Flavio Copes Follow
https://flaviocopes.com
Feb 5 · 7 min read

Service workers: the little heroes behind Progressive Web Apps

Service workers are at the core of Progressive Web Apps. They allow caching of resources and push notifications, which are two of the main distinguishing features that have set native apps apart up to this point.

A service worker is a **programmable proxy** between your web page and the network which provides the ability to intercept and cache network requests. This effectively lets you **create an offline-first experience for your app**.

Service workers are a special kind of web worker: a JavaScript file associated with a web page which runs on a worker context, separate from the main thread. This gives the benefit of being non-blocking—so computations can be done without sacrificing the UI responsiveness.

Since it's on a separate thread, it has no DOM access. Nor does it have access to the Local Storage APIs and the XHR API. It can only

communicate back to the main thread using the **Channel Messaging API**.

Service Workers cooperate with other recent Web APIs:

- **Promises**
- **Fetch API**
- **Cache API**

And they are **only available on HTTPS** protocol pages (except for local requests, which do not need a secure connection. This makes testing easier.).

Background Processing

Service workers run independently of the application they are associated with, and they can receive messages when they are not active.

For example they can work:

- when your mobile application is **in the background**, not active
- when your mobile application is **closed** and even not running in the background
- when **the browser is closed**, if the app is running in the browser

The main scenarios where service workers are very useful are:

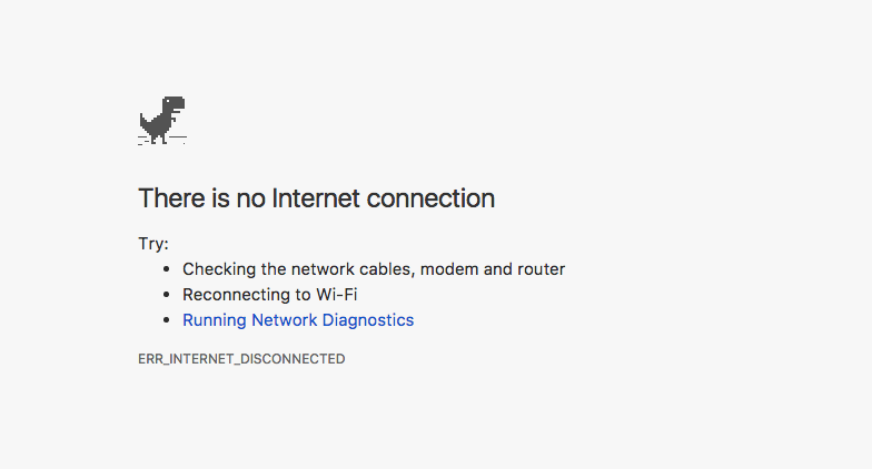
- They can be used as a **caching layer** to handle network requests, and cache content to be used when offline
- They can allow **push notifications**

A service worker only runs when needed, and it's stopped when not used.

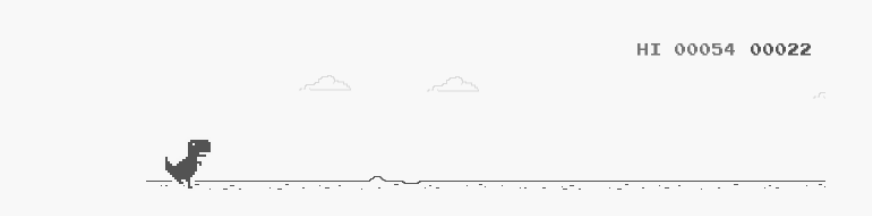
Offline Support

Traditionally, the offline experience for web apps has been very poor. Without a network, often mobile web apps simply won't work. Native mobile apps, on the other hand, have the ability to offer either a working version or some kind of nice message.

This is not a nice message, but this is what a web pages look like in Chrome without a network connection:



Possibly the only nice thing about this is that you get to play a free game by clicking the dinosaur—but it gets boring pretty quickly.



In the recent past, the HTML5 AppCache already promised to allow web apps to cache resources and work offline. But its lack of flexibility and confusing behavior made it clear that it wasn't good enough for the job (and it's been discontinued).

Service workers are the new standard for offline caching.

Which kind of caching is possible?

Precache assets during installation

Assets that are reused throughout the application, like images, CSS, JavaScript files, can be installed the first time the app is opened.

This gives the base of what is called the **App Shell architecture**.

Caching network requests

Using the **Fetch API,** we can edit the response coming from the server, determining if the server is not reachable and providing a response from the cache instead.

A Service Worker Lifecycle

A service worker goes through three steps to become fully functional:

- Registration
- Installation
- Activation

Registration

Registration tells the browser where the server worker is, and it starts the installation in the background.

