EXPERIMENT : 9

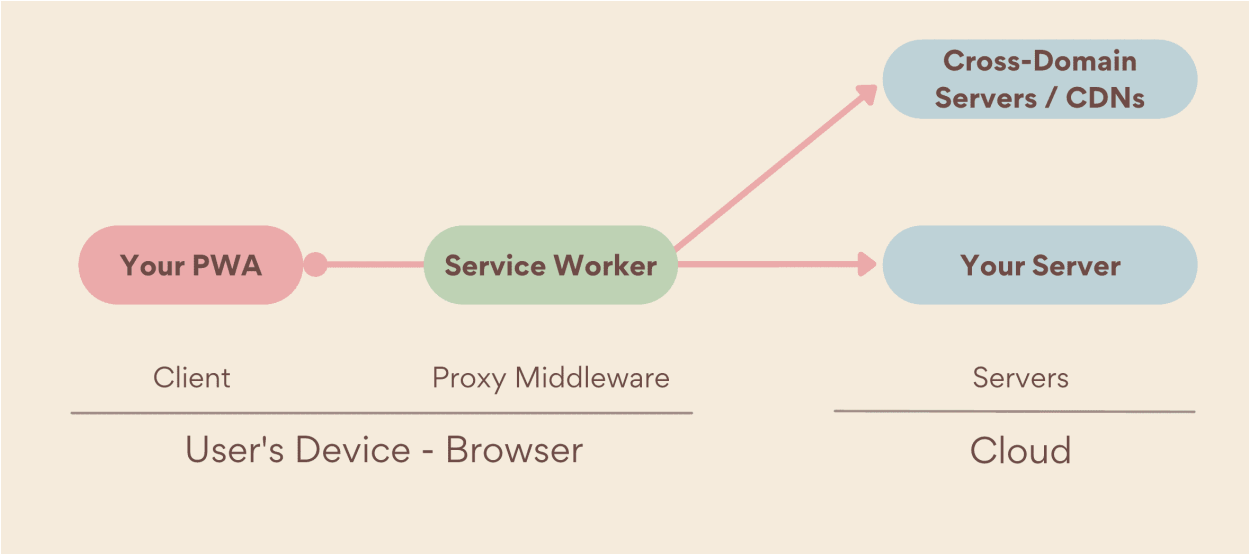
Aim : To implement Service worker events like fetch, sync and push for E-commerce PWA

Theory :

Service workers

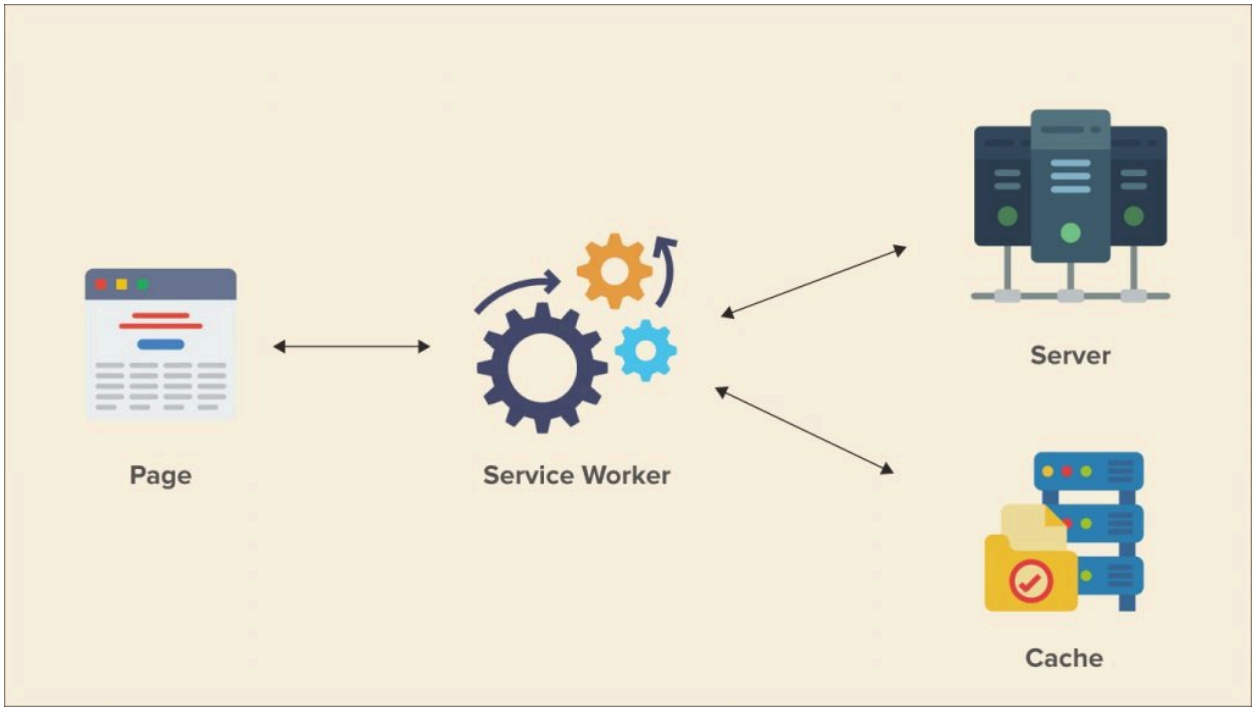
Users expect apps to start on slow or flaky network connections, or even when offline. They expect the content they've most recently interacted with, such as media tracks or tickets and itineraries, to be available and usable. When a request isn't possible, they expect the app to tell them instead of silently failing or crashing. And users wish to do it all quickly. As we can see in this study Milliseconds make millions, even a 0.1 second improvement in load times can improve conversion by up to 10%. In summary: users expect PWAs to be reliable and that's why we have service workers.

