# Service Workers

# And

# Progressive Web Apps

TABLE OF CONTENTS

Introduction 3

What is a Service Worker? 3

Defining Terms 4

Control 4

Scope 4

Client 4

Service Worker LifeCycle 5

Registration 5

Installation 6

Activation 6

Handling Service Updates 7

The Fetch Event 8

Caching Strategies 9

Progressive Web Apps 13

Service Workers in Angular 14

Adding a service worker to your project 14

Service worker in Action: a tour 16

Simulating a network issue 17

Making the application installable 17

Making changes to your application 18

Conclusion 18

Deepak Kumar Singh

# Introduction

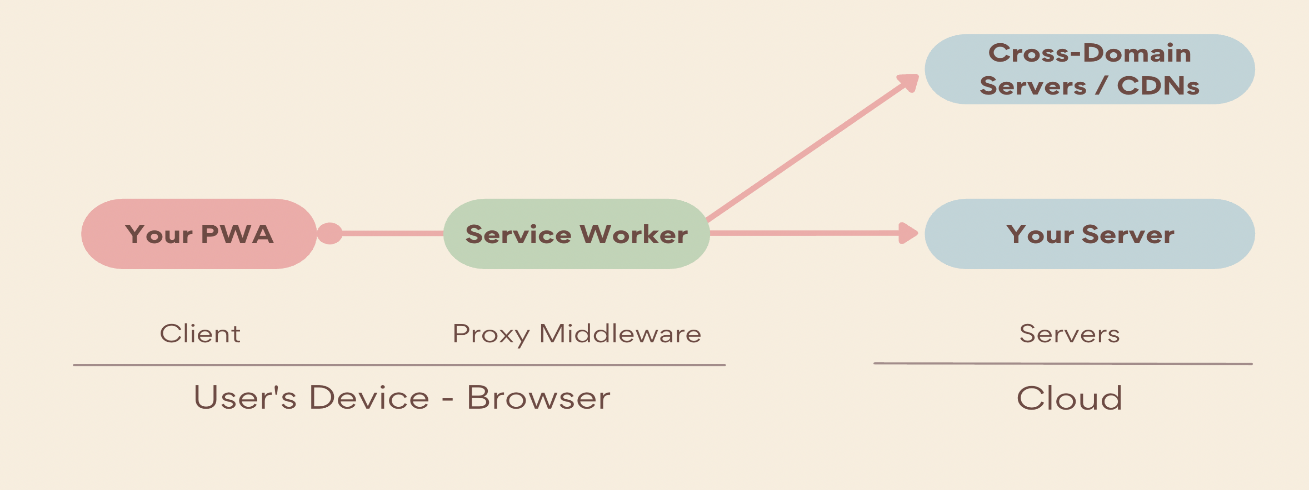
Service workers augment the traditional web deployment model and empower applications to deliver a user experience with the reliability and performance on par with code that is written to run on your operating system and hardware. Adding a service worker to an Angular application is one of the steps for turning an application into a Progressive Web App (also known as a PWA).

Service workers offer incredible utility but can be tricky to work with at first. However, because service workers solve hard problems, any abstraction of that technology will also be tricky without understanding it. Thus, these first few bits of documentation will cover that underlying technology

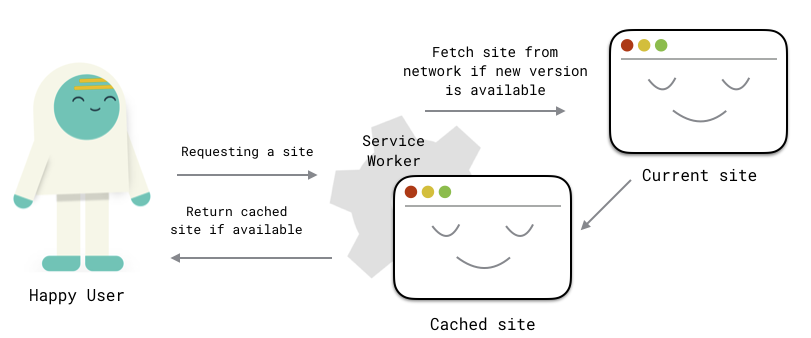
# What is a Service Worker?

