Name: Niyati V. Gaonkar
Class: D15B   Roll no.: 17

MAD-PWD Experiment - 8

AIM: To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA .

THEORY:
Making PWAs work offline with Service workers.
Service Workers are a virtual proxy between the browser and the network. They make it possible to properly cache the assets of a website and make them available when the user's device is offline.
They run on a separate thread from the main JavaScript code of our page, and don't have any access to the DOM structure. This introduces a different approach from traditional web programming — the API is non-blocking, and can send and receive communication between different contexts. You are able to give a Service Worker something to work on, and receive the result whenever it is ready using a Promise-based approach.
Service workers can do more than offering offline capabilities, including handling notifications or performing heavy calculations. Service workers are quite powerful as they can take control over network requests, modify them, serve custom responses retrieved from the cache, or synthesize responses completely.

Websites and Workers:
The foundation of all the technologies we'll discuss in this guide is the service worker. In this section we'll provide a little background about workers and how they change the architecture of a web app.
Normally, an entire website runs in a single thread. This includes the website's own JavaScript, and all the work to render the website's UI. One consequence of this is that if your JavaScript runs some long-running operation, the website's main UI is blocked, and the website appears unresponsive to the user.
A service worker is a specific type of web worker that's used to implement PWAs. Like all web workers, a service worker runs in a separate thread to the main JavaScript code. The main code creates the worker, passing in a URL to the worker's script. The worker and the main code can't directly access each other's state, but can communicate by sending each other messages. Workers can be used to run computationally expensive tasks in the background: because they run in a separate thread, the main JavaScript code in the app, that implements the app's UI, can stay responsive to the user.
So a PWA always has a high level architecture split between:

- The *main app*, with the HTML, CSS, and the part of the JavaScript that implements the app's UI (by handling user events, for example)
- The *service worker*, which handles offline and background tasks

Offline Operation:
Offline operation allows a PWA to provide a good user experience even when the device does not have network connectivity. This is enabled by adding a service worker to an app. A service worker controls some or all of the app's pages. When the service worker is installed, it can fetch the resources from the server for the pages it controls (including pages, styles, scripts, and images, for example) and add them to a local cache. The Cache interface is used to add resources to the cache. Cache instances are accessible through the caches property in the service worker global scope.
Then whenever the app requests a resource (for example, because the user opened the app or clicked an internal link), the browser fires an event called fetch in the service worker's global scope. By listening for this event, the service worker can intercept the request.
The event handler for the fetch event is passed a FetchEvent object, which:

- Provides access to the request as a Request instance
- Provides a respondWith() method to send a response to the request.

One way a service worker can handle requests is a "cache-first" strategy. In this strategy:

- If the requested resource exists in the cache, get the resource from the cache and return the resource to the app.
- If the requested resource does not exist in the cache, try to fetch the resource from the network.
- If the resource could be fetched, add the resource to the cache for next time, and return the resource to the app.
- If the resource could not be fetched, return some default fallback resource.

Registering the Service Worker
We'll start by looking at the code that registers a new Service Worker, in the app.js file:

JS
```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("./pwa-examples/js13kpwa/sw.js");
}
```
If the service worker API is supported in the browser, it is registered against the site using the ServiceWorkerContainer.register() method. Its contents reside in the sw.js file, and can be executed after the registration is successful. It's the only piece of Service Worker code that sits inside the app.js file; everything else that is Service Worker-specific is written in the sw.js file itself.

Lifecycle of a Service Worker
When registration is complete, the sw.js file is automatically downloaded, then installed, and finally activated.

Installation
The API allows us to add event listeners for key events we are interested in — the first one is the install event:
JS
```
self.addEventListener("install", (e) => {
  console.log("[Service Worker] Install");
});
```
In the install listener, we can initialize the cache and add files to it for offline use. Our js13kPWA app does exactly that.
First, a variable for storing the cache name is created, and the app shell files are listed in one array.
JS
```
const cacheName = "js13kPWA-v1";
const appShellFiles = [
  "/pwa-examples/js13kpwa/",
  "/pwa-examples/js13kpwa/index.html",
]
```

Next, the links to images to be loaded along with the content from the data/games.js file are generated in the second array. After that, both arrays are merged using the Array.prototype.concat() function.
JS
```
const gamesImages = [];
for (let i = 0; i < games.length; i++) {
  gamesImages.push(`data/img/${games[i].slug}.jpg`);
}
const contentToCache = appShellFiles.concat(gamesImages);
```

