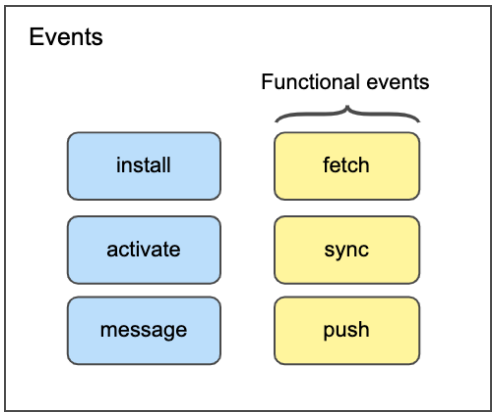
Name: Niyati V. Gaonkar
Class: D15B Roll no.: 17

MAD-PWD Experiment - 9

AIM: To implement Service worker events like fetch, sync and push for E-commerce PWA

THEORY:
Service workers are powerful JavaScript files that run in the background of a web application, separate from the main browser thread. They enable features like caching, push notifications, and background sync, allowing web applications to behave more like native applications. Service workers have access to various events that allow them to intercept and handle network requests, including the fetch event.



FETCH

Here's a breakdown of the fetch event in service workers:

1. Purpose:

The fetch event allows service workers to intercept and handle network requests made by the web application. It provides an opportunity to modify the request, respond with a cached resource, fetch from the network, or even generate a custom response.

2. Event Triggering:

The fetch event is triggered whenever a network request is made by the web application that the service worker controls. This includes requests for HTML pages, CSS stylesheets, JavaScript files, images, and any other resources.

3. Handling:

When a fetch event is triggered, the service worker can respond to the request in several ways:

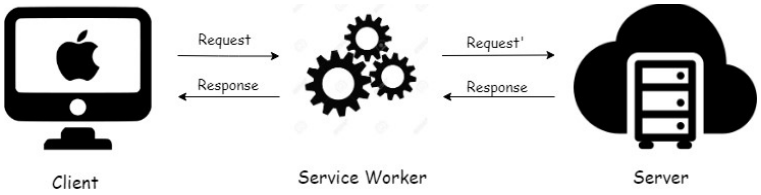
- Cache Handling: The service worker can check if the requested resource is available in the cache. If it is, the service worker can respond with the cached version, providing offline support and improving performance by serving resources from the cache.
- Network Fetching: If the requested resource is not available in the cache or if the service worker is configured to always fetch from the network, it can initiate a fetch request to the network to obtain the resource. This allows the service worker to update the cache with the latest version of the resource or fetch resources that are not cacheable.
- Custom Responses: The service worker can generate custom responses based on the request, such as serving a fallback response if the network is offline, redirecting requests to different URLs, or injecting custom content into the response.

4. Example Use Cases:

- Offline Support: By caching resources using the fetch event, service workers enable offline access to web applications by serving cached resources when the network is unavailable.
- Performance Optimization: Service workers can improve performance by serving cached resources from the local cache, reducing the dependency on the network and speeding up page load times.
- Dynamic Content: Service workers can intercept requests and fetch dynamic content from the server or other sources, allowing web applications to display real-time updates without reloading the page.

5. Considerations:

- CORS and HTTPS: Service workers have restrictions on handling cross-origin requests (CORS) and require HTTPS for security reasons.
- Cache Management: Proper cache management is essential to ensure that cached resources are up to date and that the cache does not consume excessive storage space.



PUSH

The Push API in service workers allows web applications to receive push notifications from a server even when the application is not actively running in the browser. Here's an overview of the Push API in service workers:

1. Purpose:

The Push API enables web applications to receive push notifications from a server, allowing them to deliver real-time updates, alerts, or messages to users, even when the web application is not open in the browser.

2. Subscription:

To receive push notifications, the web application first needs to subscribe to a push service provided by the browser. This typically involves requesting permission from the user to receive notifications and obtaining a unique endpoint (endpoint URL) from the push service.

3. Event Triggering:

When a push notification is sent to the subscribed endpoint by the server, a push event is triggered in the service worker associated with the web application, regardless of whether the application is currently open or not.

4. Handling:

When a push event is triggered, the service worker can handle it in various ways:

- Notification Display: The service worker can use the received data to construct a notification message and display it to the user using the Notification API. This allows the web application to notify users of new updates, messages, or events.
- Data Processing: In addition to displaying notifications, the service worker can process the received data and perform background tasks, such as updating local data, prefetching content, or syncing data with the server.

