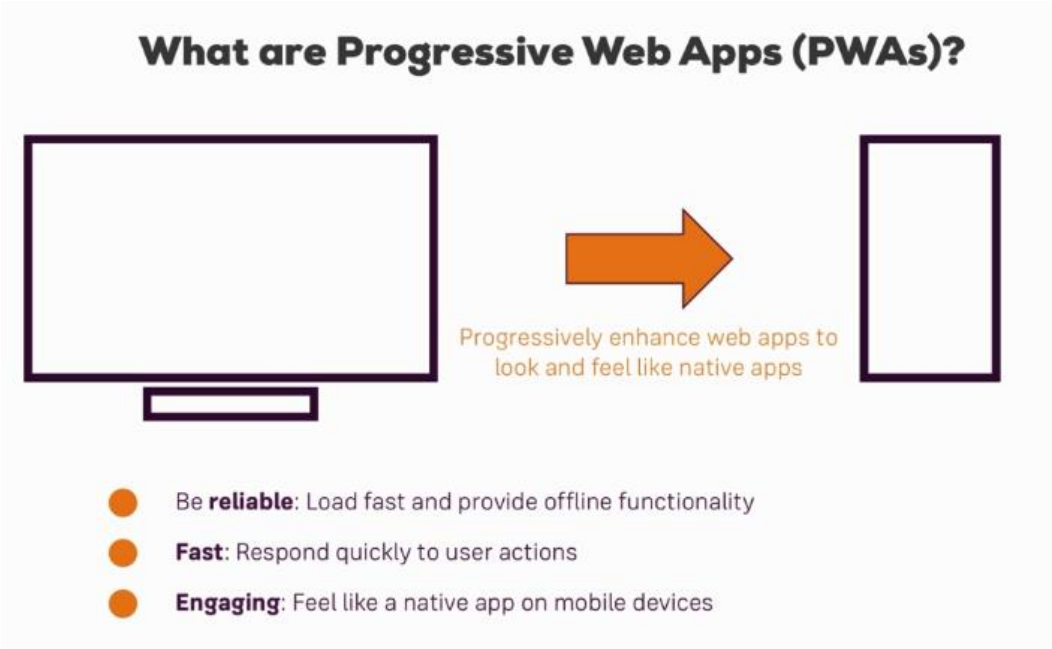
PWA
(Progressive Web Apps)

Contents

Getting Started

What are Progressive Web Apps?

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



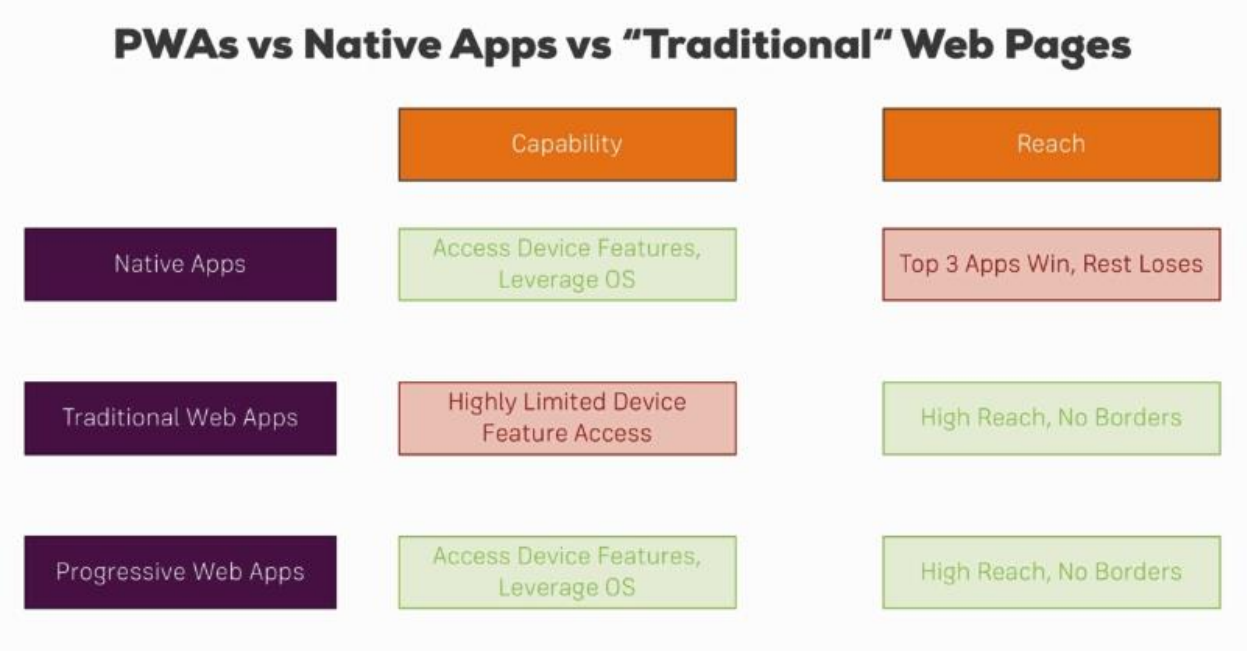
- Progressive web apps in the end describe a set of features of technologies you can add to your existing web applications to turn them into native apps like native mobile app like experiences using device features like the camera and providing offline access.
- You **progressively enhance** your existing 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

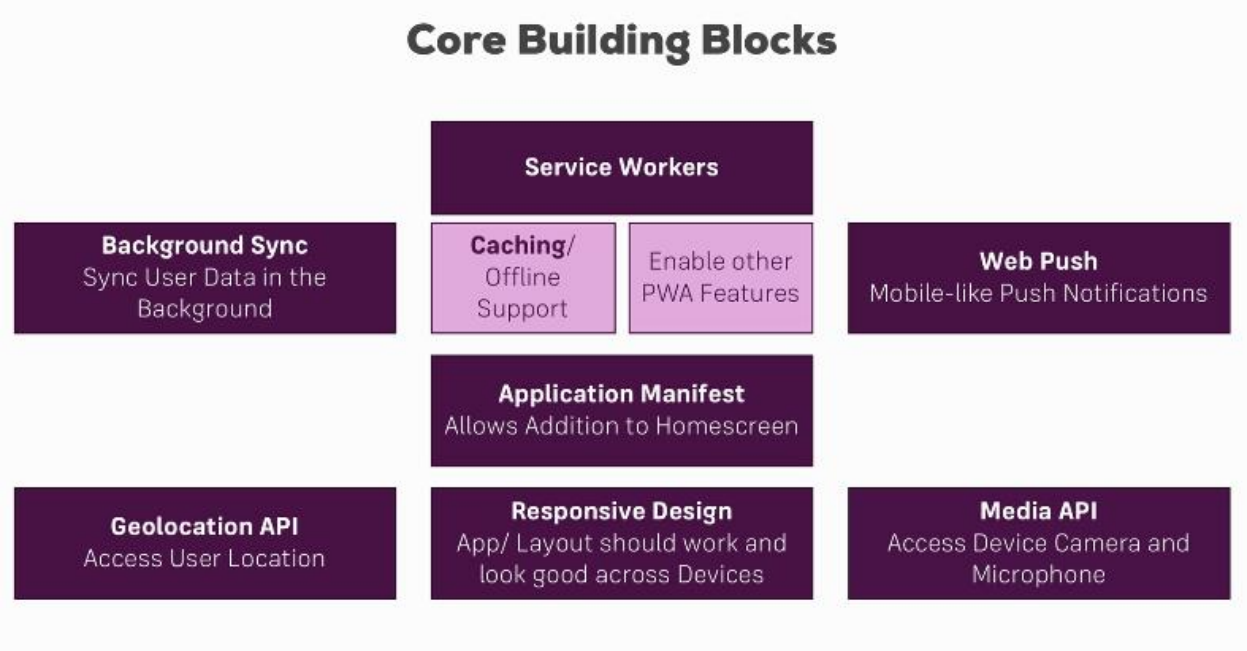
1. They should be **reliable** which means they should **load fast** and even work if you're **offline**, at least a part of the application should work if you're offline.

    a. 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.

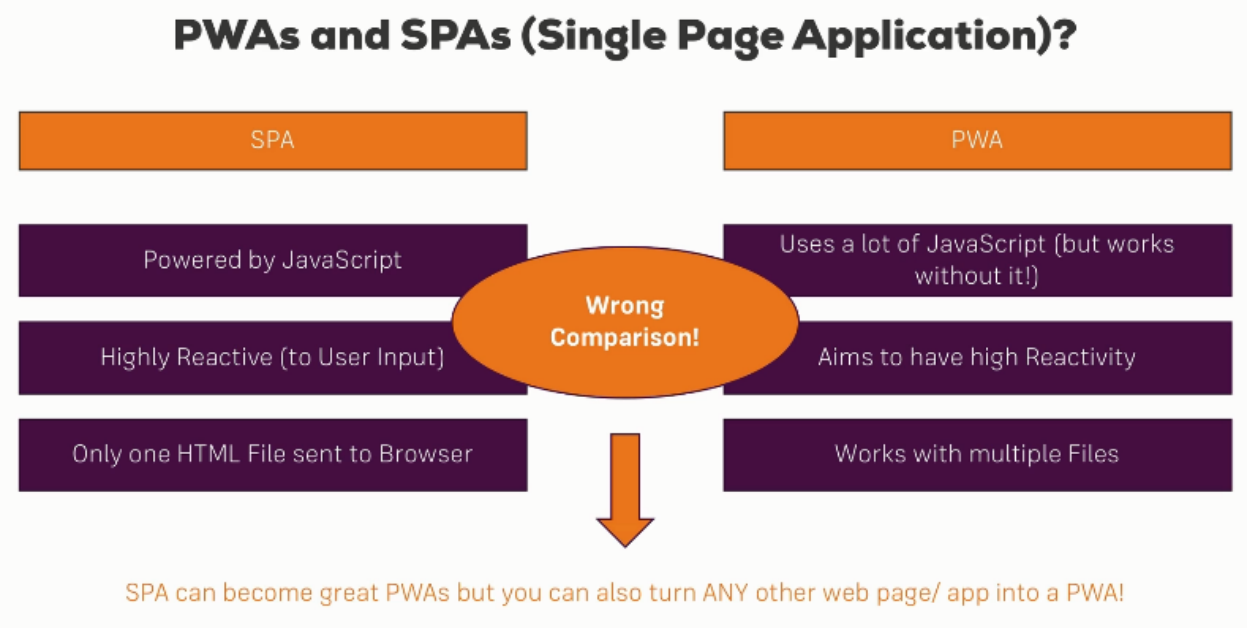
Geolocation API

- To access user's location.

