Lab Handout!
(Fetch, Manifest File, Service Workers): Building a PWA Step by Step!

Introduction:

A **Progressive Web App** (PWA) is a web app that delivers an app-like experience to users by using modern web capabilities. In the end, it's just your regular website that runs in a browser with some enhancements like the ability:

1. To install it on a mobile home screen
2. To access it when offline
3. To access the camera
4. Get push notifications
5. To do background synchronization
6. And so much more.

However, to be able to transform our traditional web app to a PWA, we have to adjust it a little bit, by adding a web app **manifest file** and a **service worker**.

STEP # 1: Defining the Mark-up Page (using HTML)

The HTML file is relatively simple. We wrap everything on the main tag. (**index.html**) - And create a navigation bar with the nav tag. Then, the div with the class .container will hold later our cards added by JavaScript.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <link rel="stylesheet" href="css/style.css" />
    <title>Dev'Coffee PWA</title>
  </head>
  <body>
    <main>
      <nav>
        <h1>Dev'Coffee</h1>
        <ul>
          <li>Home</li>
          <li>About</li>
          <li>Blog</li>
        </ul>
      </nav>
      <div class="container"></div>
    </main>
    <script src="js/app.js"></script>
  </body>
</html>
```

STEP # 2: Styling the page: Defining CSS

To make our index.html page give better look and feel. We define our (**css/style.css)** as follows:

```
@import
url("https://fonts.googleapis.com/css?family=Nunito:400,700&display=swap");
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
```

```
        }
        body {
          background: #fdfdfd;
          font-family: "Nunito", sans-serif;
          font-size: 1rem;
        }
        main {
          max-width: 900px;
          margin: auto;
          padding: 0.5rem;
          text-align: center;
        }
        nav {
          display: flex;
          justify-content: space-between;
          align-items: center;
        }
        ul {
          list-style: none;
          display: flex;
        }

        li {
          margin-right: 1rem;
        }
        h1 {
          color: #e74c3c;
          margin-bottom: 0.5rem;
        }
```

Then, we limit the main element's maximum width to 900px, to make it look good on a large screen.

For the navbar, lets place the logo to be at the left and the links at the right. Hence, for the nav tag, after making it a flex container, we use justify-content: space-between; to align them.

```
        .container {
          display: grid;
          grid-template-columns: repeat (auto-fit, minmax (15rem, 1fr));
          grid-gap: 1rem;
          justify-content: center;
          align-items: center;
          margin: auto;
          padding: 1rem 0;
        }
        .card {
          display: flex;
          align-items: center;
          flex-direction: column;
          width: 15rem auto;
          height: 15rem;
          background: #fff;
          box-shadow: 0 10px 20px rgba(0, 0, 0, 0.19), 0 6px 6px rgba(0, 0, 0, 0.23);
          border-radius: 10px;
          margin: auto;
          overflow: hidden;
        }
        .card--avatar {
          width: 100%;
          height: 10rem;
          object-fit: cover;
        }
        .card--title {
          color: #222;
          font-weight: 700;
          text-transform: capitalize;
          font-size: 1.1rem;
          margin-top: 0.5rem;
        }
        .card--link {
          text-decoration: none;
```

```
  background: #db4938;
  color: #fff;
  padding: 0.3rem 1rem;
  border-radius: 20px;
}
```

We'll have several cards, so, for the container element it will be displayed as a grid. And, with grid-template-columns: repeat (auto-fit, minmax (15rem, 1fr)), we can now make our cards responsive and let them use at least 15rem as width if there is enough space and 1fr if not.

And to make them look nice we double the shadow effect on **.card** class and use object-fit: cover on .card--avatar to prevent the image stretching. So, now it looks much better but we still not have data to show.

STEP # 3: Showing Data with JavaScript

The **.container** class will hold our cards. Therefore, we need to select it in (**js/app.js**)

