
Table of Contents

Introduction	1.1
Introduction to PWA Architectures	1.2
Introduction to Service Worker	1.3
Offline Quickstart	1.4
Working with Promises	1.5
Working with the Fetch API	1.6
Caching Files with Service Worker	1.7
Working with IndexedDB	1.8
Live data in the Service Worker	1.9
Lighthouse PWA Analysis Tool	1.10
Introduction to Gulp	1.11
Introduction to Push Notifications	1.12
Integrating Analytics	1.13
Introduction to Payment Request API	1.14
PWA Terminology	1.15

Progressive Web Apps ILT - Concepts

This instructor-led training course for progressive web apps (PWAs) was developed by Google Developer Training.

Progressive web apps (PWAs) is the term for the open and cross-browser technology that provides better user experiences on the mobile web. Google is supporting PWAs to help developers provide native-app qualities in web applications that are reliable, fast, and engaging. The goal of PWAs is to build the core of a responsive web app and add technologies incrementally when these technologies enhance the experience. That's the *progressive* in Progressive Web Apps!

Let's get started!

Introduction to Progressive Web App Architectures

Content

[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](#)

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.

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 a client-side storage API, such as the [Cache Storage API](#) or [IndexedDB](#).

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, the 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. If the application shell has been cached by service worker, then on repeat visits the app shell allows you to get meaningful pixels on the screen really fast without the network. Making use of an app shell is not a hard requirement for building PWAs, but it can result in significant performance gains when cached and served correctly.

The shell of the functionality is loaded and displayed to the user (and potentially cached by the service worker so that it can be accessed offline), 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.

Note: If your website is a templated site (i.e. built using multiple templates combined with the actual text, images, and other resources that make up the site's content), then read Jeffrey Posnick's [Offline-first for Your Templated Site](#) to learn different strategies for caching and serving templated sites.

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. On browsers that do not support service workers the offline-specific code is never called and there is no overhead or breakage.

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.

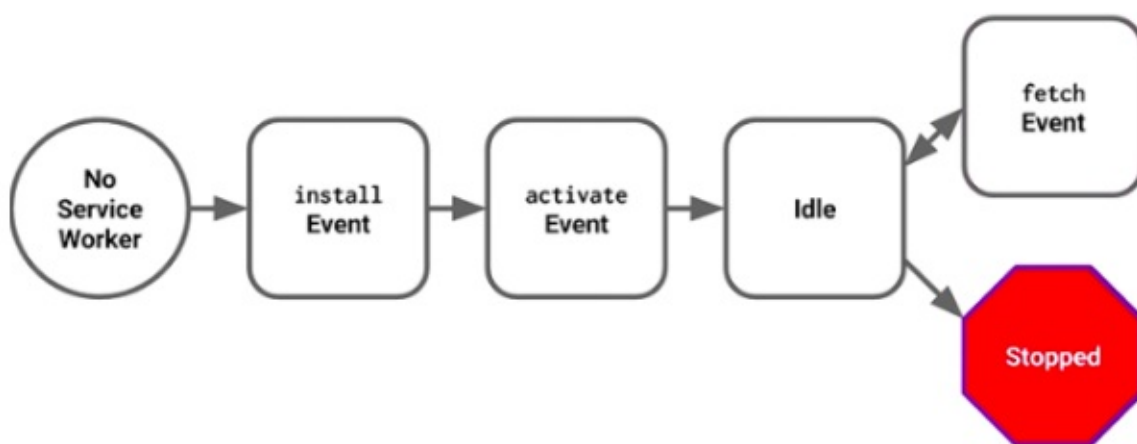
Here is a high-level description of how it works:

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. When the service worker is installed, the app shell content is added to the cache. Once installed, the service worker controls future navigations on the site. Note that you can sometimes manually activate a new service worker using browsers' developer tools or programmatically with [Service Worker APIs](#).

2. After the shell content loads, the 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. Each request triggers a `fetch` event inside the service worker that can be handled any way you choose. Once those handlers complete, your service worker enters into an idle state. So, the service worker is idle 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 the page is loaded again the service worker is started back up and can immediately respond to `fetch` events.



Note: Once for every given version of a service worker, 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. This only happens once for every given version of the service worker JavaScript file.

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 in the browser: browser-managed cache and application-managed cache (service worker).</p> <p>Browser-managed caches are 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. This cache is managed automatically by the browser and is not available offline.</p> <p>Application-managed caches are created using the Cache API independent of the browser-managed caches. This API is available to applications (via <code>window.caches</code>) and the service worker. Application-managed caches hold the same kinds of assets as a browser cache but are accessible offline (e.g. by the service worker to enable offline support). This cache is managed by developers who implement scripts that use the Cache API to explicitly update items in named cache objects. Cache is a great tool you can use when building your app, as long as the cache you use is appropriate for each resource. Several caching strategies are described in the PWA Caching Strategies tutorial.</p>
client-side rendering (CSR)	Client-side rendering means JavaScript running in the browser produces HTML (probably via templating). The benefit is that 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.
dynamic content	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.
Fetch API	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.
progressive enhancement	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.
PWA architecture styles	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 PWA Architectural Patterns .
server-side rendering (SSR)	SSR means when the browser navigates to a URL and 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, then 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> module 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 alongside 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. For example, the types of resources required to render a web page include:

- HTML, CSS and JavaScript files
- Images, media and other 'replaced' content
- Fonts
- Data retrieved and used by JavaScript to build content. This might include HTML, CSS, or JavaScript, text to update content, URLs (for images and other resources, but maybe also for content retrieved via APIs), and so on.

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 can provide a quick time to first render. 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 it. Also, SSR pages work across a range of browsers without concern over differences in JavaScript implementations.

Sites where you mostly navigate and view static content (such as news outlets) can get away with using a strictly SSR approach. For sites that are more dynamic (such as social media or shopping), 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 it offloads page updates to the client so that screen updates occur instantly when the user clicks, rather than waiting while the server is contacted for information about what to display. Thus, when data has changed after the initial page render, CSR can selectively re-render portions of the page (or reload the entire page) when new data is received from the server or following user interaction.

Note: As with SSR, the consequence of reloading the entire page 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. See also [Instant Loading Web Apps with an Application Shell Architecture](#), which provides an in-depth description of the following patterns, that are known styles for building PWAs. The patterns are listed below 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 both shell + content for entry page) + use JavaScript to fetch content for any further routes and do a "take over"

Note: 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.

Note: If you are building a PWA using Polymer leveraging this pattern, then it might be worth exploring SSR of content in the Light DOM.

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

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
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. The [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.

Note: If your website is a templated site (i.e. built using multiple templates combined with the actual text, images, and other resources that make up the site's content), then read Jake Archibald's [Offline-first for Your Templated Site](#) to learn different strategies for caching and serving templated sites.

Once you decide on a strategy for caching then you must implement it. A SPA architecture is often recommended when using an app shell. When we refer to SPA (single-page apps), we're talking about apps that are loaded once and are then *lived in* for a long period of time. SPAs rely heavily on script and have entirely separate "server rendered" versions. Google's headline web apps (GMail, Inbox, Maps, Docs, Sheets, and so on) have been pioneers in this realm. These web apps are distinguished by managing a larger share of the state of the application on the client and then doing some form of synchronization on data sets.

However, it can take some time to refactor an existing site/app over to the SPA 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 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 caching strategies in [The Offline Cookbook](#) 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.

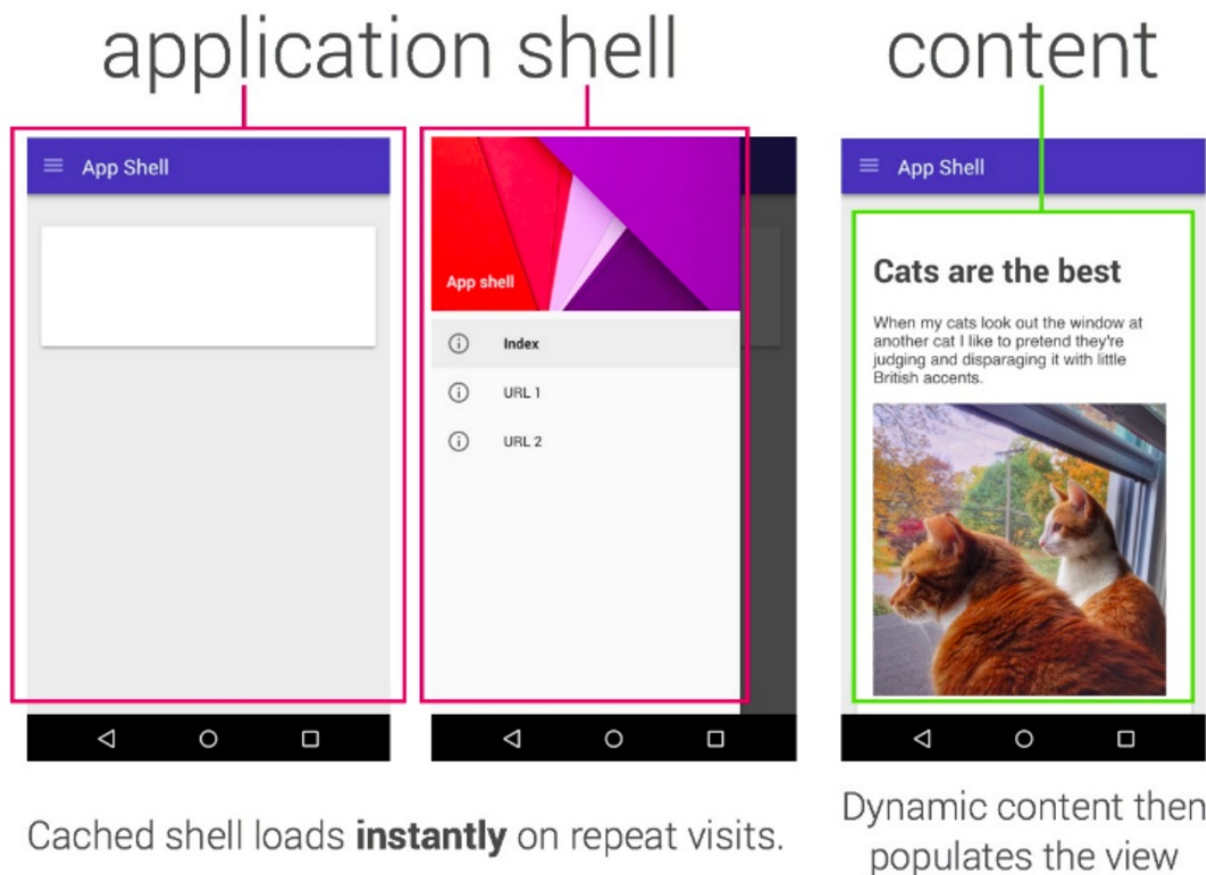
An app shell always includes HTML, usually includes JavaScript and CSS, and might include any other static resources that provide the structure for your page. However, it does not include the actual content specific to the page. In other words, the app shell contains the parts of the page that change 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 [To the Lighthouse](#) demo. . 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.

Real World Examples

For a simple example of a web app manifest file, see the

[Use a Web App Manifest File](#) section.

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.

Earlier versions of PWA used the `sw-toolbox` library and `sw-precache` module that are built on top of the service worker primitives, like the `Cache` and `Fetch` APIs. These tools abstract low-level complexities and make it easier for developers to work with service workers. This section provides some simple examples of these tools, but we recommend using `Workbox`.

Note: [Workbox](#) is the successor to `sw-precache` and `sw-toolbox`. It is a collection of libraries and tools used for generating a service worker, precaching, routing, and runtime-caching. `Workbox` also includes modules for easily integrating background sync and Google analytics into your service worker. See the [PWA Workbox Lab](#) to learn how to use `Workbox` to easily create production-ready service workers. Also, see the [Workbox page](#) on developers.google.com for an explanation of each module contained in `Workbox`.

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 detail.)

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 [app-shell](#) demo 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
}}
);
```

The `sw-precache` Module

`sw-precache` integrates with your build process and automatically generates the service worker code that takes care of caching and maintains 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 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 and caches the network response.	When dealing with remote resources that are very unlikely to change, such as static images.	<code>toolbox.cacheFirst</code>
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>
	Fires the same request to the network and the cache	When content is updated frequently, such as for articles,	

Cache/network race	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.	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 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 to 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 the [Offline Cookbook](#).

Note: If your website is a templated site (i.e. built using multiple templates combined with the actual text, images, and other resources that make up the site's content), then read Jake Archibald's [Offline-first for Your Templated Site](#) to learn different strategies for caching and

serving templated sites.

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

Note: 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. See the [Service](#)

[Worker Precache](#) documentation on Github for a complete description and examples.

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` module 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.

1. 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.
2. 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.

1. Screen transitions shouldn't feel slow due to blocking on the network
2. Tappable areas should give touch feedback
3. Touching an element while scrolling shouldn't trigger touch feedback
4. Content shouldn't jump as the page loads
5. Pressing back from a detail page should retain scroll position on the previous list page
6. Buttons and 'non-content' shouldn't be selectable
7. Ensure inputs aren't obscured by keyboard
8. Provide an easy way to share content
9. Use system fonts
10. Avoid overly "web-like" design (use links sparingly and instead carefully place "buttons" and tappable regions)
11. 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 [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.

For an look at what's required to make your app work offline, see [The Service Worker Lifecycle](#) and try the step-by-step PWA codelab [Scripting the Service Worker](#). To see a complete, sophisticated implementation, see the 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 module 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 Module](#) section 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,png,jpg,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 [Introduction to Push Notifications](#) (textbook) and [Lab: Integrating Web Push](#) (codelab).

What Happens When an Inactive App Gets a Push Message?

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

Once the user sees the notification they can ignore it until later, dismiss it, or action it. The user normally taps 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 your 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](#)

[Further reading](#)

Codelab: [Scripting the Service Worker](#)

What is a service worker?

A [service worker](#) is a type of [web worker](#). It's essentially a JavaScript file that runs separately from the main browser thread, intercepting network requests, caching or retrieving resources from the cache, and delivering push messages.

Because workers run separately from the main thread, service workers are independent of the application they are associated with. This has several consequences:

- Because the service worker is not blocking (it's designed to be fully asynchronous) synchronous XHR and `localStorage` cannot be used in a service worker.
- The service worker can receive push messages from a server when the app is not active. This lets your app show push notifications to the user, even when it is not open in the browser.

Note: Whether notifications are received when the browser itself is not running depends on how the browser is integrated with the OS. For instance on desktop OS's, Chrome and Firefox only receive notifications when the browser is running. However, Android is designed to wake up any browser when a push message is received and will always receive push messages regardless of browser state. See the [FAQ](#) in Matt Gaunt's [Web Push Book](#) for more information.

- The service worker can't access the DOM directly. To communicate with the page, the service worker uses the `postMessage()` method to send data and a "message" event listener to receive data.

Things to note about a service worker:

- Service worker is a programmable network proxy that lets you control how network requests from your page are handled.
- Service workers only run over HTTPS. Because service workers can intercept network requests and modify responses, "man-in-the-middle" attacks could be very bad.

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

- The service worker becomes idle when not in use and restarts when it's next needed. You cannot rely on a global state persisting between events. If there is information that you need to persist and reuse across restarts, you can use [IndexedDB](#) databases.
- Service workers make extensive use of promises, so if you're new to promises, then you should stop reading this and check out [Promises, an introduction](#).

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.

Service workers depend on two APIs to make an app work offline: [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.

Improve performance of your application/site

Caching resources will make content load faster under most network conditions. See [Caching files with the service worker](#) and [The Offline Cookbook](#) for a full list of caching strategies.

Make your app "offline-first"

Using the Fetch API inside a service worker, we can intercept network requests and then modify the response with content other than the requested resource. We can use this technique to serve resources from the cache when the user is offline. See [Caching files with the service worker](#) to get hands-on experience with this technique.

Act as the base for advanced features

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

- **Notifications API**: A way to display and interact with notifications using the operating system's native notification system.
- **Push API**: An API that enables your app to subscribe to a push service and receive push messages. Push messages are delivered to a service worker, which can use the information in the message to update the local state or display a notification to the user. Because service workers run independently of the main app, they can receive and display notifications even when the browser is not running.
- **Background Sync API**: Lets you defer actions until the user has stable connectivity. This is useful to ensure 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
- **Channel Messaging API**: Lets web workers and service workers communicate with each other and with the host application. Examples of this API include new content notification and updates that require user interaction.

Service worker lifecycle

A 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. Let's look at an example:

main.js

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(function(registration) {
      console.log('Registration successful, scope is:', registration.scope);
    })
    .catch(function(error) {
      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 which files the service worker controls, in other words, from which path the service worker will intercept requests. The default scope is the location of the service worker file, and extends to all directories below. 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. For example:

main.js

```
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.

If the service worker is already installed, `navigator.serviceWorker.register` returns the registration object of the currently active service worker.

Installation

Once the browser registers a service worker, **installation** can be attempted. This occurs if the service worker is considered to be new by the browser, either because the site currently doesn't have a registered service worker, or because there is a byte difference between the new service worker and the previously installed one.

A service worker installation triggers an `install` event in the installing service worker. We can include an `install` event listener in the service worker to perform some task when the service worker installs. For instance, during the install, service workers can precache parts of a web app so that it loads instantly the next time a user opens it (see [caching the application shell](#)). So, after that first load, you're going to benefit from instant repeat loads and your time to interactivity is going to be even better in those cases. An example of an installation event listener looks like this:

service-worker.js

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

Activation

Once a service worker has successfully installed, it transitions into the **activation** stage. If there are any open pages controlled by the previous service worker, the new service worker enters a `waiting` state. The new service worker only activates when there are no longer any pages loaded that are still using the old service worker. This ensures that only one version of the service worker is running at any given time.

Note: Simply refreshing the page is not sufficient to transfer control to a new service worker, because the new page will be requested before the the current page is unloaded, and there won't be a time when the old service worker is not in use.

When the new service worker activates, an `activate` event is triggered in the activating service worker. This event listener is a good place to clean up outdated caches (see the [Offline Cookbook](#) for an example).

service-worker.js

```
self.addEventListener('activate', function(event) {
  // Perform some task
});
```

Once activated, the service worker controls all pages that load within its scope, and starts listening for events from those pages. However, pages in your app that were loaded before the service worker activation will not be under service worker control. The new service worker will only take over when you close and reopen your app, or if the service worker calls `clients.claim()`. Until then, requests from this page will not be intercepted by the new service worker. This is intentional as a way to ensure consistency in your site.

Service worker events

Service workers are event driven. Both the installation and activation processes trigger 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 developer tools to check the status of service workers, see [Tools for PWA Developers](#).

Further reading

- A more detailed introduction to [The Service Worker Lifecycle](#)
- More on [Service Worker Registration](#)
- [Create your own service worker](#) (lab)
- [Take a blog site offline](#) (lab)
- [Cache files with Service Worker](#) (lab)

Offline Quickstart

Contents

[Why build offline support?](#)

[How do I take my app offline?](#)

[Further reading](#)

Codelab: [Offline Quickstart](#)

Why build offline support?

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

As application developers, we want to ensure a good user experience, preventing network shortcomings from affecting applications. With [service workers](#) we now have a way to build offline support. Service workers provide an in-browser, programmable network proxy, so that users can always get to something on your website.

Service workers provide many new features to web applications, including programmatic file caching, intercepting network request, and receiving push messages. The service worker runs independently of the web app and can even be called when the app isn't running (for example to wake it up and deliver a message).

Some benefits of implementing service workers include:

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

Offline access

Service workers can use the [Cache](#) interface to cache an application's assets. A service worker script can implement a number of [caching strategies](#), allowing fine tuning of an app's offline and low-connectivity performance.

The Cache interface's storage is controlled programmatically and **is independent** of the browser's HTTP cache. Unlike the browser's HTTP cache, the Cache interface's storage is available offline. The service worker can use this to enable offline support in browsers.

Service workers can also use [IndexedDB](#) to store data locally. This enables new features such as capturing user actions while offline and delivering them once connectivity returns.

Note: The service worker's approach was driven by the problems the community had with Application Cache (AppCache), where a purely declarative approach to caching proved to be too inflexible. Unlike AppCache, service workers don't provide defaults making all behavior explicit. If a behavior is not written into your service worker, then the behavior does not happen. By explicitly coding behaviors in a service worker, the task of writing and debugging code is made easier. For an example of working with AppCache and the challenges developers face, see Jake Archibald's [Application Cache is a Douchebag](#) article. However, using AppCache is highly discouraged because it is in the process of being removed from the Web platform. Use service workers instead.