A service worker is a script that runs in the web browser and manages caching for an application.

Service workers function as a network proxy. They intercept all outgoing HTTP requests made by the application and can choose how to respond to them. For example, they can query a local cache and deliver a cached response if one is available.



The service worker is preserved after the user closes the tab. The next time that browser loads the application, the service worker loads first, and can intercept every request for resources to load the application. If the service worker is designed to do so, it can completely satisfy the loading of the application, without the need for the network.



# Defining terms

Before getting into the service worker lifecycle, it's worth defining some terms around how that lifecycle operates.

## Control and scope

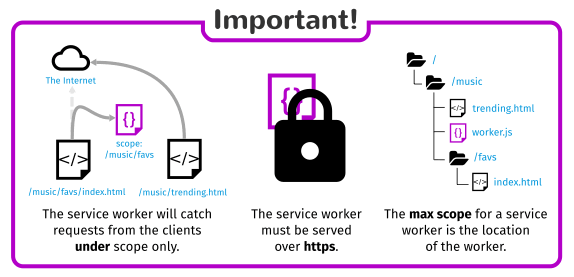
The idea of control is crucial to understanding how service workers operate. A page described as being controlled by a service worker is a page that allows a service worker to intercept network requests on its behalf. The service worker is present and able to do work for the page within a given scope.

## Scope

A service worker's scope is determined by its location on a web server. If a service worker runs on a page located at /subdir/index.html, and is located at /subdir/sw.js, the service worker's scope is /subdir/.

## Client

When it's said that a service worker is controlling a page, it's really controlling a client. A client is any open page whose URL falls within the scope of that service worker. Specifically, these are instances of a [WindowClient](https://developer.mozilla.org/docs/Web/API/WindowClient).

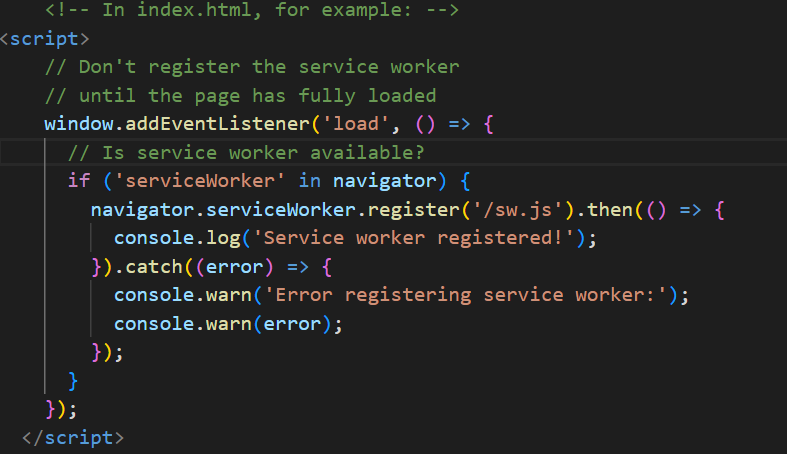


# Service Worker real world example – Tony's Tech & Language BlogService Workers LifeCycle

## These steps are explained below :

## Registration

Registration is the initial step of the service worker lifecycle:



This code runs on the main thread and does the following:

1. Because the user's first visit to a website occurs without a registered service worker, wait until the page is fully loaded before registering one. This avoids bandwidth contention if the service worker precaches anything.
2. When the page is fully loaded, and if service worker is supported, register /sw.js.

Once registration finishes, installation begins.

## Installation

A service worker fires its install event after registration. install is only called once per service worker and won't fire again until it's updated. A callback for the install event can be registered in the worker's scope with addEventListener



event.waitUntil accepts a promise, and waits until that promise has been resolved. In this example, that promise does two asynchronous things:

1. Creates a new Cache instance named 'MyFancyCache\_v1'.
2. After the cache is created, an array of asset URLs are precached using its asynchronous [addAll method](https://developer.mozilla.org/docs/Web/API/Cache/addAll).

