PWA workshop

Part 1

App Manifest and Service Worker

Understanding the App Manifest

1. Introduction

The web app manifest is a simple JSON file that tells the browser about your web application and how it should behave when 'installed' on the user's mobile device or desktop. Having a manifest is one of the requirements to show the Add to Home Screen prompt.

A typical manifest file includes information about the app `name`, `icons` it should use, the `start_url` it should start at when launched, and more.

What You'll Learn

- ✓ Create the manifest
- ✓ Tell the browser about your manifest
- ✓ Key manifest properties
- ✓ Dealing with different browsers
- ✓ Generating the manifest and other assets

What you'll need

- Chrome 67, Safari 12.1 or above
- Your favorite text editor
- Basic knowledge of HTML and JSON

2. Getting set up

Project Set Up

Starting with this code lab, we are going to build on top of an existing project. **Fork** and then **Clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa.git
$ cd fe-guild-2019-pwa
```

The `master` branch contains the final code for the application but if you want to code along you need to `checkout` the correct branch

```
$ git checkout pwa-app-manifest-init
```

First install the dependencies

```
$ npm install
```

Then type in the terminal

```
$ npm start
```

and open Chrome at `localhost:8080`

3. Create the Manifest

Create a ***manifest.json*** file in the same folder as **index.html**. A minimal manifest.json file for a progressive web app.

```
{
   "name": "Progressive Selfies",
   "short_name": "PWA Selfies",
   "icons": [
       {
           "src": "src/images/icons/app-icon-48x48.png",
           "type": "image/png",
           "sizes": "48x48"
       },
       {
           "src": "src/images/icons/app-icon-512x512.png",
           "type": "image/png",
           "sizes": "512x512"
       }
   ],
   "start_url": "/index.html",
   "scope": ".",
   "display": "standalone",
   "orientation": "portrait-primary",
   "background_color": "#fff",
   "theme_color": "#3f51b5",
   "description": "Take selfies PWA style.",
   "dir": "ltr",
   "lang": "en-US"
}
```

Tell the browser about your manifest

When you have created the manifest, add a link tag to all the pages that encompass your web app. Add this to your **index.html**

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

Key manifest properties

`short_name` and/or `name`

You must provide at least the `short_name` or `name` property. If both are provided `short_name` is used on the user's home screen, launcher, or other places where space may be limited. `name` is used in the app install prompt.

```
"name": "Progressive Selfies",
"short_name": "PWA Selfies"
```

`icons`

when a user adds your site to their home screen, you can define a set of icons for the browser to use. These icons are used in places like the home screen, app launcher, task switcher, splash screen, etc.

`icons` is an array of image objects. Each object should include the `src`, a `sizes` property, and the `type` of image.

```
"icons": [
    {
      "src": "/src/images/icons/app-icon-192x192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/src/images/icons/app-icon-512x512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
]
```

> **Tip:** include a 192x192 pixel icon and a 512x512 pixel icon. Chrome will automatically scale the icon for the device. On other browsers or if you'd prefer to scale your own icons and adjust them for pixel-perfection, provide icons in increments of 48dp.

`start_url`

The `start_url` tells the browser where your application should start when it is launched and prevents the app from starting on whatever page the user was on when they added your app to their home screen.

Your `start_url` should direct the user straight into your app, rather than a product landing page. Think about what the user will want to do once they open your app, and place them there.

```
"start_url": "/index.html"
```

> **Tip:** add a query string to the end of the **start_url** to track how often your installed app is launched.

`background_color`

The `background_color` property is used on the splash screen when the application is first launched.

`display`

Display Mode	Description	Fallback
`fullscreen`	All of the available display area is used, and no user agent chrome is shown.	`standalone`
`standalone`	The application will look and feel like a standalone application. This can include the application having a different window, its own icon in the application launcher, etc. In this mode, the user agent will exclude UI elements for controlling navigation but can include other UI elements such as a status bar.	`minimal-ui`
`minimal-ui`	**Not supported by Chrome** The application will look and feel like a standalone application but will have a minimal set of UI elements for controlling navigation. The elements will vary by browser.	`browser`
`browser`	The application opens in a conventional browser tab or new window, depending on the browser and platform. This is the default.	(None)

`orientation`

You can enforce a specific orientation, which is advantageous for apps that work in only one orientation, such as games. Use this selectively. Users prefer selecting the orientation.

```
"orientation": "portrait-primary"
```

Orientation may be one of the following values:

- `any`
- `natural`
- `landscape`
- `landscape-primary`
- `landscape-secondary`
- `portrait`
- `portrait-primary`
- `portrait-secondary`

`scope`

The `scope` defines the set of URLs that the browser considers to be within your app, and is used to decide when the user has left the app and should be bounced back out to a browser tab. The `scope` controls the URL structure that encompasses all the entry and exit points in your web app. Your `start_url` must reside within the scope.

A few other tips:

- If you don't include a `scope` in your manifest, then the default implied `scope` is the directory that your web app manifest is served from.
- The `scope` attribute can be a relative path (`../`), or any higher level path (`/`) which would allow for an increase in coverage of navigations in your web app.
- The `start_url` must be in the scope.
- The `start_url` is relative to the path defined in the `scope` attribute.
- A `start_url` starting with `/` will always be the root of the origin.

`theme_color`

The `theme_color` sets the color of the toolbar and may be reflected in the app's preview in task switchers.

> **Tip:** the **theme_color** should match the **meta** theme color specified in your document head.

`description`

Provides a general description of what the pinned website does.

```
"description": "Take selfies PWA style."
```

`dir`

Specifies the primary text direction for the `name`, `short_name`, and `description` properties. Together with the `lang` property, it helps the correct display of right-to-left languages.

```
"dir": "rtl",
"lang": "ar",
"short_name": "بطة وجه التطبيق"
```

It may be one of the following values:

- `ltr` (left-to-right)
- `rtl` (right-to-left)
- `auto`

`lang`

Specifies the language for the `name`, `short_name`, and `description` properties. This value is a string containing a single language tag.

