

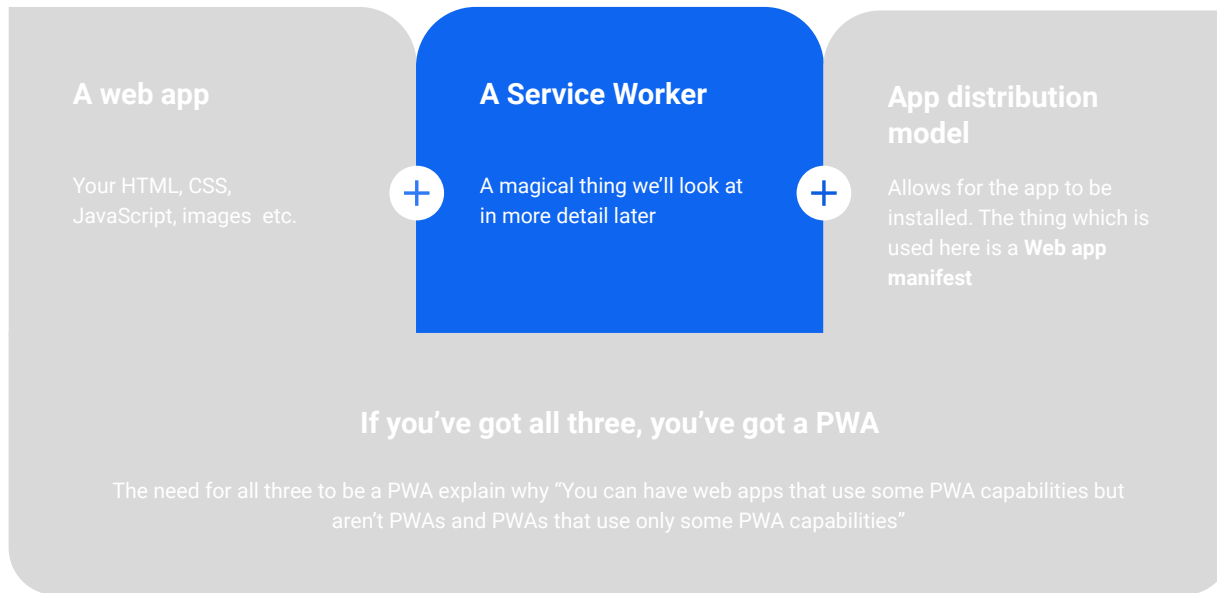
Progressive Web Apps



Part 2. The Service Worker

Part 2: The Service Worker

“At the heart of every progressive web app is the service worker.”
Building Progressive Web Apps, Tal Ater (O’Reilly)



The Service worker

The service worker is...

“JavaScript running in **its own thread** that will act as a locally installed web **server** or web **proxy** for your PWA...”

Progressive Web Apps: The
Big Picture by Maximiliano
Firtman

“At its simplest, a service worker is a script that runs in the web browser and **manages caching** for an application.

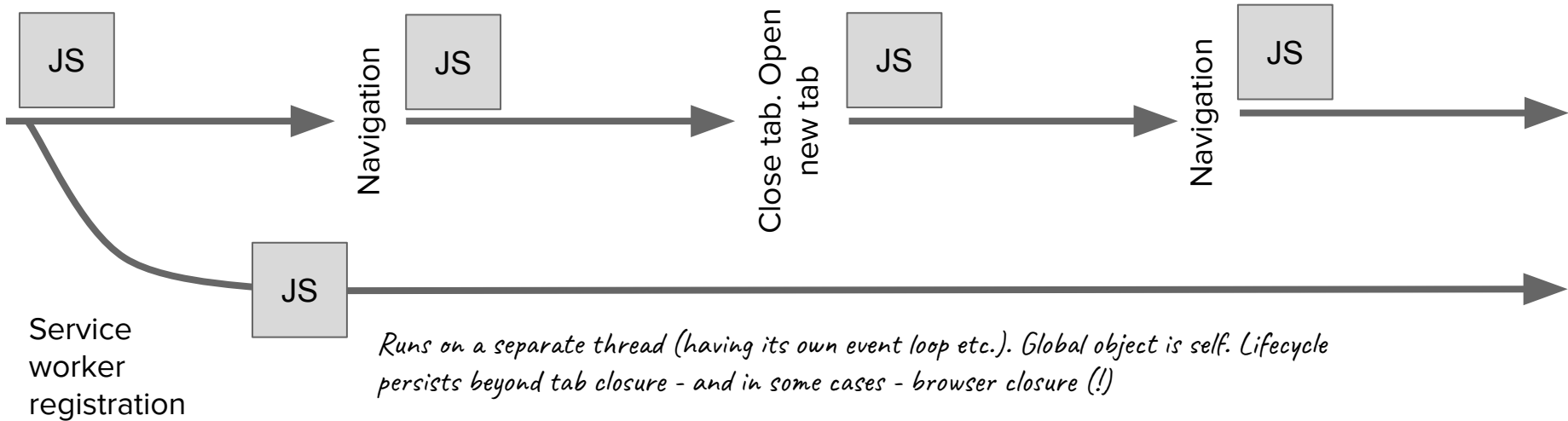
Service workers function as a network **proxy**. They **intercept all outgoing HTTP requests** made by the application and can choose how to respond to them. For example, they can query a local cache and deliver a cached response if one is available. Proxying isn't limited to requests made through programmatic APIs, such as `fetch`; it also includes resources referenced in HTML and even the initial request to `index.html`.