Hello service workers



When an app requests a resource covered by the service worker's scope, including when a user is offline, the service worker intercepts the request, acting as a network proxy. It can then decide if it should serve the resource from the cache via the Cache Storage API, from the network as normally would happen without a service worker, or create it from a local algorithm. This lets you provide a similar experience to that provided by a platform app. It can even work entirely off line.

Not all browsers support service workers. Even when present your service worker won't be available on first load or while it's waiting to activate. Therefore, treat it as optional and do not require it for core functionality.



Registering a service worker

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.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register("/serviceworker.js");
}
```

Note: There is only one service worker per PWA, but that doesn't mean you need to place the code only in one file. A service worker can include other files using importScripts in every browser or using ECMAScript module imports in some modern browsers.

Verify if a service worker is registered

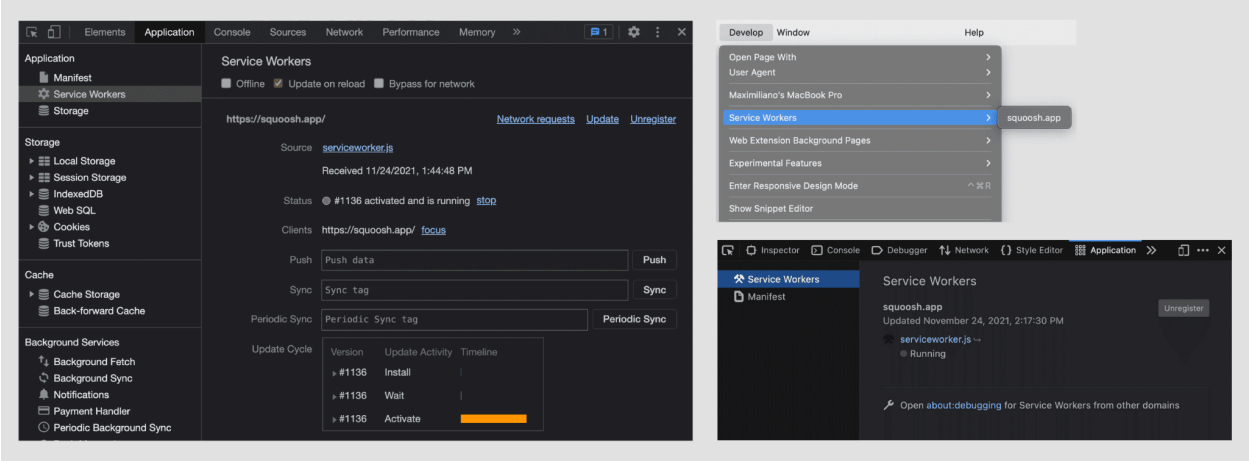
To verify if a service worker is registered, use developer tools in your favorite browser.

In Firefox and Chromium-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".

In Safari:

1. Click the Develop menu then the Service Workers submenu.
2. Check that an entry with the current origin appears in the submenu. It opens an inspector over the service worker's context.



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).

Warning: You should set the scope of your service worker as close to the root of your app as possible. This is the most common setup as it lets the service worker intercept all the requests related to your PWA. Don't put it inside, for instance, a JavaScript folder or have it loaded from a CDN.

Safari has more complex scope management, known as partitions, affecting how scopes work if you have cross-domain iframes. To read more about WebKit's implementation, read their blog post.

Lifecycle

Service workers have a lifecycle that dictates how they are installed, this is separate from your PWA installation. The service worker lifecycle starts with registering the service worker. The browser then attempts to download and parse the service worker file. If parsing succeeds, its install event is fired. The install event only fires once.
Service worker installation happens silently, without requiring user permission, even if the user doesn't install the PWA. The Service Worker API is even available on platforms that do not support PWA installation, such as Safari and Firefox on desktop devices.
Service worker registration and installation, while related, are different events. Registration happens when a page requests a service worker by calling register() as described previously. Installation happens when a registered service worker exists, can be parsed as JavaScript, and doesn't throw any errors during its first execution.
After the installation, the service worker is not yet in control of its clients, including your PWA. It needs to be activated first. When the service worker is ready to control its clients, the activate event will fire. This doesn't mean, though, that the page that registered the service worker will be managed. By default, the service worker will not take control until the next time you navigate to that page, either due to reloading the page or re-opening the PWA.

You can listen for events in the service worker's global scope using the self object.
serviceworker.js