Media API

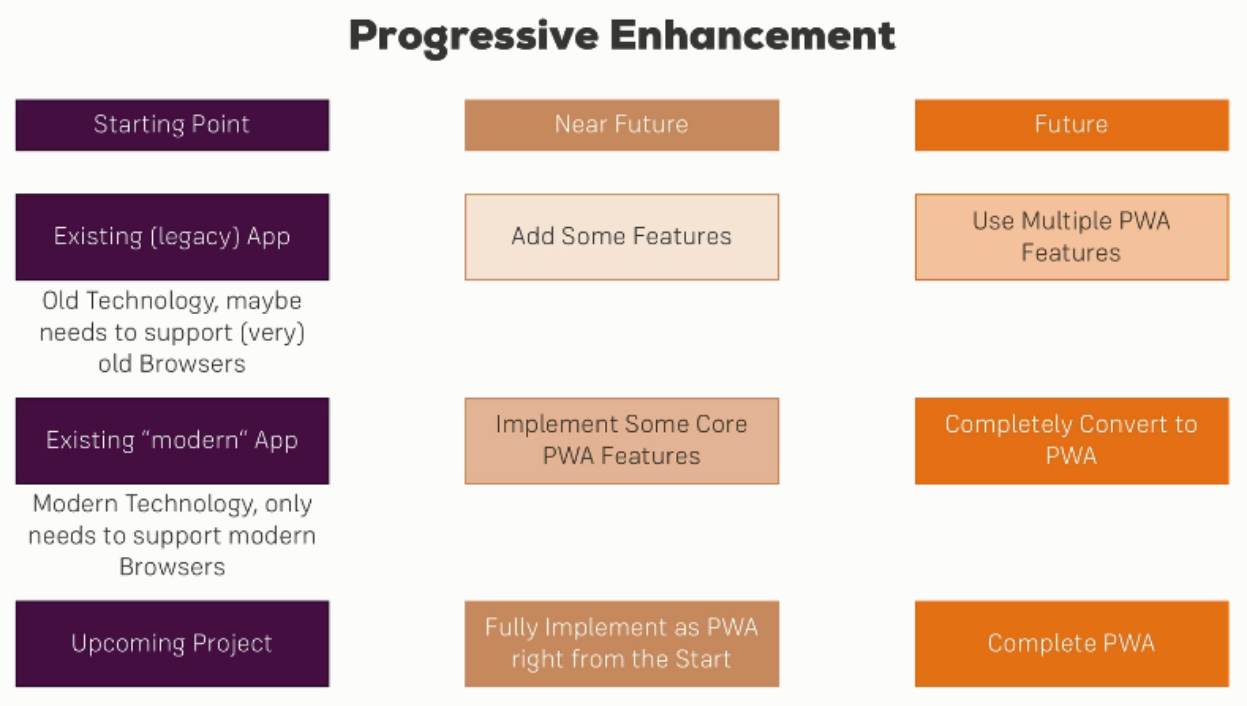
- To access device camera and microphone.

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 ho
me 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> ta
g 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 ho
me 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.

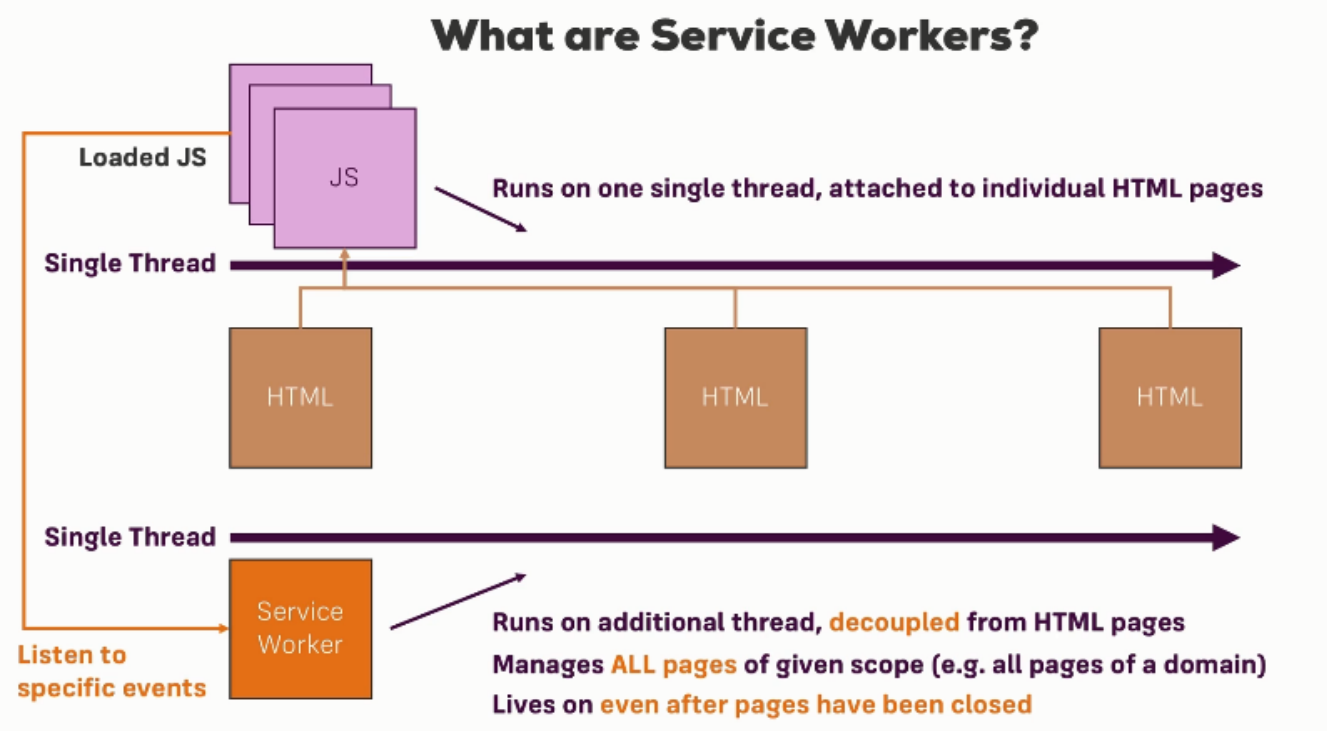
```
<!-
- This is same as theme_color property in the manifest.json file. Will be used to
 other browers -->
  <meta name="theme-color" content="#3f51b5">
```

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



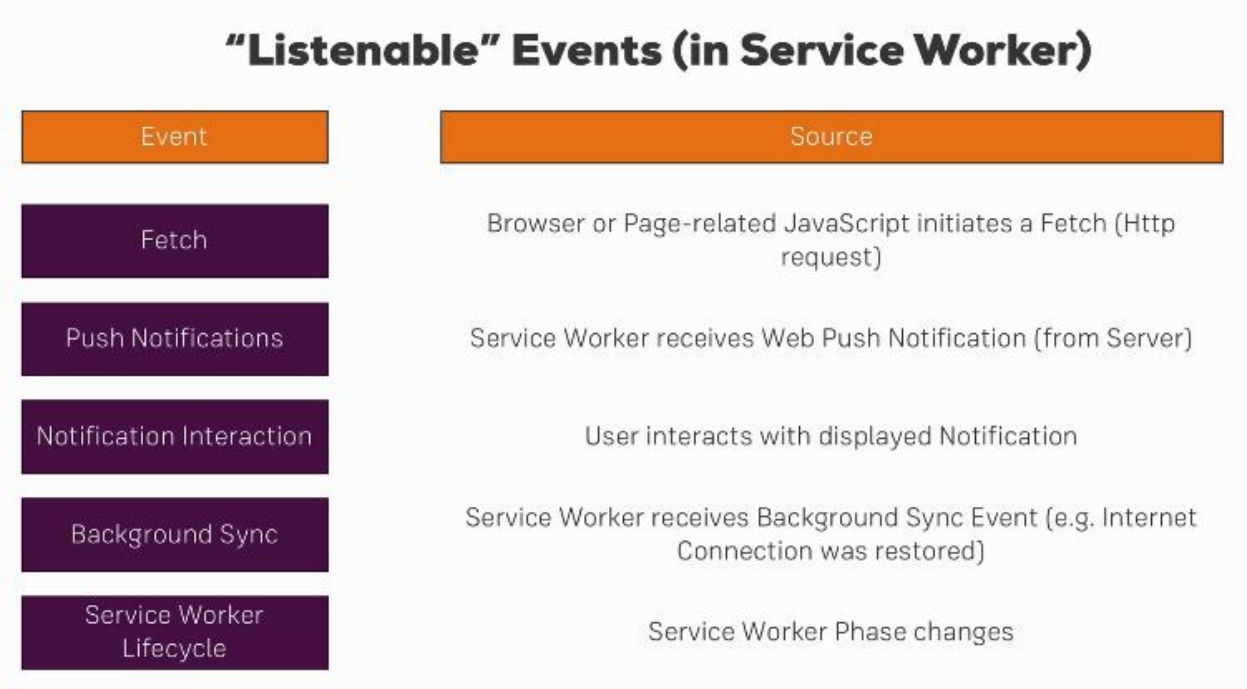
Background

- 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.

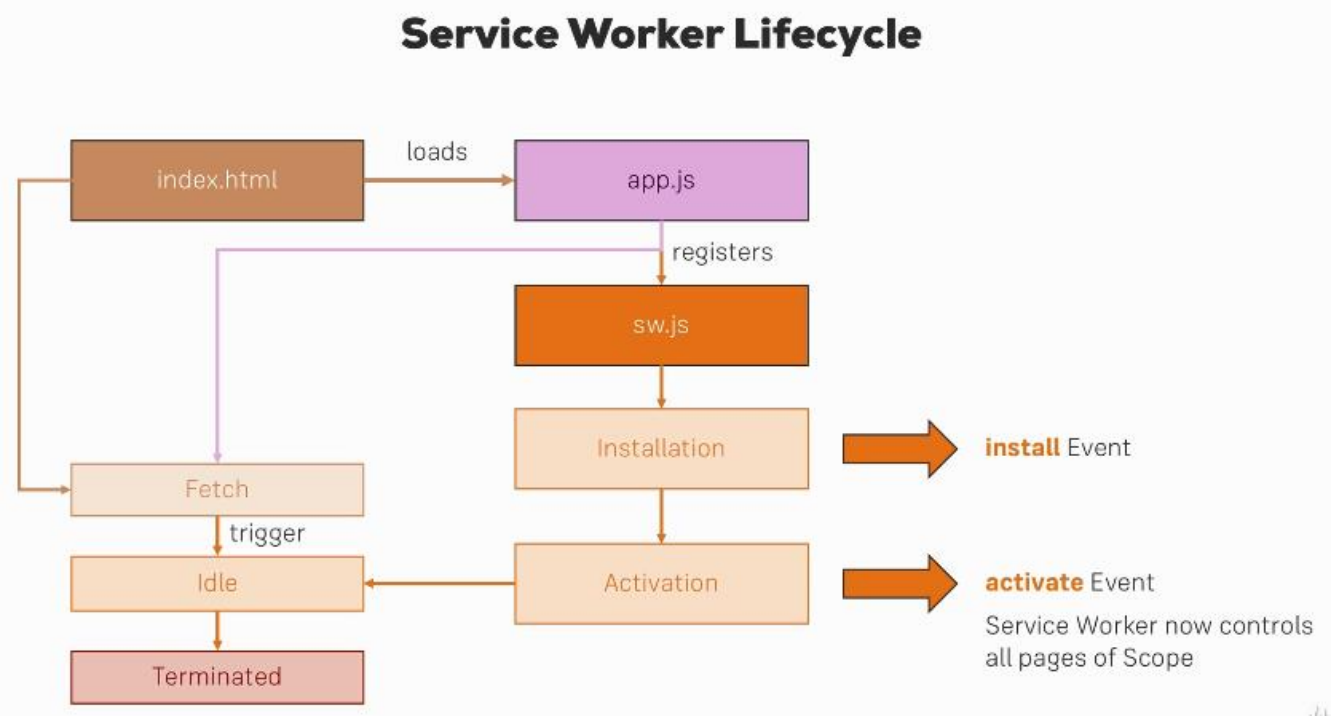
Background Synchronization Events

- 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.

