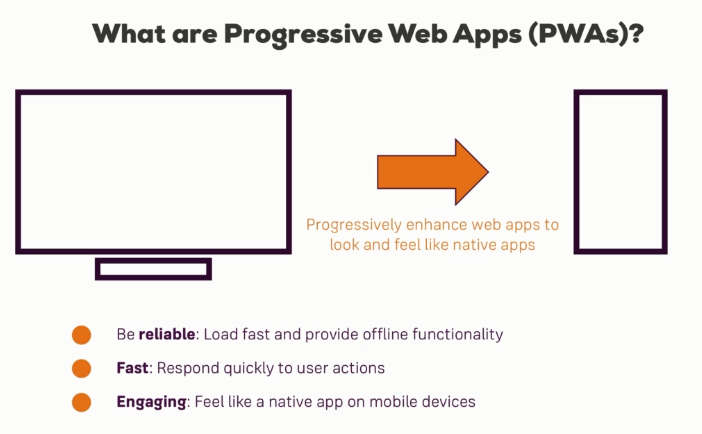
PWA  
(Progressive Web Apps)

# Getting Started

## What are Progressive Web Apps?

Docs: <https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps>

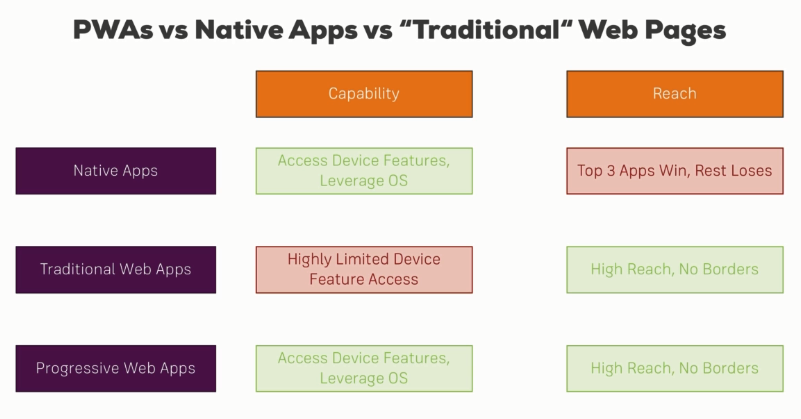


* Progressive web apps in the end describe a set of features of technologies you can add to your existing web applications to turn them into native apps like native mobile app like experiences using device features like the camera and providing offline access.
* You **progressively enhance** your existing web pages to feel and work more like native mobile apps.
* Things like your app working if you're offline, it's having an icon on the home screen, things like accessing the device camera or the location, synchronizing data in the background. These are all features which were hard to do in web applications in the past, nowadays we got browser support in a lot of browsers for that and we can therefore use that.
* You progressively enhance a web application. It's not an all or nothing move. It's not like it doesn't work on older browsers anymore. It just means if you have a modern browser on a mobile device, you get an awesome experience otherwise you'll get the experience you've gotten anyways.

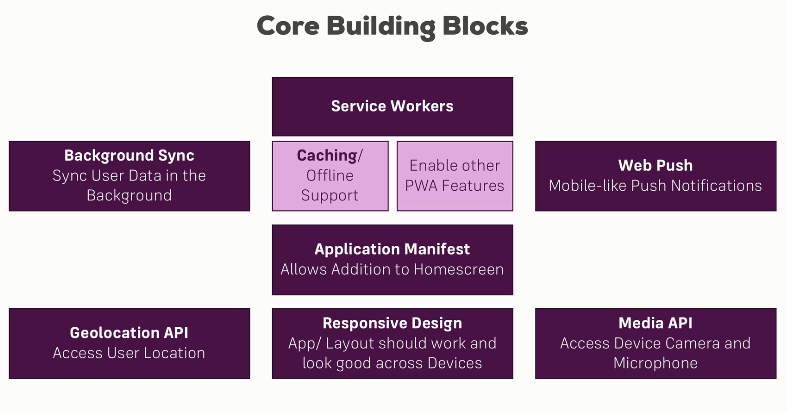
### Three main things that summarizes PWAs

1. They should be **reliable** which means they should **load fast** and even work if you're **offline**, at least a part of the application should work if you're offline.
   1. Now this is really a core thing of progressive web apps. We're talking a lot about that initial load, the first time you visit an application when you open it because if you consider an application running on your mobile device, most of them start up pretty fast, so long loading times you don't want that.
2. Additionally it should be **fast**, not just during load up but also once it runs, it should react to user input, it should provide animations, it should be able to access native device features in an intuitive way
3. And we also want to make sure that it's **engaging**. We want to get our users back into the app, we want to offer features like push notifications to inform them even if the application is closed.

## PWAs Vs Native Mobile Apps



## PWA Core Building Blocks



### Service Workers

* **Service workers are basically JavaScript running in a background process, even if your application is closed.**
* Service workers are supported in modern browsers like Chrome.
* Service workers are a core building block because they allow us to get offline access, to cache some files and serve them if we don't have internet connection.
* And they also give us access to other progressive web app related features for example background synchronization, sending a request once internet connection is re-established. Push notifications would be another example because they are running in the background independent of currently opened tabs.

