Service Worker | General Overview

Web Workers

Web Workers makes it possible to run a script operation in a **background thread** separate from the **main execution thread** of a web application. This allows the main (usually the UI) thread to run without being blocked/slowed down while heavy processing is being performed on separate threads.

Types of Workers:

- Dedicated worker
- Shared worker
- **Service worker**

Workers run in a different global context than the current `window`! While `Window` is not directly available to workers, many of the same methods are made available to workers through their own `WorkerGlobalScope`-derived contexts:

- `DedicatedWorkerGlobalScope` for Dedicated workers
- `SharedWorkerGlobalScope` for Shared workers
- `ServiceWorkerGlobalScope` for Service workers

Service Worker

A service worker is an **event-driven worker** registered against an **origin** and a path.

It is simply a JavaScript file that can control the web-page by intercepting and modifying navigation and resource requests, and caching resources such that there is complete control over how to make the app behave in certain situations such as for offline experience.

Service Workers act as proxy servers that sit between web applications, the browser, and the network. They are intended to enable the creation of effective **offline experiences** through **intercepting** network requests.

Points to remember

- A service worker is run in a **worker context**.
- It does not have DOM access

- Because it runs on separate thread instead of main thread, It is non blocking
- APIs such as synchronous XHR and Web Storage can't be used inside a service worker.
- Service workers only run over HTTPS

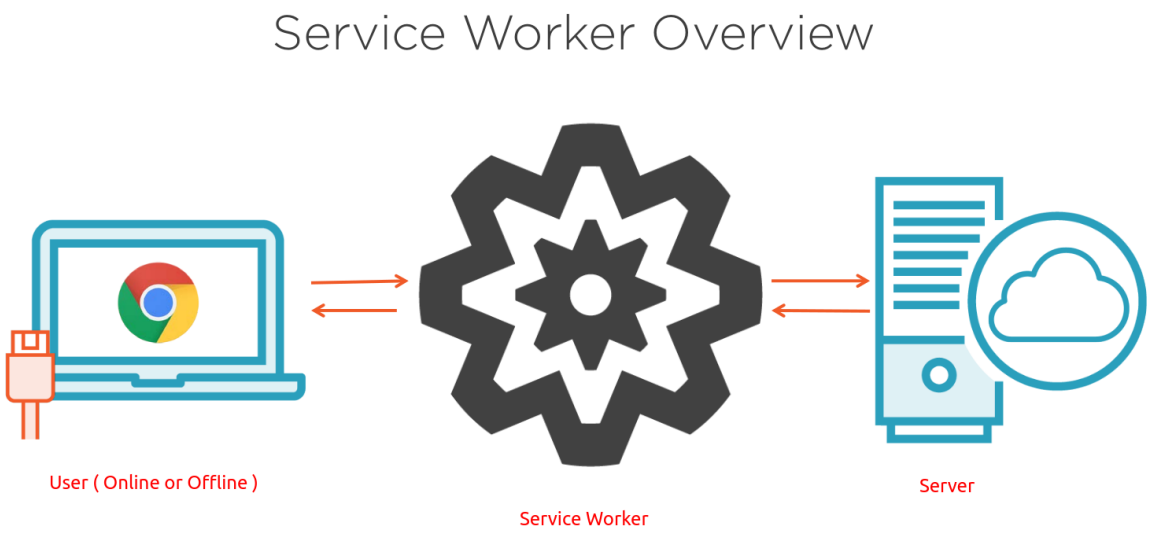

Fig 1: Overview of Service Worker

Basic architecture

With service workers, the following steps are generally observed for basic set up:

1. The service worker URL is fetched and registered via `serviceWorkerContainer.register()`.
2. If successful, the service worker is executed in a `ServiceWorkerGlobalScope`, which is a special kind of worker context, running off the main script execution thread, with no DOM access.
3. The service worker is now ready to process events.
4. **Installation** of the worker is attempted when service worker-controlled pages are accessed. An **Install event** is always the first one sent to a service worker (This can be used to start the process of caching site assets).
5. When the `oninstall` handler completes, the service worker is considered installed.
6. Next is **activation**. When the service worker is installed, it then receives an **activate event**. The primary use of `onactivate` is for cleanup of resources used in previous versions of a Service worker script.
7. The Service worker will now control pages which are loaded after the `register()` is successful. So documents will have to be reloaded to actually be controlled.

