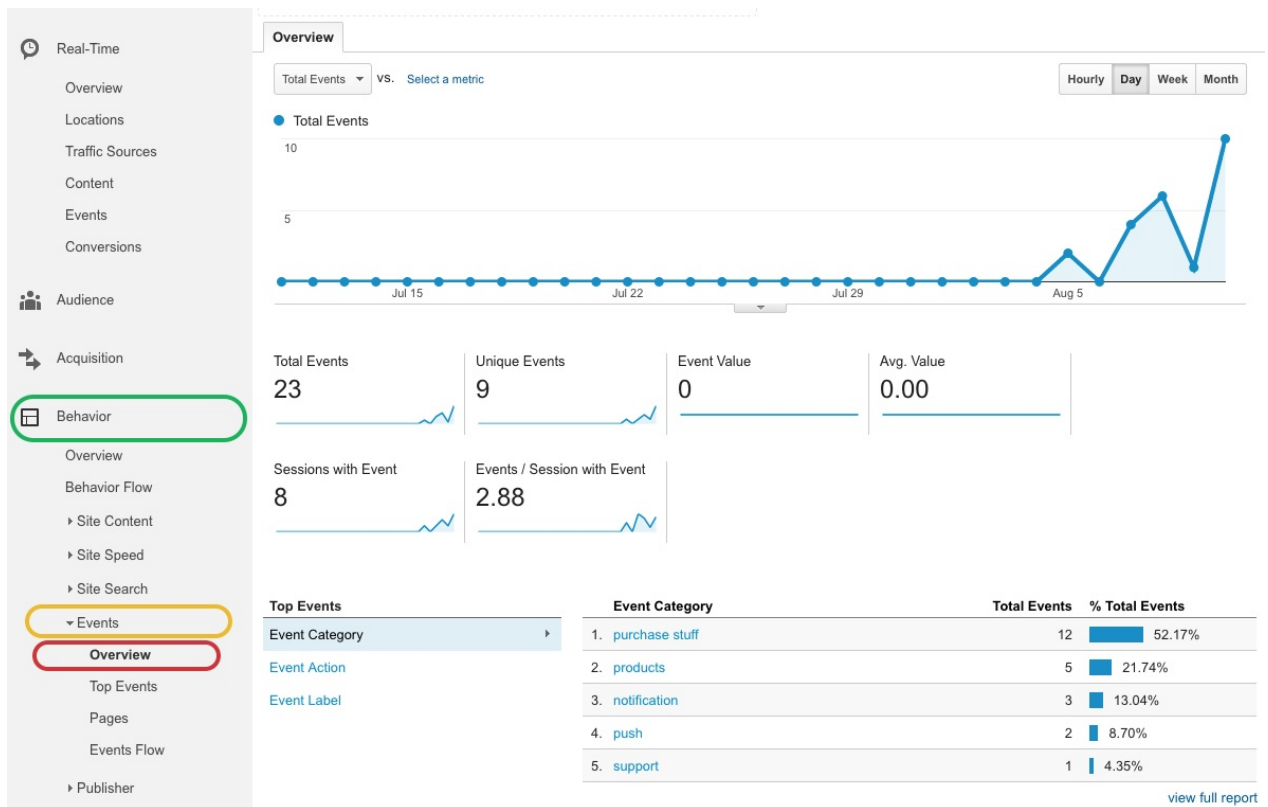
```
ga('send', {
  hitType: 'event',
  eventCategory: 'products',
  eventAction: 'purchase',
  eventLabel: 'Summer products launch'
});
```

Here the hit type is set to 'event' and values associated with the event are added as parameters. These values represent the eventCategory, eventAction, and eventLabel. All of these are arbitrary, and used to organize events. Sending these custom events allow us to deeply understand user interactions with our site.