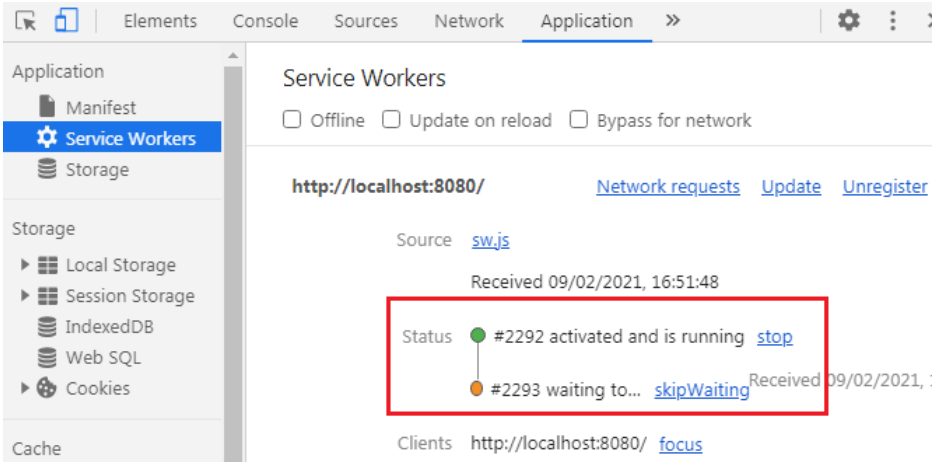
Reacting to incoming events

- 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.