The below graphic shows a summary of the available service worker events:

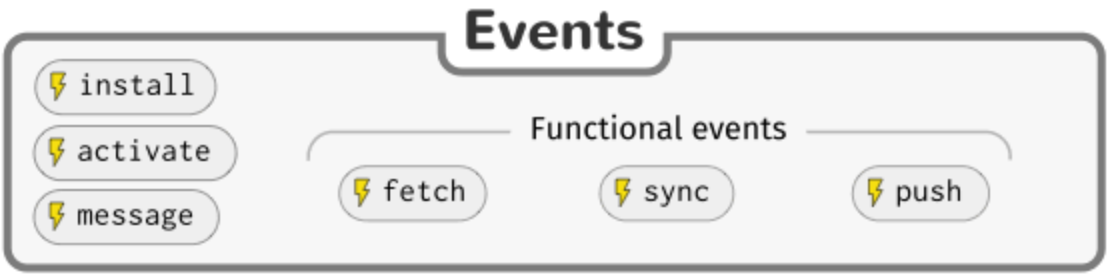

Fig 2: Different events available in service worker

Path to Service Workers

Fig 3: Path to using Service Workers

Registration

A service worker is first registered using the `ServiceWorkerContainer.register()` method. If successful, the service worker will be downloaded to the client which will **attempt installation/activation** for URLs accessed by the user inside the whole origin, or inside a subset specified through **scope**.

Scope refers to the path that the service worker will be able to intercept network calls from. The scope property can be used to explicitly define the scope it will cover.

To understand the idea around scoping refer to following table:

Registration Scopes

Script URL	Scope	Header	Result
/sw.js			Control whole domain.
/sw.js	/brand		Control just /brand.
/sw.js	/flavors		Control just /flavors.
/scripts/sw.js			Control all of /scripts.
/scripts/sw.js	/brand		Error: Invalid scope.
/scripts/sw.js	/brand	/	Control just /brand.
/scripts/sw.js		/	Control whole domain.
/scripts/min/sw.js	/	/scripts	Error: Invalid scope.

Fig 4: Registration Scopes

> **Note**: There is frequent confusion surrounding the meaning and use of *scope*. Since a service worker can't have a scope broader than its own location, only use the `scope` option when you need a scope that is narrower than the default.

Refer to this repo and the deployed site to understand the following code as a whole:

To register the service worker, copy and paste the following code in the **main.js** file or follow the structure as in the repo.

main.js

```
// Registers the service worker if supported by browser

if (navigator.serviceWorker) {
 navigator.serviceWorker
   .register('sw.js', { scope: '/' })
   .then(() => console.log('Service Worker: Registered'))
   .catch(() => console.log('Service Worker Registration: Error'));
}
```

Lifecycle of Service Worker

At this point, service worker will observe the following **lifecycle**:

1. Download
2. Install
3. Activate

1. Download

The service worker is downloaded if registration is successful.

2. Install

a. Initial Installation

After the service worker is registered, the browser will attempt to **install** and then activate the service worker for your page/site. If there is no service worker installed earlier, the installation follows the following steps as shown in the diagram below.

The **install event** is fired when an install is successfully completed. The install event can be listened to populate the browser's offline caching capabilities with the assets needed to run the app offline.