Improved performance

Caching data locally results in speed and cost benefits for mobile users (many of whom are charged based on their data usage). The service worker can cache content in the user's browser and retrieve data from the cache without going to the network. This provides a faster (and probably cheaper) experience for all users, even those with strong connectivity.

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 take actions even when the web app isn't running (for example, receiving push notifications, syncing data in the background, and geofencing). Combined with progressive enhancement, these features can be safely added to your app without breaking it in unsupported browsers. To see if a target browser supports a given service worker feature, check [Is Service Worker Ready?](#)

How do I take my app offline?

The core of an offline experience is the service worker. It lets the developer choose when to cache resources and when to retrieve content from the cache instead of from the network (see [The Offline Cookbook](#) for more information on caching strategies).

A possible implementation pattern could look like this:

1. Register a service worker (for first time app visits, this triggers service worker installation).
2. On service worker installation, cache the app's static assets (generally the minimum HTML, CSS, and JS that the app needs to open).
3. Have the service worker listen for resource fetches. When a resource is fetched, have the service worker attempt to find the resource in the cache before going to the network.
4. If a resource must be retrieved from the network, have the service worker cache a copy of the resource so that it can be retrieved from the cache in the future.

Let's walk through a simple example of taking an app offline.

Registering a service worker

The first step in offline functionality is registering a service worker. Use the following code (which should be executed when your app loads) to register a service worker:

index.html

```
if ('serviceWorker' in navigator) {  
  navigator.serviceWorker.register('service-worker.js')  
    .then(function(registration) {  
      console.log('Registered:', registration);  
    })  
    .catch(function(error) {  
      console.log('Registration failed: ', error);  
    });  
}
```

This code starts by checking for browser support, and then registers the service worker. If this is the first time the user has visited your app, the service worker will install and activate.

Caching static assets on install

A common strategy is to cache the site's static assets when the service worker installs. Then, after a user has visited your site for the first time, the static content can be retrieved from the cache on future visits. The following code (in the service worker file) shows how to do this:

service-worker.js

```
var CACHE_NAME = 'static-cache';
var urlsToCache = [
  '.',
  'index.html',
  'styles/main.css'
];
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        return cache.addAll(urlsToCache);
      })
  );
});
```

This code starts by defining a cache name and a list of URLs to be cached (the static assets). It creates an install event listener that executes the code inside of it when the service worker installs. In this example, the code in the install listener opens a cache and stores the list of assets.

Note: The `.` represents the current directory (for example, **app/**). If the user navigates to **app/**, the browser generally shows **app/index.html**. However, **app/** and **app/index.html** are separate URLs, so a 404 can still occur if the user navigates to **app/** and only **app/index.html** is available. We cache `.` as well as `index.html` to avoid this potential error.

Note: The `event.waitUntil` can be particularly confusing. This operation simply tells the browser not to preemptively terminate the service worker before the asynchronous operations inside of it have completed.

Fetching from the cache

Now that there are assets in the cache, the service worker can use those resources instead of requesting them from the network:

service-worker.js

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

function fetchAndCache(url) {
  return fetch(url)
    .then(function(response) {
      // Check if we received a valid response
      if (!response.ok) {
        throw Error(response.statusText);
      }
      return caches.open(CACHE_NAME)
        .then(function(cache) {
          cache.put(url, response.clone());
          return response;
        });
    })
    .catch(function(error) {
      console.log('Request failed:', error);
      // You could return a custom offline 404 page here
    });
}
```

Explanation

In the example, a fetch event listener is added to the service worker. When a resource is requested (a fetch event), the service worker intercepts the request and runs this code. The code does the following:

- Tries to match the request with the content of the cache and if the resource is in the cache, then returns it.
- If the resource is not in the cache, attempts to get the resource from the network using fetch.
- If the response is invalid, throws an error and logs a message to the console (`catch`).
- If the response is valid, creates a copy of the response (`clone`), stores it in the cache, and then returns the original response.

Not only does this prioritize getting resources from the cache instead of the network, but it also caches all future requests.

We `clone` the response because the request is a stream that can only be consumed once. Since we want to put it in the cache and serve it to the user, we need to create a copy. See

Jake Archibald's [What happens when you read a response](#) article for a more in-depth explanation.

What have we done?

When the app opens for the first time, the service worker is registered, installed, and activated. During installation, the app caches static assets (the main HTML and CSS). On future loads, each time a resource is requested the service worker intercepts the request, and checks the cache for the resource before going to the network. If the resource isn't cached, the service worker fetches it from the network and caches a copy of the response.

After the first user visit, the app will open even when offline!

Note: You might be thinking, why didn't we just cache everything on install? Or, why did we cache anything on install if all fetched resources are cached? This is intended as an overview of how you can bring offline functionality to an app. In practice, there are a variety of caching strategies and tools that let you customize your app's offline experience. Check out the [Offline Cookbook](#) for more info.

Further reading

- [Is ServiceWorker Ready?](#)
- [ServiceWorker interface](#) (MDN)
- [Introduction to service workers](#)
- [Fetch Event](#) (MDN)

Working with Promises

Contents

[Introduction](#)

[Why use promises?](#)

[Promise terminology](#)

[How to use promises](#)

[Further reading](#)

Codelab: [Promises](#)

Introduction

[Promises](#) offer a better 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.

See [Can I Use](#) for an up-to-date list of browsers that support promises.

Why use promises?

Asynchronous APIs are common in JavaScript to access the network or disk, to communicate with web workers and service workers, and even when using a timer. Most of these APIs use callback functions or events to communicate when a request is ready or has failed. While these techniques worked well in the days of simple web pages, they don't scale well to complete web applications.

The old way: using events

Using events to report asynchronous results has some major drawbacks:

- It fragments your code into many pieces scattered among event handlers.
- It's possible to get into race conditions between defining the handlers and receiving the events.
- It often requires creating a class or using globals just to maintain state.

These make error handling difficult. For an example, look at any XMLHttpRequest code.

The old way: using callbacks

Another solution is to use callbacks, typically with anonymous functions. An example might look like the following:

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

The callback approach has two problems:

- The more callbacks that you use in a callback chain, the harder it is to read and analyze its behavior.
- Error handling becomes problematic. For example, what happens if a function receives an illegal value and/or throws an exception?

Using promises

Promises provide a standardized way to manage asynchronous operations and handle errors. The above example becomes much simpler using promises:

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

Think of a promise as an object that waits for an asynchronous action to finish, then calls a second function. You can schedule that second function by calling `.then()` and passing in the function. When the asynchronous function finishes, it gives its result to the promise and the promise gives that to the next function (as a parameter).

Notice that there are several calls to `.then()` in a row. Each call to `.then()` waits for the previous promise, runs the next function, then converts the result to a promise if needed. This lets you painlessly chain synchronous and asynchronous calls. It simplifies your code so much that most new web specifications return promises from their asynchronous methods.

Promise terminology

When working with promises you may hear terminology commonly associated with callbacks or other asynchronous code.

In the following example, we convert the asynchronous task of setting an image `src` attribute into a promise.

```
function loadImage(url) {
  // wrap image loading in a promise
  return new Promise(function(resolve, reject) {
    // A new promise is "pending"
    var image = new Image();
    image.src = url;
    image.onload = function() {
      // Resolving a promise changes its state to "fulfilled"
      // unless you resolve it with a rejected promise
      resolve(image);
    };
    image.onerror = function() {
      // 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 resolved 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, and is either fulfilled or rejected.

How to use promises

Writing a simple promise

Here's a typical pattern for creating a promise:

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

The promise constructor takes one argument—a callback with two parameters: `resolve` and `reject`. Do something within the callback, perhaps async, then call `resolve` if everything worked, or otherwise call `reject`.

Like `throw` in plain old JavaScript, it's customary, but not required, to reject with an `Error` object. The benefit of `Error` objects is that they capture a stack trace, making debugging tools more helpful.

Here's one way to use that promise:

```
promise.then(function(result) {
  console.log("Success!", result); // "Stuff worked!"
}, function(err) {
  console.log("Failed!", err); // Error: "It broke"
});
```

The `then()` method takes two arguments, a callback for a success case, and another for the failure case. Both are optional, so you can add a callback for the success or failure case only.

A more common practice is to use `.then()` for success cases and `.catch()` for errors.

```
promise.then(function(result) {  
  console.log("Success!", result);  
}).catch(function(error) {  
  console.log("Failed!", error);  
})
```

There's nothing special about `catch()`, it's equivalent to `then(undefined, func)`, but it's more readable. **Note that the two code examples above do not behave the same way.** The latter example is equivalent to:

```
promise.then(function(response) {  
  console.log("Success!", response);  
}).then(undefined, function(error) {  
  console.log("Failed!", error);  
})
```

The difference is subtle, but extremely useful. Promise rejections skip forward to the next `then()` with a rejection callback (or `catch()`, since they're equivalent). With `then(func1, func2)`, `func1` or `func2` will be called, never both. But with `then(func1).catch(func2)`, both will be called if `func1` rejects, as they're separate steps in the chain.

Promise chains: then and catch

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

Then

The `then()` method schedules a function to be called when the previous promise is fulfilled. When the promise is fulfilled, `.then()` extracts the promise's value (the value the promise resolves to), executes the callback function, and wraps the returned value in a new promise.

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

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

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

Calling `.then()` gets the returned value from the previous 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. You can chain any number of actions using `.then()`.

Catch

Promises also provide a mechanism to simplify error handling. When a promise rejects (or throws an exception), it jumps to the first `.catch()` call following the error and passes control to its function.

Think of the part of the chain preceding `catch` as being wrapped in an implicit `try { }` block.

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 (if either the original promise or any of the subsequent steps rejects) we jump to the `catch()` statement.

Only the last `then()` statement will attach the image to the DOM. Until then, we `return` the same image so that the image will be passed to the next `then()`.

```
function processImage(imageName, domNode) {
  // returns an image for the next step. The function called in
  // the return statement must also return the image.
  // The same is true in each step below.
  return loadImage(imageName)
    .then(function(image) {
      // returns an image for the next step.
      return scaleToFit(150, 225, image);
    })
    .then(function(image) {
      // returns the image for the next step.
      return watermark('Google Chrome', image);
    })
    .then(function(image) {
      // Attach the image to the DOM after all processing has been completed.
      // This step does not need to return in the function or here in the
      // .then() because we are not passing anything on
      showImage(image);
    })
    .catch(function(error) {
      console.log('We had a problem in running processImage', error);
    });
}
```

You can use multiple catches in a promise chain to "recover" from errors in a promise chain. For example, the following code continues on with a fallback image if `processImage` or `scaleToFit` rejects:

```
function processImage(imageName, domNode) {
  return loadImage(imageName)
    .then(function(image) {
      return scaleToFit(150, 225, image);
    })
    .catch(function(error) {
      console.log('Error in loadImage() or scaleToFit()', error);
      console.log('Using fallback image');
      return fallbackImage();
    })
    .then(function(image) {
      return watermark('Google Chrome', image);
    })
    .then(function(image) {
      showImage(image);
    })
    .catch(function(error) {
      console.log('We had a problem with watermark() or showImage()', error);
    });
}
```

Note: The promise chain continues executing after a `catch()` until it reaches the last `then()` or `catch()` in the chain.

Synchronous operations

Not all promise-related functions have to return a promise. If the functions in a promise chain are synchronous, they don't need to return a promise.

The `scaleToFit` function is part of the image processing chain and doesn't return a promise:

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

However, this function does need to return the image passed into it so that it can be passed to the next function in the chain.

Promise.all

Often we want to take action only after a collection of asynchronous operations have completed successfully. **Promise.all** returns a promise that resolves if all of the promises passed into it resolve. If any of the passed-in promises reject, then `Promise.all` rejects with the reason of the first promise that rejected. This is very useful for ensuring that a group of asynchronous actions complete before proceeding to another step.

In the example below, `promise1` and `promise2` return promises. We want both of them to load before proceeding. Both promises are passing into `Promise.all`. If either request rejects, then `Promise.all` rejects with the value of the rejected promise. If both requests fulfill, `Promise.all` resolves with the values of both promises (as a list).

```
var promise1 = getJSON('/users.json');
var promise2 = getJSON('/articles.json');

Promise.all([promise1, promise2]) // Array of promises to complete
  .then(function(results) {
    console.log('all data has loaded');
  })
  .catch(function(error) {
    console.log('one or more requests have failed: ' + error);
  });
```

Note: Even if an input promise rejects, causing `Promise.all` to reject, the remaining input promises still settle. In other words, the remaining promises still execute, they simply are not returned by `Promise.all`.

Promise.race

Another promise method that you may see referenced is **Promise.race**. `Promise.race` takes a list of promises and settles as soon as the first promise in the list settles. If the first promise resolves, `Promise.race` resolves with the corresponding value, if the first promise rejects, `Promise.race` rejects with the corresponding reason. The following code shows example usage of `Promise.race`:

```
Promise.race([promise1, promise2])
  .then(function(value) {
    console.log(value);
  })
  .catch(function(reason) {
    console.log(reason);
  });
```

If one of the promises resolves first, the `then` block executes and logs the value of the resolved promise. If one of the promises rejects first, the `catch` block executes and logs the reason for the promise rejection.

It may still be tempting, however, to use `Promise.race` to race promises, as the name suggests. Consider the following example:

```
var promise1 = new Promise(function(resolve, reject) {
  // something that fails
});

var promise2 = new Promise(function(resolve, reject) {
  // something that succeeds
});

Promise.race([promise1, promise2])
  .then(function(value) {
    // Use whatever returns fastest
  })
  .catch(function(reason) {
    console.log(reason);
  });
```

At first glance it looks like this code races two promises—one that rejects, and another that resolves—and uses the first one to return. However, `Promise.race` rejects immediately if one of the supplied promises rejects, even if another supplied promise resolves later. So if `promise1` rejects before `promise2` resolves, `Promise.race` will reject even though `promise2` supplies a valid value. `Promise.race` by itself can't be used to reliably return the first promise that resolves.

Another pattern that may be appealing is the following:

```
var promise1 = new Promise(function(resolve, reject) {
  // get a resource from the Cache
});

var promise2 = new Promise(function(resolve, reject) {
  // Fetch a resource from the network
});

Promise.race([promise1, promise2])
  .then(function(resource) {
    // Use the fastest returned resource
  })
  .catch(function(reason) {
    console.log(reason);
  });
```


This example appears to race the cache against the network, using the fastest returned resource. However, both the [Cache API](#) and [Fetch API](#) can resolve with "bad" responses ([fetch](#) resolves even for 404s, and [caches.match](#) resolves with falsey values if a resource is not available). In this example, if a resource is not available in the cache (which typically responds faster than the network), `Promise.race` resolves with the falsey value from the cache, and ignores the network request (which may resolve). See the [Cache & network race](#) section in the [Offline Cookbook](#) for an example of a race function that works as expected.

Further reading

- [JavaScript Promises: an Introduction](#)
- [Promise - MDN](#)

Working with the Fetch API

Contents

[What is fetch?](#)

[Making a request](#)

[Reading the response object](#)

[Making custom requests](#)

[Cross-origin requests](#)

[Further reading](#)

Codelab: [Fetch API](#)

What is fetch?

The [Fetch API](#) is a simple interface for fetching resources. Fetch makes it easier to make web requests and handle responses than with the older [XMLHttpRequest](#), which often requires additional logic (for example, for handling redirects).

Note: Fetch supports the [Cross Origin Resource Sharing \(CORS\)](#). Testing generally requires running a local server. Note that although fetch does not require HTTPS, service workers do and so using fetch in a service worker requires HTTPS. Local servers are exempt from this.

You can check for browser support of fetch in the window interface. For example:

main.js

```
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](#) (but see the readme for important caveats.).

The `fetch()` method takes the 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:

main.js

```
fetch('examples/example.json')
  .then(function(response) {
    // Do stuff with the response
  })
  .catch(function(error) {
    console.log('Looks like there was a problem: \n', error);
  });
```

We pass the path for the resource we want to retrieve as a parameter to `fetch`. In this case this is **examples/example.json**. The `fetch` call returns a promise that resolves 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 404s) still resolve. The only time a `fetch` promise will reject is if the request was unable to complete. The previous code segment would only fall back to `.catch` if there was no network connection, but not if the response was bad (like a 404). If the previous code were updated to validate responses it would look like:

main.js

```
fetch('examples/example.json')
  .then(function(response) {
    if (!response.ok) {
      throw Error(response.statusText);
    }
    // Do stuff with the response
  })
  .catch(function(error) {
    console.log('Looks like there was a problem: \n', error);
  });
```

Now if the response object's `ok` property is false (indicating a non 200-299 response), the function throws an error containing `response.statusText` that triggers the `.catch` block. This prevents bad responses from propagating down the fetch chain.

Reading the response object

Responses must be read in order to access the body of the response. Response objects have [methods](#) for doing this. 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:

main.js

```
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('Looks like there was a problem: \n', error);
  });
```

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

main.js

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

function logError(error) {
  console.log('Looks like there was a problem: \n', error);
}

function validateResponse(response) {
  if (!response.ok) {
    throw Error(response.statusText);
  }
  return response;
}

function readResponseAsJSON(response) {
  return response.json();
}

function fetchJSON(pathToResource) {
  fetch(pathToResource) // 1
    .then(validateResponse) // 2
    .then(readResponseAsJSON) // 3
    .then(logResult) // 4
    .catch(logError);
}

fetchJSON('examples/example.json');
```

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

To summarize what's happening:

Step 1. Fetch is called on a resource, **examples/example.json**. Fetch returns a promise that will resolve to a response object. When the promise resolves, the response object is passed to `validateResponse`.

Step 2. `validateResponse` checks if the response is valid (is it a 200-299?). If it isn't, an error is thrown, skipping the rest of the `then` blocks and triggering the `catch` block. This is particularly important. Without this check bad responses are passed down the chain and could break later code that may rely on receiving a valid response. If the response is valid, it is passed to `readResponseAsJSON`.

Note: You can also handle any network status code using the `status` property of the `response` object. This lets you respond with custom pages for different errors or handle other responses that are not `ok` (i.e., not 200-299), but still usable (e.g., status codes in the 300 range). See [Caching files with the service worker](#) for an example of a custom response to a 404.

Step 3. `readResponseAsJSON` reads the body of the response using the [Response.json\(\)](#) method. This method returns a promise that resolves to JSON. Once this promise resolves, the JSON data is passed to `logResult`. (Can you think of what would happen if the promise from `response.json()` rejects?)

Step 4. Finally, the JSON data from the original request to **examples/example.json** is logged by `logResult`.

For more information

- [Response interface](#)
- [Response.json\(\)](#)
- [Promise chaining](#)

Example: fetching images

Let's look at an example of fetching an image and appending it to a web page.

main.js

```
function readResponseAsBlob(response) {
  return response.blob();
}

function showImage(responseAsBlob) {
  // Assuming the DOM has a div with id 'container'
  var container = document.getElementById('container');
  var imgElem = document.createElement('img');
  container.appendChild(imgElem);
  var imgUrl = URL.createObjectURL(responseAsBlob);
  imgElem.src = imgUrl;
}

function fetchImage(pathToResource) {
  fetch(pathToResource)
    .then(validateResponse)
    .then(readResponseAsBlob)
    .then(showImage)
    .catch(logError);
}

fetchImage('examples/kitten.jpg');
```

In this example an image (**examples/kitten.jpg**) is fetched. As in the previous example, the response is validated with `validateResponse`. The response is then read as a `Blob` (instead of as JSON), and an image element is created and appended to the page, and the image's `src` attribute is set to a data URL representing the Blob.

Note: The `URL object's createObjectURL() method` is used to generate a data URL representing the Blob. This is important to note as you cannot set an image's source directly to a Blob. The Blob must first be converted into a data URL.

For more information

- [Blobs](#)
- [Response.blob\(\)](#)
- [URL object](#)

Example: fetching text

Let's look at another example, this time fetching some text and inserting it into the page.

main.js

```
function readResponseAsText(response) {
  return response.text();
}

function showText(responseAsText) {
  // Assuming the DOM has a div with id 'message'
  var message = document.getElementById('message');
  message.textContent = responseAsText;
}

function fetchText(pathToResource) {
  fetch(pathToResource)
    .then(validateResponse)
    .then(readResponseAsText)
    .then(showText)
    .catch(logError);
}

fetchText('examples/words.txt');
```

In this example a text file is being fetched, **examples/words.txt**. Like the previous two exercises, the response is validated with `validateResponse`. Then the response is read as text, and appended to the page.

