EXPERIMENT NO. 8

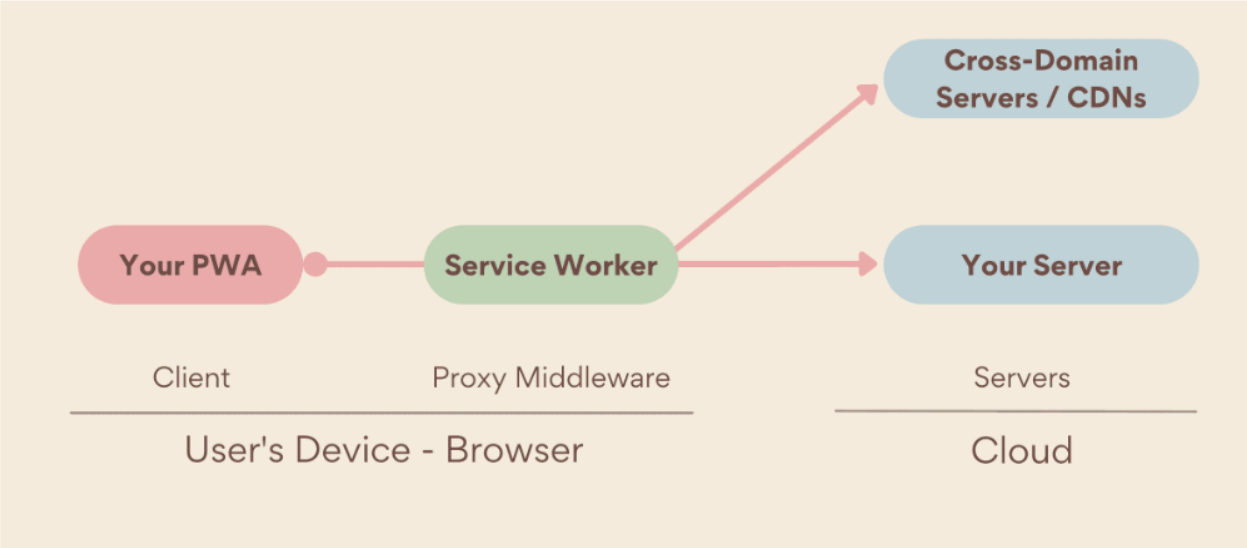
Aim: To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA.

Theory:

Service Workers are a virtual proxy between the browser and the network. They make it possible to properly cache the assets of a website and make them available when the user's device is offline.

They run on a separate thread from the main JavaScript code of our page, and don't have any access to the DOM structure. This introduces a different approach from traditional web programming — the API is non-blocking, and can send and receive communication between different contexts. You are able to give a Service Worker something to work on, and receive the result whenever it is ready using a Promise-based approach.

Service workers can do more than offering offline capabilities, including handling notifications or performing heavy calculations. Service workers are quite powerful as they can take control over network requests, modify them, serve custom responses retrieved from the cache, or synthesize responses completely.



Key Points about Service Workers:

1. Background Processing: Service workers run in the background, independent of the web page, enabling them to perform tasks like caching resources or handling push notifications even when the web app is not open.

2. Intercepting Network Requests: Service workers can intercept network requests made by the web app, allowing you to serve cached responses, make modifications to requests, or handle requests based on certain conditions (e.g., offline fallbacks).