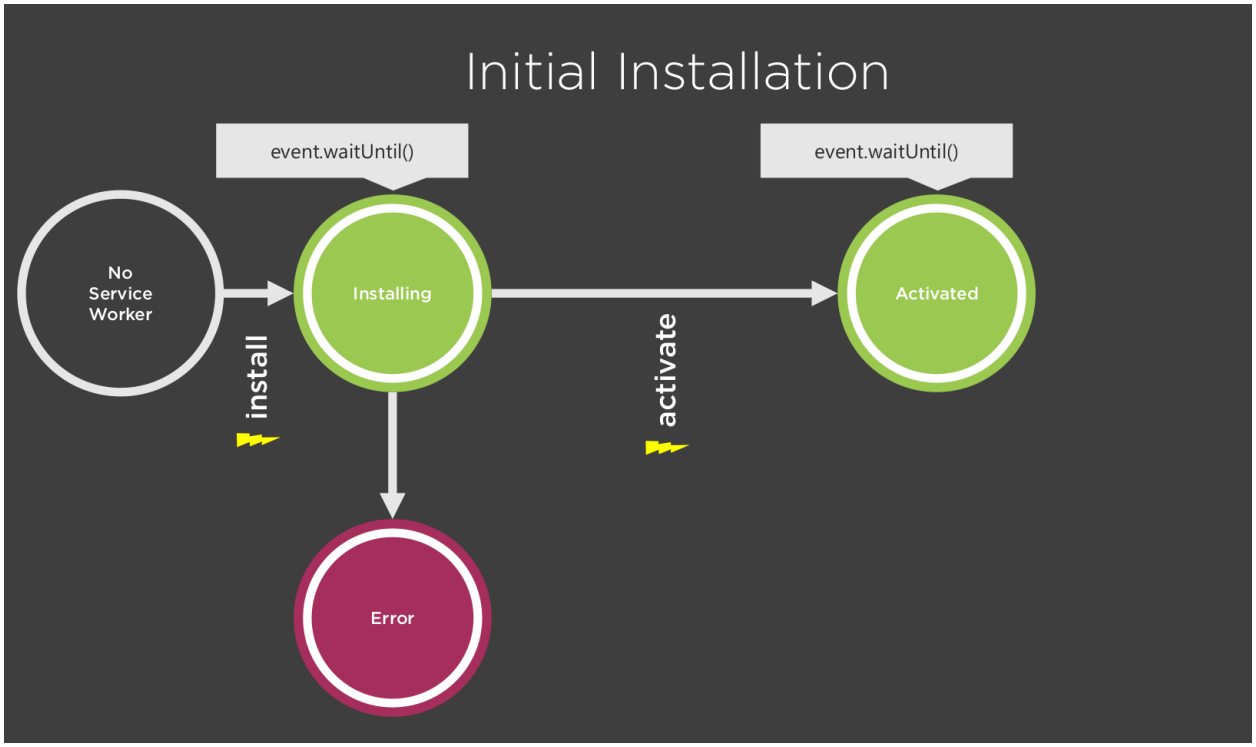

Fig 5: Initial Installation when there is no Service Worker

sw.js

```
 ...

self.addEventListener('install', (event) => {
 console.log('Service Worker: Installed');
 ...
 // Store cache for offline experience
 ...
});

 ...
```

b. Update Installation

If there is an existing service worker available, the new version is installed in the background, but not yet activated — at this point it is called the *worker in **waiting***.

It is only activated when there are no longer any pages loaded that are still using the old service worker.

As soon as there are no more pages to be loaded, the new service worker activates (becoming the *active worker*). Activation can happen sooner using `ServiceWorkerGlobalScope.skipWaiting()` and existing pages can be claimed by the active worker using `Clients.claim()`.

Update Installation occurs when:

- A navigation to a **scoped** page occurs.
- An event is fired on the service worker and it hasn't been downloaded in the last **24 hours**.
- Manually **skipWaiting** the installation process as follow:

sw.js

```
// Manually skipWaiting the waiting state via self.skipWaiting()

 ...

self.addEventListener('install', (e) => {
 console.log('Service Worker: Installed');
 self.skipWaiting();
```

```
});

 ...
```