### Application Manifest

* The Application Manifest makes your application **installable** on home screens, not installed through App Store but instead, you can basically install a web app.

### Responsive Design

* App should work and look good across devices.

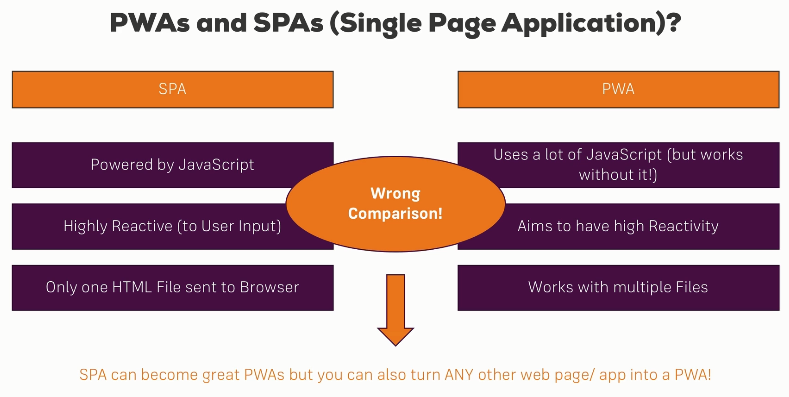
### Geolocation API

* To access user’s location.

### Media API

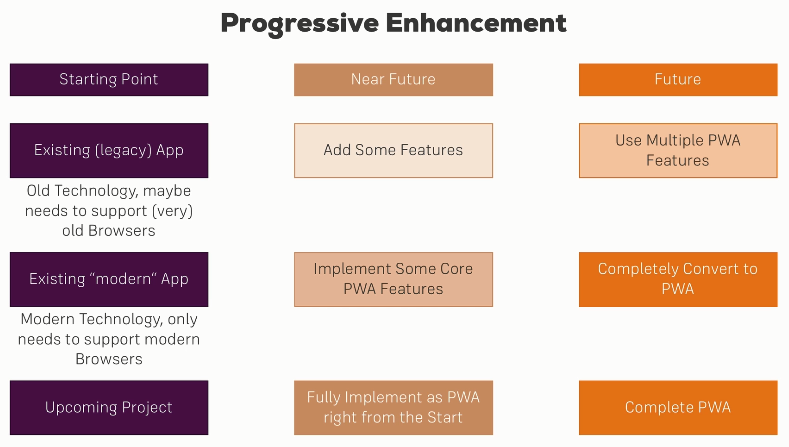
* To access device camera and microphone.

## Comparing PWAs with SPAs



* You can turn any single page application into a progressive web app and the same is true for any traditional multi-page application where you render your views from your server.
* A progressive web app is just a collection of technologies, can be used anywhere.

## What is “Progressive Enhancement”



* The term “Progressive” in PWA basically means that we can progressively enhance our web application.
* With existing project, we can add features step-by-step and you can stop at any given point. You can stop after adding an app manifest and a basic service worker. You can stop after you implement the basic caching, advanced caching, push notifications. You can stop all the time.
* For the upcoming project, you can start from scratch obviously. So you can fully implement it as a progressive web app right from the start and plan everything with that in mind.
* It’s not all or nothing. You don't have to use 0 or 100%, you can absolutely use 40% of all the features, pick the ones which enhance your application and add them.
* So basically you can “progressively” enhance any application (legacy, modern, upcoming) to use different features of PWA.

# Understanding the App Manifest

## App Manifest File

* The web app manifest is just one single file you add to your project, where you can pass some extra information to the browser.
* Now the browser can take this information to display some things about your application differently and the browser depending on which operating system it's running on can even allow your app to install itself on the home screen of your user. Then it will feel and look like a native app, so we can open it like we open any native app.

### Advantages –

* If users just visit our web page via browser, they might enjoy it but they constantly have to type the URL or manage the bookmark.
* But with a home screen icon which we can get via the app manifest, we increase user interaction with our application.

## Adding the Manifest file

* Web manifest simply is a file we add to our root web folder, so the folder containing our index.html file.
* The web manifest file name must be **manifest.json**
* Now include this manifest.json file in EVERY page (.html file) in your application.

E.g.

<link rel="manifest" href="/manifest.json">

* If you have SPA, you just need to one this in the one index.html file. For multipage applications, you must add this entry in each .html page. This is because you want to ensure that the browser is able to load that manifest with your additional configuration, no matter which URL you visit.

