

## Experiment No. 9

**Niraj Kothawade**  
**D15A - 24**

**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.

```
self.addEventListener("fetch", function (event) {
  const req = event.request;
  const url = new URL(req.url);

  if (url.origin === location.origin) {
    event.respondWith(cacheFirst(req));
  }
  else {
    event.respondWith(networkFirst(req));
  }
});

async function cacheFirst(req) {
  return await caches.match(req) || fetch(req);
}

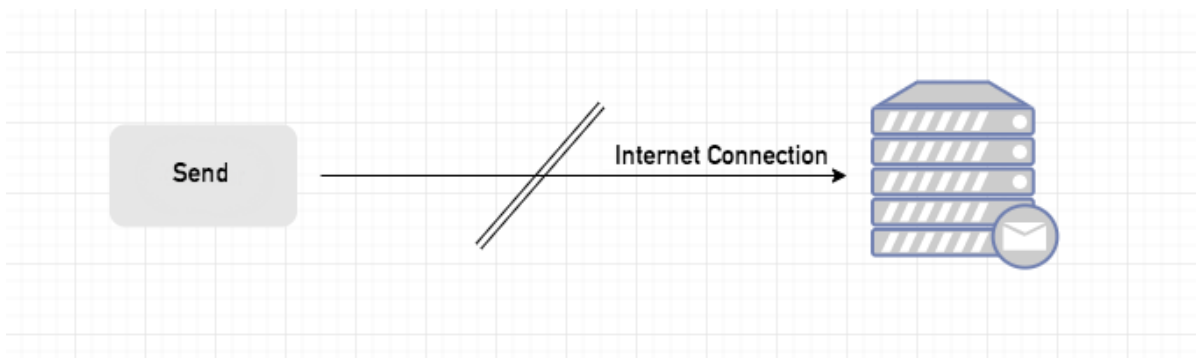
async function networkFirst(req) {
  const cache = await caches.open("pwa-dynamic");
  try {
    const res = await fetch(req);
    cache.put(req, res.clone());
    return res;
  } catch (error) {
    const cachedResponse = await cache.match(req);
    return cachedResponse || await caches.match("./noconnection.json");
  }
}
```

## Sync Event

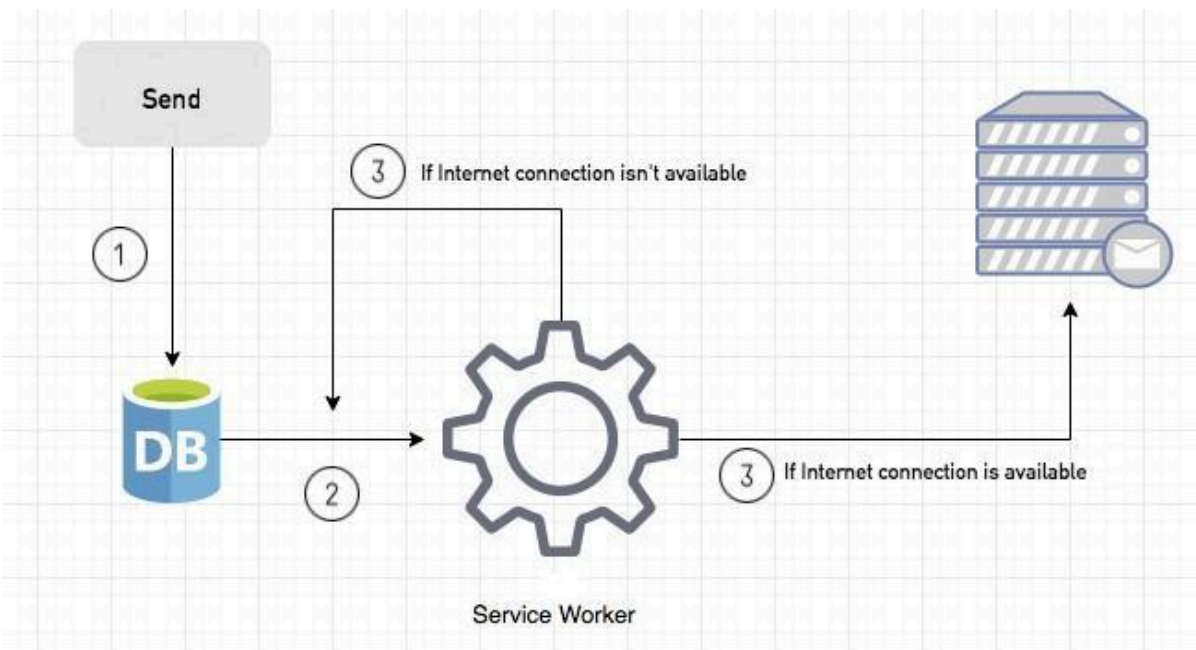
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are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

Here is a job for the Background Sync.

The following view shows the classical process of sending email to us. If the Internet Connection is broken, we can't send any content to Mail Server.



Here, you can create any scenario for yourself. A sample is in the following for this case.