```
"lang": "en-US"
```

Exercise

Now you can check that the *manifest* is enabled by opening the Chrome Developer Tools and then:

Application ==> **Manifest**

Generating the app.manifest assets

Creating the manifest by hand is not a difficult task, but for sure it is not a fun one. You can use one of the following links to generate the manifest file:

- https://pwafire.org/developer/tools/get-manifest/
- https://app-manifest.firebaseapp.com/

- *Optional:* https://www.pwabuilder.com/

Exercise: Use one of the builders to build a manifest file

5. Dynamic manifest

For some reason or another, you might not be able to serve a json file that contains your web app manifest. Maybe you'd like to construct the app manifest on the client side using custom client-selected theme color or icons without involving your server.

> In our case, this will be extremely useful when we push the code to **GitHub Pages**. More on this on **Service Workers** code lab.

In `index.html`

```
<base href="/">
<!--<link rel="manifest" href="manifest.json">-->
<link rel="manifest" id="manifestPlaceholder">
```

What we did:

- Added a `base` tag
- Removed the `href` attribute from the `link[rel="manifest"]` tag
- And added an `id` attribute

Add this in app.js

```
const manifest = { ... };
// Replace { ... } with the content of the manifest.json file and remove the "start_url" and "scope" !!!
window.addEventListener('load', () => {
    const base = document.querySelector('base');
    let baseUrl = base && base.href || '';

    if (!baseUrl.endsWith('/')) {
        baseUrl = `${baseUrl}/`;
    }

    manifest['start_url'] = `${baseUrl}index.html`;
    manifest.icons.forEach(icon => {
        icon.src = `${baseUrl}${icon.src}`;
    });

    const stringManifest = JSON.stringify(manifest);
    const blob = new Blob([stringManifest], {type: 'application/json'});
    const manifestURL = URL.createObjectURL(blob);
    document.querySelector('#manifestPlaceholder').setAttribute('href', manifestURL);
});
```

What we did:

- Set up a constant `manifest` with the content of the `manifest.json` file
- Removed the `scope` and `startup_url` properties
- In the `load` event callback read the value of the `href` tag from the `base` tag and:
    - Set the `startup_url`.
    - Set the `src` property for each `icon`.
- Created a `json` blob and an `URL` for it.
- Dynamically set the `href` tag back to the `link[rel="manifest"]` tag.

Exercises

1. Adapt your code to publish a dynamic manifest
2. Check in Chrome DevTools that the App Manifest looks similar with this:
   `blob:http://localhost:8080/9a3ad65e-8dde-4690-8245-6a6d260479d6`

Solution

Source code

```
$ git checkout pwa-app-manifest-final
```

Introduction to Service Workers

1. Introduction

What is a service worker

A service worker is a script that your browser runs in the background, separate from a web page, opening the door to features that don't need a web page or user interaction. Today, they already include features like push notifications and background sync. In the future, service workers might support other things like periodic sync or geofencing. The core feature discussed in this code lab is the ability to intercept and handle network requests, including programmatically managing a cache of responses.

The reason this is such an exciting API is that it allows you to support offline experiences, giving developers complete control over the experience.

What You'll Learn

- ✓ The Service Worker Lifecycle
- ✓ Registering a Service Worker
- ✓ Lifecycle and non-lifecycle events
- ✓ Install to the Home screen
- ✓ Getting that "App Install Banner" and then deferring it
- ✓ Caching - including versioning and prechaching resources

What you'll need

- Firefox 61, Chrome 58, Edge 11, Safari 11.1, Opera 44 or above
- Your favorite text editor
- Basic knowledge of HTML, CSS, and JavaScript (ES6/2015)

3. The Service Worker Lifecycle

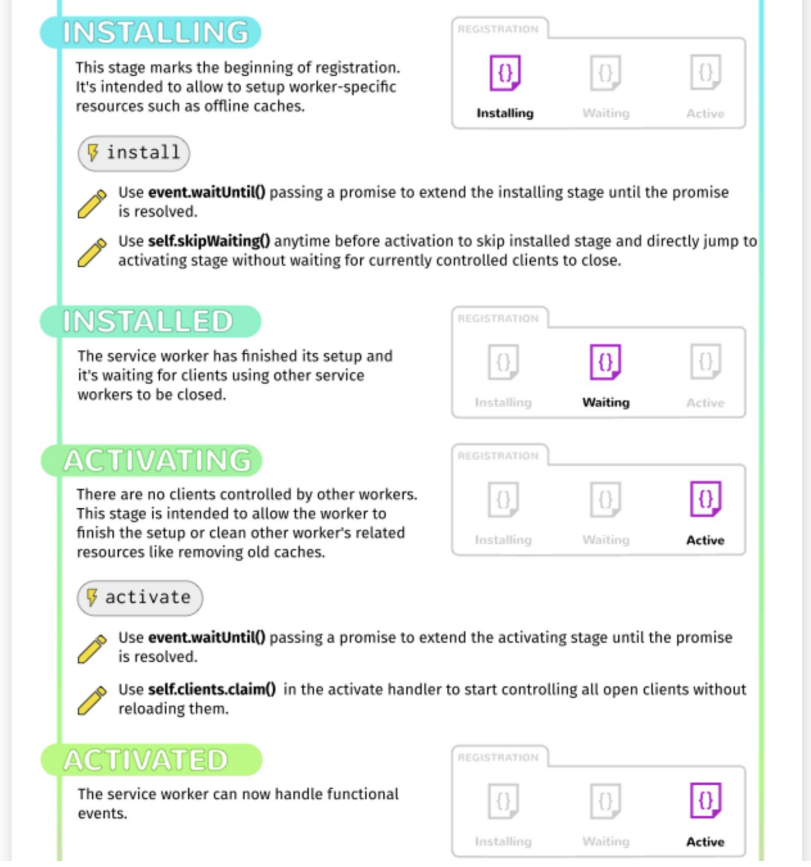
The service worker life cycle

A service worker has a lifecycle that is completely separate from your web page.

With service workers, the following steps are generally observed for basic set up:

- To install a service worker for your site, you need to register it, which you do in your page's JavaScript. Registering a service worker will cause the browser to start the service worker install step in the background.
- Next is activation. When the service worker is installed, it then receives an activate event. The primary use of onactivate is for cleanup of resources used in previous versions of a Service worker script.
- The Service worker will now control pages, but only those opened after the `register()` is successful. i.e., a document starts life with or without a Service worker and maintains that for its lifetime. So documents will have to be reloaded to actually be controlled.

Service Worker Lifecycle



Register a Service Worker in sw.js

To install a service worker, you need to kick start the process by registering it on your page. This tells the browser where your service worker JavaScript file lives. Create an empty file named **sw.js** on the same level as **index.html**. Then in **src/js/app.js** *add* this:

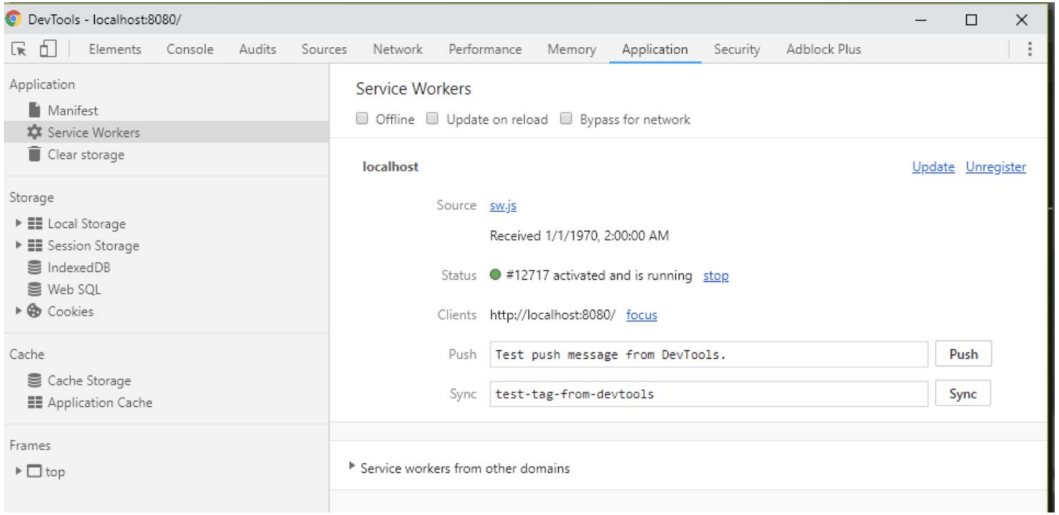
```
window.addEventListener('load', () => {
  // Place this code after the existing code !!!
   if ('serviceWorker' in navigator) {
       navigator.serviceWorker.register(`${baseUrl}sw.js`)
           .then( registration => {
           // Registration was successful
           console.log('ServiceWorker registration successful with scope: ', registration.scope);
       })
       .catch(err => {
           // registration failed :(
           console.log('ServiceWorker registration failed: ', err);
       });
   }