## Understanding App Manifest properties

* MDN Web app manifests   
  <https://developer.mozilla.org/en-US/docs/Web/Manifest>
* Web App manifest explanation by Google.  
  <https://web.dev/add-manifest/>
* **name** – allows us to assign a name to our progressive Web application. The name will be fetched in use by the browser in various places. For example on a splash screen we get once we edit the app to the home screen.
* **short\_name** – This is basically the shortened words of our full name and it is the name which is shown below the app icon. The browser will use this name whenever the space for the name is limited.
* **start\_url** – means which page should get loaded on startup when we tap this app icon.
* **scope** – This means which pages are included in our “PWA Experience”. A dot (.) means all the pages in our web folder.
* **display** – allows us to define how the app should actually load once it was added to the home screen.
  + standalone – means it looks like a full app. And we don't see the browser controls at the top, would still have a toolbar. We don't see URL bar and so on.
  + fullscreen – which basically means that it will cover up the whole screen and whilst standalone still would have a toolbar and basically behave like a standalone app, fullscreen is like the fullscreen native apps you know where you don't see any controls at all. Not typically the choice you take for web applications but you could do that.
  + minimal-ui – this will also feel like a standalone application but with a reduced amount of UI elements.
  + browser – which will open it like a normal web page in the browser.
* **background\_color** –
  + The color will display at the back of your application while it is loading and on the splash screen, so on the loading screen.
  + This is simply something which is only shown a fraction of the time but it allows you to make sure that you control this short period of time.
* **theme\_color** – This controls how for example your toolbar at the top will look like especially on a native device in the tasks switcher.
* **description** – is used for example you use when you save the application as a favorite in the browser. So whatever the browser wants additional description to use it will fall back to this field here.
* **dir** – read direction of your app. Be default it is ltr (left to right).
* **lang** – Default or main language of your application. E.g. en-US. This is also useful for the browser so that it can identify how to load your app and for whom your application actually is.
* **orientation** –
  + here you can basically control it in which orientation your web app should be opened if you tap the homescreen icon.
  + This allows you to basically ensure that what users are viewing the app the way you want to view them.
  + But keep in mind restricting the orientation is something you want to use with care because typically users don't like being restricted there.
  + Allowed values – any, portrait, landscape, portrait-primary.
* **icons** – allows us to configure a set of icons where the browser will choose the best fitting one so best fitting for it to give them the why is device density or screen density and so on. You typically provide same icons with multiple sizes and then the browser chooses the best fit for given device. At least have 48 X 48 size. You can also have 128 X 128 or all the way up to 512 X 512.
* **related\_applications** –
  + here you can basically set up related native applications, not web applications the user might be interested in installing for your app.
  + If you have for example a web app where you all have a real native app as an alternative you might specified here and then the browser can choose if he wants to display this choice to the user.

## Preparing the Emulated Device for ALL PWA Features

* We can set up an emulated Android device to test the PWA.
* Here are some more details about the process and how to prepare the device for the rest of the course though.

### **Setting up Android Studio:**

* In case you don't have Android Studio installed, make sure to do so. It's free!
* You can download it from this URL: <https://developer.android.com/studio/index.html>
* We only install it to get easy access to the Android Virtual Device (AVD) Manager though. You can access that Manager under "Tools" => "Android" => "AVD Manager".
* Detailed instructions on how to create a device with it can be found here: <https://developer.android.com/studio/run/managing-avds.html>

### **Updating Chrome on the Virtual Device**

* With an emulated device up and running, you're well-prepared to test your manifest.json file. For other features you learn about in this course, the pre-installed Chrome browser is too old though (at least at the point of time this course is created).
* You can easily update Chrome on your virtual device though. Get an updated APK (basically the app installation file) from this link: <https://www.apkmirror.com/apk/google-inc/chrome/#variants>
* Feel free to choose the latest one, download and install it. Give the device the permission to install from "unsafe" sources. If it fails, try a different APK version.

## Criteria for installing the Web app (as PWA)

* What does it take to be installable?  
  <https://web.dev/install-criteria/>
* Customizing the install experience: How to provide your own in-app install experience?  
  <https://web.dev/customize-install/>
* Patterns for promoting PWA installation  
  <https://web.dev/promote-install/>

## Adding Browser Specific Properties

* Chrome detects the manifest.json and accordingly will do the needful (e.g. showing popup for ‘Add the Home screen’, etc.)
* Safari does not detect manifest.json as of now. For that, we need to add additional data.

### For Safari

* Add below lines in all the html files in your app.

<!-- For Safari browser -->

  <!-- we want to treat this as a mobile web app which also allows us to add it the home screen -->

  <meta name="apple-mobile-web-app-capable" content="yes">

  <!-- how the status bar will display on the apple devices. -->

  <meta name="apple-mobile-web-app-status-bar-style" content="black">

  <!-- title for your installed web app. By default, it will use title from <title> tag above. -->

  <meta name="apple-mobile-web-app-title" content="PWAGram">

  <!-- which icons to use on Apple devices. BTW apple-icons\*.png are icons optimized for apple. -->

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-57x57.png" sizes="57x57">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-60x60.png" sizes="60x60">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-72x72.png" sizes="72x72">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-76x76.png" sizes="76x76">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-114x114.png" sizes="114x114">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-120x120.png" sizes="120x120">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-144x144.png" sizes="144x144">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-152x152.png" sizes="152x152">

  <link rel="apple-touch-icon" href="/src/images/icons/apple-icon-180x180.png" sizes="180x180">

### For Internet Explorer

* Add below lines in all the html files in your app.

<!-- For Internet Explorer browser -->

  <!-- this will be the image used on this Windows tile then if we save this to the home screen -->

  <meta name="msapplication-TileImage" content="/src/images/icons/app-icon-144x144.png">

  <!-- the background color behind the icon -->

  <meta name="msapplication-TileColor" content="#fff">

### For all browsers

* Add below lines in all the html files in your app.