1. When we click the “send” button, email content will be saved to IndexedDB.
2. Background Sync registration.
3. **If the Internet connection is available**, all email content will be read and sent to Mail Server.

**If the Internet connection is unavailable**, the service worker waits until the connection is available even though the window is closed. When it is available, email content will be sent to Mail Server.

You can see the working process within the following code block.

#### Event Listener for Background Sync Registration

```
document.querySelector("button").addEventListener("click", async () => {  
  var swRegistration = await navigator.serviceWorker.register("sw.js");  
  swRegistration.sync.register("helloSync").then(function () {  
    console.log("helloSync success [main.js]");  
  });  
});
```

#### Event Listener for sw.js

```
self.addEventListener('sync', event => {  
  if (event.tag == 'helloSync') {  
    console.log("helloSync [sw.js]");  
  }  
});
```

#### 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.

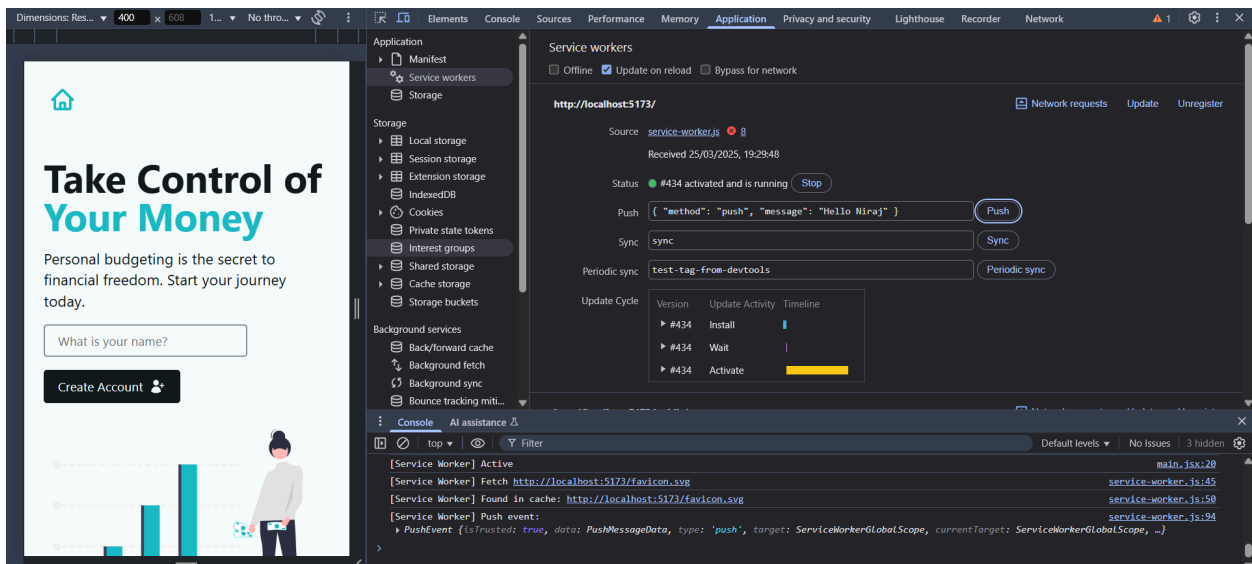
“Notification.requestPermission();” is the necessary line to show notification to the user. If you don’t want to show any notification, you don’t need this line.

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.

```
self.addEventListener('push', event => {  
  if (event && event.data) {  
    var data = event.data.json();  
    if (data.method === "pushMessage") {  
      event.waitUntil(self.registration.showNotification("Test App", {  
        body: data.message  
      }));  
    }  
  }  
});
```

You can use Application Tab from Chrome Developer Tools for testing push notification.



## Code:

### service-worker.js

```
// public/service-worker.js

const CACHE_NAME = 'expense-tracker-v1'; // Change this when you update the service
worker

const urlsToCache = [

  '/', // Cache the root URL (index.html)

  '/index.html',

  '/manifest.json', // if you have one

  // Correct paths to your built assets

  'dist/assets/index-BPUHWJbD.js', // Replace with the actual path

  'dist/assets/index-BvWQCeOo.css',

];

self.addEventListener('install', (event) => {

  console.log('[Service Worker] Install');

  event.waitUntil(

    caches.open(CACHE_NAME)

      .then((cache) => {

        console.log('[Service Worker] Caching app shell');

        return cache.addAll(urlsToCache);

      })

      .catch((error) => {

        console.error('[Service Worker] Caching failed:', error);

      })

  );

});

self.addEventListener('activate', (event) => {

  console.log('[Service Worker] Activate');

  event.waitUntil(

    caches.keys().then((cacheNames) => {
```

```

return Promise.all(
  cacheNames.map((cacheName) => {
    if (cacheName !== CACHE_NAME) {
      console.log('[Service Worker] Deleting old cache:', cacheName);
      return caches.delete(cacheName);
    }
  })
);
});

// Fetch Event: Cache-first strategy
self.addEventListener('fetch', (event) => {
  console.log('[Service Worker] Fetch', event.request.url);
  event.respondWith(
    caches.match(event.request)
      .then((response) => {
        if (response) {
          console.log('[Service Worker] Found in cache:', event.request.url);
          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;
          });
      });
  );
});

