MPL Experiment 9 (PWA)

Name: Bhagyesh Patil **Class:** D15A **Roll no: 35**

Aim: To implement Service worker events like fetch, sync and push for E-commerce PWA.

Theory:

Service Worker

Service Worker is a script that works on browser background without user interaction independently. Also, It resembles a proxy that works on the user side. With this script, you can track network traffic of the page, manage push notifications and develop “offline first” web applications with Cache API.

Things to note about Service Worker:

- A service worker is a programmable network proxy that lets you control how network requests from your page are handled.
- Service workers only run over HTTPS. Because service workers can intercept network requests and modify responses, "man-in-the-middle" attacks could be very bad.
- The service worker becomes idle when not in use and restarts when it's next needed. You cannot rely on a global state persisting between events. If there is information that you need to persist and reuse across restarts, you can use IndexedDB databases.
- Service workers make extensive use of promises, so if you're new to promises, then you should stop reading this and check out Promises, an introduction.

Fetch Event

You can track and manage page network traffic with this event. You can check existing cache, manage “cache first” and “network first” requests and return a response that you want.

Of course, you can use many different methods but you can find in the following example a “cache first” and “network first” approach. In this example, if the request’s and current location’s origin are the same (Static content is requested.), this is called “cacheFirst” but if you request a targeted external URL, this is called “networkFirst”.

- **CacheFirst** - In this function, if the received request has cached before, the cached response is returned to the page. But if not, a new response requested from the network.
- **NetworkFirst** - In this function, firstly we can try getting an updated response from the network, if this process completed successfully, the new response will be cached and returned. But if this process fails, we check whether the request has been cached

before or not. If a cache exists, it is returned to the page, but if not, this is up to you. You can return dummy content or information messages to the page.

Sync Event

Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are writing e-mail content and we didn't realize it. When completing the writing, we click the send button.

Push Event

This is the event that handles push notifications that are received from the server. You can apply any method with received data.

We can check in the following example.
"Notification.requestPermission();" is the necessary line to show notification to the user. If you don't want to show any notification, you don't need this line.

In the following code block is in sw.js file. You can handle push notifications with this event. In this example, I kept it simple. We send an object that has "method" and "message" properties. If the method value is "pushMessage", we open the information notification with the "message" property.

Code:

```
//Serviceworker.js
// sw.js - Complete Service Worker for E-commerce PWA
const CACHE_NAME = 'ecommerce-pwa-v2';
const API_CACHE = 'ecommerce-api-v1';
const ASSETS_TO_CACHE = [
  '/',
  '/index.html',
  '/manifest.json',
  '/offline.html',
  '/css/main.min.css',
  '/js/app.min.js',
  '/icons/icon-192x192.png'
  ,
  '/icons/icon-512x512.png'
  ,
```

```
  '/images/placeholder-product.jpg'
];

// ======================
// Install Event
// ======================
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => {
        console.log('[Service Worker] Cache opened');
        return cache.addAll(ASSETS_TO_CACHE);
      })
      .then(() => self.skipWaiting())
  );
});

// ======================
// Activate Event
// ======================
self.addEventListener('activate', (event) => {
  event.waitUntil(
    caches.keys().then((cacheNames) =>
      { return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheName !== CACHE_NAME && cacheName !== API_CACHE) {
            console.log('[Service Worker] Deleting old cache:',
cacheName);                              return caches.delete(cacheName);

          }
        })
      );
    })

    .then(() => self.clients.claim())
  );
});

// ======================
```

```
// Fetch Event
// ======================
self.addEventListener('fetch', (event) =>
  { const { request } = event;
  const url = new URL(request.url);

  // 1. Skip non-GET requests and chrome-extension
  if (request.method !== 'GET' || url.protocol ===
'chrome-extension:') {
    return;
  }

  // 2. API Requests (Network First with Cache Fallback)
  if (url.pathname.startsWith('/api/')) {
    event.respondWith(
      fetch(request)
        .then(networkResponse => {
          // Cache successful API
          responses if
          (networkResponse.ok) {
            const clone = networkResponse.clone();
            caches.open(API_CACHE)
              .then(cache => cache.put(request, clone));
          }
          return networkResponse;
        })
        .catch(() => {
          // Return cached version if available
          return caches.match(request)
            .then(cachedResponse => cachedResponse || Response.json(
              { error: 'Network error' },
              { status: 503 }
            ));
        })
    );
    return;
  }

  // 3. Static Assets (Cache First with Network Fallback)
```

