Name: Akanksha Shinde Class: D15C Roll No: 53

Experiment No. 8

Aim: To code and register a service worker, and complete the install and activation process for a new service worker for a 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.

What can we do with Service Workers?

- You can dominate **Network Traffic**

  You can manage all network traffic of the page and do any manipulations. For example, when the page requests a CSS file, you can send plain text as a response or when the page requests an HTML file, you can send a png file as a response. You can also send a true response too.

- You can **Cache**

You can cache any request/response pair with Service Worker and Cache API and you can access these offline content anytime.

- You can manage **Push Notifications**

  You can manage push notifications with Service Worker and show any information message to the user.

- You can **Continue**

  Although Internet connection is broken, you can start any process with Background Sync of Service Worker.

What can’t we do with Service Workers?

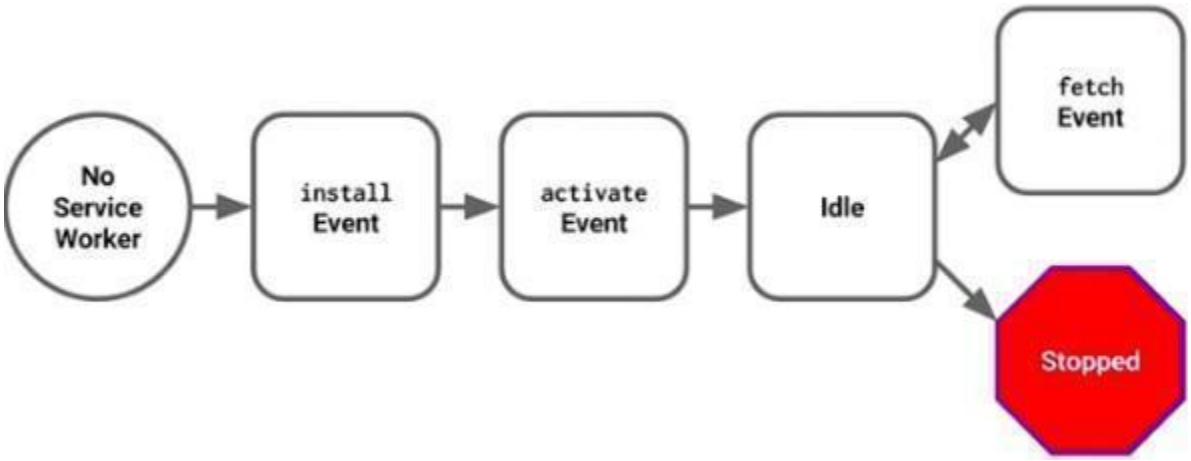
- You can’t access the **Window**

  You can’t access the window, therefore, You can’t manipulate DOM elements. But, you can communicate to the window through post Message and manage processes that you want.

- You can’t work it on **80 Port**

  Service Worker just can work on HTTPS protocol. But you can work on localhost during development.

Service Worker Cycle



A service worker goes through three steps in its life cycle:

- Registration
- Installation
- Activation

Registration

To install a service worker, you need to register it in your main JavaScript code. Registration tells the browser where your service worker is located, and to start installing it in the background.

CODE:

public/index.html

```
<!DOCTYPE html>
<html>
<head>
  <!--
    If you are serving your web app in a path other than the root, change the
    href value below to reflect the base path you are serving from.

    The path provided below has to start and end with a slash "/" in order for
    it to work correctly.

    For more details:
    * https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base
  -->
  <base href="/">

  <meta charset="UTF-8">
  <meta content="IE=Edge" http-equiv="X-UA-Compatible">
  <meta name="description" content="A new Flutter project.">

  <!-- iOS meta tags & icons -->
  <meta name="apple-mobile-web-app-capable" content="yes">
  <meta name="apple-mobile-web-app-status-bar-style" content="black">
  <meta name="apple-mobile-web-app-title" content="day35">
  <link rel="apple-touch-icon" href="icons/Icon-192.png">

  <title>HomeEase</title>
  <link rel="manifest" href="manifest.json">
</head>
<body>
  <!-- This script installs service_worker.js to provide PWA functionality to
       application. For more information, see:
       https://developers.google.com/web/fundamentals/primers/service-workers -->
  <script>
    var serviceWorkerVersion = null;
    var scriptLoaded = false;
    function loadMainDartJs() {
```

```
  if (scriptLoaded) {
    return;
  }
  scriptLoaded = true;
  var scriptTag = document.createElement('script');
  scriptTag.src = 'main.dart.js';
  scriptTag.type = 'application/javascript';
  document.body.append(scriptTag);
}

if ('serviceWorker' in navigator) {
  // Service workers are supported. Use them.
  window.addEventListener('load', function () {
    // Wait for registration to finish before dropping the <script> tag.
    // Otherwise, the browser will load the script multiple times,
    // potentially different versions.
    var serviceWorkerUrl = 'flutter_service_worker.js?v=' + serviceWorkerVersion;
    navigator.serviceWorker.register(serviceWorkerUrl)
      .then((reg) => {
        function waitForActivation(serviceWorker) {
          serviceWorker.addEventListener('statechange', () => {
            if (serviceWorker.state == 'activated') {
              console.log('Installed new service worker.');
              loadMainDartJs();
            }
          });
        }
        if (!reg.active && (reg.installing || reg.waiting)) {
          // No active web worker and we have installed or are installing
          // one for the first time. Simply wait for it to activate.
          waitForActivation(reg.installing ?? reg.waiting);
        } else if (!reg.active.scriptURL.endsWith(serviceWorkerVersion)) {
          // When the app updates the serviceWorkerVersion changes, so we
          // need to ask the service worker to update.
          console.log('New service worker available.');
          reg.update();
          waitForActivation(reg.installing);
        } else {
          // Existing service worker is still good.
          console.log('Loading app from service worker.');
          loadMainDartJs();
        }
      });

    // If service worker doesn't succeed in a reasonable amount of time,
    // fallback to plaint <script> tag.
    setTimeout(() => {
```

```
        if (!scriptLoaded) {
          console.warn(
            'Failed to load app from service worker. Falling back to plain <script> tag.',
          );
          loadMainDartJs();
        }
      }, 4000);
    });
  } else {
    // Service workers not supported. Just drop the <script> tag.
    loadMainDartJs();
  }
 </script>
</body>
</html>
```