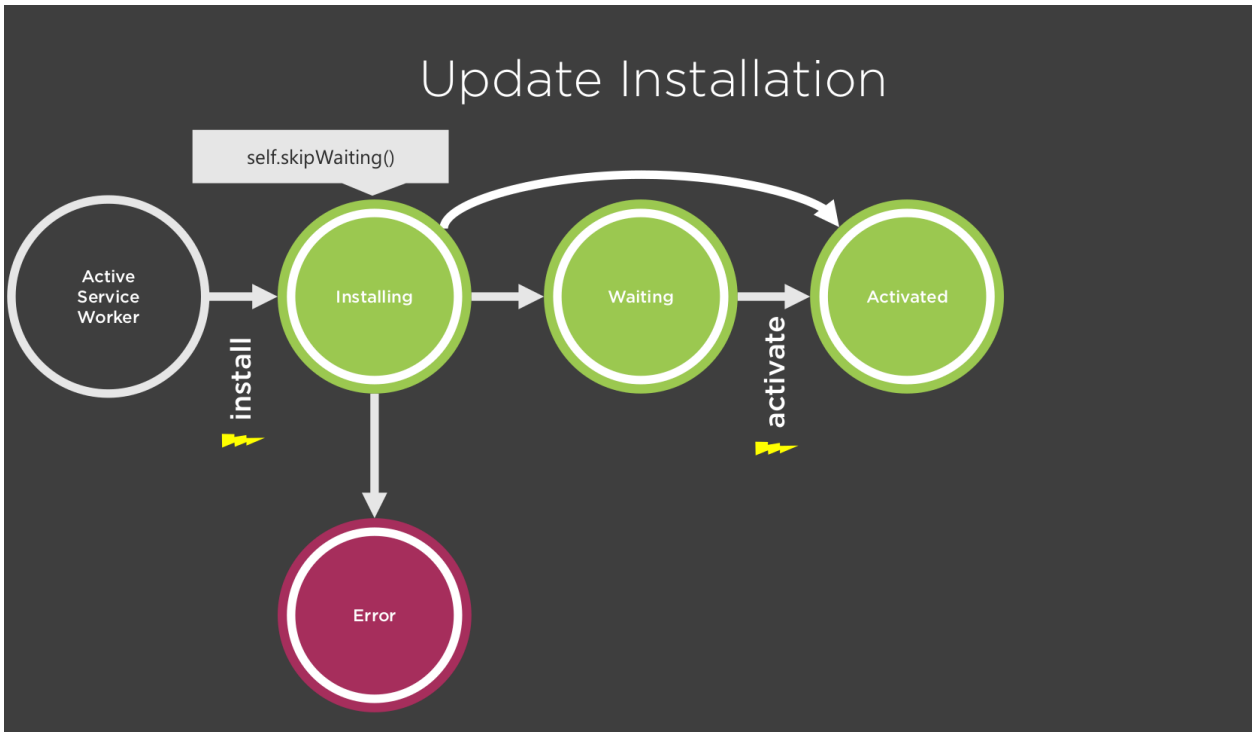

Fig 6: Update Installation when there is previous Active Service Worker

Activation

There is also an `activate` event. The point where this event fires is generally a good time to clean up old caches and other things associated with the previous version of your service worker.

```
const cacheName = 'v2';

...

self.addEventListener('activate', e => {
   console.log('Service Worker: Activated');

   e.waitUntil(
       caches.keys().then(cacheNames => {
           return Promise.all(
               cache Names.map(cache => {
                   if (cache !== cacheName) {
```

```
                    console.log('Service Worker: Clearing Old Cache');
                    return caches.delete(cache);
                }
            })
        );
    })
  );
});
```

Usage

Service workers can respond to requests using the `FetchEvent` event. You can modify the response to these requests in any way you want, using the `FetchEvent.respondWith()` method.

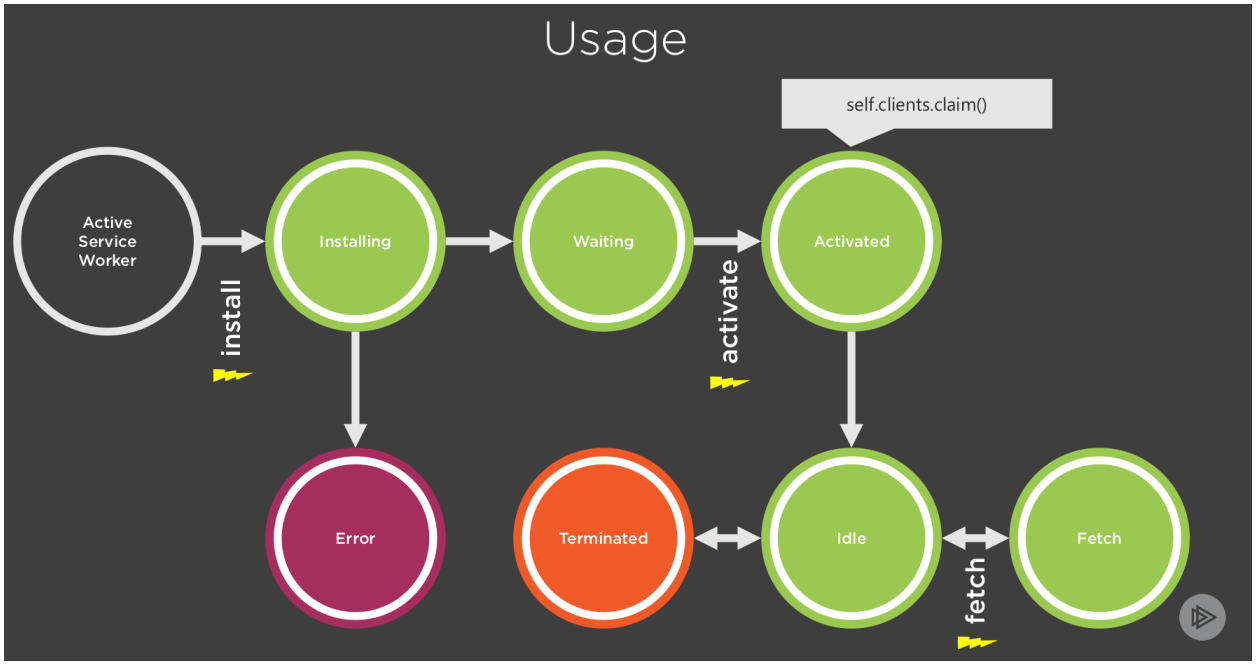

Fig 7: Usage of Service Worker

```
self.addEventListener('fetch', e => {
  console.log('Service Worker: Fetching');
  e.respondWith(
      fetch(e.request)
          .then(res => {
              // Make clone of response
              const resClone = res.clone();
              // Open cache
              caches.open(cacheName).then(cache => {
```

```
                // Add response to cache
                cache.put(e.request, resClone);
            });
            return res;
        })
        .catch(err => caches.match(e.request).then(res => res))
  );
});
```