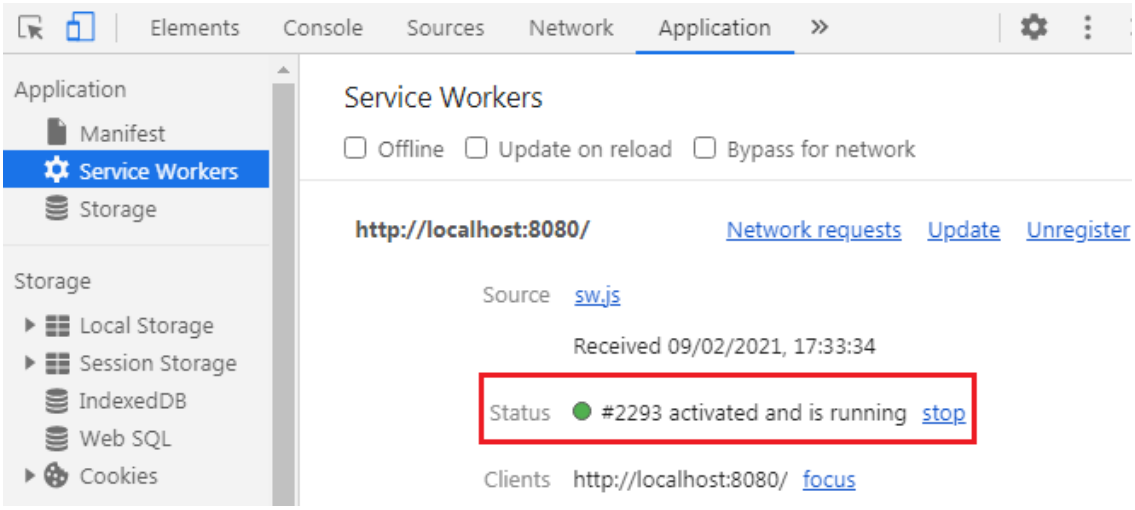
```
// 'install' event when browser installs the service worker.
self.addEventListener('install', (event) => {
  console.log('[Service Worker] Installing Service Worker ...', event);
});

// '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 ar
e 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 t
he 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).

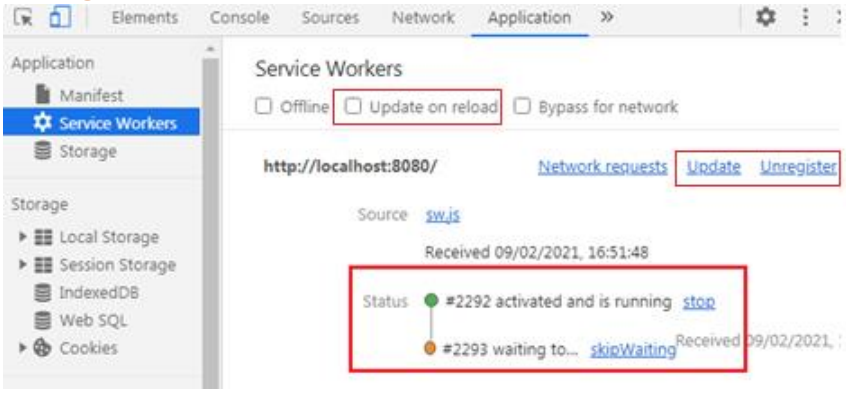


- The reason for this is that the page is maybe still communicating with the old service worker and activating a new one which might introduce breaking changes might break the running page.
- Therefore the way to activate a new version is to close the existing tab and reopen it so that you have all tabs 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.

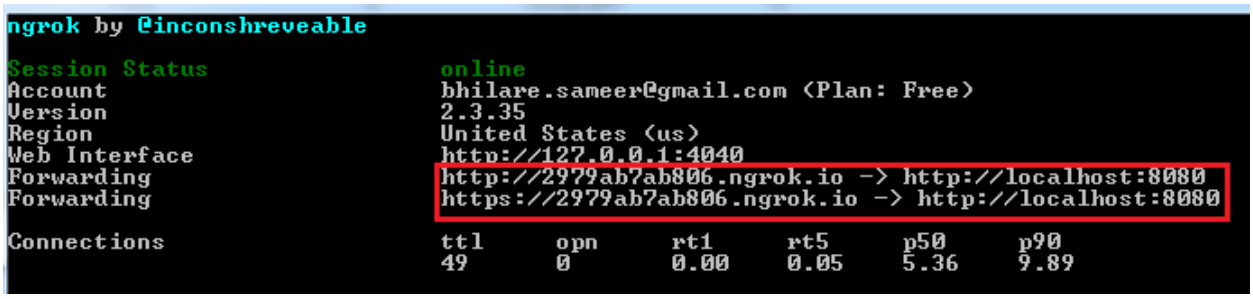
Fetch event

- 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 ob
ject
    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.

Service Workers – Caching

Why Caching?

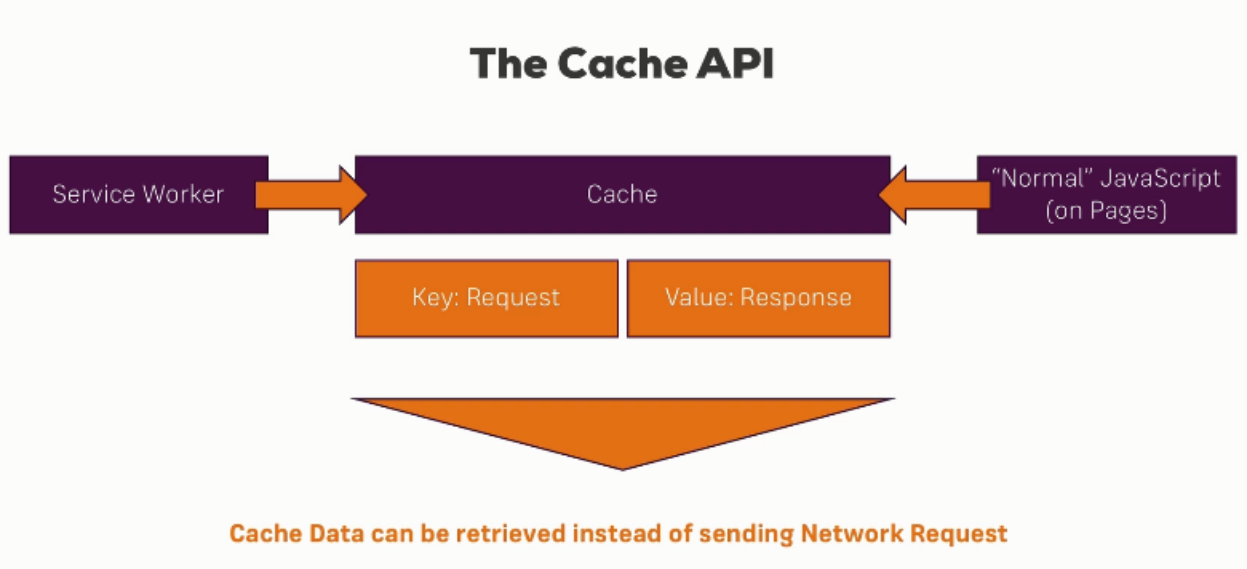
- The main purpose of using service workers is to provide offline support for our web application.
- There are a couple of use cases where you might have no internet connection and still want to interact with a web app. For instance –
    - Poor connection: Being in some crowded place or some remote place with poor connection.
    - No Connection: Being in elevator with no internet connection for couple of minutes.
    - Lie-Fie: so that's Wi-Fi which just is not, so that means you think you have a Wi-Fi connection, your phone displays that it has a Wi-Fi connection but actually its not.
- With service workers, you can pre-cache or in general, that you can cache certain files, assets your web app might need and therefore can still display the page and even allow navigation whilst your application is offline.

Understanding the Cache API

Background

- Cache is a general term used in different parts of the application. For instance –
    - Server side cache – this is of course disqualify for offline access as it is managed by the server and hence can't be accessed from client/browser.
    - Then the browser also manages its own cache totally without you telling it to do so. (You can disable this cache from browser DevTools. Network -> Disable Cache). Now this cache managed by the browser has one big disadvantage, it's managed by the browser, you can't rely on it, you can't explicitly tell it which assets to cache and which not to cache. So you as a developer have no control over it or only a very limited amount and you can't rely on these files being there when you need them. That's where the cache API steps in.
    - Cache API: A separate cache storage also living in the browser but managed by you as a developer.

Cache API

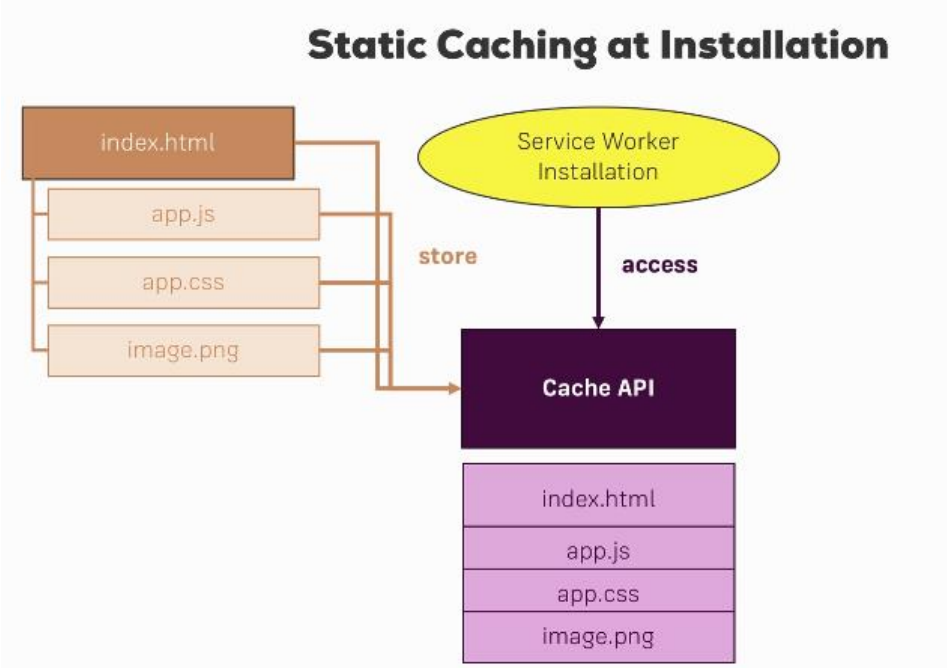


- This cache holds simple key-value pairs where the key are the HTTP requests you want to send and the value is the response you got back.
- Of course you need to have made that request successfully at least one time so that you got the response you actually want to display, otherwise you can't cache it. You can't cache what's not there but once you've got that response one time, you can store it on the key, so on that request you sent and then fetch it from this cache later on when you need to send that same request again but have no internet connection and that simply is how the cache API works – you store key-value pairs of request and response.
- The cache API can be accessed from both service workers and your ordinary Javascript on your pages so basically from your day-to-day Javascript running in your pages loaded through your HTML pages.
- Being able to access it from the service worker is powerful though because the service worker runs in the background. So normal Javascript only wouldn't do the trick because if you have no internet connection, the normal Javascript can't be loaded. There are certain use cases where it's still useful that you can access the cache from normal Javascript.
- So with service workers you get a chance of retrieving that data from cache instead of sending the network request when you have no internet connection.
- So in the end by using the **fetch event** in the service worker which is the key to intercepting any requests you want to send and the **cache API**, you have a complete **network proxy** living in your service worker which allows you to control if the request really is allowed to continue to the network or if you want to return a response from the cache if available.

Identifying (Pre-) Cacheable Items

- App shell / Application shell is the frame of our application, basically includes static pages, css, like a toolbar, a side bar, a footer maybe, some features which are really a core part of the app and which from a developer's perspective don't change that often. So everything around your dynamic content.
- A typical concept is to at least cache this app shell, so you have this frame and this basic app, maybe you have a text displaying nothing can be shown right now or something like that to give the user the feeling that your application is still working even if it's offline.
- This is how you work with service workers. You find out what your app shell is, what the core assets of your application are that are visible on most pages or are used on most pages, then you want to pre-cache that already whilst your service worker is installing and then you can think about what you want to cache on an on-demand basis i.e. to dynamically cache it. And as a last step you can think about dynamic content, so content which changes frequently, which you might want to cache.

Static Caching / Pre-caching

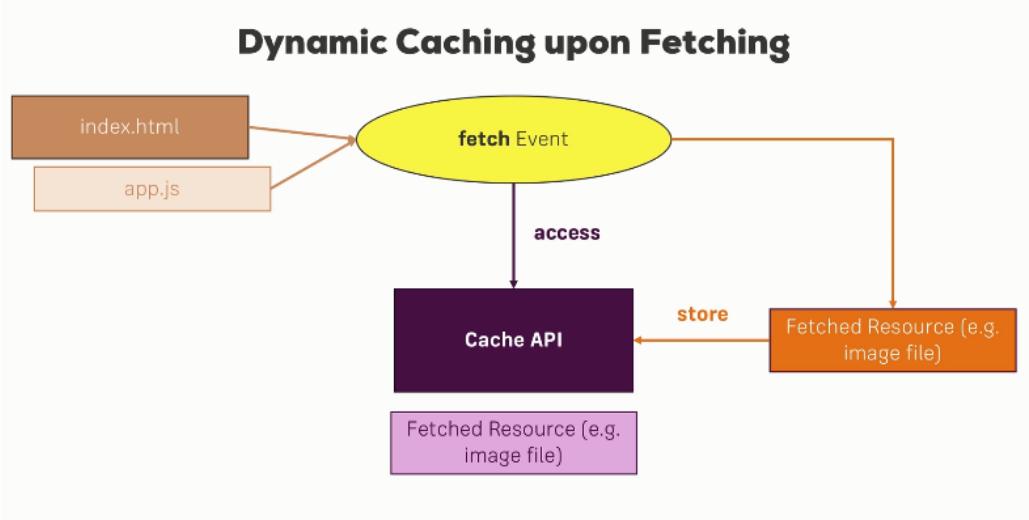


- Static Caching / Precaching is done during installation of the service worker.
- The service worker is only installed if it changed, so only if you release a new version of the service worker, otherwise it will not get replaced.
- Now for that reason, the installation phase of a service worker is a great place to cache some assets which don't change that often like your app shell, the toolbar, your basic

styling that might not change every second. Also your main scripts, these might change frequently but we would deploy a new version of your application anyways, so you can then update your cache in this case too but it doesn't change every second or it's not user created dynamic content.

- For this, we use Cache API. So we have our index.html page let's say, for a given route. We load a couple of assets there like our main script, our main styling and an image and we want to cache these files. Now we do that here during service worker installation where we get access to the cache API and can then store these files in the cache.
- **We should cache as much as needed but not everything because you don't want to overpopulate your cache. Reasons –**
    - The browser might clean it up if it runs out of space, if it gets too occupied and therefore you want to be conservative about what you put into the cache.
    - On the other hand, if you cache too much, there is the danger of loading content from the cache even though for the given resource, an up-to-date version from the network might be more appropriate.
- Cache.add() and addAll() are typically the APIs you want to use for pre-caching because we don't want to send the request manually.
- The cache is stored at Application -> Cache -> Cache Storage.
  You can have only one cache storage per given domain, though there you can open multiple sub-caches.

Dynamic Caching



- You can also add items dynamically to the cache. Maybe you don't want to pre-cache them because you don't want to bloat the cache right at the beginning during installation but you want to make sure that for example that if a user visited a page, he's able to visit again in the future.

Implementing Dynamic Caching

- If we want to cache dynamically, for that, we have to go to the fetch listener because dynamic caching means we have a fetch event anyways and we want to store the response which comes back in our cache.
- In case of dynamic caching, we store the fetched response in cache and also we need to pass this response back to the place where it was requested from (basically some action on the browser like navigating to that page). However the way responses work is you can only consume/use them once. So storing the response in the cache uses that response and you won't be able to use it again, so we should store the cloned version of response by using response.clone() and return actual response back to the browser.

Handling errors

- We should handle errors in case of access in offline mode to avoid logging those errors in the console.

Cache Versioning

- The static cache is filled with things we pre-cache. The dynamic cache is obviously populated with things we stored after we fetch them from the network.
- Caching JSON data is not really what you want to use the cache storage for. You should generally aim to store files there, not JSON files, simply images, CSS, JavaScript, HTML, that is what the cache storage is for.

Problems with Caching

- **First** we should not dynamically cache everything so every pages and everything. This might bloat the dynamic cache after some time.
- **Secondly**, which is also a crucial thing, if we change some code in our statically cached files (e.g. app.js in our project) AND if there is no change in our service worker file (sw.js in our project), then we will never get the most recent statically cached file (e.g. app.js).
- The **solution** for both these problems is to manage via cache versioning.
- With cache versioning, you include version number for your cache name.
- This will now ensure that we change the service worker file so it will be installed again and we also cache all the updated files or all the files actually into a new sub-cache. And by doing this, we ensure that we always update our cache if we update a file.
- With versioning, of course we add new caches but can clean old unused caches too.

Why do we use a new cache and not use the old one?

- Well, there are two advantages.
- First since we add version related logic in service worker file, this new file will get installed.
- Besides the fact that we need to change the service worker file, if we store it in the old cache, somehow the old cache might still be used by the page because keep in mind, the installation step executes immediately but we're waiting for activation not for installation and the activation will happen only after the use closes all the tabs of this application. So if we overwrite or change something in the old cache, we might change the cache which is currently getting used by the web application and we don't want to do that, we don't want to break the application because we mess around with the cache, hence we store it in a new one.

Cache cleanup

- We should cleanup old (versioned) caches because otherwise not only we do crowd our cache storage because we add more and more caches but also our application fetches outdated versions from cache.
- **Where do you clean up the old cache?**
- The first inclination might be to do that in the install step right after cache addAll, but this is not correct because we don't want to mess with the old cache because it's still getting used as new service worker is not yet activated. So we certainly shouldn't clean up the old cache in the 'install' event because that might break our application.
- A better place to do cleanup work is in the **activate event** because this will only be executed once the user closed all pages, all tabs and open the application in a new one. So now it's safe to update the cache because now we're not in a running application anymore.
- Always load the latest asset versions and we don't overpopulate our cache storage by removing unused caches.

Useful Links:

- About Cache Persistence and Storage Limits: https://jakearchibald.com/2014/offline-cookbook/#cache-persistence
