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.

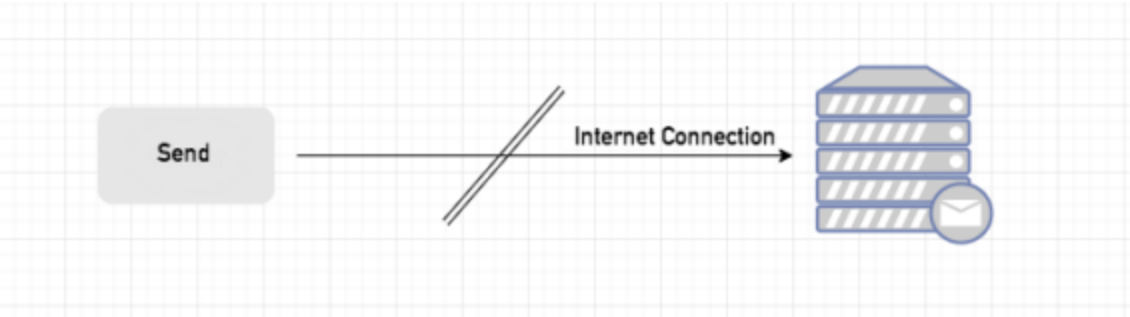
- **NetworkFirst** - In this function, firstly we can try getting an updated response from the network, if this process completed successfully, the new response will be cached and returned. But if this process fails, we check whether the request has been cached before or not. If a cache exists, it is returned to the page, but if not, this is up to you. You can return dummy content or information messages to the page.

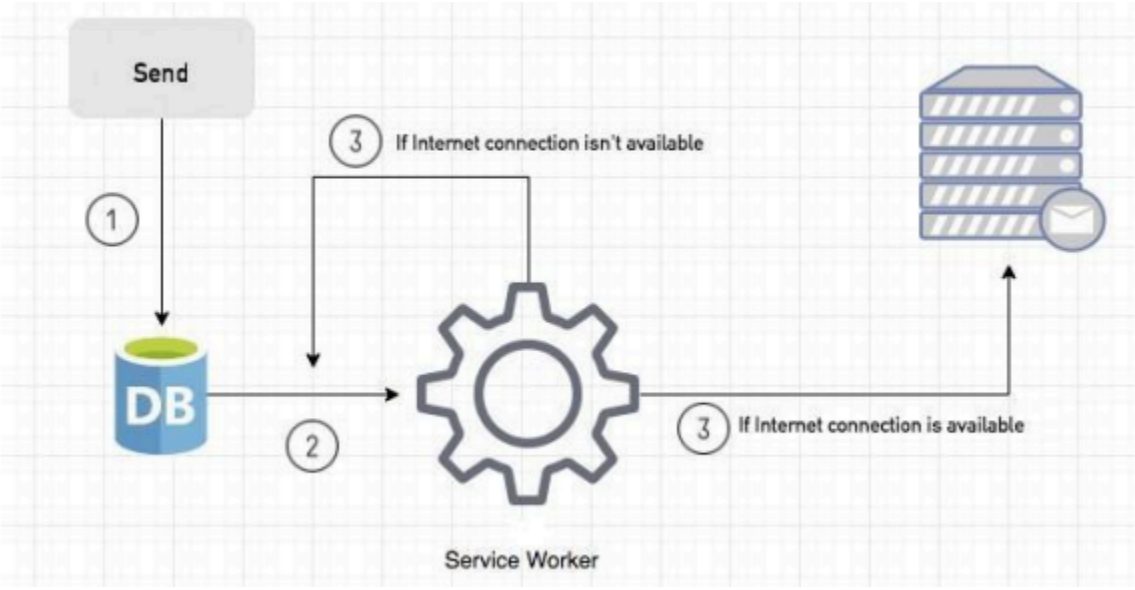
```
self.addEventListener("fetch", function (event) {
    const req = event.request;
    const url = new URL(req.url);

    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.





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.

```
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
            }));
        }
    }
});
```

sw.js:

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

  event.respondWith(
    (async function() {
      try {
        if (url.origin === location.origin) {
          const cachedResponse = await cacheFirst(req);
          if (cachedResponse) {
            console.log("Fetch successful (from cache):", req.url);
          } else {
            console.log("Fetch successful (from network):", req.url);
          }
          return cachedResponse;
        } else {
          const response = await networkFirst(req);
          console.log("Fetch successful (from network):", req.url);
          return response;
        }
      } catch (error) {
        console.error("Fetch failed:", error);
```

