**EXPERIMENT No. 9**

Name : Arya Manoj Madhavi

Class : D15B

Roll No.: 29

**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.

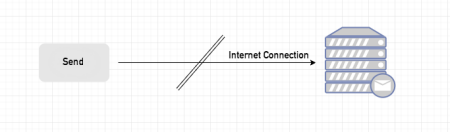
● **NetworkFirst** - In this function, firstly we can try getting an updated response from the network, if this process completed successfully, the new response will be cached and returned. But if this process fails, we check whether the request has been cached before or not. If a cache exists, it is 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**

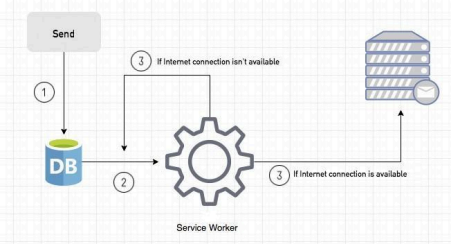
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are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

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.

**Push Event**

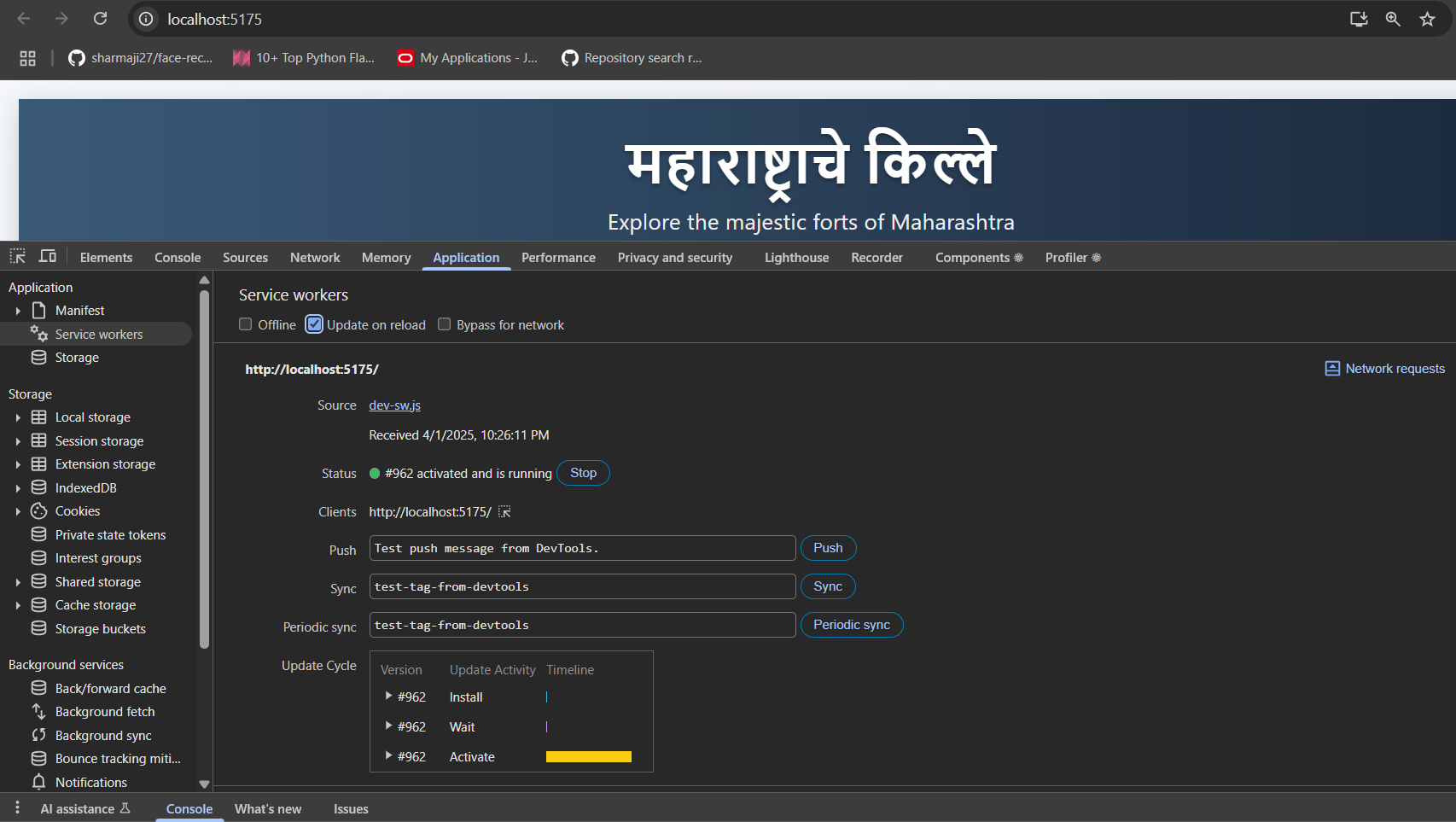
This is the event that handles push notifications that are received from the server. You can apply any method with received data.