```
const container = document.querySelector(".container")
const coffees = [
  { name: "Perspiciatis", image: "images/coffee1.jpg" },
  { name: "Voluptatem", image: "images/coffee2.jpg" },
  { name: "Explicabo", image: "images/coffee3.jpg" },
  { name: "Rchitecto", image: "images/coffee4.jpg" },
  { name: " Beatae", image: "images/coffee5.jpg" },
  { name: " Vitae", image: "images/coffee6.jpg" },
  { name: "Inventore", image: "images/coffee7.jpg" },
  { name: "Veritatis", image: "images/coffee8.jpg" },
  { name: "Accusantium", image: "images/coffee9.jpg" },
]
```

Now we create an array of cards with names and images in (**js/app.js**)

```
const showCoffees = () => {
  let output = ""
  coffees.forEach(
    ({ name, image }) =>
      (output += `
              <div class="card">
                <img class="card--avatar" src=${image} />
                <h1 class="card--title">${name}</h1>
                <a class="card--link" href="#">Taste</a>
              </div>
              `)
  )
  container.innerHTML = output
}

document.addEventListener("DOMContentLoaded", showCoffees)
```

With this code above, we can now loop through the array and show them on the HTML file. And to make everything work, we wait until the DOM (Document Object Model) content finished load to run the showCoffees method. We've done a lot, but for now, we just have a traditional web app. Now let's apply the PWA features.

Step # 4: Applying PWA features: Web App Manifest

The web app manifest is a simple JSON file that informs the browser about your web app and how it should behave when installed on the user's mobile device or desktop. And to show the Add to Home Screen prompt, the web app manifest is required.

Now we know, what a web manifest is, let's create a new file named **manifest.json** (you've to name it like that) in the root directory, and add this code block below.

```
{
  "name": "Dev'Coffee",
  "short_name": "DevCoffee",
  "start_url": "index.html",
  "display": "standalone",
  "background_color": "#fdfdfd",
  "theme_color": "#db4938",
  "orientation": "portrait-primary",
  "icons": [
    {
      "src": "/images/icons/icon-72x72.png",
      "type": "image/png", "sizes": "72x72"
    },
    {
      "src": "/images/icons/icon-96x96.png",
      "type": "image/png", "sizes": "96x96"
    },
    {
      "src": "/images/icons/icon-128x128.png",
      "type": "image/png","sizes": "128x128"
    },
    {
      "src": "/images/icons/icon-144x144.png",
      "type": "image/png", "sizes": "144x144"
    },
    {
      "src": "/images/icons/icon-152x152.png",
      "type": "image/png", "sizes": "152x152"
    },
    {
      "src": "/images/icons/icon-192x192.png",
      "type": "image/png", "sizes": "192x192"
    },
    {
      "src": "/images/icons/icon-384x384.png",
      "type": "image/png", "sizes": "384x384"
    },
    {
      "src": "/images/icons/icon-512x512.png",
      "type": "image/png", "sizes": "512x512"
    }
  ]
}
```

In the end, it's just a JSON file with some mandatory and optional properties.

- **name**: When the browser launches the splash screen, it will be the name displayed on the screen.
- **short_name**: It will be the name displayed underneath your app shortcut on the home screen.
- **start_url**: It will be the page shown to the user when your app is open.
- **display**: It tells the browser how to display the app. They are several modes like minimal-ui, fullscreen, browser etc. Here, we use the standalone mode to hide everything related to the browser.
- **background_color**: When the browser launches the splash screen, it will be the background of the screen.

- **theme_color**: It will be the background color of the status bar when we open the app.
- **orientation**: It tells the browser the orientation to have when displaying the app.
- **icons**: When the browser launches the splash screen, it will be the icon displayed on the screen. Here, I used all sizes to fit any device's preferred icon. But you can just use one or two. It's up to you.

Now, we have a web app manifest, let's add it to the HTML file (**index.html**)

