THE PROGRESSIVE WEB APPS BOOK

BY FLAVIO COPES

HTTPS://FLAVIOCOPES.COM

UPDATED MAY 19, 2018

- Service Workers
- Cache API
- Fetch API
- Progressive Web Apps
- Example: make a CMS-based website work offline

SERVICE WORKERS

Service Workers are a key technology powering Progressive Web Applications on the mobile web. They allow caching of resources and push notifications, two of the main distinguishing features that up to now set native apps apart

- Introduction to Service Workers
- Background Processing
- Offline Support
    - Precache assets during installation
    - Caching network requests
- A Service Worker Lifecycle
    - Registration
    - Scope
    - Installation
    - Activation
- Updating a Service Worker
- Fetch Events
- Background Sync
- Push Events
- A note about console logs

Introduction to Service Workers

Service Workers are at the core of Progressive Web Apps, because they allow caching of resources and push notifications, two of the main distinguishing features that up to now set native apps apart.

A Service Worker is **programmable proxy** between your web page and the network, providing the ability to intercept and cache network requests, effectively **giving you the ability to create an offline-first experience for your app**.

It's 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, giving the benefit of being non-blocking - so computations can be done without sacrificing the UI responsiveness.

Being on a separate thread it has no DOM access, and no access to the Local Storage APIs and the XHR API as well, and 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 for an easier testing.

Background Processing

Service Workers run independent of the application they are associated to, 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**, so 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:

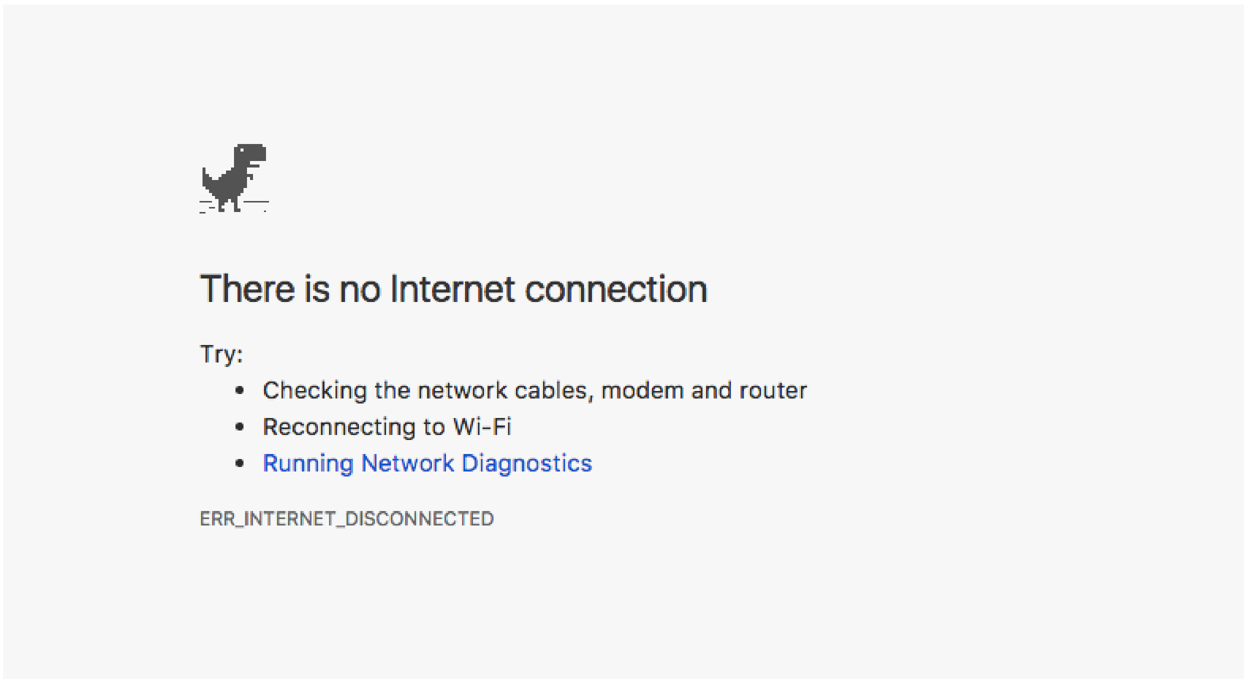
- they can be used as a **caching layer** to handle network requests, and cache content to be used when offline
- to allow **push notifications**

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 web mobile apps simply won't work, while native mobile apps 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 web pages look like in Chrome without a network connection:



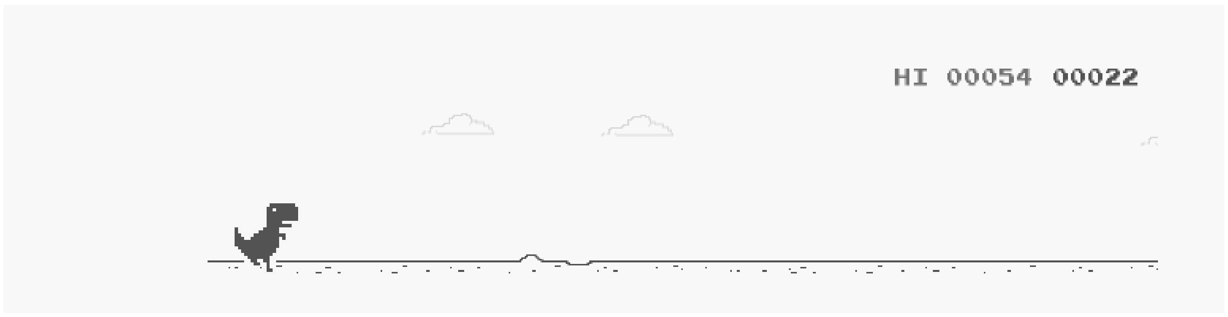
There is no Internet connection

Try:

- Checking the network cables, modem and router
- Reconnecting to Wi-Fi
- Running Network Diagnostics

ERR_INTERNET_DISCONNECTED

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, failing its promises (and it's been discontinued (https://html.spec.whatwg.org/multipage/offline.html#offline)).

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 3 steps to be fully working:

- 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, not registered previously, or if 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 good event to prepare the Service Worker to be used, by **initializing a cache**, and **cache the App Shell** and static assets using the **Cache API**.

Activation

The activation stage is the third step, once the service worker has been successfully registered and installed.

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

It cannot interact with pages already loaded, which means the service worker is only useful on 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 but 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 into it, and 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 already working.

Refreshing the page is not enough, as the old worker is still running and it's not 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 request URL was already stored in the cached responses, and return the cached response if this is the case. 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 a user can use the app offline, and 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, but 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, and 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 service 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.

CACHE API

The Cache API is part of the Service Worker specification, and is a great way to have more power on resources caching.

- Introduction
- Detect if the Cache API is available
- Initialize a cache
- Add items to the cache
    - `cache.add()`
    - `cache.addAll()`
    - Manually fetch and add
- Retrieve an item from the cache
- Get all the items in a cache
- Get all the available caches
- Remove an item from the cache
- Delete a cache

Introduction

The Cache API is part of the Service Worker specification, and is a great way to have more power on resources caching.

It allows you to **cache URL-addressable resources**, which means assets, web pages, HTTP APIs responses.

It’s **not** meant to cache individual chunks of data, which is the task of the **IndexedDB API**.

It’s currently available in Chrome >= 40, Firefox >=39, and Opera >= 27.

Internet Explorer and Safari still do not support it, while Edge has support for that is in preview release.

Mobile support is good on Android, supported on the Android Webview and in Chrome for Android, while on iOS it’s only available to Opera Mobile and Firefox Mobile users.

Detect if the Cache API is available

The Cache API is exposed through the `caches` object. To detect if the API is implemented in the browser, just check for its existance using:

```
if ('caches' in window) {
  //ok
}
```

Initialize a cache

Use the `caches.open` API, which returns a promise with a **cache object** ready to be used:

```
caches.open('mycache').then((cache) => {
  // you can start using the cache
})
```

If the cache does not exist yet, `caches.open` creates it.

Add items to the cache

The `cache` object exposes two methods to add items to the cache: `add` and `addAll`.

`cache.add()`

`add` accepts a single URL, and when called it fetches the resource and caches it.

```
caches.open('mycache').then((cache) => {
  cache.add('/api/todos')
})
```

To allow more control on the fetch, instead of a string you can pass a **Request** object, part of the Fetch API specification:

```
caches.open('mycache').then((cache) => {
  const options = {
    // the options
  }
  cache.add(new Request('/api/todos', options))
})
```

`cache.addAll()`

`addAll` accepts an array, and returns a promise when all the resources have been cached.

```
caches.open('mycache').then((cache) => {
  cache.addAll(['/api/todos', '/api/todos/today']).then(() => {
    //all requests were cached
  })
})
```

Manually fetch and add

`cache.add()` automatically fetches a resource, and caches it.