```
// This code executes in its own worker or thread
self.addEventListener("install", event => {
  console.log("Service worker installed");
});
self.addEventListener("activate", event => {
  console.log("Service worker activated");
});
```

Updating a service worker

Service workers get updated when the browser detects that the service worker currently controlling the client and the new (from your server) version of the same file are byte-different.

Warning: When updating your service worker, do so without renaming it. Do not even add file hashes to the filename. Otherwise, the browser will never get the new version of your service worker!
After a successful installation, the new service worker will wait to activate until the existing (old) service worker no longer controls any clients. This state is called "waiting", and it's how the browser ensures that only one version of your service worker is running at a time. Refreshing a page or reopening the PWA won't make the new service worker take control. The user needs to close or navigate away from all tabs and windows using the current service worker and then navigate back. Only then will the new service worker take control.

Service worker lifespan

Once installed and registered, a service worker can manage all network requests within its scope. It runs on its own thread, with activation and termination controlled by the browser. This lets it work even before or after your PWA is open. While service workers run on their own thread, there is no guarantee that in-memory state will persist between runs of a service worker, so make sure anything you want to reuse on each run is available either in IndexedDB or some other persistent storage.
If not already running, a service worker will start whenever a network request in its scope is asked for, or when a triggering event, like periodic background sync or a push message, is received.
Service workers don't live indefinitely. While exact timings differ between browsers, service workers will be terminated if they've been idle for a few seconds, or if they've been busy for too long. If a service worker has been terminated and an event occurs that would start it up, it will restart.

Capabilities

With a registered and active service worker, you have a thread with a completely different execution lifecycle than the main one on your PWA. However, by default, the service worker file itself has no behavior. It won't cache or serve any resources, as this has to be done by your code. You'll find out how in the following chapters.
Service worker's capabilities are not just for proxy or serving HTTP requests; other features are available on top of it for other purposes, such as background code execution, web push notifications, and process payments.

service-worker.js