```
<link rel="manifest" href="manifest.json" />
<!-- ios support -->
<link rel="apple-touch-icon" href="images/icons/icon-72x72.png" />
<link rel="apple-touch-icon" href="images/icons/icon-96x96.png" />
<link rel="apple-touch-icon" href="images/icons/icon-128x128.png" />
<link rel="apple-touch-icon" href="images/icons/icon-144x144.png" />
<link rel="apple-touch-icon" href="images/icons/icon-152x152.png" />
<link rel="apple-touch-icon" href="images/icons/icon-192x192.png" />
<link rel="apple-touch-icon" href="images/icons/icon-384x384.png" />
<link rel="apple-touch-icon" href="images/icons/icon-512x512.png" />
<meta name="apple-mobile-web-app-status-bar" content="#db4938" />
<meta name="theme-color" content="#db4938" />
```

We linked our **manifest.json** file to the head tag. And add some other links which handle the IOS support to show the icons and colorize the status bar with our theme color.

Step # 5: Applying PWA features: Introducing Service Worker.

NOTE: *PWAs run only on https because the service worker can access to the request and handle it. Therefore, the security is required.*

A **service worker** is a script that your browser runs in the background in a separate thread. That means it runs in a different place, it's completely separates from your web page. That's the reason why it can't manipulate your DOM element. However, it's super powerful!

The service worker can intercept and handle network requests, manage the cache to enable offline support or send push notifications to your users.

To do this: Create a service worker and place it in the root folder and named it **serviceWorker.js** (the name is up to you). But you have to put it in the root to not limit its scope to one folder.

5.1- Cache the assets (in serviceWorker.js) – See below:

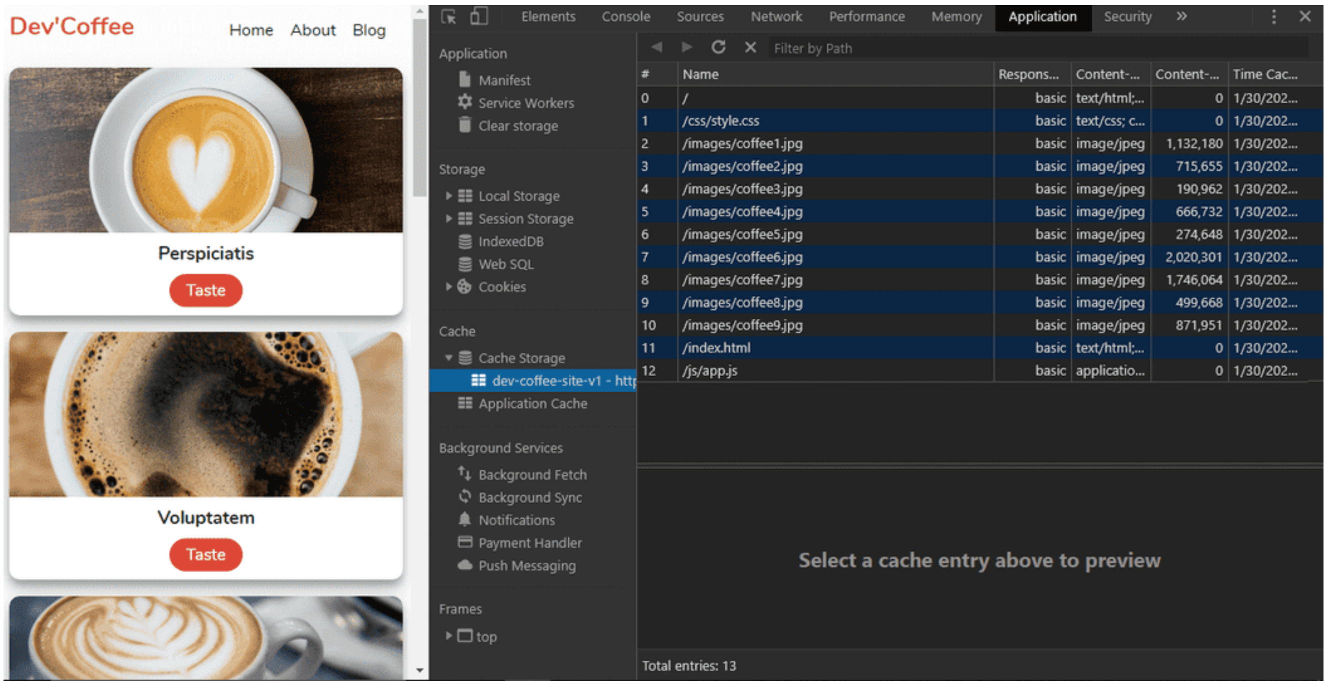
```
const staticDevCoffee = "dev-coffee-site-v1"
const assets = [
  "/",
  "/index.html",
  "/css/style.css",
  "/js/app.js",
  "/images/coffee1.jpg",
  "/images/coffee2.jpg",
  "/images/coffee3.jpg",
  "/images/coffee4.jpg",
  "/images/coffee5.jpg",
  "/images/coffee6.jpg",
  "/images/coffee7.jpg",
  "/images/coffee8.jpg",
  "/images/coffee9.jpg",
]

