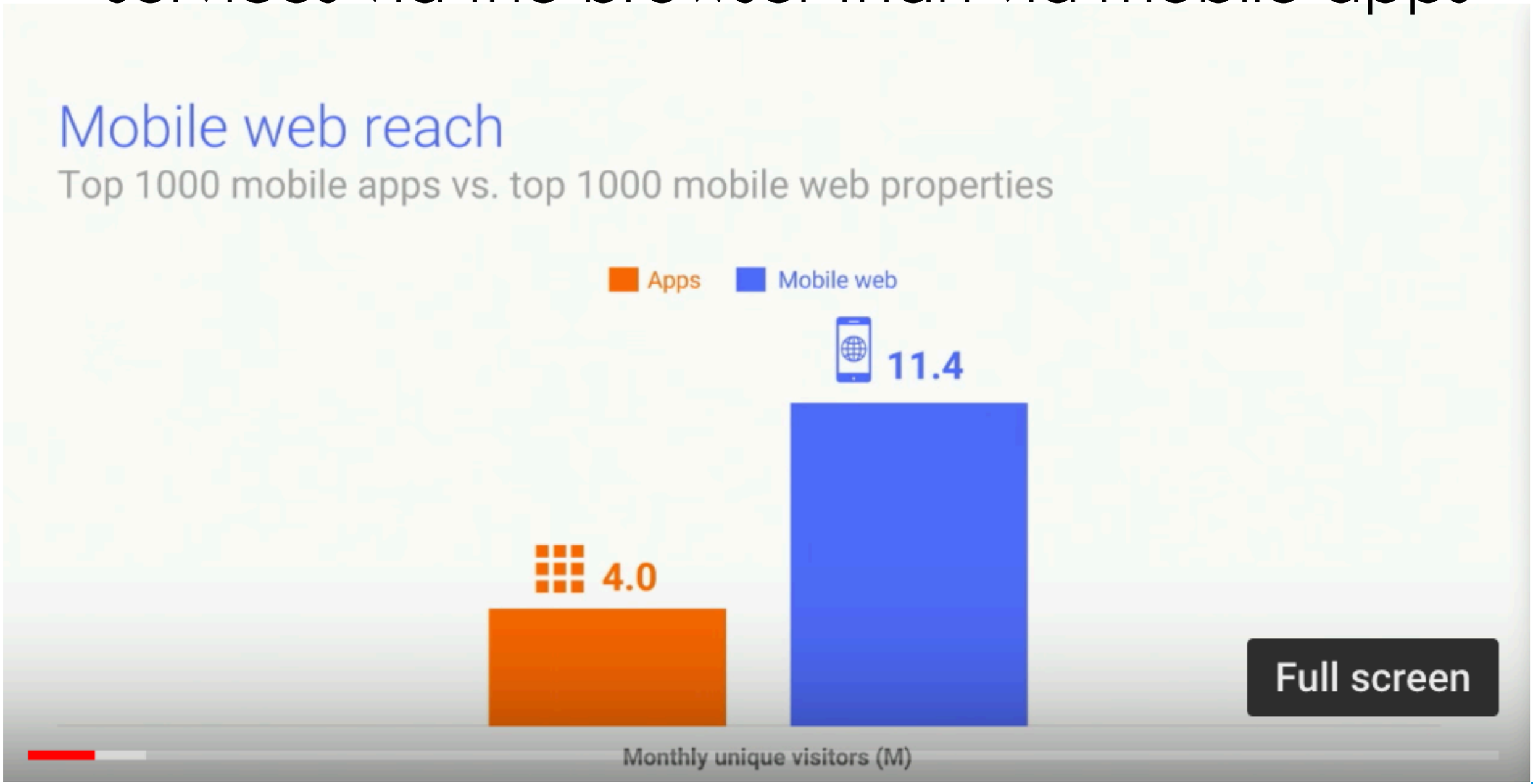
Offline Experience for the Mobile Web: Service Workers

Prof Fabio Ciravegna
Department of Computer Science, The University of Sheffield
f.ciravegna@shef.ac.uk

Apps Vs Web Users

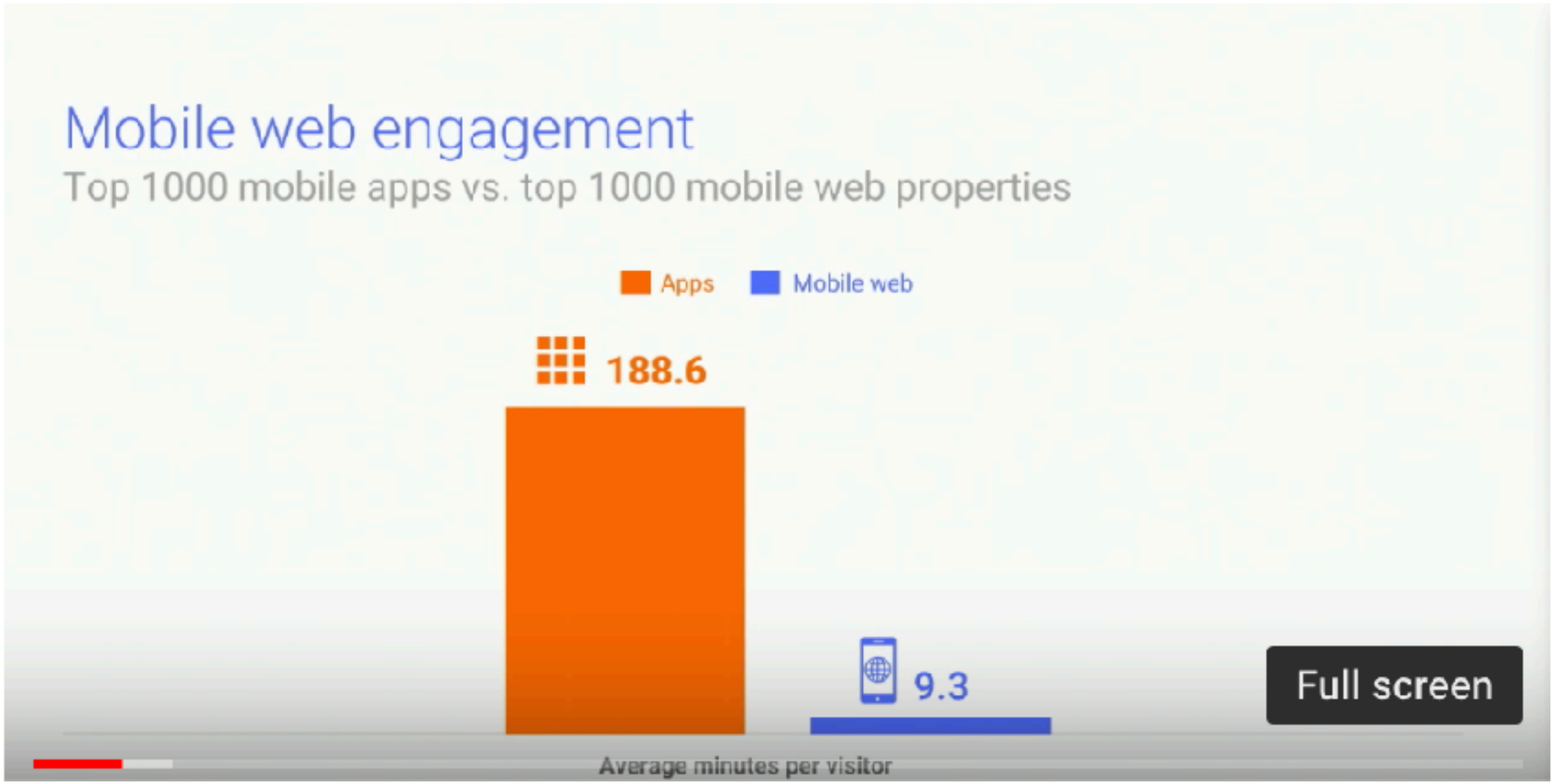
- Users are far more likely to access Web services via the browser than via mobile apps



https://developers.google.com/web/progressive-web-apps/

The Mobile Web

- However, those using apps are more easily engaged and spend more time on you content



Issues with the Mobile Web

- Despite bringing traffic to your site, it is unlikely to bring returning customers
  - difficulty in typing URLs, general cumbersome browser interface
- Despite the advances in responsive, mobile-first websites web content may still be unnatural on mobile browsers
- Break in connectivity is the norm rather than the exception in mobile web experience
  - especially if out of town or using underground trains
- Sluggish network are even more the norm
  - anywhere really

Towards a different Web Experience

- The real advantage of apps is the ability to
  - design for the specific device
    - e.g. using Android's or iOS's own design style
  - use the phone's sensors
    - HTML can use the GPS and camera only (although things are changing)
  - provide a perfect offline experience
    - Content may not be updated if offline but apps can take opportunistic strategies
      - to download content when online
      - to store content for off line viewing
  - allow push notifications even when the user is using a different app
- The HTTP protocol does not allow any of the above
  - although HTML5 allows storage of content to a limited extent

Service Workers

https://developers.google.com/web/fundamentals/primers/service-workers/

- A service worker is a script that your browser runs in the background,
  - separate from a web page,
  - opening the door to features that don't need a web page or user interaction.
- Today, they already include features like push notifications and background sync
  - In the future, service workers might support other things like periodic sync or geofencing
- The reason this is such an exciting API is that it allows you to support offline experiences, giving developers complete control over the experience

Features of SW

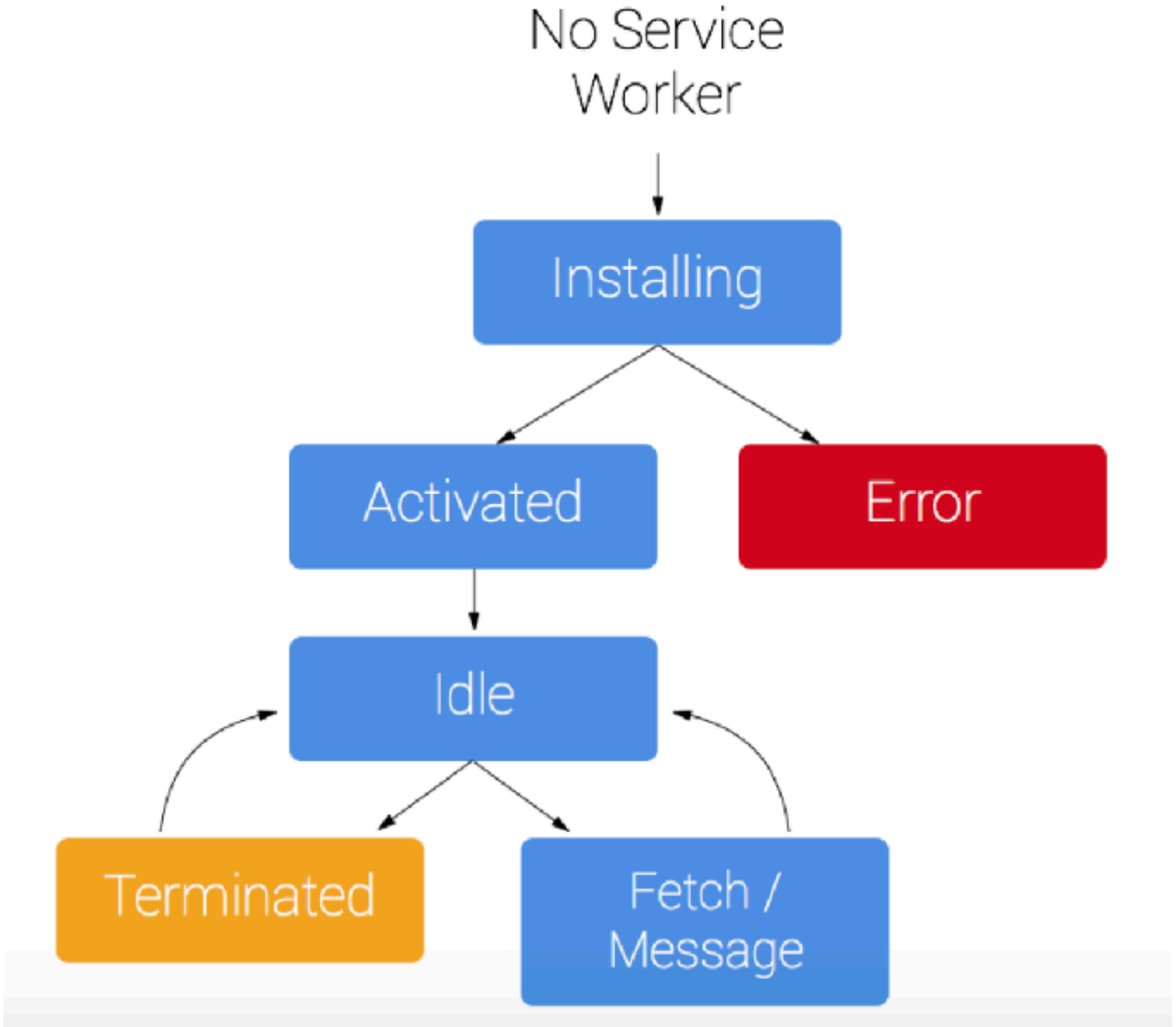
- It's a JavaScript Worker,
  - so it can't access a page's DOM directly
  - A service worker can communicate with the pages it controls via messages and communicate with pages which will manipulate the DOM if needed
- It is a programmable **network** proxy
  - allowing to control network requests from your page are handled (so it can be see as an evolution of the Ajax paradigm)
- It's terminated when not in use, and restarted when it's next needed
  - you cannot rely on global state
  - If there is information that needs persisting and reusing across restarts, use the IndexedDB API.
- Service workers make extensive use of promises
  - make sure to understand them!

Life Cycle

- Their lifecycle is completely separate from your web page
  - they are not just a Javascript script run on your page, they have an independent life
  - they are started by running a JS script in a page but then they take their own life
    - think of a background process in mobile computing
      - they are launched by the UI Thread and then become independent and can only communicate with the UI Thread via a specific communication channel
- Steps:
  - installation
    - cached assets (web pages) are loaded
  - activation
    - the service worker control network communication between the pages and the server

The University Of Sheffield.

Life Cycle of a Service Worker



Requirements

- Browser support:
  - Latest versions of browsers fully support them
    - IE is not a browser, MS Edge may not support them
      - Anyway do people really use them?
- HTTPS Secure connection to the server
  - A service worker effectively hijack connections, fabricate, and filter responses
    - making any middle in the man attack extremely dangerous
    - you can only register service workers on pages served over HTTPS,
      - so we know the service worker the browser receives hasn't been tampered with during its journey through the network.
    - See next slides for HTTPS

Life Cycle

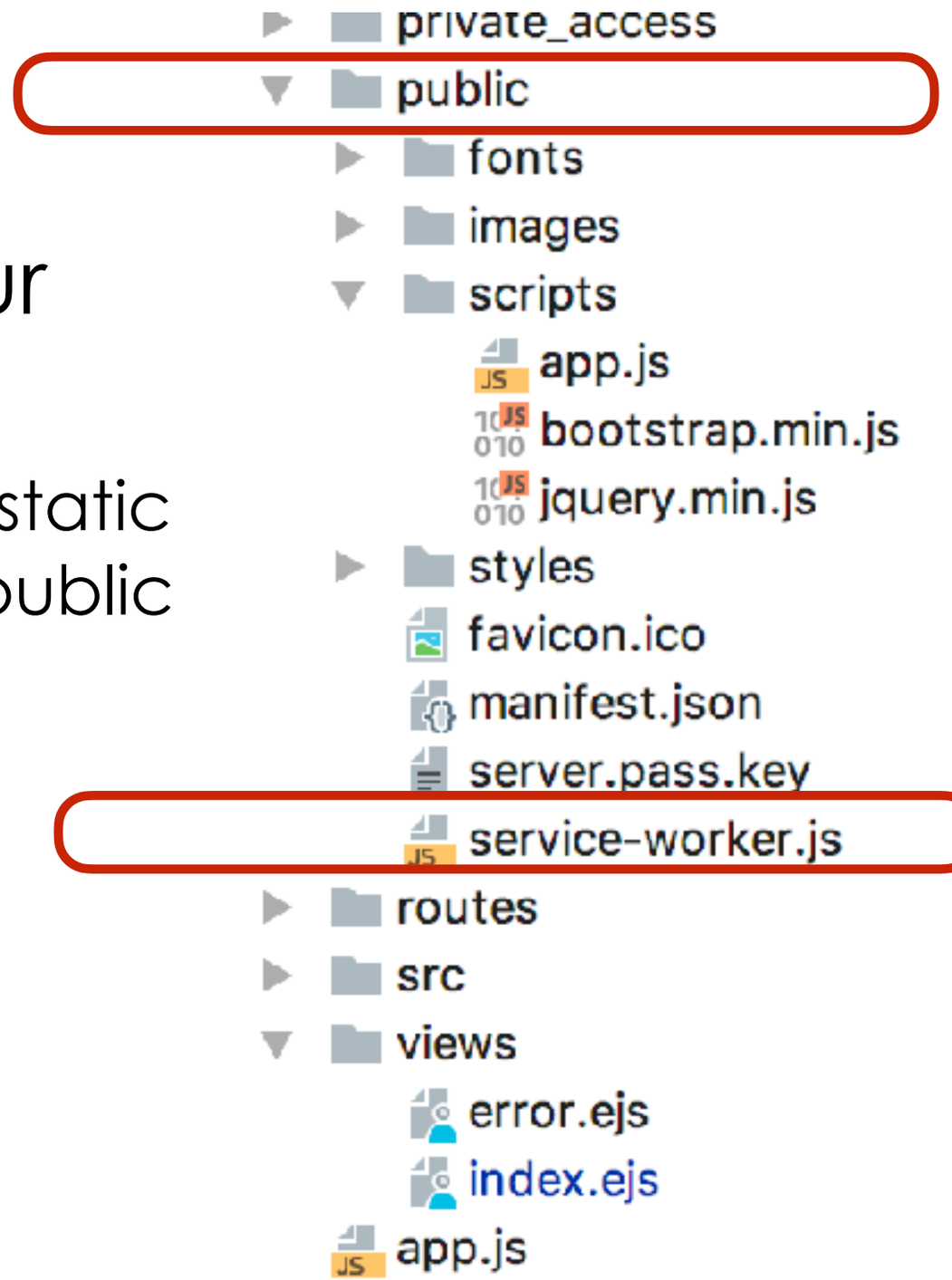
- To register a service worker, you must execute a script on the website's home page

```
// if the browser supports service workers

if ('serviceWorker' in navigator) {
  // run this code at loading time
   window.addEventListener('load', function() {
     // register the service worker. the service worker is a js file
     // AND *must* be installed on the root of the set of pages it
     // controls. You can reload the pages as many times as you want
     // do not worry about registering a service worker multiple times
    navigator.serviceWorker.register('/sw.js')
        .then(function(registration) {
          // Registration was successful
         console.log('ServiceWorker registration successful with scope:',
             registration.scope);
        }, function(err) {
          // registration failed :(
          console.log('ServiceWorker registration failed: ', err);
        });
  });}
```

Scope of a SW

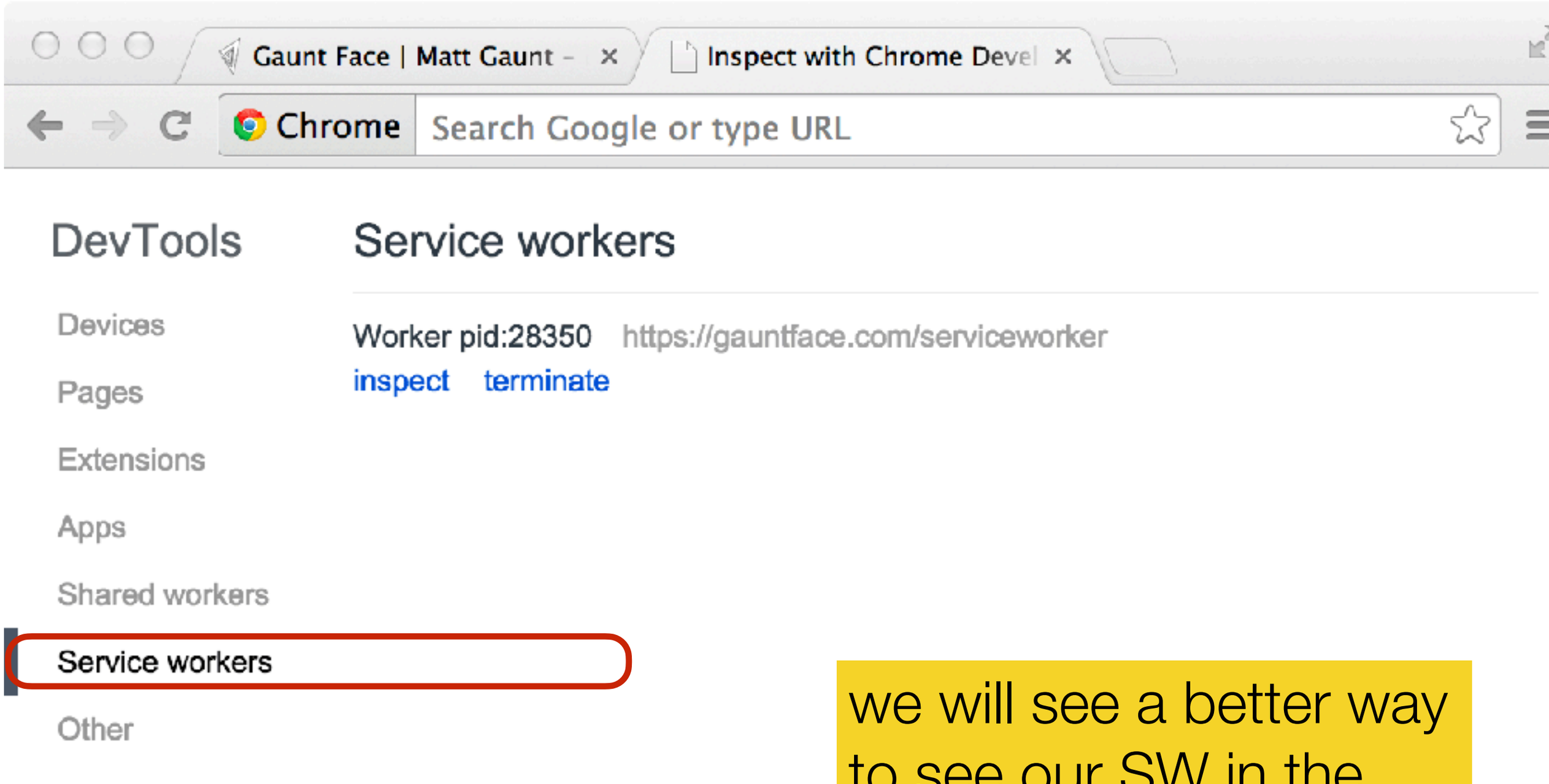
- The register() method shows the location of the service worker file
- The service worker's scope is the part of the file system controlled by its location
  - e.g. in the following example, the SW controls folder 2 and its sublders 2.1 and 2.2
    - root
      - folder 1
        - subfolder 1.1
        - subfolder 1.2
      - folder 2
        - service worker
        - subfolder 2.1
        - subfolder 2.2
- In other words, this service worker will receive fetch events for everything under folder 2 (inclusive)

- If you want the sw to control the entire site, register it a the root of your public directory
  - NOT the views directory as it is a static file and hence it must be in the public directory



See it

- chrome://inspect/#service-workers



we will see a better way to see our SW in the context of our website

Installing a SW

- The typical use of a SW is to cache files
  - Open a cache.
  - Cache our files.
  - Confirm whether all the required assets are cached or not
- The SW is a js file that is loaded by the registration

```
navigator.serviceWorker.register('/sw.js')
```

- In the sw.js file you must
  - define a callback for the install event

The University Of Sheffield.

code in sw.js

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
  // add the files you want to cache here
   '/',
   '/styles/main.css',
   '/script/main.js'
];
```

```
self.addEventListener('install', function(event) {
  // Perform install steps
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
```

```
//event.waitUntil() receives a promise as parameter
// and waits until the promise is fulfilled
// it returns if it succeeded or not
```

Careful here

- If any of the files fails to load none of the files will be cached
  - the service worker will not start either
  - your pages and services will not be available off line
- So be very careful about providing a long list of files
  - as if the loading breaks, your service worker will fail
- The service worker will become available only **after** the user navigates to a different page or refreshes the current page
  - not before!!!

Fetch event

- The fetch event is fired when the browser tries to load a new page from our site
  - the fetch event is captured by the service worker which may decide to serve the cached version of the page instead of fetching it from the network
  - this is the natural continuation of the Ajax idea
    - just go to the server when necessary and only for the information or data that is strictly necessary