<!-- This is same as theme\_color property in the manifest.json file. Will be used to other browers -->

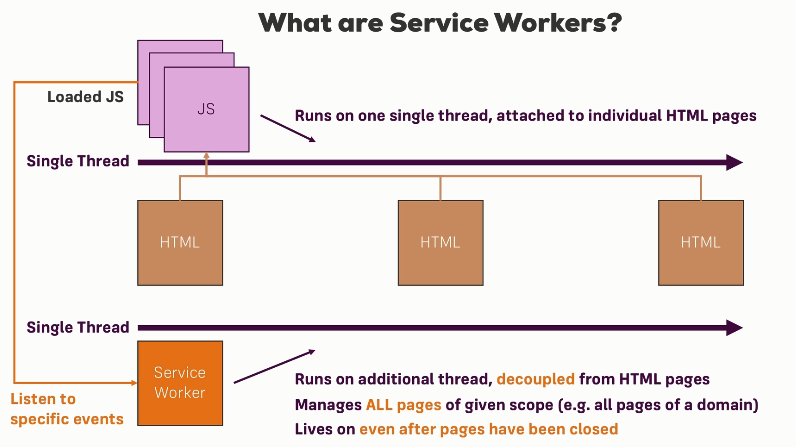
  <meta name="theme-color" content="#3f51b5">

# The Service Workers

## Why Service Workers are Amazing

* They do a lot of work behind the scenes.
* They allow us to make our application offline ready so that it works even if we have no internet connection and they allow us to use a lot of other next generation web application features, like push notifications or background synchronization.

### What are Service Workers



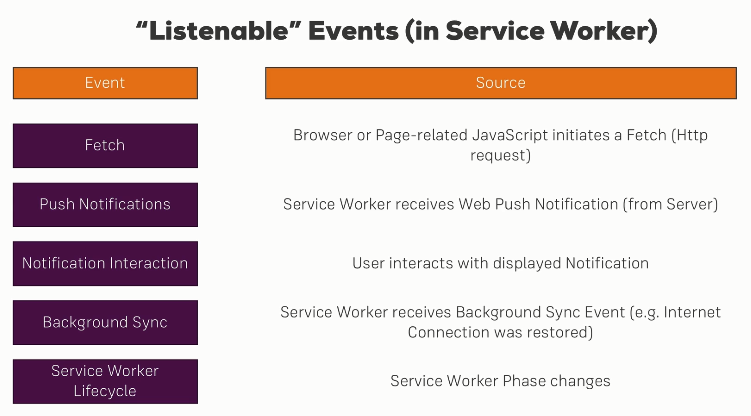
#### Background

* Javascript (loaded from your HTML page) runs on a single thread. That means that if you visit a web application and that page or that application obviously returns you a HTML page which then execute some Javascript. The Javascript code loaded by this page runs on **one thread**. Now even if you have multiple Javascript files loaded by one and the same page, they still share one thread on which they execute their code.
* Now these Javascript files loaded by HTML pages can access the DOM of these pages. You can show alerts, prompt the user for input, change the DOM, manipulate it, etc.
* Javascript frameworks like React, Angular and so on, they run exactly like this. They are loaded as JavaScript by your page and they therefore can manipulate the DOM hence allowing you to create single page applications or add widgets to your pages.

#### How do Service Workers fit into this

* Service workers are also Javascript files, Javascript code. However they have access to a different set of features.
* They also run on a **separate single thread**.
* They don't share the same thread as the normal Javascript code loaded by your HTML pages runs on. They have their own thread because service workers run in the background. So they also are decoupled from HTML pages.
* Whilst you do register a service worker through HTML code or through an HTML page initially, but once you registered it, it simply has a certain scope where it applies to, for example that could be the domain of your page.
* So the service worker is then not attached to a single page but available to all the pages of your web application. So if you have multiple HTML files served from different URLs and the service worker is attached to your root page, to your domain therefore, then it is applicable to all these HTML pages and not attached to a single page like your normal Javascript is.
* Service workers also live on even after you closed all the pages in the browser. So even if you close the tab with your web application and you only have other tabs open or sometimes for example, on mobile phones, even after you close the browser, the service worker still keeps on running, it's a background process.
* So service workers can't interact with the DOM, they are not attached to a page, they are attached to a certain scope, for example your entire web application, a domain and they run in the background.
* Since they run in the background, they're really good reacting to events. They can listen to some events, either emitted by your normal Javascript pages or also by your HTML code or by another server, like web push notifications.
* So service workers are all about **reacting to events**. They sit there in the background and they do nothing but they can react to all kinds of incoming events and then do something, like for example return a cached asset or show a notification to the user.