});
```

This code checks to see if the service worker API is available, and if it is, the service worker at `[baseUrl]/sw.js` is registered once the page is loaded.

You can call `register()` every time a page loads without concern; the browser will figure out if the service worker is already registered or not and handle it accordingly.

One subtlety with the `register()` method is the location of the service worker file. In this case, the service worker file is at the root of the domain. This means that the service worker's scope will be the entire origin. In other words, this service worker will receive fetch events for everything on this domain. If we register the service worker file at `/src/js/sw.js`, then the service worker would only see fetch events for pages whose URL starts with `/src/js/`.

Now you can check that a service worker is enabled by opening the Chrome Developer Tools and then `Application -> Service Workers`



You may find it useful to test your service worker in an Incognito window so that you can close and reopen knowing that the previous service worker won't affect the new window. Any registrations and caches created from within an Incognito window will be cleared out once that window is closed.

Why is my service worker failing to register?

This could be for the following reasons:

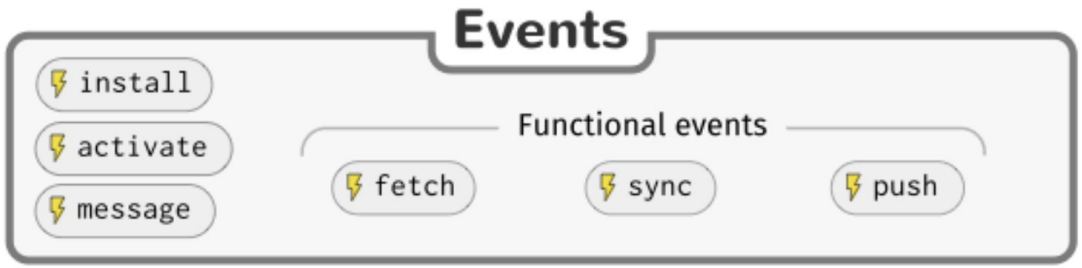
- You are not running your application through HTTPS. During development, you'll be able to use service worker through `localhost`, but to deploy it on a site you'll need to have HTTPS setup on your server.
- The path to your service worker file is not written correctly — it needs to be written relative to the origin, not your app's root directory.
- The service worker being pointed to is on a different origin to that of your app. This is also not allowed.

Exercises

1. Update the application with the code above
2. Check, in Chrome Developer Tools that the service is active

4. Service Worker Events

The below graphic shows a summary of the available service worker events:



Event	Description
`install`	It is sent when the service worker is being installed.
`activate`	It is sent when the service worker has been registered and installed. This place is where you can clean up anything related to the older version of the service worker if it's been updated.
`message`	It is sent when a message is received in a service worker from another context
`fetch`	It is sent whenever a page of your site requires a network resource. It can be a new page, a JSON API, an image, a CSS file, whatever.
`sync`	It is sent if the browser previously detected that the connection was unavailable, and now signals the service worker that the internet connection is working.
`push`	It is invoked by the Push API when a new push event is received.

Add this in sw.js

Let's listen to the main lifecycle events ***install*** and ***activate***

Add this in the **sw.js:**