Event data can be viewed in the Reporting tab of the Google Analytics dashboard. Real-time events are found in the Events subsection, as shown below:



You can view past events in the Google Analytics dashboard from the Reporting tab by selecting Behavior, followed by Events and then Overview:



For more information

[Event tracking](#)

[About events](#)

Example: Push notifications

Google Analytics can be used to understand user experience with push notifications. Here are some potential use cases:

Checking for support

You can set an event to fire when a support check fails. This can help you determine what isn't being supported by your users browsers and what aspect of your app they may be missing out on. For example:

```
if (!('Notification' in window)) {  
  console.log('This browser does not support notifications!');  
  ga('send', 'event', 'notification', 'unsupported');  
  return;  
}
```

Subscriptions

You can add events to fire when users subscribe or unsubscribe to push notifications, as well as when there is an error in a subscription process. This can give you an understanding of how many users are subscribing (or unsubscribing) to your app. For example:

```
function subscribe() {
  registration.pushManager.subscribe({userVisibleOnly: true})
  .then(function(pushSubscription) {
    console.log('Subscribed!');
    ga('send', 'event', 'push', 'subscribe');
  })
  .catch(function(error) {
    if (Notification.permission === 'denied') {
      console.warn('Subscribe failed, notifications are blocked');
      ga('send', 'event', 'push', 'subscribe-blocked');
    } else {
      console.warn('Error subscribing', error);
      ga('send', 'event', 'push', 'subscribe-error');
    }
  });
}

function unsubscribe() {
  subscription.unsubscribe()
  .then(function() {
    console.log('Unsubscribed!');
    ga('send', 'event', 'push', 'unsubscribe');
  })
  .catch(function(error) {
    console.warn('Error unsubscribing', error);
    ga('send', 'event', 'push', 'unsubscribe-error');
  });
}
```

Analytics and service worker

Much of the functionality of push notifications and other progressive web app features occur in a service worker. This is an important note for analytics because the service worker script runs on its own thread and doesn't have access to the `ga` command queue object (which is on the main thread). To send hits, we need a way to communicate between the service worker and the main thread.

This can be achieved with a [MessageChannel](#), which allows data to be sent across threads on its two [MessagePorts](#). The following code is an example of how this could be implemented.

In the main JavaScript:


```
function openCommunication() {
  var msgChannel = new MessageChannel();
  msgChannel.port1.onmessage = function(msgEvent) {
    sendAnalytics(msgEvent.data);
  };
  navigator.serviceWorker.controller.postMessage(
    'Establishing contact with service worker', [msgChannel.port2]
  );
}
openCommunication();

function sendAnalytics(message) {
  var eventCategory = message.eventCategory;
  var eventAction = message.eventAction;
  ga('send', 'event', eventCategory, eventAction);
}
```

The code added in the main JavaScript adds a message listener on the first MessagePort (port1). When the message listener receives a message (which will come from the service worker), it calls a `sendAnalytics` function with the message data. The `sendAnalytics` function parses the data and executes a Google Analytics send command.

An initial message is sent to the service worker with the `postMessage` function, containing the MessageChannel's second MessagePort (port2).

In the service worker:

```
var mainClientPort;
self.addEventListener('message', function(event) {
  console.log('Service worker received message: ', event.data);
  mainClientPort = event.ports[0];
  console.log('Communication ports established');
});
```

The code added to *service-worker.js* adds a generic listener (not on any specific MessageChannel or MessagePort) to the service worker to receive the initial message. When the message is received from *main.js*, the second message port (port2) is saved. This will allow the service worker to send data back to *main.js*.

Now we have the ability for the service worker to send messages to *main.js* (using port2). And *main.js* can hear those messages and send custom events to Google Analytics with the message data.

Building on this, you could then add more use cases:

Push events

You can add events recording when a push notification has been received and displayed. With this you can understand the timing between users receiving, viewing, and interacting with notifications, as well as the number of delivery failures. For example:

```
self.addEventListener('push', function(event) {
  mainClientPort.postMessage({
    eventCategory: 'push',
    eventAction: 'received'
  });
  if (Notification.permission === 'denied') {
    console.warn('Push notification failed, notifications are blocked');
    mainClientPort.postMessage({
      eventCategory: 'push',
      eventAction: 'blocked'
    });
    return;
  }
  var options = {...};
  event.waitUntil(
    Promise.all([
      self.registration.showNotification('Hello world!', options),
      mainClientPort.postMessage({
        eventCategory: 'notification',
        eventAction: 'displayed-push'
      })
    ])
  );
});
```