## Service Worker Events



### Fetch Event

* Fetch is one of the most important events, it is triggered whenever the browser or page-related Javascript, so Javascript running because it was loaded by the index.html or by any HTML page initiates a fetch which is a HTTP request.
* Now that could be because you have an image tag in your HTML code and when the browser tries to get that image to display it, it actually sends such a **fetch** request.
* You can react to that fetch request in the service worker and you can basically think of the **service worker serving as a network proxy**. Every HTTP request sent via fetch and that is every request sent by the browser due to HTML loading some assets goes through the service worker.
* So whenever you load a CSS, JavaScript file because you import it in your HTML file, that triggers **fetch**.
* Whenever you use the fetch API in your own Javascript code, you also trigger such a fetch request.
* **Important:** You don't trigger a fetch request if you use a normal traditional Ajax request in Javascript, so the XmlHttpRequest or any package like axios which builds up on that, that won't trigger the fetch event.
* So with that, the cool thing is since most requests go through the service worker as soon as you start listening to the fetch event at least, you have ways of manipulating these requests. You can block them, you can return cached assets, you can control them basically, so really powerful.

### Push Notification Events

* Push notifications are sent from another server.
* Basically **every browser window like Google for Chrome and Mozilla for Firefox has its own push web push server**.
* You can send push notifications to these browser web push servers from your own server and then these browser web push servers will send this push notification to your client application and in the service worker, you can listen for such a push event, so for such an incoming push request.
* Why do you do this in the service worker and not in the normal Javascript page?   
  Because remember, service workers live in the background and they even live once all your pagers were closed. So push notifications obviously are all about getting the user back into the application after he closed it.
* If you are still in the application, yes you maybe still want to push notify the user but it's far more valuable if your phone is in your pocket, make your phone ring, give you a push notification. With service workers, you can get this behavior.

### Notification Interaction Events

* Now once you've got that push notification, you often want to show an alert, a notification to the user in your application or on the phone.
* Now if the user interacts with that, for example he taps on that notification, you can also listen to that interaction in the service worker to do something with it, show an example page, load something from the cache, whatever you want to do. So you can react to the user interaction with notifications in the service worker.

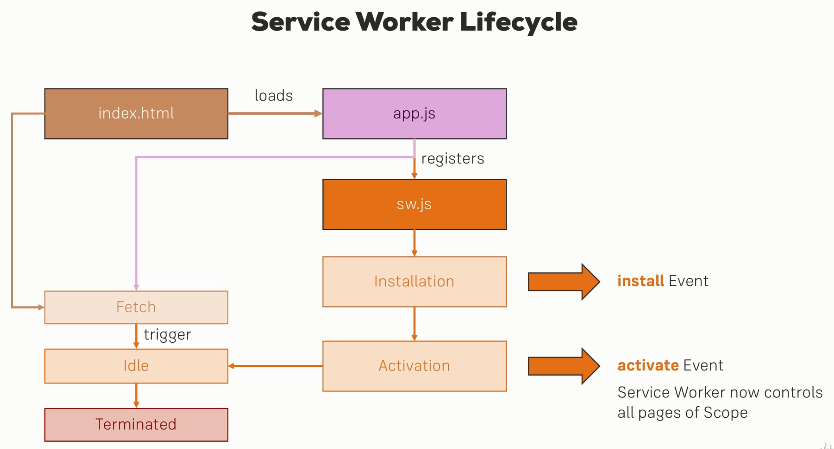
### Background Synchronization Events

* Imagine a case where you don't have that good of an internet connection and you send a post. Now if the internet connection is bad, that will fail. Now some browsers, mainly Chrome right now but support is getting better as for all PWA features, allow you to use background synchronization, which means you store a certain action if it can't be executed right now and you execute it once Internet connection was re-established.
* When this is the case, the browser which supports background synchronization will emit a certain event to which you can listen in the service worker.
* Since the service worker is still alive (runs in background) and therefore if internet connection is established again, you can react to it in the service worker and execute that action you stored which you couldn't execute earlier. So it allows you basically to do something once the internet connection is re-established.

### Service Worker Lifecycle Events

* There are also events which are related to the service worker lifecycle, so the installation and the activation, etc.
* You can hook into these lifecycle phases and execute some code, for example whilst the service worker is getting installed.

## The Service Worker Lifecycle



* From your normal JavaScript loaded by browser as part of html page load, you can register Server Worker which is just another JavaScript file (e.g. sw.js).
* The service worker JavaScript code which tells the browser, hey the content of the sw.js file is Javascript but don't execute it right now as you do with all the other Javascript files, instead please take it and register it as a background process, as a service worker.
* During that registration, two lifecycle phases are reached -
* First, the browser installs the service worker. During that it emits an **install** event which we can hook into to execute some code (e.g. to cache some assets) inside of the service worker whilst it is getting installed.
* There is another event which is executed as soon as installation finishes, once the service worker is activated. Now it's not necessarily activated right after the installation finishes. It's activated as soon as it can be activated which depends on the question whether there is an old version of the service worker running or not.
* Service workers will only get active if there is no old service worker instance running. So you might need to close existing tabs and re-open them to install a new version of the service worker.
* This is required because the service worker is not attached to a single page but to an overall domain or scope and since it lives on even if you close the tab, so you have to make sure that at least no tab is open to be able to replace it because then Chrome or whichever browser you use basically knows that it's now safe to switch it because no page is actively communicating with the service worker at least.
* So then once the browser decides that it can be activated, the **activate** event will be fired.
* Once the service worker was activated, it now controls all pages of a given scope. Scope basically means that you can decide which parts of your application are controlled by a service worker, put in other words, which pages of your application can work with the service worker. For example if we want to listen to fetch events, well if a page is outside of the scope of the service worker, the fetch listener in the service worker won't trigger up on fetched requests of the pages outside of the service worker scope.
* Once we have an activated service worker, it enters **idle** mode which means it's basically just sitting as it’s a background process handling events and if no events are coming in, it's doing nothing.
* After a time of idling around, it will terminate. This does not mean that it is uninstalled or unregistered, it just means it's kind of sleeping.
* You can wake it up though or actually it will wake up automatically as soon as events are coming in. So for example a fetch event and all the other applicable. If a fetch occurs, either due to an HTML file requesting a resource or you actively sending a fetch request in your Javascript code, the service worker is woken up and put back into idle mode and of course the fetch event listener is triggered there.