```

```

    })

    .catch((error) => {

        console.error('[Service Worker] Fetch failed:', error);

        // Handle network errors (e.g., show offline message)

    });

})

);

});

// Sync Event: Handle background sync
self.addEventListener('sync', (event) => {

    console.log('[Service Worker] Sync event:', event.tag);

    if (event.tag === 'sync-expenses') {

        event.waitUntil(

            // Your background sync logic here (e.g., send expenses to server)
            syncExpenses()

                .then(() => {

                    console.log('[Service Worker] Expenses synced successfully');

                })

                .catch((error) => {

                    console.error('[Service Worker] Expense sync failed:', error);

                    // Handle sync errors (e.g., retry later)

                })

        );

    }

});

// Push Event: Handle push notifications
self.addEventListener('push', function(event) {

    console.log('[Service Worker] Push event:', event);

    let data;

    try {

```



```

    data = event.data ? event.data.json() : {};
  } catch (error) {
    console.error('Error parsing push notification data:', error);
    data = { title: 'Error', message: 'Could not parse notification data.' };
  }

  if (Notification.permission === 'granted') {
    event.waitUntil(
      self.registration.showNotification(data.title || 'Xtrack', {
        body: data.message || 'No message provided.',
        icon: data.icon || '/your-notification-icon.png', // Provide a default icon
      })
    );
  } else {
    console.log('Notification permission not granted. Notification will not be
shown.');
```

```

//Helper function for sync event
```

```

async function syncExpenses() {
  // Replace this with your actual expense syncing logic
  // This example just simulates a delay
  return new Promise((resolve) => {
    setTimeout(resolve, 2000); // Simulate a 2-second delay
  });
}
```

**Output:**

## Fetchevent

The screenshot shows a web application on the left and Chrome DevTools on the right. The web application has a header with a home icon and a main heading "Take Control of Your Money". Below the heading is a subheading "Personal budgeting is the secret to financial freedom. Start your journey today." and a form with a text input "What is your name?" and a "Create Account" button. The DevTools Application panel shows the "Service workers" section for the URL "http://localhost:5173/". The status is "#436 activated and is running". The "Update Cycle" table shows the following data:

Version	Update Activity	Timeline
#436	Install	
#436	Wait	
#436	Activate	

The Console shows the following log messages:

```
[Service Worker] Registered with scope: http://localhost:5173/public/
[Service Worker] Active
[Service Worker] Fetch http://localhost:5173/favicon.svg
[Service Worker] Found in cache: http://localhost:5173/favicon.svg
Fetch Successfull
```

## Sync event

The screenshot shows a web application on the left and Chrome DevTools on the right. The web application has a header with "welcome back, Niraj" and a subheading "Personal budgeting is the secret to financial freedom." Below the subheading is a heading "Create a budget to get started!" and a "Create Budget" form. The form has a "Budget Name" input with the value "e.g., Groceries" and an "Amount" input with the value "e.g., ₹350". The DevTools Application panel shows the "Service workers" section for the URL "http://localhost:5173/". The status is "#439 activated and is running". The "Update Cycle" table shows the following data:

Version	Update Activity	Timeline
#439	Install	
#439	Wait	
#439	Activate	

The Console shows the following log messages:

```
[Service Worker] Fetch http://localhost:5173/favicon.svg
[Service Worker] Found in cache: http://localhost:5173/favicon.svg
Fetch Successfull
[Service Worker] Sync event: sync-expenses
[Service Worker] Expenses synced successfully
```

## Push event

The screenshot displays a web browser window with a PWA interface on the left and the Chrome DevTools Application panel on the right. The PWA interface, titled "Take Control of Your Money", features a home icon, a text input field for a name, a "Create Account" button, and a bar chart. The DevTools Application panel shows the "Service workers" section for the URL `http://localhost:5173/`. It indicates that the service worker `service-worker.js` is active and running on port 8. The "Push" event is configured with the message `{ "method": "push", "message": "Hello Niraj" }`. The "Sync" event is set to `sync`, and the "Periodic sync" event is set to `test-tag-from-devtools`. The "Update Cycle" table shows the status of the service worker: 

Version	Update Activity	Timeline
#434	Install	
#434	Wait	
#434	Activate	

. The Console panel at the bottom shows the following log messages: 

```
[Service Worker] Active  
[Service Worker] Fetch http://localhost:5173/favicon.svg  
[Service Worker] Found in cache: http://localhost:5173/favicon.svg  
[Service Worker] Push event:  
PushEvent {isTrusted: true, data: PushMessageData, type: 'push', target: ServiceWorkerGlobalScope, currentTarget: ServiceWorkerGlobalScope, ...}
```

**Conclusion:** Successfully implemented Service worker events like fetch, sync and push for PWA.