Start 8/2/2022

pwa-with-vanilla-js-summary :

blog link (free code camp):

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

Source code :

<https://github.com/ibrahima92/pwa-with-vanilla-js>

-**Purpose** :

In this article, we are going to build a PWA from scratch with HTML, CSS, and JavaScript

**Summary from previous link** :

**What is a Progressive Web App ?**

A Progressive Web App is a web app that gives you the ability:

-To install it on a mobile home screen

-To access it when offline

-To access the camera

-To get push notifications

-To do background synchronization

- How to convert traditional web app to a PWA ?

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**.

**-Show data with JavaScript :**

Notice that I used large images that take some time to load. This will show you in the best way the power of service workers.

// index.html

<!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>

// 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" },

]

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)

And to make everything work, we wait until the DOM (Document Object Model) content finishes loading to run the showCoffees method.

// css\style.css

*@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;

}

*.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;

    width: 85%;

}

*.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;

}

-**Web App Manifest** :

The web app manifest is a simple JSON file that informs the browser about your web app.

It tells 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.

-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. There 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.

//manifest.json

{

    "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"

        }

    ]

}

* Now that we have a web app manifest, let's add it to the HTML file.

In index.html (head tag)

//index.html

<head>

    <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" />

</head>

-here we add some other links for IOS which handle the iOS support to show the icons and colorize the

-**What is a Service Worker?**

-Notice that PWAs run only on **https** because the service worker can access the request and handle it. Therefore 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 and is completely separate from your web page. That's the reason why it can't manipulate يؤثر your DOM element.

However, it's super powerful. The service worker can intercept يقاطع and handle network requests, manage the cache to enable offline support or send push notifications to your users.

// serviceWorker.js (all code )

*// the name of the cashe*

 const staticDevCoffee = "dev-coffee-site-v1"

*// the assets to store in the cache.*

 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",

 ]

*// add assets to cache before install*

 self.addEventListener("install", installEvent => {

     installEvent.waitUntil(

         caches.open(staticDevCoffee).then(cache => {

             cache.addAll(assets)

         })

     )

 })

*/\**

*- 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.*

*- Caching something on the browser can take some time to finish because it's asynchronous.*

*So to handle it, we need to use waitUntil() which, as you might guess, waits for the action to finish.*

*\*/*

*//Fetch the assets that we cache during install*

 self.addEventListener("fetch", fetchEvent => {

     fetchEvent.respondWith(

         caches.match(fetchEvent.request).then(res => {

*return* res || fetch(fetchEvent.request)

         })

     )

 })

*/\**

*-  we attach respondWith() to prevent the browser's default response. 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.*

*\*/*

**Cache the assets :**

 self.addEventListener("install", installEvent => {

     installEvent.waitUntil(

         caches.open(staticDevCoffee).then(cache => {

             cache.addAll(assets)

         })

     )

 })

- 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.

- Caching something on the browser can take some time to finish because it's asynchronous.

So to handle it, we need to use **waitUntil**() which, as you might guess, waits for the action to finish.

**Fetch the assets :**

*//Fetch the assets that we cache during install*

 self.addEventListener("fetch", fetchEvent => {

     fetchEvent.respondWith(

         caches.match(fetchEvent.request).then(res => {

*return* res || fetch(fetchEvent.request)

         })

     )

 })

- we attach respondWith() to prevent the browser's default response. 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.

**Register the Service Worker :**

-But a service worker alone can't do the job. We need to register it in our project.

//js\app.js

*// register service worker*

*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 (as it's still not supported by all browsers).*

*- after register service worker => With this update, we have now transformed our regular web app to a PWA.*

*\*/*

* Note :

When work by live server like :

<http://127.0.0.1:5500/> the previous code work But when work by Xampp Or deploy to server like dev account there is some errors like images on manifest not read and server worker not registered

So we will edit the code by the following : ( // we add . if we deploy on sub domain )

//manifest.json

  "icons": [{

            "src": "./images/icons/icon-72x72.png", *// we add . if we deploy on sub domain*

            "type": "image/png",

            "sizes": "72x72"

        },

//js\app.js

*// register service worker*

*if* ("serviceWorker" in navigator) {

    window.addEventListener("load", function() {

        navigator.serviceWorker

            .register("./serviceWorker.js") *//   we add . if we deploy on sub domain*

            .then(res => console.log("service worker registered"))

            .catch(err => console.log("service worker not registered", err))

    })

}

//serviceWorker.js

*// the assets to store in the cache.*

 const assets = [

     "/",

     "./index.html", *//  we add . if we deploy on sub domain*

     "./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",

 ]

**Benefits of PWAs over native app :**

<https://www.freecodecamp.org/news/what-are-progressive-web-apps/>

-A lot of organizations both private and public are switching to PWAs not only because they are cheap to develop but also because they offer greater engagement.

Now let's look at a quick summary of the benefits of a PWA:

They are responsive and work with many different screen sizes.

They function just like normal Native Apps.

The updates are independent, you don't need to visit the play store for an update.

They're built with common web technologies.

They're fast and lightweight.

They work offline unlike other sites.

They are discoverable via search engine because native apps we can’t make SEO to it and not appear on google search ( it only app on google play or apple store )

They are easily installable.

Low maintenance cost.

تم بحمد الله

9/2/2022