Note: It may be tempting to fetch HTML and append that using the `innerHTML` attribute, but be careful -- this can expose your site to [cross site scripting attacks](#)!

For more information

- [Response.text\(\)](#)

Note: For completeness, the methods we have used are actually methods of [Body](#), a Fetch API [mixin](#) that is implemented in the Response object.

Making custom 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](#).

Example: HEAD requests

By default fetch uses the GET method, which retrieves a specific resource, but other request HTTP methods can also be used.

HEAD requests are just like GET requests except the body of the response is empty. You can use this kind of request when all you want the file's metadata, and you want or need the file's data to be transported.

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

main.js

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

This will make a HEAD request for **examples/words.txt**.

You could use a HEAD request to check the size of a resource. For example:

main.js


```
function checkSize(response) {
  var size = response.headers.get('content-length');
  // Do stuff based on response size
}

function headRequest(pathToResource) {
  fetch(pathToResource, {
    method: 'HEAD'
  })
  .then(validateResponse)
  .then(checkSize)
  // ...
  .catch(logError);
}

headRequest('examples/words.txt');
```

Here the HEAD method is used to request the size (in bytes) of a resource (represented in the **content-length** header) without actually loading the resource itself. In practice this could be used to determine if the full resource should be requested (or even how to request it).

Example: POST requests

Fetch can also send data to an API with POST requests. The following code sends a "title" and "message" (as a string) to **someurl/comment**:

main.js

```
fetch('someurl/comment', {
  method: 'POST',
  body: 'title=hello&message=world'
})
```

Note: In production, remember to always encrypt any sensitive user data.

The method is again specified with the `init` parameter. This is also where the body of the request is set, which represents the data to be sent (in this case the title and message).

The body data could also be extracted from a form using the [FormData](#) interface. For example, the above code could be updated to:

main.js

```
// Assuming an HTML <form> with id of 'myForm'
fetch('someurl/comment', {
  method: 'POST',
  body: new FormData(document.getElementById('myForm'))
})
```

Custom headers

The `init` parameter can be used with the [Headers](#) interface to perform various actions on HTTP request and response headers, including retrieving, setting, adding, and removing them. An example of reading response headers was shown in a [previous section](#). The following code demonstrates how a custom [Headers](#) object can be created and used with a fetch request:

main.js

```
var myHeaders = new Headers({
  'Content-Type': 'text/plain',
  'X-Custom-Header': 'hello world'
});

fetch('/someurl', {
  headers: myHeaders
});
```

Here we are creating a Headers object where the `Content-Type` header has the value of `text/plain` and a custom `X-Custom-Header` header has the value of `hello world`.

Note: Only some headers, like `Content-Type` can be modified. Others, like `Content-Length` and `origin` are [guarded](#), and cannot be modified (for security reasons).

Custom headers on [cross-origin](#) requests must be supported by the server from which the resource is requested. The server in this example would need to be configured to accept the `X-Custom-Header` header in order for the fetch to succeed. When a custom header is set, the browser performs a [preflight](#) check. This means that the browser first sends an `OPTIONS` request to the server to determine what HTTP methods and headers are allowed by the server. If the server is configured to accept the method and headers of the original request, then it is sent. Otherwise, an error is thrown.

For more information

- [Headers](#)
- [Preflight checks](#)

Cross-origin requests

Fetch (and XMLHttpRequest) follow the [same-origin policy](#). This means that browsers restrict [cross-origin](#) HTTP requests from within scripts. A cross-origin request occurs when one domain (for example <http://foo.com/>) requests a resource from a separate domain (for example <http://bar.com/>). This code shows a simple example of a cross-origin request:

main.js

```
// From http://foo.com/  
fetch('http://bar.com/data.json')  
.then(function(response) {  
  // Do something with response  
});
```

Note: Cross-origin request restrictions are often a point of confusion. Many resources like images, stylesheets, and scripts are fetched cross-origin. However, these are exceptions to the same-origin policy. Cross-origin requests are still restricted *from within scripts*. There have been attempts to work around the same-origin policy (such as [JSONP](#)). The [Cross Origin Resource Sharing](#) (CORS) mechanism has enabled a standardized means of retrieving cross-origin resources. The CORS mechanism lets you specify in a request that you want to retrieve a cross-origin resource (in fetch this is enabled by default). The browser adds an `origin` header to the request, and then requests the appropriate resource. The browser only returns the response if the server returns an `Access-Control-Allow-Origin` header specifying that the origin has permission to request the resource. In practice, servers that expect a variety of parties to request their resources (such as 3rd party APIs) set a wildcard value for the `Access-Control-Allow-Origin` header, allowing anyone to access that resource.

If the server you are requesting from doesn't support CORS, you should get an error in the console indicating that the cross-origin request is blocked due to the CORS `Access-Control-Allow-Origin` header being missing.

You can use `no-cors` mode to request opaque resources. [Opaque responses](#) can't be accessed with JavaScript but the response can still be served or cached by a service worker. Using `no-cors` mode with fetch is relatively simple. To update the above example with `no-cors`, we pass in the `init` object with `mode` set to `no-cors`:

main.js

```
// From http://foo.com/  
fetch('http://bar.com/data.json', {  
  mode: 'no-cors' // 'cors' by default  
})  
.then(function(response) {  
  // Do something with response  
});
```

For more information

- [Cross Origin Resource Sharing](#)

Further reading

- [Fetch API Codelab](#)
- [Learn more about the Fetch API](#)
- [Learn more about Using Fetch](#)
- [Learn more about GlobalFetch.fetch\(\)](#)
- [Get an Introduction to Fetch](#)
- [David Welsh's blog on fetch](#)
- [Jake Archibald's blog on fetch](#)

Caching Files with Service Worker

Contents

[Using the Cache API in the service worker](#)

[Using the Cache API](#)

[Further reading](#)

Codelab: [Caching Files with Service Worker](#)

Using the Cache API in the service worker

The Service Worker API comes with a [Cache interface](#), that lets you create stores of responses keyed by request. While this interface 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. However, if the amount of cached data exceeds the browser's storage limit, the browser will begin evicting all data associated with an origin, one origin at a time, until the storage amount goes under the limit again. See [Browser storage limits and eviction criteria](#) for more information.

Storing resources

In this section, we outline a few common patterns for caching resources: *on service worker install*, *on user interaction*, and *on network response*. There are a few patterns we don't cover here. See the [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(event) {
  event.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.

Note: 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 must not disrupt that. For instance, this is not a good place to delete old caches, because the previous service worker may still be using them at this point.

`event.waitUntil` extends the lifetime of the `install` event until the passed promise resolves successfully. If the promise rejects, the installation is considered a failure and this service worker is abandoned (if an older version is running, it stays active).

`cache.addAll` will reject if any of the resources fail to cache. This means the service worker will only install if all of the resources in `cache.addAll` have been cached.

On user interaction

If the whole site can't be taken offline, you can let the user select the content they want available offline (for example, a video, article, or photo gallery).

One method is to 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);
    });
  });
});
```

In the above example, when the user clicks an element with the `cache-article` class, we are getting the article ID, fetching the article with that ID, and adding the article to the cache.

Note: The Cache API is available on the window object, meaning you don't need to involve the service worker to add things to the cache.

On network response

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.

```
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;
        });
      });
    })
  );
});
```

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 you do this for a range of URLs, be careful not to 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.

Note: 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 a copy of the response that can be

read separately. See [What happens when you read a response?](#) for more information.

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
- 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, and are not part of the "version" of the site (for example, articles, avatars, social media timelines, game leader boards). Handling network requests this way means the online users get the most up-to-date content, and 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 also a good approach for resources that update frequently. 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, and one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

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.

Sometimes you can replace the current data when new data arrives (for example, game leaderboard), but be careful not to hide or replace something the user may be interacting with. For example, if you load a page of blog posts from the cache and then add new posts to the top of the page as they are fetched from the network, you might consider adjusting the scroll position so the user is uninterrupted. This can be a good solution if your app layout is fairly linear.

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.

You can also provide different fallbacks based on the network error:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      if (response) {
        return response;
      }
      return fetch(event.request).then(function(response) {
        if (response.status === 404) {
          return caches.match('pages/404.html');
        }
        return response;
      });
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
    })
  );
});
```

Network response errors do not throw an error in the `fetch` promise. Instead, `fetch` returns the response object containing the error code of the network error. This means we handle network errors in a `.then` instead of a `.catch`.

Removing outdated caches

Once a new service worker has installed and 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 using it for things you couldn't do while the old version was active.

Using the Cache API

Here we cover the Cache API properties and methods.

Checking for support

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

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

Creating 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 returns a promise that resolves to the cache object. `caches.open` accepts a string that will be the name of the cache.

Working with data

The Cache API comes with several methods that let us create and manipulate data in the cache. These can be grouped into methods that either create, match, or delete data.

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 lets you manually insert the response object. Often, you will just want to `fetch()` one or more requests and then add the result straight to your cache. In such cases you are better off just using `cache.add` or `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 of 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 returns `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 — that is, 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 it 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 returns 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` returns all of the requests in the cache. If a request is passed, it returns 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 lets you purge outdated caches in one go.

Further reading

Learn about the Cache API

- [Cache](#) - MDN
- [The Offline Cookbook](#)

Learn about using service workers

- [Using Service Workers](#)

Working with IndexedDB

Contents

[Introduction](#) [What is IndexedDB?](#) [IndexedDB terms](#) [Check for IndexedDB support](#)
[Opening a database](#) [Working with object stores](#) [Working with data](#) [Getting all the data](#)
[Using database versioning](#) [Further reading](#) [Appendix](#)

Codelab: [IndexedDB](#)

Introduction

This text 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.

What is IndexedDB?

IndexedDB is a large-scale, noSQL storage system. It lets you 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 of IndexedDB on 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 [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 is one database per app.

Object store - An object store is an individual bucket to store data. You can think of object stores as being 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 could imagine two object stores. Unlike tables in traditional databases, the actual JavaScript data types of data within the store do not need to be consistent (for example, 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, that ensures 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.

Checking for IndexedDB support

Because IndexedDB isn't supported by all browsers, we need to check that the [user's browser supports it](#) before using it. The easiest way is to check the window object:

```
if (!('indexedDB' in window)) {  
  console.log('This browser doesn\'t support IndexedDB');  
  return;  
}
```

We simply place this function at the beginning of our scripts and we're ready to use IndexedDB.

Opening a database

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.

Here is an example of `idb.open` in context:

```
(function() {
  'use strict';

  //check for support
  if (!('indexedDB' in window)) {
    console.log('This browser doesn\'t support IndexedDB');
    return;
  }

  var dbPromise = idb.open('test-db1', 1);

})();
```

We place our check for IndexedDB support at the top of the anonymous function. This exits out of the function if the browser doesn't support IndexedDB. We call `idb.open` to open a database named "test-db1". We have left out the optional callback function in this first example to keep things simple.

Working with object stores

Creating object stores

A database typically contains 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" (not JavaScript data 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 should have 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. 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.

Below is an example of the `createObjectStore` method:

```
(function() {
  'use strict';

  //check for support
  if (!('indexedDB' in window)) {
    console.log('This browser doesn\'t support IndexedDB');
    return;
  }

  var dbPromise = idb.open('test-db2', 1, function(upgradeDb) {
    console.log('making a new object store');
    if (!upgradeDb.objectStoreNames.contains('firstOS')) {
      upgradeDb.createObjectStore('firstOS');
    }
  });
})();
```

Again, we first check the browser for IndexedDB support. This time 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 we wrap the `createObjectStore` method in an `if` statement that checks if the object store exists. Inside the `if` block we call `createObjectStore` on the `UpgradeDB` object to create an object store named "firstOS".

Defining primary keys

When you define object stores, you can define how data is uniquely identified in the store using the primary key. You can define a primary key by either defining a key path, or by using a key generator.

A *key path* is a property that always exists and contains a unique value. For example, in the case of a "people" object store we could choose the email address as the key path.

```
upgradeDb.createObjectStore('people', {keyPath: 'email'});
```

This example creates an object store called "people" and assigns the "email" property as the primary key.

You could also 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 separately from the data.

```
upgradeDb.createObjectStore('notes', {autoIncrement:true});
```

This example creates an object store called "notes" and sets the primary key to be assigned automatically as an auto incrementing number.

```
upgradeDb.createObjectStore('logs', {keyPath: 'id', autoIncrement:true});
```

This example is similar to the previous example, but this time the auto incrementing value is assigned to a property called "id".

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.

Let's look at an example:

```
function() {
  'use strict';

  //check for support
  if (!('indexedDB' in window)) {
    console.log('This browser doesn\'t support IndexedDB');
    return;
  }

  var dbPromise = idb.open('test-db3', 1, function(upgradeDb) {
    if (!upgradeDb.objectStoreNames.contains('people')) {
      upgradeDb.createObjectStore('people', {keyPath: 'email'});
    }
    if (!upgradeDb.objectStoreNames.contains('notes')) {
      upgradeDb.createObjectStore('notes', {autoIncrement: true});
    }
    if (!upgradeDb.objectStoreNames.contains('logs')) {
      upgradeDb.createObjectStore('logs', {keyPath: 'id', autoIncrement: true});
    }
  });
  }());
```

This code creates three object stores demonstrating the various ways of defining primary keys in object stores.

Defining indexes

Indexes are a kind of object store used to retrieve data from the reference object store by a specified 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('indexName', 'property', 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 want to index. The final argument lets you define two 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.

Here is an example:

```
(function() {
  'use strict';

  //check for support
  if (!('indexedDB' in window)) {
    console.log('This browser doesn\'t support IndexedDB');
    return;
  }

  var dbPromise = idb.open('test-db4', 1, function(upgradeDb) {
    if (!upgradeDb.objectStoreNames.contains('people')) {
      var peopleOS = upgradeDb.createObjectStore('people', {keyPath: 'email'});
      peopleOS.createIndex('gender', 'gender', {unique: false});
      peopleOS.createIndex('ssn', 'ssn', {unique: true});
    }
    if (!upgradeDb.objectStoreNames.contains('notes')) {
      var notesOS = upgradeDb.createObjectStore('notes', {autoIncrement: true});
      notesOS.createIndex('title', 'title', {unique: false});
    }
    if (!upgradeDb.objectStoreNames.contains('logs')) {
      var logsOS = upgradeDb.createObjectStore('logs', {keyPath: 'id',
        autoIncrement: true});
    }
  });
})();
```

In this example, the "people" and "notes" object stores have indexes. To create the indexes, we first assign the result of `createObjectStore` (which is an object store object) to a variable so we can call `createIndex` on it.

Note: Indexes are updated every time you write data to the reference object store. More indexes mean more work for IndexedDB.

Working with data

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

All data operations in IndexedDB are carried out inside 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 read the data or make a change to the database.

Creating data

To create data, call the `add` method on the object store and pass in the data you want to add. `Add` has an optional second argument that lets you define the primary key for the individual object on creation, but it 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, and so forth. 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.

`Add` occurs within a transaction, so even if the promise resolves successfully it doesn't necessarily mean the operation worked. Remember, if one of the actions in the transaction fails, all of the operations in the transaction are rolled back. To be sure that the `add` operation was carried out, we need to check if the whole transaction has completed using the `transaction.complete` method. `transaction.complete` is a promise that resolves when the transaction completes and rejects if the transaction errors. Note that this method doesn't actually close the transaction. The transaction completes on its own. We must perform this check for all "write" operations, because it is our only way of knowing that the changes to the database have actually been carried out.

Let's look at an example of the `add` method:


```
dbPromise.then(function(db) {
  var tx = db.transaction('store', 'readwrite');
  var store = tx.objectStore('store');
  var item = {
    name: 'sandwich',
    price: 4.99,
    description: 'A very tasty sandwich',
    created: new Date().getTime()
  };
  store.add(item);
  return tx.complete;
}).then(function() {
  console.log('added item to the store os!');
});
```

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 in `.then`. 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 example it is just the "store" object store). The transaction method also has an optional second argument for the mode, which can be `readonly` or `readwrite`. 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 operation occurs within the transaction. Finally, we return `tx.complete` and log a success message once the transaction has completed.

Reading data

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

Let's look at an example of the `get` method:

```
dbPromise.then(function(db) {  
  var tx = db.transaction('store', 'readonly');  
  var store = tx.objectStore('store');  
  return store.get('sandwich');  
}).then(function(val) {  
  console.dir(val);  
});
```

Once again, 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 (that is, 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 to the console.

Note: If you try to get an object that doesn't exist, the success handler still executes, but the result is `undefined`.

Updating data

To update data, 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 occurs inside a transaction. As with `add`, we need to be careful to check `transaction.complete` if we want to be sure that the operation was actually carried out.

Here is an example using the `put` method:

```
dbPromise.then(function(db) {
  var tx = db.transaction('store', 'readwrite');
  var store = tx.objectStore('store');
  var item = {
    name: 'sandwich',
    price: 99.99,
    description: 'A very tasty, but quite expensive, sandwich',
    created: new Date().getTime()
  };
  store.put(item);
  return tx.complete;
}).then(function() {
  console.log('item updated!');
});
```

To update an existing item in the object store, use the `put` method on an object containing the same primary key value as the object in the store. We are assuming the keyPath for the store object store is the "name" property and we are updating the price and description of our "sandwich" object. The database interaction has the same structure as the create and read operations: get the database object, create a transaction, open an object store on the transaction, perform the operation on the object store.

Deleting data

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

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

Once again, this method returns a promise and must be wrapped in a transaction. Here is a simple example:

```
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');
});
```

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 was carried out.

Getting all the data

So far we have only retrieved objects from the store one at a time. We can also retrieve all of the data (or subset) from an object store or index using either the `getAll` method or using cursors.

Using the `getAll` method

The simplest way to retrieve all of the data is to call the `getAll` method on the object store or index, like this:

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

This method returns 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. As with all other database operations, this operation happens inside a transaction. Here is a short example:

```
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);  
});
```

Here we are calling `getAll` on the "store" object store. This returns all of the objects in the store ordered by the primary key.

Using cursors

Another way to retrieve all of the data is to use a cursor. A cursor selects each object in an object store or index one by one, letting you do something with the data as it is selected. Cursors, like the other database operations, work within transactions.