The Cache API offers a more granular control on this via `cache.put()`. You are responsible for fetching the resource and then telling the Cache API to store a response:

```
const url = '/api/todos'
fetch(url).then((res) => {
  return caches.open('mycache').then((cache) => {
    return cache.put(url, res)
  })
})
```

Retrieve an item from the cache

`cache.match()` returns a **Response** object which contains all the information about the request and the response of the network request

```
caches.open('mycache').then((cache) => {
  cache.match('/api/todos').then((res) => {
    //res is the Response Object
  })
})
```

Get all the items in a cache

```
caches.open('mycache').then((cache) => {
  cache.keys().then((cachedItems) => {
    //
  })
})
```

cachedItems is an array of **Request** objects, which contain the URL of the resource in the `url` property.

Get all the available caches

The `caches.keys()` method lists the keys of every cache available.

```
caches.keys().then((keys) => {
  // keys is an array with the list of keys
})
```

Remove an item from the cache

Given a `cache` object, its `delete()` method removes a cached resource from it.

```
caches.open('mycache').then((cache) => {
  cache.delete('/api/todos')
})
```

Delete a cache

The `caches.delete()` method accepts a cache identifier and when executed it wipes the cache and its cached items from the system.

```
caches.delete('mycache').then(() => {
  // deleted successfully
})
```

FETCH API

Learn all about the Fetch API, the modern approach to asynchronous network requests which uses Promises as a building block

- Introduction to the Fetch API
- Using Fetch
    - Catching errors
- Response Object
    - Metadata
    - headers
    - status
    - statusText
    - url
    - Body content
- Request Object

- Request headers
- POST Requests
- Fetch drawbacks
- How to cancel a fetch request

Introduction to the Fetch API

Since IE5 was released in 1998, we've had the option to make asynchronous network calls in the browser using XMLHttpRequest (XHR).

Quite a few years after this, GMail and other rich apps made heavy use of it, and made the approach so popular that it had to have a name: **AJAX**.

Working directly with the XMLHttpRequest has always been a pain and it was almost always abstracted by some library, in particular jQuery has its own helper functions built around it:

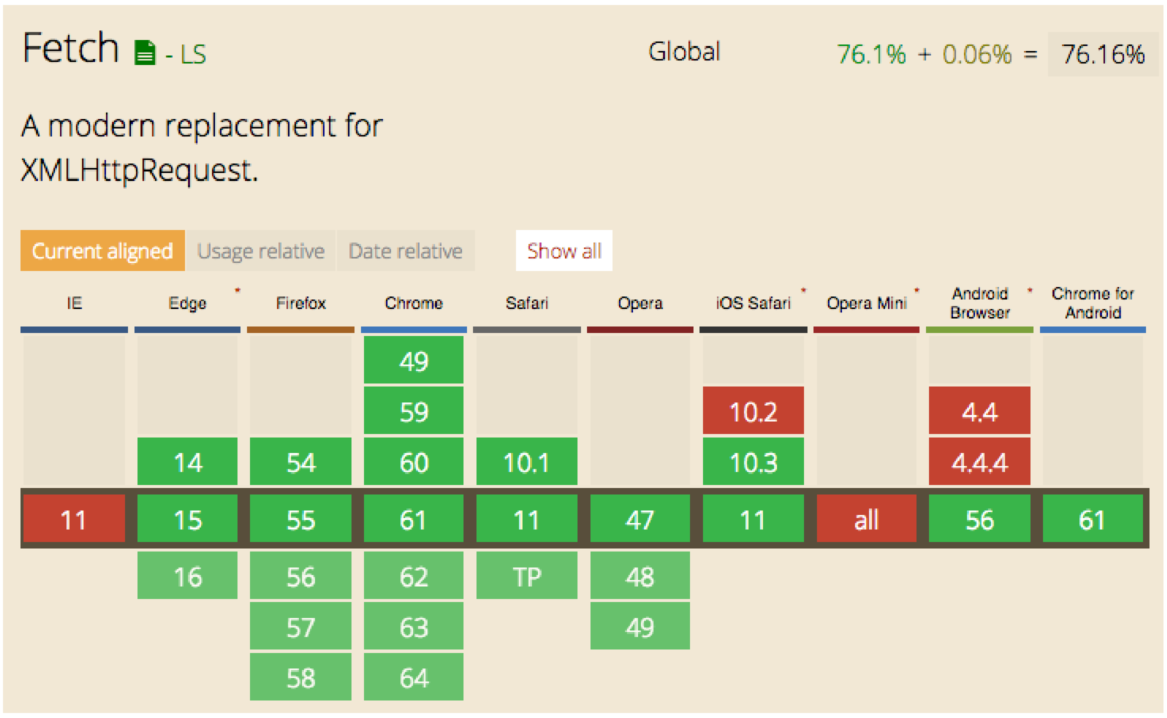
- `jQuery.ajax()`
- `jQuery.get()`
- `jQuery.post()`

and so on.

They had a huge impact on making this more accessible in particular with regards to making sure all worked on older browsers as well.

The **Fetch API**, has been standardized as a modern approach to asynchronous network requests, and uses **Promises** as a building block.

Fetch has a good support across the major browsers, except IE.



Fetch - LS

Global 76.1% + 0.06% = 76.16%

A modern replacement for XMLHttpRequest.

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome for Android
			49						
			59			10.2		4.4	
	14	54	60	10.1		10.3		4.4.4	
11	15	55	61	11	47	11	all	56	61
	16	56	62	TP	48				
		57	63		49				
		58	64						

The polyfill https://github.com/github/fetch released by GitHub allows us to use `fetch` on any browser.

Using Fetch

Starting to use Fetch for GET requests is very simple:

```
fetch('/file.json')
```

and you're already using it: fetch is going to make an HTTP request to get the `file.json` resource on the same domain.

As you can see, the `fetch` function is available in the global `window` scope.

Now let's make this a bit more useful, let's actually see what the content of the file is:

```
fetch('./file.json')
.then(response => response.json())
.then(data => console.log(data))
```

Calling `fetch()` returns a promise. We can then wait for the promise to resolve by passing a handler with the `then()` method of the promise.

That handler receives the return value of the `fetch` promise, a **Response** object.

We'll see this object in details in the next section.

Catching errors

Since `fetch()` returns a promise, we can use the `catch` method of the promise to intercept any error occurring during the execution of the request, and the processing done in the `then` callbacks:

```
fetch('./file.json')
.then(response => {
  //...
}
.catch(err => console.error(err))
```

Response Object

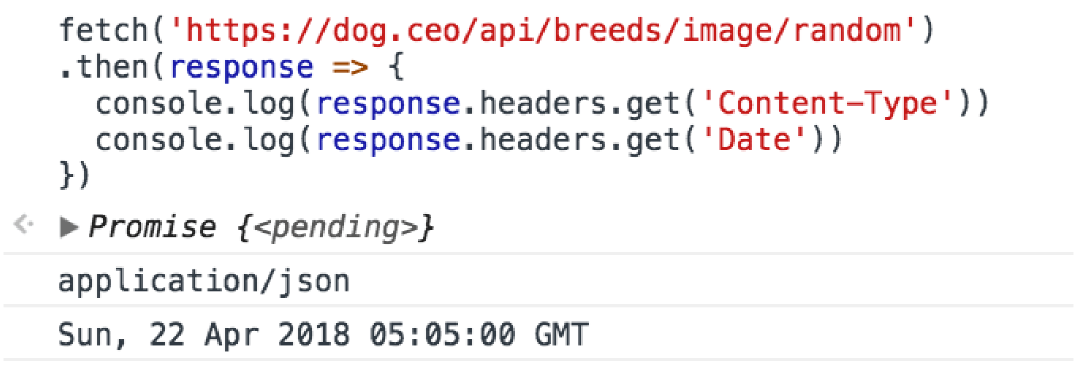
The Response Object returned by a `fetch()` call contains all the information about the request and the response of the network request.

Metadata

headers

Accessing the `headers` property on the `response` object gives you the ability to look into the HTTP headers returned by the request:

```
fetch('./file.json')
.then(response => {
  console.log(response.headers.get('Content-Type'))
  console.log(response.headers.get('Date'))
})
```



```
fetch('https://dog.ceo/api/breeds/image/random')
.then(response => {
  console.log(response.headers.get('Content-Type'))
  console.log(response.headers.get('Date'))
})
< ▶ Promise {<pending>}
application/json
Sun, 22 Apr 2018 05:05:00 GMT
```

status

This property is an integer number representing the HTTP response status.

- 101, 204, 205, or 304 is a null body status
- 200 to 299, inclusive, is an OK status (success)
- 301, 302, 303, 307, or 308 is a redirect

```
fetch('./file.json')
.then(response => console.log(response.status))
```

statusText

`statusText` is a property representing the status message of the response. If the request is successful, the status is `OK`.

```
fetch('./file.json')
.then(response => console.log(response.statusText))
```

url

`url` represents the full URL of the property that we fetched.