Installation fails if the promise(s) passed to event.waitUntil are [rejected](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject). If this happens, the service worker is discarded.

If the promises resolve, installation succeeds, and the service worker's state will change to 'installed' and will then activate.

## Activation

If registration and installation succeed, the service worker activates, and its state becomes 'activating' Work can be done during activation in the service worker's activate event. A typical task in this event is to prune old caches, but for a brand-new service worker, this isn't relevant for the moment, and will be expanded on when we talk about service worker updates.

For new service workers, activate fires immediately after install is successful. Once activation finishes, the service worker's state becomes 'activated'. Notice that, by default, the new service worker won't begin controlling the page until the next navigation or page refresh.

# Handling Service updates

Once the first service worker is deployed, it'll likely need to be updated later. For example, an update may be required if changes occur in request handling or precaching logic.

Browsers will check for updates to a service worker when:

* The user navigates to a page within the service worker's scope.
* navigator.serviceWorker.register() is called with a URL different from the currently installed service worker—but don't change a service worker's URL!
* navigator.serviceWorker.register() is called with the same URL as the installed service worker, but with a different scope.

## Activation

When an updated service worker is installed and the waiting phase ends, it activates, and the old service worker is discarded. A common task to perform in an updated service worker's activate event is to prune old caches. Remove old caches by getting the keys for all open Cache instances with [caches.keys](https://developer.mozilla.org/docs/Web/API/CacheStorage/keys) and deleting caches that aren't in a defined allow list with [caches.delete](https://developer.mozilla.org/docs/Web/API/CacheStorage/delete):

A screenshot of a computer

Description automatically generated

The activate event will finish after the old cache is removed. At this point, the new service worker will take control of the page, finally replacing the old one!

## The Fetch event

So far in this documentation, you've heard a bit about "intercepting network requests", and the fetch event inside of a service worker is where this happens:

Text

Description automatically generated

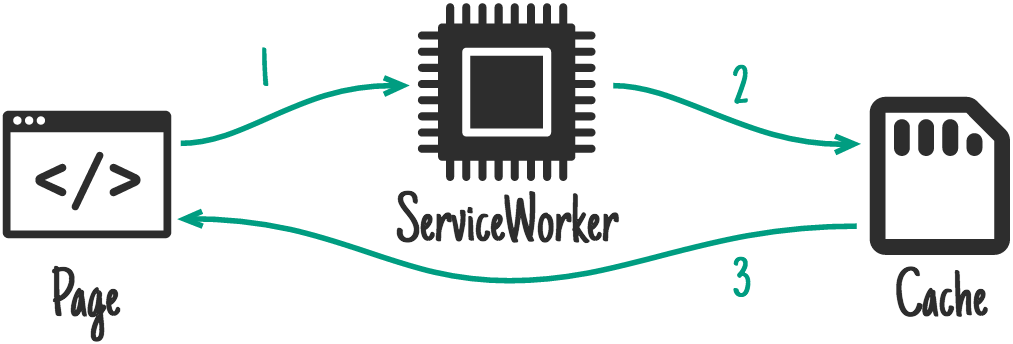
A fetch's event object contains a [request property](https://developer.mozilla.org/docs/Web/API/FetchEvent/request) with some useful bits of information to help you identify the type of each request:

* [url](https://developer.mozilla.org/docs/Web/API/Request/url), which is the URL for the network request currently being handled by the fetch event.
* [method](https://developer.mozilla.org/docs/Web/API/Request/method), which is the request method (e.g., GET or POST).
* [mode](https://developer.mozilla.org/docs/Web/API/Request/mode), which describes the request's mode. A value of 'navigate' is often used to distinguish requests for HTML documents from other requests.
* [destination](https://developer.mozilla.org/docs/Web/API/Request/destination), which describes the type of content being requested in a way that avoids using the requested asset's file extension.