```
self.addEventListener('install', event => {
 console.log('[Service Worker] Installing Service Worker ...', event);
 event.waitUntil(self.skipWaiting());
});
self.addEventListener('activate', event => {
 console.log('[Service Worker] Activating Service Worker ...', event);
 return self.clients.claim();
});
```

In the `install`, callback is calling the `skipWaiting()` function to trigger the `activate` event and tell the Service Worker to start working immediately without waiting for the user to navigate or reload the page.

The `skipWaiting()` function forces the waiting Service Worker to become the active Service Worker. The `self.skipWaiting()` function can also be used with the `self .clients.claim()` function to ensure that updates to the underlying Service Worker take effect immediately.

The code in the next listing can be combined with the `skipWaiting()` function in order to ensure that your Service Worker activates itself immediately.

Exercises

1. Update the application with the code above
2. Check, in Chrome Developer Tools that the events are triggered

5. Add to Home Screen

Add to Home Screen, sometimes referred to as the web app install prompt, makes it easy for users to install your Progressive Web App on their mobile or desktop device. After the user accepts the prompt, your PWA will be added to their launcher, and it will run like any other installed app.

Chrome handles most of the heavy lifting for you:

- On mobile, Chrome will generate a WebAPK, creating an even more integrated experience for your users.
- On desktop, your app will be installed, and run in an app window.

What are the criteria?

In order for a user to be able to install your Progressive Web App, it needs to meet the following criteria:

- The web app is not already installed
- Meets a user engagement heuristic (currently, the user has interacted with the domain for at least 30 seconds)
- Includes a web app manifest that includes:
  - `short_name` or `name`
  - `icons` must include a 192px and a 512px sized icons
  - `start_url`
  - `display` must be one of: `fullscreen`, `standalone`, or `minimal-ui`
- Is served over HTTPS (required for service workers)
- Has registered a service worker with a `fetch` event handler

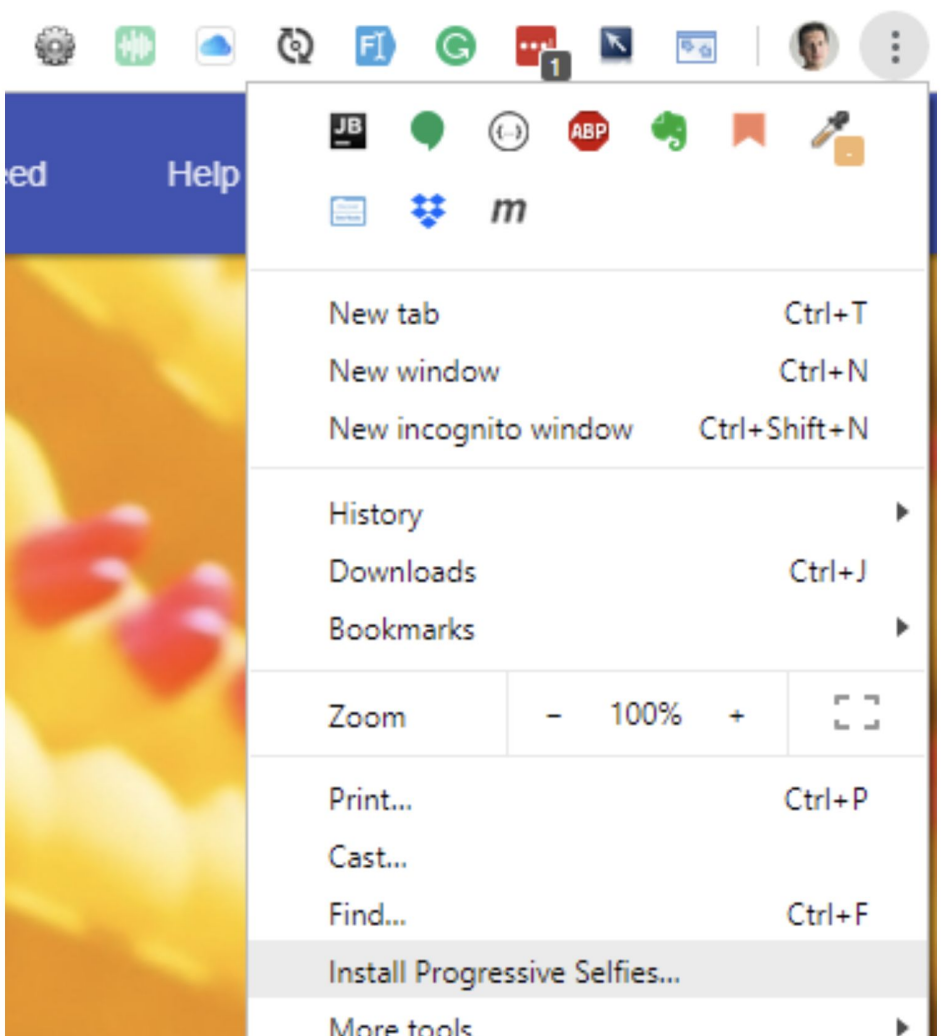
When these criteria are met, Chrome will fire a `beforeinstallprompt` event that you can use to prompt the user to install your Progressive Web App.

We meet all the criteria above but one: we don't have a `fetch` event handler. Let's fix that

Add this in sw.js

```
self.addEventListener('fetch', event => {
 console.log('[Service Worker] Fetching something ....', event);
 // This fixes a weird bug in Chrome when you open the Developer Tools
 if (event.request.cache === 'only-if-cached' && event.request.mode !== 'same-origin') {
     return;
 }
 event.respondWith(fetch(event.request));
});
```

In Chrome click the Customize button and select `Install Progressive Selfies...`



App Install Banner

> The **App Install Banner** will show up only on Chrome on Android, but we really need to serve the app over HTTPS. In theory, it could work on `localhost` with an Android device connected via USB cable, but I didn't manage to make it work.

Optional

The easiest way is to have the app served over `HTTPS` is to publish it to `GitHub Pages` but in order to do that there are several changes needed:

1. Hosting our site to `GitHub Pages` will place it under the following URL: `https://[YOUR GITHUB PROFILE].github.io/fe-guild-2019-pwa/` (assuming that you didn't change the repository name when forking). This means that our `base` tag needs to change to

   ```
   <base href="/fe-guild-2019-pwa/">
   ```

2. Changing the `base` tag will break our app locally. To fix that rename the `public` folder to `fe-guild-2019-pwa`. Stop and restart the app, and we can access it now on `localhost:8080/fe-guild-2019-pwa/`

3. To easily deploy to `GitHub Pages` install the `gh-pages` node package:

   ```
   $ npm install gh-pages --save-dev
   ```

4. Add a new script to `package.json`

   ```
   "deploy": "gh-pages -d fe-guild-2019-pwa"
   ```

   and run

   ```
   $ npm run deploy
   ```

   The first deployment will take longer because it will create a new branch `gh-pages` and will activate `GitHub Pages` on this new branch.

5. Take a few minutes (max 5) break and try

   ```
   https://[YOUR GITHUB PROFILE].github.io/fe-guild-2019-pwa/
   ```

   You can now access the application from your phone, and if you have an Android phone, you will also get an `App Install Banner` *(Try to group around a colleague that has one to see it in action)*.

Exercise

1. Experiment with the code in this step
2. ***Optional:*** Deploy the app on GitHub Pages

6. Service Worker Caching

Imagine you're on a train using your mobile phone to browse your favorite website. Every time the train enters an area with an unreliable network, the website takes ages to load—an all-too-familiar scene. This is where Service Worker caching comes to the rescue. Caching ensures that your website loads as efficiently as possible for repeat visitors.

The basics of HTTP caching

Modern browsers can interpret and understand a variety of HTTP requests and responses and are capable of storing and caching data until it's needed. After the data has expired, it will go and fetch the updated version. This ensures that web pages load faster and use less bandwidth.

A web server can take advantage of the browser's ability to cache data and use it to improve the repeat request load time. If the user visits the same page twice within one session, there's often no need to serve them a fresh version of the resources if the data hasn't changed. This way, a web server can use the Expires header to notify the web client that it can use the current copy of a resource until the specified "Expiry date." In turn, the browser can cache this resource and only check again for a new version when it reaches the expiry date.

HTTP caching is a fantastic way to improve the performance of your website, but it isn't without flaws. Using HTTP caching means that you're relying on the server to tell you when to cache a resource and when it expires. If you have content that has dependencies, any updates can cause the expiry dates sent by the server to easily become out of sync and affect your site.

The basics of caching Service Worker caching

You may be wondering why you even need Service Worker caching if you have HTTP caching. How is Service Worker caching different? Well, instead of the server telling the browser how long to cache a resource, you are in complete control. Service Worker caching is extremely powerful because it gives you programmatic control over exactly how you cache your resources. As with all Progressive Web App (PWA) features, Service Worker caching is an enhancement to HTTP caching and works hand-in-hand with it.

The power of Service Workers lies in their ability to intercept HTTP requests. In this step, we'll use this ability to intercept HTTP requests and responses to provide users with a lightning-fast response directly from cache.

Precaching during Service Worker installation

Using Service Workers, you can tap into any incoming HTTP requests and decide exactly how you want to respond. In your Service Worker, you can write logic to decide what resources you'd like to cache, what conditions need to be met, and how long to cache a resource for.

When the user visits the website for the first time, the Service Worker begins downloading and installing itself. During the installation stage, you can tap into this event and prime the cache with all the critical assets for the web app.

Add this in sw.js

```
const CACHE_STATIC_NAME = 'static';
const URLS_TO_PRECACHE = [
  '/',
  'index.html',
  'src/js/app.js',
  'src/js/feed.js',
  'src/lib/material.min.js',
  'src/css/app.css',
  'src/css/feed.css',
  'src/images/main-image.jpg',
  'https://fonts.googleapis.com/css?family=Roboto:400,700',
  'https://fonts.googleapis.com/icon?family=Material+Icons',
  // 'https://code.getmdl.io/1.3.0/material.indigo-deep_orange.min.css"'
];

