**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](https://developers.google.com/web/fundamentals/getting-started/primers/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](https://developer.mozilla.org/en-US/docs/Web/API/Cache)interface to cache an application's assets. A service worker script can implement a number of [caching strategies](https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook), 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](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API) 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.

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 or syncing data in the background). 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?](https://jakearchibald.github.io/isserviceworkerready/)

**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](https://developers.google.com/web/fundamentals/instant-and-offline/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](https://jakearchibald.com/2014/reading-responses/) 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](https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook) for more info.