Example code to register a service worker placed in `worker.js` :

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/worker.js')
    .then((registration) => {
      console.log('Service Worker registration completed
with scope: ', registration.scope)
    }, (err) => {
      console.log('Service Worker registration failed', err)
    })
  })
} else {
  console.log('Service Workers not supported')
}
```

Even if this code is called multiple times, the browser will only perform the registration if the service worker is new and not registered previously, or if it has been updated.

Scope

The `register()` call also accepts a scope parameter, which is a path that determines which part of your application can be controlled by the service worker.

It defaults to all files and subfolders contained in the folder that contains the service worker file, so if you put it in the root folder, it will have control over the entire app. In a subfolder, it will only control pages accessible under that route.

The example below registers the worker, by specifying the `/notifications/` folder scope.

```
navigator.serviceWorker.register('/worker.js', {
  scope: '/notifications/'
})
```

The `/` is important: in this case, the page `/notifications` won't trigger the Service Worker, while if the scope was

```
{ scope: '/notifications' }
```

it would have worked.

NOTE: The service worker cannot "up" itself from a folder: if its file is put under `/notifications`, it cannot control the `/` path or any other path that is not under `/notifications`.

Installation

If the browser determines that a service worker is outdated or has never been registered before, it will proceed to install it.

```
self.addEventListener('install', (event) => {
  //...
});
```

This is a great time to prepare the service worker to be used by **initializing a cache.** Then **cache the App Shell** and static assets using the **Cache API**.

Activation

Once the service worker has been successfully registered and installed, the third step is activation.

At this point, the service worker will be able to work with new page loads.

It cannot interact with pages already loaded, so the service worker is only useful the second time the user interacts with the app or reloads one of the pages already open.

```
self.addEventListener('activate', (event) => {
  //...
});
```

A good use case for this event is to cleanup old caches and things associated with the old version that are unused in the new version of the service worker.

Updating a Service Worker

To update a service worker, you just need to change one byte in it. When the register code is run, it will be updated.

Once a service worker is updated, it won't become available until all pages that were loaded with the old service worker attached are closed.

This ensures that nothing will break on the apps/pages that are already working.

Refreshing the page is not enough, as the old worker is still running and it hasn't been removed.

Fetch Events

A **fetch event** is fired when a resource is requested on the network.

This offers us the ability to **look in the cache** before making network requests.

For example, the snippet below uses the **Cache API** to check if the requested URL was already stored in the cached responses. If that's the case, it returns the cached response. Otherwise, it executes the fetch request and returns it.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => {
        if (response) {
          //entry found in cache
          return response
        }
        return fetch(event.request)
      }
    )
  )
})
```

Background Sync

Background sync allows outgoing connections to be deferred until the user has a working network connection.

This is key to ensure that a user can use the app offline, take actions on it, and queue server-side updates for when there is a connection open (instead of showing an endless spinning wheel trying to get a signal).

```
navigator.serviceWorker.ready.then((swRegistration) => {
  return swRegistration.sync.register('event1')
});
```

This code listens for the event in the service worker:

```
self.addEventListener('sync', (event) => {
  if (event.tag == 'event1') {
    event.waitUntil(doSomething())
  }
})
```

`doSomething()` returns a promise. If it fails, another sync event will be scheduled to retry automatically until it succeeds.

This also allows an app to update data from the server as soon as there is a working connection available.

Push Events

Service workers enable web apps to provide native Push Notifications to users.

Push and Notifications are actually two different concepts and technologies that are combined to provide what we know as **Push Notifications**. Push provides the mechanism that allows a server to send information to a service worker, and Notifications are the way service workers can show information to the user.

Since service workers run even when the app is not running, they can listen for push events coming. They then either provide user notifications, or update the state of the app.

Push events are initiated by a backend, through a browser push service, like the one provided by Firebase.

Here is an example of how the web worker can listen for incoming push events:

```
self.addEventListener('push', (event) => {
  console.log('Received a push event', event)

  const options = {
    title: 'I got a message for you!',
    body: 'Here is the body of the message',
    icon: '/img/icon-192x192.png',
    tag: 'tag-for-this-notification',
  }

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

A note about console logs:

If you have any console log statement (`console.log` and friends) in the service worker, make sure you turn on the `Preserve log` feature provided by the Chrome Devtools (or equivalent).

Otherwise, since the service worker acts before the page is loaded, and the console is cleared before loading the page, you won't see any log in the console.

Thanks for reading through this tutorial. There's a lot to learn about this topic! I publish a lot of related content on my blog about frontend development, don't miss it! 😀

Originally published at flaviocopes.com.