## Caching strategies

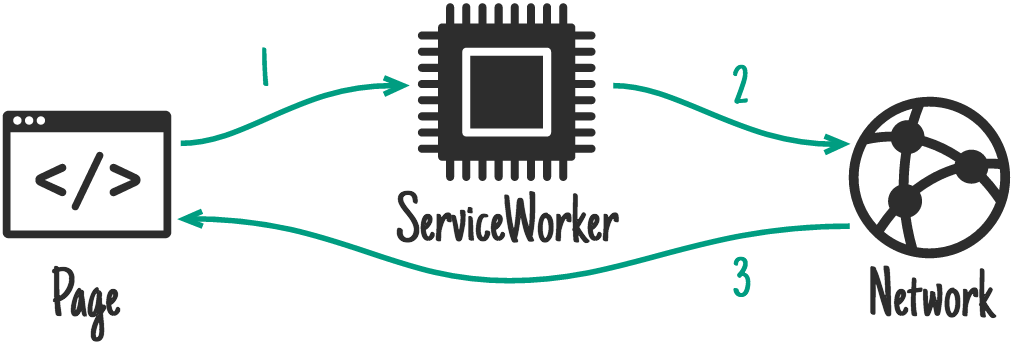
Now that you've got a little familiarity with Cache instances and the fetch event handler, you're ready to dive into some service worker caching strategies.

## Cache only



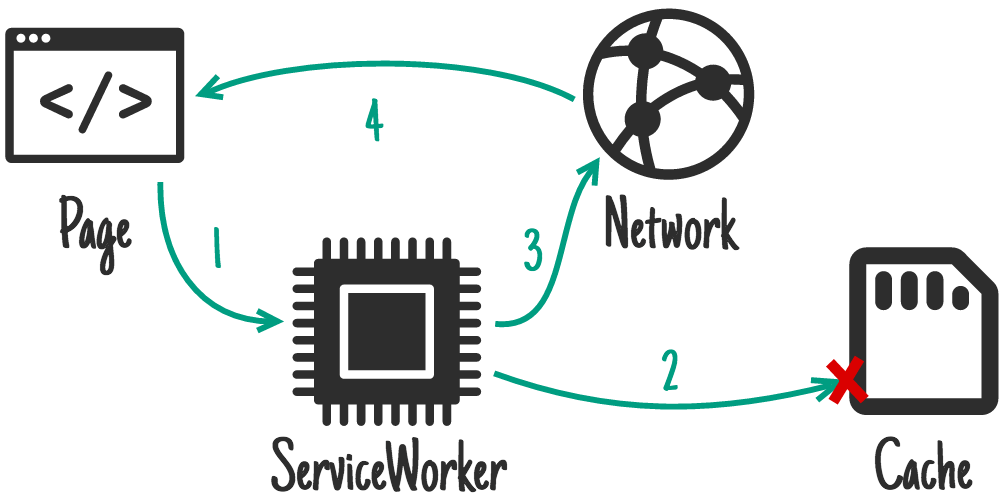
when the service worker is in control of the page, matching requests will only ever go to the cache. This means that any cached assets will need to be precached in order to be available for the pattern to work, and that those assets will never be updated in the cache until the service worker is updated.

## Network only



The opposite of "Cache Only" is "Network Only", where a request is passed through a service worker to the network without any interaction with the service worker cache. This is a good strategy for ensuring content freshness (think markup), but the tradeoff is that it will never work when the user is offline.

