

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”.

```
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));
  }
});

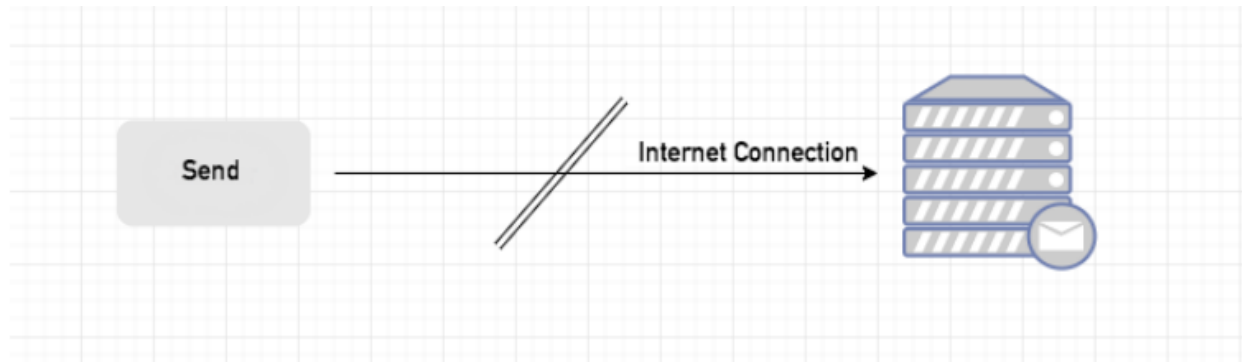
async function cacheFirst(req) {
  return await caches.match(req) || fetch(req);
}

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");
  }
}
```