```
self.addEventListener("install", function (event) {
```

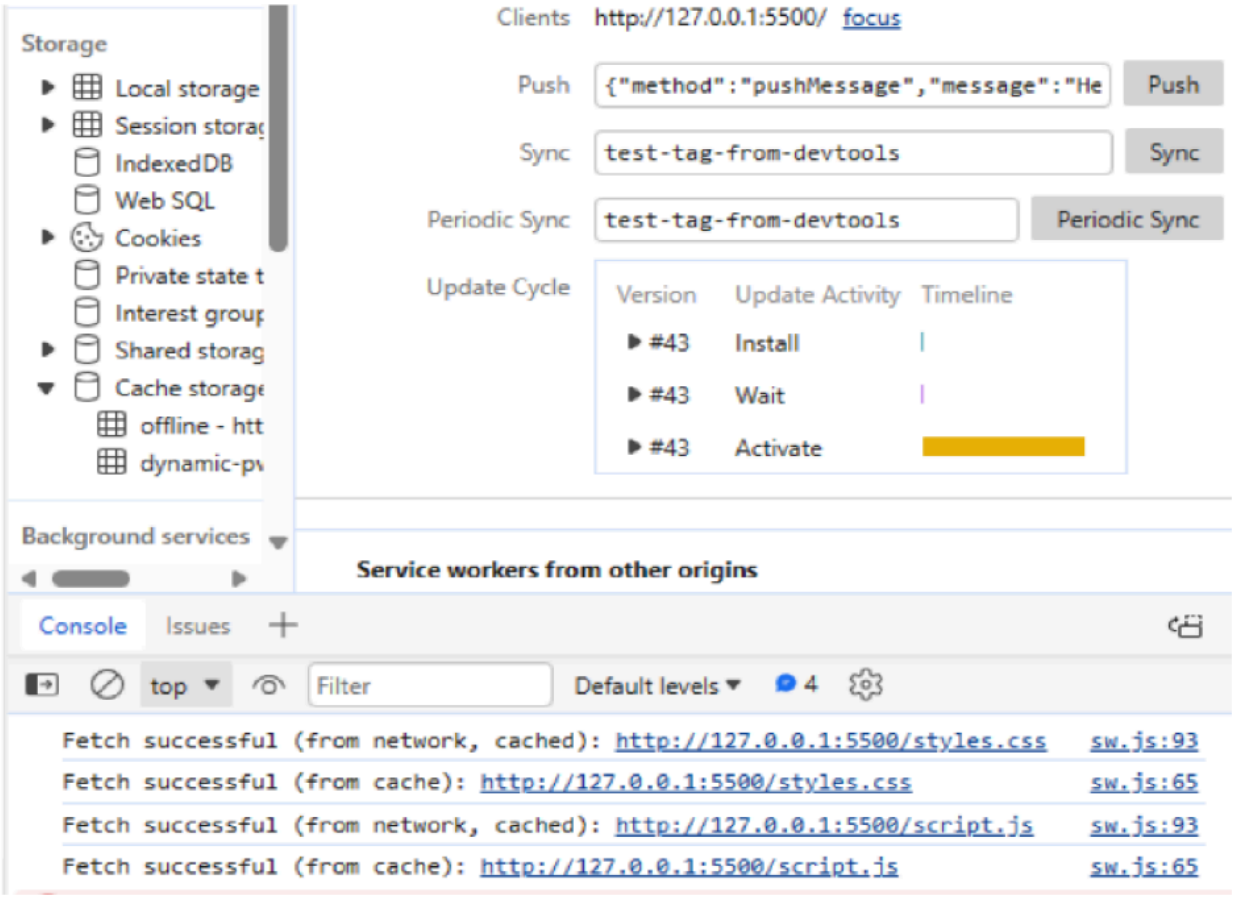
```
    event.waitUntil(preLoad());
});

self.addEventListener('sync', event => {
    if (event.tag === 'Sync from cache') {
        console.log("Sync successful!")
    }
});

self.addEventListener('push', function (event) {
    if (event && event.data) {
        var data = event.data.json();
        if (data.method == "PushMessageData") {
            console.log("Push notification sent");
            event.waitUntil(self.registration.showNotification("RED
STORE", { body: data.message }))
        }
    }
});

var filesToCache = [
    '/index.html',
    '/product_details.html',
    '/products.html',
    '/cart.html',
    '/account.html'
];

var preLoad = function () {
    return caches.open("offline").then(function (cache) { // caching index
and important routes
        return cache.addAll(filesToCache);
    });
};

self.addEventListener("fetch", function (event) {
    event.respondWith(checkResponse(event.request).catch(function () {
        return returnFromCache(event.request);
    }));
    event.waitUntil(addToCache(event.request));
```

```
});

var checkResponse = function (request) {
    return new Promise(function (fulfill, reject) {
        fetch(request).then(function (response) {
            if (response.status !== 404) {
                fulfill(response);
            } else {
                reject();
            }
        }, reject);
    });
};

var addToCache = function (request) {
    return caches.open("offline").then(function (cache) {
        return fetch(request).then(function (response) {
            return cache.put(request, response);
        });
    });
};

var returnFromCache = function (request) {
    return caches.open("offline").then(function (cache) {
        return cache.match(request).then(function (matching) {
            if (!matching || matching.status == 404) {
                return cache.match("index.html");
            } else {
                return matching;
            }
        });
    });
};

