API

An API (Application Programming Interface) is a set of features and rules that exist inside a software program (the application) enabling interaction with it through software - as opposed to a human user interface. The API can be seen as a simple contract (the interface) between the application offering it and other items, such as third party software or hardware.
In Web development, an API is generally a set of code features (e.g. `methods`, `properties`, `events`, and `URLs`) that a developer can use in their apps for interacting with components of a user's web browser, or other software/hardware on the user's computer, or third party websites and services.

For example:

- The `getUserMedia API` can be used to grab audio and video from a user's webcam, which can then be used in any way the developer likes, for example, recording video and audio, broadcasting it to another user in a conference call, or capturing image stills from the video.
- The `Geolocation API` can be used to retrieve location information from whatever service the user has available on their device (e.g. GPS), which can then be used in conjunction with the Google Maps APIs to for example plot the user's location on a custom map and show them what tourist attractions are in their area.
- The `Twitter APIs` can be used to retrieve data from a user's twitter accounts, for example, to display their latest tweets on a web page.
- The `Web Animations API` can be used to animate parts of a web page — for example, to make images move around or rotate.

Web APIs

When writing code for the Web, there are a large number of Web APIs available. Below is a list of all the APIs and interfaces (object types) that you may be able to use while developing your Web app or site :

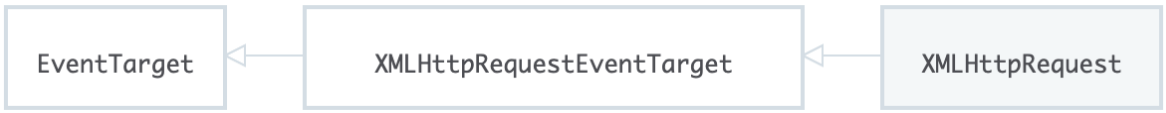
https://developer.mozilla.org/en-US/docs/Web/API

Web APIs are typically used with JavaScript, although this doesn't always have to be the case.

XMLHttpRequest

`XMLHttpRequest` (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.

`XMLHttpRequest` is used heavily in `AJAX` programming.



Despite its name, `XMLHttpRequest` can be used to retrieve any type of data, not just XML. If your communication needs to involve receiving event data or message data from a server, consider using `server-sent events` through the `EventSource` interface. For full-duplex communication, `WebSockets` may be a better choice.

Origin

Web content's **origin** is defined by the scheme (protocol), hostname (domain), and port of the `URL` used to access it. Two objects have the same origin only when the scheme, hostname, and port all match.
Some operations are restricted to same-origin content, and this restriction can be lifted using `CORS`.

Examples
These are same origin because they have the same scheme (`http`) and hostname (`example.com`), and the different file path does not matter:

- `http://example.com/app1/index.html`
- `http://example.com/app2/index.html`

These are same origin because a server delivers HTTP content through port 80 by default:

- `http://Example.com:80`
- `http://example.com`

These are not same origin because they use different schemes:

- `http://example.com/app1`
- `https://example.com/app2`

These are not same origin because they use different hostnames:

- `http://example.com`
- `http://www.example.com`
- `http://myapp.example.com`

These are not same origin because they use different ports:

- `http://example.com`
- `http://example.com:8080`

Web Workers API

Web Workers makes it possible to run a script operation in a background thread separate from the main execution thread of a web application. The advantage of this is that laborious processing can be performed in a separate thread, allowing the main (usually the UI) thread to run without being blocked/slowed down.

Note: Web Workers can also use the Web Worker API (i.e. workers can spawn workers, provided they are hosted within the same origin as the parent page).

Web Workers concepts and usage
A worker is an object created using a constructor (e.g. `Worker()`) that runs a named JavaScript file — this file contains the code that will run in the worker thread.

In addition to the standard JavaScript set of functions (such as `String`, `Array`, `Object`, `JSON`, etc.), you can run almost any code you like inside a worker thread. There are some exceptions: for example, you can't directly manipulate the DOM from inside a worker, or use some default methods and properties of the `window` object.

Data is sent between workers and the main thread via a system of messages — both sides send their messages using the `postMessage()` method, and respond to messages via the `onmessage` event handler (the message is contained within the Message event's data property). The data is copied rather than shared.

Workers may in turn spawn new workers, as long as those workers are hosted within the same origin as the parent page. In addition, workers may use `XMLHttpRequest` for network I/O, with the exception that the `responseXML` and `channel` attributes on XMLHttpRequest always return `null`.

Worker types
There are a number of different types of workers:

- Dedicated workers are workers that are utilized by a single script. This context is represented by a `DedicatedWorkerGlobalScope` object.

- `Shared workers` are workers that can be utilized by multiple scripts running in different windows, IFrames, etc., as long as they are in the same domain as the worker. They are a little more complex than dedicated workers — scripts must communicate via an active port.

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

Worker global contexts and functions
Workers run in a different global context than the current `window`! While `Window` is not directly available to workers, many of the same methods are defined in a shared mixin (`WindowOrWorkerGlobalScope`), and made available to workers through their own `WorkerGlobalScope`-derived contexts:

- `DedicatedWorkerGlobalScope` for dedicated workers
- `SharedWorkerGlobalScope` for shared workers
- `ServiceWorkerGlobalScope` for service workers

Some of the functions (a subset) that are common to all workers and to the main thread (from `WindowOrWorkerGlobalScope`) are: `atob()`, `btoa()`, `clearInterval()`, `clearTimeout()`,`dump()` , `setInterval()` , `setTimeout()`.
The following functions are only available to workers:
`WorkerGlobalScope.importScripts()` (all workers),
`DedicatedWorkerGlobalScope.postMessage` (dedicated workers only).

Worker

The `Worker` interface of the `Web Workers API` represents a background task that can be created via script, which can send messages back to its creator.
Creating a worker is done by calling the `Worker("path/to/worker/script")` constructor.
Workers may themselves spawn new workers, as long as those workers are hosted at the same origin as the parent page.

Note: nested workers are not yet implemented in WebKit.

Not all interfaces and functions are available to scripts inside a Worker. Workers may use `XMLHttpRequest` for network communication, but its `responseXML` and `channel` attributes are always `null`. (`fetch` is also available, with no such restrictions.)

Service Worker API

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

Service worker concepts and usage
A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web-page/site that it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available).

A service worker is run in a worker context: it therefore has no DOM access, and runs on a different thread to the main JavaScript that powers your app, so it is non-blocking. It is designed to be fully async; as a consequence, APIs such as synchronous `XHR` and `Web Storage` can't be used inside a service worker.

Service workers only run over HTTPS, for security reasons. Having modified network requests, wide open to man in the middle attacks would be really bad. In Firefox, Service Worker APIs are also hidden and cannot be used when the user is in `private browsing mode`.

Note: Service workers make heavy use of promises, as generally they will wait for responses to come through, after which they will respond with a success or failure action. The promises architecture is ideal for this.

Registration
A service worker is first registered using the `ServiceWorkerContainer.register()` method. If successful, your service worker will be downloaded to the client and attempt installation/activation for URLs accessed by the user inside the whole origin, or inside a subset specified by you.

Download, install and activate

At this point, your service worker will observe the following lifecycle:

- Download
- Install
- Activate

The service worker is immediately downloaded when a user first accesses a service worker–controlled site/page. After that, it is updated when:

- A navigation to an in-scope page occurs.
- An event is fired on the service worker and it hasn't been downloaded in the last 24 hours.

Installation is attempted when the downloaded file is found to be new — either different to an existing service worker (byte-wise compared), or the first service worker encountered for this page/site.

If this is the first time a service worker has been made available, installation is attempted, then after a successful installation, it is activated.

If there is an existing service worker available, the new version is installed in the background, but not yet activated — at this point it is called the worker in waiting. It is only activated when there are no longer any pages loaded that are still using the old service worker. As soon as there are no more pages to be loaded, the new service worker activates (becoming the active worker). Activation can happen sooner using `ServiceWorkerGlobalScope.skipWaiting()` and existing pages can be claimed by the active worker using `Clients.claim()`.

You can listen for the `install` event; a standard action is to prepare your service worker for usage when this fires, for example by creating a cache using the built in storage API, and placing assets inside it that you'll want for running your app offline.

There is also an `activate` event. The point where this event fires is generally a good time to clean up old caches and other things associated with the previous version of your service worker.

