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:

Service Worker

A service worker is a web worker. It is a JavaScript file that runs aside from the mobile browser. In other words, it is another technical component that promotes the functionality of PWA. The service worker retrieves the resources from the cache memory and delivers the messages.
It is independent of the application to which they are connected, and has many consequences:

- The service worker does not block the synchronized XHR, so it cannot be used in local storage (It is designed to be completely asynchronous).
- It can receive information from the server even when the application is not running. It shows notifications in the PWA application without opening in the mobile browser.
- It cannot directly access the DOM. Therefore, the PostMessage and Message Event Listener method is used to communicate with the webpage. The PostMessage method is used to send data, and the message event listener is used to receive data.

Things to understand about it:

- It is a programmable-network proxy that helps you monitor how your page handles network requests.
- It only works on the HTTPS because HTTPS is very secure, and it intercepts network requests and modifies responses.

Service worker lifecycle

The service worker lifecycle is the most complex part of the PWA. There are three stages in the lifetime of a service worker:

1. Registration
2. Installation
3. Activation

Registration

A service worker is basically a JavaScript file. You need to register it in your crucial JavaScript code to use a service worker. Registration tells your browser location of the service worker and starts installing it on the background. One thing that distinguishes a service worker file from a standard JavaScript file is that a service worker runs in the background away from the mobile's main browser. This process is the first phase of the service worker's lifecycle.
The code of service worker registration is placed in the main.js file.

```
if ("Service-Worker- in navigator.js)
{
 navigator. Service - worker. register ('/service-worker. json')
 then (function (registration)
 {
  console.log ('Registration successful finish, scope is:', registration. scope.);
 }
 Catch (function (error)
 {
  console.log ('Registration failed, error:', error.js);
 }
}
```

First, this code checks whether the browser supports the service worker. The service worker is then registered with navigator.serviceWorker.register when the service worker returns a promise. If the promise is fulfilled, the registration is successful; otherwise, the registration is failed.
Scope of registration
This scope determines the web pages that are managed by a service worker. The location of the service worker defines the default scope. Whenever you register a service worker file at the main folder of the system, it is not important to specify its scope, because it controls all webpages.

1. navigator.service.worker.register ('/sw.js', { scope: '/' });

Installation
When a new service worker is registered with the help of navigator.serviceWorker.register, the JavaScript code is downloaded, and the installation state is entered. If the installation succeeds, the service worker further proceeds to the next state.

```
const assets to.cache = [
'/js/app.js',
'/about.html',
'/index.html',
'/css/app.css',
]

self. addEventListener ('install', function (event) {
   event. wait Until (
      caches. Open('staticAssetsCache')
      then (function (cache) {
          return cache Add All (assetsToCache.);
      })
```

```
    );
});
```

Activation

Once the service worker is successfully installed, it converts to the activation phase. If there is an open page controlled by the previous service worker, the new service worker enters the waiting state. The new service worker is activated only when no pages are loaded in the old service worker. A service worker is not activated immediately after installation.

A service worker will only be active in these cases:

- If no service worker currently active.
- If the user refreshes the webpage.

Service-worker.js

```
self.addEventListener('activate', function(event) {
  // Perform some task
});
```

The service worker can manage network requests rather than caching. It roves around the latest Internet API.

1. Fetch: The Fetch API is a basic resource of the GUI. It makes it easier to control webpage requests and responses than older XMLHttprequests, and this often needs extra syntax, and its example is controlling the redirects. When a resource is requested on the network than the fetal event is terminated.

Note: It supports the CORS (Cross-Origin Resource Sharing). A local server is usually required for testing.

Fetch request example:

```
fetch('ABCs/ABC.json')
then (function (response){
// response function
})
catch (function (error)
{
console.log ('problem section: \n', error);
});
```

2. Cache API: A cache interface has been provided for the service worker API, which allows you to create a repository of responses as requested. However, this interface was designed for service workers. It does not update the memory in the cache unless specifically requested.

Features of the service worker

- **Offline:** Enabling offline functionality is possibly the most demanding service worker facility.
- **Caching:** The control of cache content is the most common feature for service workers.
- Content delivery networks: The CDN and other external material may be difficult to handle. The PWA developers sometimes select out of publicly hosted software due to same-origin rules and SSL, but you may still upload scripts from CDNs.
- **Push notifications:** The feature of push notifications is the best way to interact with users and visitors. This feature enhances the performance of the PWA application.
- **Background synchronization:** It is another very important feature of a service worker that synchronizes tasks in the background.

4. Webpack

The webpack is the fourth component of the PWA. It is used to design the PWA front-end. It allows the PWA-developers to gather all JavaScript resources and data in one location.

5. Transport Layer Security (TLS)

The transport layer security is the fifth component of the PWA. This component is a standard for all robust and secure data exchange between any two applications. The integrity of the data requires the website's service through the HTTPS and an SSL certificate installed on the server.