## Cache first, falling back to network



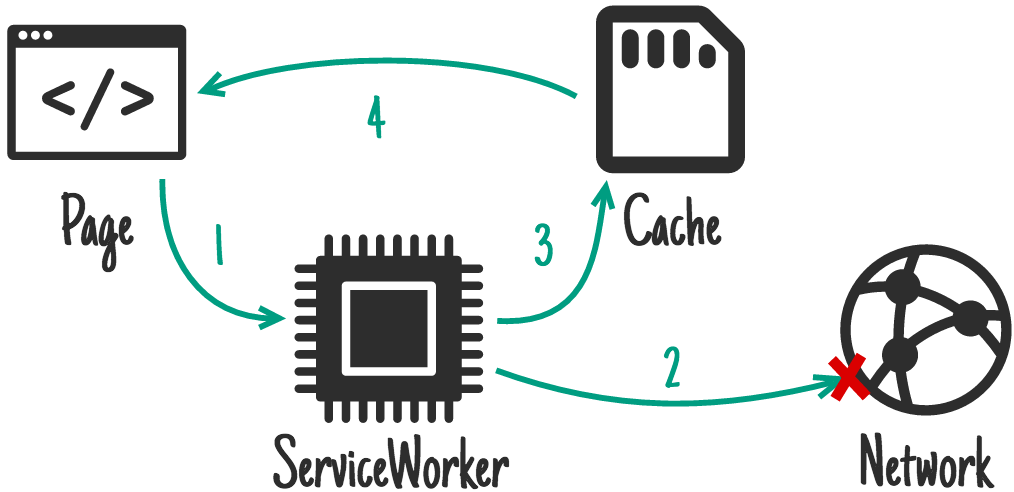
This strategy is where things get a little more involved. For matching requests, the process goes like this:

1. The request hits the cache. If the asset is in the cache, serve it from there.
2. If the request is *not* in the cache, go to the network.
3. Once the network request finishes, add it to the cache, then return the response from the network.

Text

Description automatically generated

## Network first, falling back to cache



If you were to flip "Cache first, network second" on its head, you end up with the "Network first, cache second" strategy, which is what it sounds like:

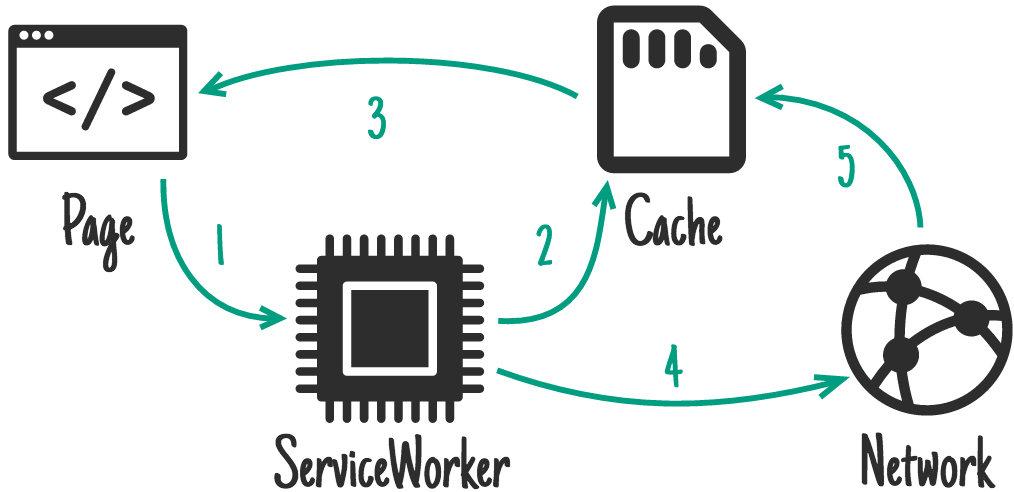
1. You go to the network first for a request and place the response in the cache.
2. If you're offline at a later point, you fall back to the latest version of that response in the cache.

This strategy is great for HTML or API requests when, while online, you want the most recent version of a resource, yet want to give offline access to the most recent available version.