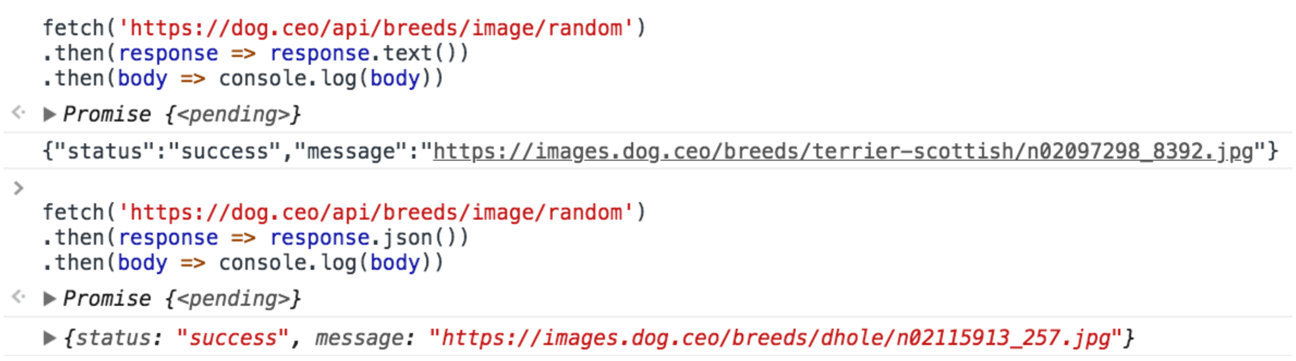
```
fetch('./file.json')
.then(response => console.log(response.url))
```

Body content

A response has a body, accessible using the `text()` or `json()` methods, which return a promise.

```
fetch('./file.json')
.then(response => response.text())
.then(body => console.log(body))
```

```
fetch('./file.json')
.then(response => response.json())
.then(body => console.log(body))
```



```
fetch('https://dog.ceo/api/breeds/image/random')
.then(response => response.text())
.then(body => console.log(body))
< ▶ Promise {<pending>}
{"status":"success","message":"https://images.dog.ceo/breeds/terrier-scottish/n02097298_8392.jpg"}
>
fetch('https://dog.ceo/api/breeds/image/random')
.then(response => response.json())
.then(body => console.log(body))
< ▶ Promise {<pending>}
▶ {status: "success", message: "https://images.dog.ceo/breeds/dhole/n02115913_257.jpg"}
```

The same can be written using the ES2017 async functions:

```
(async () => {
  const response = await fetch('./file.json')
  const data = await response.json()
  console.log(data)
})()
```

Request Object

The Request object represents a resource request, and it's usually created using the `new Request()` API.

Example:

```
const req = new Request('/api/todos')
```

The Request object offers several read-only properties to inspect the resource request details, including

- `method`: the request's method (GET, POST, etc.)
- `url`: the URL of the request.
- `headers`: the associated Headers object of the request
- `referrer`: the referrer of the request
- `cache`: the cache mode of the request (e.g., default, reload, no-cache).

And exposes several methods including `json()`, `text()` and `formData()` to process the body of the request.

The full API can be found at https://developer.mozilla.org/docs/Web/API/Request

Request headers

Being able to set the HTTP request header is essential, and `fetch` gives us the ability to do this using the Headers object:

```
const headers = new Headers();
headers.append('Content-Type', 'application/json')
```

or more simply

```
const headers = new Headers({
  'Content-Type': 'application/json'
})
```

To attach the headers to the request, we use the Request object, and pass it to `fetch()` instead of simply passing the URL.

Instead of:

```
fetch('./file.json')
```

we do

```
const request = new Request('./file.json', {
    headers: new Headers({
        'Content-Type': 'application/json'
    })
})
fetch(request)
```

The Headers object is not limited to setting value, but we can also query it:

```
headers.has('Content-Type');
headers.get('Content-Type');
```

and we can delete a header that was previously set:

```
headers.delete('X-My-Custom-Header');
```

POST Requests

Fetch also allows to use any other HTTP method in your request: POST, PUT, DELETE or OPTIONS.

Specify the method in the method property of the request, and pass additional parameters in the header and in the request body:

Example of a POST request:

```
const options = {
  method: 'post',
  headers: {
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
  },
  body: 'foo=bar&test=1'
}

fetch(url, options)
.catch((err) => {
  console.error('Request failed', err)
})
```

Fetch drawbacks

While it's a great improvement over XHR, especially considering its Service Workers integration, Fetch currently has **no way to abort a request** once it's done. With Fetch it's also **hard to measure upload progress**.

If you need those things in your app, the Axios JavaScript library might be a better fit.

How to cancel a fetch request

For a few years after `fetch` was introduced, there was no way to abort a request once opened.

Now we can, thanks to the introduction of `AbortController` and `AbortSignal`, a generic API to notify **abort** events

You integrate this API by passing a signal as a fetch parameter:

```
const controller = new AbortController()
const signal = controller.signal

fetch('./file.json', { signal })
```

You can set a timeout that fires an abort event 5 seconds after the fetch request has started, to cancel it:

```
setTimeout(() => controller.abort(), 5 * 1000)
```

Conveniently, if the fetch already returned, calling `abort()` won't cause any error.

When an abort signal occurs, fetch will reject the promise with a `DOMException` named `AbortError`:

```
fetch('./file.json', { signal })
.then(response => response.text())
.then(text => console.log(text))
.catch(err => {
  if (err.name === 'AbortError') {
    console.error('Fetch aborted')
  } else {
    console.error('Another error', err)
  }
})
```

PROGRESSIVE WEB APPS

A Progressive Web App is an app that can provide additional features based on the device support, including offline capabilities, push notifications and almost native app look and speed, and local caching of resources

- Introduction
- What is a Progressive Web App
- Progressive Web Apps alternatives
    - Native Mobile Apps
    - Hybrid Apps
    - Apps built with React Native
- Progressive Web Apps features
    - Features
    - Benefits
    - Core concepts
- Service Workers
- The App Manifest
    - Example
- The App Shell
    - Caching

Introduction

Progressive Web Apps (PWA) are the latest trend of **mobile application development** using web technologies, at the time of writing (march 2018) work on Android and iOS devices with iOS 11.3 or higher, and macOS 10.13.4 or higher.

PWA is a term that identifies a bundle of techniques that have the goal of creating a better experience for web-based apps.

What is a Progressive Web App

A Progressive Web App is an app that *can* **provide additional features based on the device support**, providing offline capability, push notifications and almost native app look and speed, and local caching of resources.

This technique was originally introduced by Google in 2015, and proves to bring many advantages to both the developer and the users.

Developers have access to building **almost-first-class** applications using a web stack, which is always considerably easier and cheaper than building native applications, especially when considering the implications of building and maintaining cross-platform apps.

Devs can benefit from a **reduced installation friction**, at a time when having an app in the store does not actually bring anything in terms of discoverability for 99,99% of the apps, and Google search can provide the same benefits if not more.

A Progressive Web App is a website which is developed with certain technologies that make the mobile experience much more pleasant than a normal mobile-optimized website, to a point that it's almost working like a native app, as it offers the following features:

- Offline support
- Loads fast
- Is secure
- Is capable of emitting push notifications

- Has an immersive, full-screen user experience without the URL bar

Mobile platforms (Android at the time of writing, but it's not technically limited to that) offer an increasing support for Progressive Web Apps to the point of asking the user to **add the app to the home screen** when they detect a site a user is visiting is a PWA.

But first, a little clarification on the name. *Progressive Web App* can be a **confusing term**, and a good definition is *web apps that take advantage of modern browsers features (like web workers and the web app manifest) to let their mobile devices "upgrade" the app to the role of a first-class citizen app*.

Progressive Web Apps alternatives

How does a PWA stand compared to the alternatives when it comes to building a mobile experience?

Let's focus on the pros and cons of each, and let's see where PWAs are a good fit.

Native Mobile Apps

Native mobile apps are the most obvious way to build a mobile app. Objective-C or Swift on iOS, Java / Kotlin on Android and C# on Windows Phone.

Each platform has its own UI and UX conventions, and the native widgets provide the experience that the user expects. They can be deployed and distributed through the platform App Store.

The main pain point with native apps is that cross-platform development requires learning, mastering and keeping up to date with many different methodologies and best practices, so if for example you have a small team or even you're a solo developer building an app on 3 platforms, you need to spend a lot of time learning the technology but also the environment, manage different libraries, and use different workflows (for example, iCloud only works on iOS devices, there's no Android version).

Hybrid Apps

Hybrid applications are built using Web Technologies, but deployed to the App Store. In the middle sits a framework or some way to package the application so it's possible to send it for review to the traditional App Store.

Most common platforms are Phonegap, Xamarin, Ionic Framework, and many others, and usually what you see on the page is a WebView that essentially loads a local website.

The key aspect of Hybrid Apps is the **write once, run anywhere** concept, the different platform code is generated at build time, and you're building apps using JavaScript, HTML and CSS, which is amazing, and the device capabilities (microphone, camera, network, gps...) are exposed through JavaScript APIs.

The bad part of building hybrid apps is that unless you do a great job, you might settle on providing a common denominator, effectively creating an app that's sub-optimal on all platforms because the app is ignoring the platform-specific human-computer interaction guidelines.

Also, performance for complex views might suffer.

Apps built with React Native

React Native exposes the native controls of the mobile device through a JavaScript API, but you're effectively creating a native application, not embedding a website inside a WebView.