### Does browser installs the service worker whenever our application basically executes, so if we refresh the page, does it then install our service worker again?

* The answer is it depends. It always executes this registration code but it doesn't necessarily install the service worker.
* It will only do that if your service worker file (e.g. sw.js) change by 1 byte or more. So if it changed at all, it will install the updated version, if it's the same file, if it is unchanged, then it will not install it.
* It will still execute that registration code but in the registration step, it will basically make this check, recognize that the file might not have changed at all and then not trigger the install event here.
* So keep in mind that the browser will only install a service worker if it is an updated version or if it's for the first time ever that you have one, not on every page refresh.

## Registering Service Worker

### Scope of Service Worker

* The scope by default always is the folder the service worker sits in. So if we add it to the /js folder, the service worker will only apply to HTML pages inside that folder. (Here /js is a wrong place as it will not have any html pages, but will have only .js files.)
* We typically add the service worker file in the root folder so that it applied to all HTML pages in the application. E.g. /public.
* You can add another level of restriction using the navigator.serviceWorker.register() API by overriding the ‘scope’. (Note: If your server worker (e.g. sw.js) file is at some sub-folder (not at root level), then you cannot override this scope to the folders above that sub-folder.)

E.g. navigator.serviceWorker.register('/sw.js', {scope: '/'})

### Registering Service Worker

* We need to register service worker in our normal JavaScript code.
* You could add this code to the index.html file with some inline code between script tags, and then you would have to do it in every HTML page, because you want to register the service worker in every page of your application because you don't know which page the user is going to visit. Once you registered it, it won't be replaced easily or quickly unless you have a new version of the service worker but you need to be able to register it from anywhere because otherwise, you need to bet on the user visiting a specific page of your app.
* So a nice place to add the registration code would be some .js file (e.g. app.js) which is common and used in al html pages of your application.
* It's also important to understand that service workers and the manifest.json file are not related.
* Service workers only work on pages served via https. localhost is an exception to enable you or to make development easier for you. If you want to deploy this on a real server, you need to serve your application through https otherwise it will not work.

Why only https?   
Because service workers have a lot of power. You can intercept any request, so you want to make sure that this is served on a secure host and not on any host. You want to make sure it's encrypted and that this power isn't abused, hence this requirement.

## Reacting to incoming events

* Once we register a service worker, we can start listening to events specific to service workers. (In reality, a service worker is registered, when we receive the ‘install’ event.)
* We can react to the ‘install’ and ‘activate’ service worker events.

E.g.

// 'install' event when browser installs the service worker.

self.addEventListener('install', (event) => {

  console.log('[Service Worker] Installing Service Worker ...', event);

});

// 'activate' event when the installed service worker is activated.

self.addEventListener('activate', (event) => {

  console.log('[Service Worker] Activating Service Worker ...', event);

  /\*

    Below line basically ensures that the service workers are loaded or are activated correctly.

    It should work without that line but it can fail from time to time or behave strangely.

    Adding this line simply makes it more robust, might not be needed in the future.

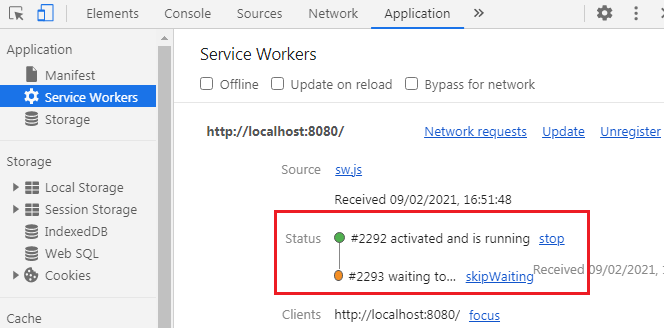
  \*/

  return self.clients.claim();

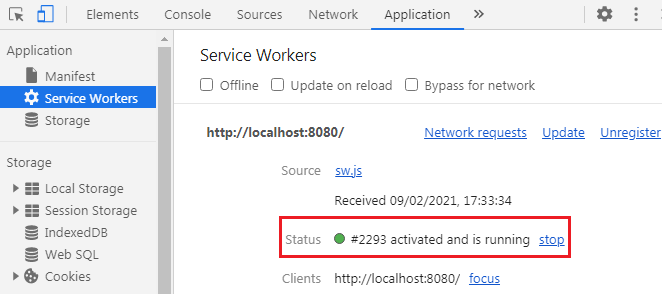
});

### Gotcha for ‘activate’ event

* If you have a tab open or if you have a window open with your page, then new service workers will get installed but not activated. (So you won’t see above logs corr. To ‘activate’ event).