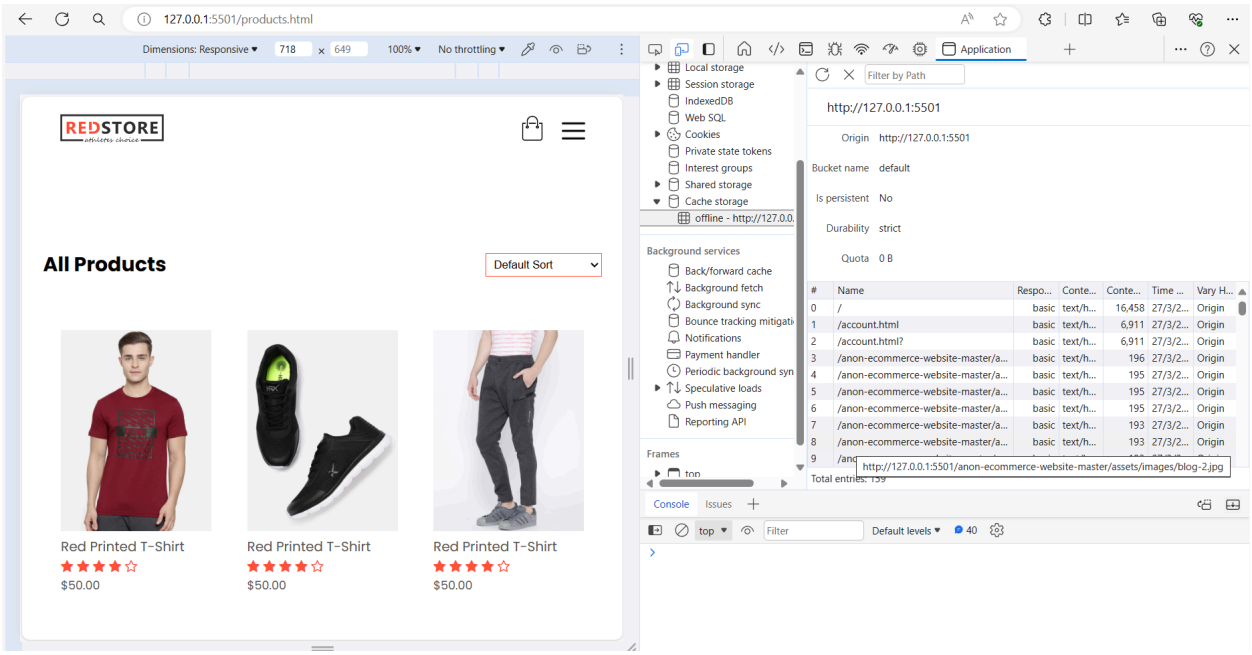
service-worker.js

```
self.addEventListener("install", function (event) {
    event.waitUntil(preLoad());
});

self.addEventListener('sync', event => {
    if (event.tag === 'Sync from cache') {
        console.log("Sync successful!")
    }
});

self.addEventListener('push', function (event) {
    if (event && event.data) {
        var data = event.data.json();
        if (data.method == "PushMessageData") {
            console.log("Push notification sent");
            event.waitUntil(self.registration.showNotification("RED
STORE", { body: data.message }))
        }
    }
});
```

```
var filesToCache = [
    '/index.html',
    '/product_details.html',
    '/products.html',
    '/cart.html',
    '/account.html'
];

var preLoad = function () {
    return caches.open("offline").then(function (cache) { // caching index
and important routes
        return cache.addAll(filesToCache);
    });
};

self.addEventListener("fetch", function (event) {
    event.respondWith(checkResponse(event.request).catch(function () {
        return returnFromCache(event.request);
    }));
    event.waitUntil(addToCache(event.request));
});

var checkResponse = function (request) {
    return new Promise(function (fulfill, reject) {
        fetch(request).then(function (response) {
            if (response.status !== 404) {
                fulfill(response);
            } else {
                reject();
            }
        }, reject);
    });
};

var addToCache = function (request) {
    return caches.open("offline").then(function (cache) {
        return fetch(request).then(function (response) {
            return cache.put(request, response);
        });
    });
```

```
};

var returnFromCache = function (request) {
    return caches.open("offline").then(function (cache) {
        return cache.match(request).then(function (matching) {
            if (!matching || matching.status == 404) {
                return cache.match("index.html");
            } else {
                return matching;
            }
        });
    });
};

// Add event listener for beforeinstallprompt
self.addEventListener('beforeinstallprompt', event => {
    event.preventDefault();
    const installButton = document.getElementById('install-button');
    if (installButton) {
        installButton.style.display = 'block';
        installButton.addEventListener('click', () => {
            event.prompt();
        });
    }
});
```

OUTPUT:



Conclusion: By following these steps and best practices, we have effectively code, register, and manage a service worker for the E-commerce PWA, providing users with a seamless and reliable shopping experience both online and offline.