- Learn more about Service Workers: https://developer.mozilla.org/en/docs/Web/API/Service_Worker_API
- Google's Introduction to Service Workers: https://developers.google.com/web/fundamentals/getting-started/primers/service-workers
- Cache (MDN) https://developer.mozilla.org/en-US/docs/Web/API/Cache

Service Workers – Advanced Caching

Offering "Cache on Demand"

- A typical use case for that is an article on a news site which we want to save to access it later, maybe even offline.
- We can access the cache storage (so the "caches" object) from our normal Javascript files as well.
- Cache on Demand is mostly useful if you have a resource which the user should request to be cached, like an article he wants to store for reading it later.

Providing offline fallback page

- In case of offline access, we can have a default fallback page that we could present for pages we haven't cached yet.
- So with this, we can present the fallback page with our own styling and text something like "Hey the page is not available yet as you are offline" to enhance user experience.

Steps

- Create the fallback html page (e.g. offline.html) with your app styling (mostly based on your app shell). You can texts and links to other already cached pages to allow the user to navigate to already cached pages.
- Now the goal is to load this page (offline.html page whenever the user is offline and visits some page we haven't pre-cached or cached at all.
- Pre-cache (static cache) this fallback page in the Service Worker's 'install' event step so that we can use is later.
- Now in the service worker's 'fetch' event where we are making the actual network request (e.g. fetch(event.request)), add the catch() block for that fetch() promise. And from that cache block, access the static cache and return this fallback page (offline.html). This is because in case of no internet access, the actual fetch() network request will fail and the catch() block will be executed. So we must return the offline fallback page from this catch block.
- So with this fallback page, we get a better user experience because whilst the page is not available, we can show something meaningful to the user with links to other already cached pages.
- You can extend this feature to other parts like if you are expecting an image and as a fallback you could use a dummy image.
- So you can return more than just fallback pages (.html), you can return any fallback file and the image example is pretty good, if the image is missing, maybe an avatar or some dummy image does it for now.

Caching Strategies

Strategy: Cache with Network Fallback

- We first try to access the cache and then if that fails or if we don't find the item in the cache, we fallback to the network by making the fetch() network call.
  - This is how this strategy works –
    - Our page sends a fetch request. The service worker intercepts any request the page sends and then the service worker has a look at the cache.
    - Now if a resource is found, it is directly returned, if it is not found, we execute another step where we reach out to the network and the network then returns the response.
    - Now what we can also do is we step in with the service worker and put that response onto the cache again to have "dynamic" caching and make sure that for future requests, we can find the resource in the cache but even without that, this is what the strategy is called, cache and then network fallback.
- The **good thing** about this strategy is that we can instantly load assets if we have them in the cache, which is even good if we do have Internet access because it's very fast.
- The **bad thing** of course is especially when use it generically, that we parse everything and for some resources, especially resources which should be highly up-to-date, this is bad because we might return old versions which are still in the cache because we don't reach out to the network by default, we only reach out to network if an element is not in the cache.

Strategy: Cache Only

- Our page sends a fetch request. The service worker intercepts the request. We then have a look at the cache and if we find a resource there, we return it to the page. We totally ignore the network.
- Cache only strategy has huge problems. In case of offline mode, only pre-cached (static cached) items can be shown. It will fail for all non-pre-cached items.
- We can parse the incoming request and use different strategies for different resources/request.
- In general, cache only strategy is not useful. But it makes sense for assets in the App Shell because whenever we make changes in these files, we anyway have to update cache version of service worker so that latest service worker is installed. So we always have the latest version of these files in the cache. Hence it makes sense to load all these files directly from the cache instead of trying to get them from the network if the cache fails.
- Minor edge case: If the cache fails for some other reason, then these files not being present, we kind of break our app unnecessarily, though that shouldn't really happen.

Strategy: Network Only

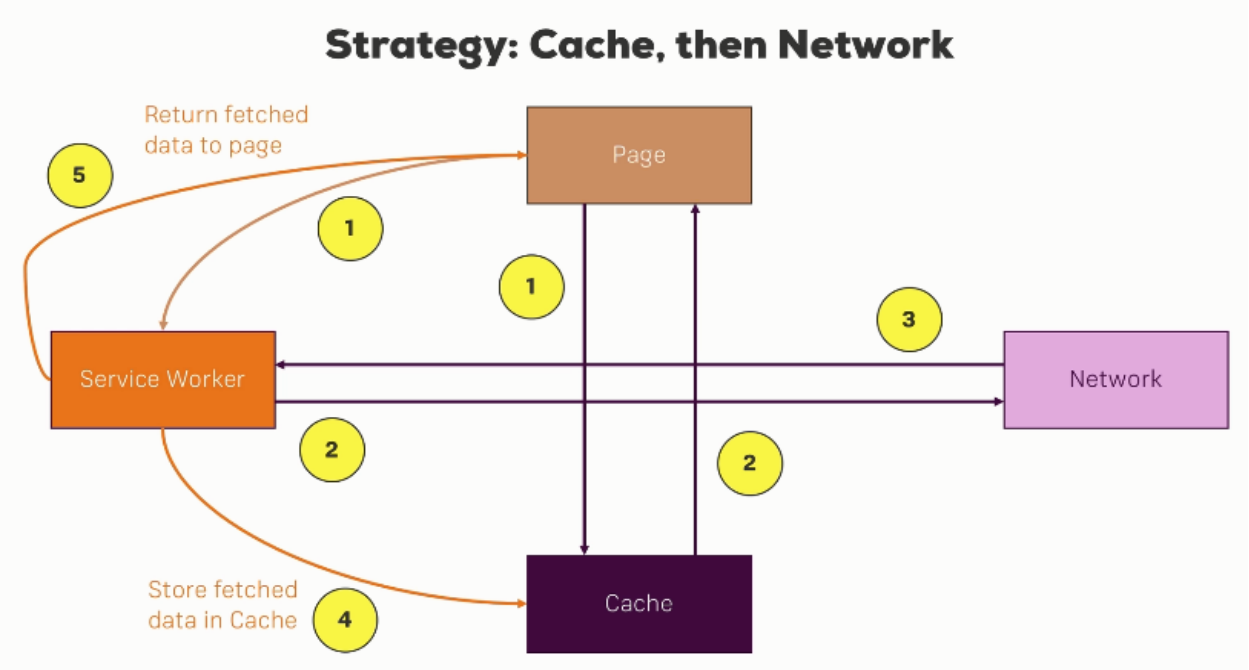
- The opposite of cache only is the network only. There we don't use the service worker at all, instead the page sends a request to the network and we return that.
- This could make sense for some resources which you can split up when you do parse the incoming request to funnel some through to just return the network result but standalone, this doesn't make much sense.
- We can parse the incoming request and use different strategies for different resources/request.

Strategy: Network with Cache Fallback

- We intercept a request in a service worker and then we try to reach out to the network and only if that fetch fails, we reach out to the cache and return the asset from the cache, otherwise if the network access would have been successful, we would have returned the network response and ignored the cache.
- Here, we take the best from both worlds. We have assets stored in the cache but we only use it if we have no network access.
- Now one downside is that we don't take advantage of the faster response with the cache which is also good even if we have Internet connection but the major problem is the way that the network behaves. If a fetch fails, it doesn't happen instantly. Imagine a very poor connection where the request times out after maybe 60 seconds, then you would have to wait for the full 60 seconds until you reach out to the cache as a backup, that's a horrible user experience of course.
- Using this strategy is not that common. You might find some cases where you can wait that long, maybe some background fetching you are doing but in general if it's critical to present something to the user quickly, don't use this strategy. Use it for assets which you can fetch in the background which don't need to be there immediately.

Strategy: Cache then Network Also

- This strategy is really useful in a lot of cases.
- The idea is to get an asset as quickly as possible from the cache and then also try to fetch a more up-to-date version from the network.
- This way we can always present something to the user quickly and still get an updated version if the network is available.
- It's the improved version of the network first fallback to cache version because here we don't need to wait for the network to timeout. If it does timeout, doesn't matter, we already served something from the cache. If it is successful, we override whatever we fetched from the cache.
- This is how it works –



  - We directly access the cache from the page and we get the value back. No service worker involved here.
  - We also use some variable to see if a network response is already there because in case the network is faster than us retrieving data from the cache, we don't want to override the network response with the cache response.
  - We also send a network request which is intercepted by the service worker and this happens simultaneously to us reaching out to the cache actually.
  - The service worker will then reach out to the network and try to get a response from there.
  - Now this network response goes back to the service worker, there we then also store that in the cache. Storing the response in cache is optional here if you are not using dynamic caching.
  - Then we also return the fetch data to the page.
- It's a powerful strategy because it allows us to get something on the screen quickly but update it whenever a new version is available.

Cache then Network (also) without Dynamic Caching

- Step 4 in above diagram is NOT used. That is, we're not storing the fetched data in the cache (dynamic cache).

Cache then Network (also) & Dynamic Caching

- Step 4 in above diagram is used. Here we do store the fetched data in the cache (dynamic cache).
- With this in place, we're making sure that we do reach out to the cache first (in the feed.js). If the item is there, we display it immediately. We also make a network request SIMULTANEOUSLY (in the feed.js). Once the response is back from the network, if it's a valid response, we store it in the cache here in service worker's 'fetch' event. If it's not, we don't do anything with it. But then we still have something served from the cache (code in feed.js). If we don't have it in the cache and we can't get it from the network, well there's nothing we can do.
- Problem – If we go offline, we fail ('site can't be reached'). This is because we never have any code in service worker fetch event where we try to fetch something from the cache.
- This strategy is really good for use cases where we do have Internet access and we want to get something on the screen quickly because we fetch it from the cache first. It's bad for cases where we don't have internet access though because we never try to fetch it only from the cache.

Cache then Network (also) & Offline Support

- Here we parse the request url to use different strategy for different urls.
- Use "Cache then Network" strategy for the urls which are initiated from normal javascript via "Cache then Network" strategy. And for other requests, use our old strategy of Cache with network fallback.
- Here the "Cache with network fallback" strategy will ensure the offline access by getting the data from cache.

Cache Strategies and Routing

- Being able to look into the request URL and deciding what the best strategy is.

POST request and Cache API

- Theoretically you can't store post requests in the cache, it can store the response of the post request though, but it's not storing the post request itself.
- So if you go offline and hit the POST request, it will fail as you can see in the network tab, but if its response if already cached, then application will work properly.

Cleaning/Trimming the Cache

- (With Cache with Network Fallback and Dynamic Caching) The more pages your app has the more you're going to load into the dynamic cache. So sometimes, you maybe want to clean up your dynamic cache because we can't let the cache grow indefinitely as we have limited memory for cache on the browser.

- We can write logic to limit number of items in a cache and write helper function to trim the cache items to the max allowed number of items.
- You can basically call this trim function wherever you feel it's appropriate. E.g. In service worker fetch event, before storing any server fetched request into cache. Or from server worker's install or activate events as well.
- Feel free to make this algorithm far more complex.
- It's important to be aware of the possibility of trimming or managing your cache. So you can control how it grows, how it behaves.
- So keep in mind, since you can access caches from your normal Javascript code too, you could also implement some logic there, not just in the service worker.

Getting Rid of Service Worker

- Like the way we can register a Service Worker, we can also get rid of a Service Worker.
- As soon as you unregister a service worker and if you go offline, the cache storage is also cleared for you.