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

```
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 moves the cursor object onto the next object or returns `undefined` if there isn't another object. We put this inside a loop to move through all of the entries in the store one by one. The optional key range 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.

The next example uses a cursor to iterate through all the items in the "store" object store and log them to the console:

```
dbPromise.then(function(db) {
  var tx = db.transaction('store', 'readonly');
  var store = tx.objectStore('store');
  return store.openCursor();
}).then(function logItems(cursor) {
  if (!cursor) {return;}
  console.log('Cursored at:', cursor.key);
  for (var field in cursor.value) {
    console.log(cursor.value[field]);
  }
  return cursor.continue().then(logItems);
}).then(function() {
  console.log('Done cursoring');
});
```

As usual, we start by getting the database object, creating a transaction, and opening an object store. We call the `openCursor` method on the object store and pass the cursor object to the callback function in `.then`. This time we name the callback function "logItems" 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` (that is, 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. At the end of `logItems`, we return `cursor.continue().then(logItems)`. `cursor.continue` that resolves to a cursor object representing the next item in the store or `undefined` if it doesn't exist. This is passed to the callback function in `.then`, which we have chosen to be `logItems`, so that the function loops. `logItems` continues to call itself until `cursor.continue` runs out of objects.

Working with ranges and indexes

We can get all the data in a couple of different ways, but what if we want only a subset of the data based on a particular property? This is where indexes come in. Indexes let us 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 that 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 one argument which is be 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 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.

Let's look at an example. For this demo, we have created an index on the "price" property in the "store" object store. We have also added a small form with two inputs for the upper and lower limits of the range. Imagine we are passing in the lower and upper bounds to the function as floating point numbers representing prices:

```
function searchItems(lower, upper) {
  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);
  }

  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.key);
    for (var field in cursor.value) {
      console.log(cursor.value[field]);
    }
    return cursor.continue().then(showRange);
  }).then(function() {
    console.log('Done cursoring');
  });
}
```

The code first gets the values for the limits and checks if the limits exist. The next block of code decides which method to use to limit the range based on the values. In the database interaction, we open the object store on the transaction as usual, then we open the "price" index on the object store. The "price" index allows us to search for the items by price. We open a cursor on the index and pass in the range. The cursor now returns a promise representing the first object in the range, or `undefined` if there is no data within the range. `cursor.continue` returns a cursor a object representing the next object and so on through the loop until we reach the end of the range.

Using database versioning

When we call `idb.open`, we can specify the database version number in the second parameter. If this version number is greater than the version of the existing database, the upgrade callback executes, allowing us to add object stores and indexes to the database.

Note: The browser throws an error if we try to create object stores or indexes that already

exist in the database. We can wrap the calls to `createObjectStore` in `if` statements checking if the object store already exists using

`upgradeDb.objectStoreNames.contains('objectStoreName')`. We can also use a `switch` statement on the `oldVersion` property as in the next example.

The UpgradeDB object gets a special `oldVersion` method that 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. Let's look at an example:

```
var dbPromise = idb.open('test-db7', 2, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      upgradeDb.createObjectStore('store', {keyPath: 'name'});
    case 1:
      var peopleStore = upgradeDb.transaction.objectStore('store');
      peopleStore.createIndex('price', 'price');
  }
});
```

In the example we have set the newest version of the database at 2. When this code first executes, since 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. So in our example, the browser continues executing through `case 1`, creating a "price" index on the "store" object store. Once this has finished executing, the database in the browser is at version 2 and contains a "store" object store with a "price" index.

Let's say we now want to create a "description" index on the "store" object store. We need to update the version number and add a case, like this:

```
var dbPromise = idb.open('test-db7', 3, function(upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      upgradeDb.createObjectStore('store', {keyPath: 'name'});
    case 1:
      var storeOS = upgradeDb.transaction.objectStore('store');
      storeOS.createIndex('price', 'price');
    case 2:
      var storeOS = upgradeDb.transaction.objectStore('store');
      storeOS.createIndex('description', 'description');
  }
});
```


Assuming the database we created in the previous example still exists in the browser, when this executes `upgradeDb.oldVersion` is 2. `case 0` and `case 1` are skipped and the browser executes the code in `case 2`, which creates a "description" index. Once all this has finished, the browser has a database at version 3 containing a "store" object store with "price" and "description" indexes.

Further reading

IndexedDB Documentation

- [Using IndexedDB](#) - MDN
- [Basic Concepts Behind indexedDB](#) - MDN
- [Indexed Database API](#) - W3C

Data storage limits

- [Working with quota on mobile browsers](#)
- [Browser storage limits and eviction criteria](#)

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 will act the same way. But there are a few differences because of the differences between requests and promises, which we will cover here.

All database interactions in the IndexedDB API are requests and have associated `onsuccess` and `onerror` event handlers. These are similar to the `.then` and `.catch` promise functions. The `indexedDB.open` method in the raw API 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 will open a database, add an object store, and add one item to the object 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 done with requests and event handlers rather than promises and promise chains.

Here is a short 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>

Live Data in the Service Worker

Contents

[Introduction](#) [Where should offline data be stored?](#) [Using IndexedDB and the Cache interface](#) [Further reading](#)

Introduction

Offline support and reliable performance are key features of Progressive Web Apps. This text describes some recommendations for storing different kinds of data in a PWA.

Where should offline data be stored?

A general guideline for data storage is that URL addressable resources should be stored with the [Cache](#) interface, and other data should be stored with [IndexedDB](#). For example HTML, CSS, and JS files should be stored in the cache, while JSON data should be stored in IndexedDB. Note that this is only a guideline, not a firm rule.

Why IndexedDB and the Cache interface?

There are a [variety of reasons](#) to use IndexedDB and the Cache interface. Both are asynchronous and accessible in service workers, web workers, and the window interface. IndexedDB is [widely supported](#), and the Cache interface [is supported](#) in Chrome, Firefox, Opera, and Samsung Internet.

In this text we use Jake Archibald's [IndexedDB Promised](#) library, which enables promise syntax for IndexedDB. There are also [other IndexedDB libraries](#) that can be used to abstract some of the less convenient aspects of the API.

Debugging support for IndexedDB is available in Chrome, Opera, Firefox and Safari. Debugging support for Cache Storage is available in Chrome, Opera, and Firefox. These are covered in [Tools for PWA Developers](#).

Note: Some developers have run into issues with Safari 10's IndexedDB implementation. Test your app to make sure it works on your target browser. File browser bugs with your

browser's vendor so that browser implementors and library maintainers can investigate.

How Much Can You Store

Different browsers allow different amounts of offline storage. This table summarizes storage limits for major browsers:

Browser	Limitation	Notes
Chrome, Opera, and Samsung Internet	Up to a quota . Check usage with the Quota API	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

Using IndexedDB and the Cache interface

Storing data with IndexedDB

IndexedDB is a noSQL database. IndexedDB data is stored as key-value pairs in **object stores**. The table below shows an example of an object store, in this case containing beverage items:

#	Key (keypath 'id')	Value
0	1234	{id: 123, name: 'coke', price: 10.99, quantity: 200}
1	9876	{id: 321, name: 'pepsi', price: 8.99, quantity: 100}
2	4567	{id: 222, name: 'water', price: 11.99, quantity: 300}

The data is organized by a **keypath**, which in this case is the item's `id` property. You can learn more about IndexedDB in the corresponding [text](#), or in the [code lab](#).

The following function could be used to create an IndexedDB object store like the example above:

service-worker.js

```
function createDB() {
  idb.open('products', 1, function(upgradeDB) {
    var store = upgradeDB.createObjectStore('beverages', {
      keyPath: 'id'
    });
    store.put({id: 123, name: 'coke', price: 10.99, quantity: 200});
    store.put({id: 321, name: 'pepsi', price: 8.99, quantity: 100});
    store.put({id: 222, name: 'water', price: 11.99, quantity: 300});
  });
}
```

Note: All IndexedDB code in this text uses Jake Archibald's [IndexedDB Promised](#) library, which enables promise syntax for IndexedDB.

Here we create a 'products' database, version 1. Inside the 'products' database, we create a 'beverages' object store. This holds all of the beverage objects. The `beverages` object store has a keypath of `id`. This means that the objects in this store will be organized and accessed by the `id` property of the `beverage` objects. Finally, we add some example beverages to the object store.

Note: If you're familiar with IndexedDB, you may be asking why we didn't use a transaction when creating and populating the database. In IndexedDB, a transaction is built into the database creation operation.

The service worker activation event is a good time to create a database. Creating a database during the activation event means that it will only be created (or opened, if it already exists) when a new service worker takes over, rather than each time the app runs (which is inefficient). It's also likely better than using the service worker's installation event, since the old service worker will still be in control at that point, and there could be conflicts if a new database is mixed with an old service worker. The following code (in the service worker file) could be used to create the database shown earlier on service worker activation:

service-worker.js

```
self.addEventListener('activate', function(event) {
  event.waitUntil(
    createDB()
  );
});
```

Note: `event.waitUntil` ensures that a service worker does not terminate during asynchronous operations.

Once an IndexedDB database is created, data can then be read locally from IndexedDB rather than making network requests to a backend database. The following code could be used to retrieve data from the example database above:

service-worker.js

```
function readDB() {
  idb.open('products', 1).then(function(db) {
    var tx = db.transaction(['beverages'], 'readonly');
    var store = tx.objectStore('beverages');
    return store.getAll();
  }).then(function(items) {
    // Use beverage data
  });
}
```

Here we open the `products` database and create a new transaction on the `beverages` store of type `readonly` (we don't need to write data). We then access the store, and retrieve all of the items. These items can then be used to update the UI or perform whatever action is needed.

Note: A transaction is wrapper around an operation, or group of operations, that ensures 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.

Storing assets in the Cache interface

URL addressable resources are comparatively simple to store with the Cache interface. The following code shows an example of caching multiple resources:

service-worker.js

```
function cacheAssets() {
  return caches.open('cache-v1')
    .then(function(cache) {
      return cache.addAll([
        '.',
        'index.html',
        'styles/main.css',
        'js/offline.js',
        'img/coke.jpg'
      ]);
    });
}
```

This code opens a `cache-v1` cache, and stores **index.html**, **main.css**, **offline.js**, and **coke.jpg**.

The service worker installation event is a good time to cache static assets like these. This ensures that all the resources a service worker is expected to have are cached when the service worker is installed. The following code (in the service worker file) could be used to cache these types of files during the service worker install event:

service-worker.js

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    cacheAssets()
  );
});
```

Once assets are cached, they can be retrieved during fetch events. The following code (in the service worker file) allows resources to be fetched from the cache instead of the network:

service-worker.js

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request).then(function(response) {
      // Check cache but fall back to network
      return response || fetch(event.request);
    })
  );
});
```


This code adds a `fetch` listener on the service worker that attempts to get resources from the cache before going to the network. If the resource isn't found in the cache, a regular network request is still made.

Further reading

- [Offline Storage for Progressive Web Apps](#)
- [IndexedDB Promised](#)
- [Support for the Cache interface](#)
- [Support for IndexedDB](#)

Lighthouse PWA Analysis Tool

Contents

[Introduction](#) [Running Lighthouse as a Chrome extension](#) [Running Lighthouse from the command line](#)

Codelab: [Auditing with Lighthouse](#)

Introduction

How do I tell if all of my Progressive Web App (PWA) features are 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 PWA with a full application-like experience for your users.


Lighthouse tests if your app:

- Can load in offline or flaky network conditions
- Is relatively fast
- Is served from a secure origin
- Uses certain accessibility best practices

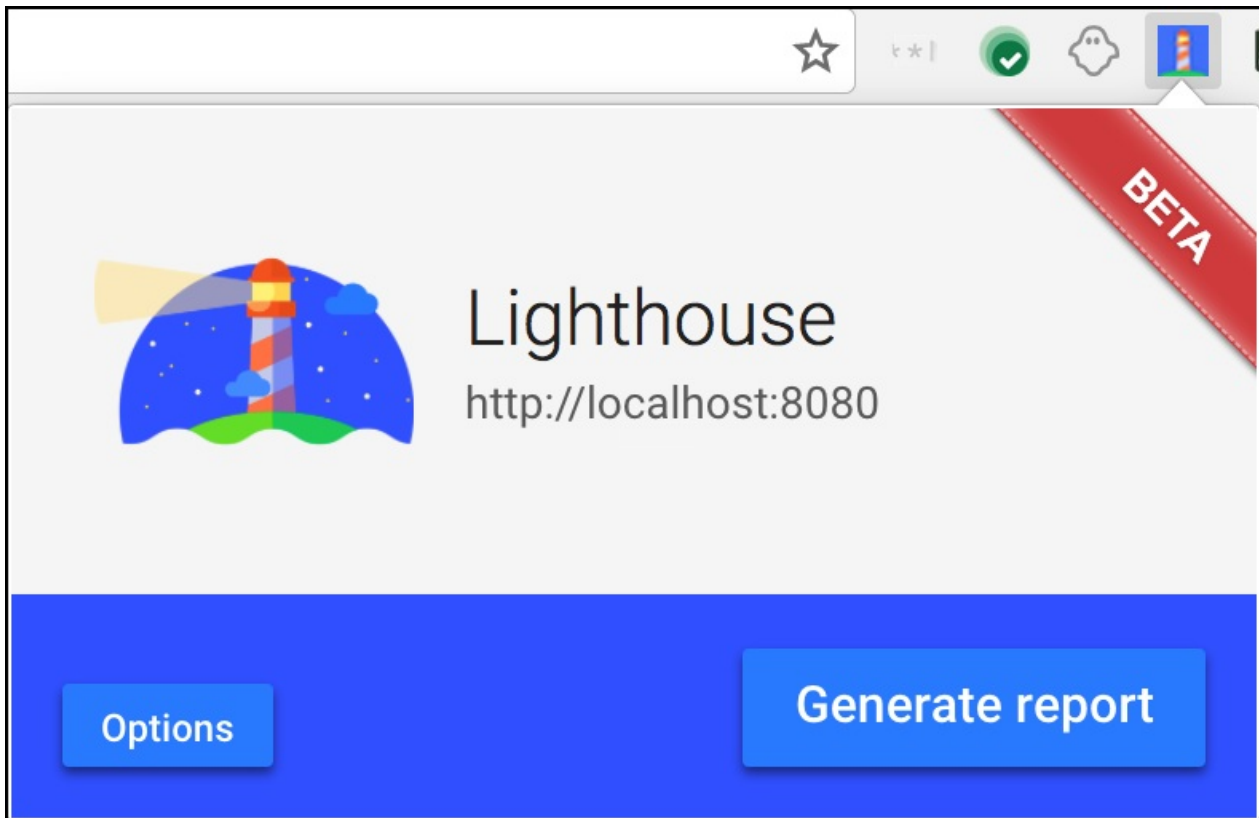
Lighthouse is available as a Chrome extension for Chrome 52 (and later) and a command line tool.

Running Lighthouse as a Chrome extension

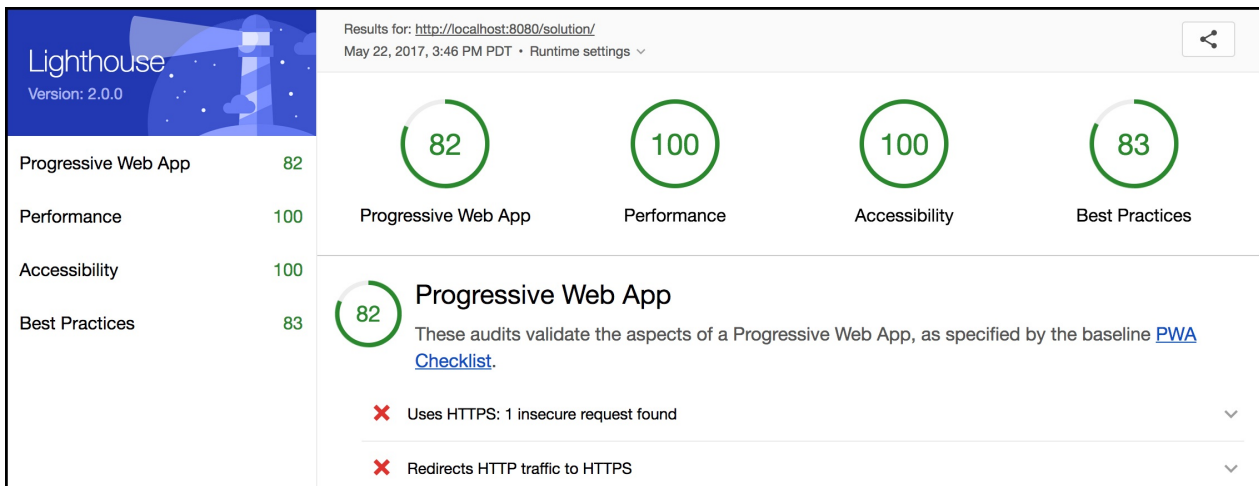
Download the Lighthouse Chrome extension from the [Chrome Web Store](#).

When installed it places an  icon in your taskbar.

Run Lighthouse on your application by selecting the icon and choosing **Generate report** (with your app open in the browser page).



Lighthouse generates an HTML page with the results. An example page is shown below.



Note: You can test it out on an example PWA, airhorner.com.

Running Lighthouse from the command line

If you want to run Lighthouse from the command line (for example, to integrate it with a build process) it is available as a [Node](#) module.

You can download Node from nodejs.org (select the version that best suits your environment and operating system).

Note: You need Node v6 or greater to run Lighthouse.

To install Lighthouse's Node module from the command line, use the following command:

```
npm install -g lighthouse
```

This installs the tool globally. You can then run Lighthouse from the command line (where <https://airhorner.com/> is your app):

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

You can check Lighthouse flags and options with the following command:

```
lighthouse --help
```

Introduction to Gulp

Contents

[Introduction](#)

[What is gulp?](#)

[How to set up gulp](#)

[Creating tasks](#)

[Examples](#)

[More automation](#)

[Review](#)

[Further reading](#)

Codelab: [Gulp Setup](#)

Introduction

Modern web development has many repetitive tasks like running a local server, minifying code, optimizing images, preprocessing CSS and more. This text discusses [gulp](#), a build tool for automating these tasks.

What is gulp?

[Gulp](#) is a cross-platform, streaming task runner that lets developers automate many development tasks. At a high level, gulp reads files as streams and pipes the streams to different tasks. These tasks are code-based and use plugins. The tasks modify the files, building source files into production files. To get an idea of what gulp can do check the [list of gulp recipes](#) on GitHub.

How to set up gulp

Setting up gulp for the first time requires a few steps.

Node

Gulp requires [Node](#), and its package manager, [npm](#), which installs the gulp plugins.

If you don't already have Node and npm installed, you can install them with [Node Version Manager](#) (nvm). This tool lets developers install multiple versions of Node, and easily switch between them.

Note: If you have issues with a specific version of Node, you can [switch to another version](#) with a single command.

Nvm can then be used to install Node by running the following in the command line:

```
nvm install node
```

This also installs Node's package manager, [npm](#). You can check that these are both installed by running the following commands from the command line:

```
node -v  
  
npm -v
```

If both commands return a version number, then the installations were successful.

Gulp command line tool

Gulp's command line tool should also be installed globally so that gulp can be executed from the command line. Do this by running the following from the command line:

```
npm install --global gulp-cli
```

Creating a new project

Before installing gulp plugins, your application needs to be initialized. Do this by running the following command line command from within your project's working directory:

```
npm init
```

This command begins the generation of a **package.json** file, prompting you with questions about your application. For simplicity these can all be left blank (either by skipping the prompts with the return key or by using `npm init -y` instead of the above command), but in

production you could store [application metadata](#) here. The file looks like this (your values may be different):

package.json

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Don't worry if you don't understand what all of these values represent, they are not critical to learning gulp.

This file is used to track your project's packages. Tracking packages like this allows for quick reinstallation of all the packages and their dependencies in future builds (the `npm install` command will read **package.json** and automatically install everything listed).

Note: It is a best practice not to push packages to version control systems. It's better to use `npm install` and `package.json` to install project packages locally.

Installing packages

Gulp and Node rely on plugins (packages) for the majority of their functionality. Node plugins can be installed with the following command line command:

```
npm install pluginName --save-dev
```

This command uses the npm tool to install the `pluginName` plugin. Plugins and their dependencies are installed in a **node_modules** directory inside the project's working directory.

The `--save-dev` flag updates **package.json** with the new package.

The first plugin that you want to install is gulp itself. Do this by running the following command from the command line from within your project's working directory:

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

Gulp and its dependencies are then present in the **node_modules** directory (inside the project directory). The **package.json** file is also updated to the following (your values may vary):

package.json

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "gulp": "^3.9.1"
  }
}
```

Note that there is now a `devDependencies` field with gulp and its current version listed.

Gulpfile

Once packages are installed (in **node_modules**), you are ready to use them. All gulp code is written in a **gulpfile.js** file. To use a package, start by including it in **gulpfile.js**. The following code in your gulpfile includes the gulp package that was installed in the previous section:

gulpfile.js

```
var gulp = require('gulp');
```

Creating tasks

Gulp tasks are defined in the **gulpfile.js** file using `gulp.task`. A simple task looks like this:

gulpfile.js


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

This code defines a `hello` task that can be executed by running the following from the command line:

```
gulp hello
```

A common pattern for gulp tasks is the following:

1. Read some source files using `gulp.src`
2. Process these files with one or more functions using Node's `pipe` functionality
3. Write the modified files to a destination directory (creating the directory if doesn't exist) with `gulp.dest`

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

A complete gulpfile might look like this:

gulpfile.js

```
// Include plugins  
var gulp = require('gulp'); // Required  
var pluginA = require('pluginA');  
var pluginB = require('pluginB');  
var pluginC = require('pluginC');  
  
// Define tasks  
gulp.task('task-A', function() {  
  gulp.src('some-source-files')  
    .pipe(pluginA())  
    .pipe(gulp.dest('some-destination'));  
});  
  
gulp.task('task-BC', function() {  
  gulp.src('other-source-files')  
    .pipe(pluginB())  
    .pipe(pluginC())  
    .pipe(gulp.dest('some-other-destination'));  
});
```

Where each installed plugin is included with `require()` and tasks are then defined using functions from the installed plugins. Note that functionality from multiple plugins can exist in a single task.

Examples

Let's look at some examples.