Your service worker can respond to requests using the `FetchEvent` event. You can modify the response to these requests in any way you want, using the `FetchEvent.respondWith()` method.

Using Service Workers

This article provides information on getting started with service workers, including basic architecture, registering a service worker, the install and activation process for a new service worker, updating your service worker, cache control and custom responses, all in the context of a simple app with offline functionality.

The premise of service workers
One overriding problem that web users have suffered with for years is loss of connectivity. The best web app in the world will provide a terrible user experience if you can’t download it. There have been various attempts to create technologies to solve this problem, and some of the issues have been solved. But the overriding problem is that there still isn’t a good overall control mechanism for asset caching and custom network requests.

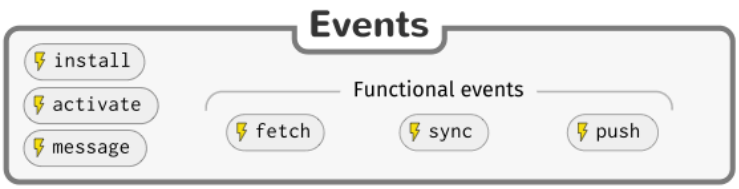
The previous attempt, `AppCache`, seemed to be a good idea because it allowed you to specify assets to cache really easily. However, it made many assumptions about what you were trying to do and then broke horribly when your app didn't follow those assumptions exactly.

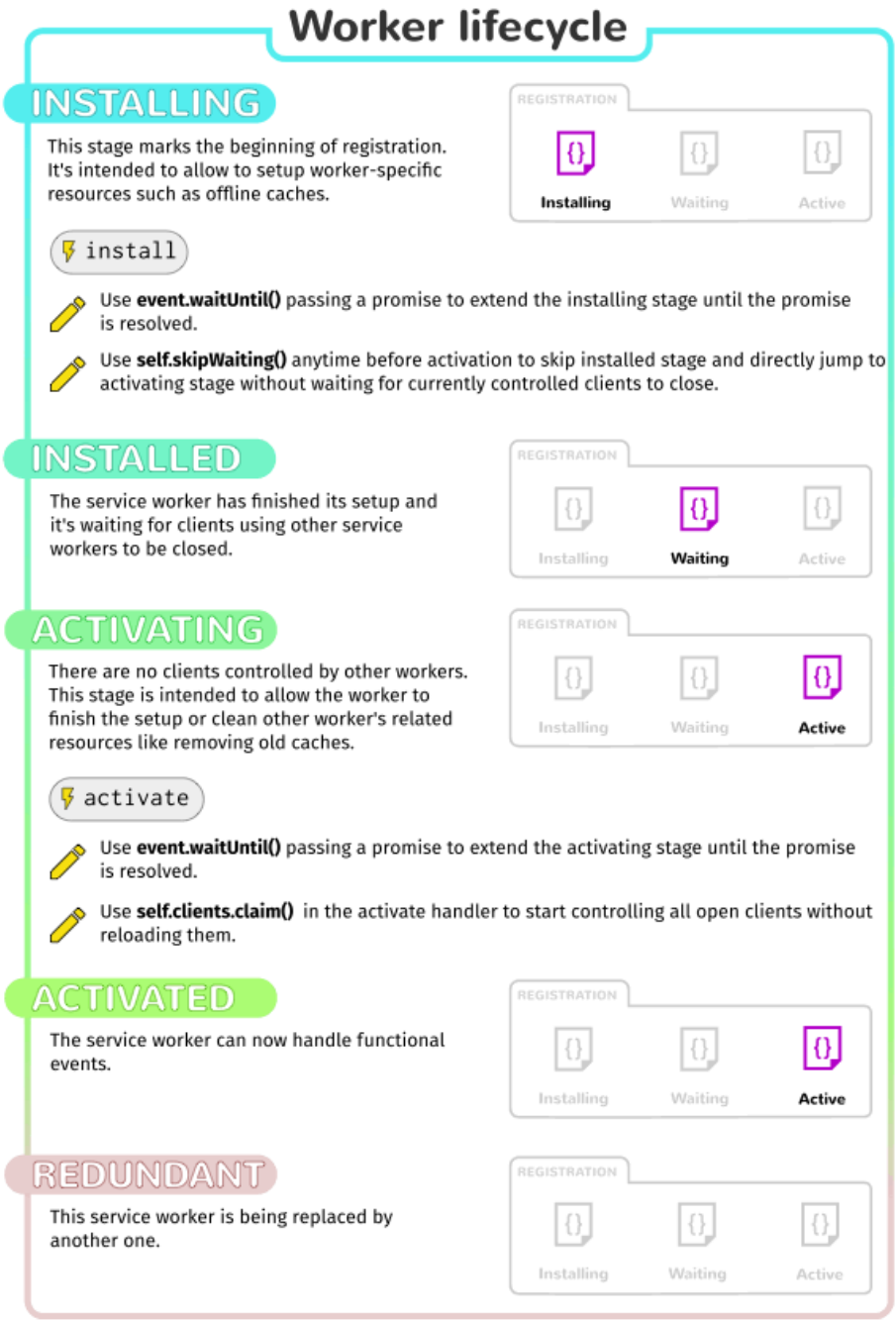
Service workers should finally fix these issues. Service worker syntax is more complex than that of AppCache, but the trade off is that you can use JavaScript to control your AppCache-implied behaviors with a fine degree of granularity, allowing you to handle this problem and many more. Using a Service worker you can easily set an app up to use cached assets first, thus providing a default experience even when offline, before then getting more data from the network (commonly known as `Offline First`). This is already available with native apps, which is one of the main reasons native apps are often chosen over web apps.

Setting up to play with service workers
These days, service workers are enabled by default in all modern browsers. To run code using service workers, you'll need to serve your code via HTTPS — Service workers are restricted to running across HTTPS for security reasons. GitHub is therefore a good place to host experiments, as it supports HTTPS. In order to facilitate local development, `localhost` is considered a secure origin by browsers as well.

Basic architecture
With service workers, the following steps are generally observed for basic set up:

- The service worker URL is fetched and registered via `serviceWorkerContainer.register()`.
- If successful, the service worker is executed in a `ServiceWorkerGlobalScope`; this is basically a special kind of worker context, running off the main script execution thread, with no DOM access.
- The service worker is now ready to process events.
- Installation of the worker is attempted when service worker-controlled pages are accessed subsequently. An Install event is always the first one sent to a service worker (this can be used to start the process of populating an IndexedDB, and caching site assets). This is really the same kind of procedure as installing a native or Firefox OS app — making everything available for use offline.
- When the `oninstall` handler completes, the service worker is considered installed.
- Next is activation. When the service worker is installed, it then receives an activate event. The primary use of `onactivate` is for cleanup of resources used in previous versions of a Service worker script.
- The Service worker will now control pages, but only those opened after the `register()` is successful. i.e. a document starts life with or without a Service worker and maintains that for its lifetime. So documents will have to be reloaded to actually be controlled.





Promises
`Promises` are a great mechanism for running async operations, with success dependant on one another. This is central to the way service workers work.
Promises can do a variety of things, but all you need to know for now is that if something returns a promise, you can attach `.then()` to the end and include callbacks inside it for success cases, or you can insert `.catch()` on the end if you want to include a failure callback.

Let's compare a traditional synchronous callback structure to its asynchronous promise equivalent.

sync

```
try {
  const value = myFunction();
  console.log(value);
} catch(err) {
  console.log(err);
}
```

async

```
myFunction().then((value) => {
  console.log(value);
}).catch((err) => {
  console.log(err);
});
```

In the first example, we have to wait for `myFunction()` to run and return `value` before any more of the code can execute. In the second example, myFunction() returns a promise for `value`, then the rest of the code can carry on running. When the promise resolves, the code inside `then` will be run, asynchronously.

Now for a real example — what if we wanted to load images dynamically, but we wanted to make sure the images were loaded before we tried to display them? This is a standard thing to do, but it can be a bit of a pain. We can use `.onload` to only display the image after it's loaded, but what about events that start happening before we start listening to them? We could try to work around this using `.complete`, but it's still not foolproof, and what about multiple images? And, ummm, it's still synchronous, so blocks the main thread.
Instead, we could build our own promise to handle this kind of case.