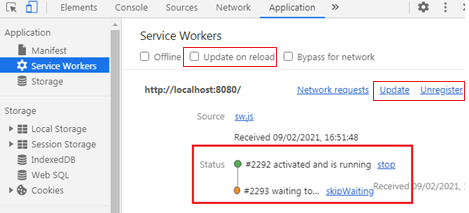


* The reason for this is that the page is maybe still communicating with the old service worker and activating a new one which might introduce breaking changes might break the running page.
* Therefore the way to activate a new version is to close the existing tab and reopen it so that you have all tabs closed. If you have multiple tabs of your application open, close all those tabs, re-open it and then you see the new version is activated and running and in the console.



* This is crucial to understand, if you change code in your service worker, you need to close all the existing tabs of your application and open a new one to load the newest version.
* Another important thing is, once you manually activated the service worker, all pages are under control of the service worker except for the page you are currently on. You need to reload this once for the service worker to control this too, simply because you loaded the page first before the service worker was activated, it doesn't know this page yet.

### Using DevTools



* With developer tools, there are shortcuts.
* You can check “Update on reload” and then reload will have hard overwrite existing service workers, so you don't need to close that page.
* You can also click “Unregister” and reload to fetch a new version or click “Update” link.
* Also you can click on “skipWaiting” link to immediately skip waiting and activate the new service worker.

### Fetch event

* It's important to understand that fetch is triggered by your web application, by the actual page whilst install and activate are triggered by the browser when the service worker is getting installed or was activated.

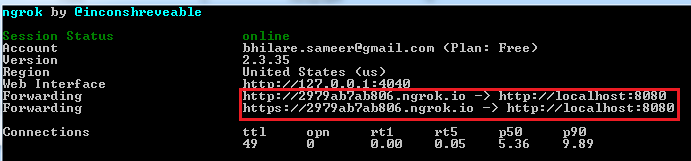
## Connecting Chrome Developer Tools to a Real/ Emulated Device

### Way 1

* Use ngrok to open HTTPS connection to your localhost.
* Login to <https://ngrok.com/>
* Download the zip file for your OS.
* Copy the extracted content into say C:\ngrok. (On unzipping you will only see ngrok.exe)
* Whichever application you want to test, make sure it is running. E.g. lets say we are running PWA app from vscode on port 8080
* Open command prompt and go to folder where you have kept the ngrok.exe and then run command ngrok http <port>

E.g. C:\ngrok>ngrok http 8080

* It will open both HTTP and HTTPS forwards which you can access from your phone directly. See below screenshot.



### Way 2

* Remote Debugging Android Devices  
  <https://developers.google.com/web/tools/chrome-devtools/remote-debugging/>
* Make sure you enabled "Developer Mode" on your Device! You do that by tapping your Android Build Number (in the Settings) 7 times.

### Way 3

* Connect your workstation and android device to the same WiFi.
* Get IP address of your workstation. Command Prompt -> ipconfig -> IPv4 Address.
* Use the same IP followed by port number to connect via browser on your android phone.
* E.g. If your IP is 192.168.43.130, then hit 192.168.43.130:8080 from your android chrome browser.

## Service Worker FAQ

### Is the Service Worker installed everytime I refresh the page?

No, whilst the browser does of course (naturally) execute the register() code everytime you refresh the page, it won't install the service worker if the service worker file hasn't changed. If it only changed by 1 byte though, it'll install it as a new service worker (but wait with the activation as explained).

### Can I unregister a Service Worker?

Yes, this is possible, the following code does the trick:

navigator.serviceWorker.getRegistrations().then(function(registrations) {

for(let registration of registrations) {

registration.unregister()

} })

### My app behaves strangely/ A new Service Worker isn't getting installed.

It probably gets installed but you still have some tab/ window with your app open (in one and the same browser). New service workers don't activate before all tabs/ windows with your app running in it are closed. Make sure to do that and then try again.

### Can I have multiple 'fetch' listeners in a service worker?

Yes, this is possible.

### Can I have multiple service workers on a page?

Yes, but only with different scopes. You can use a service worker for the /help "subdirectory" and one for the rest of your app. The more specific service worker (=> /help) overwrites the other one for its scope.

### Can Service Workers communicate with my Page/ the "normal" JavaScript code there?

Yes, that's possible using messages. Have a look at the following thread for more infos: <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#Sending_messages_to_and_from_a_dedicated_worker>

This is actually not Service Worker specific, it applies to all Web Workers.

### What's the difference between Web Workers and Service Workers?

Service Workers are a special type of Web Workers. Web Workers also run on a background thread, decoupled from the DOM. They don't keep on living after the page is closed though. The Service Worker on the other hand, keeps on running (depending on the operating system) and also is decoupled from an individual page.