Notification Interactions

You can add events for the various interactions the user takes with notifications. This can tell how your users are interacting with your notifications and how much they value them (based on how quickly or frequently they interact with or dismiss them). For example:

```
self.addEventListener('notificationclose', function(event) {
  mainClientPort.postMessage({
    eventCategory: 'notification',
    eventAction: 'closed'
  });
});

self.addEventListener('notificationclick', function(event) {
  var action = event.action;
  if (action === 'decline') {
    ...
    mainClientPort.postMessage({
      eventCategory: 'notification',
      eventAction: 'decline'
    });
  } else if (action === 'accept') {
    ...
    mainClientPort.postMessage({
      eventCategory: 'notification',
      eventAction: 'accepted'
    });
  } else {
    ...
    mainClientPort.postMessage({
      eventCategory: 'notification',
      eventAction: 'clicked'
    });
  }
});
```

For more information

[MessageChannel](#)

[MessagePorts](#)

[postMessage](#)

[Web workers](#)

[Communicating with service worker example](#)

Offline analytics

Using service worker and [IndexedDB](#), analytics data can be stored when users are offline and sent at a later time when they have reconnected. The *sw-offline-google-analytics* [npm package](#) abstracts this process. You can install the npm package in your project with

```
$ npm install --save-dev sw-offline-google-analytics
```

Once the package is installed. The following code needs to be added to your service worker file:

```
importScripts('path/to/offline-google-analytics-import.js');  
goog.offlineGoogleAnalytics.initialize();
```

[Importing](#) and initializing the *offline-google-analytics-import.js* library adds a fetch event handler to the service worker that only listens for requests made to the Google Analytics domain. The handler attempts to send Google Analytics data just like we have done so far, by network requests. If the network request fails, the request is stored in IndexedDB. Each time the service worker starts up again it attempts to resend the stored requests.

You can test this behavior by enabling Offline mode in the Network tab of dev tools, and then triggering Google Analytics hits. The IndexedDB section of Storage in the Application tab will show a list of URLs that represent these unsent hit requests (you may need to click the refresh icon inside the cache interface). If you disable Offline mode and refresh the page, you should see that the URL's are cleared (indicating that they have been sent to Google Analytics).

You can read more about the [offline analytics package](#).

For more information

[ImportScripts](#)

[Offline Google Analytics](#)

[Google I/O offline example](#)

[Node package manager \(npm\)](#)

[IndexedDB](#)

Additional resources

[analytics.js documentation](#)

[Google Analytics Academy](#) (non-technical)

[Using the User Timing API with Google Analytics code lab](#)

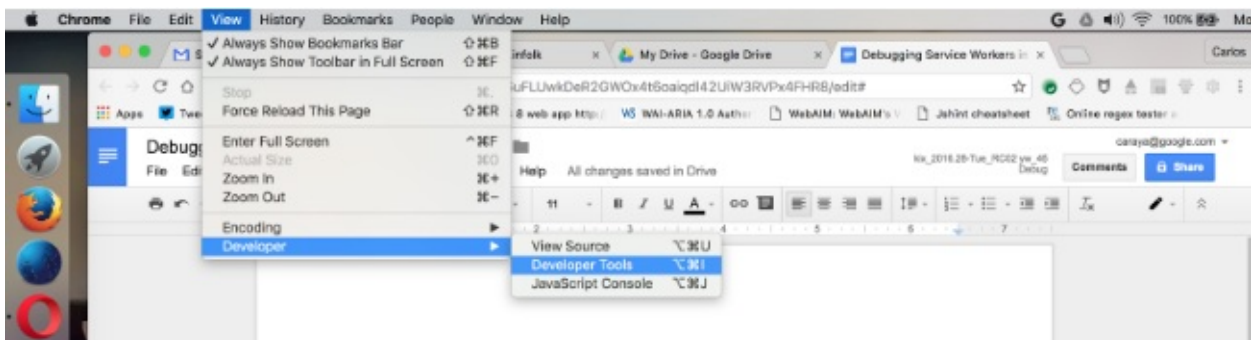
Practical Progressive Enhancement

Debugging Service Workers in Browsers

Chrome

Open DevTools with **F12** or **Ctrl + Shift + I** on Windows Or **Cmd + Opt + I** on Mac