// Add event listener for beforeinstallprompt
self.addEventListener('beforeinstallprompt', event => {
    event.preventDefault();
    const installButton = document.getElementById('install-button');
    if (installButton) {
        installButton.style.display = 'block';
```

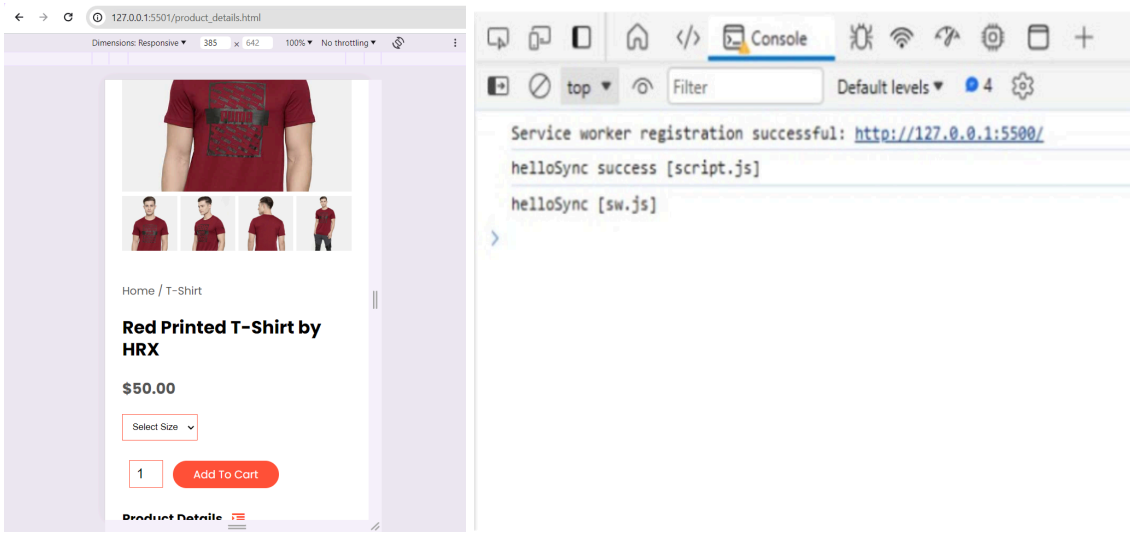
```
        installButton.addEventListener('click', () => {
            event.prompt();
        });
    }
});
```

Output:

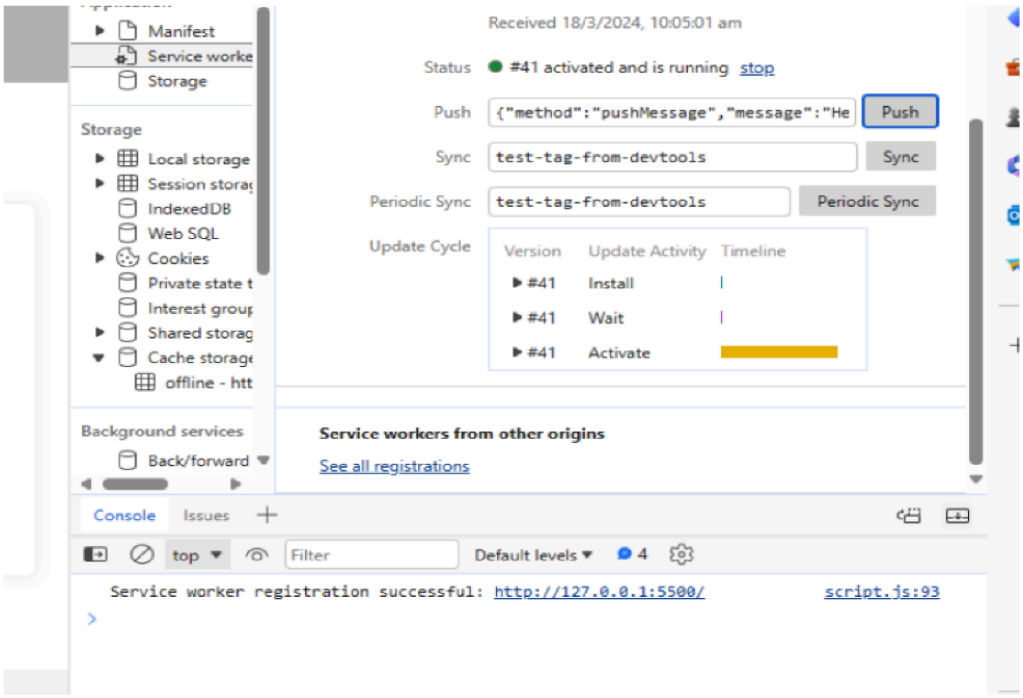
Fetch Event



Sync Event



Push Event



Conclusion : Therefore, integrating service worker events fetch, sync, and push into the E-commerce PWA enhances its functionality, performance, and user engagement, making it a powerful and reliable platform for online shopping experiences.