```
// when the worker receives a fetch request
self.addEventListener('fetch', function(event) {
  event.respondWith(
   // it checks if the requested page is among the cached ones
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return the cache response (the cached page)
        if (response) {
          return response;
        } //cache does not have the page – go to the server
        return fetch(event.request); )  );});});
```

Caching a retrieved page

- If the page is not among the cached one, the service worker may cache it for you

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.match(event.request)
      .then(function(response) {
        // Cache hit - return response
        if (response) {
          return response;
        }
      // here we will cache the page — see next slide
```

The University Of Sheffield.

ctd

```
// IMPORTANT: booth requests and responses are streams and can only be consumed
// once. Since we are consuming them for both the cache and for the browser,
// we need to clone both the response and the request before using them for the
// second time
        var fetchRequest = event.request.clone();
        return fetch(fetchRequest).then(
          function(response) {
            // Check if we received a valid response. A basic response is one that
           // is made to our own site. Do not cache responses to requests made
           // to other sites
            if(!response || response.status !== 200 || response.type !== 'basic') {
              return response;
            }
            // IMPORTANT: as mentioned we must clone the response.
          //  A response is a stream
            // and because we want the browser to consume the response
            // as well as the cache consuming the response, we need
            // to clone it so we have two streams.
            var responseToCache = response.clone();
            caches.open(CACHE_NAME)
              .then(function(cache) {
                cache.put(event.request, responseToCache);
              });
            return response;
          }
        );       })
```

The University Of Sheffield.

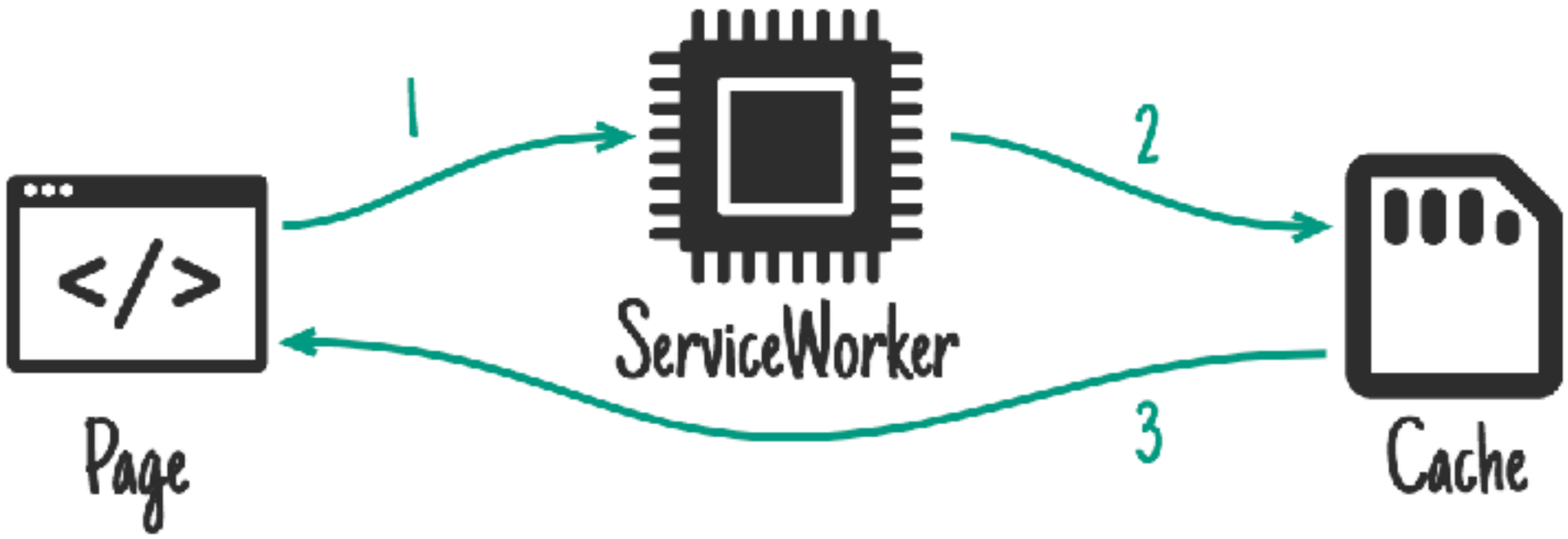
That's it!

- Now we have a service worker that caches pages and can therefore provide our service even when the phone is off-line

Caching strategies

- The strategy above is called CacheOnly
  - The browser will never be able to update the pages afterwards
    - that is because in that code, we never check if the page has been updated
- In reality there are several strategies that we may need to implement depending on the requirements of the site
  - we will now see some of them

Cache only



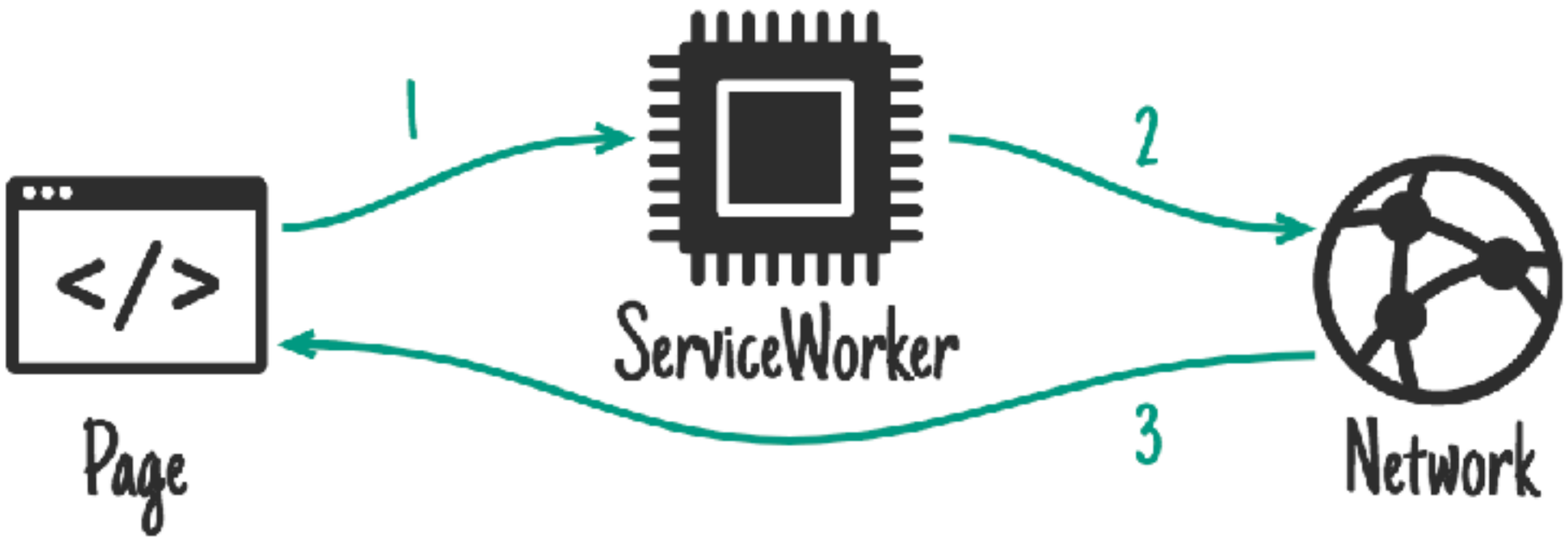
Ideal for: Anything you'd consider static to that "version" of your site. You should have cached these in the install event, so you can depend on them being there.

```
self.addEventListener('fetch', function(event) {
  // If a match isn't found in the cache, the response
  // will look like a connection error
  event.respondWith(caches.match(event.request));
});
```

...although you don't often need to handle this case specifically, "Cache, falling back to network" covers it.

https://jakearchibald.com/2014/offline-cookbook/#on-activate

Network only

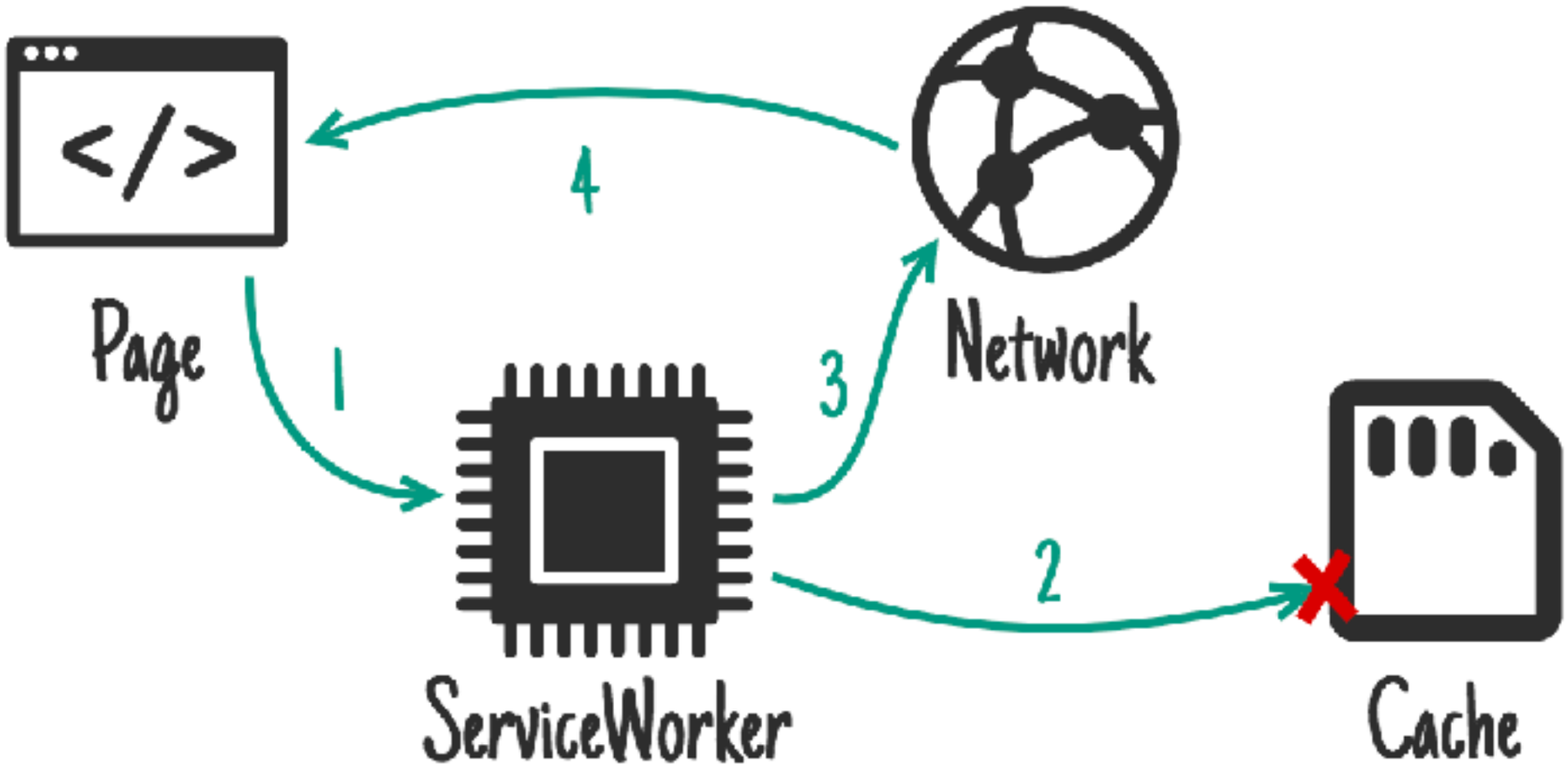


Ideal for: Things that have no offline equivalent, such as analytics pings, non-GET requests.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(fetch(event.request));
  // or simply don't call event.respondWith, which
  // will result in default browser behaviour
});
```

...although you don't often need to handle this case specifically, "Cache, falling back to network" covers it.

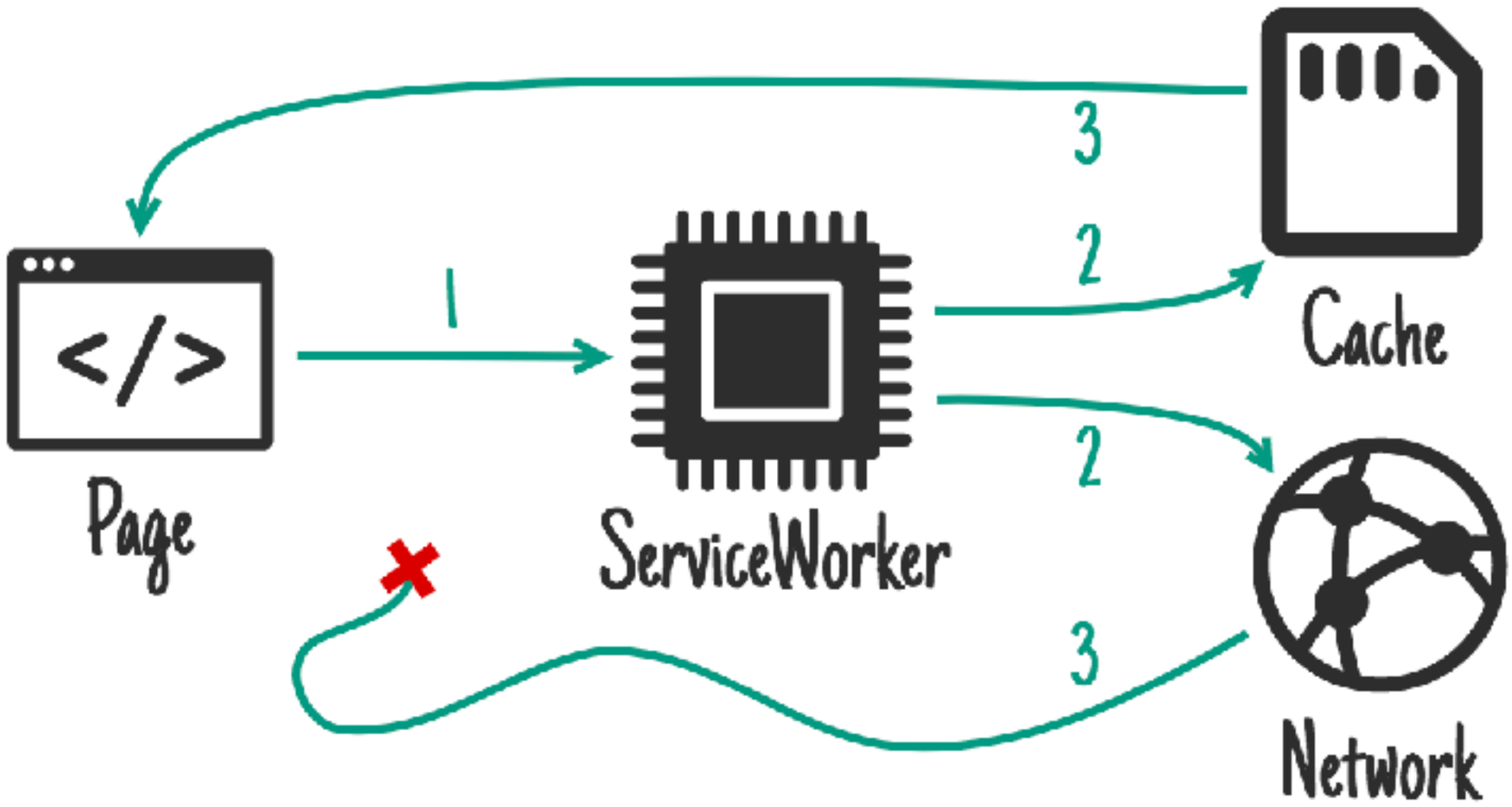
Cache, falling back to network



Ideal for: If you're building offline-first, this is how you'll handle the majority of requests. Other patterns will be exceptions based on the incoming request.

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

Cache & network race



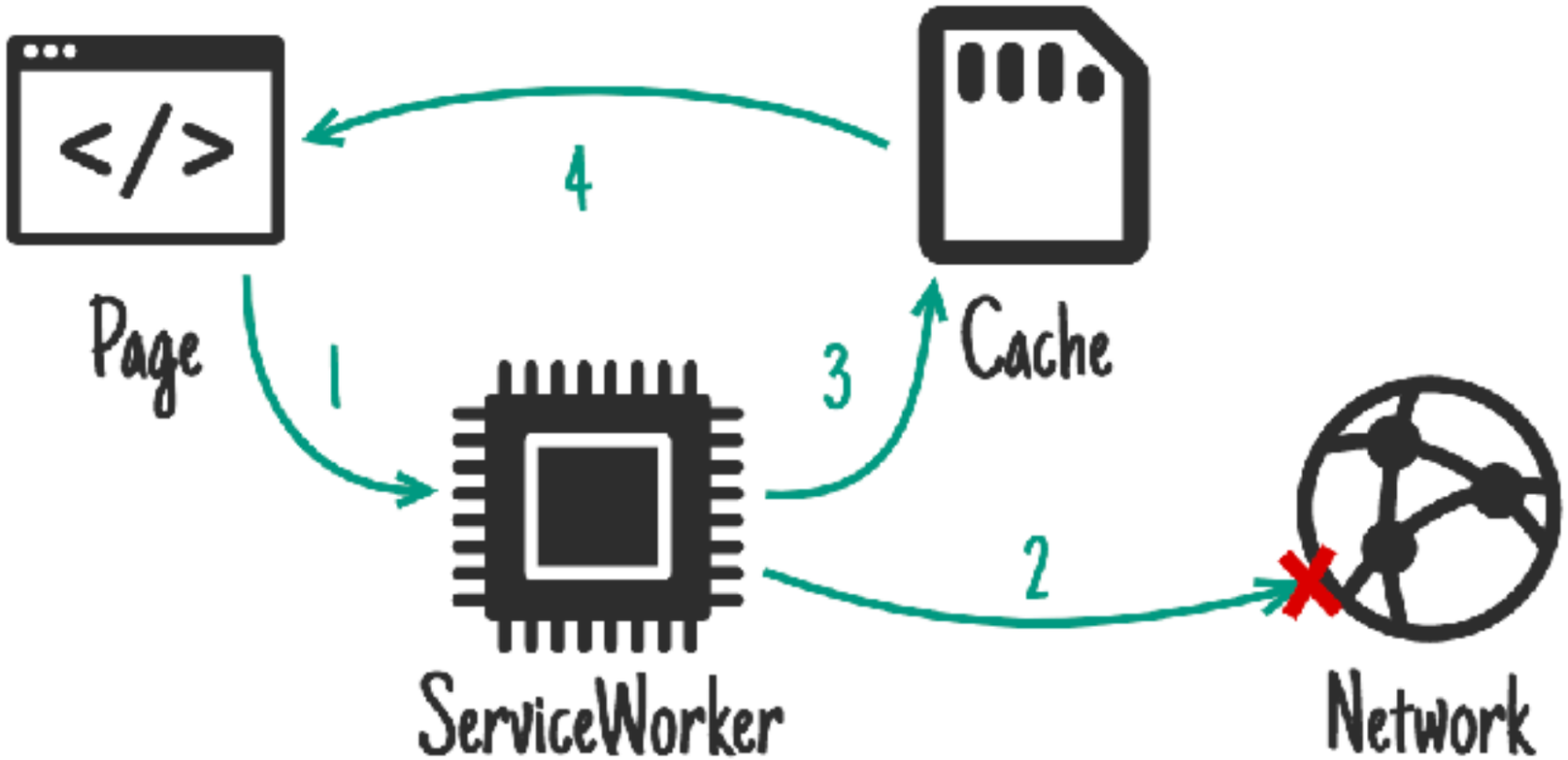
Ideal for: Small assets where you're chasing performance on devices with slow disk access.

With some combinations of older hard drives, virus scanners, and faster internet connections, getting resources from the network can be quicker than going to disk. However, going to the network when the user has the content on their device can be a waste of data, so bear that in mind.

The University Of Sheffield.

```
// Promise.race is no good to us because it rejects if
// a promise rejects before fulfilling. Let's make a p
// race function:
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    // make sure promises are all promises
    promises = promises.map(p => Promise.resolve(p));
    // resolve this promise as soon as one resolves
    promises.forEach(p => p.then(resolve));
    // reject if all promises reject
    promises.reduce((a, b) => a.catch(() => b))
      .catch(() => reject(Error("All failed")));
  });
};

self.addEventListener('fetch', function(event) {
  event.respondWith(
    promiseAny([
      caches.match(event.request),
      fetch(event.request)
    ])
  );
});
```

Network falling back to cache



Ideal for: A quick-fix for resources that update frequently, outside of the "version" of the site. E.g. articles, avatars, social media timelines, game leader boards.

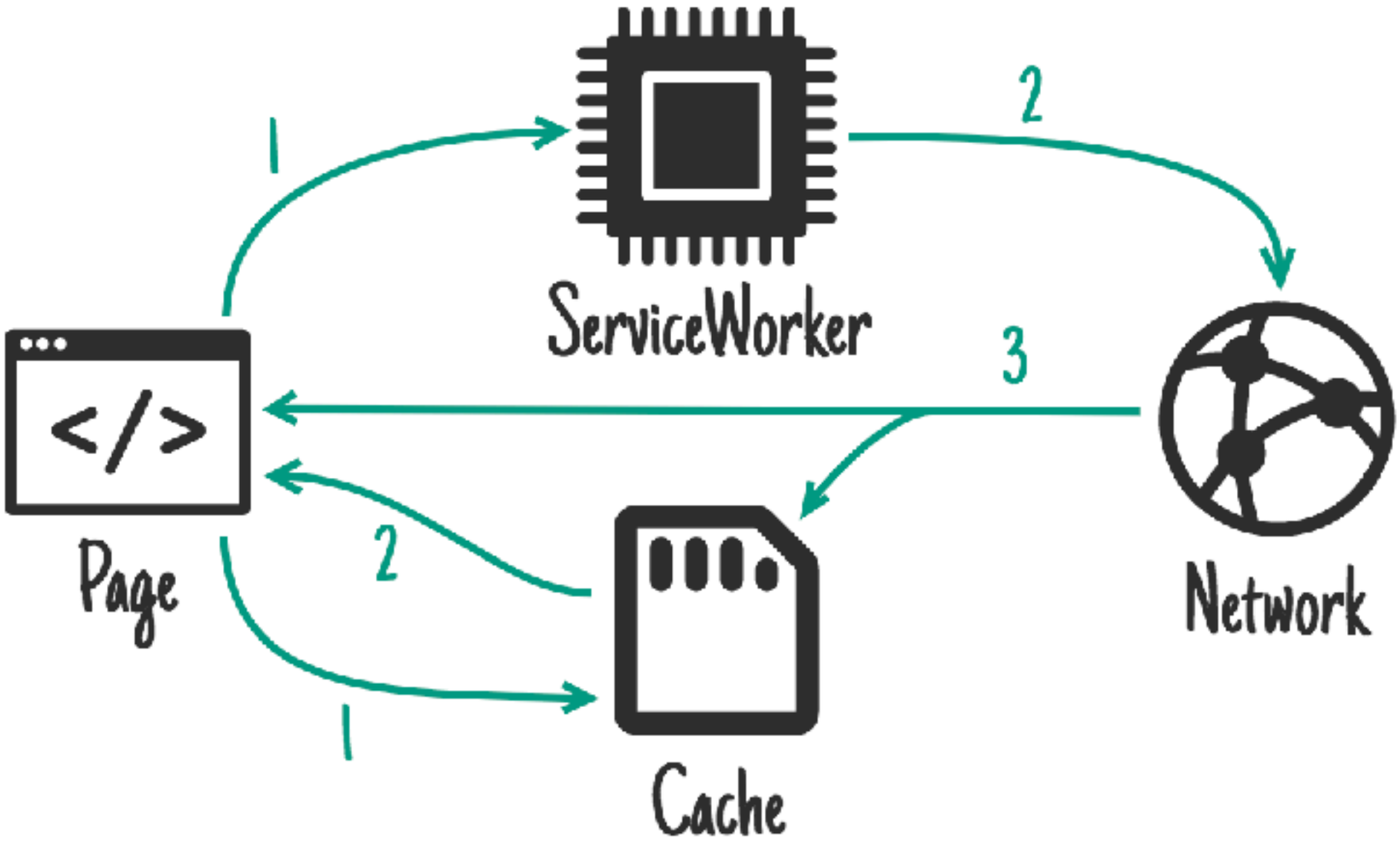
This means you give online users the most up-to-date content, but offline users get an older cached version. If the network request succeeds you'll most-likely want to update the cache entry.

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get the perfectly acceptable content already on their device. This can take an

However, this method has flaws. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get the perfectly acceptable content already on their device. This can take an extremely long time and is a frustrating user experience. See the next pattern, "Cache then network", for a better solution.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function() {
      return caches.match(event.request);
    })
  );
});
```