Uglify JavaScript

Uglifying (or minifying) JavaScript is a common developer chore. The following steps set up a gulp task to do this for you (assuming Node, npm, and the gulp command line tool are installed):

1. Create a new project & **package.json** by running the following in the command line (from the project's working directory):

```
npm init
```

2. Install the gulp package by running the following in the command line:

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

3. Install the [gulp-uglify](#) package by running the following in the command line:

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

4. Create a **gulpfile.js** and add the following code to include gulp and gulp-uglify:

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
```

5. Define the `uglify` task by adding the following code to **gulpfile.js**:

```
gulp.task('uglify', function() {
  gulp.src('js/**/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('build'));
});
```

6. Run the task from the command line with the following:

```
gulp uglify
```

The task reads all JavaScript files in the **js** directory (relative to the **gulpfile.js** file), executes the `uglify` function on them (uglifying/minifying the code), and then puts them in a **build** directory (creating it if it doesn't exist).

Prefix CSS & build sourcemaps

Multiple plugins can be used in a single task. The following steps set up a gulp task to prefix CSS files and create [sourcemaps](#) for them (assuming Node, npm, and the gulp command line tool are installed):

1. As in the previous example, create a new project and install gulp, [gulp-autoprefixer](#), and [gulp-sourcemaps](#) by running the following in the command line (from the project's working directory):

```
npm init
npm install gulp --save-dev
npm install gulp-autoprefixer --save-dev
npm install gulp-sourcemaps --save-dev
```

2. Include the installed plugins by adding the following code to a **gulpfile.js** file:

```
var gulp = require('gulp');
var autoprefixer = require('gulp-autoprefixer');
var sourcemaps = require('gulp-sourcemaps');
```

3. Create a task that prefixes CSS files, creates sourcemaps on the files, and writes the new files to the **build** directory by adding the following code to **gulpfile.js**:

```
gulp.task('processCSS', function() {
  gulp.src('styles/**/*.css')
    .pipe(sourcemaps.init())
    .pipe(autoprefixer())
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build'));
});
```

This task uses two plugins in the same task.

More automation

Default tasks

Usually, developers want to run multiple tasks each time an application is updated rather than running each task individually. Default tasks are helpful for this, executing anytime the `grunt` command is run from the command line.

Let's add the following code to **gulpfile.js** to set `task1` and `task2` as default tasks:

gulpfile.js

```
gulp.task('default', ['task1', 'task2']);
```

Running `gulp` in the command line executes both `task1` and `task2`.

Gulp.watch

Even with default tasks, running tasks each time a file is updated during development can become tedious. `gulp.watch` watches files and automatically runs tasks when the corresponding files change. For example, the following code in **gulpfile.js** watches CSS files and executes the `processCSS` task any time the files are updated:

gulpfile.js

```
gulp.task('watch', function() {  
  gulp.watch('styles/**/*.css', ['processCSS']);  
});
```

Running the following in the command line starts the watch:

```
gulp watch
```

Note: The watch task continues to execute once initiated. To stop the task, use **Ctrl + C** in the command line or close the command line window.

Review

Using a build tool for the first time can be daunting with multiple tools to install and new files to create. Let's review what we've covered and how it all fits together.

Because gulp and its plugins are node packages, gulp requires [Node](#) and its package manager, [npm](#). They are global tools so you only need to install them once on your machine, not each time you create a project. In this text, we used [Node Version Manager](#) (nvm) to install Node and npm, but they could also have been installed directly.

Gulp runs from the command line, so it requires a command line tool to be installed. Like Node, it's a global tool, and only needs to be installed on your machine (not per project).

When you want to use gulp in a project, you start by initializing the project with `npm init`. This creates a file called **package.json**. The **package.json** file tracks the Node packages that are installed for that project. Each time a new package is installed, such as the gulp-uglify plugin, **package.json** is updated with the `--save-dev` flag. If the project is stored in version control or transferred without including all of the packages (a best practice), the packages can be quickly re-installed with `npm install`. This reads **package.json** and installs all required packages.

Once plugins are installed, they need to be included in the **gulpfile.js** file. This file is where all gulp code belongs. This file is also where gulp tasks are defined. Gulp tasks use JavaScript code and the imported functions from plugins to perform various tasks on files.

With everything installed and tasks defined, gulp tasks can be run by executing command line commands (such as `gulp uglify`).

Further reading

- [Gulp's Getting Started guide](#)
- [List of gulp Recipes](#)
- [Gulp Plugin Registry](#)

Introduction to Push Notifications

Contents

[What Are Push Notifications?](#)

[Push Notification Terms](#)

[Understanding Push Notifications on the Web](#)

[Notifications API](#)

[Designing with the Future in Mind](#)

[Push API](#)

[Best Practices](#)

[More Resources](#)

Codelab: [Integrating Web Push](#)

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 re-engage users with customized content.

Push Notifications are assembled using two APIs: the [Notifications API](#) and the [Push API](#). The Notifications API lets the app 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.

Note: 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
- **Notifications API** – an interface used to configure and display notifications to the user
- **Push API** – an interface used to subscribe your app to a push service and receive push messages in the service worker
- **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

Push notifications let your app extend beyond the browser, and are an incredibly powerful way to engage with the user. They can do simple things, such as alert the user to an important event, display an icon and a small piece of text that the user can then click 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.

There are several pieces that come together to make push notifications work. Browsers that support web push each implement their own push service, which is a system for processing messages and routing them to the correct clients. Push messages destined to become notifications are sent from a server directly to the push service, and contain the information necessary for the push service to send it to the right client and wake up the correct service worker. The section on the [Push API](#) describes this process in detail.

When it receives a message, the service worker wakes up just long enough to display the notification and then goes back to sleep. Because notifications are paired with a service worker, the service worker can listen for notification interactions in the background without using resources. When the user interacts with the notification, by clicking or closing it, the service worker wakes up for a brief time to handle the interaction before going back to sleep.

Notifications API

The [Notifications API](#) lets us display notifications to the user. It is incredibly powerful and simple to use. Where possible, it uses the same mechanisms a native app would use, giving a completely native look and feel.

We can split the Notifications API into two core areas (these are non-technical and are not part of the spec). The *Invocation API* controls how to make your notification appear, including styling and vibration. We create (or invoke) the notification from the page (or from the server, in the case of push notifications). The *Interaction API* controls what happens when the user engages with the notification. User interaction is handled in the service worker.

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 (the "Invocation API"). 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.

Note: You can also create a notification using a [notification constructor](#). However, a notification created this way is not paired with a service worker and is therefore not interactive.

Add notification options

The `showNotification` method has an optional second argument for configuring the notification. The following example code demonstrates some of the available options. See the [showNotification reference on MDN](#) for a complete explanation of each option.

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. It 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's a message from their friend you might include an image of the sender'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 clicked. This is a massive step forward for the web, but it's still a 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 (using the "Interaction API"). Once the user has seen your notification they can either *dismiss* it or *act on* it.

The notificationclose event

If the user dismisses the notification through a direct action on the notification (such as a swipe in Android), it raises a `notificationclose` event 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 tells you how the user is interacting with your notifications. You might, for example, log the event to your analytics database. Or, you might use the event to synchronize your database and avoid re-notifying 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 is to 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.example.com');  
    notification.close();  
  }  
});
```

We can determine what action button the user clicked by inspecting the `action` property on the event object.

When a user clicks on a notification they usually expect to be taken directly to where they can get more information about the notification. You can open a new window by calling `clients.openWindow` 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. Handling actions in this way provides a default experience that works everywhere.

Designing with the future in mind

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 Notifications API. Note that:

- Not all browsers implement the Notifications API to the same level
- Operating systems may not 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 consistent 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 the user's 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 Notifications API:

- 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 Notifications 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, some OSs may display a limited number of actions and others may not make actions directly visible to the user.

You can check the maximum number of action buttons that can be displayed by calling `Notification.maxActions`. Do this when you create notifications so you can adapt them if needed. You can also check this in the `notificationclick` handler in the service worker to determine the right response.

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 execute the default response. You can then 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 may 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), be sure to display that information in the title or the body so the user can determine the importance of the notification if the image is not visible.

Button labels should be clear and concise. Although action buttons can have images, not every system can display them.

Also, don't rely on vibrations to notify the user. Many systems can't vibrate, or won't vibrate if the user has their device volume muted.

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. We can now do the same on the web through the Push API.

Note: 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 lets developers engage users by providing timely and customized content outside the context of the web page. It is one of the most critical APIs 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.

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 (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 an overview of how 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 can then subscribe the app to the browser's 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 (see the example below). You send your push messages to this URL, encrypted with the public key, and the push service sends it to the right client. A typical subscription object looks like this:

```
{
  "endpoint": "https://fcm.googleapis.com/fcm/send/dpH5lCsTSSM:APA91bHqjZxM0VImWwqDRN7U0a3AycjUf40-byuxb_wJsKRakVv_iKw56s16ekq6FUqoCF7k2nICUpd8fHPxVTgqLunFeVeB91LCQZyohyAztTH8ZQL9WCxKpA6dvTG_TUIhQUFq_n",
  "keys": {
    "p256dh": "BLQELIDm-6b9B107YrEuXJ4BL_YBVQ0dvt9NQGGJxIQidJWHPNa9YrouvcQ9d7_MqzvGS9A1z60SZNCG3qfpk=",
    "auth": "4vQK-SvRAN5eo-8ASlrwA=="
  }
}
```

How does the push service know which client to send the message to? The endpoint URL contains a unique identifier. This identifier is used to route the message that you send to the correct device, and when processed by the browser, identifies which service worker should handle the request.

The identifier is opaque. As a developer, you can't determine any personal data from it. Also, it is not stable, so it can't be used to track users.

Because push notifications are paired with a service worker, apps that use push notifications must be on HTTPS. This ensures that the communication channel between your server and the push service is secure, and from the push service to the user is also secure.

However, HTTPS doesn't ensure that the push service 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.

The following summarizes the process of sending and receiving a push message and then displaying a push notification.

On the client:

1. Subscribe to the push service
2. Send the subscription object to the server

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, which finds the correct service worker and invokes a "push" event. Now, on the client:

1. Receive the message data (if there is any) in the "push" event
2. Perform some custom logic in the push event
3. Show a notification

That completes the path from server push to user notification. Let's look at each part. We'll start with receiving the message in the service worker, since that's the simplest, and then move on to subscribing to the push service and sending the push message from the server.

Handling the push event in the service worker

Let's see how the service worker handles push messages. The service worker both receives the push message and creates the notification.

When a [browser that supports push messages](#) receives a 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. In general, we use the `waitUntil` method to ensure the service worker doesn't terminate before an asynchronous operation has completed.

Subscribing to Push Notifications

Before we can send a push message we must first subscribe to a push service. Subscribing returns a subscription object, or `subscription`. The `subscription` is a critical piece of the process to send push messages. It tells us, the developer, to which push service we should send our push messages (remember, each browser will provide their own push service). The subscription also details which client the push service should route the messages to. Finally, the `subscription` contains the public key to encrypt the data so that it is delivered securely to the user.

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 will log results to the 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.register('sw.js').then(function(reg) {
    console.log('Service Worker Registered!', reg);

    reg.pushManager.getSubscription().then(function(sub) {
      if (sub === null) {
        // 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);
      }
    });
  })
  .catch(function(err) {
    console.log('Service Worker registration failed: ', err);
  });
}
```

We should perform this check whenever the user accesses our app because `subscription` objects may change during their lifetime. We need to make sure that it is synchronized with our server. If there is no `subscription` object we can 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

```
function subscribeUser() {
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.ready.then(function(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);
        }
      });
    })
  }
}
```

It's best practice to call the `subscribeUser()` function in response to a user action signalling they would like to subscribe to push messages from our app.

In the above example 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 ensures that every incoming message has a matching (and visible) notification.

Note: 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. That is, we can't do it silently without the user knowing. If we don't display a notification the browser automatically creates one to let the user know that the app 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 then 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 to create your push message, encrypt it, and send it to a Push messaging platform. The protocol abstracts the details of which messaging platform and browser the user has.

The Web Push protocol is complex, but we don't need to understand all of 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 currently uses [Firebase Cloud Messaging](#) (FCM) as its push service. [FCM recently adopted the Web Push protocol](#). FCM is the successor to Google Cloud Messaging (GCM) and supports the same functionality and more.

To use Firebase Cloud Messaging, you need to set up a project on [Firebase](#) (see the [section on VAPID](#) to get around this step). Here's how:

1. In the [Firebase console](#), select **Create New Project**.
2. Supply a project name and click **Create Project**.

3. Click the Settings icon next to your project name in the Navigation panel and select **Project Settings**.
4. Open 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 your app's `manifest.json` file. For example, the manifest could look like this:

```
{
  "name": "Push Notifications app",
  "gcm_sender_id": "370072803732"
}
```

Note: The `gcm_sender_id` is required for Chrome prior to version 52, Opera Android, and Samsung Internet.

To get FCM to push a notification without a payload to your web client, the request must include the following:

- The subscription endpoint URL
- The public Server key. FCM uses this to check whether the server making the requests is actually allowed to send messages to the receiving user.

A production site or app normally sets up a service to interact with FCM from your server. Check out the [Web Fundamentals documentation](#) for more information.

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 "ENDPOINT_URL" --request POST --header "TTL: 60" --header "Content-Length: 0" \
--header "Authorization: key=SERVER_KEY"
```

For example:

```
curl "https://android.googleapis.com/gcm/send/fYFVeJQJ2CY:APA91bGrFGRmy-sY6NaF8a...gls7HZcwJL4 \
LFxjg0y0-ksEhKjpeFC5P" --request POST --header "TTL: 60" --header "Content-Length: 0" \
--header "Authorization: key=AIZA91bGrFGRmy-sY6NaF8a...gls7HZcwJL4"
```

You can send a message to Firefox using the same cURL command, but without the

Authorization header:

```
curl "ENDPOINT_URL" --request POST --header "TTL: 60" --header "Content-Length: 0" --header
```

For example:

```
curl "https://updates.push.services.mozilla.com/wpush/v1/gAAAAABYGm18oAFQC2a-HYb...7hKVui9zuT" \
--request POST --header "TTL: 60" --header "Content-Length: 0"
```

Working with Data Payloads

It's 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 using 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) {
  var body;

  if (e.data) {
    body = e.data.text();
  } else {
    body = 'Push message no payload';
  }

  var options = {
    body: body,
    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('Push Notification', 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 BLOB, a typed array, or raw text.

Sending the message from the Server

In this section, we cover how to send a push message from the server.

In order to send data, the push message must be encrypted with the key information from the subscription object. As with anything related to encryption, it's usually easier to use an actively maintained 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(pushSubscription, payload, options)
```

The first argument is the `subscription` object. The second argument is the `payload`. The third is an `options` object that contains various options to configure the message. See [the documentation](#) for details.

While we 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 pushSubscription = {"endpoint":"https://android.googleapis.com/gcm/send/f1LsxxKphf
Q:APA91bFUx7ja4BK4JVrNgVjpg1cs9lGSGI6IMNL4mQ3Xe6mDGxvt_C_gItKYJI9CAx5i_Ss6cmDxdwZoLyhS
2RJhkcv7LeE6hki0sK6oBzbyifvKCdUYU7ADIRBiYNxIVpLIYeZ8kq_A",
"keys":{"p256dh":"BLC4xRzKlKORKWlbdgFaBrrPK3ydWAHo4M0gs0i1oEKgPpWC5cw80CzVrOQRv-1npXRW
k8udnW3oYhIO4475rds=", "auth":"5I2Bu2oKdyy9CwL8QVF0NQ=="}};

var payload = 'Here is a payload!';

var options = {
  gcmAPIKey: 'AIzaSyD1JcZ8WM1vTtH6Y0tXq_Pnuw4jgj_92yg',
  TTL: 60
};

webPush.sendNotification(
  pushSubscription,
  payload,
  options
);
```

This example passes the subscription object, payload, and server key into the `sendNotification` method. It also passes in a time-to-live, which is the value in seconds that describes how long a push message is retained by the push service (by default, four weeks).

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:

- An unauthenticated push service is exposed to a greater risk of denial of service attack
- Any application server in possession of the endpoint is able to send messages to your users
- There's no way for the push service to contact the developer if there are problems

The solution is to have the publisher optionally identify themselves using the [Voluntary Application Server Identification for Web Push \(VAPID\) protocol](#). At a minimum, this provides a stable identity for the application server, though this could also include contact information, such as an email address.

The spec lists several benefits of using VAPID:

- A consistent identity can be used by a push service to establish behavioral expectations for an application server. Significant deviations from an established norm can then be used to trigger exception handling procedures.
- Voluntarily-provided contact information can be used to contact an application server operator in the case of exceptional situations.
- Experience with push service deployment has shown that software errors or unusual circumstances can cause large increases in push message volume. Contacting the operator of the application server has proven to be valuable.
- Even in the absence of usable contact information, an application server that has a well-established reputation might be given preference over an unidentified application server when choosing whether to discard a push message.

Using VAPID also lets you avoid the FCM-specific steps for sending a push message. You no longer need a Firebase project, a `gcm_sender_id`, or an `Authorization` header.

Using VAPID

The process is pretty simple:

1. Your application server creates a public/private key pair. The public key is given to your web app.
2. When the user elects to receive pushes, add the public key to the `subscribe()` call's options object.
3. When your app server sends a push message, include a signed JSON web token along with the public key.

Let's look at these steps in detail.

Note: We recommend using a [library](#) to implement VAPID in your push messages. This spares you from the details of encryption and JWT signing. We show an [example](#) using the [web-push library](#) for Node.js at the end of this section.

Create a public/private key pair

Here's the relevant section from the spec regarding the format of the VAPID public/private keys:

Application servers SHOULD generate and maintain a signing key pair usable with elliptic curve digital signature (ECDSA) over the P-256 curve.

You can see how to do this in the [web-push node library](#):

```
function generateVAPIDKeys() {
  const vapidKeys = webpush.generateVAPIDKeys();

  return {
    publicKey: vapidKeys.publicKey,
    privateKey: vapidKeys.privateKey,
  };
}
```

Subscribing with the public key

To subscribe a Chrome user for push with the VAPID public key, pass the public key as a `Uint8Array` using the `applicationServerKey` parameter of the `subscribe()` method.

```
const publicKey = new Uint8Array([0x4, 0x37, 0x77, 0xfe, .... ]);
serviceWorkerRegistration.pushManager.subscribe(
  {
    userVisibleOnly: true,
    applicationServerKey: publicKey
  }
);
```

You'll know if it has worked by examining the endpoint in the resulting subscription object; if the origin is **fcm.googleapis.com**, it's working.

Note: Even though this is an FCM URL, use the [Web Push Protocol](#) **not** the FCM protocol, this way your server-side code will work for any push service.

Sending a push message

To send a message using VAPID, you make a normal Web Push Protocol request with two additional HTTP headers: an `Authorization` header and a `Crypto-Key` header. Let's look at these new headers in detail.

Note: This is where web push [libraries](#) really shine, as the process of signing and sending a message can be quite complex. We include an [example](#) of sending a message with VAPID using the [web-push library](#) for Node.js at the end of this section.

Authorization header

The `Authorization` header is a signed [JSON Web Token \(JWT\)](#) with "WebPush " in front of it.

A JWT is a way of sharing a JSON object with a second party in such a way that the sending party can sign it and the receiving party can verify the signature is from the expected sender. The structure of a JWT is three encrypted strings, joined with a single dot between them.

```
<JWTHeader>.<Payload>.<Signature>
```

JWT header

The JWT Header contains the algorithm name used for signing and the type of token. For VAPID this must be:

```
{
  "typ": "JWT",
  "alg": "ES256"
}
```

This is then base64 url encoded and forms the first part of the JWT.

Payload

The Payload is another JSON object containing the following:

- Audience (`aud`)
- This is the origin of the push service (NOT the origin of your site). In JavaScript, you could do the following to get the audience: `const audience = new URL(subscription.endpoint).origin`
- Expiration Time (`exp`)
- This is the number of seconds until the request should be regarded as expired. This MUST be within 24 hours of the request being made, in UTC.
- Subject (`sub`)
- The subject needs to be a URL or a mailto: URL. This provides a point of contact in case the push service needs to contact the message sender.

An example payload could look like the following:

```
{
  "aud": "http://push-service.example.com",
  "exp": Math.floor((Date.now() / 1000) + (12 <em> 60 </em> 60)),
  "sub": "mailto: my-email@some-url.com"
}
```

This JSON object is base64 url encoded and forms the second part of the JWT.

Signature

The Signature is the result of joining the encoded header and payload with a dot then encrypting the result using the VAPID private key you created earlier. The result itself should be appended to the header with a dot.

There are a [number of libraries](#) that will take the header and payload JSON objects and generate this signature for you.