Service worker-based caching is thus **completely programmable** and doesn't rely on server-specified caching headers. Unlike the other scripts that make up an application, such as the Angular app bundle, 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.”

[Angular documentation](#)

Service Workers are a bit different to the JavaScript we're used to

Runs on a single thread (event loop etc.). Attached to a single HTML page and its lifecycle. Global object is window



What Service Workers can do

The big capabilities they can provide
(if you chose for them to do so)

1. **Network independence** - can completely satisfy loading the application
 2. **Instant loading** of the application from their cache
 3. **Background data synchronisation** with the server
 4. **Background fetching** of files
 5. **Respond to the context** in which the application is running
 6. **Passively receive messages** via the Push API
-

Not all features are available on all platforms, so use progressive enhancement.

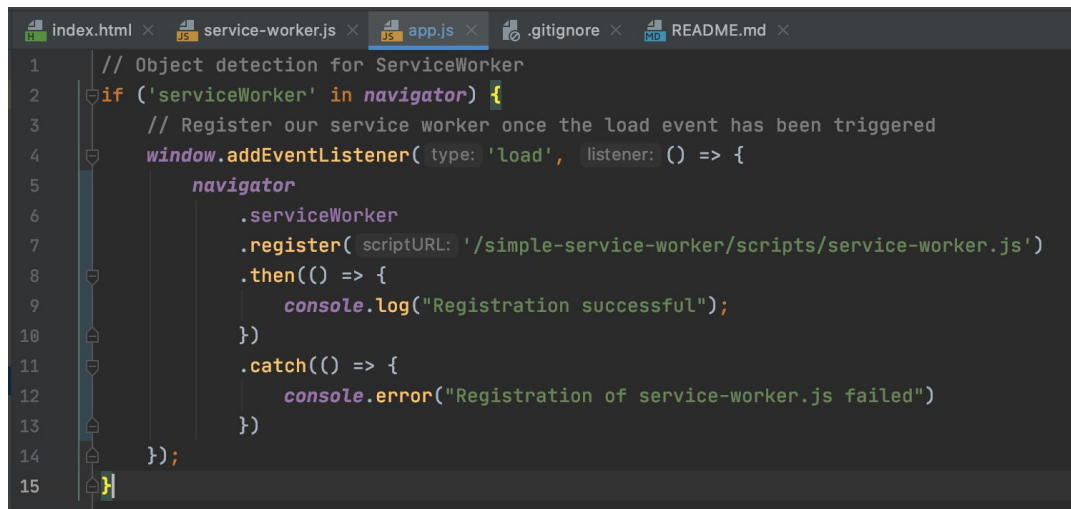
Example 1: Registering a Service Worker

To run this code locally visit <https://github.com/gtvj/exploring-pwas> and follow the instructions in the README.md file.

This first example can be reached by clicking the 'Simple Service Worker' link (and the code is in the /simple-service-worker/ directory)

A few things to note here:

1. Object detection is used (for progressive enhancement)
2. The 'load' event is used to improve performance
3. The register() method returns a promise
4. We can see that our Service Worker has been registered by looking at
 - a. the message printed to the console
 - b. Either finding it in the 'Applications -> Service Workers' menu item in Developer Tools (if using Chrome) or putting `about:debugging#/runtime/this-firefox` in the address bar (if using Firefox)



```
index.html x service-worker.js x app.js x .gitignore x README.md x
1 // Object detection for ServiceWorker
2 if ('serviceWorker' in navigator) {
3 // Register our service worker once the load event has been triggered
4 window.addEventListener( type: 'load', listener: () => {
5     navigator
6     .serviceWorker
7     .register( scriptURL: '/simple-service-worker/scripts/service-worker.js' )
8     .then(() => {
9         console.log("Registration successful");
10    })
11    .catch(() => {
12        console.error("Registration of service-worker.js failed")
13    })
14 });
15 }
```


Example 2. Service Worker events

“A service worker is an **event-driven** worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web-page/site that it is associated with, intercepting and modifying navigation and resource requests”

MDN

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

```
(() => {  
  
  self.addEventListener( type: 'install', listener: event => {  
    // The install event is fired at the end of registration  
    console.log('Service Worker installed');  
  })  
  
  self.addEventListener( type: 'activate', listener: event => {  
    // The 'activate' event signals the Service Worker is ready to take over  
    console.log('Service Worker activated');  
  })  
  
  self.addEventListener( type: 'fetch', listener: event => {  
    console.log(`Fetch event for ${event.request.url}`);  
  
    if (/pwa-light.png/.test(event.request.url)) {  
      event.respondWith(fetch( input: '/service-worker-events/pwa-dark.png'))  
    }  
  })  
})
```