5. Example Use Cases:

- Real-Time Updates: Web applications can use push notifications to deliver real-time updates to users, such as news alerts, social media notifications, or chat messages.

- Event Reminders: Push notifications can be used to remind users of upcoming events, appointments, or deadlines, even when the web application is not actively used.
- Offline Tasks: Service workers can use push notifications to trigger background tasks or data synchronization processes, allowing web applications to perform tasks even when the user is not actively interacting with the application.

6. Considerations:

- Permission: Push notifications require explicit permission from the user. The web application must request permission to display notifications using the Notifications API before subscribing to push notifications.
- Security: Push notifications are subject to security considerations, including ensuring that notifications are delivered securely over HTTPS and preventing unauthorized access to the push service endpoint.

SYNC:
The Sync API in service workers allows web applications to perform background synchronization tasks, enabling them to sync data with a server even when the application is offline or the browser is closed. Here's an overview of the Sync API in service workers:

1. Purpose:

The Sync API enables web applications to schedule and perform background synchronization tasks with a server, allowing them to update local data, send queued requests, or perform other actions even when the application is offline or the browser is closed.

2. Event Triggering:

The Sync API is triggered by the browser when network connectivity is available and the service worker is registered. The browser schedules a sync event for the service worker, indicating that it can perform synchronization tasks.

3. Handling:

When a sync event is triggered, the service worker can handle it in various ways:

- Data Synchronization: The service worker can synchronize data with the server by sending queued requests, updating local caches, or performing other actions that require network connectivity.

- Error Handling: If synchronization fails due to network issues or server errors, the service worker can handle errors gracefully by retrying the synchronization task later or displaying an error message to the user.

4. Background Sync Queue:

The Sync API works in conjunction with a background sync queue, which allows web applications to queue synchronization tasks when the application is offline or the browser is closed. The browser automatically retries queued tasks when network connectivity is restored and the service worker is registered.

5. Example Use Cases:

- Offline Data Sync: Web applications can use the Sync API to synchronize offline changes with a server when the application comes back online, ensuring that local data is kept up to date with the server.
- Background Tasks: Service workers can perform background tasks, such as sending analytics data, uploading files, or updating cached content, without requiring user interaction or keeping the application open in the browser.

6. Considerations:

- Browser Support: The Sync API is supported in modern browsers that support service workers, such as Chrome, Firefox, and Edge. Older browsers or browsers without service worker support do not support the Sync API.
- Limitations: The Sync API has limitations, such as a maximum delay for scheduled sync events and a limited queue size for queued synchronization tasks. Developers should be aware of these limitations when implementing background synchronization tasks.

IMPLEMENTATION:

service-worker.js:

```
const CACHE_NAME = 'ecommerce-v1';
const urlsToCache = [
    '/',
    'index.html',
    'style.css',

    'images/products/n8.jpg',
];
```

```
self.addEventListener('install', event => {
    event.waitUntil(
        caches.open(CACHE_NAME)
            .then(cache => {
                return cache.addAll(urlsToCache);
            })
    );
});

self.addEventListener('activate', event => {
    event.waitUntil(
        caches.keys().then(cacheNames => {
            return Promise.all(
                cacheNames.map(cacheName => {
                    if (cacheName !== CACHE_NAME) {
                        return caches.delete(cacheName);
                    }
                })
            );
        })
    );
});

self.addEventListener('fetch', event => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                if (response) {
                    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;
                    });
            })
    );
});

self.addEventListener('push', event => {
    const title = 'E-commerce';
    const options = {
        body: event.data.text()
    };

    event.waitUntil(
        self.registration.showNotification(title, options)
    );
});

self.addEventListener('sync', event => {
    if (event.tag === 'sync-products') {
        event.waitUntil(syncProducts());
    }
});

function syncProducts() {
    // Implement syncing logic here
    console.log('Syncing products...');
}
```

index.html

Taking permission for push notification.