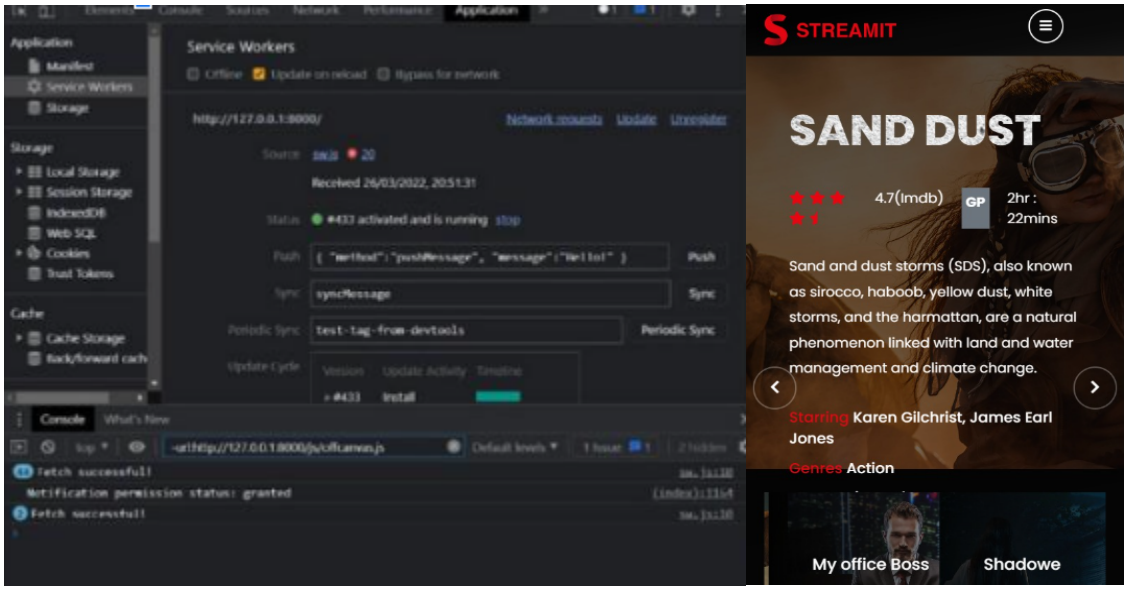
```
  event.respondWith(
    caches.match(request)
      .then(cachedResponse => {
        // Return cached version if found
        if (cachedResponse) {
          return cachedResponse;
        }

        // Otherwise fetch from network
        return fetch(request)
          .then(networkResponse => {
            // Cache successful responses
            if (networkResponse.ok) {
              const clone = networkResponse.clone();
              caches.open(CACHE_NAME)
                .then(cache => cache.put(request, clone));
            }
            return networkResponse;
          })
          .catch(() => {
            // Special handling for HTML pages
            if (request.headers.get('accept').includes('text/html')) {
              return caches.match('/offline.html');
            }
            // Return placeholder for images
            if (request.headers.get('accept').includes('image')) {
              return caches.match('/images/placeholder-product.jpg');
                                    }
                                    }
                                    )
                                    ;
  );                                }
});


// ======================
// Background Sync
// ======================
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-cart') {
```

```
    event.waitUntil(
      // Get cart data from IndexedDB
      getCartData()
        .then(cartItems => {
          return fetch('/api/cart/sync', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(cartItems)
          });
        })
        .then(() => {
          return showNotification('Cart Synced', 'Your cart has been
updated');
        })
        .catch(err => {
          console.error('Sync failed:', err);
        })
    );
  }
 })
  ;

// ======================
// Push Notifications
// ======================
self.addEventListener('push', (event) => {
  let data = {};
  try {
    data = event.data.json();
  } catch (e) {
    data = {
      title: 'New Update',
      body: 'Check out our latest products!',
      icon: '/icons/icon-192x192.png',
      url: '/'
    };
  }

  const options = {
```

```
    body: data.body,
    icon: data.icon || '/icons/icon-192x192.png',
    badge: '/icons/icon-96x96.png',
    data: {
      url: data.url || '/'
    }
  };

  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});

self.addEventListener('notificationclick', (event) => {
  event.notification.close();
  event.waitUntil(
    clients.matchAll({ type: 'window' })
      .then(clientList => {
        for (const client of clientList) {
          if (client.url === event.notification.data.url && 'focus' in
client) {
            return client.focus();
          }
        }
        if (clients.openWindow) {
          return clients.openWindow(event.notification.data.url);
        }
      })
  );
});

// ======================
// Helper Functions
// ======================
async function getCartData() {
  // In a real app, you would use
  IndexedDB return new Promise(resolve =>
  {
    resolve([]);
```

```
  });
}

async function showNotification(title, body) {
  return self.registration.showNotification(title, { body });
}
```

Output:



t