```
/*
  Unregister Service Worker.
  You can call this function from appropriate place.
*/
function unregisterServiceWorker() {
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.getRegistrations().then((registrations) => {
      for (var i = 0; i < registrations.length; i++) {
        registrations[i].unregister();
      }
    });
  }
}
```

Useful Links

- Great overview over Strategies - the Offline Cookbook: https://jakearchibald.com/2014/offline-cookbook/
- Advanced Caching Guide: https://afasterweb.com/2017/01/31/upgrading-your-service-worker-cache/
- Mozilla Strategy Cookbook: https://serviceworke.rs/strategy-cache-and-update_service-worker_doc.html

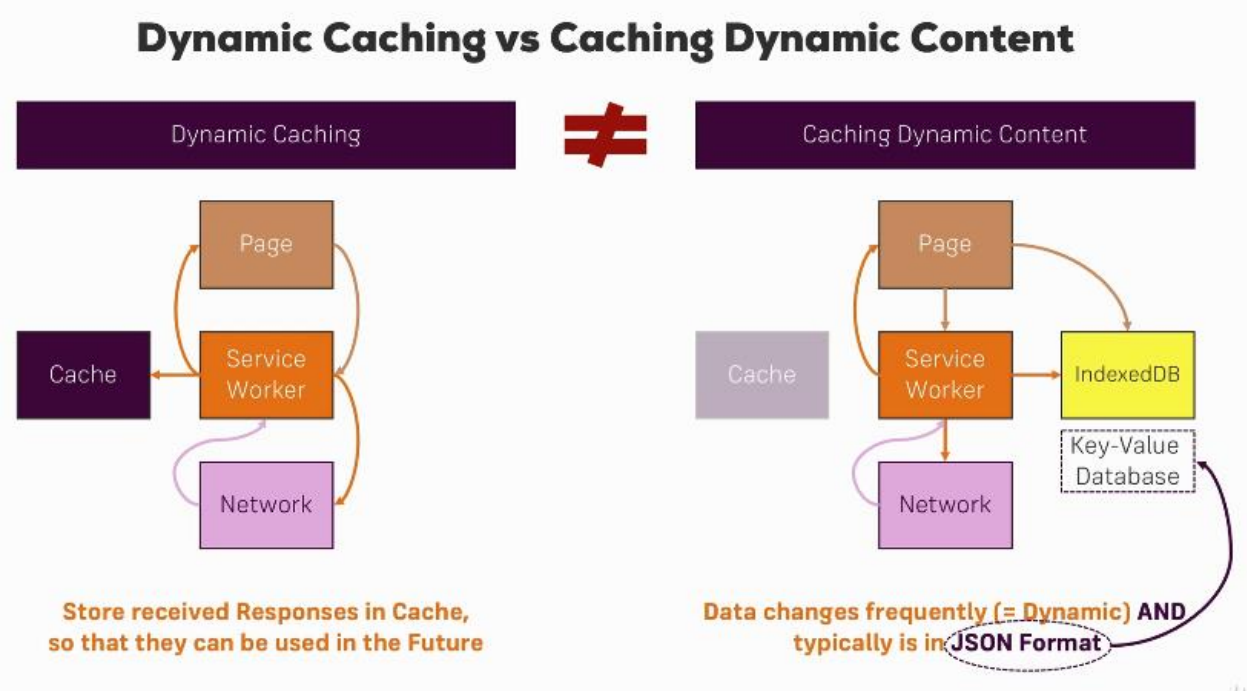
IndexedDB and Dynamic Data

- The cache API is awesome for caching assets like script files, CSS files or images.
- But what about dynamic data, so the data we get from a server. We can also cache this but it's best to not use the cache API for that but IndexedDB.

Setting up Firebase as Backend

- We can use Firebase as a dummy/ easy-to-use backend.
- Firebase is a managed server solution by Google.
- On Firebase, we can basically build a real back-end but without creating the whole NodeJS server from scratch.
- So with Firebase, we can easily set up a back-end which offers us a database, file storage and even the possibility to run some code and build our own restful endpoints. And under the hood, it uses NodeJS.

Dynamic Caching vs. Caching Dynamic Content



Dynamic Caching

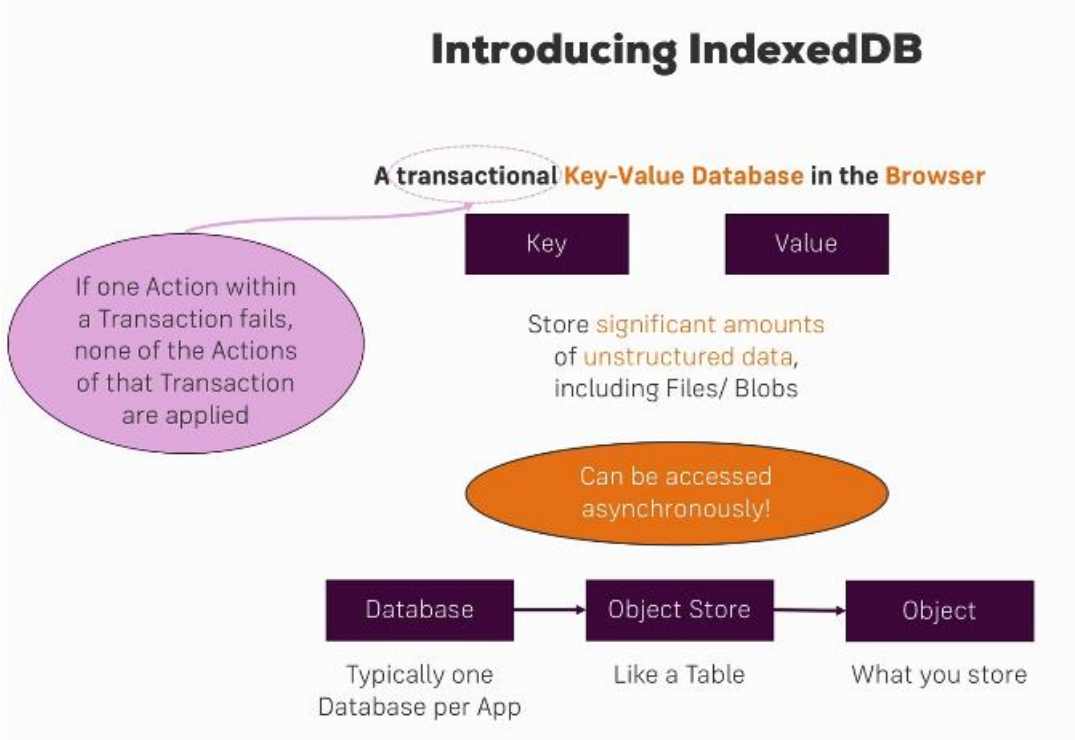
- Dynamic Caching simply means if we have our page and we send a network request which is intercepted by the service worker, therefore the service worker reaches out to the network and we then get the response back and store this in the cache and return it to the page like in our cache then network strategy.

- We make sure that we store assets, resources the user requests in the cache once we got them.
- We use that not only in the cache then network strategy but in other strategies too.
- Basically dynamic caching always happens whenever you not just pre-cache files during the installation but you add files, resources to the cache once you've got a response during a fetch request.
- This is dynamic caching, it just means you dynamically add items to your cache, that's all.

Caching Dynamic Content

- Here we still you the same pattern. So we have a page and we send a network request which gets intercepted by the service worker but we don't use the cache API, instead we use a new tool, indexedDB which is an in-browser database, a key-value database.
- It runs in the browser and it's there to help you store data as all other databases.
- The difference to the cache API is that indexedDB is used differently and that it's not so much about storing CSS or script files or images in there but instead storing JSON data, XML data, so basically the structured or unstructured data you're getting back from your back-end, which defines the meat of your application so the content and not the look and feel.
- So basically this still is a cache then network strategy, just that our cache is no longer the application cache we used with the cache API but instead indexedDB.
- The Dynamic Content means the data that changes frequently. That data is typically in JSON format, though can also have XML. And for that, the indexedDB just happens to be really good because it's a key-value database, we can store JSON data there and we can have more power over it than we have in the normal cache API.
- In **cache API**, we can just store the response and we have to parse the response, we never really use it, it's the whole HTTP response. In **indexedDB**, we can already store the formatted or transformed response or whatever we want, we can store pieces of the response, we have the full power over what we want to store, which part of the response we want to store.
- IndexedDB is a great choice for dynamic content typically in a JSON format, we can transform and we have more control over it. The cache API is possible to be used and it's not necessarily wrong or bad but you can definitely use indexedDB for that and it might hold some significant advantages.
- The approach (Caching Dynamic Content) is the same (as Dynamic Caching) that is cache then network but the nature and format of data is different and therefore we might want to store it differently.
- Of course, we can use indexedDB not only with the cache then network approach, you can basically use it whenever you use the cache API, just for JSON data.

Introducing IndexedDB



- It's a transactional key-value database running in the browser.
- Transactional simply means that if one of the actions in a given transaction fails, none of the actions are applied to keep database integrity.
- This is because you want to ensure that the database doesn't enter some invalid state thereafter so therefore you have this transactional approach, every operation of a given transaction has to succeed for the whole transaction to succeed.
- You can store significant amounts of unstructured data including files and blobs. So e.g. If you have some data which is like a title, description and a content of a text field but you also have an image of the post or course or whatever you're storing in indexedDB, you can store that file directly in the database. And unstructured data just means you don't have to define a schema upfront, you can store whichever data in whichever structure you want to store.
- Due to this key-value nature, it's basically like a Javascript object. You can have nested objects, nested properties, you can have arrays in there, you can store all of that in indexedDB.
- IndexedDB can also be accessed asynchronously. Now this is important because whilst you can access it synchronously too, due to it being accessible asynchronously, you can use it in a service worker because remember, service workers are all about listening events, they have this asynchronous nature, therefore it's synchronous only code can't be run in service workers.

- You can use it from both your normal Javascript code which is loaded through your pages as well as through service workers.
- That as a side note is also the difference to local storage. In the browser there is local storage and session storage, now both of these are accessed synchronously and therefore can't be used in service workers, that's the reason why you typically use indexedDB.
- Inside an indexedDB, you can actually have multiple databases but typically, you have one per application. In a given database though, you again have multiple object stores which s is something like a table. And since your data can be unstructured and doesn't have to follow a schema, you can basically put whichever data you want into a database and into an object store.

Using IndexDB

- You can use indexedDB just like that. It's a normal browser feature and therefore you can access it directly in Javascript.
- The normal indexedDB API is a bit clunky and it also uses a lot of callbacks. It's better if we use a special package wrapping indexedDB. This is optional and the general way you use indexedDB won't change but it gives us a nicer way of using it. The npm package name is 'idb'. This 'idb' package simply wraps indexedDB and allows you to use promises.
- Third party package – https://github.com/jakearchibald/idb#readme
- To Access IndexedDB in browser, Open DevTools -> Application tab -> See 'IndexedDB' in left pane under 'Storage'.

Handling Deleted Data in Firebase

- Any inserts on updates in Firebase will reflect properly in browser. But what is data is actually deleted in the Firebase (server). In this case, we would still have the deleted data in browser's indexedDB. How should be delete that?
- One easy and good way to handle this is to simply go to the service worker and whenever we write new data into our indexedDB, before we do so, we could clear the entire database because we're fetching all the data we want to store anyways. So we could clear it and then we're writing all the new entries anyway, so we would have a clear database and re-populate it every time we get back data.

Useful Resources:

- IndexedDB Browser Support: http://caniuse.com/#feat=indexeddb
- IDB on Github: https://github.com/jakearchibald/idb
- IndexedDB explained on MDN: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
- Alternative to IDB: http://dexie.org/

Creating Response User Interface

- For PWA, we should follow mobile first approach of UI design.
- Consider adding animations to have a feel of native mobile apps.

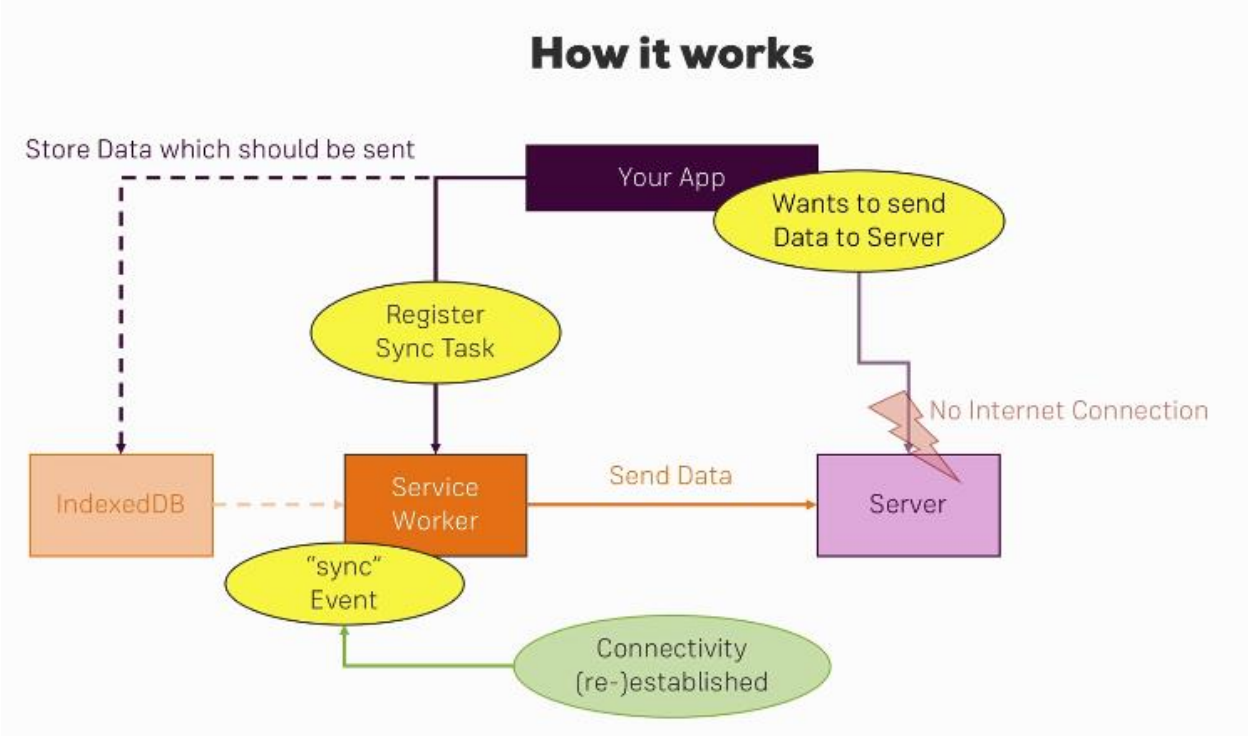
Useful Resources

- Responsive Design Basics by Google: https://developers.google.com/web/fundamentals/design-and-ui/responsive/
- Responsive Design Patterns (Google): https://developers.google.com/web/fundamentals/design-and-ui/responsive/patterns
- Responsive Images (Google): https://developers.google.com/web/fundamentals/design-and-ui/responsive/images
- Using CSS Media Queries: https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries
- Responsive Images (MDN): https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images
- Responsive Images in CSS: https://css-tricks.com/responsive-images-css/
- Using CSS Animations: http://learn.shayhowe.com/advanced-html-css/transitions-animations/

Background Sync

- With background sync, we'll be able to store requests we want to send in our application if we are offline and send them once connectivity is re-established.

How does Background Sync work



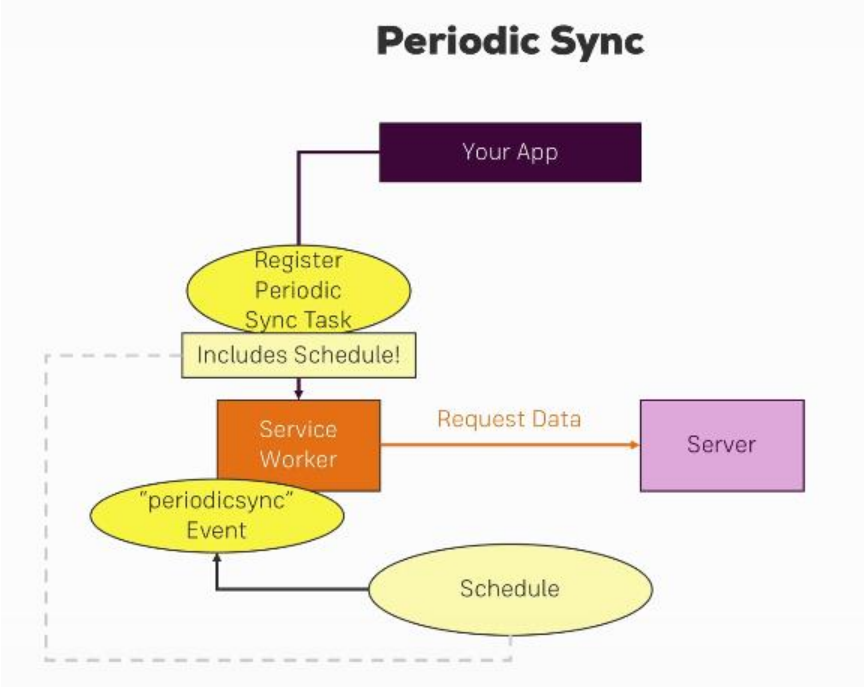
- Background synchronization is all about sending data to a server when we have no internet connection.
- We can use a service worker to register a sync task. So basically tell the service worker that I want to do eventually send data, please be aware of that and register it as a synchronization task.
- We also need to store the data we want to send with the request, with the post or put request or whatever it is and as this is probably JSON data, even if it were an image it would be possible though, we can store it in an indexedDB.
- Now if we always had internet connectivity, the service worker will go ahead and execute that task immediately, so we can take this route even if we know we have internet connection, it will then just execute it immediately. However if we didn't have connectivity and it is re-established, it will also work in both situations.
- So us having connectivity or us getting connectivity again, a so-called sync event will be executed on the service worker and you can listen to that event in the service worker.

- Once this event occurs, the service worker will, if we instruct it to do so, fetch the request data and send the data to the server, so it will then make the request.
- The **cool thing is** this will even work if we close the tab. So our website doesn't have to be open for this to work, we can just go away. That is why it's also a good idea to register a sync task even if we do have internet connection because it'll still execute even if the user immediately closes the browser after hitting a button or something like that.

Periodic Sync

- Periodic sync is all about **getting** data on a regular basis from the server.
- This could be useful if we have some back-end. Imagine Twitter, the Twitter API where data changes frequently and it would be nice if our application could get updates in the background so that the next time the user opens our web app, it already has the fresh data, it doesn't need to fetch it at the point of time we start the app.