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.

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

**Code**:   
**sw.js**  
/\*\*

\* Copyright 2018 Google Inc. All Rights Reserved.

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\* http://www.apache.org/licenses/LICENSE-2.0

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

// If the loader is already loaded, just stop.

if (!self.define) {

let registry = {};

// Used for `eval` and `importScripts` where we can't get script URL by other means.

// In both cases, it's safe to use a global var because those functions are synchronous.

let nextDefineUri;

const singleRequire = (uri, parentUri) => {

uri = new URL(uri + ".js", parentUri).href;

return registry[uri] || (

new Promise(resolve => {

if ("document" in self) {

const script = document.createElement("script");

script.src = uri;

script.onload = resolve;

document.head.appendChild(script);

} else {

nextDefineUri = uri;

importScripts(uri);

resolve();

}

})

.then(() => {

let promise = registry[uri];

if (!promise) {

throw new Error(`Module ${uri} didn’t register its module`);

}

return promise;

})

);

};

self.define = (depsNames, factory) => {

const uri = nextDefineUri || ("document" in self ? document.currentScript.src : "") || location.href;

if (registry[uri]) {

// Module is already loading or loaded.

return;

}

let exports = {};

const require = depUri => singleRequire(depUri, uri);

const specialDeps = {

module: { uri },

exports,

require

};

registry[uri] = Promise.all(depsNames.map(

depName => specialDeps[depName] || require(depName)

)).then(deps => {

factory(...deps);

return exports;

});

};

}

define(['./workbox-d9a5ed57'], (function (workbox) { 'use strict';

self.skipWaiting();

workbox.clientsClaim();

/\*\*

\* The precacheAndRoute() method efficiently caches and responds to

\* requests for URLs in the manifest.

\* See https://goo.gl/S9QRab

\*/

workbox.precacheAndRoute([{

"url": "registerSW.js",

"revision": "3ca0b8505b4bec776b69afdba2768812"

}, {

"url": "index.html",

"revision": "0.7e7jl0981k"

}], {});

workbox.cleanupOutdatedCaches();

workbox.registerRoute(new workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"), {

allowlist: [/^\/$/]

}));

workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/, new workbox.CacheFirst({

