Progressive web app

https://dev.to/paco\_ita/service-workers-and-caching-strategies-explained-step-3-m4f

https://www.freecodecamp.org/news/build-a-pwa-from-scratch-with-html-css-and-javascript/

Service worker:

A SW is similar to a web worker, both a simple javascript file.  
A web worker does not have a specific task and it is typically used to offload the main thread (where the main web app is running on).

On the other side, a service worker has a specific task and it is to act as a proxy between our web application and the network. It can intercept http requests and serve the responses from the network or from a local cache, according to which caching strategy we implemented (more details later).

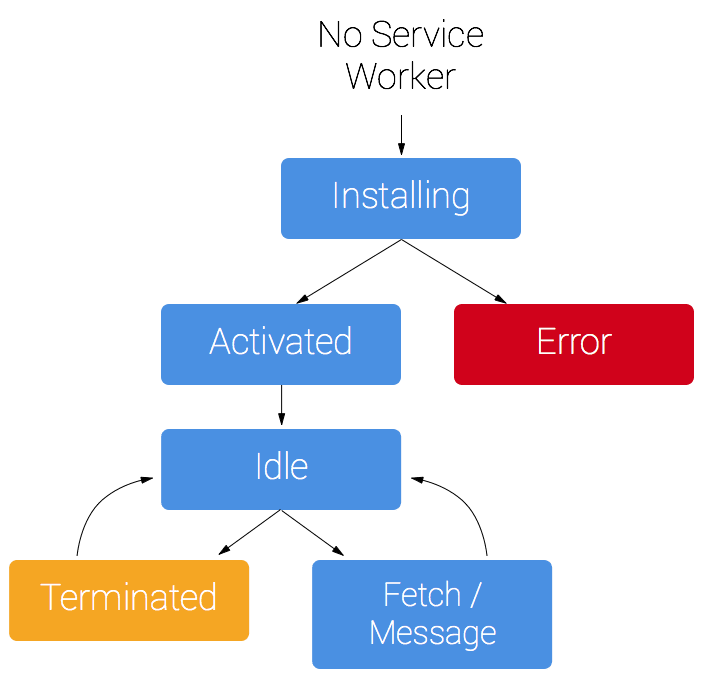
**SW are Secure**

Service workers will **function only on HTTPs connection**.

This is a safe decision, because otherwise we would easily expose our application to man-in-the-middle attacks. Let's just imagine what might happen if anybody could substitute our SW with a manipulated one...scary, isn't it?

On the other side, localhost is considered secure, allowing to test the application before deploying it.  
If we work with Angular though, we cannot use the ng serve command to build and serve our application locally, as it does not work with service workers. In this case we have to use an HTTP Server of our choice, for example [http-server](https://www.npmjs.com/package/http-server) package or the [Web Server](https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb?hl=en) Chrome extension.

Service Workers lifecycle



The schema above describes the different lifecycle steps of a service worker.  
During the registration the whole operation is canceled if an error occurs or the SW file cannot be fetched.  
The register method will be newly triggered when the user loads the page again. The browser is able to identify whether the SW is already installed or not and call the method accordingly.  
  
Once registered, a SW does not remain constantly active. The browser can unpredictably terminate it and reactivate it again when an event needs to be triggered. That's the reason why, if we need to persist a state used within the service worker (I do not mean caching assets or API requests here), we should better use IndexeDB, or a similar solution.  
   
In the install step, pre-fecth operations are typically executed. Their goal is to ensure target assets are downloaded and made already available in the cache for the SW. These assets are commonly static files (eg. js, css) representing the core shell of our application, the minimum files and styles that should be available immediately to the user, even when offline.

Display property in manifest.json

In the context of a web app manifest file (**manifest.json**), the **"display": "standalone"** property indicates how a Progressive Web App (PWA) should be presented or launched on a user's device.

When **"display": "standalone"** is set in the manifest file, it means that the web application will open as a standalone application, separate from the browser. This setting ensures that the PWA opens in its standalone window, hiding browser UI elements like the address bar, back/forward buttons, and other typical browser features. It gives the user an experience closer to a native mobile app.

In a web app manifest file (**manifest.json**), the **"display"** property specifies how a Progressive Web App (PWA) should be presented or launched on a user's device. There are several values that can be used for this property, each influencing how the PWA appears and behaves. Here are the commonly used **display** values:

1. **fullscreen:** This setting hides the browser UI completely, giving the web app a full-screen experience. It opens the PWA without any browser elements visible.
2. **standalone:** This value launches the web app as a standalone application. It appears without browser elements like the address bar, providing a more app-like experience.
3. **minimal-ui:** This setting is similar to **standalone**, but it allows a minimal set of browser UI elements to be displayed, such as back and reload buttons.
4. **browser:** This value opens the web app within the browser as a regular tab or window, retaining all the standard browser features and interface elements.

The **scope** property in a web app manifest file (**manifest.json**) defines the navigation scope of a Progressive Web App (PWA) and determines which pages are under the PWA's control. It sets the base URL for the PWA's actions and interactions, specifying the URLs that the PWA can control and display.

Maskable icons are a specific type of icon used in Progressive Web Apps (PWAs) that allow the browser to adapt the icon's shape for different devices and themes.

In a PWA manifest, the icons used for the app on the home screen, task switcher, or app launcher can have different shapes and sizes based on the device or the user's preferences. Maskable icons are designed to be adjustable, allowing the browser to mask or clip them to fit the shape needed without cropping or stretching.

The icon is considered "maskable" if it meets specific criteria:

1. **Transparent Background:** The icon should have a transparent background, allowing the browser to apply its own mask or shape without being obstructed by a background color.
2. **Visible Content within Safe Zone:** The icon's key content should be located within a designated safe zone. This ensures that important parts of the icon, such as logos or central images, remain visible after being masked or clipped by the browser.

By using maskable icons, PWAs can have more consistent and adaptive iconography across various platforms and devices. This helps in maintaining a unified visual appearance and better integration with different themes or shapes imposed by the platform.

https://danielk.tech/home/angular-app-shell-ultimate-guide

https://blog.angular-university.io/angular-app-shell/

ng new my-app-shell --routing

ng build --prod

ng serve --prod

ng generate universal

ng generate app-shell --route=app-shell-path

ng run AppShellDemo:app-shell:production

ng run angular-pwa-app-shell:app-shell:production

ng run app-shell-test:app-shell:production

App shell commands:

Ng new app-shell

Ng g c todo

Ng generate app-shell

ng run app-shell:app-shell:production

ng run app-shell-universal1:app-shell:production

npm install --save @angular/cdk @angular/material ngx-spinner

npm install cors

const cors = require('cors');

const express = require('express');

const cors = require('cors');

const app = express();

// Allow requests from all origins

app.use(cors());

// Or, specify allowed origins explicitly

app.use(

cors({

origin: ['http://localhost:4200', 'https://example.com'],

})

);

// Your other route handlers here

// ...

npm install web-push -g

web-push generate-vapid-keys

ng g c components/notification

cd server

npm i --save web-push

ng build --prod  
http-server -p4200 -c-1 dist/<name-of-app>node server/server.js

https://arjenbrandenburgh.medium.com/angulars-pwa-swpush-and-swupdate-15a7e5c154ac

https://dev.to/paco\_ita/service-workers-and-caching-strategies-explained-step-3-m4f

https://github.com/anuroopjoy/pwa-sample/blob/master/server/src/main.ts

https://www.freecodecamp.org/news/build-a-pwa-from-scratch-with-html-css-and-javascript/

Testing localhost website with ngRok

You can test your localhost webpage using WebPageTest, which is a powerful tool for analyzing the performance of web pages. Testing a local or development environment with WebPageTest can provide insights into page loading speed, performance metrics, and various other details.

1. **Set Up a Publicly Accessible URL:** WebPageTest usually requires a publicly accessible URL to test a website. For testing a local environment, you can use tools like **ngrok** or **localtunnel** to create a temporary public URL for your local server. These tools expose your local development server to a publicly accessible URL.
2. **Expose Localhost using Ngrok:** For example, using Ngrok, you can expose your local server to a public URL by running Ngrok from the command line after starting your local server:

ngrok http 8080

Replace **4200** with the port your local server is running on.

1. **Access WebPageTest:** Visit the WebPageTest website (<https://www.webpagetest.org/>) and enter the public URL provided by Ngrok or any similar service into the "URL" field.
2. **Choose Testing Options:** Select the location from which you want to run the test (different locations may produce varied results), and set the testing options as per your requirements.
3. **Run the Test:** Click on the "Start Test" button to initiate the test.