- **It's not related to connectivity issues at all. It's about fetching data even if your app is not opened.**

How Periodic Sync works



- We will register a periodic sync task with the service worker.
- There we also pass a schedule where we basically say, please update every hour or something like that.
- And then this schedule will trigger the 'periodicSync' event on the service worker to which we can listen to, based on the schedule.
- So this now then makes the service worker reach out to the server and request new data.

Useful Resources

- Introducing Background Sync: https://developers.google.com/web/updates/2015/12/background-sync
- A Basic Guide to Background Sync: https://ponyfoo.com/articles/backgroundsync
- More about Firebase Cloud Functions: https://firebase.google.com/docs/functions/

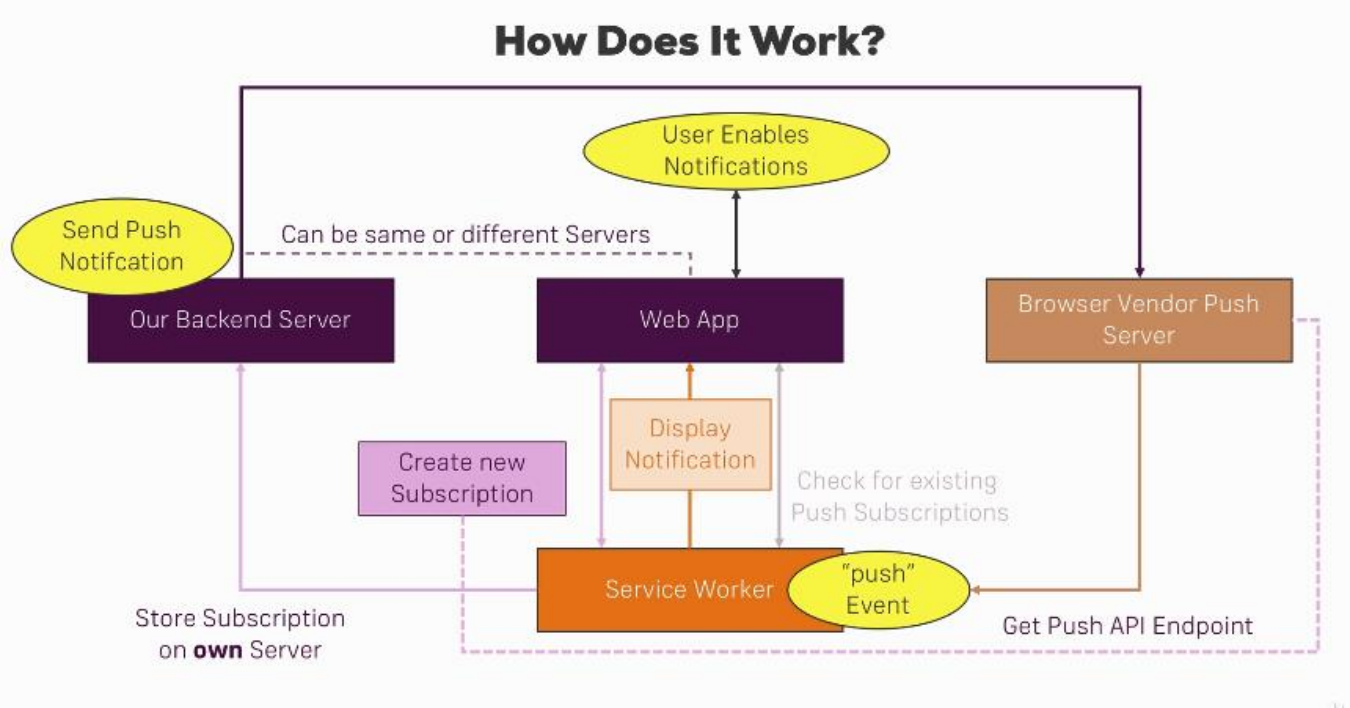
Web Push Notifications

- With Push notifications, you can get your users back into your application.
- That's the best way to drive user engagement and re-engage users with your application. And from a user perspective, you get informed about things which matter to you, at least if it's done in a good way.
- Web push notifications is an extremely powerful feature we can add to progressive web apps to really turn them into native application-like experiences.

Why we need Web Push Notifications

- Show up even if App (and browser) is closed.
- Drive user engagement.
- Mobile-app like experience.

How Push & Notifications work

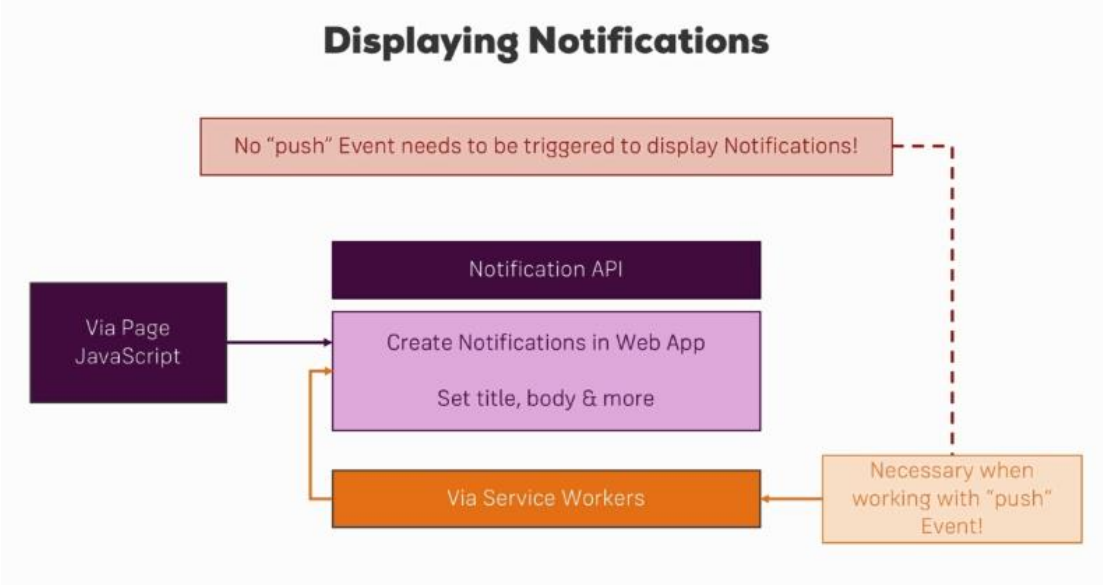


- Push notifications actually contain of two parts – we have **push** which means that some thing, some server pushes some information to our application and we have **notifications** which is what we show to the user in the end.
- First of all, we need the user to enable notifications. We need to prompt for permission otherwise we can't show any notifications and that also means we can’t send push messages to our web app from any server.

- This is a onetime task though, once we have permission it's always granted. Of course the user could remove it through browser tools but if he doesn't do that, we have these notifications in future instances of this web application too.
- Now with notifications enabled, we can already display notifications. It's totally independent of pushing messages to our app.
- Notifications themselves are triggered from Javascript but of course that becomes most interesting if we push messages from some other server to our web application. For instance – Imagine an application where users can create posts and we want to inform all the users about new posts. Well then we can push this message that a new post was created to all web apps that subscribed in our service.
- In the web app, we can check for existing push subscriptions. **A subscription refers to a browser on a given device.**
- So if you use Chrome on Mac, you may subscribe with this browser device combination to push and if you use Firefox on the same Mac, you could set up a new subscription there, the subscription will be managed by the browser for you. You use the service worker to request and work with subscriptions.
- So you can check if there is already an existing subscription because maybe the users might have set one up in the past. If that's not the case, you can create a new one.
- Now a new subscription is created in Javascript with the service worker. However we need more information than just our web app and the service worker.
- Subscriptions for push notifications of course need external servers because a push notification needs to be delivered to our web application and for that we need other servers otherwise it's just our web app and if our web app is closed, there's no way we could show a notification.
- Now a subscription is referring to a browser device combination and to deliver a message per push to our web app, we need help by the browser because by default, our web app is just running and is not receiving any incoming messages.
- We could build something using web sockets but that's a totally different technology and also only works if our web app is open. So we need the help of the browser to get informed even if our web app isn't open.
- That's why any created subscription involves some information by the browser vendor namely each browser vendor has its own push server. So Google has its own server handling push messages, Mozilla has its own server and so on.
- We don't configure these push servers, we don't own them, they are owned by Google, Mozilla and so on. But when we set up a new subscription in Javascript, Javascript will automatically reach out to that push server and fetch an endpoint for us.
- Now this endpoint is simply a URL to which we can later send new push messages so that the browser vendor service will forward them to our web app.

- So a newly created subscription holds that information – the push API endpoint. It will also hold some authentication information to make sure that only we can push to our web app because everyone with this endpoint could send push messages there. So this is one side, we have a subscription with an endpoint to which we can deliver push messages.
- We also need our own server though. You need a server which you work on where you can run your code. It's important because the service worker which creates this new subscription which holds information about that endpoint and so on, this browser vendor push server endpoint, your service worker can now send this created subscription to your back-end server to store it there.
- Why store it in the server? Well to keep it. If you would store it in your web app, not only could you not really work with it due to the way push works, it also would be lost at the point of time you shutdown/restart your web app. You could store it in indexedDB or something like that but even that is not a persistent storage.
- So we store it on our back-end server in some database there because it will be our back-end server which later wants to push messages to the front-end app. E.g. if a post from a user is saved in database, we could issue push messages to all our subscriptions.
- Now if we decide to send a new push notification, so we execute some server side code to do so and this push notification is then sent to the browser vendor push server API endpoint that we stored in the subscription.
- This reaches the notification to the push server by the browser vendor. We also send some authentication information with that push message to be able to really deliver it to the web app and the browser vendor push server then delivers it to our web application.
- To be precise it delivers that to the service worker by triggering the “push” event there. That's a special event we can listen to with the event listener in the service worker and then in the service worker, we can display a notification in our web app.

Displaying Notifications



- You don't need to listen to "push" event to display notifications. This is totally independent of push messages.
- We use the Notification API to create and show notifications. The notification API allows you to create notifications, set a title, a body, possibly images and more.
- You can use the notification API directly from your normal page Javascript as well as from Service Worker.
- Most of the time you use notifications in conjunction with service workers since you want to react to events and then show notification.

Requesting Permissions

- Theoretically if we want to display a notification, the browser will automatically prompt the user. But it's better if we do it manually which allows us to handle the user response and of course control when we in the end ask for it.
- If you ask for **notification** permissions, you implicitly also get **push** permissions. As I mentioned, these are two separate technologies. Notifications will be a simple box you display whilst push is simply the technique of pushing messages from a server to the browser vendor server to your web app.
- If the user block permissions, we can't even ask again. If it permission requested is just undecided and user closed the tab or something like that, he'll get asked next time again but nothing more we can do, so we should try to pick the best possible point of time for asking the permission.
- Web Notification does not mean a Javascript alert or something like that, it's a real system notification, really showing up like all other system notifications.

Displaying Notifications

- We can show notifications via normal JavaScript or from Service worker as well.
- Oftentimes, you actually display notifications through a service worker and actually, you can always use a service worker if your browser supports it.
- Keep in mind – Notifications do not show up in the browser, they are shown by your device (phone, MacBook, Laptop, etc.).

Understanding Notification Options

- Keep in mind – Notifications do not show up in the browser, they are shown by your device (phone, MacBook, Laptop, etc.).
- Therefore it's your phone or your computer deciding which things it can show in a system notification. Sometimes that's the only title and the body, sometimes you can display an image. So you can always add more to options but you can't rely on it being displayed.
- Definitely add it to enhance the user experience on devices where it is supported but be prepared that not all users might see it, so put your core content into the title and the body.
- Refer MDN to see all options available for Notification – https://developer.mozilla.org/en-US/docs/Web/API/Notification
- In the options object to the Notification constructor, we can add properties like body, icon, image, dir (direction of the text), lang, badge, tag, renotify, actions, data, etc.
- The difference between an icon and image is the icon is that thing showing up to the right or left, so next to the title somehow, image is also a part of the content of the notification. By the way, these options are not necessarily supported on all devices.
- Badge – recommended resolution by Google for Android is 96 x 96. Android will automatically mask it for you to display it as black and white. For desktop chrome, it will be displayed at the top left corner just before your app name.
- Tag allows you to assign a tag to your notification. Now the cool thing about the tag is it acts like an ID for that notification and if you send more or display more notifications with the same tag, they will actually not be displayed beneath each other but they will stack onto each other. So basically, the latest notification with the same tag will replace the previous one with the same tag.
- renotify if set to true makes sure that even if you use the same tag, a new notification will still vibrate and alert the user. If it's set to false and you use the same tag on notifications, new notifications of the same tag actually won't vibrate the phone again and won't notify the user again. Renotify is used along with tag.
- **Important –**

    - So if you set **renotify** to **false** and use **tag**, you have very passive notifications, only sitting in the top bar and vibrating on the very first notification.
    - If you set **renotify** to **true**, you still don't spam the user but you have a vibration on each new notification.
