**PWA** stands for Progressive Web Apps.

**What is a PWA?**

A Progressive Web Application (PWA) is a type of application software delivered through the web, built using HTML, CSS, and JavaScript. It is intended to work on any platform that uses a standards-compliant browser

Internally, a PWA uses service worker browser API to provide access to some native features. Also, it gives you an app-like feel. Cache storage is a really great feature that helps drastically improve page load time.

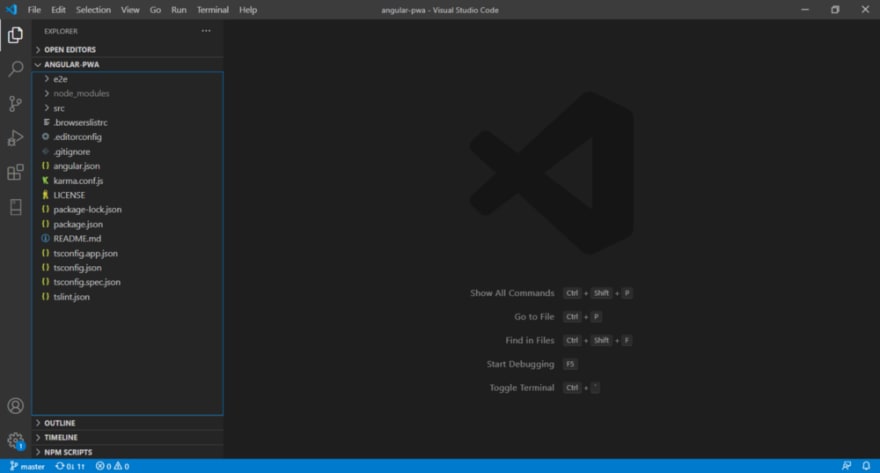
### ****Getting Started with a PWA****

### ****1. Create an Angular application****

ng new angular-pwa

The above command will create a scaffold angular application inside the **angular-pwa** folder. Let us navigate to the **angular-pwa** folder and open VSCode.

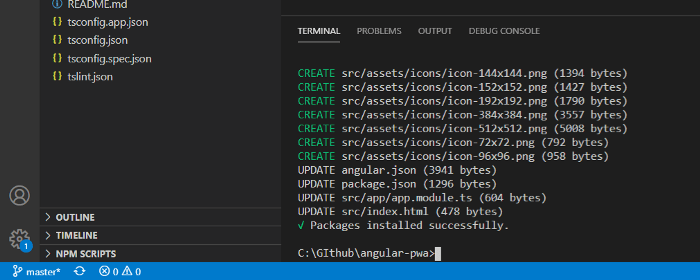
cd angular-pwa && code .



### ****2. Add****@angular/pwa****package****

ng add @angular/pwa

The above command triggers schematics which add and modify several files.



### ****3. Understand the files added/modified by @angular/pwa package****

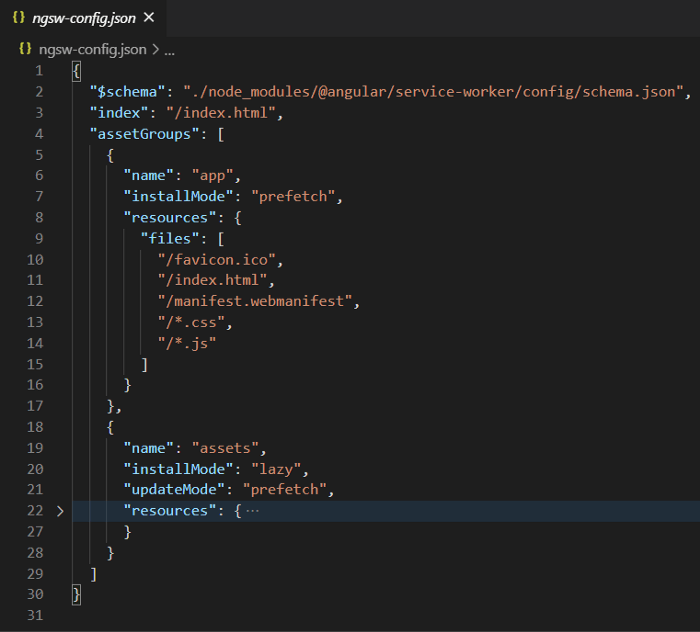
It adds different png files for different splash images for various resolutions icon-128x128.png, icon-144x144.png, icon-152x152.png, icon-192x192.png, icon-384x384.png, icon-512x512.png. Additionally, it adds **ngsw-config.json** and manifest.webmanifest for configuration purposes. Also, it modifies **angular.json, package.json, index.html** and **app.module.ts.**