Installation

Once the browser registers a service worker, installation can be attempted. This occurs if the service worker is considered to be new by the browser, either because the site currently doesn't have a registered service worker, or because there is a byte difference between the new service worker and the previously installed one.

A service worker installation triggers an install event in the installing service worker. We can include an install event listener in the service worker to perform some task when the service worker installs. For instance, during the install, service workers can precache parts of a web app so that it loads instantly the next time a user opens it (see caching the application shell). So, after that first load, you're going to benefit from instant repeat loads and your time to interactivity is going to be even better in those cases. An example of an installation event listener looks like this:

serviceworker.js

```
// Install the service worker and cache assets
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      console.log("Opened cache");
      return cache.addAll(urlsToCache);
    })
  );
});
```

});

Activation

Once a service worker has successfully installed, it transitions into the activation stage. If there are any open pages controlled by the previous service worker, the new service worker enters a waiting state. The new service worker only activates when there are no longer any pages loaded that are still using the old service worker. This ensures that only one version of the service worker is running at any given time.

When the new service worker activates, an activate event is triggered in the activating service worker. This event listener is a good place to clean up outdated caches.

Activate the service worker and clean up old caches

```
self.addEventListener("activate", (event) => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

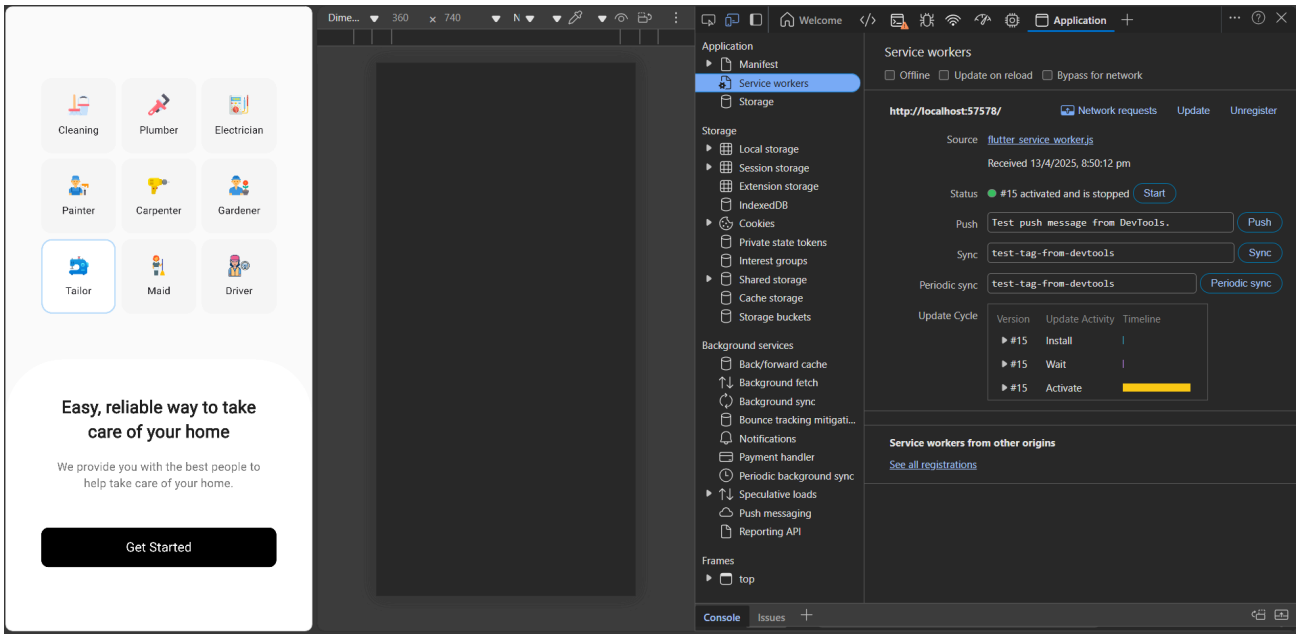
Therefore, serviceworker.js finally looks like this:

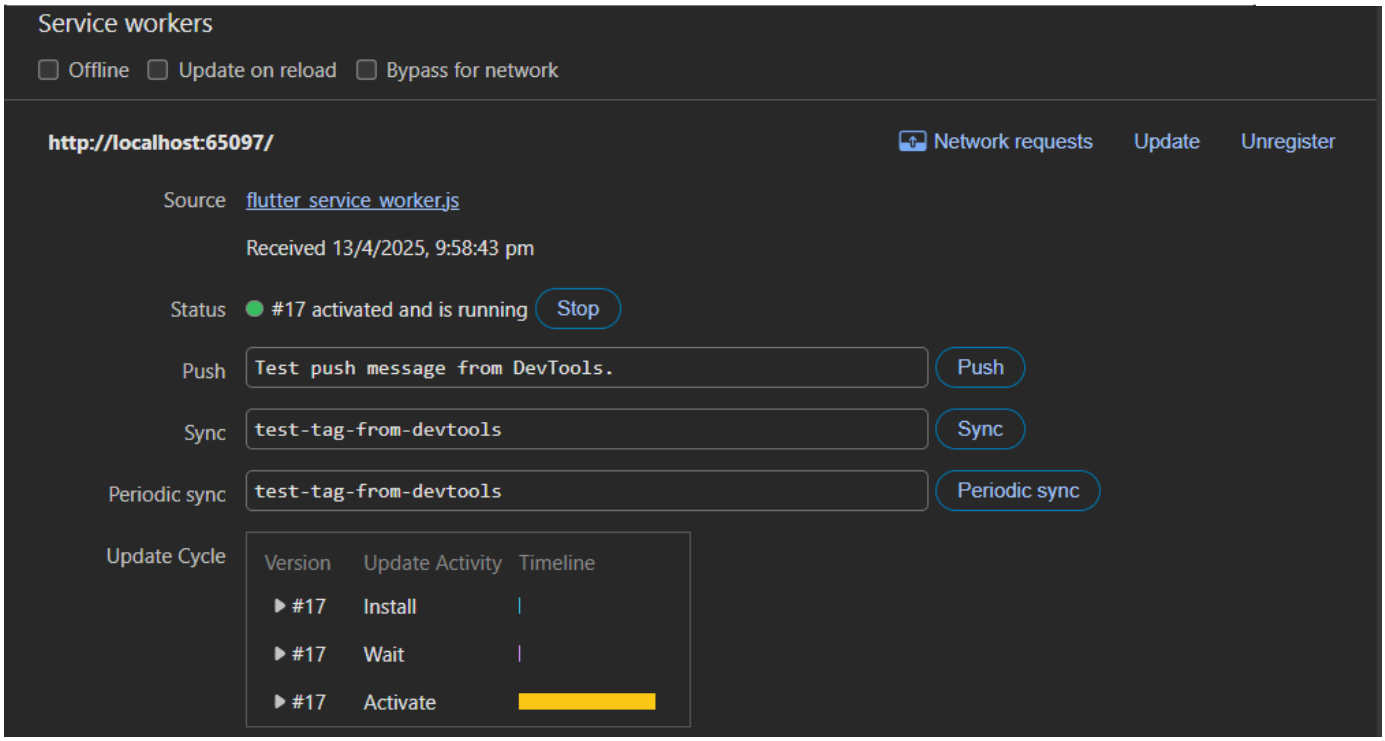
```
const CACHE_NAME = "homeease-cache-v1";
const urlsToCache = [
  "/",
  "/index.html",
  "/manifest.json",
  "/icons/icon-192x192.png",
  "/icons/icon-512x512.png",
  "/styles.css",
  "/script.js"
];
```

```
// Install the service worker and cache assets
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      console.log("Opened cache");
      return cache.addAll(urlsToCache);
    })
  );
});

// Fetch assets from the cache or network
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});

// Activate the service worker and clean up old caches
self.addEventListener("activate", (event) => {
  const cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

We now run the app -

On the browser:





Conclusion:

The aim was to implement a service worker for aProgressive Web Application (PWA), enabling offline functionality and improving performance. This involved coding and registering the service worker (serviceworker.js) to intercept network requests, cache essential resources, and enhance the offline experience. By completing the install and activation process, the PWA gained features like offline access and faster page loads, enhancing user experience and resilience to network issues.