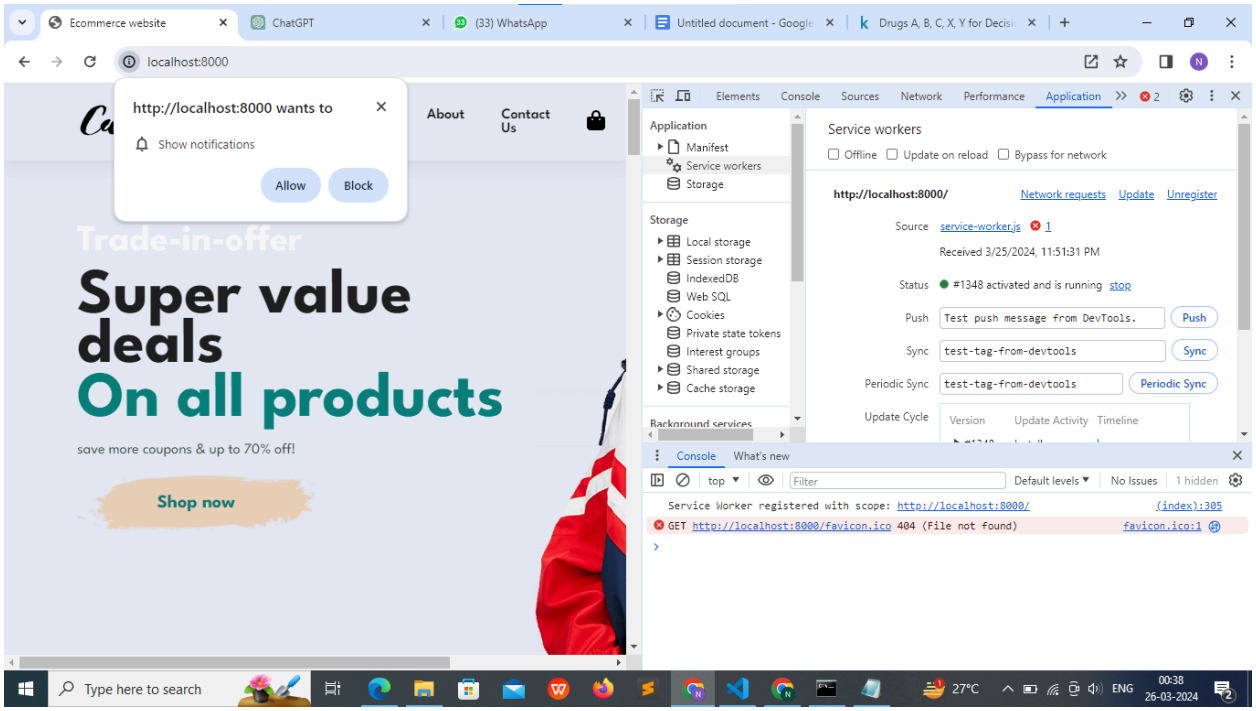
```
<script>
    if ('serviceWorker' in navigator) {
        window.addEventListener('load', () => {
            navigator.serviceWorker.register('service-worker.js')
                .then(registration => {
```

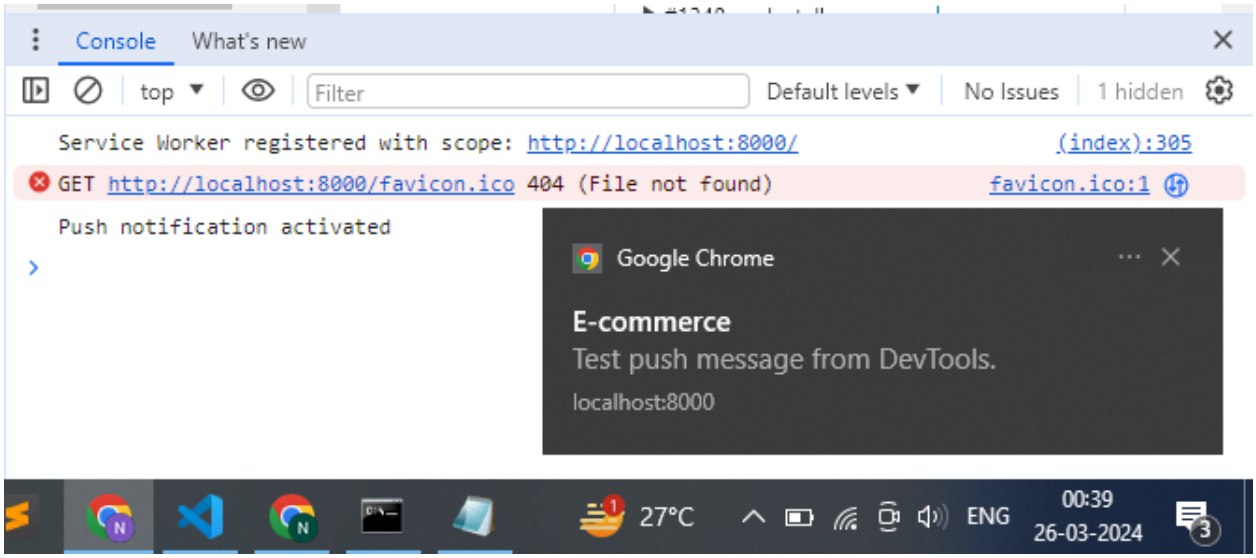
```
            console.log('Service Worker registered with scope:', registration.scope);
                        Notification.requestPermission().then(function(permission) {
                        if (permission === 'granted') {
                            console.log('Push notification activated');
                        }
                    });
            })
            .catch(error => {
                console.error('Service Worker registration failed:', error);
            });
    });
}

</script>
```