↑
Let's take a closer look at the
'fetch' event

A closer look at the Service Worker 'fetch' event

*Not to be confused
with the Fetch API*

To demo:

- Clear site data, from the Application panel
- Click the 'Service worker events' link
- Note the colour of the logo - and that this logo is loaded via an `` tag in the HTML
- Reload the page and look at the logo
- Close the browser and past the image URL into a new window

```
self.addEventListener( type: 'fetch', listener: event => {  
  console.log(`Fetch event for ${event.request.url}`);  
  
  if (/pwa-light.png/.test(event.request.url)) {  
    event.respondWith(fetch( input: '/service-worker-events/pwa-dark.png'))  
  }  
})
```

The service worker receives a fetch event for every request within its scope.

This includes requests that are made within CSS files, HTML tags and by JavaScript. It even includes the initial request for the HTML file.

Within the handlers we can use `respondWith()` to deliver a different resource - either from a cache, by making an alternative network request, or building a response on the fly

An interesting exercise would be to update the Service Worker so that an alternative HTML file is served.

Example 3. Creating responses 'on the fly'

In addition to fetching 'different' files, we can use the `Response()` constructure to create a response 'on the fly'.

An example of this can be found in the `service-worker-response-on-the-fly` directory

```
self.addEventListener('fetch', event => {
  console.log(`Fetch event for ${event.request.url}`);

  if (event.request.url.includes("style.css")) {
    console.log('Intercepted call to boilerstrap');

    event.respondWith(
      new Response(
        body: "body {background: green!important;} div {outline:40px solid pink;}",
        init: {headers: {"Content-Type": "text/css"}}
      )
    );
  } else if (event.request.url.includes("script.js")) {

    let response_body = `
    let heading = document.getElementsByTagName('h1')[0];
    let initial_text = heading.innerHTML;
    heading.innerHTML = initial_text + ' - appended by service worker';
  `;

    event.respondWith(
      new Response(
        response_body,
        init: { headers: {"Content-Type": "application/javascript; charset=UTF-8"}}
      )
    )
  } else {
    console.log(`Request for ${event.request.url}`)
  }
})
```

Example 4. 'Catching' failed fetches

By intercepting a browser's default response to a Fetch event and interjecting Fetch API call we can detect a user being offline and provide a custom experience.

Note:

- Test event.request.url
- event.respondWith
- Use of catch() on the promise returned by Fetch

```
self.addEventListener( type: "fetch", listener: function (event :Event ) {
  if (event.request.url.includes("index.html")) {
    event.respondWith(
      fetch(event.request).catch(function () {
        return new Response(
          body: `

# Welcome to the The National Archives.</h1> <p>There seems to be a problem with your connection</p>` ); }) ); } });


```

Example 3. Broad brush caching

A slightly more involved example can be seen by clicking the **'Service Worker cache everything'** on the home page when running <https://github.com/gtvj/exploring-pwas>. Then:

1. Disable the cache (in the Network panel) and set it to 'Slow 3G' connection
2. Click the 'Service Worker cache everything' link
3. Check the
 - a. browser console to see the service registration message and confirmation that cache_list has been added to cache
 - b. Application tab to see the registered Service Worker
4. Stop your web server
5. Reload the page

Something to note:

Despite being on a 'Slow 3G' connection the page loads instantly

The files we want
to cache

self is the worker global scope -
like window in normal browser JS
or global in Node

Opening the cache we want
to use and add our items

Attaching an event listener
to all fetch events - including
those raised by HTML or
CSS

Preventing the default fetch
handling and providing our own
Response promise

A closer look at this example

```
const cache_list = [
  'index.html',
  'images/cartographer.png',
  'images/logo-white.png',
  'favicon.png',
  'styles/styles.css',
  'scripts/app.js'
]

self.addEventListener( type: "install", listener: event => {
  caches.open( cacheName: "PRECACHE")
    .then(cache => {
      cache.addAll(cache_list)
        .then(() => {
          console.log('cache_list URLs added to cache');
        })
    })
    .catch(err => {
      console.log(`Error in Service Worker install: ${err}`);
    });
});

self.addEventListener( type: "fetch", listener: event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        if (response) {
          // If it's in the cache return that
          return response;
        }
        // If it's not in the cache, fetch it.
        return fetch(event.request);
      })
    )
    .catch(err => {
      console.log(`Error in Service Worker fetch: ${err}`);
    });
});
```

While this isn't a mountain of code, it's quite a lot considering it's simply saying:

- Cache a bunch of files when the Service Worker is registered
- From then on, respond to fetch requests with the cached version - if one exists. Otherwise make a network request

The point here is that Service Worker APIs are low-level requiring developers to do a lot of 'reinventing the wheel' for common use cases.

In summary

“Service workers essentially act as proxy servers that sit between web applications, the browser, and the network (when available). They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server. They will also allow access to push notifications and background sync APIs.”

[MDN documentation](#)