self.addEventListener('install', event => {
  console.log('[Service Worker] Installing Service Worker ...', event);
  event.waitUntil(
      caches.open(CACHE_STATIC_NAME)
          .then(cache => {
              console.log('[Service Worker] Precaching App Shell');
              cache.addAll(URLS_TO_PRECACHE);
          })
          .then(() => {
              console.log('[ServiceWorker] Skip waiting on install');
              return self.skipWaiting();
          })
  );
});
```

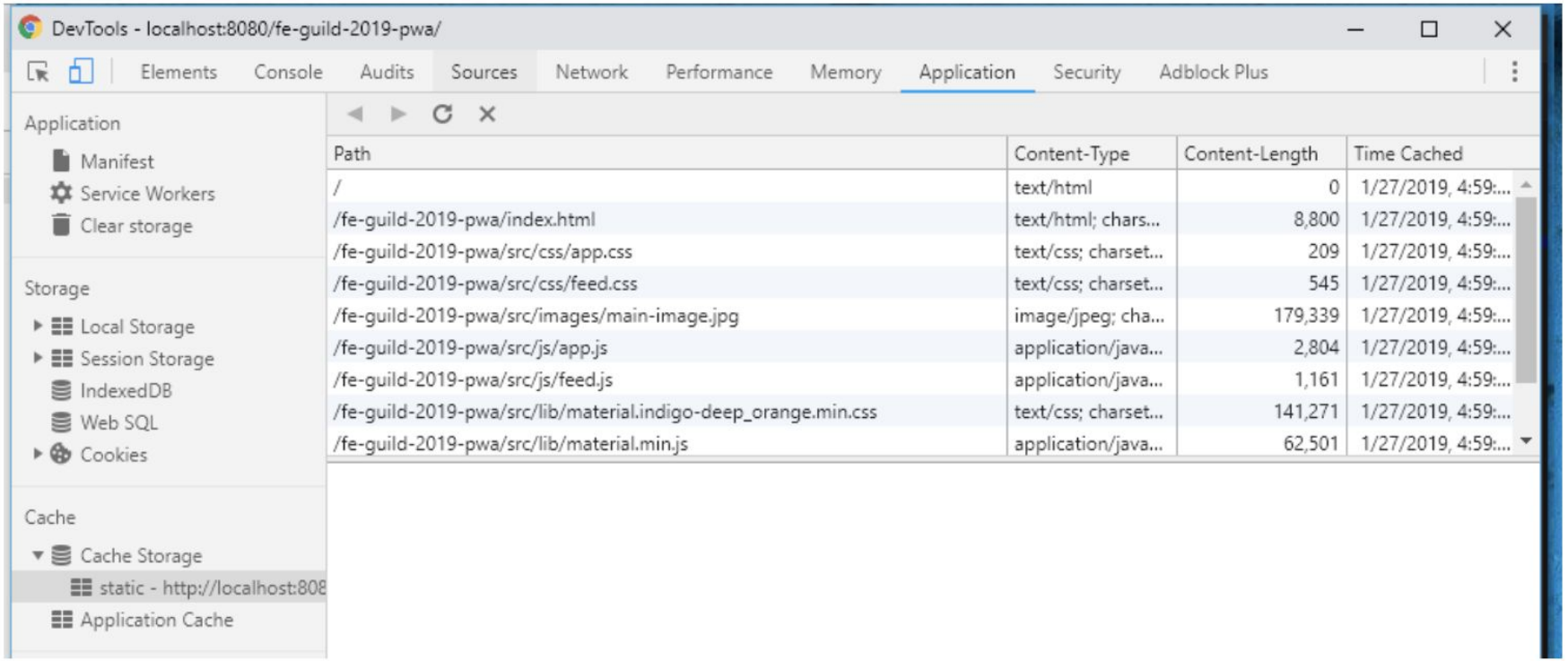
The code above taps into the install event and adds a list of files `URLS_TO_PRECACHE` during this stage. It also references a variable called `CACHE_STATIC_NAME`. This is a string value that I've set to name the cache. You can name each cache differently, and you can even have multiple different copies of the cache because each new string makes it unique. This will come in handy later in the step when we look at versioning and cache busting.

You can see that once the cache has been opened, you can then begin to add resources into it. Next, you call `cache.addAll()` and pass in your array of files. The `event.waitUntil()` method uses a JavaScript promise to know how long installation takes and whether it succeeded.

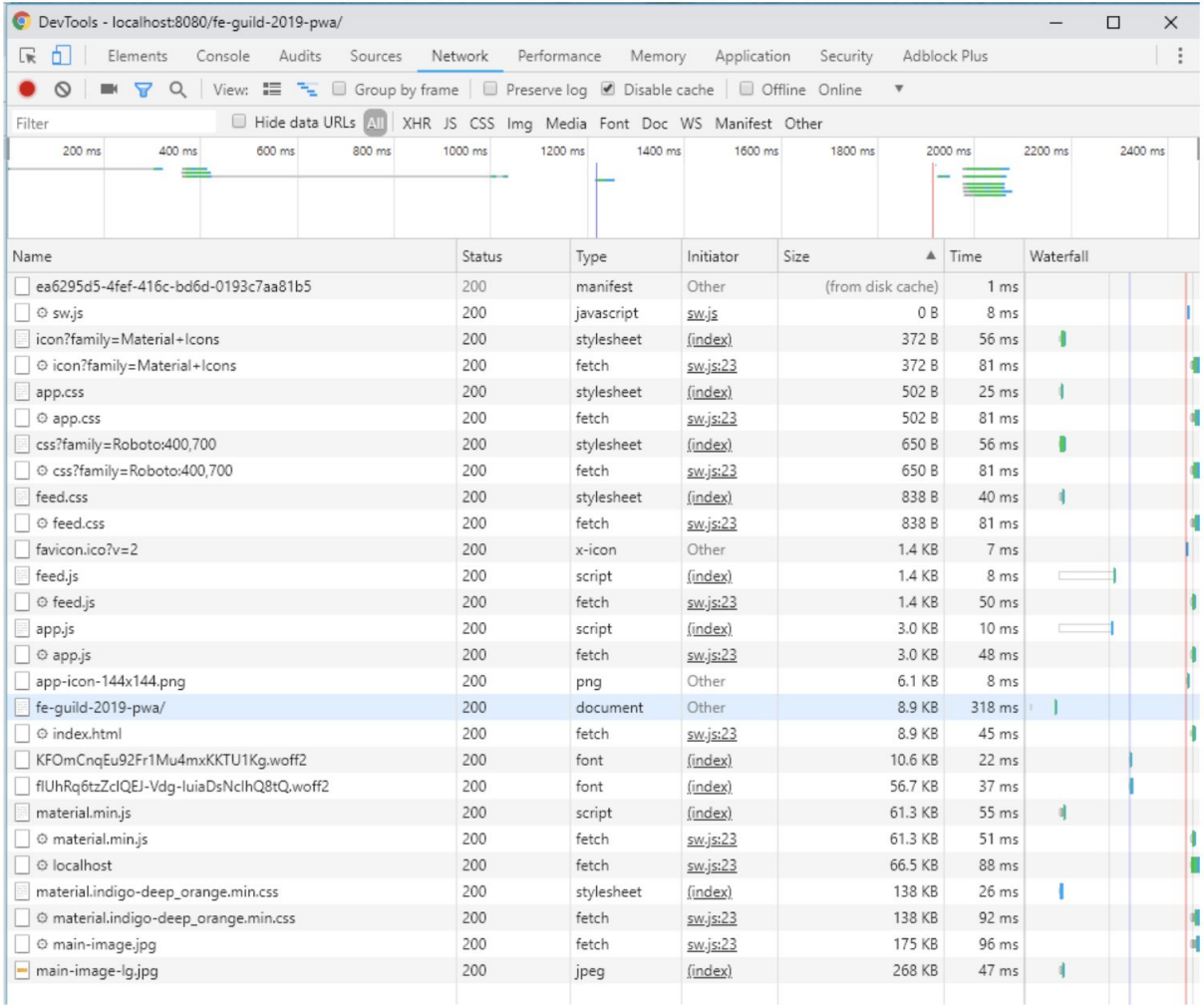
Important to return the promise here to have `skipWaiting()` fire after the cache has been updated.

If all the files are successfully cached, the Service Worker will be installed. If any of the files fails to download, the `install` step will fail. This is important because it means you need to rely on all the assets being present on the server and you need to be careful with the list of files that you decide to cache in the install step. Defining a long list of files will increase the chances that one file may fail to cache, leading to your Service Worker not being installed.

If you check now in Chrome Developer Tools you will see that the cache is filled with the static files from the `URLS_TO_PRECACHE` array:



But if you look in the `Network` tab (even after a refresh) the files are still fetched over the network:



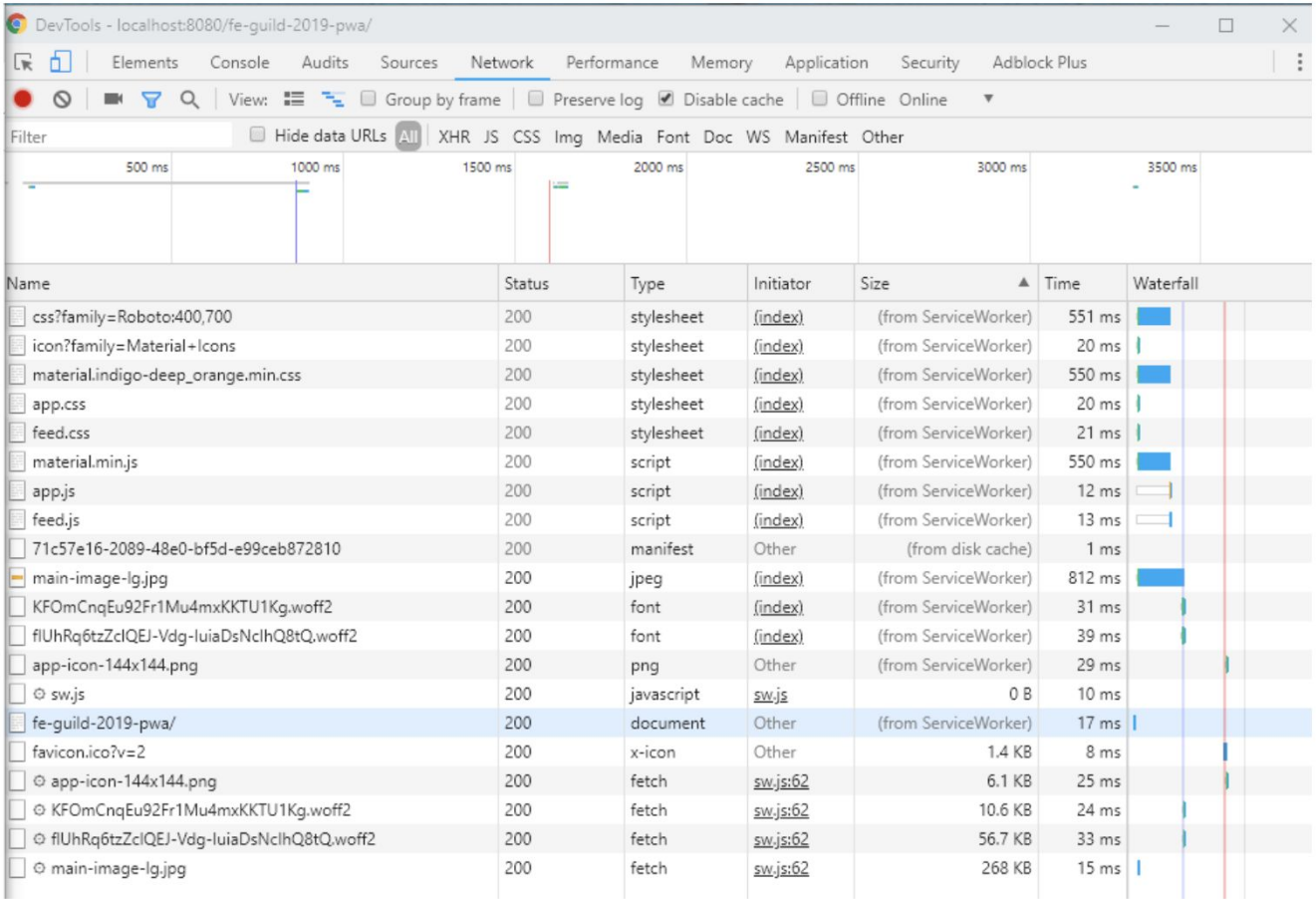
Add this in sw.js

The reason is that the cache is primed and ready to go, but we are not reading assets from it. In order to do that we need to add the code in the next listing to our Service Worker in order to start listening to the ***fetch*** event.

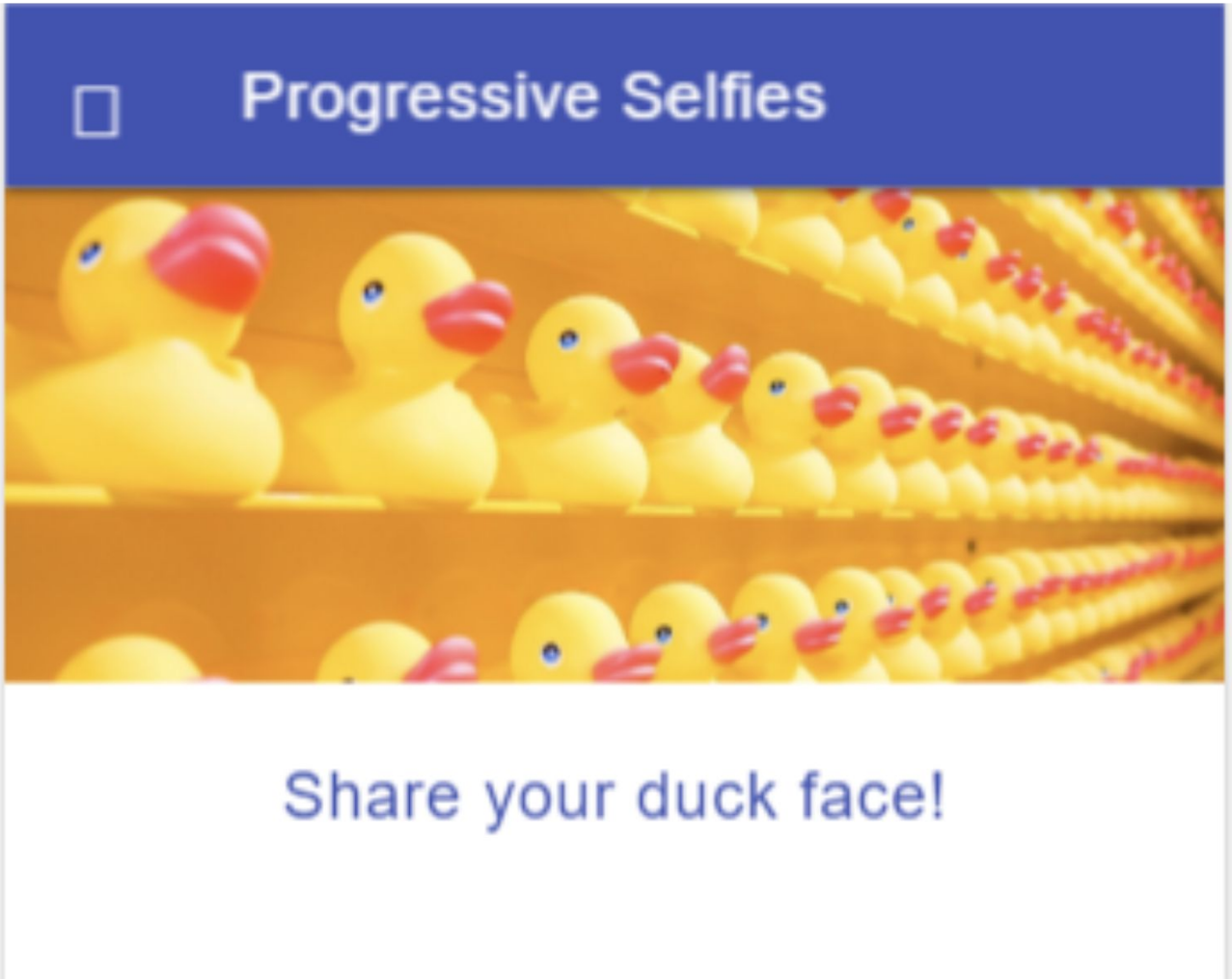
```
self.addEventListener('fetch', event => {
 console.log('[Service Worker] Fetching something ....', event);
 event.respondWith(
     caches.match(event.request)
         .then(response => {
             if (response) {
                 console.log(response);
                 return response;
             }
             return fetch(event.request);
         })
 );
});
```

We are checking if the incoming URL matches anything that might exist in our current cache using the ***caches.match()*** function. If it does, return that cached resource, but if the resource doesn't exist in the cache, continue as normal and fetch the requested resource.

After the Service Worker installs and activates refresh the page and check the `Network` tab again. The Service Worker will now intercept the HTTP request and load the appropriate resources instantly from the cache instead of making a network request to the server.



At this moment if we set `Offline` mode in the `Network` tab our app will look like this:



Intercept and cache

So far we cached important resources during the installation of a Service Worker, which is known as precaching. This works well when you know exactly the resources that you want to cache, but what about resources that might be dynamic or that you might not know about? For example, our website might be a sports news website that needs constant updating during a match; we won't know about those files during Service Worker installation.

Because Service Workers can intercept HTTP requests, this is the perfect opportunity to make the HTTP request and then store the response in the cache. This means that we will request the resource and then cache it immediately. That way, as the next HTTP request is made for the same resource, we can instantly fetch it out of the Service Worker cache.

Add this in sw.js