"cacheName": "images-cache",

plugins: [new workbox.ExpirationPlugin({

maxEntries: 50,

maxAgeSeconds: 2592000

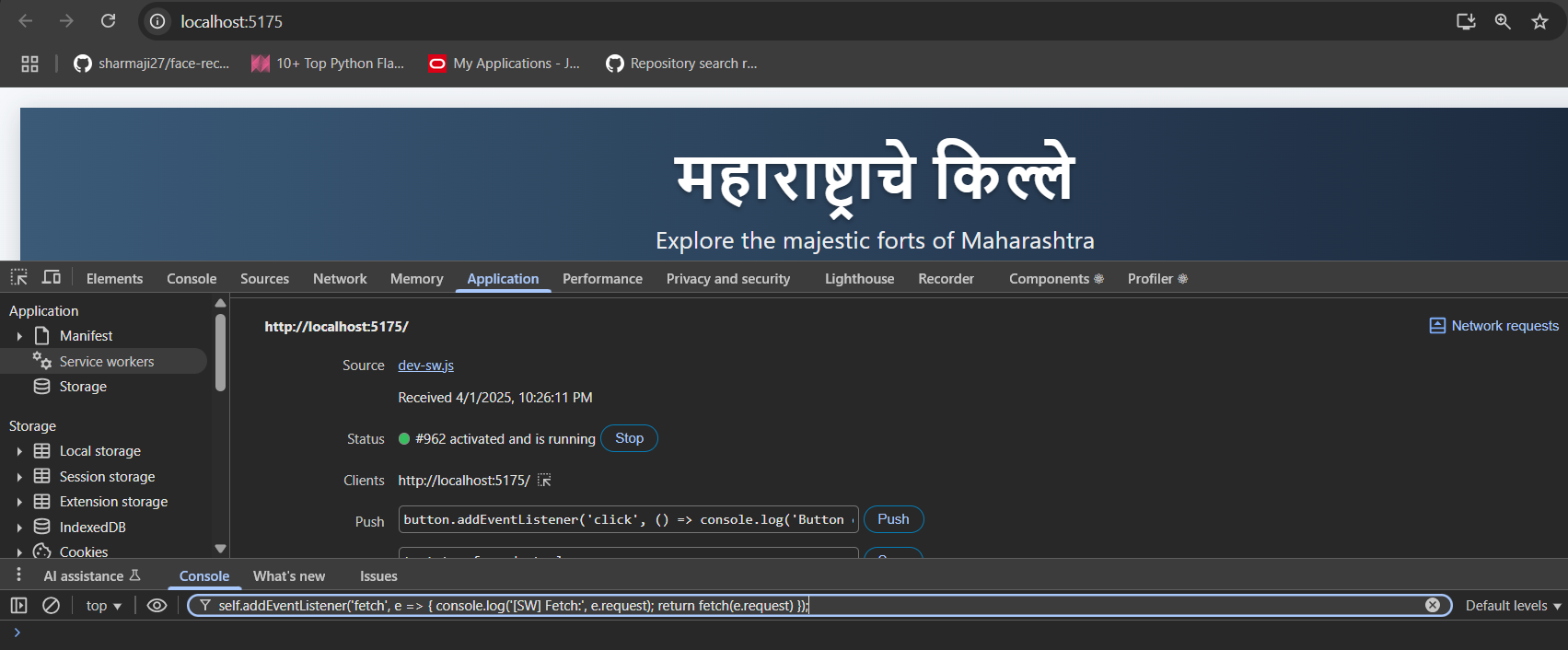
})]

}), 'GET');

}));

