**Introduction to Service Worker**

**What is a service worker?**

A [service worker](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API) is a type of [web worker](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API). 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](https://web-push-book.gauntface.com/chapter-07/01-faq/#why-doesnt-push-work-when-the-browser-is-closed) in Matt Gaunt's [Web Push Book](https://web-push-book.gauntface.com/) for more information.

* The service worker can't access the DOM directly. To communicate with the page, the service worker uses the [postMessage()](https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage) method to send data and a "message" event listener to receive data.

Things to note about Service Worker:

* A 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 <https://letsencrypt.org/> 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](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API) 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](https://developers.google.com/web/fundamentals/getting-started/primers/promises).

**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](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) (a standard way to retrieve content from the network) and [Cache](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developers.google.com/web/ilt/pwa/lab-caching-files-with-service-worker) and [The Offline Cookbook](https://developers.google.com/web/fundamentals/instant-and-offline/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](https://developers.google.com/web/ilt/pwa/lab-caching-files-with-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](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API): A way to display and interact with notifications using the operating system's native notification system.
* [Push API:](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developers.google.com/web/updates/2015/12/background-sync): 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](https://developer.mozilla.org/en-US/docs/Web/API/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. If the service worker is already installed, navigator.serviceWorker.register returns the registration object of the currently active service worker.

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 you want the service worker to control higher pages e.g. /app (without the trailing slash) you can indeed change the scope option, but you'll also need to set the [Service-Worker-Allowed HTTP Header](https://w3c.github.io/ServiceWorker/#service-worker-script-response) in your server config for the request serving the service worker script.

**main.js**

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

**Your server config**

**Note:** This is an example of adding the **Service-Worker-Allowed** header to the service worker script response using the popular [nginx web server](https://www.nginx.com/resources/wiki/start/topics/examples/full/). If you're using a different web server, please refer to its documentation on setting response headers to make the equivalent change.

location ~\* (service-worker\.js)$ {  
  # tells browsers the service worker scope  
  add\_header 'Service-Worker-Allowed' '/app';  
}

**Installation**

Once the 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](https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook#on-install-as-dependency)). 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](https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook#on-activate) 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()](https://developer.mozilla.org/en-US/docs/Web/API/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](https://developers.google.com/web/ilt/pwa/tools-for-pwa-developers).