# Promise and Fetch

## Promise

* Javascript is single threaded, it executes on a single thread.
* Promises are used to handle Asynchronous tasks in JavaScript.
* The promise object takes a function as an argument and that function itself receives two parameters resolve and reject (internal functions). Each promise has to either resolve or reject.
* The real value of promises shines once you have multiple asynchronous tasks chained after each other because then you can really just chain then() calls.
* Having then() calls chained after each other is a convenient and easy to understand way of handling values which you basically pass through a chain of tasks you could say.
* A promise must either resolve or reject.
* A resolved promise is handled via then() block. A rejected promise is handled via catch() block. (You can also handle rejected promise via then() block by adding a function in the second argument of then(). In this case then we don’t need catch() block. But this method is not common. Recommended to use catch() as it is much better for chain of promises.)
* The cool thing about catch() is it will catch any errors occurring at any step prior to catch.

### Creating Promises

* We can create our own promises but that is not too often, mostly we consume promises created by some APIs.
* Here is an example to **create promise**.

// creating promise

var promise = new Promise((resolve, reject) => {

  setTimeout(() => {

    if (true) { // make it false to test 'reject' flow

      resolve('equal');

    } else {

      reject('not equal');

    }

  }, 2000);

});

// consuming promise

promise

  .then((result) => {

    console.log(result);

  })

  .catch((err) => console.log(err));

### Deep dive into Proimise

* MDN: Introduction to Promises: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>
* Google: Introduction to Promises: <https://web.dev/promises/>

## Fetch

* A modern replacement for XMLHttpRequest.  
  fetch is a method provided by Javascript in your browser. It has decent browser support.
* fetch basically allows you to send HTTP requests.
* You might already know Ajax which behind the scenes uses the XmlHttpRequest object, fetch could be thought of as an alternative to that, though a bit nicer to work with.
* Fetch method returns a promise which we need to handle.
* response**.json()** is a utility method provided by the fetch API on the response object to extract the data and convert it into a Javascript object.

### Using fetch to GET

fetch('http://httpbin.org/ip')

  .then((response) => {

    console.log(response);

    // It is an asynchronous operation though because it gets a readable stream

    return response.json();

  })

  .then((data) => {

    console.log(data);

  })

  .catch((err) => {

    console.log(err);

  });

### Using fetch to POST

// using fetch to POST

fetch('http://httpbin.org/post', {

  method: 'POST',

  headers: {

    'Content-Type': 'application/json',

    Accept: 'application/json',

  },

  body: JSON.stringify({

    // since we are setting content type to json, we need to stringigy this JS object

    message: 'This is test message',

  }),

})

  .then((response) => {

    console.log(response);

    // It is an asynchronous operation though because it gets a readable stream

    return response.json();

  })

  .then((data) => {

    console.log(data);

  })

  .catch((err) => {

    console.log(err);

  });

### AJAX

* It's more complex and verbose - opening it, setting the response type specifically, setting up event listeners and then sending it.

// ===========================

// using AJAX

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://httpbin.org/ip');

xhr.responseType = 'json';

xhr.onload = function () {

  console.log('ajax', xhr.response);

};

xhr.onerror = function () {

  console.log('ajax error');

};

xhr.send();

### Fetch vs AJAX

* For service workers, you have to use fetch because service workers only allow you to use asynchronous ready operations like fetch. You can't use the traditional Ajax request in service workers.
* The traditional AJAX XmlHttpRequest works differently behind the scenes. It has a lot of synchronous code in there and therefore won't work in service workers which are all about asynchronous code.
* AJAX requests do not work with Service Workers.

### Deep dive into Fetch API

* An Overview on MDN:   
  <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>
* Detailed Usage Guide (MDN):   
  <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>
* Detailed Usage Guide (and comparison with XMLHttpRequest): <https://davidwalsh.name/fetch>
* Introduction to Fetch (Google): <https://developers.google.com/web/updates/2015/03/introduction-to-fetch>

## Adding Polyfills for Legacy Browsers

* fetch and promises do have good browser support, but they aren't supported by all browsers, especially not by older ones. Hence we need to polyfill those.
* If we use fetch and promises in our normal Javascript code, so not in the service worker which only works on modern browsers anyways but in the code which runs on every browser, we obviously get an error if we execute our app on an older browser.
* So we have two options, we either don't use fetch and promises there, that's possible to an extent but if we don't use fetch, we can't intercept these fetch events here in our service worker, so might not be the best option.
* The other option is to use polyfills, written by other authors, two very popular polyfills for both fetch and promises in the html pages where you intend to use. (promise.js and fetch.js).
* For these polyfills, first include promise.js and then fetch.js because latter depends on former.

# Tips and Tricks

* MDN Docs for PWAs  
  <https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps>
* MDN Web app manifests   
  <https://developer.mozilla.org/en-US/docs/Web/Manifest>
* Web App manifest explanation by Google.  
  <https://web.dev/add-manifest/>
* Starting local server via http-server package

> http-server -c-1

It will ensure that we don't cache any assets using the normal browser cache.

* To connect to your localhost from Android emulator, you need to use 10.0.2.2 as IP followed by port number.
* httpbin.org

This is a page which simply gives you some example rest API endpoints you can send requests to. You can post data to it, fetch data from it and so on.