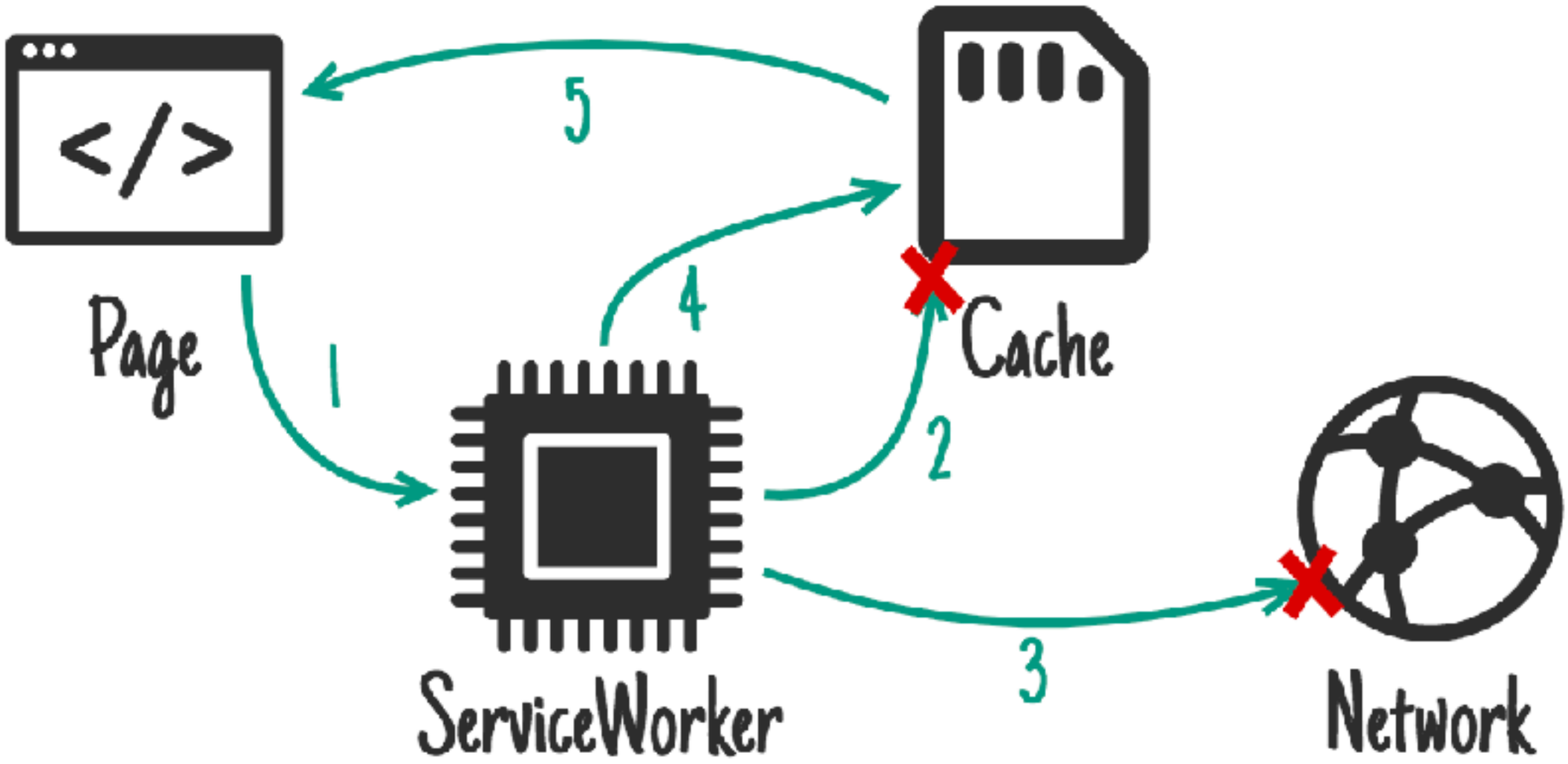
Cache then network



Ideal for: Content that updates frequently. E.g. articles, social media timelines, game leaderboards.

This requires the page to make two requests, one to the cache, one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

Generic fallback



If you fail to serve something from the cache and/or network you may want to provide a generic fallback.

Ideal for: Secondary imagery such as avatars, failed POST requests, "Unavailable while offline" page.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
      // However, in reality you'd have many different
      // fallbacks, depending on URL & headers.
      // Eg, a fallback silhouette image for avatars.
    })
  );
});
```

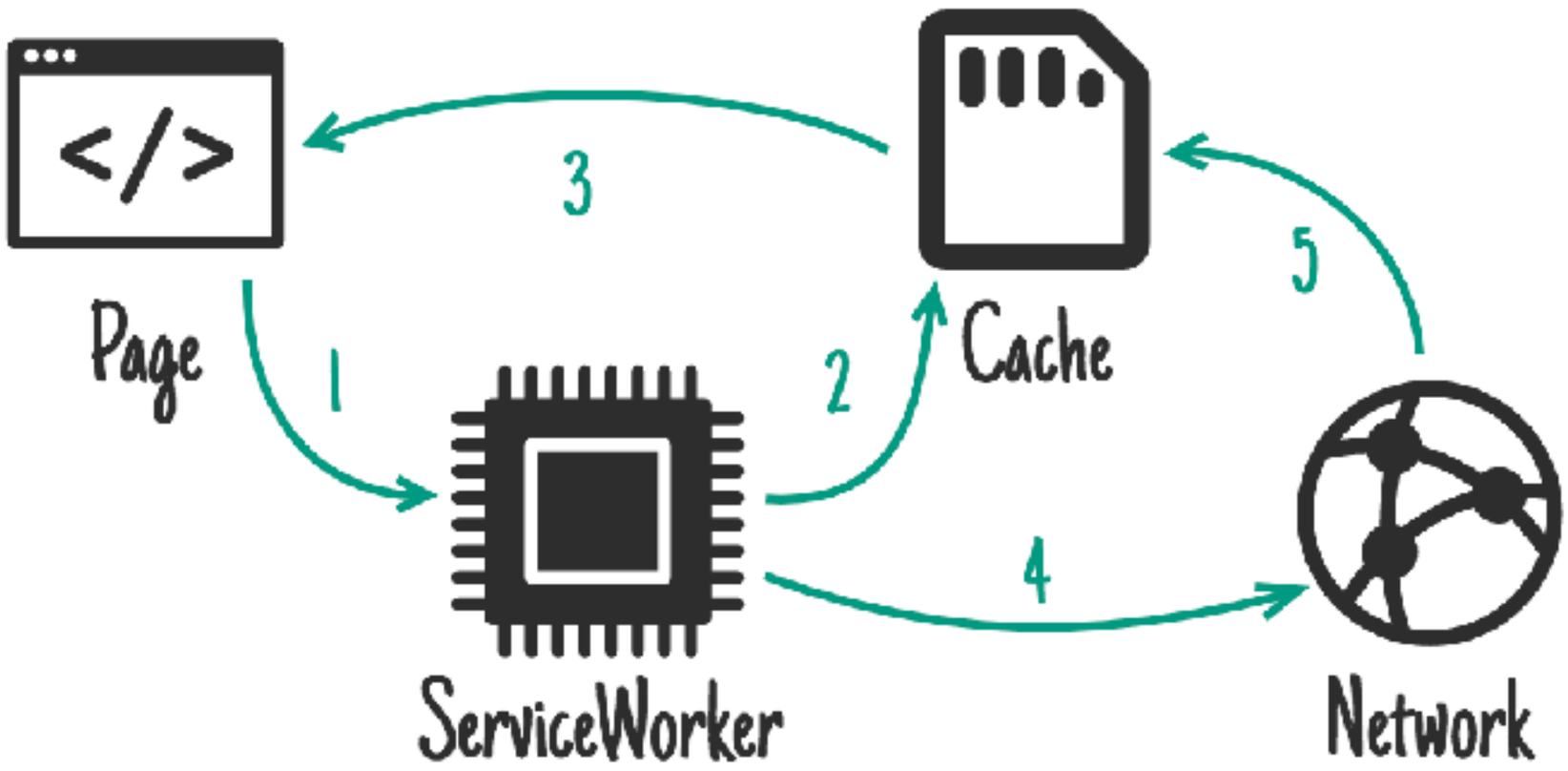
The University Of Sheffield.

Sometimes you can just replace the current data when new data arrives (e.g. game leaderboard), but that can be disruptive with larger pieces of content. Basically, don't "disappear" something the user may be reading or interacting with.

Twitter adds the new content above the old content & adjusts the scroll position so the user is uninterrupted. This is possible because Twitter mostly retains a mostly-linear order to content. I copied this pattern for trained-to-thrill to get content on screen as fast as possible, but still display up-to-date content once it arrives.

Stale-while-revalidate

Stale-while-revalidate



Ideal for: Frequently updating resources where having the very latest version is non-essential. Avatars can fall into this category.

If there's a cached version available, use it, but fetch an update for next time.

https://jakearchibald.com/2014/offline-cookbook/#on-activate

The University Of Sheffield.

Stale-while-revalidate

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open(CACHE_NAME).then(function(cache) {
      return cache.match(event.request)
                  .then(function(response) {
                    var fetchPromise = fetch(event.request)
                      .then(function(networkResponse) {
                        cache.put(event.request,
                                  networkResponse.clone());
                        return networkResponse;
                      })
                    return response || fetchPromise;
      })));
});
```