The signed JWT is used as the `Authorization` header, with "WebPush" prepended to it, and looks something like the following:

```
WebPush eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhdWQiOiJodHRwczovL2ZjbS5nb29nbGVhcGlzLmNvbSIsImV4cCI6MTQ2NjY2ODU5NCwic3ViIjoibWpHRvOnNpbXBsZS1wdXNoLWRlbW9AZ2F1bnRmYWNLmNvLnVrIn0.Ec0VR8dtf5qb8Fb5Wk91br-evfho9sZT6jBRuQwxVMFyK5S8bh0jk8kuxvilLqTBMdXJM5l3uVrV0QirSsjq0A
```

There are a few things to point out here. First, the `Authorization` header literally contains the word `WebPush` and should be followed by a space then the JWT. Also notice the dots separating the JWT header, payload, and signature.

Crypto-Key header

As well as the `Authorization` header, you must add your VAPID public key to the `Crypto-Key` header as a base64 url encoded string with `p256ecdsa=` prepended to it.

```
p256ecdsa=BDd3_hVL9fZi9Ybo2UUzA284WG5FZR30_95YeZJsiApwXKpNcF1rRPF3foIiBHXRDJI2Qhumhf6_LFTeZaNdIo
```

When you are sending a notification with encrypted data, you will already be using the `Crypto-Key` header, so to add the application server key, you just need to add a comma before adding the above content, resulting in:

```
dh=BGEw2wsHgLwzerjvnMTkbKrFRxmwJ5S_k7zi7A1coR_sVjHmGr1vzYpAT1n4NPbioFlQkIrTNL8EH4V3ZZ
4vJE,
p256ecdsa=BDd3_hVL9fZi9Ybo2UUzA284WG5FZR30_95YeZJsiApwXKpNcF1rRPF3foIiBHXRdJI2Qhumhf6_
LFTeZaN
```

Note: There is a bug in Chrome prior to version 52 that requires the use of a semicolon instead of a comma in the Crypto-key header.

Examples

Here's an example cURL request using VAPID:

```
curl "https://updates.push.services.mozilla.com/wpush/v1/gAAAAABXmk...dyR" --request
POST --header "TTL: 60" --header "Content-Length: 0" --header "Authorization: WebPush
eyJ0eXAiOiJKV1QiLCJhbGciOiJFUzI1NiJ9.eyJhdWQiOiJodHRwczovL2ZjbS5nb29nbGVhcGlzLmNvbSIsI
mV4cCI6MTQ2NjY2ODU5NCwic3ViIjoibWFpbHRvOnNpbXBsZS1wdXNoLWRLbW9AZ2F1bnRmYWNIbmNvLnVrIn0
.Ec0VR8dtf5qb8Fb5Wk91br-evfho9sZT6jBRuQwxVMFyK5S8bh0jk8kuxvillqTBmDXJM5l3uVrV0QirSsjq0
A" --header "Crypto-Key: p256ecdsa=BDd3_hVL9fZi9Ybo2UUzA284WG5FZR30_95YeZJsiApwXKpNcF1
rRPF3foIiBHXRdJI2Qhumhf6_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.

Here is an example of sending a payload with VAPID in a node script using the web-push library:

```
var webPush = require('web-push');

var pushSubscription = {"endpoint":"https://fcm.googleapis.com/fcm/send/c0NI73v1E0Y:APA91bEN7z2weTCpJmCS-MFyfbgjtmlAWuV5YaaNw625_Rq2-f0ZrVLdRPXKGm7B3uwfygic0CeEoWQxCKIx1L3RWG2xkHs6C8-H_cxq-4Z-isAiZ3ix084-2HeXB9eUvkfNO_t1jd5s", "keys":{"p256dh":"BHxSHtYS0q3i0Tb3Ni6chC132ZDPd5uI4r-exy1KsevRqHJv0M5hNX-M83zgYjp-1kdirHv0E1hjw6Hivw1Be5M=", "auth":"4a3vf9MjR9CtPSHLHcsLzQ=="}};

var vapidPublicKey = 'BADXhdGDgXJeJadxabiFhm1TyF17HrCsfiYj3XEhg1j-RmT2wXU3lHiBqPSKSotvtfejZlAaPywJ9E-7AxXQBj4';
var vapidPrivateKey = 'VCgMIYe2BnuNA4iCfR94hA6pLPT3u3ES1n1x0TrmyLw';

var payload = 'Here is a payload!';

var options = {
  vapidDetails: {
    subject: 'mailto:example_email@example.com',
    publicKey: vapidPublicKey,
    privateKey: vapidPrivateKey
  },
  TTL: 60
};

webPush.sendNotification(
  pushSubscription,
  payload,
  options
);
```

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. The library takes care of encrypting the message, generating and signing the JWT, and adding the `Authorization` and `Crypto-Key` headers to the request. See the [web-push documentation](#) for more information on how to use the library.

Best Practices

While it's relatively simple to get notifications up and running, making an experience that users really value is trickier. There are also many edge cases to consider when building an experience that works well.

This lesson discusses best practices for implementing push notifications.

Using Notifications Wisely

Notifications should be timely, precise, and relevant. By following these three rules, you'll keep your users happier 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 is 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 only 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 notifications that are not directed specifically at the user, 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 of any kind.

Design Notifications According to Best Principles

This section provides best practices to make your notifications timely, precise, and relevant.

To show notifications we need to prompt the user to give permission. But when is the best time to do that?

Geolocation offers a good example of where we can look at people's experience with its prompts. Although geolocation is a great API, 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. 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.

The following interaction patterns 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, if the user 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 offers the user.

Managing the Number of Notifications

It's 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 may 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 it navigates that client to the correct URL and focuses the window. If a window that suits our needs is not found, it opens a new window.

Managing Notifications at the Server

So far, we've been assuming the user is around to see our 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, corresponding 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 message is not sent within the TTL, it is not delivered.

Another advantage of specifying the lifespan of a message is that FCM never throttles messages with a `time_to_live` value of 0 seconds. In other words, FCM 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 to display redundant notifications that have been removed elsewhere, but you currently 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

- [Adding Push Notifications to a Web App](#)
- [Adding Push Notifications to a Web App \(Codelab\)](#)
- [Using the Push API](#)

When to use push notifications

- [Web Push Notifications: Timely, Relevant, and Precise](#)

Demos

- [Push demo](#)
- [Notification Generator](#)

Messaging concepts and options

- [Setting the lifespan of a message](#)

Web-push documentation

- [web-push - Web Push library for Node.js](#)

Web Push Libraries

- [web-push-lib](#)

VAPID

- [Voluntary Application Server Identification for Web Push](#)
- [Using VAPID with WebPush - Mozilla Cloud Services](#)
- [Sending VAPID identified WebPush Notifications via Mozilla's Push Service - Mozilla Cloud Services](#)
- [Easy VAPID generation](#)
- [VAPID verification](#)

Encryption

- [Web Push Payload Encryption](#)
- [Push Encryption Verifier](#)
- [Message Encryption for Web Push](#)

JWT Signing

- [Libraries for Token Signing/Verification](#)

Firebase Cloud Messaging

- [Firebase Cloud Messaging](#)
- [Set Up a JavaScript Firebase Cloud Messaging Client App](#)

Integrating Analytics

Contents

[What is Google Analytics?](#) [Creating an account](#) [Add analytics to your site](#) [Google Analytics dashboard](#) [Custom events](#) [Analytics and service worker](#) [Offline analytics](#) [Further reading](#)

Codelab: [Integrating Analytics](#)

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, and so forth. 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 (dashboard) and through the [reporting API](#).

Why use it?

Using analytics tools gives developers valuable information about their application such as:

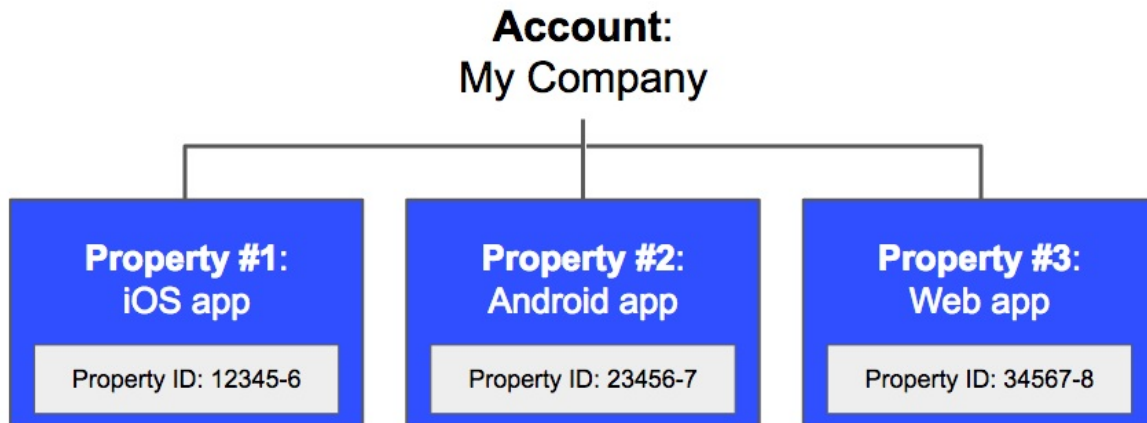
- User's geographic location, user agent, screen resolution, and language
- How long users spend on pages, how often they visit pages, and the order in which pages are viewed
- What times users are visiting the site and from where they arrived at the site

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 IDs (also called property IDs) 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.



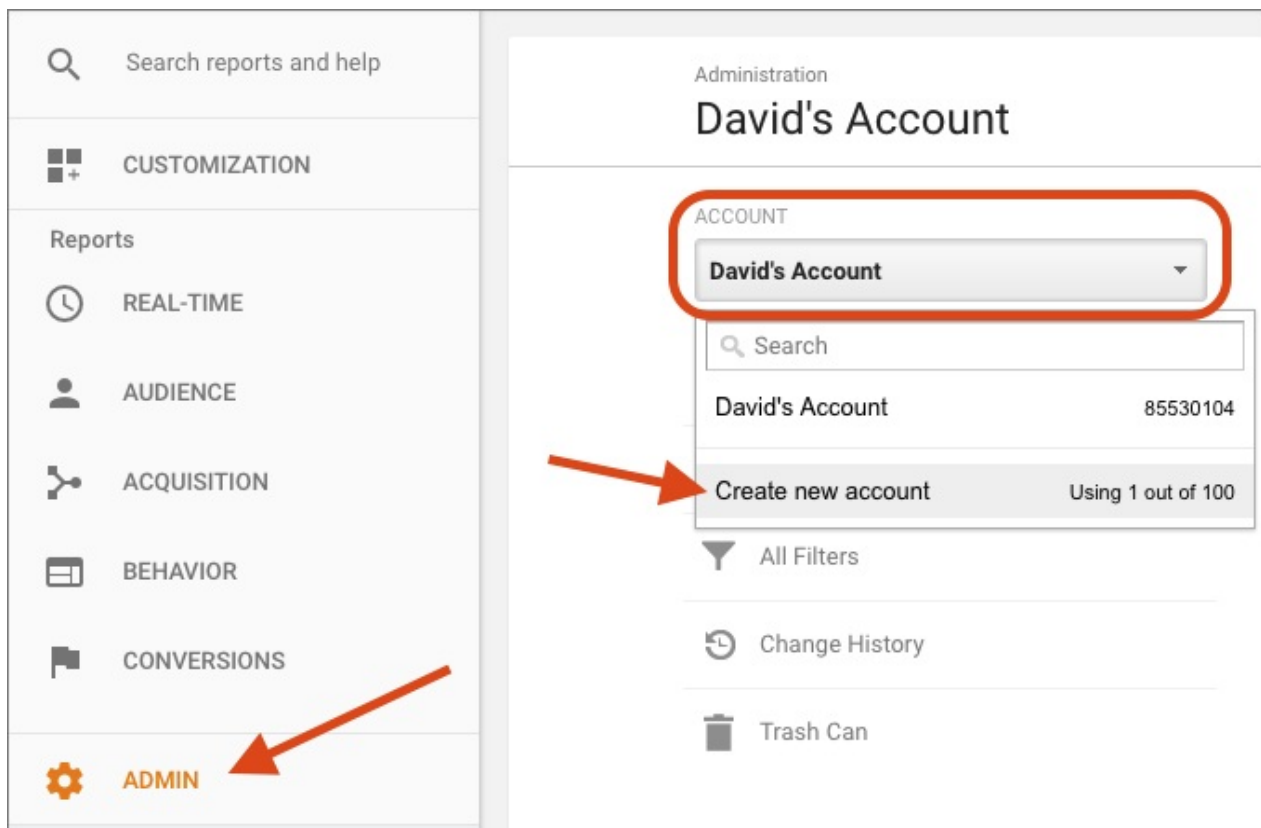
If you only have one app, the simplest scenario is to create a single Google Analytics account, and add a single property to that account. That property can represent your app.

A Google Analytics account can be created from analytics.google.com.

Note: The Google Analytics UI is subject to updates and may not look exactly like the screenshots presented here.

If you already have a Google Analytics account

Create another one. Select the **Admin** tab. Under **account**, select your current Google Analytics account and choose **create new account**. A single Gmail account can have multiple (currently 100) Google Analytics accounts.



If you don't have a Google Analytics account

Select **Sign up** to begin creating your account. The account creation screen should look 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.

My New Account Name

Setting up your property

Website Name

My New Website

Website URL

http://

Example: http://www.mywebsite.com

Industry Category

Select One

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.

Note: Analytics will still work even if this URL does not match your site. The only thing that

ties analytics data to your account is the value of your **tracking / property ID**. The site URL that you use to create your Google Analytics account is only used for things like automated testing.

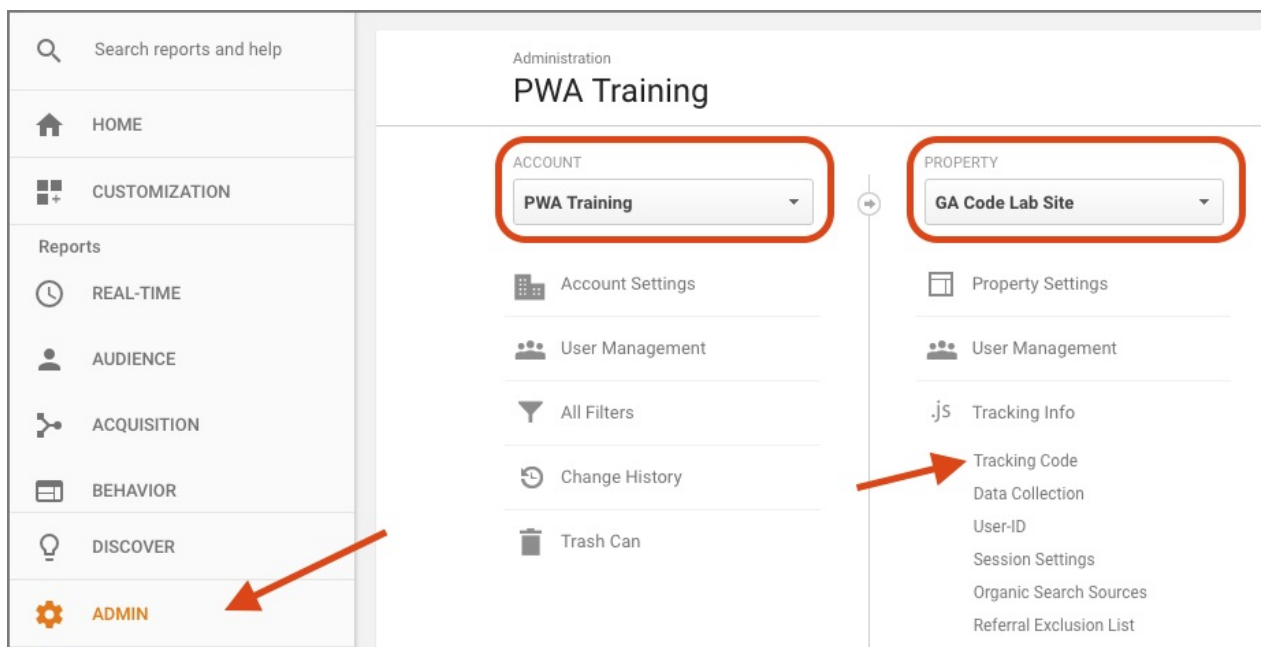
You can set an industry category to get benchmarking information later (in other words, to compare your app with other apps in the same industry). You can set your timezone here as well. You may also see data sharing options, but these are not required.

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 dropdown list.
3. Then under **property**, select your property (for example "GA Code Lab Site") from the dropdown list.
4. Now choose **Tracking Info**, and then **Tracking Code**.



Your tracking ID looks like `UA-XXXXXXXX-Y` and your tracking code snippet looks like:

index.html

```
<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,'script', \
'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 ([Immediately Invoked Function Expression](#)) in the script does two things:

- Creates another `script` tag that starts asynchronously downloading **analytics.js**, the library that does all of the analytics work.
- Initializes a global `ga` function, called the command queue.

The `ga` command queue is the main interface for using **analytics.js**. The command queue stores commands (in order) until **analytics.js** has loaded. Once **analytics.js** has loaded, the commands are executed sequentially. 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.

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

You can learn more about [creating trackers](#) in the documentation.

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. You can learn more about [sending data](#) in the Google Analytics documentation.

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
- The time spent on each page and the order in which the pages are visited
- Which internal links are clicked (based on the URL of the next pageview)

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](#).

Debugging and development

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 break down 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).

Note: The debug version should not be used in production as it is a much larger file.

Note: You can also use the [Chrome debugger extension](#).

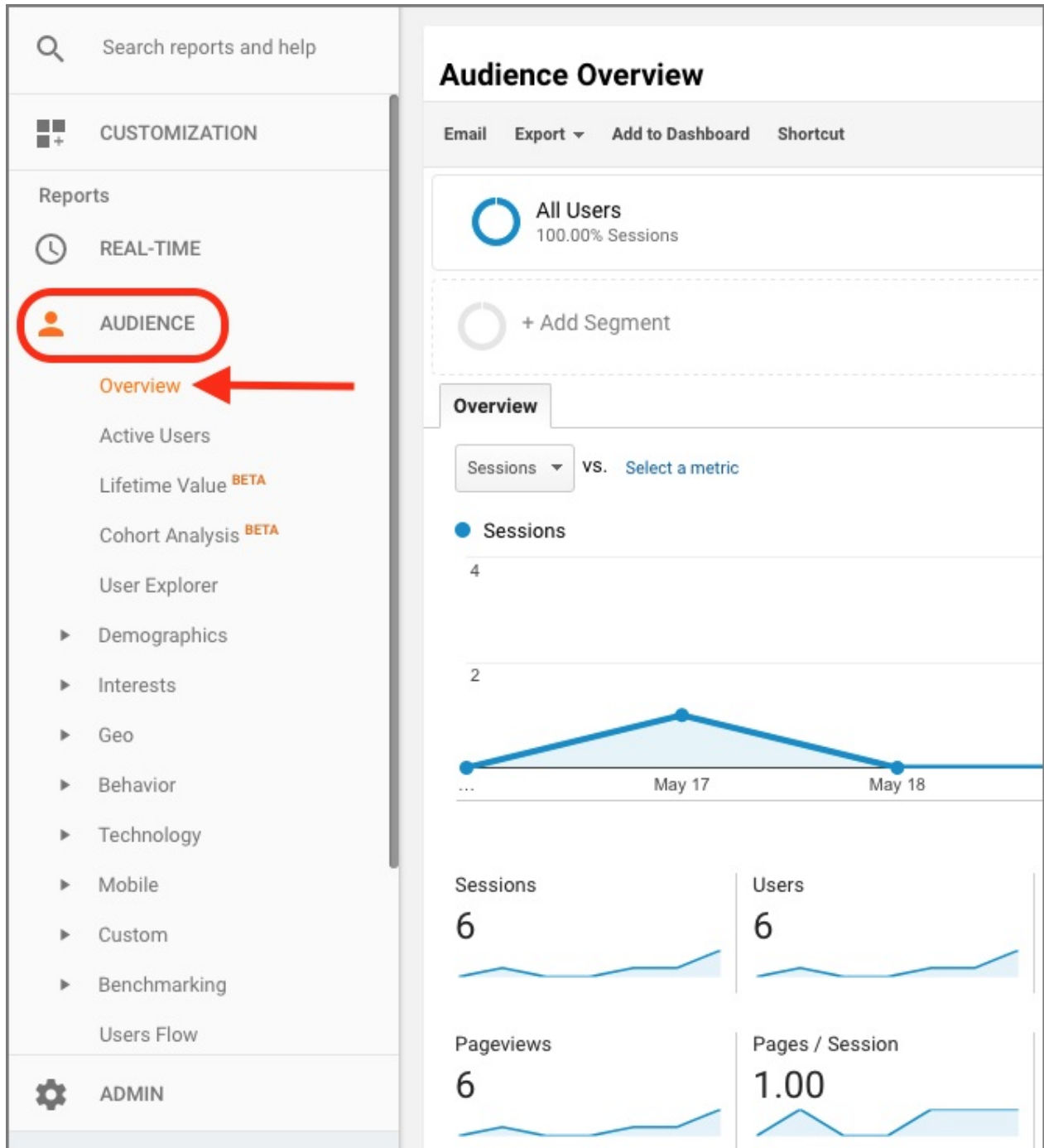
For more information

- [Adding analytics.js to Your Site](#)
- [Google Reporting API v4](#)
- [Google Analytics Debugger](#)
- [Google Analytics Debugging](#)
- [Getting and Setting Tracker Data](#)

Google Analytics dashboard

All of the data that is sent to Google Analytics can be viewed in the Google Analytics dashboard (the Google Analytics web interface). For example, overview data is available by selecting **Audience** and then **Overview** (shown below).