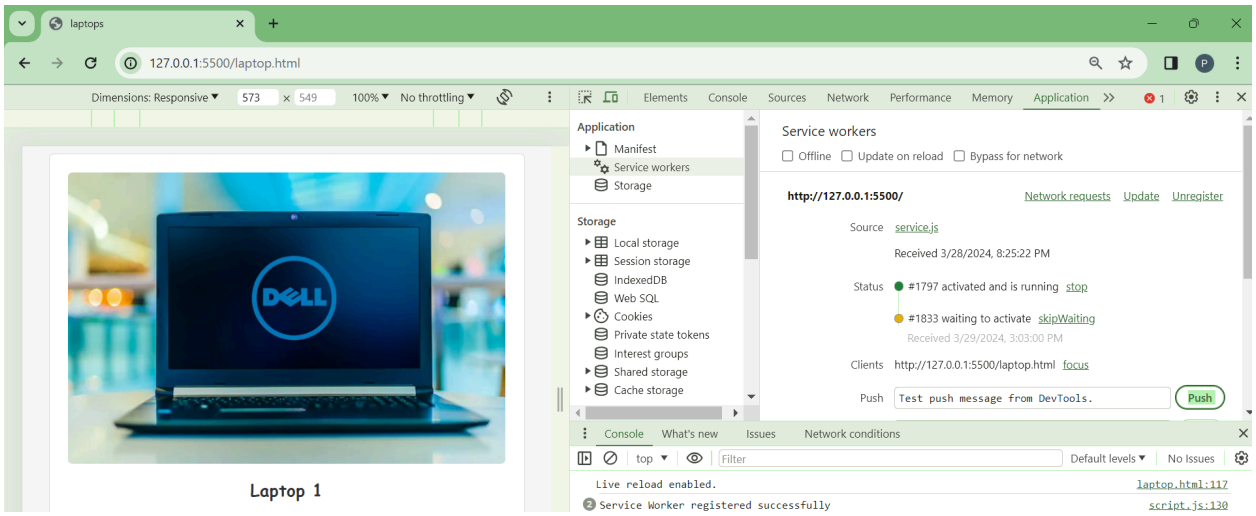
```
        // Handle errors gracefully here, e.g., return a cached response
or display an error message
      }
    })()
  );
});

async function cacheFirst(req) {
  const cachedResponse = await caches.match(req);
  if (cachedResponse) {
    return cachedResponse;
  }

  try {
    const response = await fetch(req);
    caches.open("dynamic-pwa").then((cache) => cache.put(req,
response.clone()));
    // Log after caching to accurately reflect success
    console.log("Fetch successful (from network, cached):", req.url);
    return response;
  } catch (error) {
    console.error("CacheFirst fetch failed:", req.url, error);
    throw error; // Re-throw to allow handling in the main fetch event
listener
  }
}

async function networkFirst(req) {
  try {
    const cache = await caches.open("dynamic-pwa");
    const response = await fetch(req);
    cache.put(req, response.clone());
    return response;
  } catch (error) {
    const cachedResponse = await cache.match(req);
    if (cachedResponse) {
      console.log("Fetch successful (from cache):", req.url);
      return cachedResponse;
    }
```

```
    console.error("NetworkFirst fetch failed, no cached response:",
req.url, error);
    throw error; // Re-throw to allow handling in the main fetch event
listener
  }
}


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

// Push notification event listener
self.addEventListener('push', event => {
  if (event && event.data) {
    var data = event.data.json();
    if (data.method === "pushMessage") {
      if ('showNotification' in self.registration) {
        event.waitUntil(self.registration.showNotification("Siddhi is
Testing", {
          body: data.message
        }));
      } else {
        // Handle browsers that don't support `showNotification`
      }
    }
  }
});
```

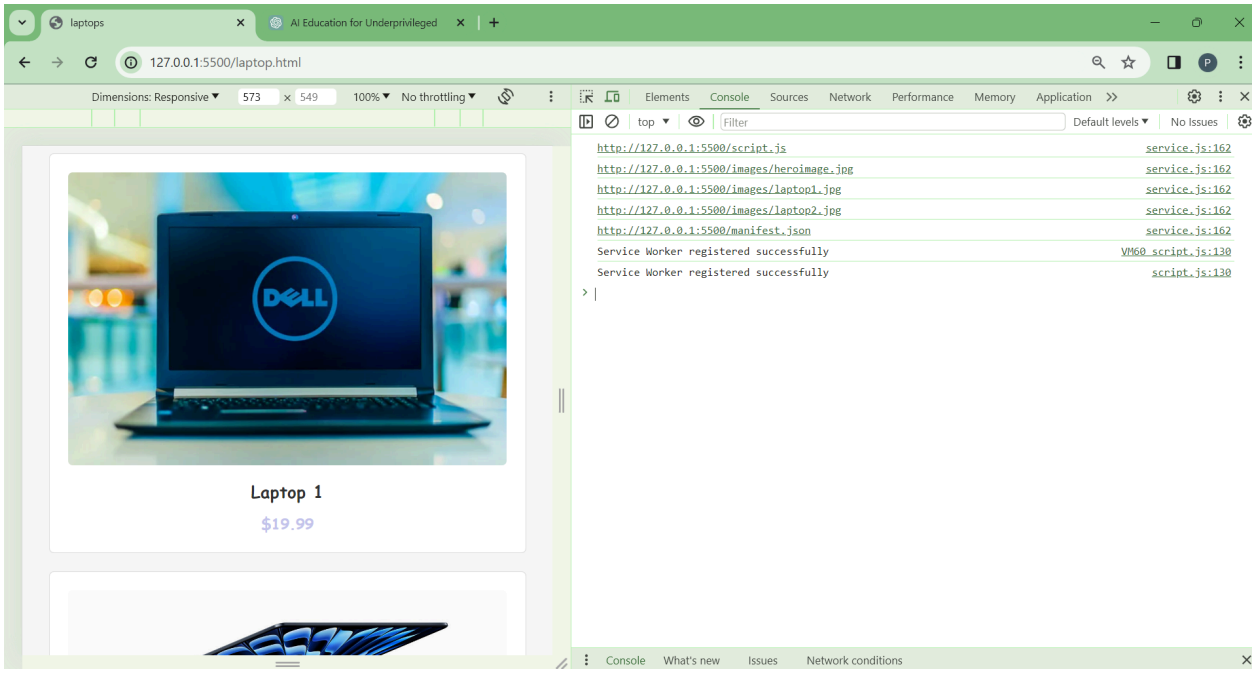
Name :Goutam Chandnani **Exp:9** D15B_08

Output :

Push :



Fetched files:



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