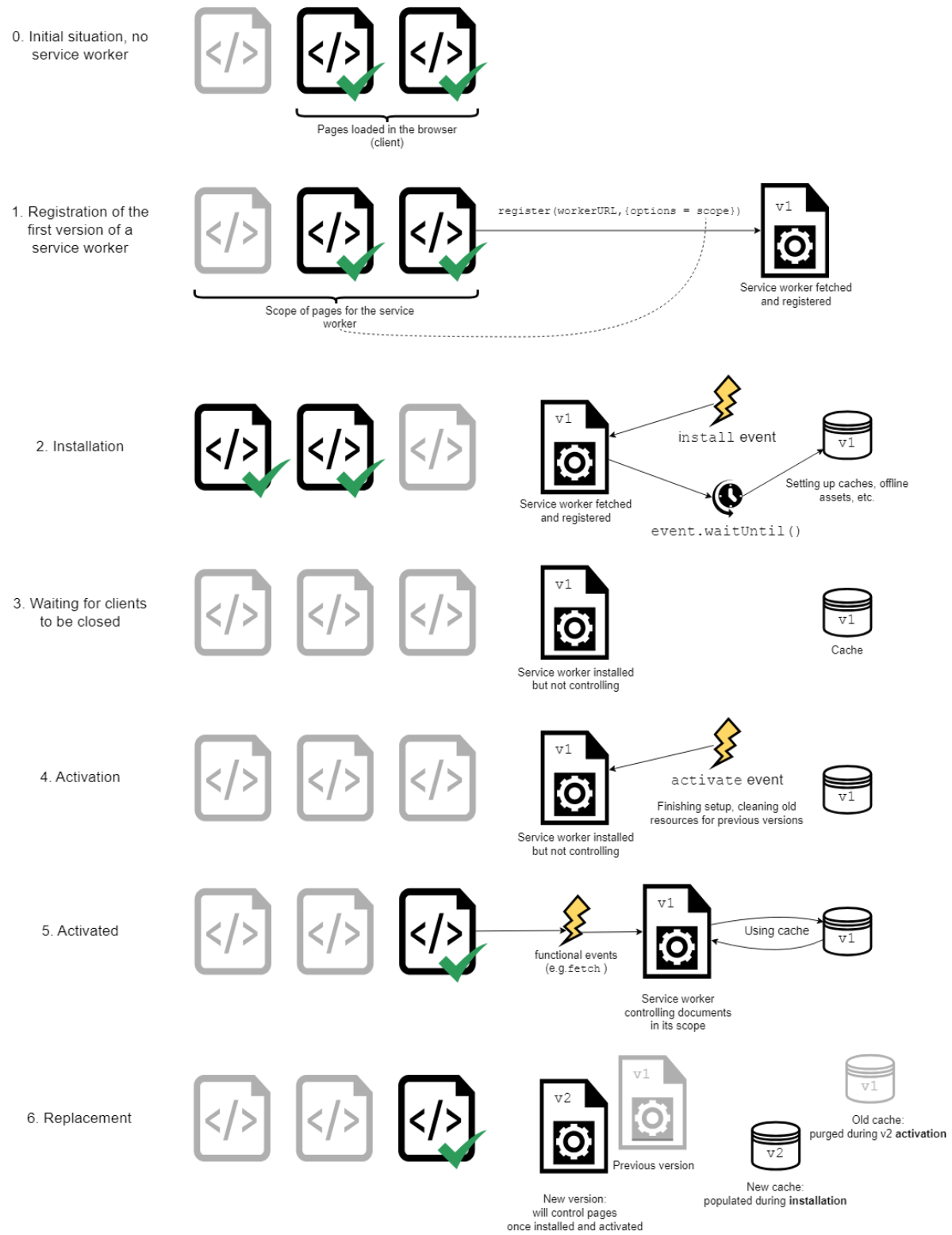
3. Caching: Service workers have access to a cache storage API, which allows you to cache resources such as HTML, CSS, JavaScript, images, and API responses. This enables web apps to work offline or in low-connectivity environments by serving cached content when network access is unavailable.

4. Push Notifications: Service workers can receive push notifications from a server and show them to the user, even when the web app is not open. This feature enables web apps to engage users with timely updates and messages.

5. Background Sync: Service workers can schedule tasks to be performed in the background, even if the web app is closed. This is useful for syncing data with a server when a connection is available, ensuring that the web app stays up to date.

6. HTTPS Requirement: Service workers require a secure context (HTTPS) to ensure that the service worker script and the resources it caches are not tampered with by malicious actors.

7. Lifecycle: Service workers have a lifecycle that includes registration, installation, activation, and termination. Understanding this lifecycle is important for managing the behavior of your service worker.



Service Worker Life Cycle

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.

main.js

```
if ('serviceWorker' in navigator) { navigator.serviceWorker.register('/service-worker.js')
.then(function(registration) {
console.log('Registration successful, scope is:', registration.scope);
})
.catch(function(error) {
console.log('Service worker registration failed, error:', error);
});
}
```

This code starts by checking for browser support by examining navigator.serviceWorker. The service worker is then registered with navigator.serviceWorker.register, which returns a promise that resolves when the service worker has been successfully registered. The scope of the service worker is then logged with registration.scope. If the service worker is already installed, navigator.serviceWorker.register returns the registration object of the currently active service worker.