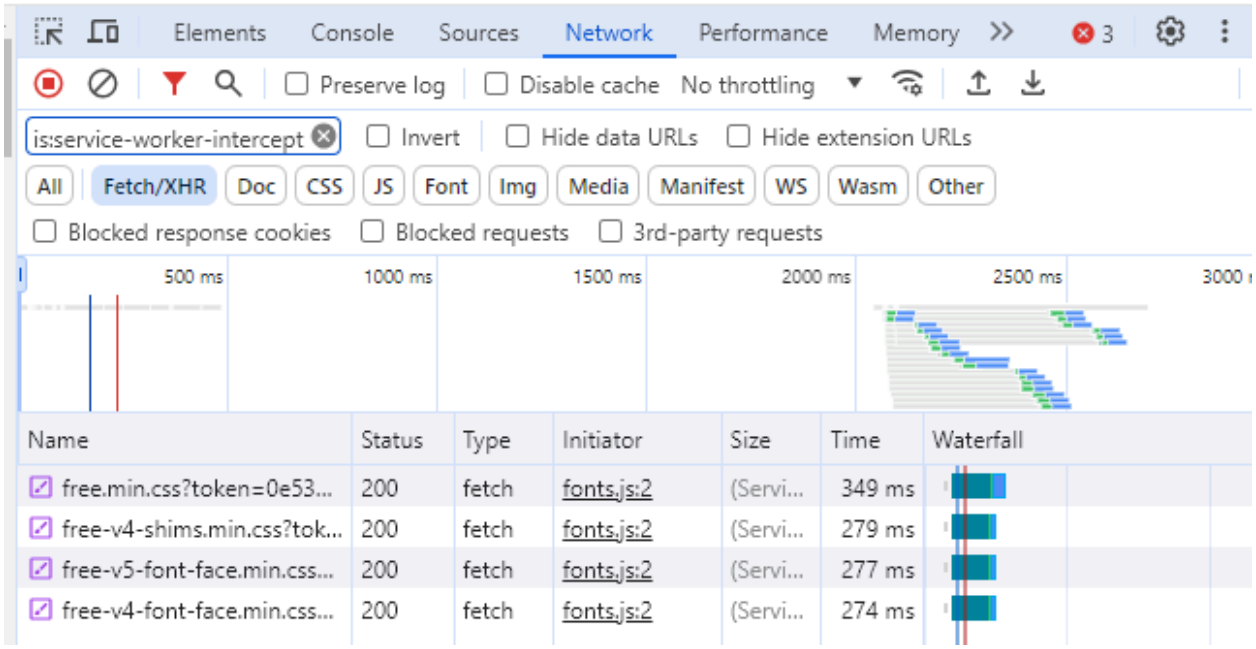
OUTPUT:

Push:





Fetch:



CONCLUSION: Thus we have understood the service worker functional events like fetch,sync and push and implemented all of them. In conclusion, service workers empower web applications with powerful capabilities such as intercepting fetch requests, receiving push notifications, and performing background synchronization tasks. Leveraging these APIs enhances offline functionality, real-time communication, and background processing, ultimately improving user experience and application reliability in modern web development.