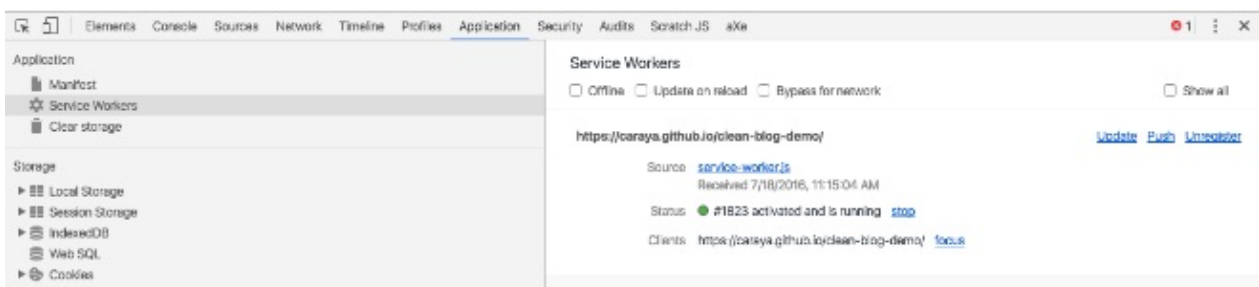
The command is also available under the **View > Developer > Developer tools** as shown in the image below:



Once in DevTools you can check the status of your service worker under application. When you click the application tab you will see the current page's service worker in the right side of the screen along with some developer helpers to make working with the service worker easier.

There are three checkboxes:

- **Offline**: forces the browser offline, simulating no network connectivity
- **Update on Reload**: reloads the service worker every time the browser reloads, picking up all changes during development
- **Bypass for network**: Ignores the service worker and fetches all resources from the network

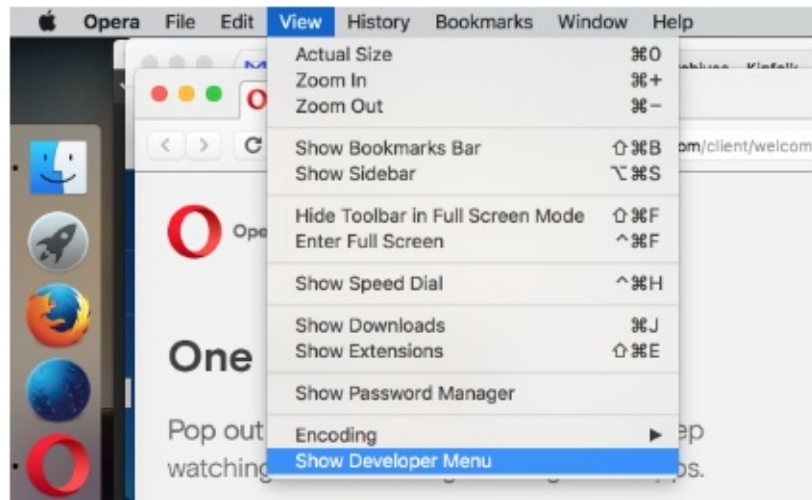


For more information:

<https://developers.google.com/web/tools/chrome-devtools/?hl=en>

Opera

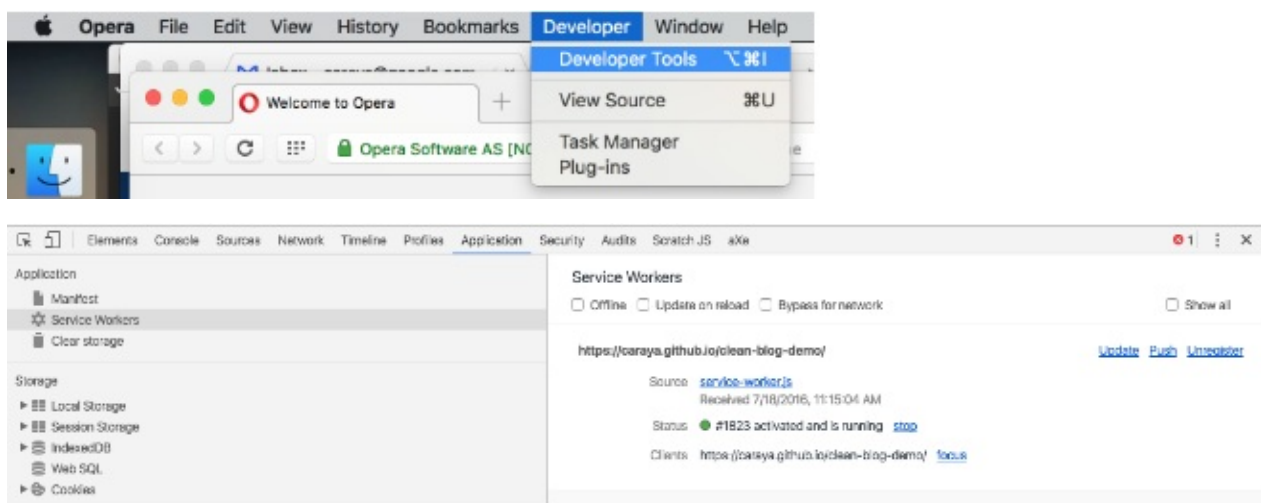
To activate Devtools in Opera first show the developer menu. From the **View** menu click



Show Developer Menu

Then

from the **Developer** menu open the Developer Tools by clicking on **Developer Tools** or using **F12 or Ctrl + Shift+ I on Windows Or Cmd + Opt + I on Mac**



Once in Devtools you can check the status of your service worker under application. When you click the application tab you will see the current page's service worker in the right side of the screen along with some developer helpers to make working with the service worker easier.

There are three checkboxes:

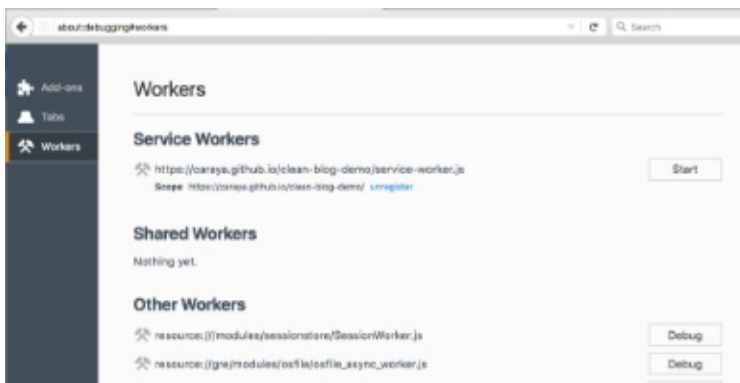
- **Offline**: forces the browser offline, simulating no network connectivity
- **Update on Reload**: reloads the service worker every time the browser reloads, picking up all changes during development
- **Bypass for network**: Ignores the service worker and fetches all resources from the network

Firefox

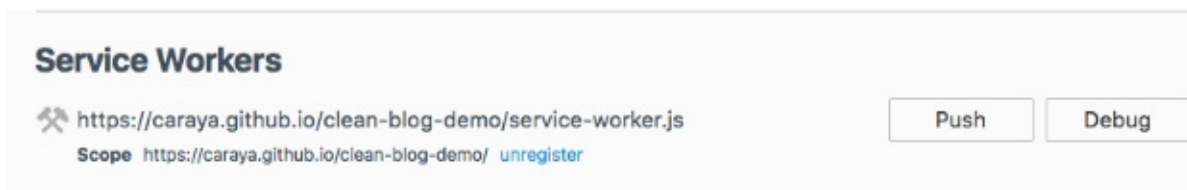
Firefox service worker debugging tools work slightly different than Chrome and Opera. Rather than opening the Dev Tools window, type the following in your location bar:

about:debugging

This will open the debugger window. It is a global tool; from here you can debug add-ons, tabs and their content as well as workers (web workers, shared workers and service workers).



An active service worker looks like the image below:



From here we can unregister the service worker, push messages to active clients and debug it in Firefox's Javascript debugger.