```
// Add a new cache for dynamic content
const CACHE_DYNAMIC_NAME = 'dynamic';
self.addEventListener('fetch', event => {
   console.log('[Service Worker] Fetching something ....', event);
   event.respondWith(
       caches.match(event.request)
           .then(response => {
               if (response) {
                   return response;
               }
               // Clone the request - a request is a stream and can be only consumed once
               const requestToCache = event.request.clone();
               return fetch(requestToCache)
                   .then(response => {
                       if (!response || response.status !== 200) {
                           return response;
                       }
                       // Again clone the response because you need to add it into the cache
                       const responseToCache = response.clone();
                       caches.open(CACHE_DYNAMIC_NAME)
                           .then(cache => {
                               cache.put(requestToCache, responseToCache);
                           });
                       return response;
                   })
           })
           .catch(error => console.log('[Service Worker] Dynamic cache error.', error))
   );
});
```

The code above caches the resource fetched from the network and returns it back to the page. If we reload the page, the resources cached in both caches will be matched.

Cache versioning

There will be a point in time where the Service Worker cache will need updating. If we make changes to the web application when need to be sure users receive the newer version of files instead of older versions. As you can imagine, serving older files by mistake would cause havoc on a site.

The great thing about Service Workers is that each time we make any changes to the Service Worker file itself, it automatically triggers the Service Worker update flow. In step 3, we looked at the Service Worker lifecycle. Remember that when a user navigates to the site, the browser tries to re-download the Service Worker in the background. If there's even a byte's difference in the Service Worker file compared to what it currently has, it considers it new.

This useful functionality gives you the perfect opportunity to update your cache with new files. The best things to change are the `cache` names by adding a version to them. But if we update cache name, this would automatically create a new cache and start serving your files from that cache. The original cache would be orphaned and no longer used, and we need to delete it. The best place to delete after the old Service Worker is the `activate` event.

In `sw.js` first add `_v1` to both cache names:

```
const CACHE_STATIC_NAME = 'static_v1';
const CACHE_DYNAMIC_NAME = 'dynamic_v1';
```

Add this in sw.js

```
self.addEventListener('activate', event => {
 console.log('[Service Worker] Activating Service Worker ...', event);

 event.waitUntil(
     caches.keys()
         .then(cacheNames => {
             return Promise.all(cacheNames.map(cacheName => {
                 if (cacheName !== CACHE_STATIC_NAME && cacheName !== CACHE_DYNAMIC_NAME) {
                     console.log('[Service Worker] Removing old cache.', cacheName);
                     return caches.delete(cacheName);
                 }
             }));
         })
         .then(() => {
             console.log('[ServiceWorker] Claiming clients');
             return self.clients.claim();
         })
 );
});
```

Custom *offline* page

Even if we cache the help page when we navigate to it, we may happen to become offline before it gets cached, and in this case, we get the browser's default offline page. To account for this case, we can create our own ***offline.html*** page:

1. Duplicate the *index.html* file and name the duplicate *offline.html*
2. Replace the main tag with:

```
<main class="mdl-layout__content mat-typography">
 <div class="page-content">
  <h5 class="text-center mdl-color-text--primary">We're sorry, this page hasn't been cached yet :/</h5>
  <p>But why don't you try one of our <a href="/fe-guild-2019-pwa/">other pages</a>?</p>
  </div>
</main>
```

- Add the offline.html to the list of files to be ***precached***.
- ***Increase the version*** of the *static* cache.
- Then inside the ***fetch*** handler replace the:

```
.catch(error => console.log('[Service Worker] Dynamic cache error.', error)));
```

with:

```
.catch(error => {
 return caches.open(CACHE_STATIC_NAME)
   .then(cache => {
     if (event.request.headers.get('accept').includes('text/html')) {
       return cache.match('/fe-guild-2019-pwa/offline.html');
     }
   });
})
```

- Finally refresh the application, manually delete the help-page entry in the dynamic cache if you have it.
- **Go offline** and try to navigate to the help page. You should see the new ***offline page*** being displayed.

Exercises

1. Experiment with the code in this step
2. Try to do it in stages in order to see the differences
3. (Optional) Deploy the app on GitHub Pages

Solution

```
$ git checkout pwa-service-workers-final
```