Their motto, to distinguish this approach from hybrid apps, is **learn once, write anywhere**, meaning that the approach is the same across platforms, but you're going to create completely separate apps in order to provide a great experience on each platform.

Performance is comparable to native apps, since what you build is essentially a native app, which is distributed through the App Store.

Progressive Web Apps features

In the last section you saw the main *competitors* of Progressive Web Apps. So how do PWAs stand compared to them, and what are their main features?

> *Remember, currently Progressive Web Apps are Android-only*

Features

Progressive Web Apps have one thing that separates them completely from the above approaches: **they are not deployed to the app store.**.

This is a key advantage, since the app store is beneficial if you have the reach and luck to be featured, which can make your app go viral, but unless you're in the 0,001% you're not going to get much benefits from having your little place on the App Store.

Progressive Web Apps are **discoverable using Search Engines**, and when a user gets to your site which has PWAs capabilities, **the browser in combination with the device asks the user if they want to install the app to the home screen**. This is huge because regular SEO can apply to your PWA, leading to much less reliance on paid acquisition.

Not being in the App Store means **you don't need the Apple or Google approval** to be in the users pockets, and you can release updates when you want, without having to go through the standard approval process which is typical of iOS apps.

PWAs are basically HTML5 applications / responsive websites on steroids, with some key technologies that were recently introduced that make some of the key features possible. If you remember the original iPhone came without the option to develop native apps, and developers were told to develop HTML5 mobile apps, that could be installed to the home screen, but the tech back then was not ready for this.

Progressive Web Apps **run offline**.

The use of **service workers** allow the app to always have fresh content, and download it in the background, and provide support for **push notifications** to provide greater re-engagement opportunities.

Also, shareability makes for a much nicer experience for users that want to share your app, as they just need a URL.

Benefits

So why should users and developers care about Progressive Web Apps?

1. PWA are lighter. Native Apps can weight 200MB or more, while a PWA could be in the range of the KBs.
2. No native platform code
3. Lower the cost of acquisition (it’s much more hard to convince a user to install an app than to visit a website to get the first-time experience)
4. Significant less effort is needed to build and release updates
5. Much more support for deep links than regular app-store apps

Core concepts

- **Responsive**: the UI adapts to the device screen size
- **App-like feel**: it doesn’t feel like a website, but rather as an app as much as possible
- **Offline support**: it will use the device storage to provide offline experience
- **Installable**: the device browser prompts the user to install your app
- **Re-engaging**: push notifications help users re-discover your app once installed

- **Discoverable**: search engines and SEO optimization can provide a lot more users than the app store
- **Fresh**: the app updates itself and the content once online
- **Safe**: uses HTTPS
- **Progressive**: it will work on any device, even older one, even if with less features (e.g. just as a website, not installable)
- **Linkable**: easy to point to it, using URLs

Service Workers

Part of the Progressive Web App definition is that it must work offline.

Since the thing that allows the web app to work offline is the Service Worker, this implies that **Service Workers are a mandatory part of a Progressive Web App**.

> *WARNING: Service Workers are currently only supported by Chrome (Desktop and Android), Firefox and Opera. See http://caniuse.com/#feat=serviceworkers for updated data on the support.*
> *TIP: Don't confuse Service Workers with Web Workers. They are a completely different thing.*

A Service Worker is a JavaScript file that acts as a middleman between the web app and the network. Because of this it can provide cache services and speed the app rendering and improve the user experience.

Because of security reasons, only HTTPS sites can make use of Service Workers, and this is part of the reasons why a Progressive Web App must be served through HTTPS.

Service Workers are not available on the device the first time the user visits the app. What happens is that the first visit the web worker is installed, and then on subsequent visits to separate pages of the site will call this Service Worker.

> *Check out the complete guide to Service Workers*

The App Manifest

The App Manifest is a JSON file that you can use to provide the device information about your Progressive Web App.

You add a link to the manifest in **all** your web site pages header:

```
<link rel="manifest" href="/manifest.webmanifest">
```

This file will tell the device how to set:

- The name and short name of the app
- The icons locations, in various sizes
- The starting URL, relative to the domain
- The default orientation
- The splash screen

Example

```
{
  "name": "The Weather App",
  "short_name": "Weather",
  "description": "Progressive Web App Example",
  "icons": [{
    "src": "images/icons/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-144x144.png",
      "sizes": "144x144",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-152x152.png",
      "sizes": "152x152",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }, {
      "src": "images/icons/icon-256x256.png",
      "sizes": "256x256",
      "type": "image/png"
    }],
  "start_url": "/index.html?utm_source=app_manifest",
  "orientation": "portrait",
  "display": "standalone",
  "background_color": "#3E4EB8",
  "theme_color": "#2F3BA2"
}
```

The App Manifest is a W3C Working Draft, reachable at
https://www.w3.org/TR/appmanifest/

The App Shell

The App Shell is not a technology but rather a **design concept** aimed at loading and rendering the web app container first, and the actual content shortly after, to give the user a nice app-like impression.

This is the equivalent of the Apple HIG (Human Interface Guidelines) suggestions to use a splash screen that resembles the user interface, to give a psychological hint that was found to lower the perception of the app taking a long time to load.

Caching

The App Shell is cached separately from the contents, and it's setup so that retrieving the shell building blocks from the cache takes very little time.

Find out more on the App Shell at
https://developers.google.com/web/updates/2015/11/app-shell

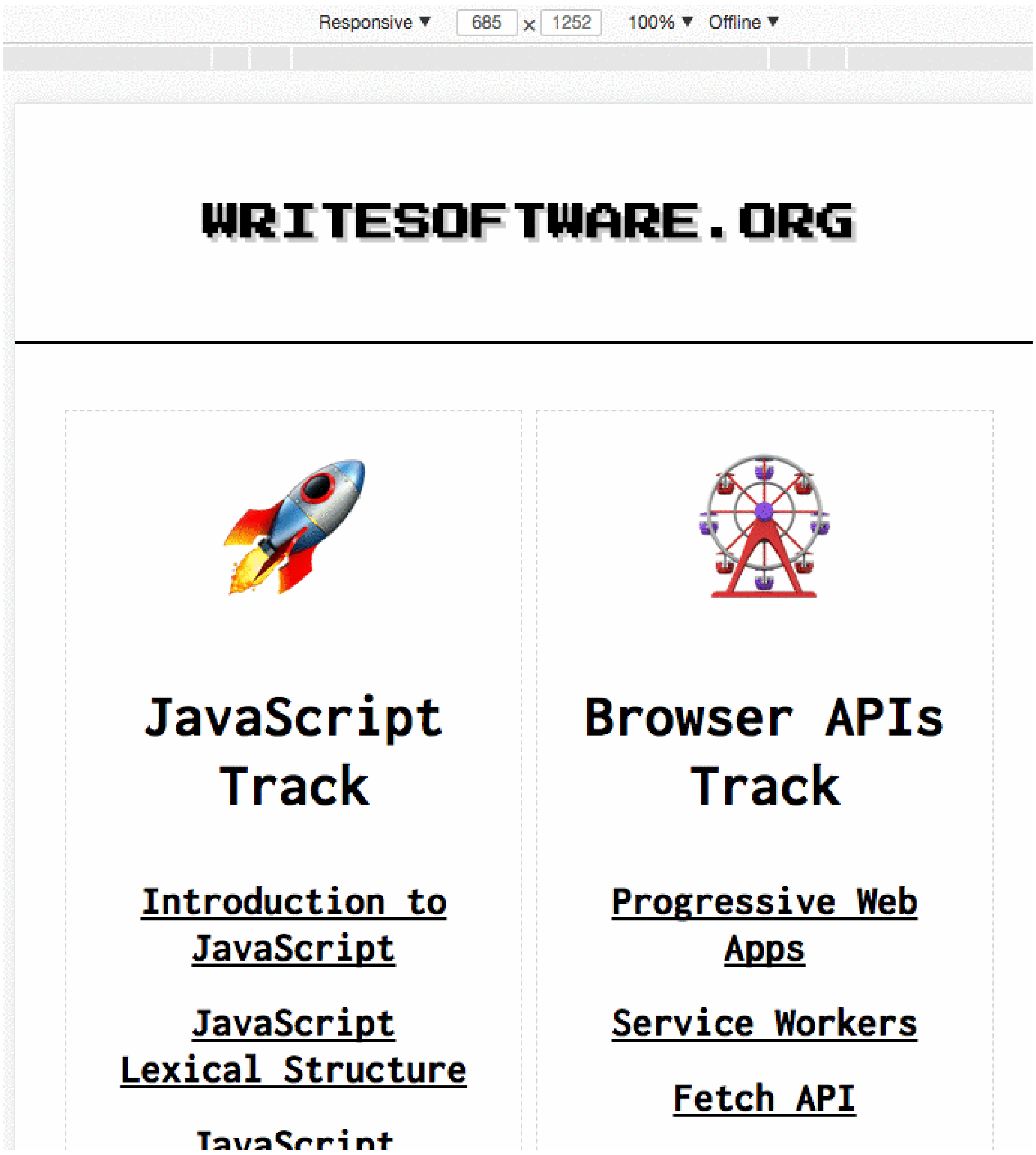
MAKE A CMS-BASED WEBSITE WORK OFFLINE

Progressively enhance a website when viewed on modern devices

- First approach: cache-first
    - Introducing a service worker
    - Fix URL, title and back button with the History API
    - Fix Google Analytics
- Second approach: network-first, drop the app shell
- Going simpler: no partials

This case study explains how I added the capability of working offline to the writesoftware.org (https://writesoftware.org) website, which is based on Grav, a great PHP-based CMS for developers (https://getgrav.org) , by introducing a set of technologies grouped under the name of Progressive Web Apps (in particular Service Workers and the Cache API).