Text

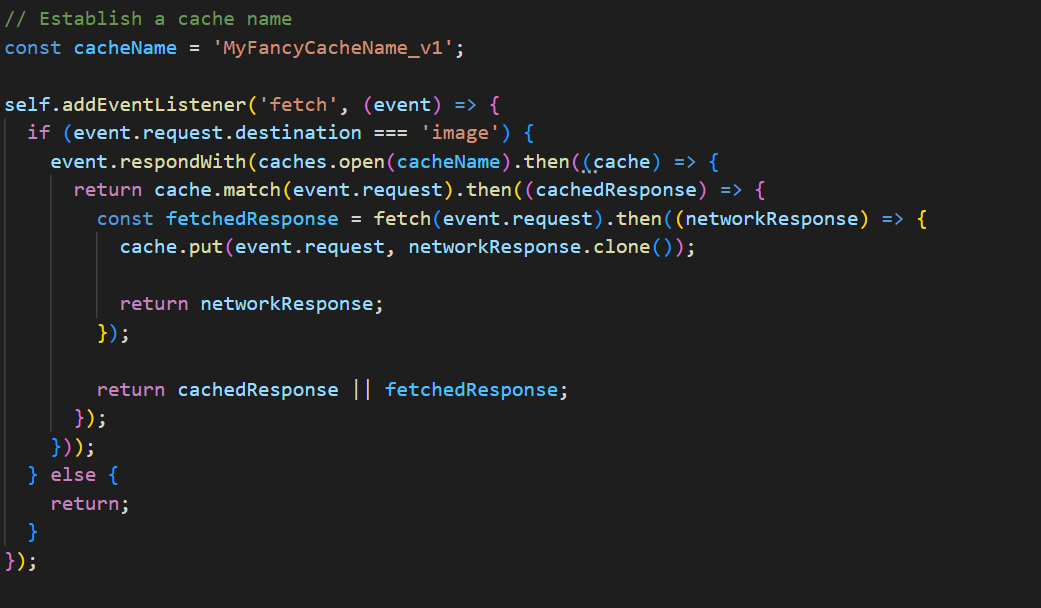
Description automatically generated

## Stale-while-revalidate



Of the strategies we've covered so far, "Stale-while-revalidate" is the most complex. It's similar to the last two strategies in some ways, but the procedure prioritizes speed of access for a resource, while also keeping it up to date in the background. This strategy goes something like:

1. On the first request for an asset, fetch it from the network, place it in the cache, and return the network response.
2. On subsequent requests, serve the asset from the cache first, then "in the background," re-request it from the network and update the asset's cache entry.
3. For requests after that, you'll receive the last version fetched from the network that was placed in the cache in the prior step.



## Some key things to understand are:

* Service workers are only available over HTTPS or localhost.
* If a service worker's contents contain syntax errors, registration fails and the service worker is discarded.
* Reminder: service workers operate within a scope. Here, the scope is the entire origin, as it was loaded from the root directory.
* If you are not using HTTPS, the service worker will only be registered when accessing the application on localhost.

# Progressive web Apps

The term Progressive Web Apps was first coined as a way of describing applications that take advantage of new features supported by modern browsers, including service workers and web app manifests, and let users upgrade web apps to progressive web applications regardless of their native operating system.

Progressive Web Applications use Modern Web APIs to create cross-platform applications. They are web applications that work like native applications

## Setting up a PWA

### A service worker

This aids in offline experience. As discussed above

### Web App Manifest

This controls what the user sees when launching the application after installation. This includes splash screens, themes and so on. It is saved as manifest.json.

Text

Description automatically generated

# Service workers in Angular

Starting with version 5.0.0, Angular ships with a service worker implementation. Angular developers can take advantage of this service worker and benefit from the increased reliability and performance it provides, without needing to code against low-level APIs.

Angular's service worker is designed to optimize the end user experience of using an application over a slow or unreliable network connection, while also minimizing the risks of serving outdated content.

To achieve this, the Angular service worker follows these guidelines:

* Caching an application is like installing a native application. The application is cached as one unit, and all files update together.
* A running application continues to run with the same version of all files. It does not suddenly start receiving cached files from a newer version, which are likely incompatible.
* When users refresh the application, they see the latest fully cached version. New tabs load the latest cached code.
* Updates happen in the background, relatively quickly after changes are published. The previous version of the application is served until an update is installed and ready.
* The service worker conserves bandwidth when possible. Resources are only downloaded if they've changed.

## Adding a service worker to your project

## use the CLI command ng add @angular/pwa. It takes care of configuring your application to use service workers by adding the @angular/service-worker package along with setting up the necessary support files.

ng add @angular/pwa --project <project-name>

The preceding command completes the following actions:

1. Adds the @angular/service-worker package to your project.

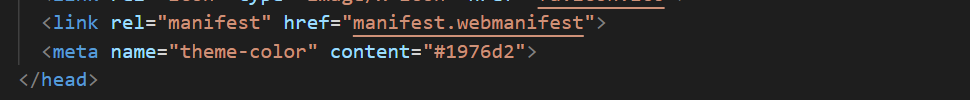


1. Enables service worker build support in the CLI.
2. Imports and registers the service worker in the application module.

Text

Description automatically generated

1. Updates the index.html file:
   * Includes a link to add the manifest.webmanifest file
   * Adds a meta tag for theme-color



1. Installs icon files to support the installed Progressive Web App (PWA).
2. Creates the service worker configuration file called [ngsw-config.json](https://angular.io/guide/service-worker-config), which specifies the caching behaviors and other settings.

Text

Description automatically generated

Now, build the project:

## Service worker in action: a tour

With the server running, point your browser at http://localhost:8080. Your application should load normally.

Graphical user interface, application

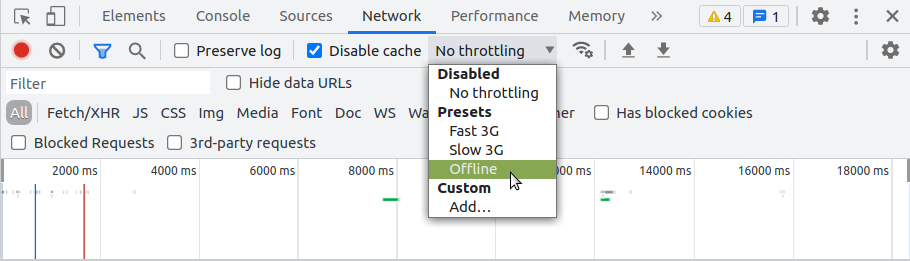
Description automatically generated

## Simulating a network issue

To simulate a network issue, disable network interaction for your application.

In Chrome:

1. Select **Tools** > **Developer Tools** (from the Chrome menu located in the top right corner).
2. Go to the **Network tab**.
3. Select **Offline** in the **Throttling** dropdown menu.



Now the application has no access to network interaction.

For applications that do not use the Angular service worker, refreshing now would display Chrome's Internet disconnected page that says, "There is no Internet connection".

With the addition of an Angular service worker, the application behaviour changes. On a refresh, the page loads normally.

## Making the Application Installable

Most browsers indicate to the user that your Progressive Web App (**PWA**) is **installable** when it meets certain criteria.

A picture containing funnel chart

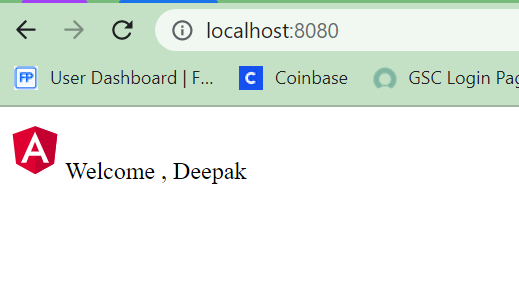
Description automatically generated

After clicking Install, the app is installed to the desktop behaving like it’s a native app.

## Making changes to your application

Now that you've seen how service workers cache your application, the next step is understanding how updates work. Make a change to the application, and watch the service worker install the update:

1. If you're testing in an incognito window, open a second blank tab. This keeps the incognito and the cache state alive during your test.
2. Close the application tab, but not the window. This should also close the Developer Tools.
3. Shut down http-server.
4. Open src/app/app.component.html for editing.
5. Change the text Welcome to “Welcome, Deepak”.
6. Build and run the server again:





# Conclusion

As seen above, service workers help in offline experience. In most native applications (including those which require an internet connection for some operations), users could still perform some other activities without network.

The same goes for Progressive Web Applications. When you have been able to enable one on your previous Web application, you’d need a service worker to improve the offline experience.

## Progressive web apps are a significant step forward into application development segment. Mobile application and websites both contribute significantly towards enhancing customer experience. PWA apps are a stride ahead in this approach.