1. **angular.json:**  
   It includes the serviceWorker: true property to the build configuration, as well as specifying src/manifest.webmanifest be included as a build asset.
2. **manifest.webmanifest:**  
   It is the Web Manifest file associated with the app, which defines things like theme colors and app icons.

Changes in **index.html** and **package.json** are pretty straight forward. Let’s take a quick look at the files changes by CLI.

**ngsw-config.json**

It’s a configuration file in JSON format. Mainly this file is responsible for the generation of ngsw-worker.js(serviceworker.js). You don’t have to write code for that. Just set a certain configuration and you’re done. Ultimately this ngsw-worker.js helps to caches resources/assets with a specific caching strategy. As explained before, under the hood it uses service worker API.



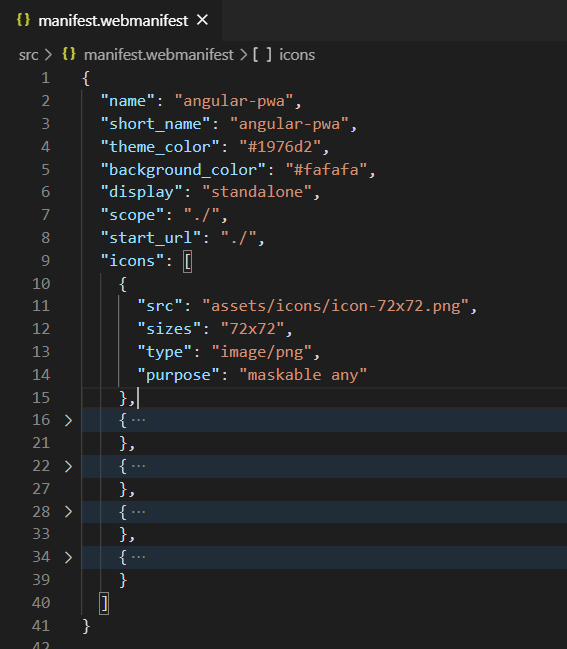
Configurable properties in **ngsw-config.json**

index — This specifies the entry point HTML path.

assetGroups — Here you can the specifies assets or resources that need to be cached and specify the caching strategy, whether it should be network first, cache first, or a combination of both.

**manifest.webmanifest**

Primarily, it consists of how the PWA application will look when it opens up. Here you can set options like splash screen icon, background color, display, etc.



**angular.json**

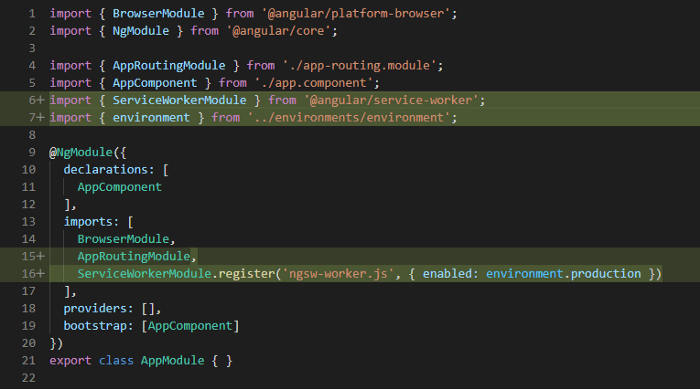
Added src/manifest.webmanifest file under assets, so that it will be served with the site. That links ngswConfigPath and serviceWorker enabling the production configuration in **build** schematics.

Text

Description automatically generated

### ****app.module.ts****

“app.module.ts is used to import the ServiceWorkerModule for registering ngsw-config.js (only for production mode).



### ****4. Run the application locally.****

A PWA only runs on https and localhost environment. The Angular CLI is limited because the service worker doesn’t work with the ng serve command. You have to create a build and host it separately, perhaps using **http-server**

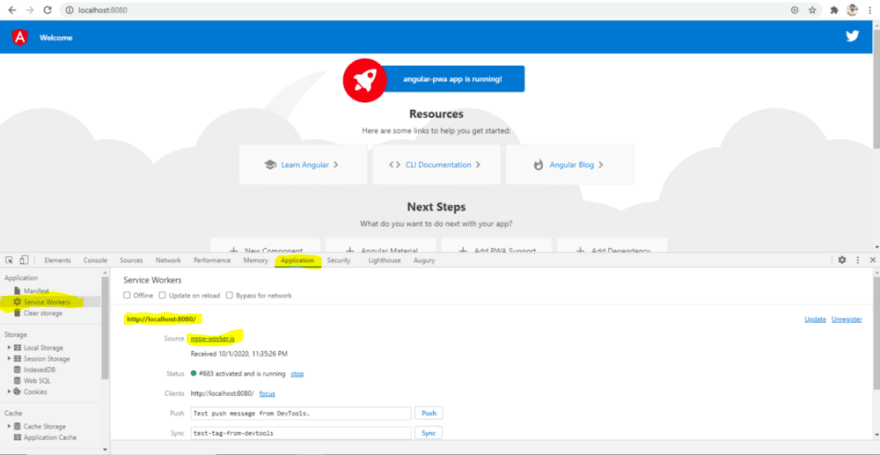
To see a PWA in action, follow the below steps.

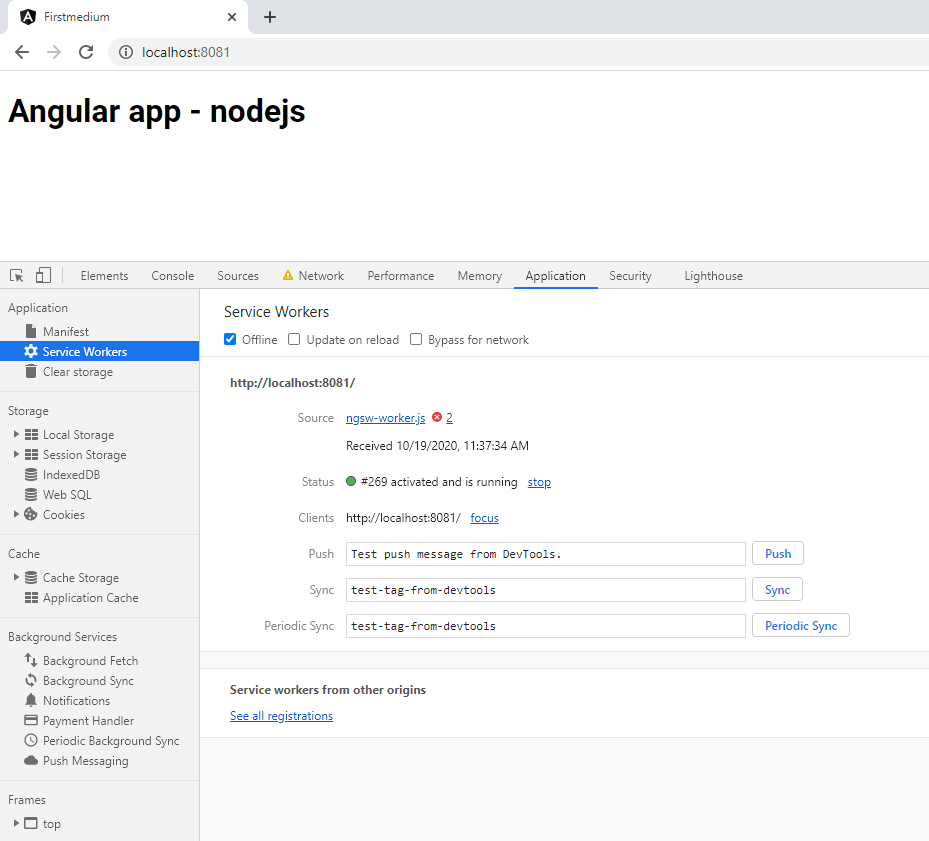
1. Run **ng build –configuration production** command. It will create files under **dist/angular-pwa** folder.
2. Navigate to that folder using **cd dist/angular-pwa**
3. Run **http-server** command (npm i -g http-server)
4. Open

you can see that a PWA is running on http://localhost:8080. Now open the developer console and navigate to Application > Service Workers. You can see there is a service worker file **ngsw-worker.js**installed for <http://localhost:8080>

We can create a script to shorten this process. Open a terminal and run **npm run start-pwa** command. That’s it!

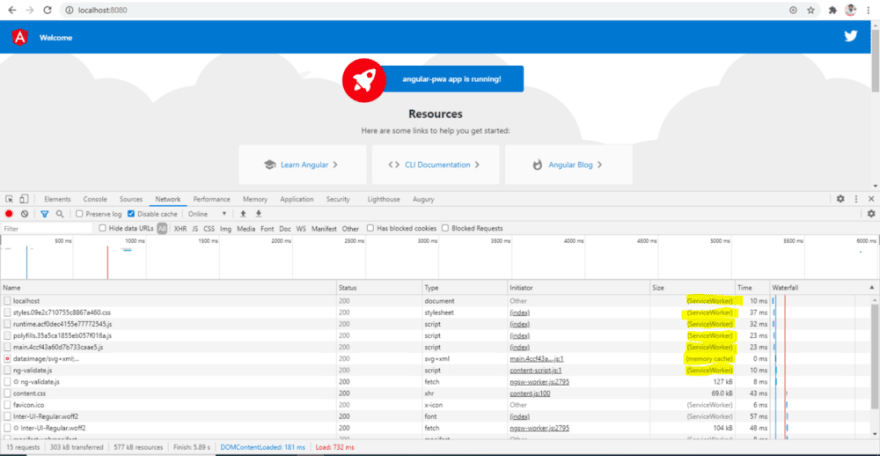
**"start-pwa": "ng build –configuration production && http-server -p 8080 -c-1 dist/angular-pwa"**



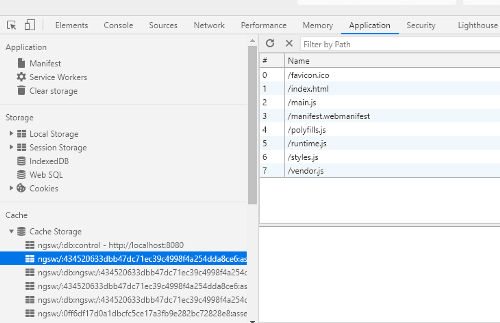


###### **Verifying a PWA installation**

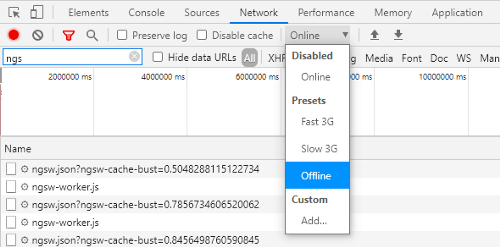
The next time that you reload the browser, all the assets should be loaded from the service worker offline cache.



**Reload browser window network result**



It means that for the first time we are loading all resources, but afterwards all resources will come from cache storage.  
Inside the developer console navigate to “Application > Cache > Cache Storage”. It shows a list of cached resources.



Cache Storage API helps to keep the application accessible in offline mode. Let’s make application offline from Developer Console “Network > Offline (checkbox)”

Reload the page and check what content is loaded so far.

**2. Caching assets for offline use:**

Let’s say i have linked google fonts which is downloaded over internet. But we could cache this fonts in offline mode.  
To cache the static assets like fonts, images etc., we should do the below changes in ‘ngsw-config.json’.

assetGroups is an array of objects and each object will have provision to add static assets path in the ‘urls’ property which is an array. Please refer the image below,

Text

Description automatically generated

After adding the property urls with asset path, rebuild again and launch the app in the browser. Now the request to ‘**https://fonts.googleapis.com/css?family=Roboto&display=swap**’ is cached and served locally in offline mode.

Graphical user interface, application, email

Description automatically generated

**3. Caching dynamic assets and URLs:**We could add dataGroups where we can set our API url which needs to be cached. To demonstrate, I have my serverjs file where i have ‘GET’ method named ‘recipesData’ and we will see how we cache the API for offline mode without internet.

Next to assetGroups, just add the dataGroups as mentioned below in screenshot.

Text

Description automatically generated

The value of name property would be your choice. and the property ‘urls’ contains the path of REST Api(CRUD). And cacheConfig object has certain parameters to define the policy by which matching requests will be cached.

We will see detail about the other properties below,

**maxSize:** Maximum number of outgoing requests needs to be cached.

**maxAge:** To define how long the data needs to be cached.

**timeout:** Time to wait for the response and fallback to the cached data if response is not fetched.

**strategy:** freshness or performance. freshness is looking for the new response with timeout property. Performance brings the response as soon as possible and takes maxage property into the account.

Now you can reload your application and look for the cached API in the browser. The ‘receipesData’ GET method is fetched and cached through Service workers.

# **Web Push Notification**

### Step 1 : Create a server

Let's create a server directory inside our root directory.

npm init -y

{

"name": "server",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

npm i web-push body-parser express cors

Package.json

{

"name": "server",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"start": "node server.js"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"body-parser": "^1.19.0",

"cors": "^2.8.5",

"express": "^4.17.1",

"web-push": "^3.4.3"

}}

server

└───index.js

└───package-lock.json

└───package.json

└───node\_modules

Import  as below

let express = require("express");

let bodyParser = require("body-parser");

let cors = require("cors");

let webpush = require("web-push");

let app = express();

To subscribe push messages, we need to pass **VAPID** keys.  
We can generate **VAPID** keys as below.

console.log(webpush.generateVAPIDKeys()); // new

const vapidKeys = { // new

publicKey: '<YOUR\_PUBLIC\_KEY>', // new

privateKey: '<YOUR\_PRIVATE\_KEY>' // new

}; // new

### Step 2 : Create the client

import { SwPush } from "@angular/service-worker";

export class AppComponent implements OnInit{

title = 'client';

constructor(private \_swPush: SwPush) {}

ngOnInit() {}

}

Then create a method requestSubscription() which will request for notification permission and will give us the subscription object.

//push notification

pushNotification() {

// this.swPush.notificationClicks.subscribe((x) => console.log('data', x));

this.swPush

.requestSubscription({

serverPublicKey: this.VAPID\_PUBLIC\_KEY,

})

.then((subcription) => {

console.log('Send-- ', subcription);

this.api.postSubscription(subcription).subscribe(

(x) => console.log(x),

(err) => console.log(err)

);

})

.catch((err) => console.error('could not subscribe', err));

}

Call pushNotification() method on button click ()

<button class="btn btn-info mt-3 mx-3" (click)="pushNotification()">

Push Notification

</button>

PWA references:-

<https://angular.io/guide/service-worker-intro>.

<https://angular-university.io/course/angular-pwa-course>

<https://www.youtube.com/watch?v=HlYFW2zaYQM&t=1437s>

<https://www.youtube.com/watch?v=rtdP6dvWfus&list=PL4cSPhAvl8xUHh6ojmhFDGdMQmJWFcxrc>