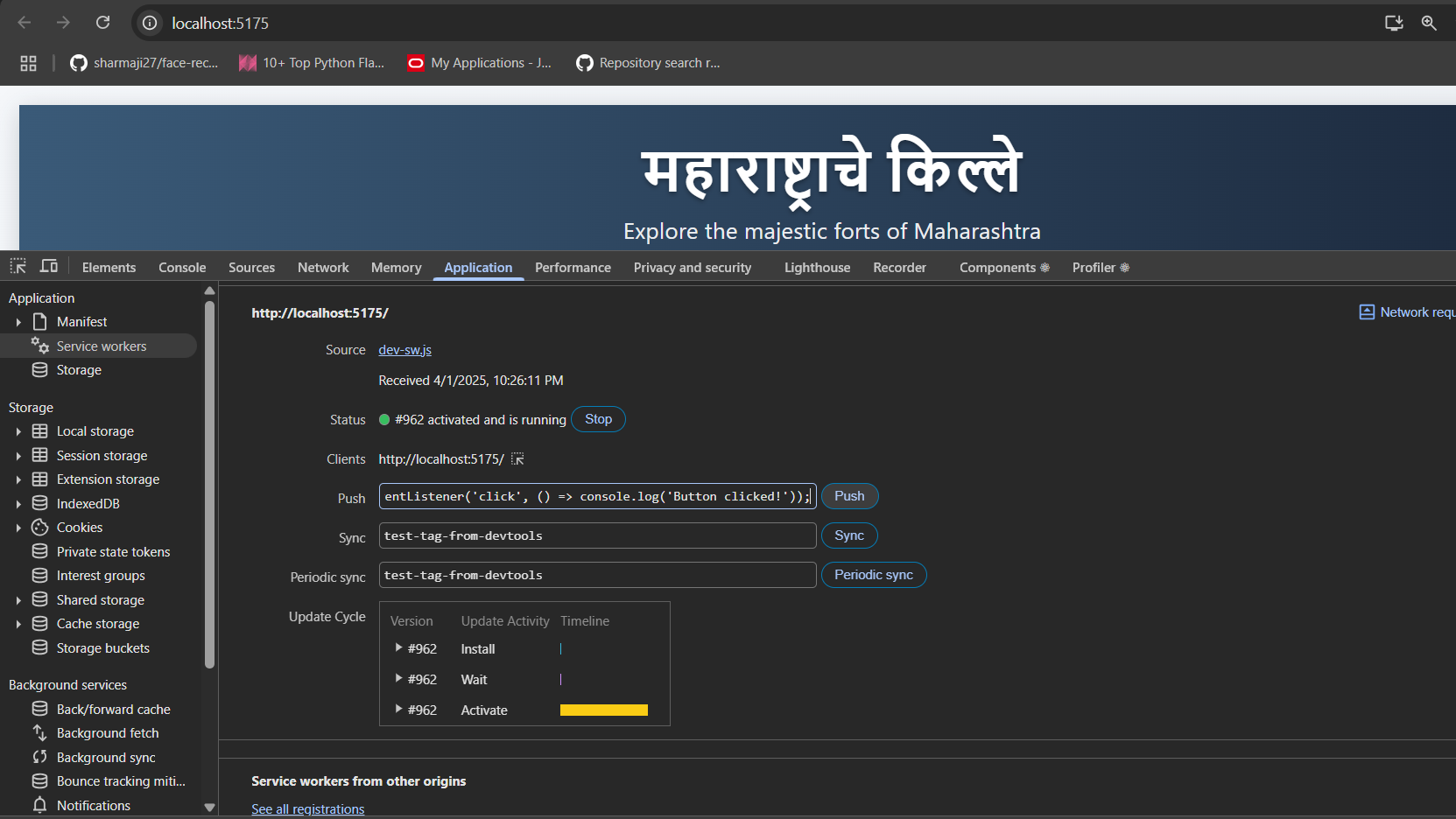
**Output**:

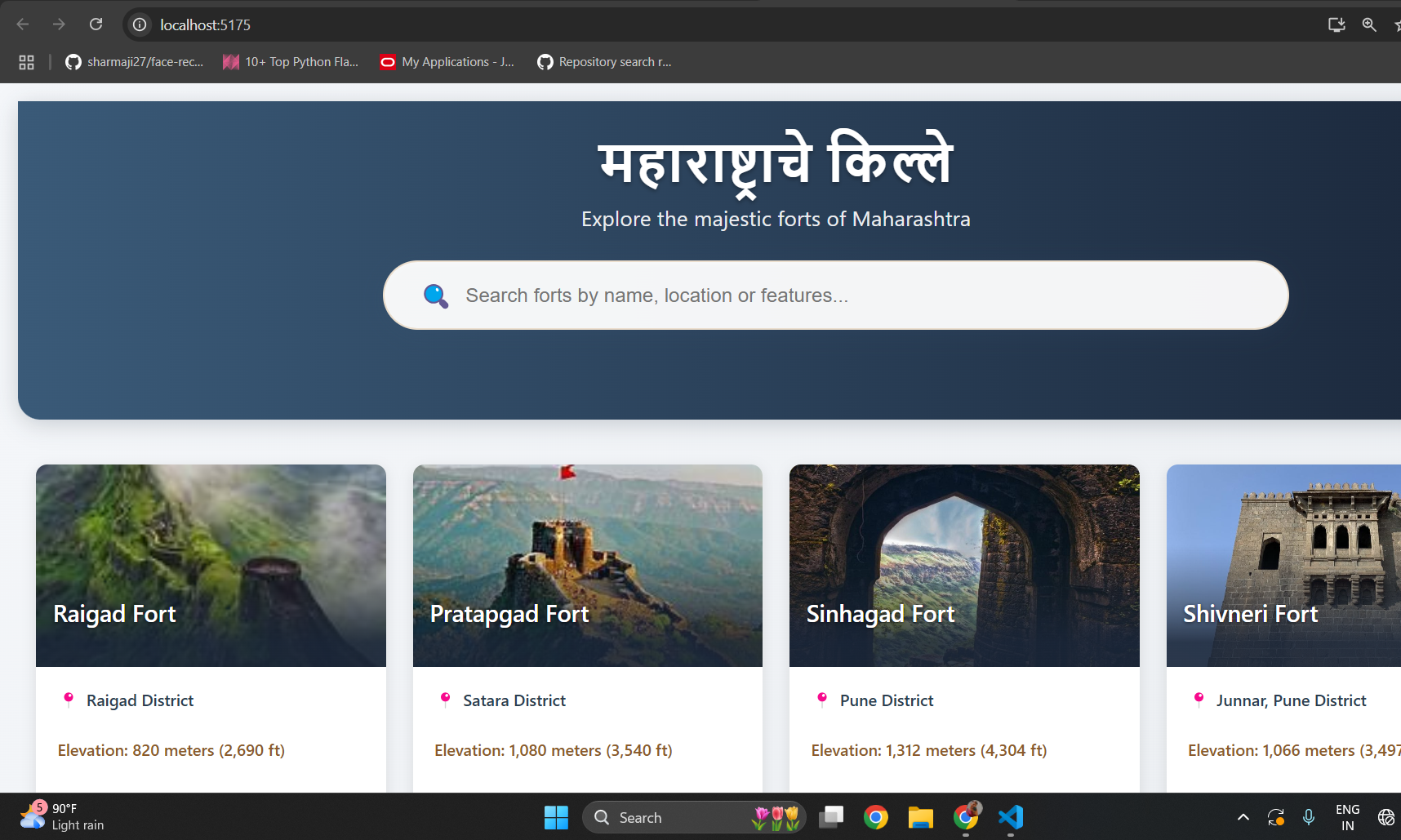
**Fetch event**