When we're finished, we'll be able to use our site on a mobile device or on a desktop, even if offline, like shown here below (notice the "Offline" option in the network throttling settings)


Responsive ▼ 685 × 1252 100% ▼ Offline ▼

WRITESOFTWARE.ORG

JavaScript Track

Introduction to JavaScript

JavaScript Lexical Structure

Browser APIs Track

Progressive Web Apps

Service Workers

Fetch API

First approach: cache-first

I first approached the task by using a cache-first approach. In short, when we intercept a fetch request in the Service Worker, we first check if we have it cached already. If not, we fetch it from the network. This has the advantage of making the site blazing fast when loading pages already cached, even when online - in particular with slow networks and lie-fi (https://developers.google.com/web/fundamentals/performance/poor-connectivity/#what_is_lie-fi) - but also introduces some complexity in managing updating the cache when I ship new content.

This will not be the final solution I adopt, but it’s worth going through it for demonstration purposes.

I’ll go through a couple phases:

1. I introduce a service worker and load it as part of the website JS scripts
2. when installing the service worker, I cache the site **skeleton**
3. I intercept requests going to additional links, caching it

Introducing a service worker

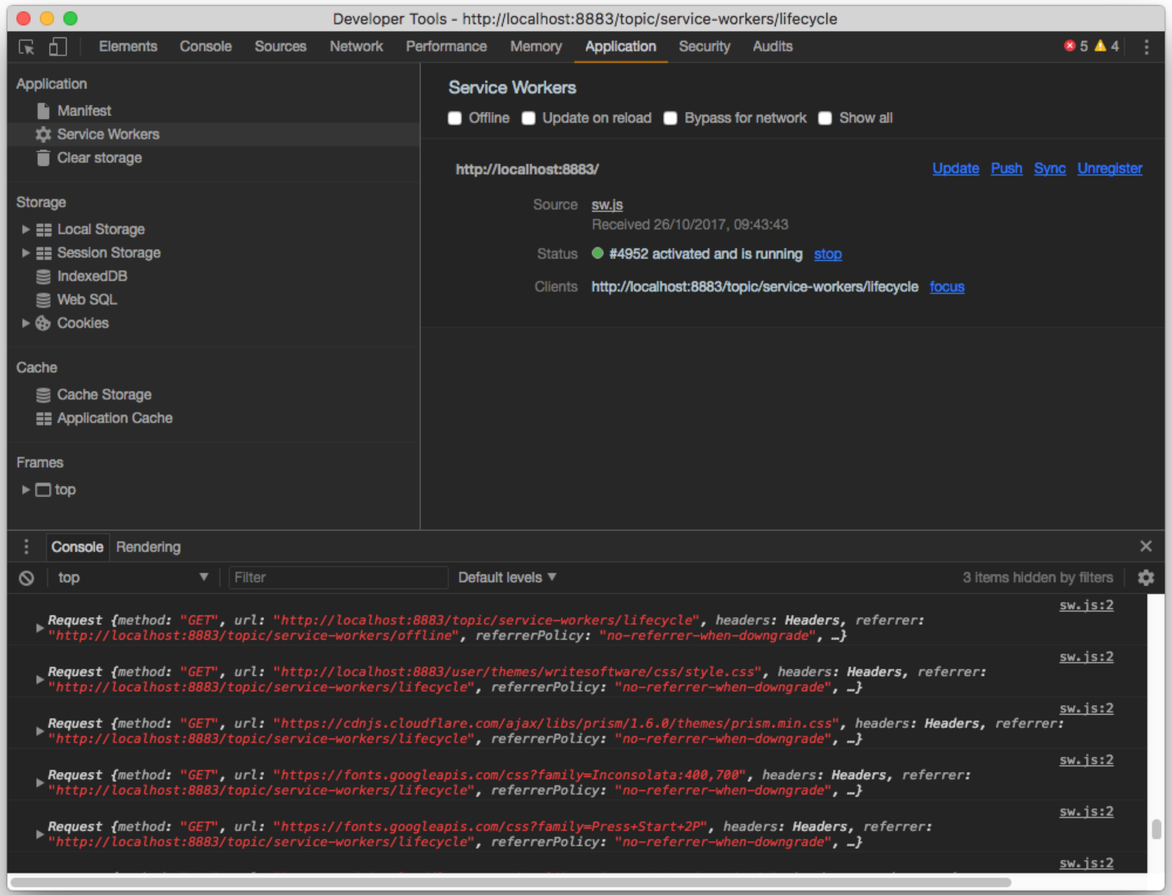
I add the service worker in a `sw.js` file in the site root. This allows it to work on all the site subfolders, and on the site home as well. The SW at the moment is pretty basic, it just logs any network request:

```
self.addEventListener('fetch', (event) => {
  console.log(event.request)
})
```

I need to register the service worker, and I do this from a script that I include in every page:

```
window.addEventListener('load', () => {
  if (!navigator.serviceWorker) {
    return
  }

  navigator.serviceWorker.register('/sw.js', {
    scope: '/'
  }).then(() => {
    //...ok
  }).catch((err) => {
    console.log('registration failed', err)
  })
})
```

If service workers are available, we register the `sw.js` file and the next time I refresh the page it should be working fine:



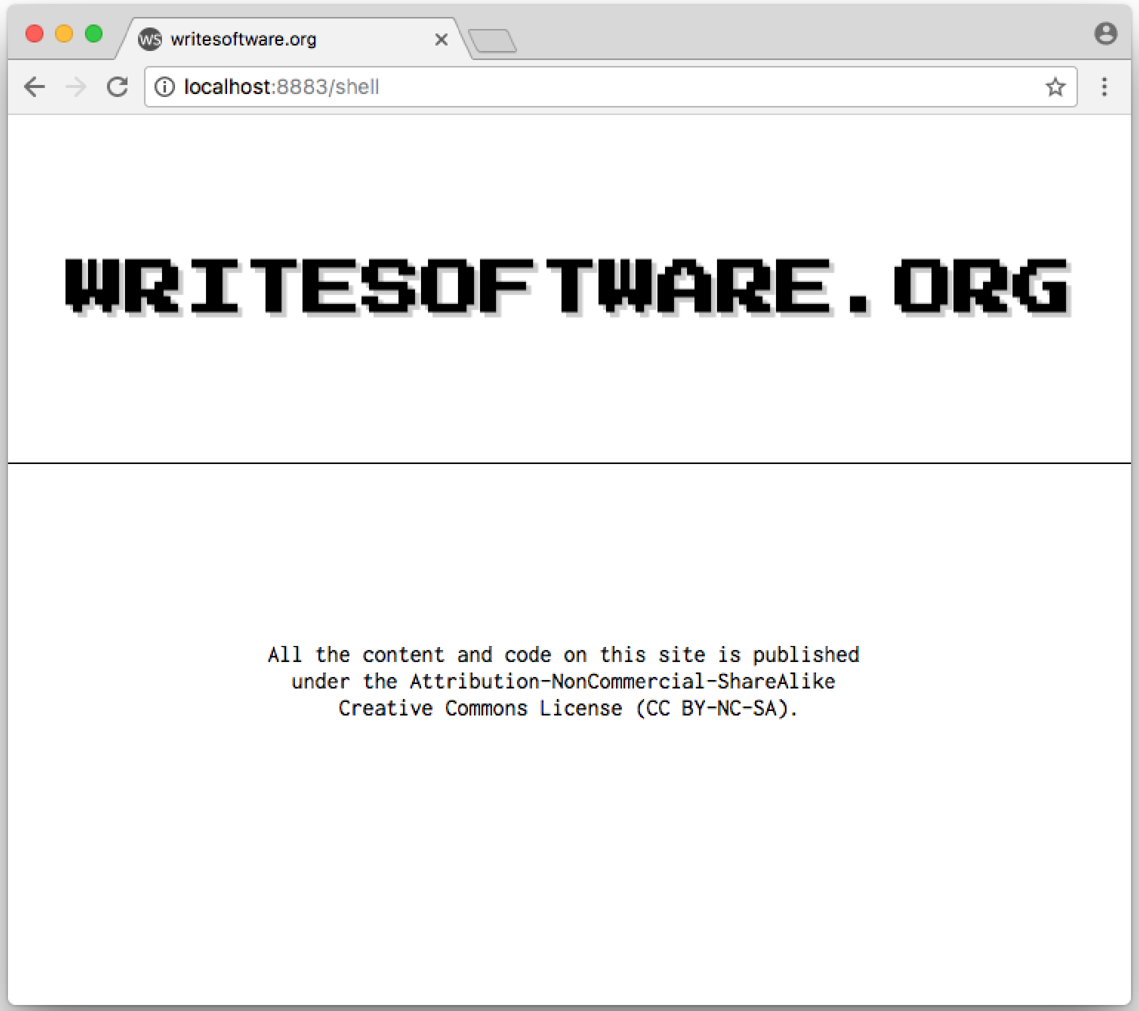
http://localhost:8883/

Source sw.js
Received 26/10/2017, 09:43:43
Status #4952 activated and is running stop
Clients http://localhost:8883/topic/service-workers/lifecycle focus

```
Request {method: "GET", url: "http://localhost:8883/topic/service-workers/lifecycle", headers: Headers, referrer:   sw.js:2
"http://localhost:8883/topic/service-workers/offline", referrerPolicy: "no-referrer-when-downgrade", …}
Request {method: "GET", url: "http://localhost:8883/user/themes/writesoftware/css/style.css", headers: Headers, referrer:   sw.js:2
"http://localhost:8883/topic/service-workers/lifecycle", referrerPolicy: "no-referrer-when-downgrade", …}
Request {method: "GET", url: "https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/themes/prism.min.css", headers: Headers, referrer:   sw.js:2
"http://localhost:8883/topic/service-workers/lifecycle", referrerPolicy: "no-referrer-when-downgrade", …}
Request {method: "GET", url: "https://fonts.googleapis.com/css?family=Inconsolata:400,700", headers: Headers, referrer:   sw.js:2
"http://localhost:8883/topic/service-workers/lifecycle", referrerPolicy: "no-referrer-when-downgrade", …}
Request {method: "GET", url: "https://fonts.googleapis.com/css?family=Press+Start+2P", headers: Headers, referrer:   sw.js:2
"http://localhost:8883/topic/service-workers/lifecycle", referrerPolicy: "no-referrer-when-downgrade", …}
                                                                                                                             sw.js:2
```

At this point I need to do some heavy lifting on the site. First of all, I need to come up with a way to serve only the **App Shell**: a basic set of HTML + CSS and JS that will be always available and shown to the users, even when offline.

It's basically a stripped down version of the website, with a `<div class="wrapper row" id="content-wrapper"></div>` empty element, which we'll fill with content later, available under the `/shell` route:



All the content and code on this site is published
under the Attribution-NonCommercial-ShareAlike
Creative Commons License (CC BY-NC-SA).

So the first time the user loads the site, the normal version will be shown (full-HTML version), and the service worker is installed.

Now any other page that is clicked is intercepted by our Service Worker. Whenever a page is loaded, we load the shell first, and then we load a stripped-down version of the page, without the shell: just the content.

How?

We listen for the `install` event, which fires when the Service Worker is installed or updated, and when this happens we initialize the cache with the content of our shell: the basic HTML layout, plus some CSS, JS and some external assets:

```
const cacheName = 'writesoftware-v1'

self.addEventListener('install', (event) => {
  event.waitUntil(caches.open(cacheName).then(cache => cache.addAll([
    '/shell',
    'user/themes/writesoftware/favicon.ico',
    'user/themes/writesoftware/css/style.css',
    'user/themes/writesoftware/js/script.js',
    'https://fonts.googleapis.com/css?family=Press+Start+2P',
    'https://fonts.googleapis.com/css?family=Inconsolata:400,700',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/themes/prism.min.css',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/prism.min.js',
    'https://cdn.jsdelivr.net/prism/1.6.0/components/prism-jsx.min.js'
  ])))
})
```

Then when we perform a fetch, we intercept requests to our pages, and fetch the shell from the Cache instead of going to the network.

If the URL belongs to Google Analytics or ConvertKit I avoid using the local cache, and I fetch them without using CORS, since they deny accessing them through this method.

Then, if I'm requesting a local partial (just the content of a page, not the full page) I just issue a fetch request to get it.

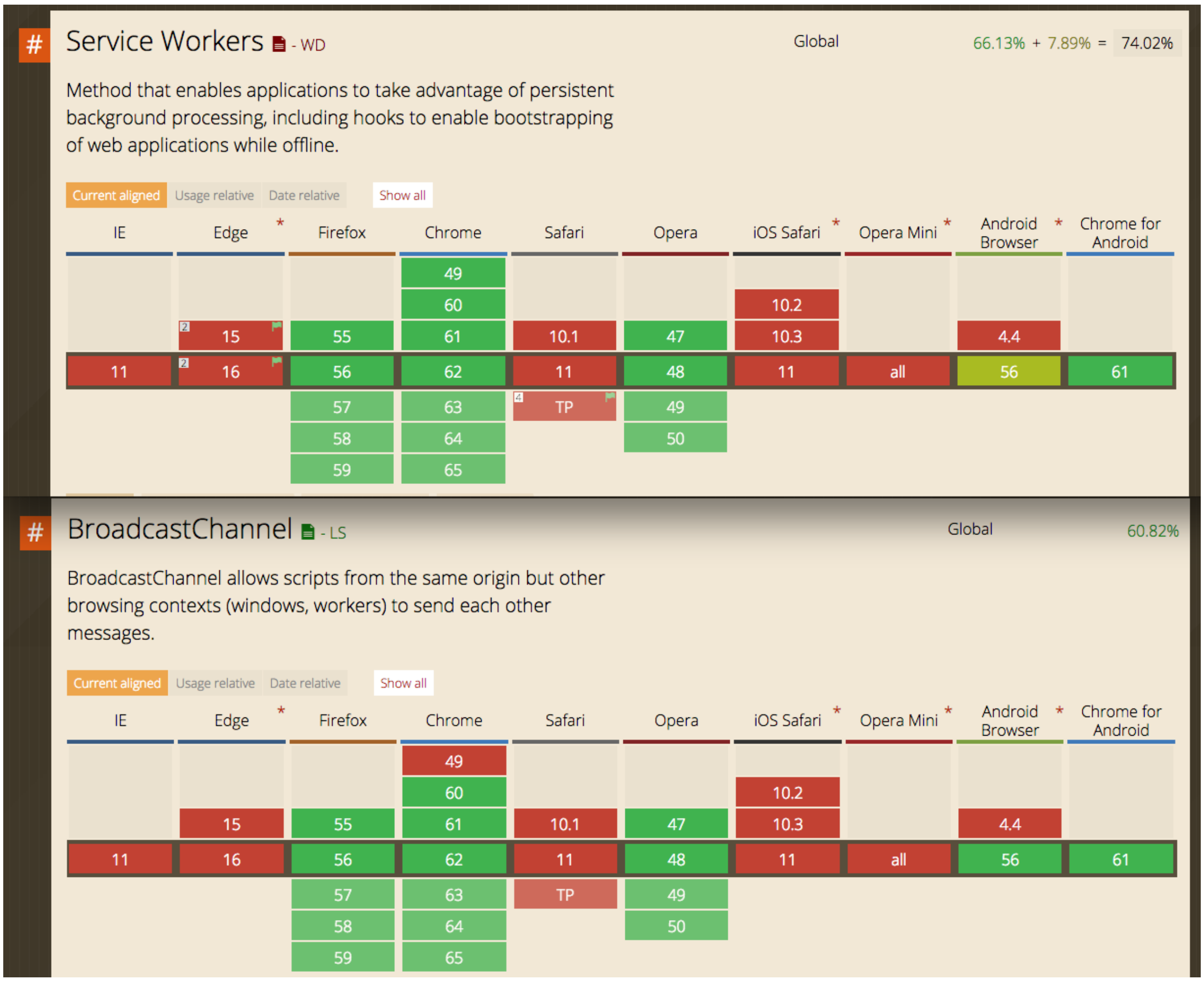
If it's not a partial, we return the shell, which is already cached when the Service Worker is first installed.

Once the fetch is done, I cache it.

```
self.addEventListener('fetch', (event) => {
  const requestUrl = new URL(event.request.url)

  if (requestUrl.href.startsWith('https://www.googletagmanager.com') ||
      requestUrl.href.startsWith('https://www.google-analytics.com') ||
      requestUrl.href.startsWith('https://assets.convertkit.com')) {
    // don't cache, and no cors
    event.respondWith(fetch(event.request.url, { mode: 'no-cors' }))
    return
  }

  event.respondWith(caches.match(event.request)
    .then((response) => {
      if (response) { return response }
      if (requestUrl.origin === location.origin) {
        if (requestUrl.pathname.endsWith('?partial=true')) {
          return fetch(requestUrl.pathname)
        } else {
          return caches.match('/shell')
        }

        return fetch(`${event.request.url}?partial=true`)
      }
      return fetch(event.request.url)
    })
    .then(response => caches.open(cacheName).then((cache) => {
      cache.put(event.request.url, response.clone())
      return response
    }))
    .catch((error) => {
      console.error(error)
    }))
})
```

Now, I edit the `script.js` file to introduce an important feature: whenever a link is clicked on my pages, I intercept it and I issue a message to a Broadcast Channel.

Since Service Workers are currently only supported in Chrome, Firefox and Opera, I can safely rely on the BroadcastChannel API (https://developers.google.com/web/updates/2016/09/broadcastchannel) for this.



Service Workers - WD

Global 66.13% + 7.89% = 74.02%

Method that enables applications to take advantage of persistent background processing, including hooks to enable bootstrapping of web applications while offline.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
			60			10.2			
	15	55	61	10.1	47	10.3		4.4	
11	16	56	62	11	48	11	all	56	61
		57	63	TP	49				
		58	64		50				
		59	65						

BroadcastChannel - LS

Global 60.82%

BroadcastChannel allows scripts from the same origin but other browsing contexts (windows, workers) to send each other messages.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
			60			10.2			
	15	55	61	10.1	47	10.3		4.4	
11	16	56	62	11	48	11	all	56	61
		57	63	TP	49				
		58	64		50				
		59	65						

First, I connect to the `ws_navigation` channel and I attach a `onmessage` event handler on it. Whenever I receive an event, it's a communication from the Service Worker with new content to show inside the App Shell, so I just lookup the element with id `content-wrapper` and I put the partial page content into it, effectively changing the page the user is seeing.

As soon as the Service Worker is registered I issue a message to this channel, with a `fetchPartial` task and a partial page URL to fetch. This is the content of the initial page load.

The shell is loaded immediately, since it's always cached and soon after, the actual content is looked up, which might be cached as well.

```
window.addEventListener('load', () => {
  if (!navigator.serviceWorker) { return }
  const channel = new BroadcastChannel('ws_navigation')

  channel.onmessage = (event) => {
    if (document.getElementById('content-wrapper')) {
      document.getElementById('content-wrapper').innerHTML = event.data.content
    }
  }

  navigator.serviceWorker.register('/sw.js', {
    scope: '/'
  }).then(() => {
    channel.postMessage({
      task: 'fetchPartial',
      url: `${window.location.pathname}?partial=true`
    })
  }).catch((err) => {
    console.log('SW registration failed', err)
  })
})
```

The missing bit is handing a click on the page. When a link is clicked, I intercept the event, halt it and I send a message to the Service Worker to fetch the partial with that URL.

When fetching a partial, I attach a `?partial=true` query to tell my backend to only serve the content, not the shell.

```
window.addEventListener('load', () => {

  //...

  window.onclick = (e) => {
    let node = e.target
    while (node !== undefined && node !== null && node.localName !== 'a') {
      node = node.parentNode
    }
    if (node !== undefined && node !== null) {
      channel.postMessage({
        task: 'fetchPartial',
        url: `${node.href}?partial=true`
      })
      return false
    }
    return true
  }
})
```

Now we just miss to handle this event. On the Service Worker side, I connect to the `ws_navigation` channel and listen for an event. I listen for the `fetchPartial` message task name, although I could simply avoid this condition check as this is the only event that's being sent here (messages in the Broadcast Channel API are not dispatched to the same page that's originating them - only between a page and a web worker).

I check if the url is cached. If so, I just send it as a response message on the channel, and return.

If it's not cached, I fetch it, send it back as a message to the page, and then cache it for the next time it might be visited.

```
const channel = new BroadcastChannel('ws_navigation')
channel.onmessage = (event) => {
  if (event.data.task === 'fetchPartial') {
    caches
      .match(event.data.url)
      .then((response) => {
        if (response) {
          response.text().then((body) => {
            channel.postMessage({ url: event.data.url, content: body })
          })
          return
        }

        fetch(event.data.url).then((fetchResponse) => {
          const fetchResponseClone = fetchResponse.clone()
          fetchResponse.text().then((body) => {
            channel.postMessage({ url: event.data.url, content: body })
          })

          caches.open(cacheName).then((cache) => {
            cache.put(event.data.url, fetchResponseClone)
          })
        })
      })
      .catch((error) => {
        console.error(error)
      })
  }
}
```

We’re almost done.

Now the Service Worker is installed on the site as soon as a user visits, and subsequent page loads are handled dynamically through the Fetch API, not requiring a full page load. After the first visit, pages are cached and load incredibly fast, and - more importantly - then even load when offline!

And - all this is a progressive enhancement. Older browsers, and browsers that don’t support service workers, simply work as normal.

Now, hijacking the browser navigation poses us a few problems:

1. the URL must change when a new page is shown. The back button should work normally, and the browser history as well
2. the page title must change to reflect the new page title
3. we need to notify the Google Analytics API that a new page has been loaded, to avoid missing an important metric such as the page views per visitor.
4. the code snippets are not highlighted any more when loading new content dynamically

Let’s solve those challenges.

Fix URL, title and back button with the History API

In the message handler in script.js in addition to injecting the HTML of the partial, we trigger the `history.pushState()` method of the History API:

```
channel.onmessage = (event) => {
  if (document.getElementById('content-wrapper')) {
    document.getElementById('content-wrapper').innerHTML = event.data.content
    const url = event.data.url.replace('?partial=true', '')
    history.pushState(null, null, url)
  }
}
```

This is working but the page title does not change in the browser UI. We need to fetch it somehow from the page. I decided to put in the page content partial a hidden span that keeps the page title, so we can fetch it from the page using the DOM API, and set the `document.title` property:

```
channel.onmessage = (event) => {
  if (document.getElementById('content-wrapper')) {
    document.getElementById('content-wrapper').innerHTML = event.data.content
    const url = event.data.url.replace('?partial=true', '')
    if (document.getElementById('browser-page-title')) {
      document.title = document.getElementById('browser-page-title').innerHTML
    }
    history.pushState(null, null, url)
  }
}
```

Fix Google Analytics

Google Analytics works fine out of the box, but when loading a page dynamically, it can't do miracles. We must use the API it provides to inform it of a new page load. Since I'm using the Global Site Tag (`gtag.js`) tracking, I need to call:

```
gtag('config', 'UA-XXXXXX-XX', {'page_path': '/the-url'})
```

into the code above that handles changing page:

```
channel.onmessage = (event) => {
  if (document.getElementById('content-wrapper')) {
    document.getElementById('content-wrapper').innerHTML = event.data.content
    const url = event.data.url.replace('?partial=true', '')
    if (document.getElementById('browser-page-title')) {
      document.title = document.getElementById('browser-page-title').innerHTML
    }
    history.pushState(null, null, url)
    gtag('config', 'UA-XXXXXX-XX', {'page_path': url})
  }
}
```

The last thing I need to fix on my page is the code snippets login their highlighing. I use the Prism syntax highlighter and they make it very easy, I just need to add a call `Prism.highlightAll()` in my onmessage handler:

```
channel.onmessage = (event) => {
  if (document.getElementById('content-wrapper')) {
    document.getElementById('content-wrapper').innerHTML = event.data.content
    const url = event.data.url.replace('?partial=true', '')
    if (document.getElementById('browser-page-title')) {
      document.title = document.getElementById('browser-page-title').innerHTML
    }
    history.pushState(null, null, url)
    gtag('config', 'UA-XXXXXX-XX', {'page_path': url})
    Prism.highlightAll()
  }
}
```

The full code of `script.js` is:

```
window.addEventListener('load', () => {
  if (!navigator.serviceWorker) { return }
  const channel = new BroadcastChannel('ws_navigation')

  channel.onmessage = (event) => {
    if (document.getElementById('content-wrapper')) {
      document.getElementById('content-wrapper').innerHTML = event.data.content
      const url = event.data.url.replace('?partial=true', '')
      if (document.getElementById('browser-page-title')) {
        document.title = document.getElementById('browser-page-title').innerHTML
      }
      history.pushState(null, null, url)
      gtag('config', 'UA-1739509-49', {'page_path': url})
      Prism.highlightAll()
    }
  }

  navigator.serviceWorker.register('/sw.js', {
    scope: '/'
  }).then(() => {
    channel.postMessage({
      task: 'fetchPartial',
      url: `${window.location.pathname}?partial=true`
    })
  }).catch((err) => {
    console.log('SW registration failed', err)
  })
  window.onclick = (e) => {
    let node = e.target
    while (node !== undefined && node !== null && node.localName !== 'a') {
      node = node.parentNode
    }
    if (node !== undefined && node !== null) {
      channel.postMessage({
        task: 'fetchPartial',
        url: `${node.href}?partial=true`
      })
      return false
    }
    return true
  }
})
```

and `sw.js`:

```
const cacheName = 'writesoftware-v1'

self.addEventListener('install', (event) => {
  event.waitUntil(caches.open(cacheName).then(cache => cache.addAll([
    '/shell',
    'user/themes/writesoftware/favicon.ico',
    'user/themes/writesoftware/css/style.css',
    'user/themes/writesoftware/js/script.js',
    'user/themes/writesoftware/img/offline.gif',
    'https://fonts.googleapis.com/css?family=Press+Start+2P',
    'https://fonts.googleapis.com/css?family=Inconsolata:400,700',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/themes/prism.min.css',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/prism.min.js',
    'https://cdn.jsdelivr.net/prism/1.6.0/components/prism-jsx.min.js'
  ])))
})

self.addEventListener('fetch', (event) => {
  const requestUrl = new URL(event.request.url)

  if (requestUrl.href.startsWith('https://www.googletagmanager.com') ||
      requestUrl.href.startsWith('https://www.google-analytics.com') ||
      requestUrl.href.startsWith('https://assets.convertkit.com')) {
    // don't cache, and no cors
    event.respondWith(fetch(event.request.url, { mode: 'no-cors' }))
    return
  }

    event.respondWith(caches.match(event.request)
  event.respondWith(caches.match(event.request)
    .then((response) => {
      if (response) { return response }
      if (requestUrl.origin === location.origin) {
        if (requestUrl.pathname.endsWith('?partial=true')) {
          return fetch(requestUrl.pathname)
        } else {
          return caches.match('/shell')
        }

        return fetch(`${event.request.url}?partial=true`)
      }
      return fetch(event.request.url)
    })
    .then(response => caches.open(cacheName).then((cache) => {
      if (response) {
        cache.put(event.request.url, response.clone())
      }
      return response
    }))
    .catch((error) => {
      console.error(error)
    }))
})

const channel = new BroadcastChannel('ws_navigation')
channel.onmessage = (event) => {
  if (event.data.task === 'fetchPartial') {
    caches
      .match(event.data.url)
      .then((response) => {
        if (response) {
          response.text().then((body) => {
            channel.postMessage({ url: event.data.url, content: body })
          })
          return
        }

        fetch(event.data.url).then((fetchResponse) => {
          const fetchResponseClone = fetchResponse.clone()
          fetchResponse.text().then((body) => {
            channel.postMessage({ url: event.data.url, content: body })
          })

          caches.open(cacheName).then((cache) => {
            cache.put(event.data.url, fetchResponseClone)
          })
        })
      })
      .catch((error) => {
        console.error(error)
      })
      })
    }
  }
```

Second approach: network-first, drop the app shell

While the first approach gave us a fully working app, I was a bit skeptical and worried about having a copy of a page cached for too long on the client, so I decided for a network-first approach: when a user loads a page it is fetched from the network first. If the network call fails for some reason, I lookup the page in the cache to see if we got it cached, otherwise I show the user a GIF if it's totally offline, or another GIF if the page does not exist (I can reach it but I got a 404 error).

As soon as we get a page we cache it (not checking if we cached it previously or not, we just store the latest version).

As an experiment I also got rid of the app shell altogether, because in my case I had no intentions of creating an installable app yet, as without an up-to-date Android device I could not really test-drive it and I preferred to avoid throwing out stuff without proper testing.

To do this I just stripped the app shell from the `install` Service Worker event and I relied on Service Workers and the Cache API to just deliver the plain pages of the site, without managing partial updates. I also dropped the `/shell` fetch hijacking when loading a full page, so on the first page load there is no delay, but we still load partials when navigating to other pages later.

I still use `script.js` and `sw.js` to host the code, with `script.js` being the file that initializes the Service Worker, and also intercepts click on the client-side.

Here's `script.js`:

```
const OFFLINE_GIF = '/user/themes/writesoftware/img/offline.gif'

const fetchPartial = (url) => {
  fetch(`${url}?partial=true`)
  .then((response) => {
    response.text().then((body) => {
      if (document.getElementById('content-wrapper')) {
        document.getElementById('content-wrapper').innerHTML = body
        if (document.getElementById('browser-page-title')) {
          document.title = document.getElementById('browser-page-
title').innerHTML
        }
        history.pushState(null, null, url)
        gtag('config', 'UA-XXXXXX-XX', { page_path: url })
        Prism.highlightAll()
      }
    })
  })
  .catch(() => {
    if (document.getElementById('content-wrapper')) {
    document.getElementById('content-wrapper').innerHTML = `<center>
<h2>Offline</h2><img src="${OFFLINE_GIF}" /></center>`
    }
  })
}

window.addEventListener('load', () => {
  if (!navigator.serviceWorker) { return }

  navigator.serviceWorker.register('/sw.js', {
    scope: '/'
  }).then(() => {
    fetchPartial(window.location.pathname)
  }).catch((err) => {
    console.log('SW registration failed', err)
  })

  window.onclick = (e) => {
    let node = e.target
    while (node !== undefined && node !== null && node.localName !== 'a') {
      node = node.parentNode
    }
    if (node !== undefined && node !== null) {
      fetchPartial(node.href)
      return false
    }
    return true
  }
})
```

and here's `sw.js`:

```
const CACHE_NAME = 'writesoftware-v1'
const OFFLINE_GIF = '/user/themes/writesoftware/img/offline.gif'
const PAGENOTFOUND_GIF = '/user/themes/writesoftware/img/pagenotfound.gif'

self.addEventListener('install', (event) => {
  event.waitUntil(caches.open(CACHE_NAME).then(cache => cache.addAll([
    '/user/themes/writesoftware/favicon.ico',
    '/user/themes/writesoftware/css/style.css',
    '/user/themes/writesoftware/js/script.js',
    '/user/themes/writesoftware/img/offline.gif',
    '/user/themes/writesoftware/img/pagenotfound.gif',
    'https://fonts.googleapis.com/css?family=Press+Start+2P',
    'https://fonts.googleapis.com/css?family=Inconsolata:400,700',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/themes/prism.min.css',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/prism.min.js',
    'https://cdn.jsdelivr.net/prism/1.6.0/components/prism-jsx.min.js'
  ])))
})

self.addEventListener('fetch', (event) => {
  if (event.request.method !== 'GET') return
  if (event.request.headers.get('accept').indexOf('text/html') === -1) return

  const requestUrl = new URL(event.request.url)
  let options = {}

  if (requestUrl.href.startsWith('https://www.googletagmanager.com') ||
      requestUrl.href.startsWith('https://www.google-analytics.com') ||
      requestUrl.href.startsWith('https://assets.convertkit.com')) {
    // no cors
    options = { mode: 'no-cors' }
  }

  event.respondWith(fetch(event.request, options)
    .then((response) => {
      if (response.status === 404) {
        return fetch(PAGENOTFOUND_GIF)
      }
      const resClone = response.clone()
      return caches.open(CACHE_NAME).then((cache) => {
        cache.put(event.request.url, response)
        return resClone
      })
    })
    .catch(() => caches.open(CACHE_NAME).then(cache =>
cache.match(event.request.url)
      .then((response) => {
        if (response) {
```

```
        if (response) {
          return response
        }
        return fetch(OFFLINE_GIF)
      })
      .catch(() => fetch(OFFLINE_GIF)))))
})
```

Going simpler: no partials

As an experiment I dropped the click interceptor that fetches partials, and I relied on Service Workers and the Cache API to just deliver the plain pages of the site, without managing partial updates:

`script.js`:

```
window.addEventListener('load', () => {
  if (!navigator.serviceWorker) { return }
  navigator.serviceWorker.register('/sw.js', {
    scope: '/'
  }).catch((err) => {
    console.log('SW registration failed', err)
  })
})
```

`sw.js`:

```
const CACHE_NAME = 'writesoftware-v1'
const OFFLINE_GIF = '/user/themes/writesoftware/img/offline.gif'
const PAGENOTFOUND_GIF = '/user/themes/writesoftware/img/pagenotfound.gif'

self.addEventListener('install', (event) => {
  event.waitUntil(caches.open(CACHE_NAME).then(cache => cache.addAll([
    '/user/themes/writesoftware/favicon.ico',
    '/user/themes/writesoftware/css/style.css',
    '/user/themes/writesoftware/js/script.js',
    '/user/themes/writesoftware/img/offline.gif',
    '/user/themes/writesoftware/img/pagenotfound.gif',
    'https://fonts.googleapis.com/css?family=Press+Start+2P',
    'https://fonts.googleapis.com/css?family=Inconsolata:400,700',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/themes/prism.min.css',
    'https://cdnjs.cloudflare.com/ajax/libs/prism/1.6.0/prism.min.js',
    'https://cdn.jsdelivr.net/prism/1.6.0/components/prism-jsx.min.js'
  ])))
})

self.addEventListener('fetch', (event) => {
  if (event.request.method !== 'GET') return
  if (event.request.headers.get('accept').indexOf('text/html') === -1) return

  const requestUrl = new URL(event.request.url)
  let options = {}

  if (requestUrl.href.startsWith('https://www.googletagmanager.com') ||
      requestUrl.href.startsWith('https://www.google-analytics.com') ||
      requestUrl.href.startsWith('https://assets.convertkit.com')) {
    // no cors
    options = { mode: 'no-cors' }
  }

  event.respondWith(fetch(event.request, options)
    .then((response) => {
      if (response.status === 404) {
        return fetch(PAGENOTFOUND_GIF)
      }
      const resClone = response.clone()
      return caches.open(CACHE_NAME).then((cache) => {
        cache.put(event.request.url, response)
        return resClone
      })
    })
    .catch(() => caches.open(CACHE_NAME).then(cache =>
cache.match(event.request.url)
      .then((response) => {

        return response || fetch(OFFLINE_GIF)
      })
      .catch(() => fetch(OFFLINE_GIF)))))
})
```

I think this is the bare bones example of adding offline capabilities to a website, still keeping things simple. Any kind of website can add such Service Worker without too much complexity.

In the end for me this approach was not enough to be viable, and I ended up implementing the version with fetch partial updates.