Experiment 09

Aim: To implement **Service Worker events** like `fetch`, `sync`, and `push` in an **Weather Progressive Web App (PWA)**. These features enable the web application to function offline, perform background synchronization when the device regains connectivity, and receive push notifications from the server even when the app is not open.

Theory:

A **Service Worker** is a script that runs in the background, separate from the main browser thread. It acts as a proxy between the network and the browser, allowing developers to intercept network requests, cache resources, and handle actions even when the user is offline. Service Workers are at the core of making a web application a **PWA** by enabling features like offline functionality, background sync, and push notifications.

Key features of a Service Worker include offline access by caching files, receiving and displaying push notifications, and syncing data when the user comes back online. The lifecycle of a Service Worker involves events such as `install` (to cache essential assets), `activate` (to clean old caches), `fetch` (to intercept and serve requests), `sync` (to handle background synchronization), and `push` (to show notifications from the server).

In this project, we have used `fetch` to manage asset caching and offline loading, `sync` to handle cart syncing when the network is available again, and `push` to receive and display promotional alerts from the server.

Implementation:

To implement this, we used **Node.js** with **Express** for the backend, and **Web Push API** to handle notifications. The backend receives push subscriptions from the client and stores them. When a notification needs to be sent, the server sends a push message to all subscribed clients.

On the frontend, we register a Service Worker using JavaScript. Once registered, the app subscribes to push notifications by using the `pushManager` API and

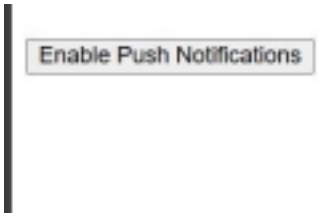
sends the subscription details to the backend using a POST request. When the user clicks a button to enable push notifications, this process is triggered.

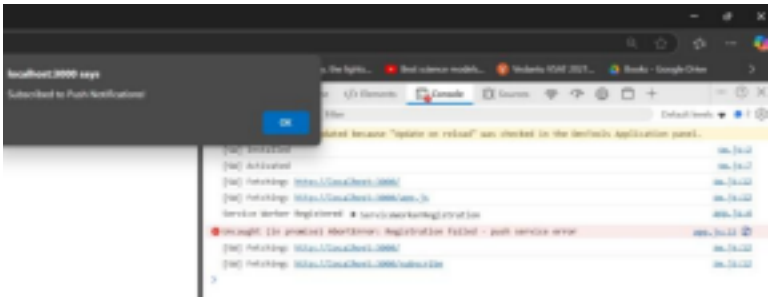
Inside the Service Worker (`sw.js`), we handle the `install` event to cache assets like HTML, CSS, and JS. The `fetch` event listens for any network requests and serves cached files if available, enabling offline usage. The `sync` event registers a background task named `sync-cart`, which simulates syncing cart data to the server when the connection is restored. Finally, the `push` event listens for messages from the server and displays them as notifications with a title, body, and icon.

To test the push feature, we start the server using Node, open the app in a browser, and click the button to enable push notifications. Then, using a curl command from a second terminal, we POST to the `/send-push` endpoint. This sends a push message to the browser, and the Service Worker logs `[SW] Push received` and displays a desktop notification.

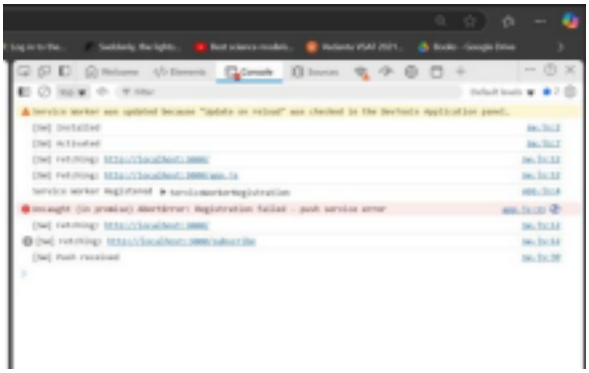
Output:

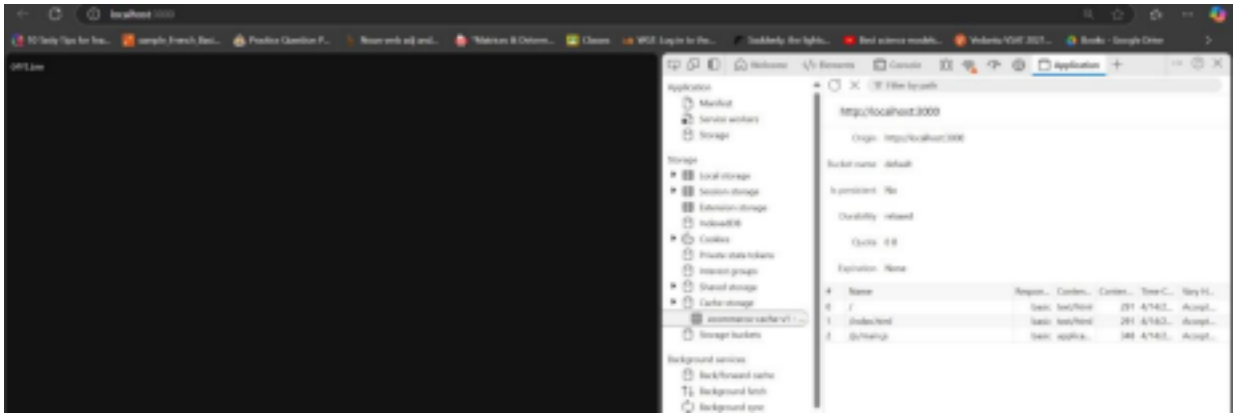
After running the application and registering the Service Worker, we can observe console logs like `[SW] Installed`, `[SW] Activated`, `[SW] Fetching`, and `[SW] Push received`. When the app is offline, assets are still accessible because they are served from cache. When the `sync` event is triggered, the Service Worker logs that a background sync was handled. On running the curl command to trigger push, a notification pops up on the desktop confirming that push notifications are working as expected.





```
C:\Users\adity\Desktop\MAD\pwa_09>curl -X POST http://localhost:3000/send-push
OK
C:\Users\adity\Desktop\MAD\pwa_09>
```





Conclusion:

This project demonstrates the powerful capabilities of Service Workers in enhancing user experience in modern web applications. By implementing `fetch`, `sync`, and `push` events, we successfully added offline support, background data sync, and push notification features to our Weather PWA. These features improve reliability, engagement, and responsiveness, making the app more robust and user-friendly, even in poor network conditions.