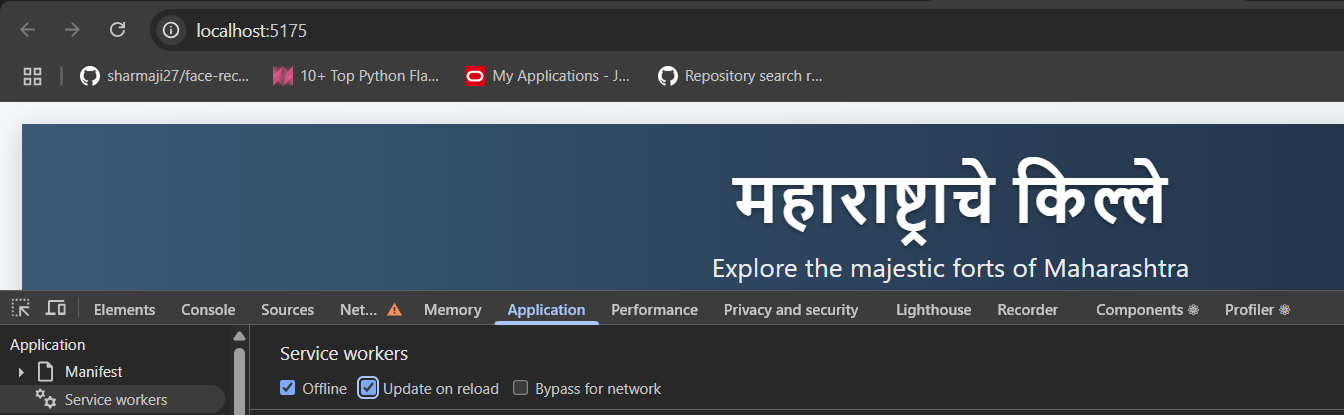
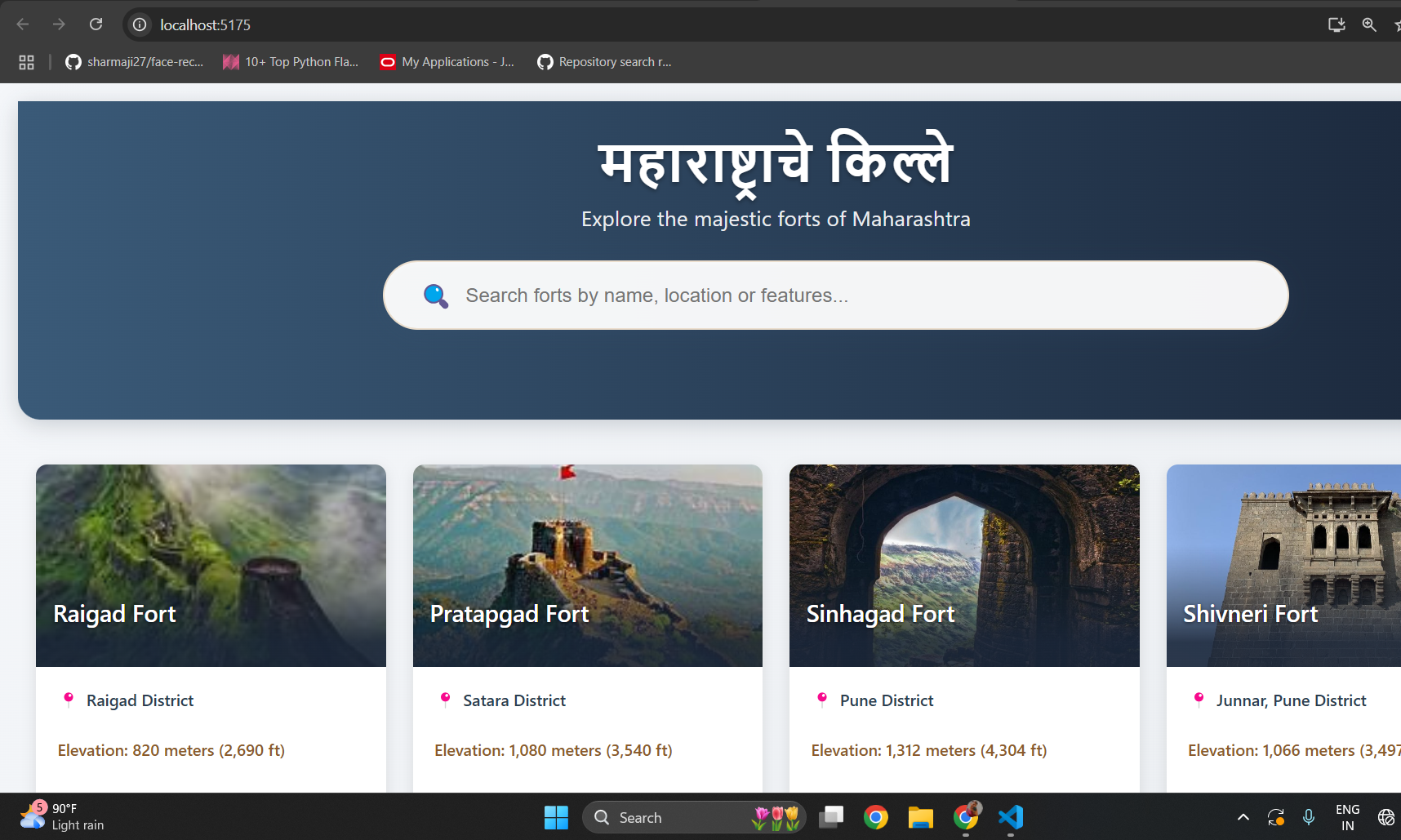


