Experiment 09

Aim: To implement **Service Worker events** like fetch, sync, and push in an **E-commerce 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.

1. **fetch Event**
Purpose: Cache static files and serve them offline.
Implementation:

- Cached core assets during Service Worker install.
- Used caches.match() in the fetch event to serve cached content when offline.

Code:

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

2. sync Event

Purpose: Perform background tasks (e.g., sending data) when the device is back online.

Implementation:

- Registered a sync event using reg.sync.register("sync-qr") in the main thread.
- Handled the sync in the Service Worker using sync listener.

Code:

service-worker.json

```
self.addEventListener("sync", (event) => {
  if (event.tag === "sync-qr") {
    event.waitUntil(sendDataToServer());
  }
});

async function sendDataToServer() {
  // Example function: can use IndexedDB to queue requests
  console.log("Syncing data to server...");
}
```

Script.js

```
navigator.serviceWorker.ready.then((reg) => {
  return reg.sync.register("sync-qr");
});
```

3. push Event

Purpose: Send push notifications to the user, even when the app is in the background.

Implementation:

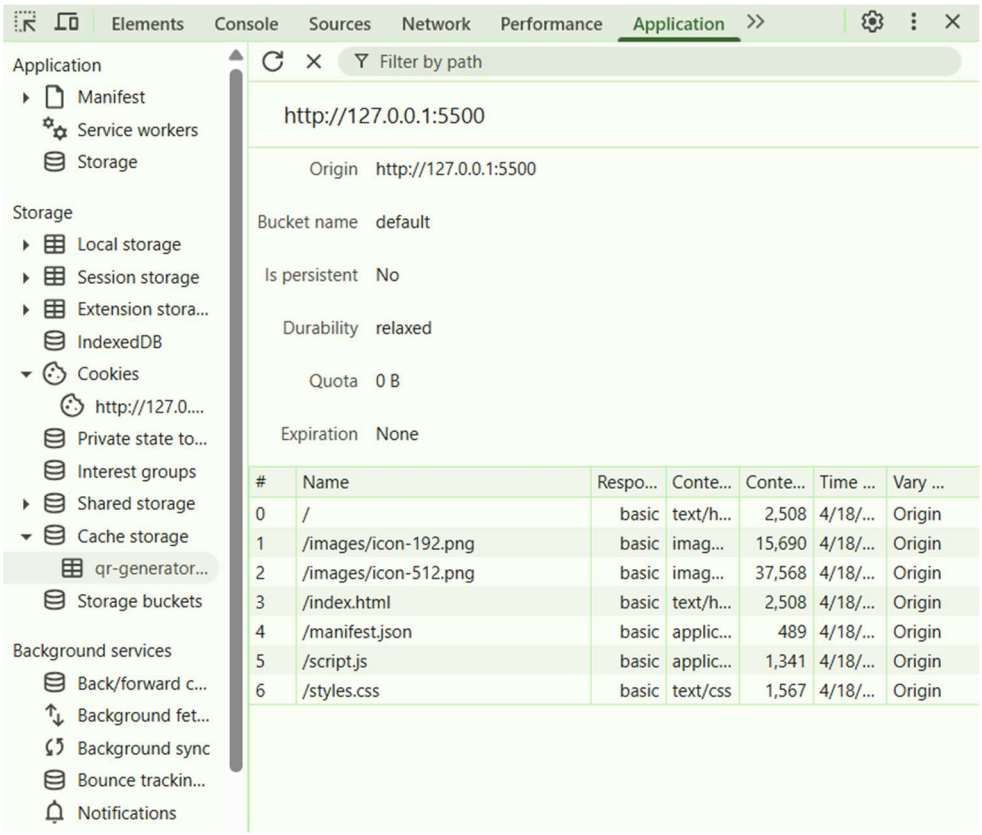
- Handled push event in Service Worker and displayed a notification using showNotification().

Code:

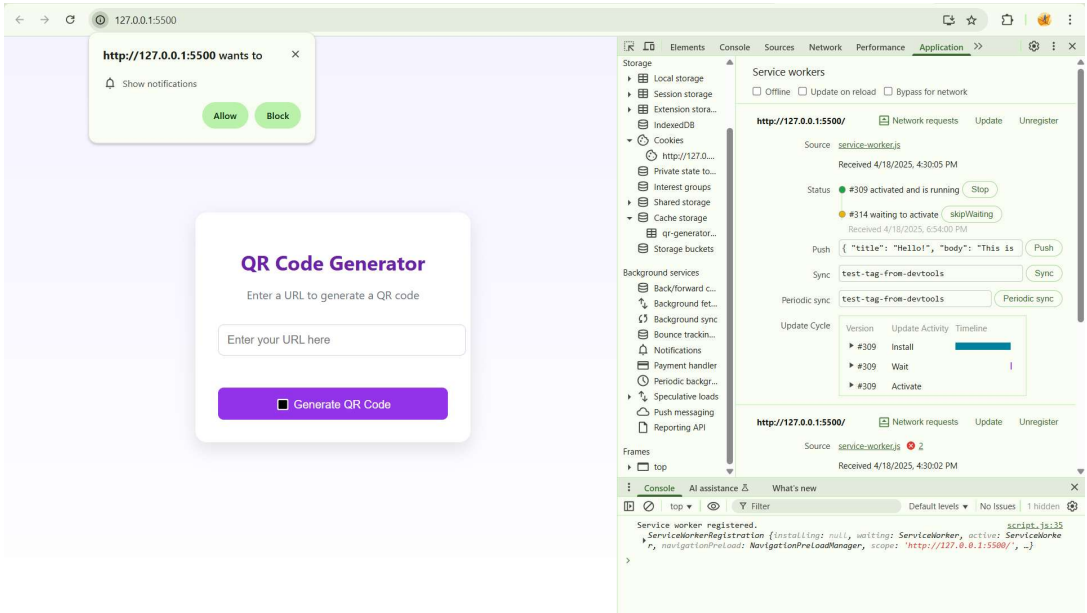
```
self.addEventListener("push", (event) => {
  const data = event.data.json();
  self.registration.showNotification(data.title, {
    body: data.body,
    icon: "images/icon-192.png"
  });
});
```

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.



Cached App Data



Push Event

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 E-commerce PWA. These features improve reliability, engagement, and responsiveness, making the app more robust and user-friendly, even in poor network conditions.