Note: A real service worker implementation would use caching and onfetch rather than the XMLHttpRequest API. Those features are not used here so that you can focus on understanding Promises.

```
const imgLoad = (url) => {
  return new Promise((resolve, reject) => {
    var request = new XMLHttpRequest();
    request.open('GET', url);
    request.responseType = 'blob';

    request.onload = () => {
      if (request.status == 200) {
        resolve(request.response);
      } else {
        reject(Error('Image didn\'t load successfully; error code:' + request.statusText));
      }
    };

    request.onerror = () => {
      reject(Error('There was a network error.'));
    };

    request.send();
  });
}
```

We return a new promise using the `Promise()` constructor, which takes as an argument a callback function with `resolve` and `reject` parameters. Somewhere in the function, we need to define what happens for the promise to resolve successfully or be rejected — in this case return a 200 OK status or not — and then call `resolve` on success, or `reject` on failure. The rest of the contents of this function is fairly standard XHR stuff, so we won't worry about that for now.

When we come to call the `imgLoad()` function, we call it with the url to the image we want to load, as we might expect, but the rest of the code is a little different:

```
let body = document.querySelector('body');
let myImage = new Image();

imgLoad('myLittleVader.jpg').then((response) => {
  var imageURL = window.URL.createObjectURL(response);
  myImage.src = imageURL;
  body.appendChild(myImage);
}, (Error) => {
  console.log(Error);
});
```

On to the end of the function call, we chain the promise then() method, which contains two functions — the first one is executed when the promise successfully resolves, and the second is called when the promise is rejected. In the resolved case, we display the image inside myImage and append it to the body (its argument is the request.response contained inside the promise's resolve method); in the rejected case we return an error to the console.

This all happens asynchronously.

Note: You can also chain promise calls together, for example: myPromise().then(success, failure).then(success).catch(failure);

Service workers demo
To demonstrate just the very basics of registering and installing a service worker, we have created a simple demo called sw-test, which is a simple Star wars Lego image gallery. It uses a promise-powered function to read image data from a JSON object and load the images using Ajax, before displaying the images in a line down the page. We've kept things static and simple for now. It also registers, installs, and activates a service worker, and when more of the spec is supported by browsers it will cache all the files required so it will work offline!

Registering your worker

The first block of code in our app's JavaScript file — app.js — is as follows. This is our entry point into using service workers.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('./sw-test/sw.js', {scope: './sw-test/'})
  .then((reg) => {
    // registration worked
    console.log('Registration succeeded. Scope is ' + reg.scope);
  }).catch((error) => {
    // registration failed
    console.log('Registration failed with ' + error);
  });
}
```

1. The outer block performs a feature detection test to make sure service workers are supported before trying to register one.

2. Next, we use the ServiceWorkerContainer.register() function to register the service worker for this site, which is just a JavaScript file residing inside our app (note this is the file's URL relative to the origin, not the JS file that references it.)

3. The scope parameter is optional, and can be used to specify the subset of your content that you want the service worker to control. In this case, we have specified '/sw-test/', which means all content under the app's origin. If you leave it out, it will default to this value anyway, but we specified it here for illustration purposes.

4. The .then() promise function is used to chain a success case onto our promise structure. When the promise resolves successfully, the code inside it executes.

5. Finally, we chain a .catch() function onto the end that will run if the promise is rejected.

A single service worker can control many pages. Each time a page within your scope is loaded, the service worker is installed against that page and operates on it. Bear in mind therefore that you need to be careful with global variables in the service worker script: each page doesn't get its own unique worker.

Note: Your service worker functions like a proxy server, allowing you to modify requests and responses, replace them with items from its own cache, and more.

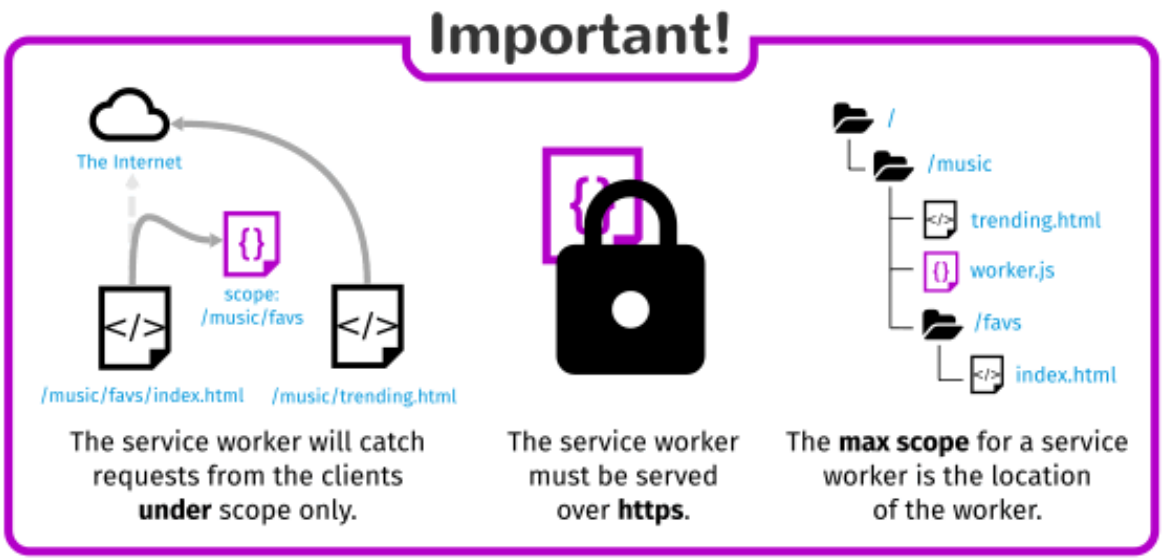
Note: One great thing about service workers is that if you use feature detection like we've shown above, browsers that don't support service workers can just use your app online in the normal expected fashion.

Why is my service worker failing to register?
This could be for the following reasons:

- You are not running your application through HTTPS.

- The path to your service worker file is not written correctly — it needs to be written relative to the origin, not your app's root directory. In our example, the worker is at https://mdn.github.io/sw-test/sw.js, and the app's root is https://mdn.github.io/sw-test/. But the path needs to be written as /sw-test/sw.js, not /sw.js.

- It is also not allowed to point to a service worker of a different origin than that of your app.



Also note:

- The service worker will only catch requests from clients under the service worker's scope.
- The max scope for a service worker is the location of the worker.
- If your service worker is active on a client being served with the `Service-Worker-Allowed` header, you can specify a list of max scopes for that worker.
- In Firefox, Service Worker APIs are hidden and cannot be used when the user is in private browsing mode.

After your service worker is registered, the browser will attempt to install then activate the service worker for your page/site.

The install event is fired when an install is successfully completed. The install event is generally used to populate your browser's offline caching capabilities with the assets you need to run your app offline. To do this, we use Service Worker's storage API — cache — a global object on the service worker that allows us to store assets delivered by responses, and keyed by their requests. This API works in a similar way to the browser's standard cache, but it is specific to your domain. It persists until you tell it not to — again, you have full control.

Install and activate: populating your cache
After your service worker is registered, the browser will attempt to install then activate the service worker for your page/site.

The install event is fired when an install is successfully completed. The install event is generally used to populate your browser's offline caching capabilities with the assets you need to run your app offline. To do this, we use Service Worker's storage API — `cache` — a global object on the service worker that allows us to store assets delivered by responses, and keyed by their requests. This API works in a similar way to the browser's standard cache, but it is specific to your domain. It persists until you tell it not to — again, you have full control.
Let's start this section by looking at a code sample — this is the first block you'll find in our service worker:

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('v1').then((cache) => {
      return cache.addAll([
        './sw-test/',
        './sw-test/index.html',
        './sw-test/style.css',
        './sw-test/app.js',
        './sw-test/image-list.js',
        './sw-test/star-wars-logo.jpg',
        './sw-test/gallery/',
        './sw-test/gallery/bountyHunters.jpg',
        './sw-test/gallery/myLittleVader.jpg',
        './sw-test/gallery/snowTroopers.jpg'
      ]);
    })
  );
});
```

1. Here we add an `install` event listener to the service worker (hence `self`), and then chain a `ExtendableEvent.waitUntil()` method onto the event — this ensures that the service worker will not install until the code inside waitUntil() has successfully occurred.

2. Inside `waitUntil()` we use the `caches.open()` method to create a new cache called `v1`, which will be version 1 of our site resources cache. This returns a promise for a created cache; once resolved, we then call a function that calls `addAll()` on the created cache, which for its parameter takes an array of origin-relative URLs to all the resources you want to cache.

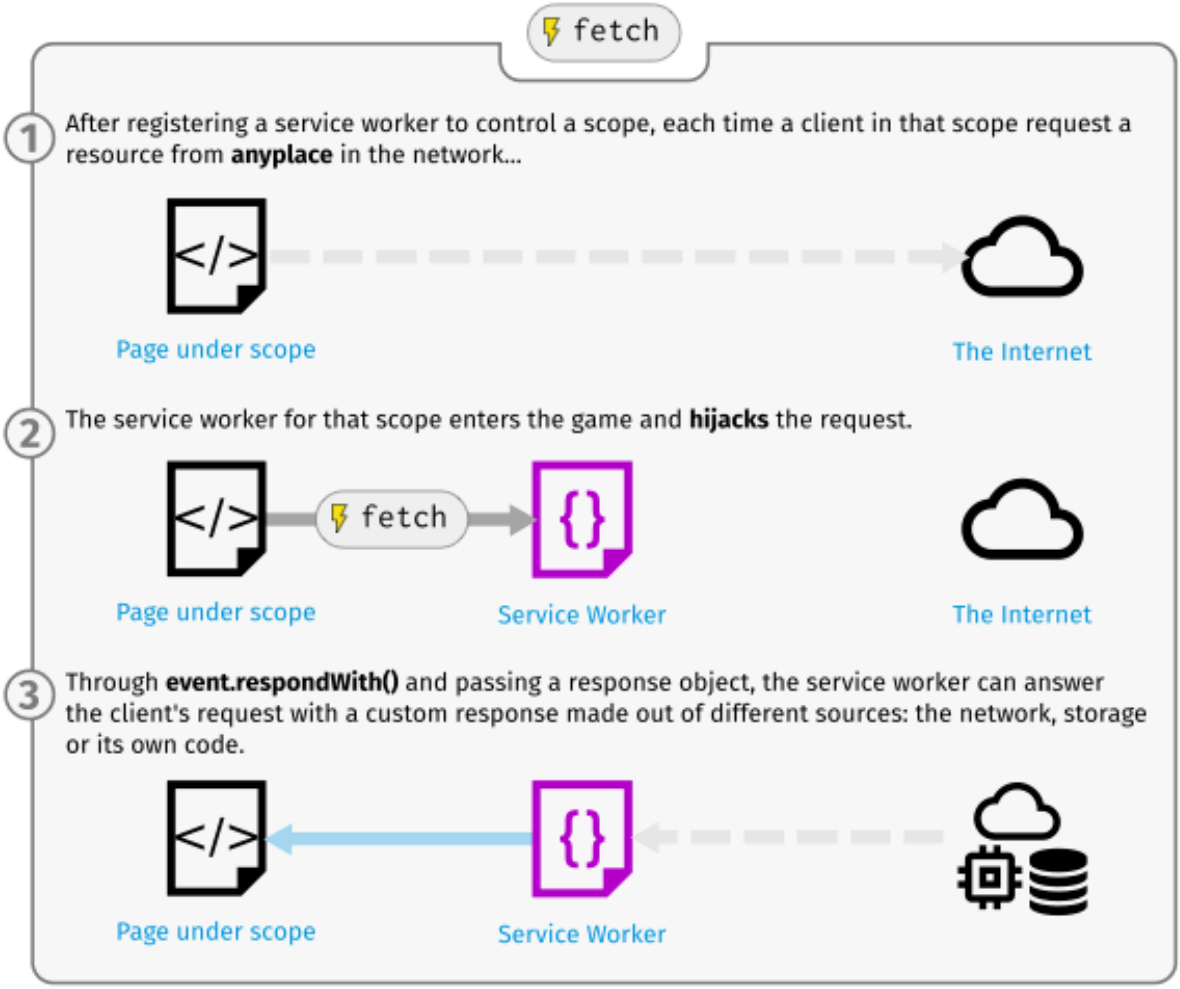
3. If the promise is rejected, the install fails, and the worker won't do anything. This is OK, as you can fix your code and then try again the next time registration occurs.

4. After a successful installation, the service worker activates. This doesn't have much of a distinct use the first time your service worker is installed/activated, but it means more when the service worker is updated.

Note: `localStorage` works in a similar way to `service worker cache`, but it is synchronous, so not allowed in service workers.

Note: `IndexedDB` can be used inside a service worker for data storage if you require it.

Custom responses to requests
Now you've got your site assets cached, you need to tell service workers to do something with the cached content. This is easily done with the fetch event.



A `fetch` event fires every time any resource controlled by a service worker is fetched, which includes the documents inside the specified scope, and any resources referenced in those documents (for example if `index.html` makes a cross origin request to embed an image, that still goes through its service worker.)
You can attach a `fetch` event listener to the service worker, then call the `respondWith()` method on the event to hijack our HTTP responses and update them with your own magic.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    // magic goes here
  );
});
```

We could start by responding with the resource whose url matches that of the network request, in each case:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
  );
});
```

`caches.match(event.request)` allows us to match each resource requested from the network with the equivalent resource available in the cache, if there is a matching one available. The matching is done via URL and various headers, just like with normal HTTP requests.

Let's look at a few other options we have when defining our magic:

- The `Response()` constructor allows you to create a custom response. In this case, we are just returning a simple text string:

```
new Response('Hello from your friendly neighborhood service worker!');
```

- This more complex `Response` below shows that you can optionally pass a set of headers in with your response, emulating standard HTTP response headers. Here we are just telling the browser what the content type of our synthetic response is:

```
new Response('<p>Hello from your friendly neighborhood service work
  headers: { 'Content-Type': 'text/html' }
});
```

- If a match wasn't found in the cache, you could tell the browser to `fetch()` the default network request for that resource, to get the new resource from the network if it is available:

```
fetch(event.request);
```

- If a match wasn't found in the cache, and the network isn't available, you could just match the request with some kind of default fallback page as a response using `match()`, like this:

```
caches.match('./fallback.html');
```

- You can retrieve a lot of information about each request by calling parameters of the `Request` object returned by the `FetchEvent`:

```
event.request.url
event.request.method
event.request.headers
event.request.body
```

Recovering failed requests
So caches.match(event.request) is great when there is a match in the service worker cache, but what about cases when there isn't a match? If we didn't provide any kind of failure handling, our promise would resolve with undefined and we wouldn't get anything returned.

Fortunately, service workers' promise-based structure makes it trivial to provide further options towards success. We could do this:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

If the resources aren't in the cache, they are requested from the network.

If we were being really clever, we would not only request the resource from the network; we would also save it into the cache so that later requests for that resource could be retrieved offline too! This would mean that if extra images were added to the Star Wars gallery, our app could automatically grab them and cache them. The following would do the trick:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((resp) => {
      return resp || fetch(event.request).then((response) => {
        return caches.open('v1').then((cache) => {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

Here we return the default network request with `return fetch(event.request)`, which returns a promise. When this promise is resolved, we respond by running a function that grabs our cache using `caches.open('v1')`; this also returns a promise. When that promise resolves, `cache.put()` is used to add the resource to the cache. The resource is grabbed from `event.request`, and the response is then cloned with `response.clone()` and added to the cache. The clone is put in the cache, and the original response is returned to the browser to be given to the page that called it.

Cloning the response is necessary because request and response streams can only be read once. In order to return the response to the browser and put it in the cache we have to clone it. So the original gets returned to the browser and the clone gets sent to the cache. They are each read once.

The only trouble we have now is that if the request doesn't match anything in the cache, and the network is not available, our request will still fail. Let's provide a default fallback so that whatever happens, the user will at least get something:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((resp) => {
      return resp || fetch(event.request).then((response) => {
        let responseClone = response.clone();
        caches.open('v1').then((cache) => {
          cache.put(event.request, responseClone);
        });

        return response;
      }).catch(() => {
        return caches.match('./sw-test/gallery/myLittleVader.jpg');
      })
    });
  );
});
```

We have opted for this fallback image because the only updates that are likely to fail are new images, as everything else is depended on for installation in the `install` event listener we saw earlier.

Updating your service worker
If your service worker has previously been installed, but then a new version of the worker is available on refresh or page load, the new version is installed in the background, but not yet activated. It is only activated when there are no longer any pages loaded that are still using the old service worker. As soon as there are no more such pages still loaded, the new service worker activates.

You'll want to update your `install` event listener in the new service worker to something like this (notice the new version number):

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('v2').then((cache) => {
      return cache.addAll([
        './sw-test/',
        './sw-test/index.html',
        './sw-test/style.css',
        './sw-test/app.js',
        './sw-test/image-list.js',

        …

        // include other new resources for the new version...
      ]);
    })
  );
});
```

While this happens, the previous version is still responsible for fetches. The new version is installing in the background. We are calling the new cache `v2`, so the previous `v1` cache isn't disturbed.

When no pages are using the current version, the new worker activates and becomes responsible for fetches.

Deleting old caches
You also get an `activate` event. This is a generally used to do stuff that would have broken the previous version while it was still running, for example getting rid of old caches. This is also useful for removing data that is no longer needed to avoid filling up too much disk space — each browser has a hard limit on the amount of cache storage that a given service worker can use. The browser does its best to manage disk space, but it may delete the Cache storage for an origin. The browser will generally delete all of the data for an origin or none of the data for an origin.

Promises passed into `waitUntil()` will block other events until completion, so you can rest assured that your clean-up operation will have completed by the time you get your first fetch event on the new service worker.

```
self.addEventListener('activate', (event) => {
  var cacheKeeplist = ['v2'];

  event.waitUntil(
    caches.keys().then((keyList) => {
      return Promise.all(keyList.map((key) => {
        if (cacheKeeplist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

Developer tools
Chrome has `chrome://inspect/#service-workers`, which shows current service worker activity and storage on a device, and `chrome://serviceworker-internals`, which shows more detail and allows you to start/stop/debug the worker process. In the future they will have throttling/offline modes to simulate bad or non-existent connections, which will be a really good thing.
Firefox has also started to implement some useful tools related to service workers:

- You can navigate to `about:debugging` to see what SWs are registered and update/remove them.
- When testing you can get around the HTTPS restriction by checking the "Enable Service Workers over HTTP (when toolbox is open)" option in the Firefox Developer Tools settings.
- The "Forget" button, available in Firefox's customization options, can be used to clear service workers and their caches.

Web app manifests

Web app manifests are part of a collection of web technologies called `progressive web apps (PWAs)`, which are websites that can be installed to a device's homescreen without an app store. Unlike regular web apps with simple homescreen links or bookmarks, PWAs can be downloaded in advance and can work offline, as well as use regular `Web APIs`.

The web app manifest provides information about a web application in a `JSON` text file, necessary for the web app to be downloaded and be presented to the user similarly to a native app (e.g., be installed on the homescreen of a device, providing users with quicker

access and a richer experience). PWA manifests include its name, author, icon(s), version, description, and list of all the necessary resources (among other things).

Members
Web manifests can contain the following keys. Click on each one to link through to more information about it:

- background_color
- categories
- description
- dir
- display
- iarc_rating_id
- icons
- lang
- name
- orientation
- prefer_related_applications
- protocol_handlers
- related_applications
- scope
- screenshots
- short_name
- shortcuts
- start_url
- theme_color

Example manifest

```
{
  "name": "HackerWeb",
  "short_name": "HackerWeb",
  "start_url": ".",
  "display": "standalone",
  "background_color": "#fff",
  "description": "A readable Hacker News app.",
  "icons": [{
    "src": "images/touch/homescreen48.png",
    "sizes": "48x48",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen72.png",
    "sizes": "72x72",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen96.png",
    "sizes": "96x96",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen144.png",
    "sizes": "144x144",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen168.png",
    "sizes": "168x168",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen192.png",
    "sizes": "192x192",
    "type": "image/png"
  }],
  "related_applications": [{
    "platform": "play",
    "url": "https://play.google.com/store/apps/details?id=cheeaun.hackerweb"
  }]
}
```

Deploying a manifest
Web app manifests are deployed in your HTML pages using a <link> element in the <head> of a document:

```
<link rel="manifest" href="manifest.json">
```

The `.webmanifest` extension is specified in the Media type registration section of the specification (the response of the manifest file should return `Content-Type: application/manifest+json`). Browsers generally support manifests with other appropriate extensions like `.json (Content-Type: application/json)`.
If the manifest requires credentials to fetch, the `crossorigin` attribute must be set to `use-credentials`, even if the manifest file is in the same origin as the current page.

```
<link rel="manifest" href="/app.webmanifest" crossorigin="use-credentials">
```

Splash screens
In some browsers (Chrome 47 and later, for example), a splash screen is displayed for sites launched from a homescreen. This splash screen is auto-generated from properties in the web app manifest, specifically:

- name
- background_color
- The icon in the icons array that is closest to 128dpi for the device.

Progressive Enhancement

`Progressive enhancement` is a design philosophy that provides a baseline of essential content and functionality to as many users as possible, while delivering the best possible experience only to users of the most modern browsers that can run all the required code.

The word progressive in progressive enhancement means creating a design that achieves a simpler-but-still-usable experience for users of older browsers and devices with limited capabilities, while at the same time being a design that progresses the user experience up to a more-compelling, fully-featured experience for users of newer browsers and devices with richer capabilities.

`Feature detection` is generally used to determine whether browsers can handle more modern functionality, while `polyfills` are often used to add missing features with JavaScript.
Special notice should be taken of accessibility. Acceptable alternatives should be provided where possible.

Progressive enhancement is a useful technique that allows web developers to focus on developing the best possible websites while making those websites work on multiple unknown user agents. Graceful degradation is related but is not the same thing and is often seen as going in the opposite direction to progressive enhancement. In reality both approaches are valid and can often complement one another.

Progressive Enhancement

Progressive enhancement is a design philosophy that provides a baseline of essential content and functionality to as many users as possible, while delivering the best possible experience only to users of the most modern browsers that can run all the required code. The word progressive in progressive enhancement means creating a design that achieves a simpler-but-still-usable experience for users of older browsers and devices with limited capabilities, while at the same time being a design that **progresses the user experience** up to a more-compelling, fully-featured experience for users of newer browsers and devices with richer capabilities.

`Feature detection` is generally used to determine whether browsers can handle more modern functionality, while `polyfills` are often used to add missing features with JavaScript.

Special notice should be taken of accessibility. Acceptable alternatives should be provided where possible.

Progressive enhancement is a useful technique that allows web developers to focus on developing the best possible websites while making those websites work on multiple unknown user agents. `Graceful degradation` is related but is not the same thing and is often seen as going in the opposite direction to progressive enhancement. In reality both approaches are valid and can often complement one another.

Progressive web apps (PWAs)

`Progressive Web Apps (PWAs)` are web apps that use `service workers`, `manifests`, and other web-platform features in combination with `progressive enhancement` to give users an experience on par with native apps.

PWAs provide a number of `advantages` to users — including being `installable`, `progressively enhanced`, `responsively designed`, `re-engageable`, `linkable`, `discoverable`, `network independent`, and `secure`.

Introduction to progressive web apps

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?

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?
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`. 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`, so the contents can be found through search engines.
- `Installable`, so it can be available on the device's home screen or app launcher.
- `Linkable`, so you can share it by sending a URL.
- `Network independent`, so it works offline or with a poor network connection.
- `Progressively enhanced`, so it's still usable on a basic level on older browsers, but fully-functional on the latest ones.
- `Re-engageable`, so it's able to send notifications whenever there's new content available.
- `Responsively designed`, so it's usable on any device with a screen and a browser—mobile phones, tablets, laptops, TVs, refrigerators, etc.
- `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 offered by web applications can create a compelling, highly flexible offering for your users and customers.

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

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`, which provides a format for specifying similar metadata in the HTML `<head>` block using `<meta>` tags.
The relevant web standard here is the `Web app manifest`, which defines features of an app such as name, icon, splash screen, and theme colors in a `JSON`-formatted manifest file. This is for use in contexts such as app listings and device home screens.

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.`

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:
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` to control page requests (for example storing them offline), the `Cache API` for storing responses to network

requests offline (very useful for storing site assets), and client-side data storage technologies such as `Web Storage` and `IndexedDB` to store application data offline.

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`, 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:
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` for sending updates straight from server to app via a service worker, and the `Notifications API` for generating system notifications to help engage users when they're not actively using their web browser.

Responsiveness:
Responsive web apps use technologies like media queries and viewport to make sure that their UIs will fit any form factor: desktop, mobile, tablet, or whatever comes next.

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` 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
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` support. Thankfully service workers are now supported on all major browsers on desktop and mobile.

Other features such as `Web App Manifest`, `Push Notifications`, and `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.

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, which we'll mention briefly.

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`, which will be explained in detail in the next article — for now let's focus on the structure itself.

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 or push notifications, 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
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`, `CSS media queries`, `Flexbox`, and `CSS Grid`.

Different concept: streams
An entirely different approach to server- or client-side rendering can be achieved with the `Streams API`. With a little help from service workers, streams can greatly improve the way we parse content.

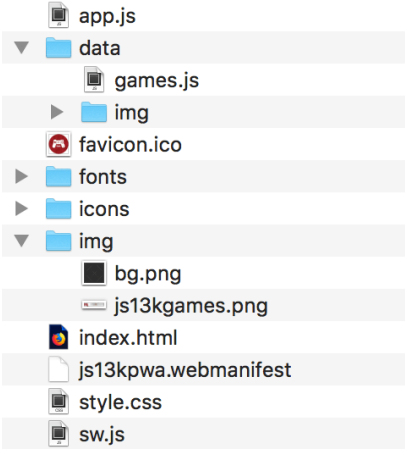
The app shell model requires all the resources to be available before the website can start rendering. It's different with HTML, as the browser is actually streaming the data already and you can see when the elements are loaded and rendered on the website. To have the JavaScript "operational", however,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

Structure of our example application
The `js13kPWA` website structure is quite simple: it consists of a single HTML file (index.html) with basic CSS styling (style.css), and a few images, scripts, and fonts. The folder structure looks like this:



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 c
        <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"
</header>
<main>
        <h1>js13kGames A-Frame entries</h1>
        <p class="description">List of games submitted to the <a href="http://js13kgames.com/aframe">A
        <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<
</footer>
</body>
</html>
```

The `<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>` is split into the `<header>` (containing linked image), `<main>` page (with title, description and place for a content), and `<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
The CSS is also as plain as possible: it uses `@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
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>
  </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
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
The games data is present in the data folder in a form of a JavaScript object (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.

Making PWAs work offline with Service workers

Now that we've seen what the structure of js13kPWA looks like and have seen the basic shell up and running, let's look at how the offline capabilities using Service Worker are implemented. We examine how to add offline functionality.

Service workers explained
Service Workers are a virtual proxy between the browser and the network. They finally fix issues that front-end developers have struggled with for years — most notably how to properly cache the assets of a website and make them available when the user's device is offline.

They run on a separate thread from the main JavaScript code of our page, and don't have any access to the DOM structure. This introduces a different approach from traditional web programming — the API is non-blocking, and can send and receive communication between different contexts. You are able to give a Service Worker something to work on, and receive the result whenever it is ready using a Promise-based approach.
They can do a lot more than "just" offering offline capabilities, including handling notifications, performing heavy calculations on a separate thread, etc. Service workers are quite powerful as they can take control over network requests, modify them, serve custom responses retrieved from the cache, or synthesize responses completely.

Security:
Because they are so powerful, Service Workers can only be executed in secure contexts (meaning HTTPS). If you want to experiment first before pushing your code to production, you can always test on a localhost or setup GitHub Pages — both support HTTPS.

Offline First
The "offline first" — or "cache first" — pattern is the most popular strategy for serving content to the user. If a resource is cached and available offline, return it first before trying to download it from the server. If it isn't in the cache already, download it and cache it for future usage.

"Progressive" in PWA
When implemented properly as a progressive enhancement, service workers can benefit users who have modern browsers that support the API by providing offline support, but won't break anything for those using legacy browsers.

Service workers in the js13kPWA app
Enough theory — let's see some source code!

Registering the Service Worker
We'll start by looking at the code that registers a new Service Worker, in the app.js file:

```
if('serviceWorker' in navigator) {
  navigator.serviceWorker.register('./pwa-examples/js13kpwa/sw.js');
};
```

If the service worker API is supported in the browser, it is registered against the site using the `ServiceWorkerContainer.register()` method. Its contents reside in the sw.js file, and can be executed after the registration is successful. It's the only piece of Service Worker code that sits inside the app.js file; everything else that is Service Worker-specific is written in the sw.js file itself.

Lifecycle of a Service Worker
When registration is complete, the sw.js file is automatically downloaded, then installed, and finally activated.

Installation:
The API allows us to add event listeners for key events we are interested in — the first one is the install event:

```
self.addEventListener('install', (e) => {
  console.log('[Service Worker] Install');
});
```

In the install listener, we can initialize the cache and add files to it for offline use. Our js13kPWA app does exactly that.

First, a variable for storing the cache name is created, and the app shell files are listed in one array.

```
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'
];
```

Next, the links to images to be loaded along with the content from the data/games.js file are generated in the second array. After that, both arrays are merged using the `Array.prototype.concat()` function.

```
const gamesImages = [];
for (let i = 0; i < games.length; i++) {
  gamesImages.push(`data/img/${games[i].slug}.jpg`);
}
const contentToCache = appShellFiles.concat(gamesImages);
```

Then we can manage the install event itself:

```
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);
  })());
});
```

There are two things that need an explanation here: what `ExtendableEvent.waitUntil` does, and what the caches object is.

The service worker does not install until the code inside waitUntil is executed. It returns a promise — this approach is needed because installing may take some time, so we have to wait for it to finish.

`caches` is a special `CacheStorage` object available in the scope of the given Service Worker to enable saving data — saving to `web storage` won't work, because web storage is synchronous. With Service Workers, we use the Cache API instead.

Here, we open a cache with a given name, then add all the files our app uses to the cache, so they are available next time it loads (identified by request URL).

You may notice we haven't cached `game.js`. This is the file that contains the data we use when displaying our games. In reality this data would most likely come from an API endpoint or database and caching the data would mean updating it periodically when there was network connectivity. We won't go into that here, but the `Periodic Background Sync API` is good further reading on this topic.

Activation:
There is also an activate event, which is used in the same way as install. This event is usually used to delete any files that are no longer necessary and clean up after the app in general. We don't need to do that in our app, so we'll skip it.

Responding to fetches
We also have a fetch event at our disposal, which fires every time an HTTP request is fired off from our app. This is very useful, as it allows us to intercept requests and respond to them with custom responses. Here is a simple usage example:

```
self.addEventListener('fetch', (e) => {
  console.log(`[Service Worker] Fetched resource ${e.request.url}`);
});
```

The response can be anything we want: the requested file, its cached copy, or a piece of JavaScript code that will do something specific — the possibilities are endless.

In our example app, we serve content from the cache instead of the network as long as the resource is actually in the cache. We do this whether the app is online or offline. If the file is not in the cache, the app adds it there first before then serving it:

```
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;
  })());
});
```

Here, we respond to the fetch event with a function that tries to find the resource in the cache and return the response if it's there. If not, we use another fetch request to fetch it from the network, then store the response in the cache so it will be available there next time it is requested.

The `FetchEvent.respondWith` method takes over control — this is the part that functions as a proxy server between the app and the network. This allows us to respond to

every single request with any response we want: prepared by the Service Worker, taken from cache, modified if needed.

That's it! Our app is caching its resources on install and serving them with fetch from the cache, so it works even if the user is offline. It also caches new content whenever it is added.

Updates
There is still one point to cover: how do you upgrade a Service Worker when a new version of the app containing new assets is available? The version number in the cache name is key to this:

```
var cacheName = 'js13kPWA-v1';
```

When this updates to v2, we can then add all of our files (including our new files) to a new cache:

```
contentToCache.push('/pwa-examples/js13kpwa/icons/icon-32.png');

// ...

self.addEventListener('install', (e) => {
  e.waitUntil((async () => {
    const cache = await caches.open(cacheName);
    await cache.addAll(contentToCache);
  })());
});
```

A new service worker is installed in the background, and the previous one (v1) works correctly up until there are no pages using it — the new Service Worker is then activated and takes over management of the page from the old one.

Clearing the cache
Remember the activate event we skipped? It can be used to clear out the old cache we don't need anymore:

```
self.addEventListener('activate', (e) => {
  e.waitUntil(caches.keys().then((keyList) => {
    return Promise.all(keyList.map((key) => {
      if (key === cacheName) { return; }
      return caches.delete(key);
    }))
  }));
});
```

This ensures we have only the files we need in the cache, so we don't leave any garbage behind; the available cache space in the browser is limited, so it is a good idea to clean up after ourselves.

Other use cases
Serving files from cache is not the only feature Service Worker offers. If you have heavy calculations to do, you can offload them from the main thread and do them in the worker, and receive results as soon as they are available. Performance-wise, you can prefetch

resources that are not needed right now, but might be in the near future, so the app will be faster when you actually need those resources.

How to make PWAs installable

In the last article, we read about how the example application, `js13kPWA`, works offline thanks to its `service worker`, but we can go even further and allow users to install the web app on mobile and desktop browsers that support doing so. The installed web app can then be launched by users just as if it were any native app. This article explains how to achieve this using the web app's manifest.
These technologies allow the app to be launched directly from the device's home screen, rather than the user having to open the browser and then navigate to the site by using a bookmark or typing the URL. Your web app can sit next to native applications as first class citizens. This makes the web app easier to access; additionally, you can specify that the app be launched in fullscreen or standalone mode, thus removing the default browser user interface that would otherwise be present, creating an even more seamless and native-like feel.

Requirements
To make the web site installable, it needs the following things in place:

- A web manifest, with the correct fields filled in
- The web site to be served from a secure (HTTPS) domain
- An icon to represent the app on the device
- A service worker registered, to allow the app to work offline (this is required only by Chrome for Android currently)

Note: Currently, only the Chromium-based browsers such as Chrome, Edge, and Samsung Internet require the service worker. If developing your app using Firefox, be aware that you will need a service worker to be compatible with Chromium-based browsers.

The manifest file
The key element is a web manifest file, which lists all the information about the website in a JSON format.

It usually resides in the root folder of a web app. It contains useful information, such as the app's title, paths to different-sized icons that can be used to represent the app on an OS (such as an icon on the home screen, an entry in the Start menu, or an icon on the desktop), and a background color to use in loading or splash screens. This information is needed for the browser to present the web app properly during the installation process, as well as within the device's app-launching interface, such as the home screen of a mobile device.

The `js13kpwa.webmanifest` file of thejs13kPWA web app is included in the `<head>` block of the `index.html` file using the following line of code:

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

Note: There are a few common kinds of manifest file that have been used in the past: `manifest.webapp` was popular in Firefox OS app manifests, and many

use `manifest.json` for web manifests as the contents are organized in a JSON structure. However, the `.webmanifest` file format is explicitly mentioned in the W3C manifest specification

The content of the file looks like this:

```
{
  "name": "js13kGames Progressive Web App",
  "short_name": "js13kPWA",
  "description": "Progressive Web App that lists games submitted to the A-Frame category in the js13k
  "icons": [
    {
      "src": "icons/icon-32.png",
      "sizes": "32x32",
      "type": "image/png"
    },
    // ...
    {
      "src": "icons/icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ],
  "start_url": "/pwa-examples/js13kpwa/index.html",
  "display": "fullscreen",
  "theme_color": "#B12A34",
  "background_color": "#B12A34"
}
```

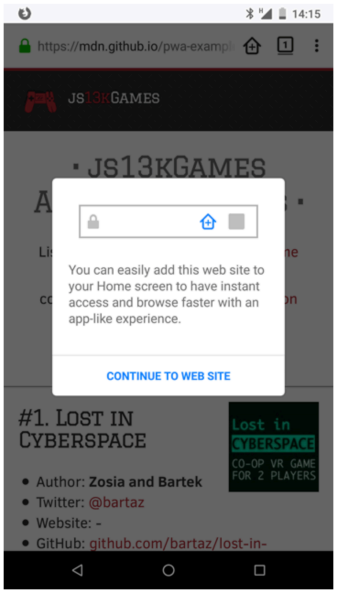
Most of the fields are self-explanatory, but to be certain we're on the same page:

- `name`: The full name of your web app.
- `short_name`: Short name to be shown on the home screen.
- `description`: A sentence or two explaining what your app does.
- `icons`: A bunch of icon information — source URLs, sizes, and types. Be sure to include at least a few, so that one that fits best will be chosen for the user's device.
- `start_url`: The index document to launch when starting the app.
- `display`: How the app is displayed; can be `fullscreen`, `standalone`, `minimal-ui`, or `browser`.
- `theme_color`: A primary color for the UI, used by operating system.
- `background_color`: A color used as the app's default background, used during install and on the splash screen.

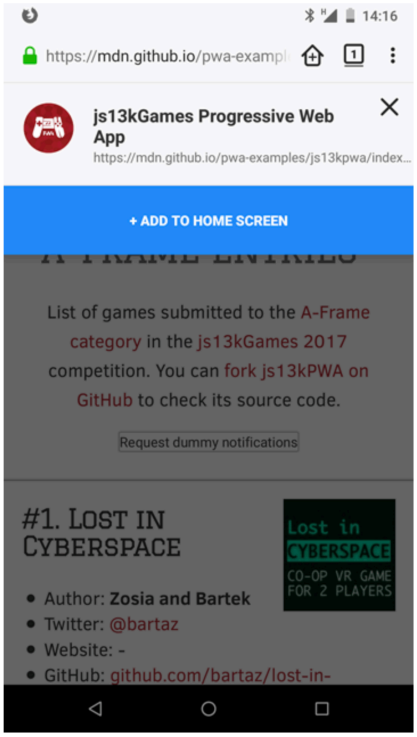
A minimal web manifest must have at least a `name` and an `icons` field with at least one icon defined; that icon must have at least the `src`, `sizes`, and `type` sub-fields as well. Beyond that, everything is optional, though the `description`, `short_name`, and `start_url` fields are recommended. There are even more fields you can use than listed above — be sure to check the Web App Manifest reference for details.

Add to home screen
"Add to home screen" (or a2hs for short) is a feature implemented by mobile browsers that takes the information found in an app's web manifest and uses them to represent the app on the device's home screen with an icon and name. This only works if the app meets all the necessary requirements, as described above.

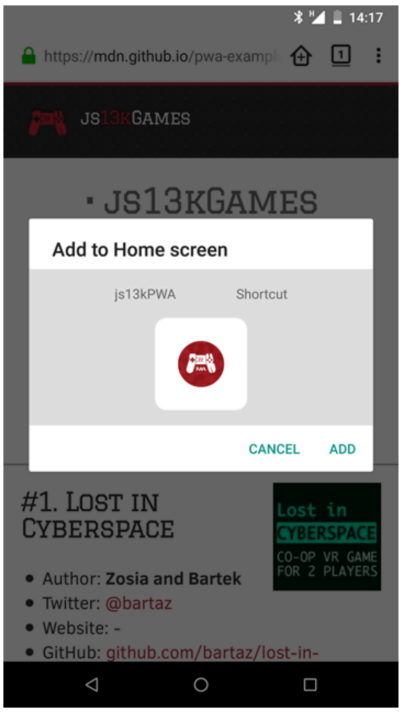
When the user visits the PWA with a supporting mobile browser, it should display a notification (such as a banner or dialog box) indicating that it's possible to install the app as a PWA.



After the user indicates they wish to proceed with installation, the install banner is shown. That banner is automatically created by the browser, based on the information from the manifest file. For instance, the prompt includes the app's name and icon.



If the user clicks the button, there is a final step showing what the app will look like, and letting the user choose if they definitely want to add the app.



When confirmed, the app will be installed on the home screen.

Now the user can launch and use the web app just like any other application on their device. Depending on the device and operating system, the web app's icon may be badged with a small icon that indicates that it's a web app. In the screen shot above, for example, the app has a tiny Firefox icon, indicating that it's a web app that uses the Firefox runtime.

Splash screen
In some browsers, a splash screen is also generated from the information in the manifest, which is shown when the PWA is launched and while it's being loaded started up.

The icon and the theme and background colors are used to create this screen.

How to make PWAs re-engageable using Notifications and Push

Having the ability to cache the contents of an app to work offline is a great feature. Allowing the user to install the web app on their home screen is even better. But instead of relying only on user actions, we can do more, using push messages and notifications to automatically re-engage and deliver new content whenever it is available.

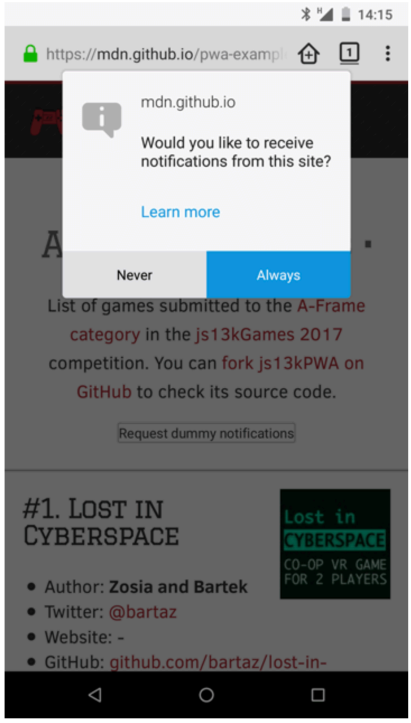
Two APIs, one goal
The `Push API` and `Notifications API` are two separate APIs, but they work well together when you want to provide engaging functionality in your app. Push is used to deliver new content from the server to the app without any client-side intervention, and its operation is handled by the app's service worker. Notifications can be used by the service worker to show new information to the user, or at least alert them when something has been updated.
They work outside of the browser window, just like service workers, so updates can be pushed and notifications can be shown when the app's page is out of focus or even closed.

Notifications
Let's start with notifications — they can work without push, but are very useful when combined with them. Let’s look at them in isolation to begin with.

Request permission:
To show a notification, we have to request permission to do so first. Instead of showing the notification immediately though, best practice dictates that we should show the popup when the user requests it by clicking on a button:

```
const button = document.getElementById('notifications');
button.addEventListener('click', () => {
  Notification.requestPermission().then((result) => {
    if (result === 'granted') {
      randomNotification();
    }
  });
})
```

This shows a popup using the operating system's own notifications service:



When the user confirms to receive notifications, the app can then show them. The result of the user action can be default, granted or denied. The default option is chosen when the user won't make a choice, and the other two are set when the user clicks yes or no respectively.

When accepted, the permission works for both notifications and push.

Create a notification
The example app creates a notification out of the available data — a game is picked at random, and the chosen one feeds the notification with the content: it sets the game's name as the title, mentioning the author in the body, and showing the image as an icon:

```
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);
}
```

A new random notification is created every 30 seconds until it becomes too annoying and is disabled by the user. (For a real app, the notifications should be much less frequent, and more useful.) The advantage of the Notifications API is that it uses the operating system's notification functionality. This means that notifications can be displayed to the user even when they are not looking at the web app, and the notifications look similar to ones displayed by native apps.

Push
Push is more complicated than notifications — we need to subscribe to a server that will then send the data back to the app. The app's Service Worker will receive data from the push server, which can then be shown using the notifications system, or another mechanism if desired.

The technology is still at a very early stage — some working examples use the Google Cloud Messaging platform, but are being rewritten to support `VAPID` (Voluntary Application Identification), which offers an extra layer of security for your app. You can examine the `Service Workers Cookbook examples`, try to set up a push messaging server using `Firebase`, or build your own server (using Node.js for example).

As mentioned before, to be able to receive push messages, you have to have a service worker, the basics of which are already explained in the Making PWAs work offline with Service workers article. Inside the service worker, a push service subscription mechanism is created.

```
registration.pushManager.getSubscription().then( /* ... */ );
```

Once the user is subscribed, they can receive push notifications from the server.

From the server-side, the whole process has to be encrypted with public and private keys for security reasons — allowing everyone to send push messages unsecured using your app would be a terrible idea. See the Web Push data encryption test page for detailed information about securing the server. The server stores all the information received when the user subscribed, so the messages can be sent later on when needed.

To receive push messages, we can listen to the push event in the Service Worker file:

```
self.addEventListener('push', (e) => { /* ... */ });
```

The data can be retrieved and then shown as a notification to the user immediately. This, for example, can be used to remind the user about something, or let them know about new content being available in the app.

Push example
Push needs the server part to work, This demo consists of three files:

- index.js, which contains the source code of our app
- server.js, which contains the server part (written in Node.js)
- service-worker.js, which contains the Service Worker-specific code.
- Let's explore all of these

index.js
The index.js file starts by registering the service worker:

```
navigator.serviceWorker.register('service-worker.js')
  .then((registration) => {
    return registration.pushManager.getSubscription()
      .then(async (subscription) => {
        // registration part
      });
  })
  .then((subscription) => {
    // subscription part
  });
```

It is a little bit more complicated than the service worker we saw in the `js13kPWA` demo. In this particular case, after registering, we use the registration object to subscribe, and then use the resulting subscription object to complete the whole process.

In the registration part, the code looks like this:

```
if(subscription) {
  return subscription;
}
```

If the user has already subscribed, we then return the subscription object and move to the subscription part. If not, we initialize a new subscription:

```
const response = await fetch('./vapidPublicKey');
const vapidPublicKey = await response.text();
const convertedVapidKey = urlBase64ToUint8Array(vapidPublicKey);
```

The app fetches the server's public key and converts the response to text; then it needs to be converted to a Uint8Array (to support Chrome).

The app can now use the PushManager to subscribe the new user. There are two options passed to the PushManager.subscribe() method — the first is userVisibleOnly: true, which means all the notifications sent to the user will be visible to them, and the second one is the applicationServerKey, which contains our successfully acquired and converted VAPID key.

```
return registration.pushManager.subscribe({
  userVisibleOnly: true,
  applicationServerKey: convertedVapidKey
});
```

Now let's move to the subscription part — the app first sends the subscription details as JSON to the server using Fetch.

```
fetch('./register', {
  method: 'post',
  headers: {
    'Content-type': 'application/json'
  },
  body: JSON.stringify({
    subscription: subscription
  }),
});
```

Then the GlobalEventHandlers.onclick function on the Subscribe button is defined:

```
document.getElementById('doIt').onclick = function() {
  const payload = document.getElementById('notification-payload').value;
  const delay = document.getElementById('notification-delay').value;
  const ttl = document.getElementById('notification-ttl').value;

  fetch('./sendNotification', {
    method: 'post',
    headers: {
      'Content-type': 'application/json'
    },
    body: JSON.stringify({
      subscription: subscription,
      payload: payload,
      delay: delay,
      ttl: ttl,
    }),
  });
};
```

When the button is clicked, fetch asks the server to send the notification with the given parameters: payload is the text that to be shown in the notification, delay defines a delay in seconds until the notification will be shown, and ttl is the time-to-live setting that keeps the notification available on the server for a specified amount of time, also defined in seconds.

Now, onto the next JavaScript file.

server.js
The server part is written in Node.js and needs to be hosted somewhere suitable, which is a subject for an entirely separate article. We will only provide a high-level overview here.
The web-push module is used to set the VAPID keys, and optionally generate them if they are not available yet.

```
const webPush = require('web-push');

if (!process.env.VAPID_PUBLIC_KEY || !process.env.VAPID_PRIVATE_KEY) {
  console.log("You must set the VAPID_PUBLIC_KEY and VAPID_PRIVATE_KEY "+
    "environment variables. You can use the following ones:");
  console.log(webPush.generateVAPIDKeys());
  return;
}

webPush.setVapidDetails(
  'https://serviceworke.rs/',
  process.env.VAPID_PUBLIC_KEY,
  process.env.VAPID_PRIVATE_KEY
);
```

Next, a module defines and exports all the routes an app needs to handle: getting the VAPID public key, registering, and then sending notifications. You can see the variables from the index.js file being used: payload, delay and ttl.

```
module.exports = function(app, route) {
  app.get(route + 'vapidPublicKey', function(req, res) {
    res.send(process.env.VAPID_PUBLIC_KEY);
  });

  app.post(route + 'register', function(req, res) {

    res.sendStatus(201);
  });

  app.post(route + 'sendNotification', function(req, res) {
    const subscription = req.body.subscription;
    const payload = req.body.payload;
    const options = {
      TTL: req.body.ttl
    };

    setTimeout(function() {
      webPush.sendNotification(subscription, payload, options)
      .then(function() {
        res.sendStatus(201);
      })
      .catch(function(error) {
        console.log(error);
        res.sendStatus(500);
      });
    }, req.body.delay * 1000);
  });
};
```

service-worker.js
The last file we will look at is the service worker:

```
self.addEventListener('push', function(event) {
  const payload = event.data ? event.data.text() : 'no payload';
  event.waitUntil(
    self.registration.showNotification('ServiceWorker Cookbook', {
        body: payload,
    })
  );
});
```