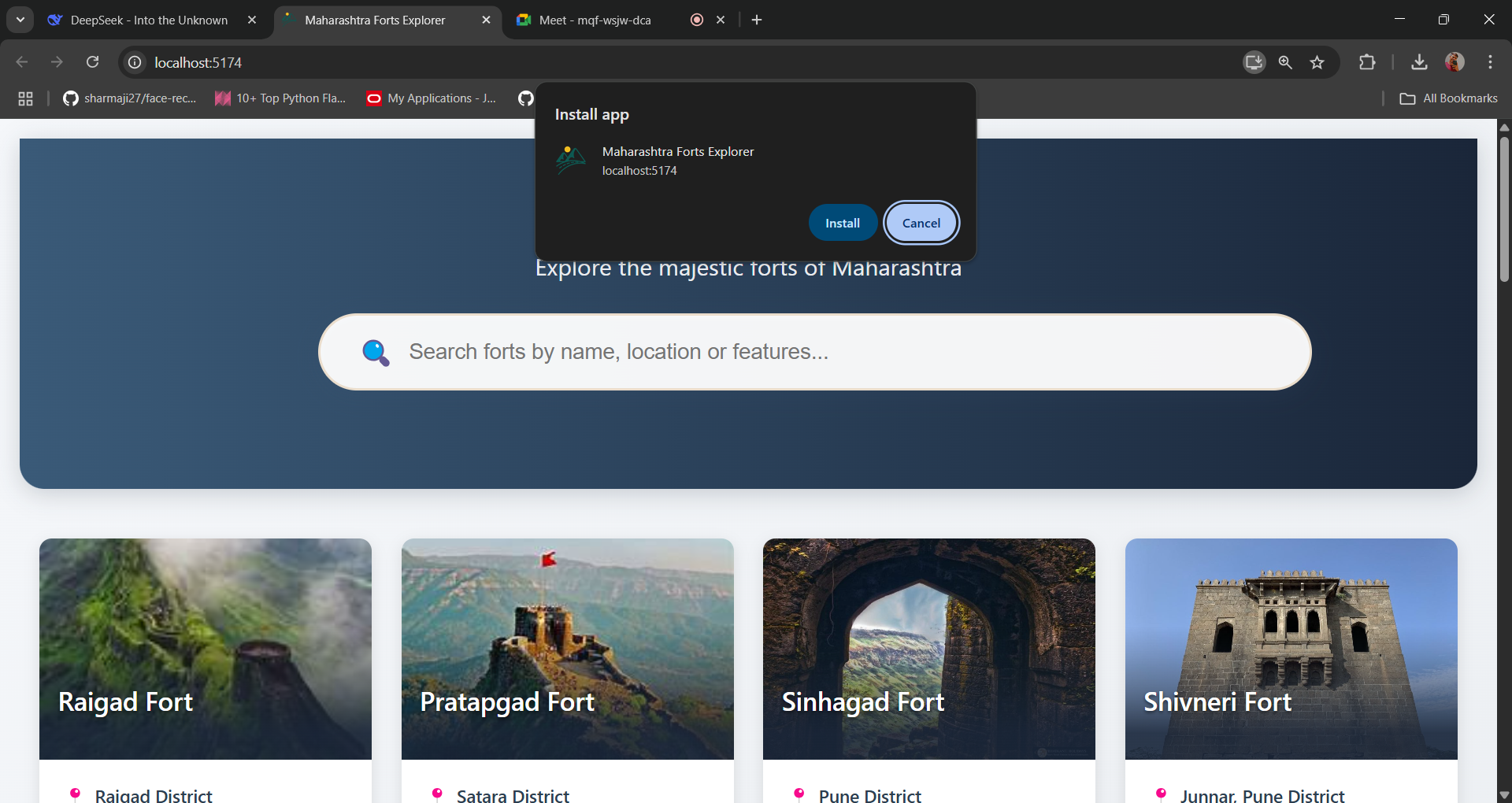
**Sync event and Push event**







**OFFLINE ACCESS  
  
Without Network**

**BY Download in System**

### Conclusion

Service workers play a vital role in making Progressive Web Apps (PWAs) more efficient, reliable, and user-friendly. They enable background processes such as caching (fetch event), delayed tasks for better connectivity (sync event), and push notifications (push event). By implementing these service worker events, the e-commerce PWA ensures better performance, offline accessibility, seamless user experience, and reduced server load. Using Workbox simplifies service worker management, making it easier to handle caching strategies, push notifications, and background synchronization, ultimately enhancing the reliability and responsiveness of the web application.