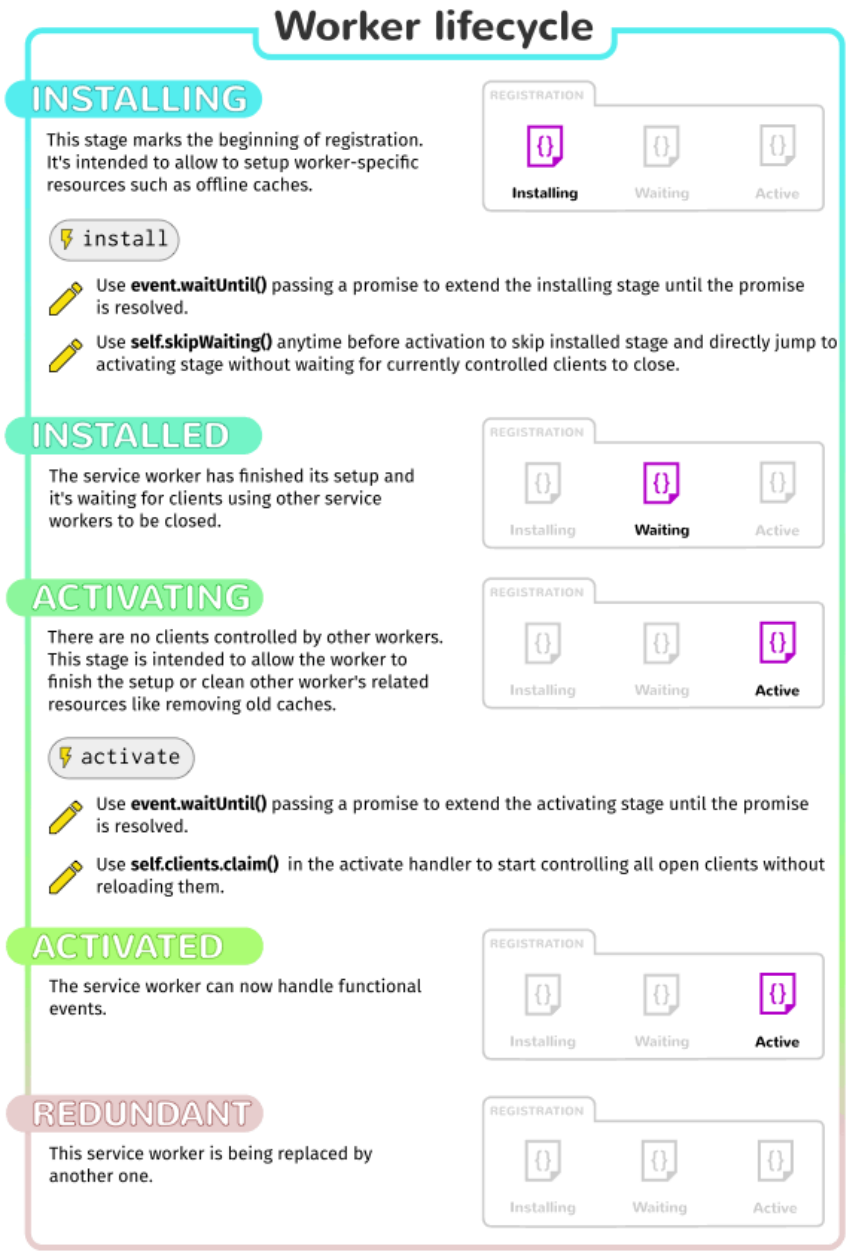

Fig 8: Lifecycle of Service Worker

Note

- Not all interfaces and functions are available to scripts inside a `Worker`. Similarly, synchronous requests are not allowed from within a service worker — only asynchronous requests, like those initiated via the `fetch()` method, can be used.
- Developers should keep in mind that the ServiceWorker state is not persisted across the termination/restart cycle, so each event handler should assume it's being invoked with a bare, default global state.

References

1. https://developers.google.com/web/ilt/pwa/introduction-to-service-worker-slides
2. https://developers.google.com/web/ilt/pwa/introduction-to-service-worker
3. https://developers.google.com/web/fundamentals/primers/service-workers
4. https://developer.mozilla.org/en-US/docs/Web/API/Worker
5. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers
6. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
7. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers