**Introduction to progressive web apps**

* [Overview: Progressive web apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)
* [Next](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure)

This article provides an introduction to Progressive Web Apps (PWAs), discussing what they are and the advantages they offer over regular web apps.

[**What is a Progressive Web App?**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#what_is_a_progressive_web_app)

**Note:** The term "Progressive Web App" isn't a formal or official name. It's just a shorthand used initially by Google for the concept of creating a flexible, adaptable app using only web technologies.

PWAs are web apps developed using a number of specific technologies and standard patterns to allow them to take advantage of both web and native app features. For example, web apps are more discoverable than native apps; it's a lot easier and faster to visit a website than to install an application, and you can also share web apps by sending a link.

On the other hand, native apps are better integrated with the operating system and therefore offer a more seamless experience for the users. You can install a native app so that it works offline, and users love tapping their icons to easily access their favorite apps, rather than navigating to it using a browser.

PWAs give us the ability to create web apps that can enjoy these same advantages.

It's not a brand new concept—such ideas have been revisited many times on the web platform with various approaches in the past. Progressive Enhancement and responsive design already allow us to build mobile friendly websites.

PWAs, however, provide all this and more without losing any of the existing features that make the web great.

[**What makes an app a PWA?**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#what_makes_an_app_a_pwa)

As we hinted at above, PWAs are not created with a single technology. They represent a new philosophy for building web apps, involving some specific patterns, APIs, and other features. It's not that obvious if a web app is a PWA or not from first glance. An app could be considered a PWA when it meets certain requirements, or implements a set of given features: works offline, is installable, is easy to synchronize, can send push notifications, etc.

In addition, there are tools to measure how complete (as a percentage) a web app is, such as [Lighthouse](https://developers.google.com/web/tools/lighthouse/). By implementing various technological advantages, we can make an app more progressive, thus ending up with a higher Lighthouse score. But this is only a rough indicator.

There are some key principles a web app should try to observe to be identified as a PWA. It should be:

* [Discoverable](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#discoverability), so the contents can be found through search engines.
* [Installable](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#installability), so it can be available on the device's home screen or app launcher.
* [Linkable](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#linkability), so you can share it by sending a URL.
* [Network independent](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#network_independence), so it works offline or with a poor network connection.
* [Progressively enhanced](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#progressive_enhancement_support), so it's still usable on a basic level on older browsers, but fully-functional on the latest ones.
* [Re-engageable](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#re-engageability), so it's able to send notifications whenever there's new content available.
* [Responsively designed](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#responsiveness), so it's usable on any device with a screen and a browser—mobile phones, tablets, laptops, TVs, refrigerators, etc.
* [Secure](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#secure), so the connections between the user, the app, and your server are secured against any third parties trying to get access to sensitive data.

Offering these features and making use of all the [advantages](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#advantages_of_web_applications) offered by web applications can create a compelling, highly flexible offering for your users and customers.

[**Is it worth doing all that?**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#is_it_worth_doing_all_that)

Absolutely! With a relatively small amount of effort required to implement the core PWA features, the benefits are huge. For example:

* A decrease in loading times after the app has been installed, thanks to caching with [service workers](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API), along with saving precious bandwidth and time. PWAs have near-instantaneous loading (from the second visit).
* The ability to update only the content that has changed when an app update is available. In contrast, with a native app, even the slightest modification can force the user to download the entire application again.
* A look and feel that is more integrated with the native platform—app icons on the home screen or app launcher, applications that automatically run in full screen mode, etc.
* Re-engaging with users through the use of system notifications and push messages, leading to more engaged users and better conversion rates.

It's well worth trying out a PWA approach, so you can see for yourself if it works for your app.

[**Advantages of web applications**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#advantages_of_web_applications)

A fully-capable progressive web application should provide all of the following advantages to the user.

[**Discoverability**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#discoverability)

The eventual aim is that web apps should have better representation in search engines, be easier to expose, catalog and rank, and have metadata usable by browsers to give them special capabilities.

Some of the capabilities have already been enabled on certain web-based platforms by proprietary technologies like [Open Graph](https://ogp.me/), which provides a format for specifying similar metadata in the [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) [<head>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/head) block using [<meta>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta) tags.

The relevant web standard here is the [Web app manifest](https://developer.mozilla.org/en-US/docs/Web/Manifest), which defines features of an app such as name, icon, splash screen, and theme colors in a [JSON](https://developer.mozilla.org/en-US/docs/Glossary/JSON)-formatted manifest file. This is for use in contexts such as app listings and device home screens.

[**Installability**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#installability)

A core part of the web app experience is for users to have app icons on their home screen, and be able to tap to open apps into their own native container that feels nicely integrated with the underlying platform.

Modern web apps can have this native app feel via properties set inside the Web app manifest, and via a feature available in modern smartphone browsers called [web app installation](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Installing).

[**Linkability**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#linkability)

One of the most powerful features of the web is the ability to link to an app at a specific URL without the need for an app store or complex installation process. This is how it has always been.

[**Network independence**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#network_independence)

Modern web apps can work when the network is unreliable, or even non-existent. The basic ideas behind network independence are to be able to:

* Revisit a site and get its contents even if no network is available.
* Browse any kind of content the user has previously visited at least once, even under situations of poor connectivity.
* Control what is shown to the user in situations where there is no connectivity.

This is achieved using a combination of technologies: [Service Workers](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API) to control page requests (for example storing them offline), the [Cache API](https://developer.mozilla.org/en-US/docs/Web/API/Cache) for storing responses to network requests offline (very useful for storing site assets), and client-side data storage technologies such as [Web Storage](https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API) and [IndexedDB](https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API) to store application data offline.

[**Progressive enhancement support**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#progressive_enhancement_support)

Modern web apps can be developed to provide an excellent experience to fully capable browsers, and an acceptable (although not quite as shiny) experience to less capable browsers. We've been doing this for years with best practices such as progressive enhancement. By using [progressive enhancement](https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement), PWAs are cross-browser. This means developers should take into account the differences in implementation of some PWA features and technologies between different browser implementations.

[**Re-engageability**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#re-engageability)

One major advantage of native platforms is the ease with which users can be re-engaged by updates and new content, even when they aren't looking at the app or using their devices. Modern web apps can now do this too, using new technologies such as Service Workers for controlling pages, the [Web Push API](https://developer.mozilla.org/en-US/docs/Web/API/Push_API) for sending updates straight from server to app via a service worker, and the [Notifications API](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API) for generating system notifications to help engage users when they're not actively using their web browser.

[**Responsiveness**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#responsiveness)

Responsive web apps use technologies like [media queries](https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries) and [viewport](https://developer.mozilla.org/en-US/docs/Glossary/Viewport) to make sure that their UIs will fit any form factor: desktop, mobile, tablet, or whatever comes next.

[**Secure**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#secure)

The web platform provides a secure delivery mechanism that prevents snooping while simultaneously ensuring that content hasn’t been tampered with, as long as you take advantage of [HTTPS](https://developer.mozilla.org/en-US/docs/Glossary/https) and develop your apps with security in mind.

It's also easy for users to ensure that they're installing the right app, because its URL will match your site's domain. This is very different from apps in app stores, which may have a number of similarly-named apps, some of which may even be based on your own site, which only adds to the confusion. Web apps eliminate that confusion and ensure that users get the best possible experience.

[**Browser support**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#browser_support)

As mentioned before, PWAs don't depend on a single API, but rather using various technologies to achieve the goal of delivering the best web experience possible.

The key ingredient required for PWAs is [service worker](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API) support. Thankfully service workers are [now supported on all major browsers](https://jakearchibald.github.io/isserviceworkerready/) on desktop and mobile.

Other features such as [Web App Manifest](https://developer.mozilla.org/en-US/docs/Web/Manifest), [Push](https://developer.mozilla.org/en-US/docs/Web/API/Push_API) [Notifications](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API), and [Add to Home Screen](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen) functionality have wide support too. Currently, Safari has limited support for Web App Manifest and Add to Home Screen and no support for web push notifications. However, other major browsers support all these features.

Above all you should follow the progressive enhancement rule: use technologies that enhance the appearance and utility of your app when they're available, but still offer the basic functionality of your app when those features are unavailable. Presenting a trusted website with a good performance is a consequence of using these enhancements; this in turn means building web apps which follow better practices. This way everybody will be able to use the app, but those with modern browsers will benefit from PWA features even more.

[**An example application**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction#an_example_application)

In this series of articles we will examine the source code of a super simple website that lists information about games submitted to the [A-Frame category](https://js13kgames.com/aframe) in the [js13kGames 2017](https://2017.js13kgames.com/) competition. You don't have to think about what the actual content on the website is; the main point is to learn how to use PWA features in your own projects.

You can [see this app in action](https://mdn.github.io/pwa-examples/js13kpwa/) online, and the source code is [available on GitHub](https://github.com/mdn/pwa-examples/tree/master/js13kpwa). We'll be examining this code carefully over the course of this series of articles.

For now, let's move to the second part of this series, where we’ll be looking at the structure of our example app.

**Progressive web app structure**

* [Previous](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Introduction)
* [Overview: Progressive web apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)
* [Next](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers)

Now that we know the theory behind PWAs, let's look at the recommended structure of an actual app. We will start with analyzing the [js13kPWA](https://mdn.github.io/pwa-examples/js13kpwa/) application, why it is built that way, and what benefits it brings.

[**Architecture of an app**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#architecture_of_an_app)

There are two main, different approaches to rendering a website — on the server or on the client. They both have their advantages and disadvantages, and you can mix the two approaches to some degree.

* Server-side rendering (SSR) means a website is rendered on the server, so it offers quicker first load, but navigating between pages requires downloading new HTML content. It works great across browsers, but it suffers in terms of time navigating between pages and therefore general perceived performance — loading a page requires a new round trip to the server.
* Client-side rendering (CSR) allows the website to be updated in the browser almost instantly when navigating to different pages, but requires more of an initial download hit and extra rendering on the client at the beginning. The website is slower on an initial visit, but can be faster to navigate.

Mixing SSR with CSR can lead to the best results — you can render a website on the server, cache its contents, and then update the rendering on the client-side as and when needed. The first page load is quick because of the SSR, and the navigation between pages is smooth because the client can re-render the page with only the parts that have changed.

PWAs can be built using any approach you like, but some will work better than the others. The most popular approach is the "app shell" concept, which mixes SSR and CSR in exactly the way described above, and in addition follows the "offline first" methodology which we will explain in detail in upcoming articles and use in our example application. There is also a new approach involving the [Streams API](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API), which we'll mention briefly.

[**App shell**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#app_shell)

The App shell concept is concerned with loading a minimal user interface as soon as possible and then caching it so it is available offline for subsequent visits before then loading all the contents of the app. That way, the next time someone visits the app from the device, the UI loads from the cache immediately and any new content is requested from the server (if it isn’t available in the cache already).

This structure is fast, and also feels fast as the user sees "something" instantly, instead of a loading spinner or a blank page. It also allows the website to be accessible offline if the network connection is not available.

We can control what is requested from the server and what is retrieved from the cache with a [service worker](https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API), which will be explained in detail in the next article — for now let's focus on the structure itself.

[**Why should I use it?**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#why_should_i_use_it)

This architecture allows a website to benefit the most from all the PWA features — it caches the app shell and manages the dynamic content in a way that greatly improves the performance. In addition to the basic shell, you can add other features such as [add to home screen](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen) or [push notifications](https://developer.mozilla.org/en-US/docs/Web/API/Push_API), safe in the knowledge that the app will still work OK if they are not supported by the user's browser — this is the beauty of progressive enhancement.