The scope of the service worker determines which files the service worker controls, in other words, from which path the service worker will intercept requests. The default scope is the location of the service worker file, and extends to all directories below. So if service-worker.js is located in the root directory, the service worker will control requests from all files at this domain.

You can also set an arbitrary scope by passing in an additional parameter when registering. For example: main.js

```
navigator.serviceWorker.register('/service-worker.js', { scope: '/app/'
});
```

In this case we are setting the scope of the service worker to /app/, which means the service worker will control requests from pages like /app/, /app/lower/ and /app/lower/lower, but not from pages like /app or /, which are higher.

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:

service-worker.js

```
// Listen for install event, set callback
self.addEventListener('install', function(event) {
// Perform some task
});
```

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 (see the Offline Cookbook for an example).

service-worker.js

```
self.addEventListener('activate', function(event) {
// Perform some task
});
```

Once activated, the service worker controls all pages that load within its scope, and starts listening for events from those pages. However, pages in your app that were loaded before the service worker activation will not be under service worker control. The new service worker will only take over when you close and reopen your app, or if the service worker calls clients.claim(). Until then, requests from this page will not be intercepted by the new service worker. This is intentional as a way to ensure consistency in your site.

Steps:

Before a service worker takes control of your page, it must be registered for your PWA. That means the first time a user comes to your PWA, network requests will go directly to your server because the service worker is not yet in control of your pages.

After checking if the browser supports the Service Worker API, your PWA can register a service worker. When loaded, the service worker sets up shop between your PWA and the network, intercepting requests and serving the corresponding responses.

```
<script>
      if ('serviceWorker' in navigator) {
        window.addEventListener('load', function() {
          navigator.serviceWorker.register('/service-worker.js').then(function(registration) {
            // Registration was successful
            console.log('ServiceWorker registration successful with scope: ',
registration.scope);
          }, function(err) {
            // Registration failed
            console.log('ServiceWorker registration failed: ', err);
          });
        });
      }
```

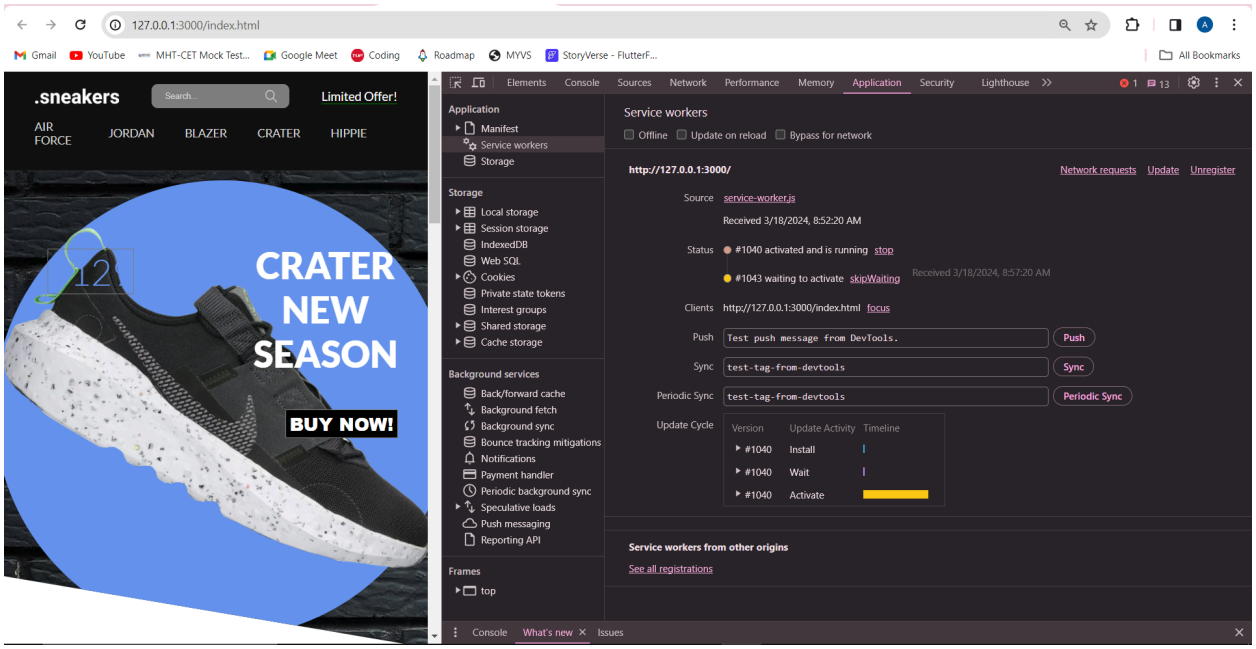
```
</script>
```

Verify if a service worker is registered

To verify if a service worker is registered, use developer tools in your favorite browser.

In Firefox and Chrome-based browsers (Microsoft Edge, Google Chrome, or Samsung Internet):

1.Open developer tools, then click the Application tab.
2.In the left pane, select Service Workers.
3.Check that the service worker's script URL appears with the status "Activated". (You'll learn what this status means in the lifecycle section in this chapter). On Firefox the status can be "Running" or "Stopped".



Scope

The folder your service worker sits in determines its scope. A service worker that lives at example.com/my-pwa/sw.js can control any navigation at the my-pwa path or below, such as example.com/my-pwa/demos/. Service workers can only control items (pages, workers, collectively "clients") in their scope. Scope applies to browser tabs and PWA windows.

Only one service worker per scope is allowed. When active and running, only one instance is typically available no matter how many clients are in memory (such as PWA windows or browser tabs).

Code:

service-worker.js

```
const CACHE_NAME = 'nike-store-cache-v1';

// Define the files to cache
const urlsToCache = [
  '/',
  '/index.html',
  '/style.css',
  '/app.js',
  // Add other files you want to cache
];

// Install the service worker
self.addEventListener('install', function(event) {
  // Perform installation process
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});

// Activate the service worker
```

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

// Fetch resources from cache or network
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }
        // Clone the request
        var fetchRequest = event.request.clone();

        // Fetch from network
        return fetch(fetchRequest).then(
          function(response) {
            // Check if valid response
```

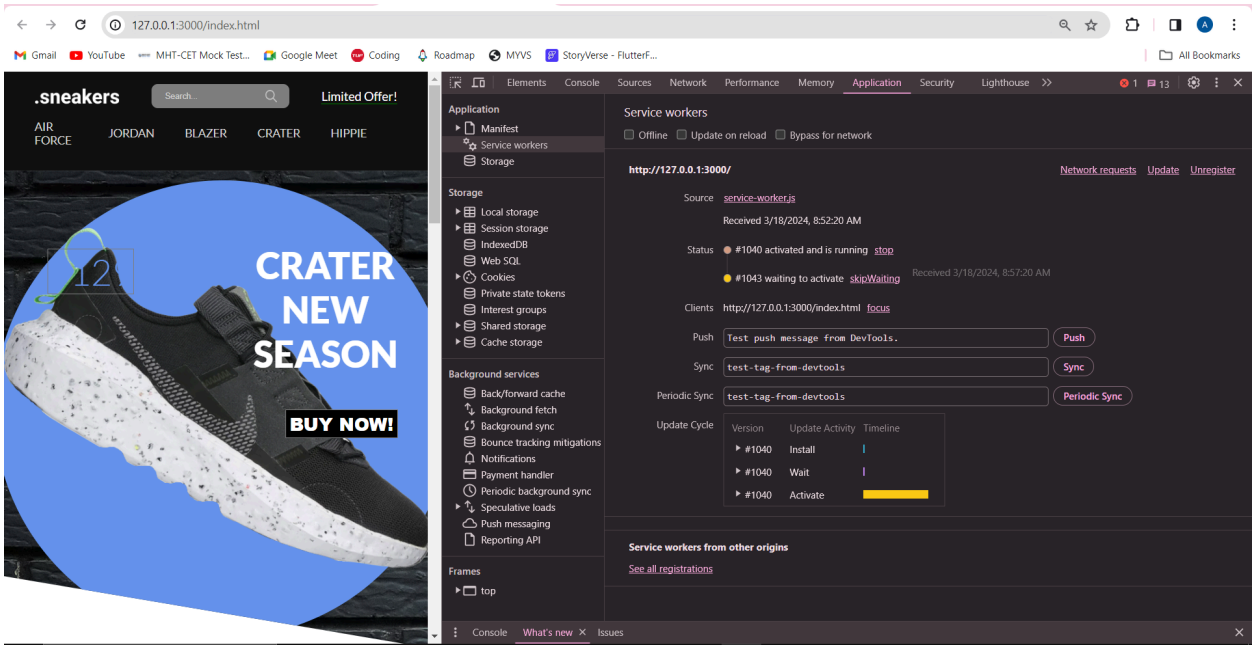
```
        if (!response || response.status !== 200 || response.type !== 'basic') {
          return response;
        }

        // Clone the response
        var responseToCache = response.clone();

        // Cache the fetched response
        caches.open(CACHE_NAME)
          .then(function(cache) {
            cache.put(event.request, responseToCache);
          });

        return response;
      }
    );
  })
);
});
```

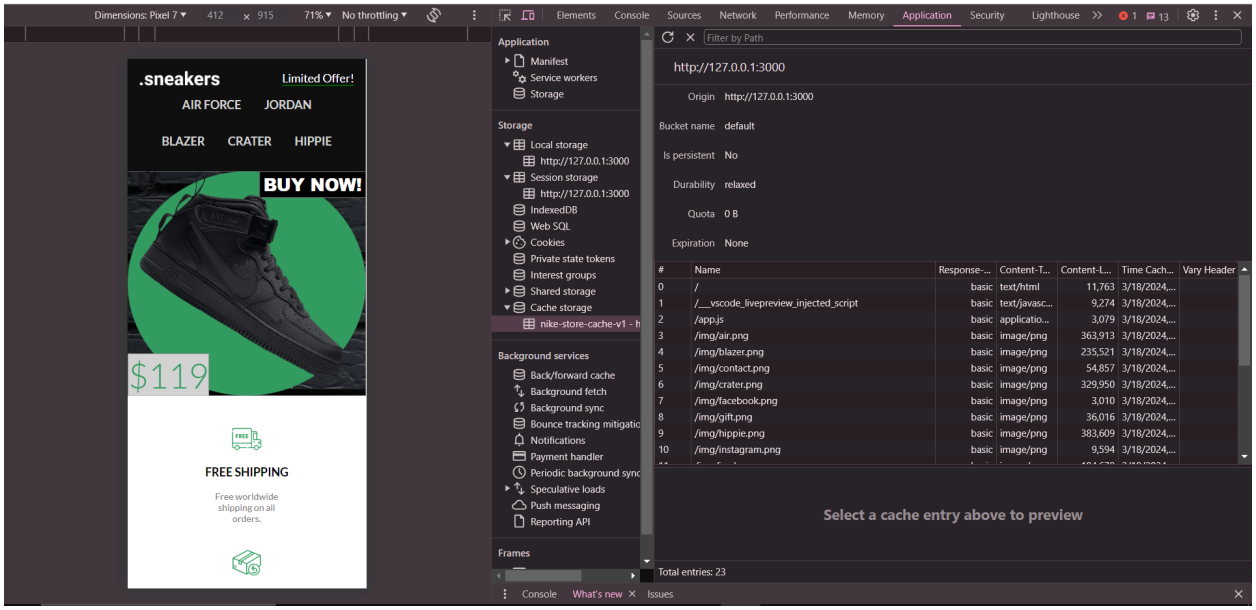
Index.html

```
  <script>
    if ('serviceWorker' in navigator) {
      window.addEventListener('load', function() {
        navigator.serviceWorker.register('/service-worker.js').then(function(registration) {
          // Registration was successful
          console.log('ServiceWorker registration successful with scope: ',
registration.scope);
        }, function(err) {
          // Registration failed
          console.log('ServiceWorker registration failed: ', err);
```

```
        });
      });
    }
  </script>
```

Output:





Conclusion:

In conclusion, the experiment of coding and registering a service worker for the E-commerce Progressive Web App (PWA) was successful. By implementing the service worker, we were able to improve the reliability and performance of the web application. The service worker allowed us to cache important resources, such as HTML, CSS, and JavaScript files, ensuring that the app could still function even when the user was offline or had a poor internet connection. Additionally, by intercepting network requests, the service worker enabled us to serve cached responses quickly, providing a smoother user experience.

Completing the install and activation process for the new service worker ensured that it was correctly registered with the browser and ready to handle requests. This experiment highlights the importance of service workers in building PWAs that are fast, reliable, and engaging, similar to native mobile applications.