From the overview page you can see general information such as pageview records, bounce rate, ratio of new and returning visitor, and other statistics.

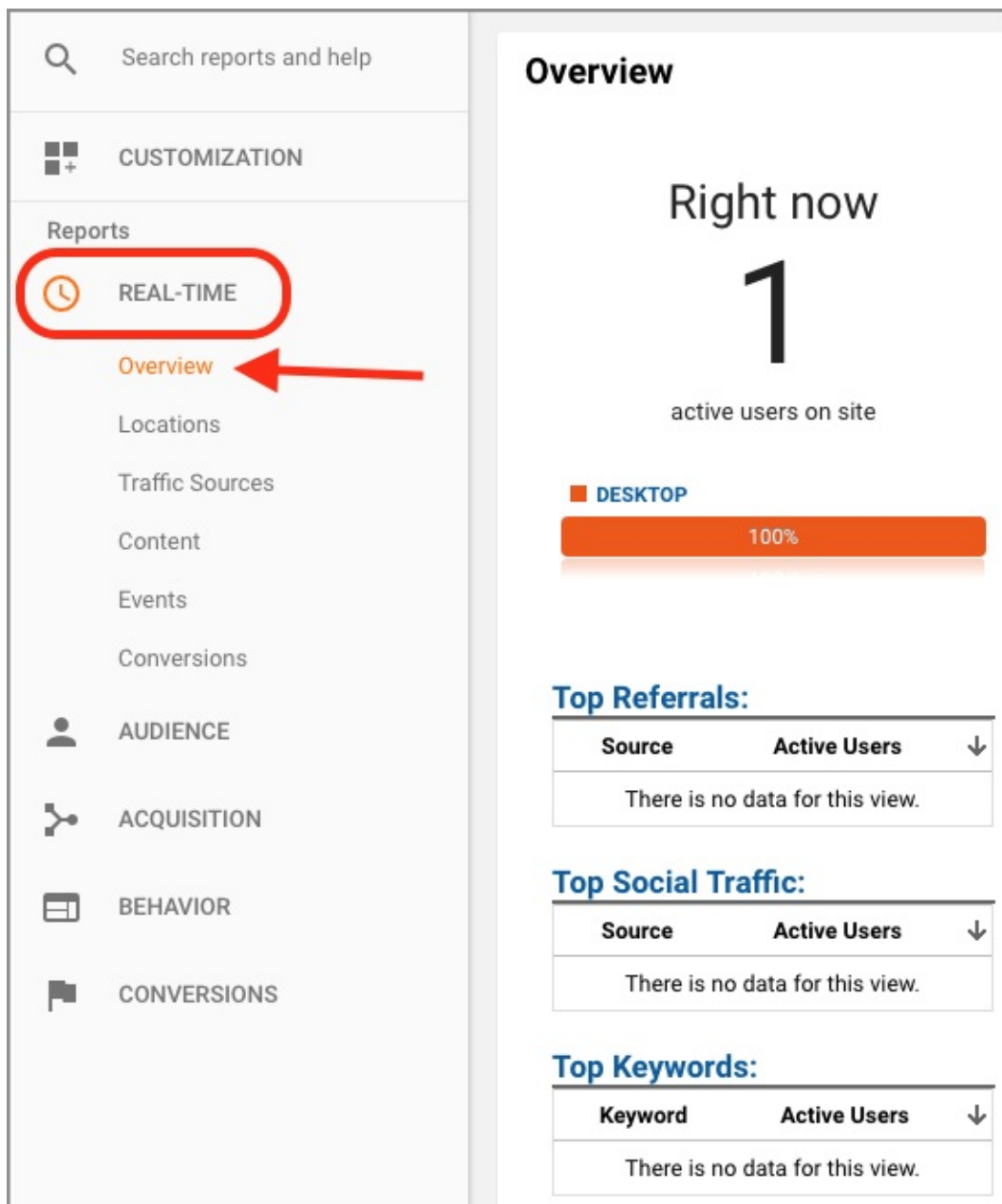


You can also see specific information like visitors' language, country, city, browser, operating system, service provider, screen resolution, and device.

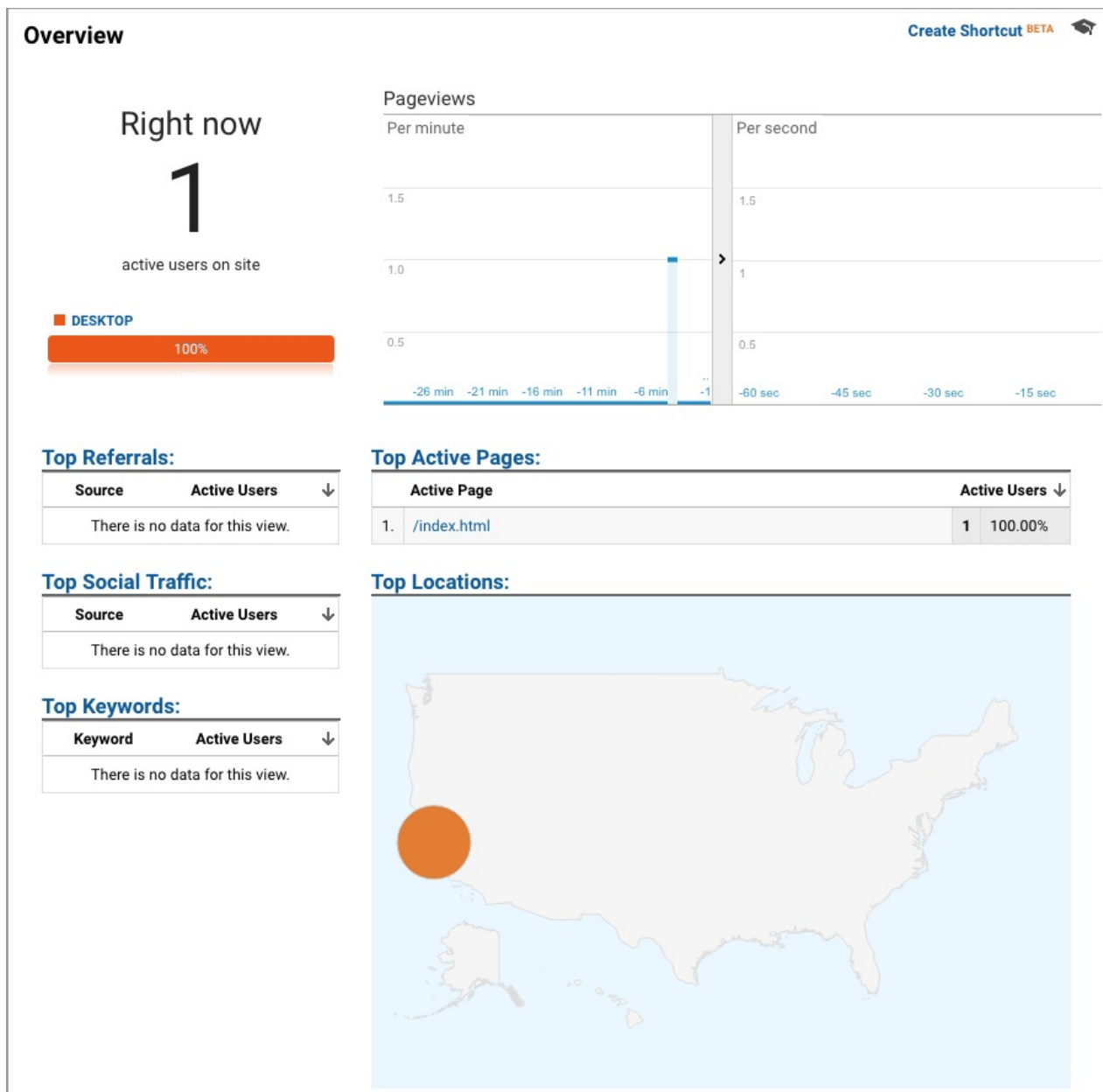
Demographics	City	Sessions	% Sessions
Language	1. Mountain View	146	<div></div> 84.39%
Country	2. (not set)	9	<div></div> 5.20%
City	3. Venice	9	<div></div> 5.20%
System	4. London	3	<div></div> 1.73%
Browser	5. San Mateo	2	<div></div> 1.16%
Operating System	6. Paris	1	<div></div> 0.58%
Service Provider	7. Gig Harbor	1	<div></div> 0.58%
Mobile	8. Lakewood	1	<div></div> 0.58%
Operating System	9. Vancouver	1	<div></div> 0.58%
Service Provider			view full report
Screen Resolution			

Real time analytics

It's also possible to view analytics information in real time from the **Real-Time** tab. The **Overview** section is shown below:



If you are visiting your app in another tab or window, you should see yourself being tracked. The screen should look similar to this:



These are only the basic aspects of the Google Analytics dashboard. There is an extensive set of features and functionality.

For more information

- [Learn about Google Analytics for business](#)

Custom events

Google Analytics supports custom events that allow for fine-grain analysis of user behavior.

For example, the following code will send a custom event:

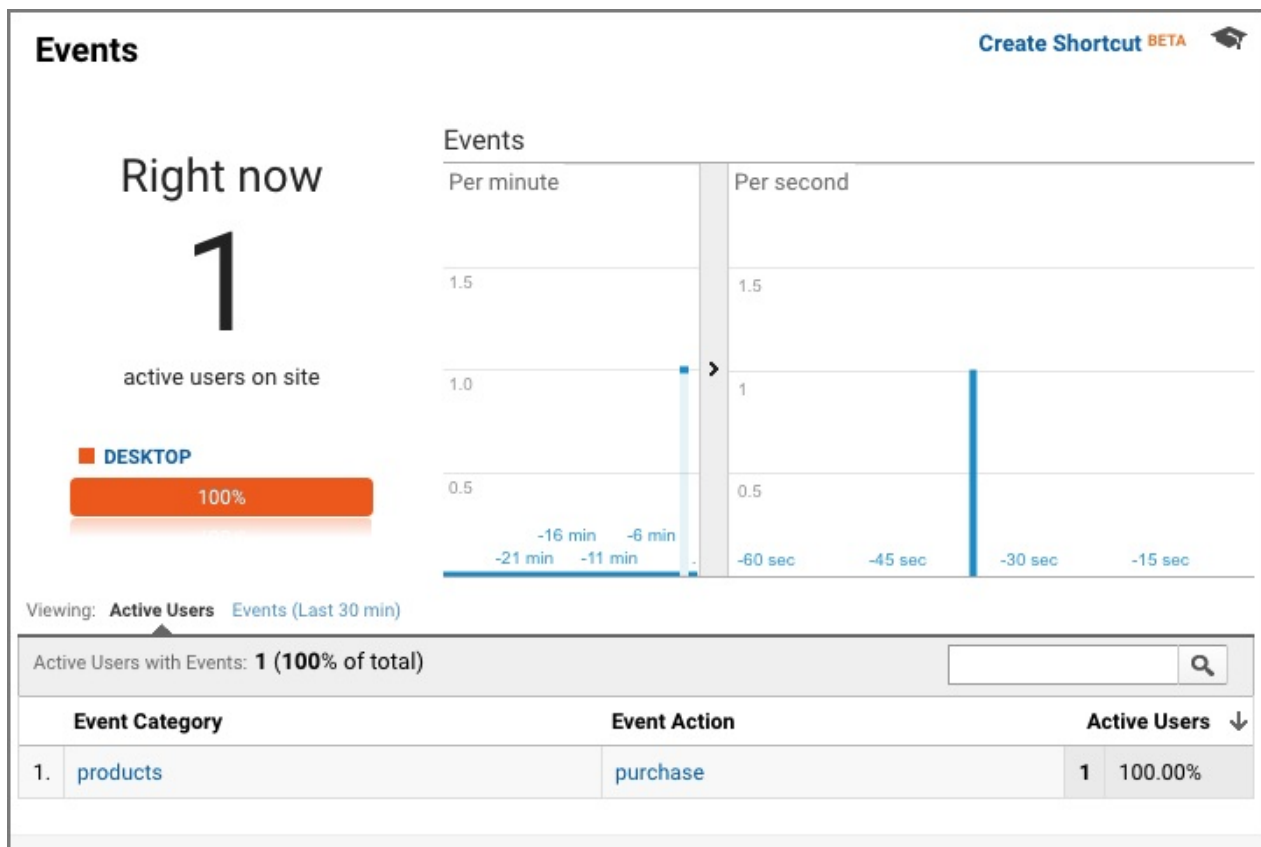
main.js

```
ga('send', {
  hitType: 'event',
  eventCategory: 'products',
  eventAction: 'purchase',
  eventLabel: 'Summer products launch'
});
```

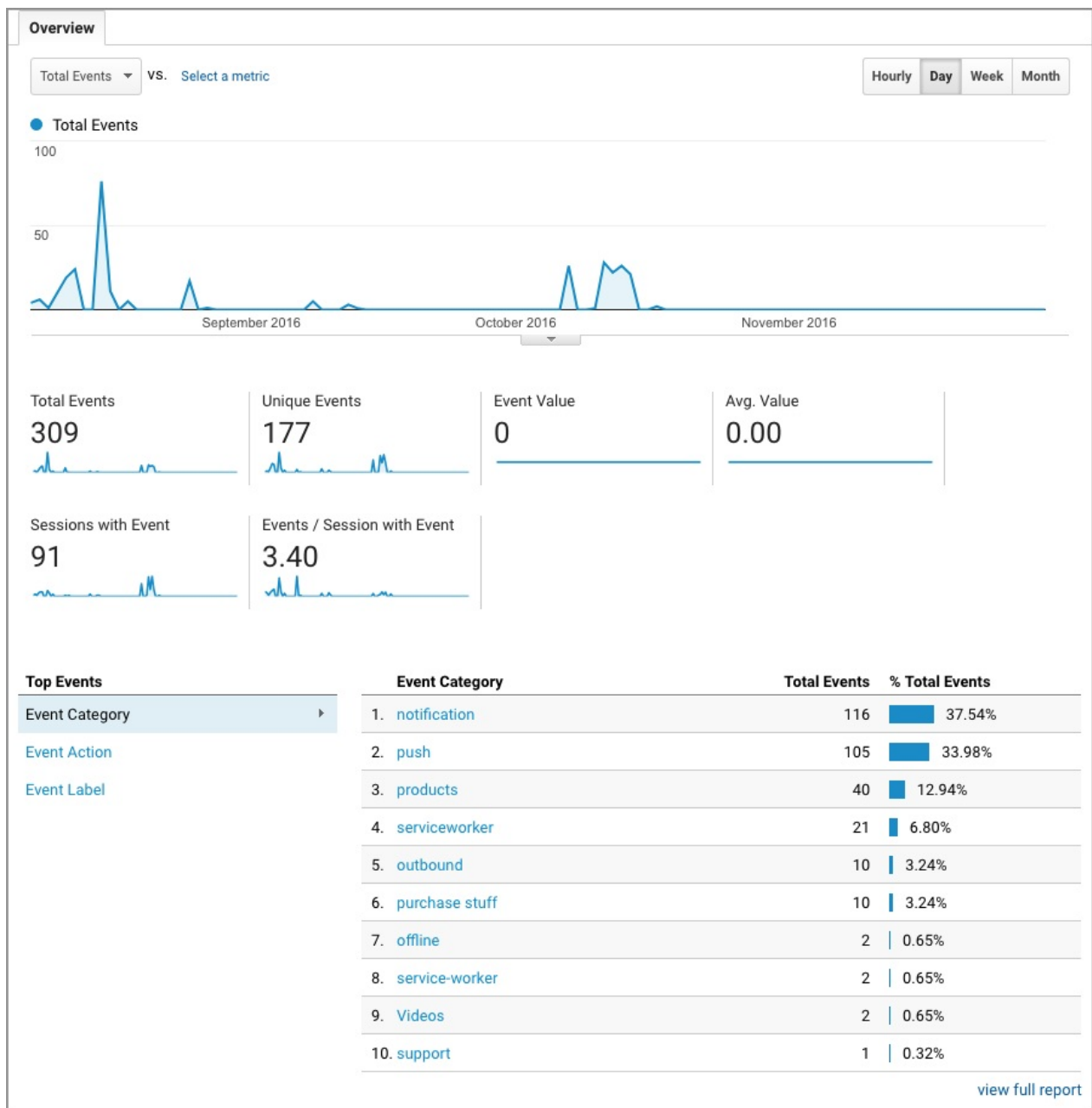
Here the hit type is set to 'event' and values associated with the event are added as parameters. These values represent the `eventCategory`, `eventAction`, and `eventLabel`. All of these are arbitrary, and used to organize events. Sending these custom events allow us to deeply understand user interactions with our site.

Note: Many of the `ga` commands are flexible and can use multiple signatures. You can see all method signatures in the [command queue reference](#).

Event data can also be viewed in the Google Analytics dashboard. Real-time events are found in the **Real-Time** tab under **Events**, and should look like the following:



Here you can see events as they are occurring. You can view past events in the Google Analytics dashboard by selecting **Behavior**, followed by **Events** and then **Overview**:



For more information

- [Event tracking](#)
- [About events](#)
- [The ga Command Queue Reference](#)

Analytics and service worker

Service workers do not have access to the analytics command queue, `ga`, because the command queue is in the main thread (not the service worker thread) and requires the `window` object. You need to use the [Measurement Protocol](#) interface to send hits from the service worker.

This interface allows us to make HTTP requests to send hits, regardless of the execution context. This can be achieved by sending a URI containing your **tracking ID** and the custom event parameters (`eventCategory` , `eventAction` , and `eventLabel`) along with some [required parameters](#) (**version number**, **client ID**, and **hit type**) to the [API endpoint](#) (<https://www.google-analytics.com/collect>). Let's look at an example using the Measurement Protocol interface to send hits related to push events in the service worker.

A helper script, **analytics-helper.js** has the following code:

analytics-helper.js

```
// Set this to your tracking ID
var trackingId = 'UA-XXXXXXXX-Y';

function sendAnalyticsEvent(eventAction, eventCategory) {
  'use strict';

  console.log('Sending analytics event: ' + eventCategory + '/' + eventAction);

  if (!trackingId) {
    console.error('You need your tracking ID in analytics-helper.js');
    console.error('Add this code:\nvar trackingId = \'UA-XXXXXXXX-X\';');
    // We want this to be a safe method, so avoid throwing unless absolutely necessary
    .
    return Promise.resolve();
  }

  if (!eventAction && !eventCategory) {
    console.warn('sendAnalyticsEvent() called with no eventAction or ' +
      'eventCategory.');
```

// We want this to be a safe method, so avoid throwing unless absolutely necessary
 .
 return Promise.resolve();
 }
}

return self.registration.pushManager.getSubscription()
.then(function(subscription) {
 if (subscription === null) {
 throw new Error('No subscription currently available.');

}

 // Create hit data
 var payloadData = {
 // Version Number
 v: 1,
 // Client ID
 cid: subscription.endpoint,
 // Tracking ID
 tid: trackingId,
 // Hit Type

```

    t: 'event',
    // Event Category
    ec: eventCategory,
    // Event Action
    ea: eventAction,
    // Event Label
    el: 'serviceworker'
  };

  // Format hit data into URI
  var payloadString = Object.keys(payloadData)
    .filter(function(analyticsKey) {
      return payloadData[analyticsKey];
    })
    .map(function(analyticsKey) {
      return analyticsKey + '=' + encodeURIComponent(payloadData[analyticsKey]);
    })
    .join('&');

  // Post to Google Analytics endpoint
  return fetch('https://www.google-analytics.com/collect', {
    method: 'post',
    body: payloadString
  });
})
.then(function(response) {
  if (!response.ok) {
    return response.text()
      .then(function(responseText) {
        throw new Error(
          'Bad response from Google Analytics:\n' + response.status
        );
      });
  } else {
    console.log(eventCategory + '/' + eventAction +
      'hit sent, check the Analytics dashboard');
  }
})
.catch(function(err) {
  console.warn('Unable to send the analytics event', err);
});
}

```

The script starts by creating a variable with your tracking ID (replace `UA-XXXXXXXX-Y` with your actual tracking ID). This ensures that hits are sent to your account and property, just like in the analytics snippet.