Then we can manage the install event itself:
JS
```
self.addEventListener("install", (e) => {
  console.log("[Service Worker] Install");
  e.waitUntil(
    (async () => {
      const cache = await caches.open(cacheName);
      console.log("[Service Worker] Caching all: app shell and content");
      await cache.addAll(contentToCache);
    })(),
  );
});
```

There are two things that need an explanation here: what ExtendableEvent.waitUntil does, and what the caches object is.
The service worker does not install until the code inside waitUntil is executed. It returns a promise — this approach is needed because installing may take some time, so we have to wait for it to finish.
caches is a special CacheStorage object available in the scope of the given Service Worker to enable saving data — saving to web storage won't work, because web storage is synchronous. With Service Workers, we use the Cache API instead.
Here, we open a cache with a given name, then add all the files our app uses to the cache, so they are available next time it loads. Resources are identified by their request URL, which is relative to the worker's location.
You may notice we haven't cached game.js. This is the file that contains the data we use when displaying our games. In reality this data would most likely come from an API endpoint or database and caching the data would mean updating it periodically when there was network connectivity. We won't go into that here, but the Periodic Background Sync API is good further reading on this topic.

IMPLEMENTATION:
Index.html:

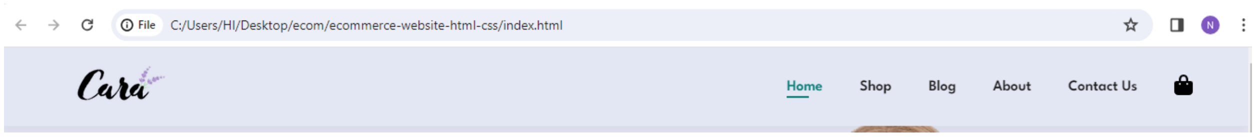
```
<body>
<-- Rest code —>
<script>
      if ('serviceWorker' in navigator) {
          window.addEventListener('load', () => {
              navigator.serviceWorker.register('/service-worker.js')
                  .then(registration => {
                      console.log('Service Worker registered with scope:', registration.scope);
                  })
                  .catch(error => {
                      console.error('Service Worker registration failed:', error);
                  });
          });
      }
  </script>
</body>
```

service-worker.js:

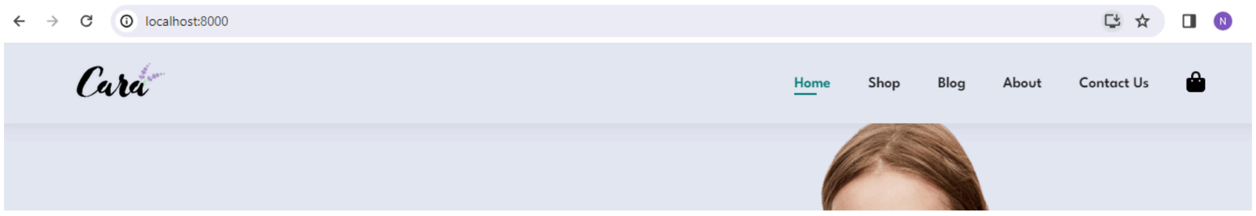
```
const CACHE_NAME = 'ecommerce-v1';
const urlsToCache = [
    '/',
    'index.html',
    'style.css',
    'fonts.js',
    'icons/bag.svg',
    'script.js',
    'images/button.png',
    'images/hero4.png',
    'images/logo.png',
    'images/about/a1.png',
    'images/about/a2.jpg',
    'images/about/a3.png',
    'images/about/a4.png',
    'images/about/a5.jpg',
    'images/about/a6.jpg',
    'images/about/banner.png',
    'images/banner/b1.jpg',
    'images/banner/b2.jpg',
    'images/banner/b4.jpg',
    'images/banner/b7.jpg',
    'images/banner/b10.jpg',
    'images/banner/b14.png',
    'images/banner/b16.jpg',
    'images/banner/b17.jpg',
    'images/banner/b18.jpg',
    'images/banner/b19.jpg',
    'images/banner/b20.jpg',
    'images/blog/b1.jpg',
    'images/blog/b2.jpg',
    'images/blog/b3.jpg',
    'images/blog/b4.jpg',
    'images/blog/b5.jpg',
    'images/blog/b6.jpg',
    'images/blog/b7.jpg',
    'images/features/f1.png',
    'images/features/f2.png',
    'images/features/f3.png',
    'images/features/f4.png',
    'images/features/f5.png',
    'images/features/f6.png',
    'images/pay/app.jpg',
    'images/pay/pay.png',
    'images/pay/play.jpg',
    'images/people/1.png',
    'images/people/2.png',
    'images/people/3.png',
    'images/products/f1.jpg',
    'images/products/f2.jpg',
    'images/products/f3.jpg',
    'images/products/f4.jpg',
    'images/products/f5.jpg',
    'images/products/f6.jpg',
    'images/products/f7.jpg',
    'images/products/f8.jpg',
    'images/products/n1.jpg',
    'images/products/n2.jpg',
    'images/products/n3.jpg',
    'images/products/n4.jpg',
    'images/products/n5.jpg',
    'images/products/n6.jpg',
    'images/products/n7.jpg',
    'images/products/n8.jpg',
];

self.addEventListener('install', event => {
    event.waitUntil(
        caches.open(CACHE_NAME)
            .then(cache => {
                return cache.addAll(urlsToCache);
            })
    );
});

self.addEventListener('activate', event => {
    event.waitUntil(
        caches.keys().then(cacheNames => {
            return Promise.all(
                cacheNames.map(cacheName => {
                    if (cacheName !== CACHE_NAME) {
                        return caches.delete(cacheName);
                    }
                })
            );
        })
    );
});

self.addEventListener('fetch', event => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                if (response) {
                    return response;
                }

                return fetch(event.request)
                    .then(response => {
                        if (!response || response.status !== 200 ||
response.type !== 'basic') {
                            return response;
                        }

                        const responseToCache = response.clone();
                        caches.open(CACHE_NAME)
                            .then(cache => {
                                cache.put(event.request, responseToCache);
                            });

                        return response;
                    });
            })
    );
});
```

OUTPUT:

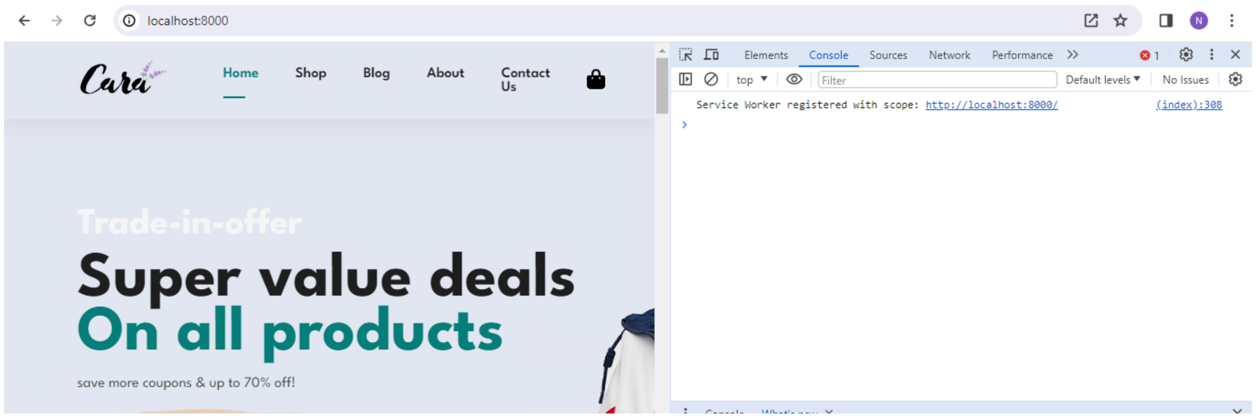
Before:


C:/Users/HI/Desktop/ecom/ecommerce-website-html-css/index.html
Cara
Home
Shop
Blog
About
Contact Us