    - And if you **disable both**, you send a notification every time and they will sit beneath each other and maybe spam the user.
    - It really depends on which kind of app you're building.
- So these are tag and renotify are options to control how you display your notifications and important for you, to control that you use notifications responsibly and don't spam the user.
- actions – Actions are the buttons displayed next to your notification. Might not be supported by devices. So be careful.
- **data** – The data option is a useful option to pass some extra metadata, data you can later use upon interaction with your notification to the notification and you could pass any data you want, as many properties as you want. Then in the notificationclick listener, you can extract that data.

Reacting to Notification Interaction (clicks)

- We can react to different interactions on the Notification. E.g. click on the notification itself, clicking on the 'action' buttons.
- The interaction/actions with the notification happens only in the Service Worker. This is because a notification is a system feature. It's not displayed in our web application, it's not HTML or something like that, it's displayed by the operating system. Hence the user may interact with it when our page isn't even opened. Infact this is something service workers are about, they run in the background and for example when using Chrome on Android, you will get notifications even if your application is closed, even if the browser is closed.
- Different notification events available in service worker are – `notificationclick, notificationclose.`

From Notifications to Push Messages

- At certain point or click in your application, check if there is already a subscription, if not create a new subscription.
- While creating new subscription, if we have an existing subscription, it will render the old one useless.
- A subscription contains the endpoint of that browser vendor server to which we push our push messages, anyone with this endpoint can send messages to that server and this server will forward them to our web app.

- The security mechanism which we need to have is that we will identify our own application server / our own back-end server as the only valid source sending you push messages, so that anyone else sending push messages to the API endpoint by the browser vendor server will simply not get through.
- Now to identify our own application server, passing just the IP or something like that certainly isn't enough because that's easy to trick and not really secure. So for that we can use VAPID.
- VAPID can be thought of as an approach where we have two keys, a public and a private one. The public one as the name implies is the one which can be exposed to the public, that's the one we'll use in Javascript because as you know, everyone can read our Javascript. The private key is connected to the public one but can't be derived from the public one and is stored on our application server only, and therefore can't be accessed by others. Only the two (public and private) together work.
- The vapid keys use JSON Web Tokens to carry identifying information and simply are converted to base 64 strings keys we can use easily.
- Now you could theoretically create them from scratch manually but we can use package, a Javascript library to generate them. We'll also need a library to send push messages because whilst you can build everything from ground, there is absolutely no sense in doing so as there is a great push library we will use, it's called **web-push**.
  `npm install web-push`

Storing Subscriptions

- Once you decide to create a subscription, you in the end need also need to store on the backend server so that it can be later used to push messages.
- Using web-push package, generate vapid keys (both public and private keys will be created).
- From the vapid public key, generate int8 array from this base64 key as this required by the push manager.
- Use this converted vapid public key and call subscribe on the server worker registration object. It will return a promise which will resolve and yield an actual new subscription object.
- We need to store this new subscription object in our server database.
- Once stored on server, it looks like this –

- This subscription object holds an endpoint which is one of Google's server (since we are using Chrome). That is the endpoint of browser vendor server. This is the URL we have to send push messages to. Anyone with that URL could send the messages.
- We also have keys (auth and p256dh). All three (endpoint and keys) things together incorporate our vapid public key information.
- As soon as we start **sending push messages** from our application server, we need to pass auth and that p256dh key as well as our private vapid key and then the web-push package will use all that information to send a push request to this endpoint which identifies itself as coming from our identified and verified application server.

Important Note about Subscription

- The subscription refers to a browser on the device. However it also refers to a given service worker because we setup subscription through the service worker.
- Now if we clear site data on our browser, we also get rid of the service worker and then also get rid of our existing subscriptions.
- However these subscriptions are still there on the server which we should also get rid of. because these subscriptions on the server won't work anymore if we unregister the service worker.
- If we just update the service worker by simply adding code to the service worker, reloading the page, closing the tab and opening a new one (normal update procedure), it won't be a problem because there, we won't re-install a completely new one but if we unregister a service worker, we render all existing subscriptions useless.
- **Keep in mind** – Keep your service worker registered, update it, whatever you need to do but if you unregister it, also make sure to clear up your subscriptions on the server because they will no longer be valid.

Sending Push Messages from the Server

- We use web-push package for sending the push messages as well.
- Here we will need to use all the stored subscriptions and send the push message to those endpoints from the subscriptions and pass public and private keys to it.

Listening to Push Messages in client

- The service worker is always running in the background at least on some devices and we want to react to push messages when we don't have a web page open. So the service worker is only place where we can listen to them.
- When do we get an incoming push message?
  Well if THIS service worker on THIS browser on THIS device has a subscription to which THIS push message was sent. Each subscription is stored on the server and has its own endpoint and therefore if we send a push message from the server to that subscription, this service worker who created that subscription will receive it. That's the reason why if you unregister a service worker, you won't get it.
- Payload of push messages is limited to 4kb.
- The active service worker itself can't show the notification, it's there to listen to events it's running in the background. That's why we have to get access to the registration of the service worker, that is the part running in the browser.
- Service Worker Registration is the part which connects the (active) service worker to the browser.

Useful Resources:

- More about Web Push by Google:
  https://developers.google.com/web/fundamentals/engage-and-retain/push-notifications/
- More about VAPID:
  https://blog.mozilla.org/services/2016/04/04/using-vapid-with-webpush/
- More about VAPID by Google:
  https://developers.google.com/web/updates/2016/07/web-push-interop-wins
- The web-push npm Package:
  https://github.com/web-push-libs/web-push
- More about showNotification (display Notifications from Service Workers):
  https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/showNotification
- The Notification API:
  https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API
- The Push API:
  https://developer.mozilla.org/en/docs/Web/API/Push_API

Native Device Features

- Let's dive into native device features and how we can use them in our web app.
- Previously, these features were basically only available in native Android and iOS apps, now we can also access some of them through the web. Of course always in a progressive way as not all browsers support this.
- Here we going to use Camera and Geolocation features of Native device.
- Geolocation API allows you to not only get the current position but also watch a position to follow the user, to show that on a map or something.

Useful Resources:

- More about the Media Stream API on MDN: https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API/Constraints
- More about getUserMedia: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia
- How to use geolocation: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation/Using_geolocation

Service Worker Management with Workbox

Understanding the Basics

- Your service worker file can become really big especially if you add a lot of Route parsing or request URL parsing to it to handle different URLs with different caches and strategies.
- Thankfully there is a tool to help you with that. **Workbox** a tool by Google which makes managing your service worker easier and way more automatic.
- You can set up a project in a thousand different ways e.g. webpack, Grunt or without any tools. Whichever setup you use chances are that when your project grows that you want to manage some of the aspects of your service worker automatically and a workbox by Google is a tool relatively new way helps you with that.
- It's a very powerful tool which allows you to easily set up routes with the friend caching strategies and more.

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/
- About Cache Persistence and Storage Limits:
  https://jakearchibald.com/2014/offline-cookbook/#cache-persistence
- Learn more about Service Workers:
  https://developer.mozilla.org/en/docs/Web/API/Service_Worker_API
- Google's Introduction to Service Workers:
  https://developers.google.com/web/fundamentals/getting-started/primers/service-workers
- Cache (MDN)
  https://developer.mozilla.org/en-US/docs/Web/API/Cache
- 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.
- Cache.add() Vs Cache.put() – add() will execute the request first, so it will send the request to the server to get the response and store the response which you need to do. Put() basically needs two arguments, that is both request and response.
- The way responses work is you can only consume/use them once. So storing the response in the cache uses that response and you won't be able to use it again, so we should store the cloned version of response by using response.clone().
- Service Worker file will and should never be cached otherwise we would never get latest Service Worker changes from server.
- Local storage and session storage can be accessed synchronously only. IndexedDB can be accessed synchronously or asynchronously.
- For PWA, we should follow mobile first approach of UI design. Consider adding animations to have a feel of native mobile apps.
- Service Worker Registration is the part which connects the (active) service worker to the browser.
- Read and follow Important Note about Subscription