The website feels like a native app with instant interaction and solid performance while keeping all the benefits of the web.

[**Being linkable, progressive and responsive by design**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#being_linkable_progressive_and_responsive_by_design)

It's important to remember the PWA advantages and keep them in mind when designing the application. The app shell approach allows websites to be:

* Linkable: Even though it behaves like a native app, it is still a website — you can click on the links within the page and send a URL to someone if you want to share it.
* Progressive: Start with the "good, old basic website” and progressively add new features while remembering to detect if they are available in the browser and gracefully handle any errors that crop up if support is not available. For example, an offline mode with the help of service workers is just an extra trait that makes the website experience better, but it's still perfectly usable without it.
* Responsive: Responsive web design also applies to progressive web apps, as both are mainly for mobile devices. There are so many varied devices with browsers — it's important to prepare your website so it works on different screen sizes, viewports or pixel densities, using technologies like [viewport meta tag](https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport_meta_tag), [CSS media queries](https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries), [Flexbox](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout), and [CSS Grid](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout).

[**Different concept: streams**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#different_concept_streams)

An entirely different approach to server- or client-side rendering can be achieved with the [Streams API](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API). With a little help from service workers, streams can greatly improve the way we parse content.

The app shell model requires all the resources to be available before the website can start rendering. It's different with HTML, as the browser is actually streaming the data already and you can see when the elements are loaded and rendered on the website. To have the JavaScript "operational", however, it has to be downloaded in its entirety.

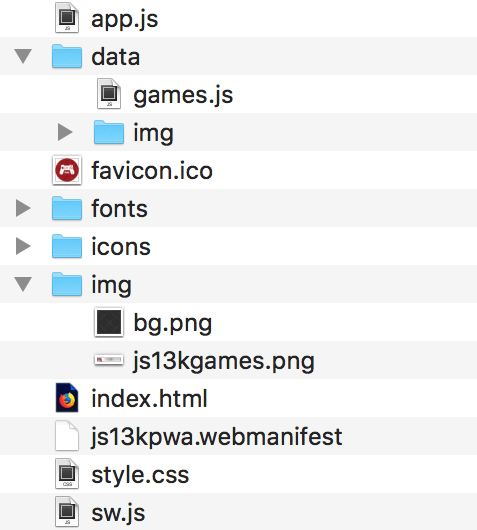
The Streams API allows developers to have direct access to data streaming from the server — if you want to perform an operation on the data (for example, adding a filter to a video), you no longer need to wait for all of it to be downloaded and converted to a blob (or whatever) — you can start right away. It provides fine-grained control — the stream can be started, chained with another stream, cancelled, checked for errors, and more.

In theory, streaming is a better model, but it's also more complex, and at the time of writing (March 2018) the Streams API is still a work-in-progress and not yet fully available in any of the major browsers. When it is available, it will be the fastest way of serving the content — the benefits are going to be huge in terms of performance.

For working examples and more information, see the [Streams API documentation](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API).

[**Structure of our example application**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#structure_of_our_example_application)

The [js13kPWA](https://mdn.github.io/pwa-examples/js13kpwa/) website structure is quite simple: it consists of a single HTML file ([index.html](https://github.com/mdn/pwa-examples/blob/master/js13kpwa/index.html)) with basic CSS styling ([style.css](https://github.com/mdn/pwa-examples/blob/master/js13kpwa/style.css)), and a few images, scripts, and fonts. The folder structure looks like this:



[**The HTML**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#the_html)

From the HTML point of view, the app shell is everything outside the content section:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>js13kGames A-Frame entries</title>

<meta name="description" content="A list of A-Frame entries submitted to the js13kGames 2017 competition, used as an example for the MDN articles about Progressive Web Apps.">

<meta name="author" content="end3r">

<meta name="theme-color" content="#B12A34">

<meta name="viewport" content="width=device-width, initial-scale=1">

<meta property="og:image" content="icons/icon-512.png">

<link rel="icon" href="favicon.ico">

<link rel="stylesheet" href="style.css">

<link rel="manifest" href="js13kpwa.webmanifest">

<script src="data/games.js" defer></script>

<script src="app.js" defer></script>

</head>

<body>

<header>

<p><a class="logo" href="http://js13kgames.com"><img src="img/js13kgames.png" alt="js13kGames"></a></p>

</header>

<main>

<h1>js13kGames A-Frame entries</h1>

<p class="description">List of games submitted to the <a href="http://js13kgames.com/aframe">A-Frame category</a> in the <a href="http://2017.js13kgames.com">js13kGames 2017</a> competition. You can <a href="https://github.com/mdn/pwa-examples/blob/master/js13kpwa">fork js13kPWA on GitHub</a> to check its source code.</p>

<button id="notifications">Request dummy notifications</button>

<section id="content">

// Content inserted in here

</section>

</main>

<footer>

<p>© js13kGames 2012-2018, created and maintained by <a href="http://end3r.com">Andrzej Mazur</a> from <a href="http://enclavegames.com">Enclave Games</a>.</p>

</footer>

</body>

</html>

The [<head>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/head) section contains some basic info like title, description and links to CSS, web manifest, games content JS file, and app.js — that's where our JavaScript application is initialized. The [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) is split into the [<header>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header) (containing linked image), [<main>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main) page (with title, description and place for a content), and [<footer>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/footer) (copy and links).

The app's only job is to list all the A-Frame entries from the js13kGames 2017 competition. As you can see it is a very ordinary, one page website — the point is to have something simple so we can focus on the implementation of the actual PWA features.

[**The CSS**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#the_css)

The CSS is also as plain as possible: it uses [@font-face](https://developer.mozilla.org/en-US/docs/Web/CSS/@font-face) to load and use a custom font, and it applies some simple styling of the HTML elements. The overall approach is to have the design look good on both mobile (with a responsive web design approach) and desktop devices.

[**The main app JavaScript**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#the_main_app_javascript)

The app.js file does a few things we will look into closely in the next articles. First of all it generates the content based on this template:

const template = `<article>

<img src='data/img/placeholder.png' data-src='data/img/SLUG.jpg' alt='NAME'>

<h3>#POS. NAME</h3>

<ul>

<li><span>Author:</span> <strong>AUTHOR</strong></li>

<li><span>Twitter:</span> <a href='https://twitter.com/TWITTER'>@TWITTER</a></li>

<li><span>Website:</span> <a href='http://WEBSITE/'>WEBSITE</a></li>

<li><span>GitHub:</span> <a href='https://GITHUB'>GITHUB</a></li>

<li><span>More:</span> <a href='http://js13kgames.com/entries/SLUG'>js13kgames.com/entries/SLUG</a></li>

</ul>

</article>`;

let content = '';

for (let i = 0; i < games.length; i++) {

let entry = template.replace(/POS/g, (i + 1))

.replace(/SLUG/g, games[i].slug)

.replace(/NAME/g, games[i].name)

.replace(/AUTHOR/g, games[i].author)

.replace(/TWITTER/g, games[i].twitter)

.replace(/WEBSITE/g, games[i].website)

.replace(/GITHUB/g, games[i].github);

entry = entry.replace('<a href=\'http:///\'></a>', '-');

content += entry;

}

document.getElementById('content').innerHTML = content;

Next, it registers a service worker:

if('serviceWorker' in navigator) {

navigator.serviceWorker.register('/pwa-examples/js13kpwa/sw.js');

};

The next code block requests permission for notifications when a button is clicked:

const button = document.getElementById('notifications');

button.addEventListener('click', () => {

Notification.requestPermission().then((result) => {

if (result === 'granted') {

randomNotification();

}

});

});

The last block creates notifications that display a randomly-selected item from the games list:

function randomNotification() {

const randomItem = Math.floor(Math.random() \* games.length);

const notifTitle = games[randomItem].name;

const notifBody = `Created by ${games[randomItem].author}.`;

const notifImg = `data/img/${games[randomItem].slug}.jpg`;

const options = {

body: notifBody,

icon: notifImg,

};

new Notification(notifTitle, options);

setTimeout(randomNotification, 30000);

}

[**The service worker**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#the_service_worker)

The last file we will quickly look at is the service worker: sw.js — it first imports data from the games.js file:

self.importScripts('data/games.js');

Next, it creates a list of all the files to be cached, both from the app shell and the content:

const cacheName = 'js13kPWA-v1';

const appShellFiles = [

'/pwa-examples/js13kpwa/',

'/pwa-examples/js13kpwa/index.html',

'/pwa-examples/js13kpwa/app.js',

'/pwa-examples/js13kpwa/style.css',

'/pwa-examples/js13kpwa/fonts/graduate.eot',

'/pwa-examples/js13kpwa/fonts/graduate.ttf',

'/pwa-examples/js13kpwa/fonts/graduate.woff',

'/pwa-examples/js13kpwa/favicon.ico',

'/pwa-examples/js13kpwa/img/js13kgames.png',

'/pwa-examples/js13kpwa/img/bg.png',

'/pwa-examples/js13kpwa/icons/icon-32.png',

'/pwa-examples/js13kpwa/icons/icon-64.png',

'/pwa-examples/js13kpwa/icons/icon-96.png',

'/pwa-examples/js13kpwa/icons/icon-128.png',

'/pwa-examples/js13kpwa/icons/icon-168.png',

'/pwa-examples/js13kpwa/icons/icon-192.png',

'/pwa-examples/js13kpwa/icons/icon-256.png',

'/pwa-examples/js13kpwa/icons/icon-512.png',

];

const gamesImages = [];

for (let i = 0; i < games.length; i++) {

gamesImages.push(`data/img/${games[i].slug}.jpg`);

}

const contentToCache = appShellFiles.concat(gamesImages);

The next block installs the service worker, which then actually caches all the files contained in the above list:

self.addEventListener('install', (e) => {

console.log('[Service Worker] Install');

e.waitUntil((async () => {

const cache = await caches.open(cacheName);

console.log('[Service Worker] Caching all: app shell and content');

await cache.addAll(contentToCache);

})());

});

Last of all, the service worker fetches content from the cache if it is available there, providing offline functionality:

self.addEventListener('fetch', (e) => {

e.respondWith((async () => {

const r = await caches.match(e.request);

console.log(`[Service Worker] Fetching resource: ${e.request.url}`);

if (r) { return r; }

const response = await fetch(e.request);

const cache = await caches.open(cacheName);

console.log(`[Service Worker] Caching new resource: ${e.request.url}`);

cache.put(e.request, response.clone());

return response;

})());

});

[**The JavaScript data**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#the_javascript_data)

The games data is present in the data folder in a form of a JavaScript object ([games.js](https://github.com/mdn/pwa-examples/blob/master/js13kpwa/data/games.js)):

var games = [

{

slug: 'lost-in-cyberspace',

name: 'Lost in Cyberspace',

author: 'Zosia and Bartek',

twitter: 'bartaz',

website: '',

github: 'github.com/bartaz/lost-in-cyberspace'

},

{

slug: 'vernissage',

name: 'Vernissage',

author: 'Platane',

twitter: 'platane\_',

website: 'github.com/Platane',

github: 'github.com/Platane/js13k-2017'

},

// ...

{

slug: 'emma-3d',

name: 'Emma-3D',

author: 'Prateek Roushan',

twitter: '',

website: '',

github: 'github.com/coderprateek/Emma-3D'

}

];

Every entry has its own image in the data/img folder. This is our content, loaded into the content section with JavaScript.

[**Next up**](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/App_structure#next_up)

In the next article we will look in more detail at how the app shell and the content are cached for offline use with the help from the service worker.