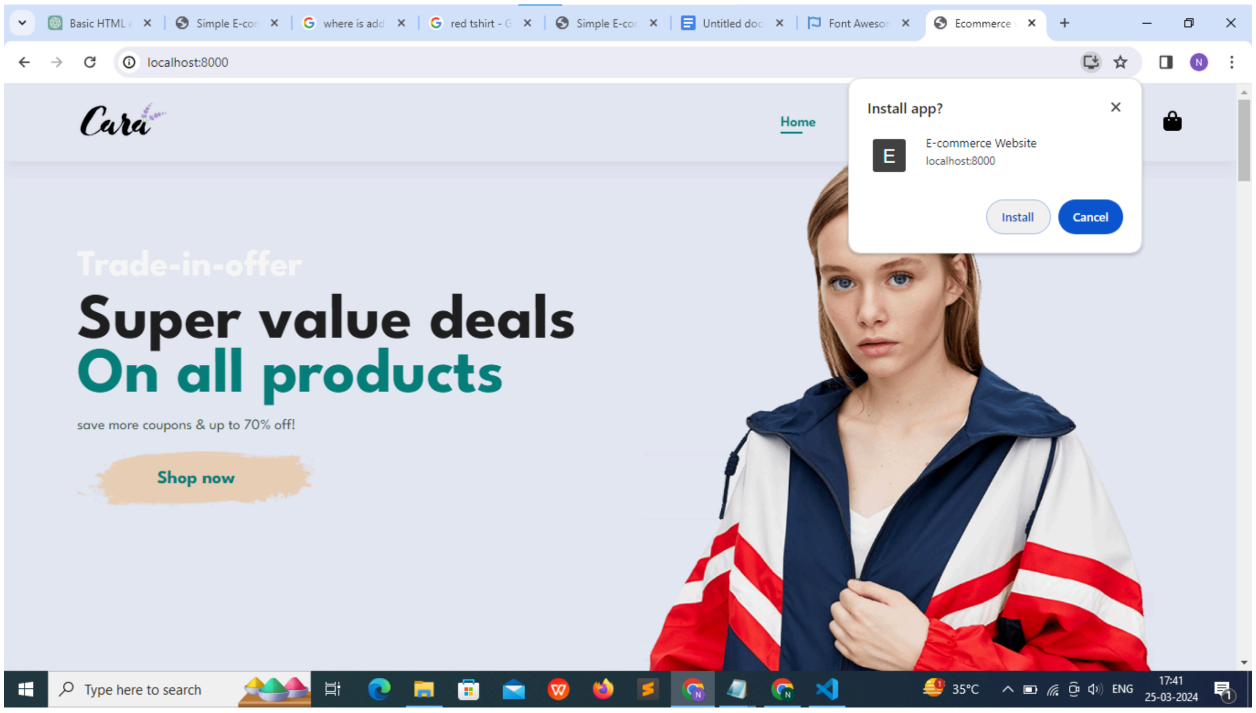
After: the download web app option



Service worker

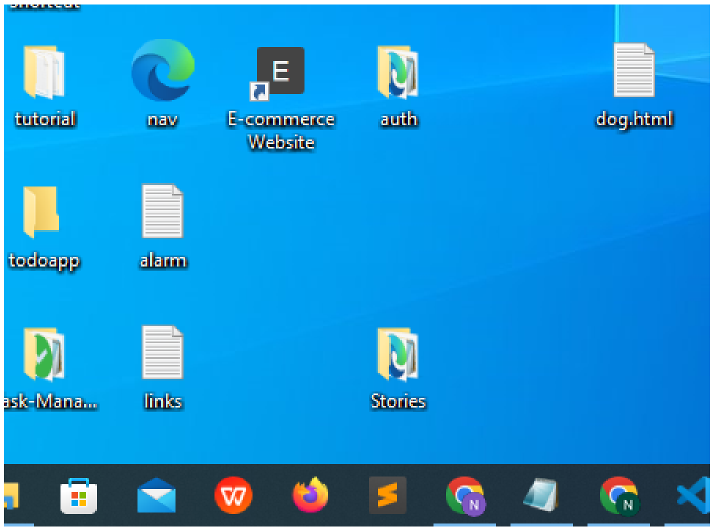


```
Service Worker registered with scope: http://localhost:8000/          (index):308
```



Install app?

E-commerce Website
localhost:8000



CONCLUSION: Successfully implementing a Progressive Web App (PWA), we've downloaded it onto our desktop. Confirming registration in the console, our app enables offline access, ensuring our website remains accessible even without an internet connection. PWA adoption enhances user experience by providing seamless offline functionality, a hallmark of modern web development.