self.addEventListener("install", installEvent => {
  installEvent.waitUntil(
    caches.open(staticDevCoffee).then(cache => {
      cache.addAll(assets)
    })
  )
})
```

We declare the name of our cache **staticDevCoffee** and the assets to store in the cache. And to perform that action, we need to attach a listener to **self**.

self is the service worker itself. It enables us to listen to life cycle events and do something in return. The service worker has several life cycles, and one of them is the install event. It runs when a service worker is installed. It's triggered as soon as the worker executes, and it's only called once per service worker. When the install event is fired, we run the callback which gives us access to the event object.

Caching something on the browser can take some time to finish because it's asynchronous. So to handle it, we need to use waitUntil() to as you might guess, waiting for the action to finish. Once the cache API ready, we can now run the open() method and create our cache by passing its name as an argument to caches.open(staticDevCoffee). Then, it returns a promise, which helps us store our assets in the cache with cache.addAll(assets).



#	Name	Respons...	Content-...	Content-...	Time Cac...
0	/	basic	text/html;...	0	1/30/202...
1	/css/style.css	basic	text/css; c...	0	1/30/202...
2	/images/coffee1.jpg	basic	image/jpeg	1,132,180	1/30/202...
3	/images/coffee2.jpg	basic	image/jpeg	715,655	1/30/202...
4	/images/coffee3.jpg	basic	image/jpeg	190,962	1/30/202...
5	/images/coffee4.jpg	basic	image/jpeg	666,732	1/30/202...
6	/images/coffee5.jpg	basic	image/jpeg	274,648	1/30/202...
7	/images/coffee6.jpg	basic	image/jpeg	2,020,301	1/30/202...
8	/images/coffee7.jpg	basic	image/jpeg	1,746,064	1/30/202...
9	/images/coffee8.jpg	basic	image/jpeg	499,668	1/30/202...
10	/images/coffee9.jpg	basic	image/jpeg	871,951	1/30/202...
11	/index.html	basic	text/html;...	0	1/30/202...
12	/js/app.js	basic	applicatio...	0	1/30/202...

Total entries: 13

After successfully cached our assets on the browser. And the next time we load the page, the service worker will handle the request and fetch the cache if we are offline. So, let's fetch our cache.

5.2-Fetch the Assets (in serviceWorker.js) – See below:

```
  self.addEventListener("fetch", fetchEvent => {
    fetchEvent.respondWith(
      caches.match(fetchEvent.request).then(res => {
        return res || fetch(fetchEvent.request)
      })
    )
  })
```

Here, we use the fetch event to, well, get back our data. The callback gives us access to fetchEvent, then we attach respondWith() to prevent the browser's default response and instead it returns a promise. Because the fetch action can take time to finish.

And once the cache ready, we apply the caches.match(fetchEvent.request). It will check if something in the cache matches fetchEvent.request. By the way, fetchEvent.request is just our array of assets.

Then, it returns a promise, and finally, we can return the result if it exists or the initial fetch if not.

Now, our assets can be cached and fetched by the service worker which increases a lot the load time of our images. And most important, it makes our app available on offline mode.

But a service worker only can't do the job, we need to register it in our project.

5.3-Registering the Service Worker (in js/app.js) – See below:

```
if ("serviceWorker" in navigator) {
  window.addEventListener("load", function() {
    navigator.serviceWorker
      .register("/serviceWorker.js")
      .then(res => console.log("service worker registered"))
      .catch(err => console.log("service worker not registered", err))
  })
}
```

Here, we start by checking if the serviceWorker is supported by the current browser. Because it's still not supported by all browsers.

Then, we listen to the page load event to register our service worker by passing the name of our file serviceWorker.js to navigator.serviceWorker.register() as a parameter to register our worker.

The above Lab Tutorial helps you to convert your regular web app to a PWA.

Notes: (To make PWA works and particularly service workers, you need a server. If you deploy your app - Try using VS Code Live Server to make your service worker work)