The `sendAnalyticsEvent` helper function starts by checking that the tracking ID is set and that the function is being called with the correct parameters. After checking that the client is subscribed to push, the hit data is created in the `payloadData` variable:

analytics-helper.js

```
var payloadData = {
  // Version Number
  v: 1,
  // Client ID
  cid: subscription.endpoint,
  // Tracking ID
  tid: trackingId,
  // Hit Type
  t: 'event',
  // Event Category
  ec: eventCategory,
  // Event Action
  ea: eventAction,
  // Event Label
  el: 'serviceworker'
};
```

Again, the **version number**, **client ID**, **tracking ID**, and **hit type** parameters are [required by the API](#). The `eventCategory`, `eventAction`, and `eventLabel` are the same parameters that we have been using with the command queue interface.

Next, the hit data is [formatted into a URI](#) with the following code:

analytics-helper.js

```
var payloadString = Object.keys(payloadData)
  .filter(function(analyticsKey) {
    return payloadData[analyticsKey];
  })
  .map(function(analyticsKey) {
    return analyticsKey + '=' + encodeURIComponent(payloadData[analyticsKey]);
  })
  .join('&');
```

Finally the data is sent to the [API endpoint \(https://www.google-analytics.com/collect\)](https://www.google-analytics.com/collect) with the following code:

analytics-helper.js

```
return fetch('https://www.google-analytics.com/collect', {
  method: 'post',
  body: payloadString
});
```

This sends the hit with the [Fetch API](#) using a POST request. The body of the request is the hit data.

Now we can import the helper script functionality into a service worker by adding the following code to the service worker file:

sw.js

```
self.importScripts('path/to/analytics-helper.js');
```

Where `path/to/analytics-helper.js` is the path to the **analytics-helper.js** file. Now we should be able to send custom events from the service worker by making calls to the `sendAnalyticsEvent` function. For example, to send a custom "notification close" event, we could add code like this to the service worker file:

sw.js

```
self.addEventListener('notificationclose', function(event) {  
  event.waitUntil(  
    sendAnalyticsEvent('close', 'notification')  
  );  
});
```

Observe that we have used `event.waitUntil` to wrap an asynchronous operation. If unfamiliar, `event.waitUntil` extends the life of an event until the asynchronous actions inside of it have completed. This ensures that the service worker will not be terminated prematurely while waiting for an asynchronous action to complete.

Note: Because this event uses the Measurement Protocol interface instead of **analytics_debug.js**, the debug console log won't appear. You can debug Measurement Protocol hits with [hit validation](#).

For more information

- [WorkerGlobalScope.importScripts\(\)](#)
- [Measurement Protocol Overview](#)
- [Simple Push Demo](#) (includes Measurement Protocol example code)

Offline analytics

With the help of service workers, analytics data can be stored when users are offline and sent at a later time when they have reconnected based on an [npm package](#).

Install the package with the following command-line command:

```
npm install sw-offline-google-analytics
```

This imports the [node](#) module.

In your service worker file, add the following code:

sw.js

```
importScripts('path/to/offline-google-analytics-import.js');  
goog.offlineGoogleAnalytics.initialize();
```

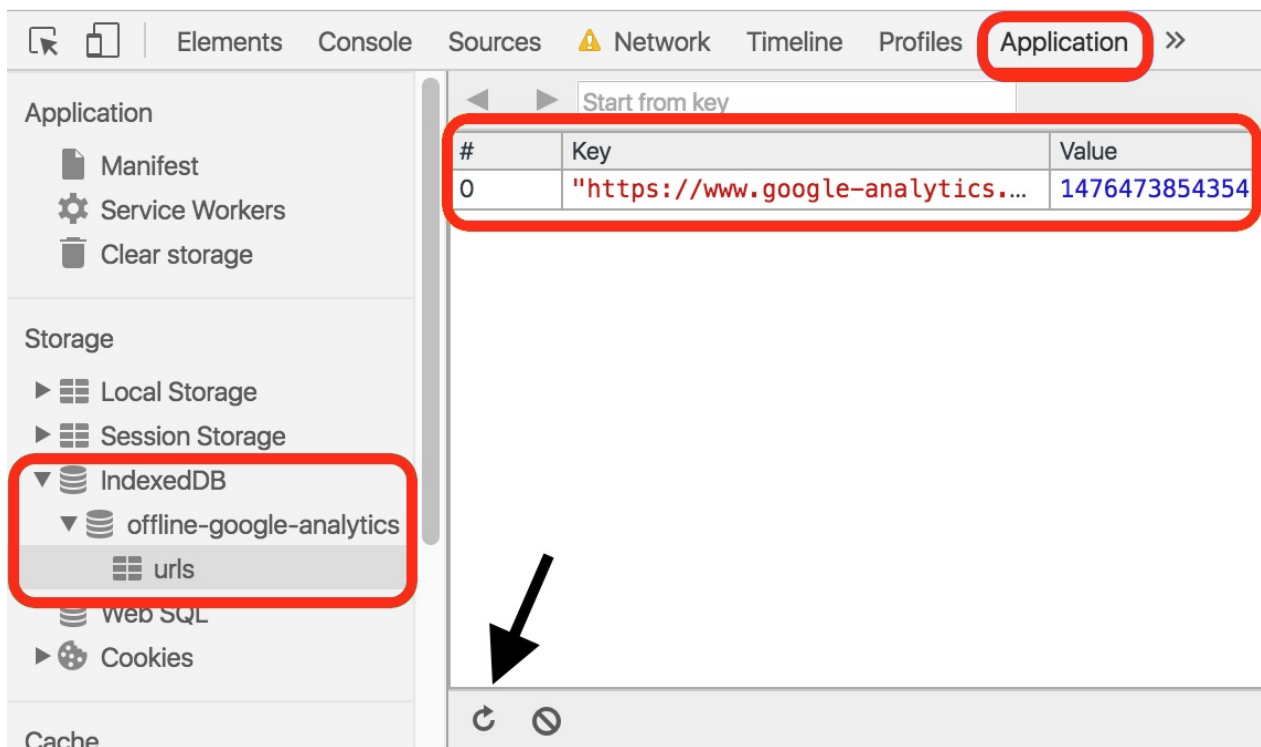
Where `path/to/offline-google-analytics-import.js` is the path to the **offline-google-analytics-import.js** file in the node module. This will likely look something like:

sw.js

```
node_modules/sw-offline-google-analytics/offline-google-analytics-import.js
```

We import and initialize the **offline-google-analytics-import.js** library. This 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 first by network requests. If the network request fails, the request is stored in [IndexedDB](#). The requests are then sent later when connectivity is re-established.

You can test this by [simulating offline behavior](#), and then firing hit events. You will see an error in the console since you are offline and can't make requests to Google Analytics servers. Then [check IndexedDB](#). Open **offline-google-analytics**. You should see URLs cached in **urls** (you may need to click the refresh icon inside the indexedDB interface). These are the stored hits.



Now disable offline mode, and refresh the page. Check **IndexedDB** again, and observe that the URL is no longer cached (and has been sent to analytics servers).

Note: This strategy won't work for hits sent from a service worker since the service worker doesn't listen to fetch events from itself (that could cause some serious problems!). This may not be too important in many cases, because many of the hits that we would want to send from the service worker are tied to online events (like push notifications) anyways.

Note: These events don't use **analytics_debug.js**, so the debug console logs don't appear.

Note: Some users have reported a bug in Chrome that recreates deleted databases on reload.

For more information

- [Offline Google Analytics Made Easy](#)
- [Google I/O offline example](#)
- [Node package manager \(npm\)](#)
- [IndexedDB](#)

Further reading

- [Adding analytics.js to Your Site](#)
- [Learn analytics with free online courses](#) (Google Analytics Academy)
- [Measuring Critical Performance Metrics with Google Analytics](#)
- [Improving session duration calculations](#)

Introduction to the 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 Payment Request 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).

The Payment Request API allows merchants to easily collect payment information with minimal integration. The 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 eliminates manual and tedious entry by storing the user's information securely in the browser. The browser passes addresses and credit card (or other payment) details directly to the website. And, because the browser is collecting payment information from the user, making a payment goes from "*n*" taps to one tap.

Note: The [Payment Request API](#) is very new and still subject to developmental changes, especially while it is under development in the [W3C Web Payments Working Group](#). 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

The Payment Request API is a system that is meant *to eliminate checkout forms*. It vastly improves user workflow during the purchase process, providing a more convenient and consistent user experience and enabling web merchants to easily leverage disparate payment methods. The Payment Request API is not a new payment method, nor does it integrate directly with payment processors. Rather, it is a process layer whose goals are:

- To let the browser act as intermediary among merchants, users, and payment methods
- To standardize the payment communication flow as much as possible
- To seamlessly support different secure payment methods
- To work on any browser, device, or platform—mobile or otherwise

The Payment Request API is currently under development by the [W3C Web Payments Working Group](#). The group's goal is to create a universal cross-browser standard for any website to accept any form of payment. See the [Can I use](#) website for up-to-date browser support.

The Payment Request API's open-web approach is designed:

- 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 (as of this writing the Payment Request API is available on Chrome for Android (v53), Samsung Internet (v5.0), and partially supported on Edge (v15), 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.

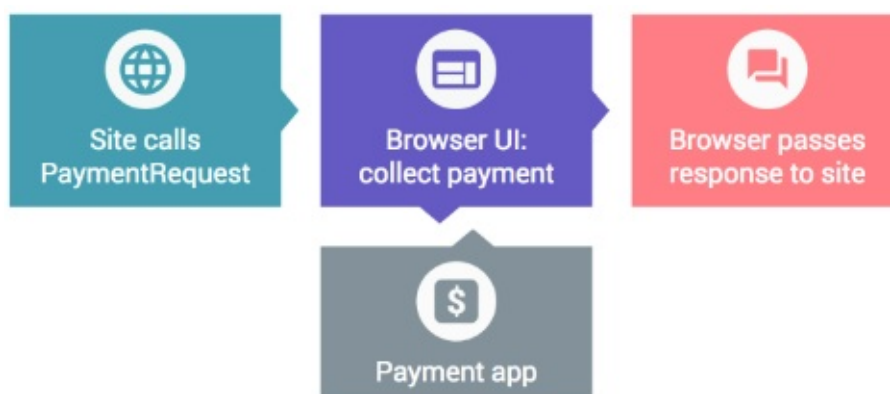
Demos

Payment Request demos are available at these URLs:

- Demo: <https://emerald-eon.appspot.com/>
- Polymer Shop demo: <https://polykart-credential-payment.appspot.com/>
- Simple demos and sample code:
<https://googlechrome.github.io/samples/paymentrequest/>
- Deep dive documentation: <https://developers.google.com/web/fundamentals/discovery-and-monetization/payment-request/deep-dive-into-payment-request>
- Samsung Internet demos:
 - <https://samsunginter.net/examples/payment-request-demo-simple/>
 - <https://samsunginter.net/examples/payment-request-demo-options/>

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

[Bringing Easy and Fast Checkout with Payment Request API](#)

[Payment Request API: an Integration Guide](#)

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](#)

Progressive Web Apps Terminology

add to home screen

A method where Progressive Web App developers can implement web app install banners to help users quickly and easily (with one touch) add web apps to their Android mobile device's home screen, making it easy to launch and return to an app. Android, iOS, Mozilla, and Windows all have different ways to have the browser prompt the user to add to homescreen. For example, Android and Mozilla both use a manifest file or meta tag that is specific to browser requirements while Apple uses meta tags.

AirHorner.com

A simple but powerful Progressive Web App that shows the power of service workers and the Web Audio API to give you your very own air horn. AirHorner works no matter whether you're online, offline, or on a flaky 2G network.

AMP - Accelerated Mobile Pages

A set of tools that uses these core components to build web pages and quickly load initial content in a browser:

- AMP HTML - HTML customized with AMP-specific tags that make common patterns easy to implement in a performant way
- AMP JS - A library that ensures the fast rendering of AMP HTML pages
- AMP Cache - A proxy-based content delivery network used to serve cached AMP HTML pages

app

A short-hand term for application, as in "[webapp](#)," "[progressive web app](#)," or "[native app](#)."

application shell (App Shell)

Separates the common HTML, CSS, and JavaScript necessary to power the user interface of a progressive web app from the main content of the web page. This PWA technique ensures reliably good performance. Larger applications might load a subset for the initial load and then dynamically load additional code, styling, and HTML to service user requests.

Bower

A package manager for web components that installs the latest versions of HTML, CSS, JavaScript frameworks, libraries, fonts and image files.

cache

There are two types of caches in the browser: browser-managed cache and application-managed cache (service worker).

- **Browser-managed caches** are 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. This cache is managed automatically by the browser and is not available offline.
- **Application-managed caches** are created using the [Cache API](#) independent of the browser-managed caches. This API is available to applications (via `window.caches`) and the service worker. Application-managed **caches** hold the same kinds of assets as a browser cache but are accessible offline (e.g. by the service worker to enable offline support.) Code that uses the Cache API manages the contents of its own caches.

Cache is a great tool you can use when building your app, as long as the cache you use is appropriate for each resource. Several caching strategies are described in the PWA [Caching Strategies](#) tutorial.

codelabs

[Google Codelabs](#) provide guided tutorials with a hands-on coding experience.

content delivery network (CDN)

A network of geographically distributed servers that cooperate to satisfy requests for content. CDNs optimize content delivery by distributing copies of files (such as videos, images, HTML, CSS and JavaScript) to multiple servers. This reduces latency by placing the content closer to the requestor. For example, if a user in India requests a web page from a Brazilian website, the request could be rerouted to deliver assets served from a local CDN server in Mumbai.

e-commerce

Electronic commerce, commonly written as e-commerce or eCommerce, is the buying and selling of goods and services using computer networks, such as the Internet or online social networks. Also referred to as e-business.

See also [m-commerce](#).

Fetch API

A simple API for fetching resources. [Fetch](#) makes it easier to make web requests and handle responses than with the older [XMLHttpRequest](#), which often requires additional logic (for example, for handling redirects). Fetch is [promise](#) based.

Firebase

The collection of common back-end services that most apps need. Because Google manages the Firebase back-end services and provides cross-platform libraries, app developers can use instant notifications, real-time analytics, cloud messaging, and other features running on the Google infrastructure.

(FYI, Chrome requires either using Firebase or a [VAPID](#) identifier when subscribing to the [push service](#).)

HTTPS

Most modern web APIs (including those used for progressive web apps) require HTTPS ([Secure HTTP](#)). HTTPS encrypts the connection between the server and client, helping to protect users' information and prevent tampering. APIs such as the Service Worker, Google Maps API, and File API must be called from a script served over HTTPS (technically, "called from a [secure context](#)"); this prevents an attacker from tampering with the script and getting access to these features.

HTTP is implemented using the TLS protocol. Although TLS supersedes SSL it is often referred to as SSL.

hybrid app

A [native app](#) that is developed with HTML, CSS and JavaScript (often using a framework such as Cordova) running in the platform's own web view. This allows applications to be distributed via an app store and use native OS features that don't have a web equivalent.

Lighthouse

An extensible, open source development tool that tests and provides feedback on aspects of your progressive web application. It is available as both a command-line tool and a Chrome extension for Chrome 56 and later.

m-commerce (mobile commerce)

The buying and selling of goods and services through wireless handheld devices, such as cellular telephone and personal digital assistants (PDAs). Known as *next-generation e-commerce*, m-commerce enables users to access the Internet without needing to find a place to plug in.

See also [e-commerce](#).

manifest

See [web app manifest](#)

native application (native app)

An installable application that is developed for a specific platform (such as iOS, Windows, or Android). Because it is tied to a specific platform, a native app can take advantage of operating system features and other software available on that platform.

Similarly, a native *mobile* app is developed for a specific platform (such as iOS, Windows, or Android) to run on mobile devices. Native mobile apps can provide fast performance and a high degree of reliability, have access to functions such as the camera and address book on mobile devices, and can often run offline (the apps run without an Internet connection).

progressive web app (PWA)

A web-design approach that encourages a new level of caring about the quality of the user experience. PWAs leverage modern web standards to deliver native application performance with the added benefits of the Web's global reach and instant deployment. Built using Service Workers and other modern web capabilities, a progressive web app can *progressively* change through use and user consent to provide an experience similar to a native app with offline support, push notifications, and the ability to be added to the home screen.

See also [webapp](#).

promise

An object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation. [JavaScript promises](#) are a new addition to ECMAScript 6 that provide a cleaner, more intuitive way to deal with the completion (or failure) of asynchronous tasks. Promises can be chained so that, as the promise methods return promises, the output of one function serves as input for the next. Service workers are promise based. See [Is Service Worker Ready?](#) for an up-to-date list of browsers that support promises.

push notifications

Web push notifications make it easy to re-engage with users by showing relevant, timely, and contextual notifications, even when the browser is closed.

In a PWA environment using service workers, all the work to perform a push is done by the browser and the operating system. To deliver a Push Notification to a user, send a message from your server to the push messaging service used by the browser (e.g. Firebase for Chrome) along with an ID to identify the recipient. On the receiving end, the browser wakes up the service worker, which then fires a push event, allowing your web app to display a notification. This series of events happens even if the browser isn't open, which saves battery and CPU usage.

responsive apps and websites

A website that is structured and coded to use a single URL and deliver the same HTML to both mobile and desktop devices, regardless of the viewport size. This way, content looks great on displays of any size or type, such as phones, tablets, and laptops. You can also add media queries to dynamically tailor CSS for particular viewports (for example <http://udacity.github.io/responsive-images/examples/3-03/sizesMediaQuery>, <https://simpl.info/mq>, or using JavaScript as with <https://simpl.info/video/mq>).

In contrast, so-called mdot sites deliver different files to different devices, by detecting the device type (the [user agent](#)) and then delivering files appropriate to either a mobile or a desktop experience.

same-origin policy

Web security for many modern APIs and progressive web apps is rooted in the [same-origin policy](#). The single-origin model is designed to protect servers from malicious browser scripts. That is, code from `https://mybank.com` should only have access to `https://mybank.com` data, and `https://evil.example.com` should never be allowed to access `mybank.com`. The browser enforces these restrictions, blocking any illegal attempts.

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 requests made from web pages (including HTTPS). For example, a service worker can intercept outgoing network requests and provide alternatives when the network is offline (such as by serving up cached web content). Service workers can also respond to incoming events, such as push notifications.

Service workers 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.)

single-page applications (SPAs)

Snappy user-friendly apps that perform more like desktop apps. Typically, SPA content is rendered dynamically using JavaScript, rather than opening a new page. SPAs load data asynchronously so users can do something else while the data loads. For example, you can start multiple emails in Gmail in parallel without waiting for the first process of starting an email to finish successfully. This improves user experience because SPAs can function more like desktop apps.

Single-page applications typically load only data (e.g. in JSON or XML format) rather than pre-rendered HTML. This decreases the data transferred on the wire. Google's headline web apps—such as GMail, Inbox, Maps, Docs, Sheets, and so on—have been pioneers here.

splash screen

A graphic that is shown to users while an application is loading in the background, giving immediate feedback to the user while the application is launching.

user agent

Software (a software agent) that acts on behalf of a user. A user agent on the web could be a web browser or other application that provide information about themselves to a web server (website).

web app

A web application that is implemented using web technologies and is executed within the context of a web user agent (such as in a web browser or other web run-time environment).

See also [progressive web app](#).

web app manifest [W3C Spec](#)

A JSON-formatted file named **manifest.json** that is a centralized place to put metadata that controls how the web application appears to the user and how it can be launched. (Do not confuse this with a `manifest` file used by AppCache.) PWAs use the web app manifest to enable "[add to homescreen](#)".