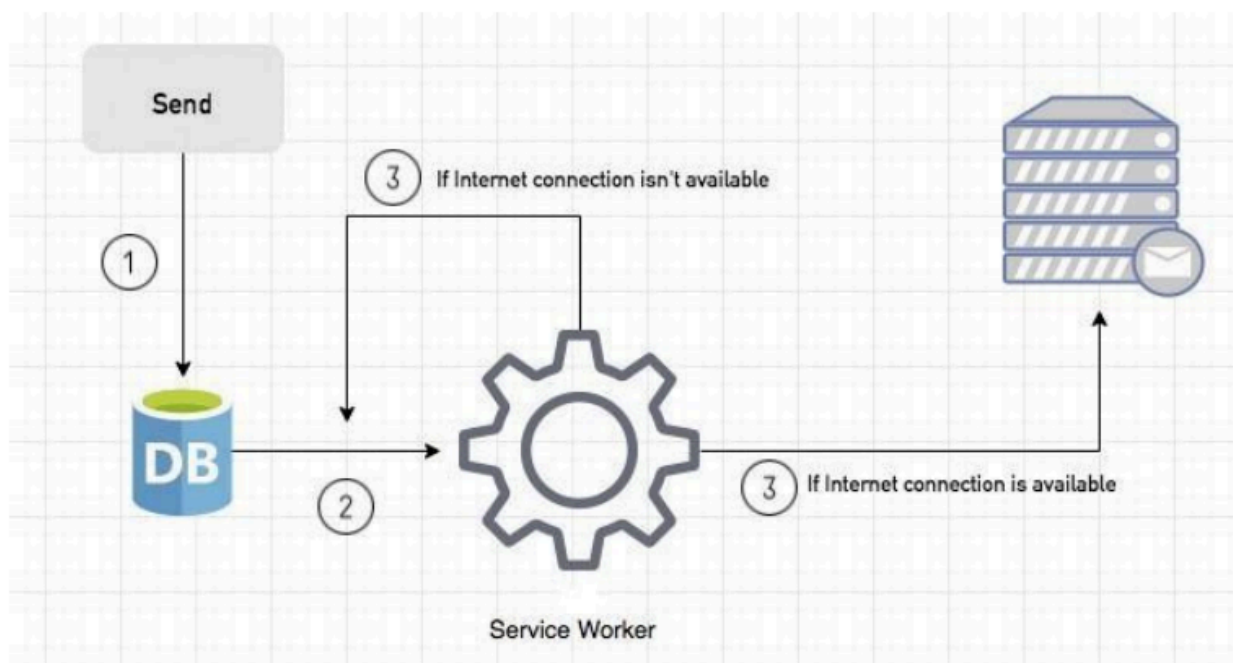
Sync Event

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]");
  }
});
```

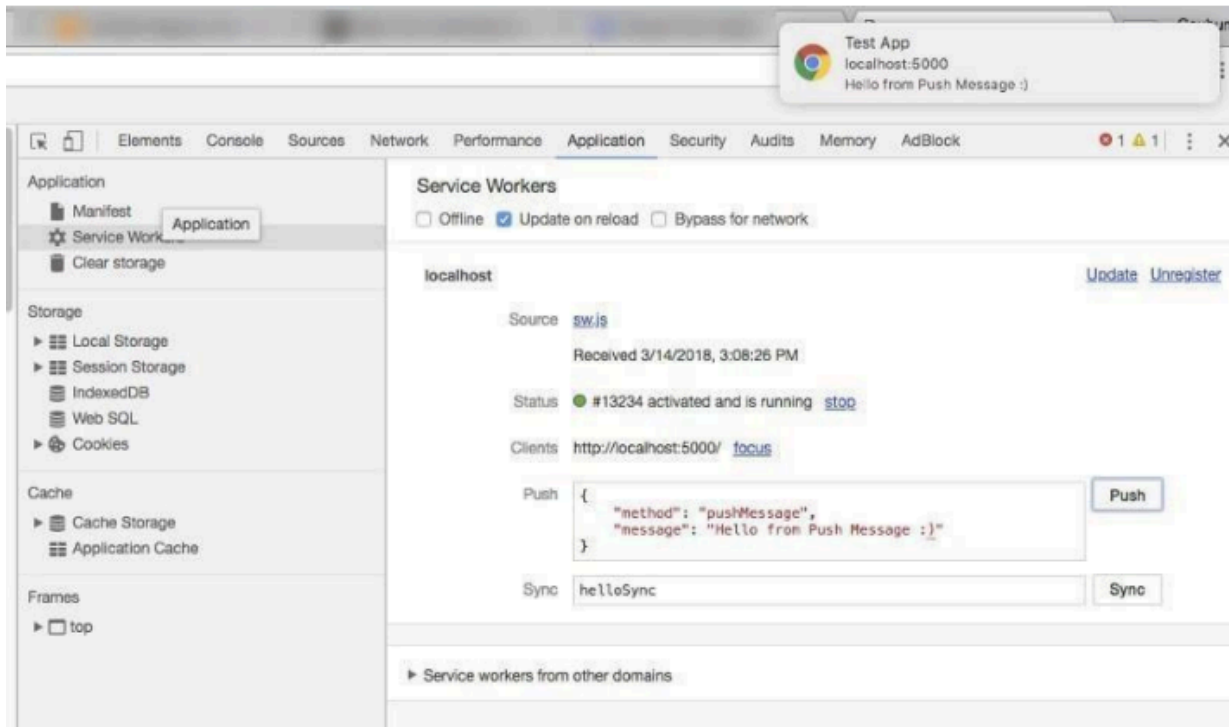
Push Event This is the event that handles push notifications that are received from the server. You can apply any method with received data.

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: sw.js

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

self.addEventListener("fetch", function (event) {
  event.respondWith(checkResponse(event.request).catch(function () {
    console.log("Fetch from cache successful!") return
    returnFromCache(event.request);
  }));
  console.log("Fetch successful!") event.waitUntil(addToCache(event.request));
});

self.addEventListener('sync', event => { if
  (event.tag === 'syncMessage') {
```

```
        console.log("Sync successful!")
    }
});

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

var filesToCache = [
    '/',
    '/menu',
    '/contactUs',
    '/offline.html',
];
```

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

```
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("offline.html");  
            } else {  
return matching;  
            }  
        });  
    });  
};
```

Output:

Fetch event:

The screenshot displays the Chrome DevTools interface. On the left, the 'Storage' panel is open, showing a list of storage areas: Local storage, Session storage, IndexedDB, Web SQL, Cookies, Private state t, Interest group, Shared storage, and Cache storage. The 'Background services' panel is also visible, showing 'Service workers from other origins'. The 'Update Cycle' table lists three updates for version #43: Install, Wait, and Activate. The 'Console' panel at the bottom shows four log messages indicating successful fetches of styles.css and script.js from both network and cache.

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

```
Fetch successful (from network, cached): http://127.0.0.1:5500/styles.css sw.js:93
Fetch successful (from cache): http://127.0.0.1:5500/styles.css sw.js:65
Fetch successful (from network, cached): http://127.0.0.1:5500/script.js sw.js:93
Fetch successful (from cache): http://127.0.0.1:5500/script.js sw.js:65
```

Sync event

The screenshot shows a web application for 'Ice Cream Haven'. The header includes the title 'Welcome to Ice Cream Haven' and the tagline 'Indulge in Sweet Delights of Creamy Goodness'. The main content area displays four ice cream flavors: Chocolate Delight (Price: ₹50), Decadent Oreo (Price: ₹40), Strawberry Sensation (Price: ₹45), and Black Current (Price: ₹55). Each flavor has a 'Buy' button. The console on the right shows three log messages: 'Service worker registration successful: http://127.0.0.1:5500/', 'helloSync success [script.js]', and 'helloSync [sw.js]'.

```
Service worker registration successful: http://127.0.0.1:5500/
helloSync success [script.js]
helloSync [sw.js]
```


Push event

The screenshot displays the Chrome DevTools Service Worker interface. On the left, the 'Storage' panel is expanded, showing various storage areas like Local storage, Session storage, IndexedDB, Web SQL, Cookies, Private state t, Interest group, Shared storage, Cache storage, and offline - htt. The main panel shows the 'Service workers from other origins' section with a link to 'See all registrations'. Below this, the 'Update Cycle' table is visible:

Version	Update Activity	Timeline
#41	Install	
#41	Wait	
#41	Activate	<div></div>

Below the table, the 'Service workers from other origins' section is visible with a link to 'See all registrations'. The 'Console' panel at the bottom shows a message: 'Service worker registration successful: <http://127.0.0.1:5500/> [script.js:93](#)'. The 'Push' event configuration is also visible, showing a status of '#41 activated and is running' and a 'Push' button.

Conclusion : We have understood and successfully implemented events like fetch , push and sync for our ecommerce pwa.