Progressive Apps

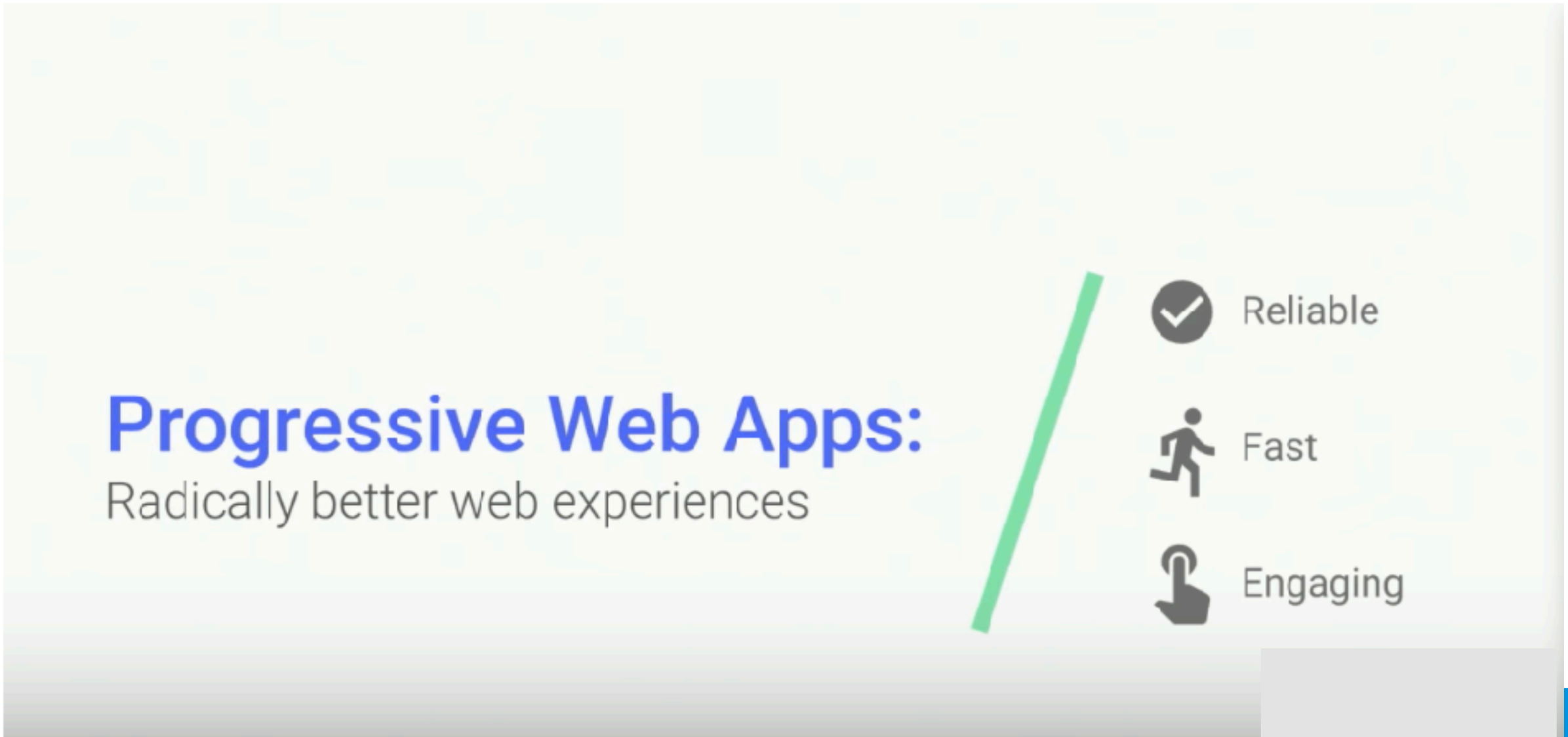
The University Of Sheffield.

remember what we said before

- 53% of users will abandon a site if it takes longer than 3 seconds to load!
  - Once loaded, users expect them to be fast—no janky scrolling or slow-to-respond interfaces
- The real advantage of apps is the ability to
- provide a perfect offline experience
  - Content may not be updated if offline but apps can take opportunistic strategies
    - to download content when online
    - to store content for off line viewing
- allow push notifications even when the user is using a different app

Progressive Web Apps

- Are websites that provide a native app experience without downloading an app
  - they use web service workers



PWAs

https://en.wikipedia.org/wiki/Progressive_web_app

- Progressive - Work for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet.
- Responsive - Fit any form factor: desktop, mobile, tablet, or forms yet to emerge.
- Connectivity independent - Service workers allow work offline, or on low quality networks.
- App-like - Feel like an app to the user with app-style interactions and navigation.
- Fresh - Always up-to-date thanks to the service worker update process.
- Safe - Served via HTTPS to prevent snooping and ensure content hasn't been tampered with.
- Discoverable - Are identifiable as "applications" thanks to W3C manifests[6] and service worker registration scope allowing search engines to find them.
- Re-engageable - Make re-engagement easy through features like push notifications.
- Installable - Allow users to "keep" apps they find most useful on their home screen without the hassle of an app store.

Application shell architecture

- Some progressive web apps use an architectural approach called the App Shell Model
- For rapid loading, service workers store the Basic User Interface or "shell" of the responsive web design web application.
- This shell provides an initial static frame, a layout or architecture into which content can be loaded progressively as well as dynamically,
  - allowing users to engage with the app despite varying degrees of web connectivity.
  - Technically, the shell is a code bundle stored locally in the browser cache of the mobile device

Advantages

- Cost saving in terms of development and maintenance
  - Average customer acquisition costs may be up to ten times smaller compared to those of native applications
  - (this claim is disputable but it is good to understand the potential impact)
- Always up to date
  - one of the major issues of apps is that the average user does not update native apps
    - in my experience 30% to 50% of users do not
    - this is a serious issue in terms of effectiveness and ability to fix issues

The University Of Sheffield.

PWAs

Progressive Web Apps are user experiences that have the reach of the web, and are:

- **Reliable** - Load instantly and never show the downasaur, even in uncertain network conditions.
- **Fast** - Respond quickly to user interactions with silky smooth animations and no janky scrolling.
- **Engaging** - Feel like a natural app on the device, with an immersive user experience.

This new level of quality allows Progressive Web Apps to earn a place on the user's home screen.

https://developers.google.com/web/progressive-web-apps/

Based on Web Service Workers

Reliable

When launched from the user's home screen, service workers enable a Progressive Web App to load instantly, regardless of the network state.

A service worker, written in JavaScript, is like a client-side proxy and puts you in control of the cache and how to respond to resource requests. By pre-caching key resources you can eliminate the dependence on the network, ensuring an instant and reliable experience for your users.

LEARN MORE

No app store

Engaging

Progressive Web Apps are installable and live on the user's **home screen**, without the need for an app store. They offer an **immersive full screen** experience with help from a web app manifest file and can even re-engage users with web **push notifications**.

The Web App Manifest allows you to control how your app appears and how it's launched. You can specify home screen icons, the page to load when the app is launched, screen orientation, and even whether or not to show the browser chrome.

WEB APP MANIFEST WEB PUSH NOTIFICATIONS

The University Of Sheffield.

Why build a Progressive Web App?

Building a high-quality Progressive Web App has incredible benefits, making it easy to delight your users, grow engagement and increase conversions.

✓ **Worthy of being on the home screen**

When the Progressive Web App criteria are met, Chrome prompts users to add the Progressive Web App to their home screen.

✓ **Work reliably, no matter the network conditions**

Service workers enabled Konga to send 63% less data for initial page loads, and 84% less data to complete the first transaction!

✓ **Increased engagement**

Web push notifications helped eXtra Electronics increase engagement by 4X. And those users spend twice as much time on the site.

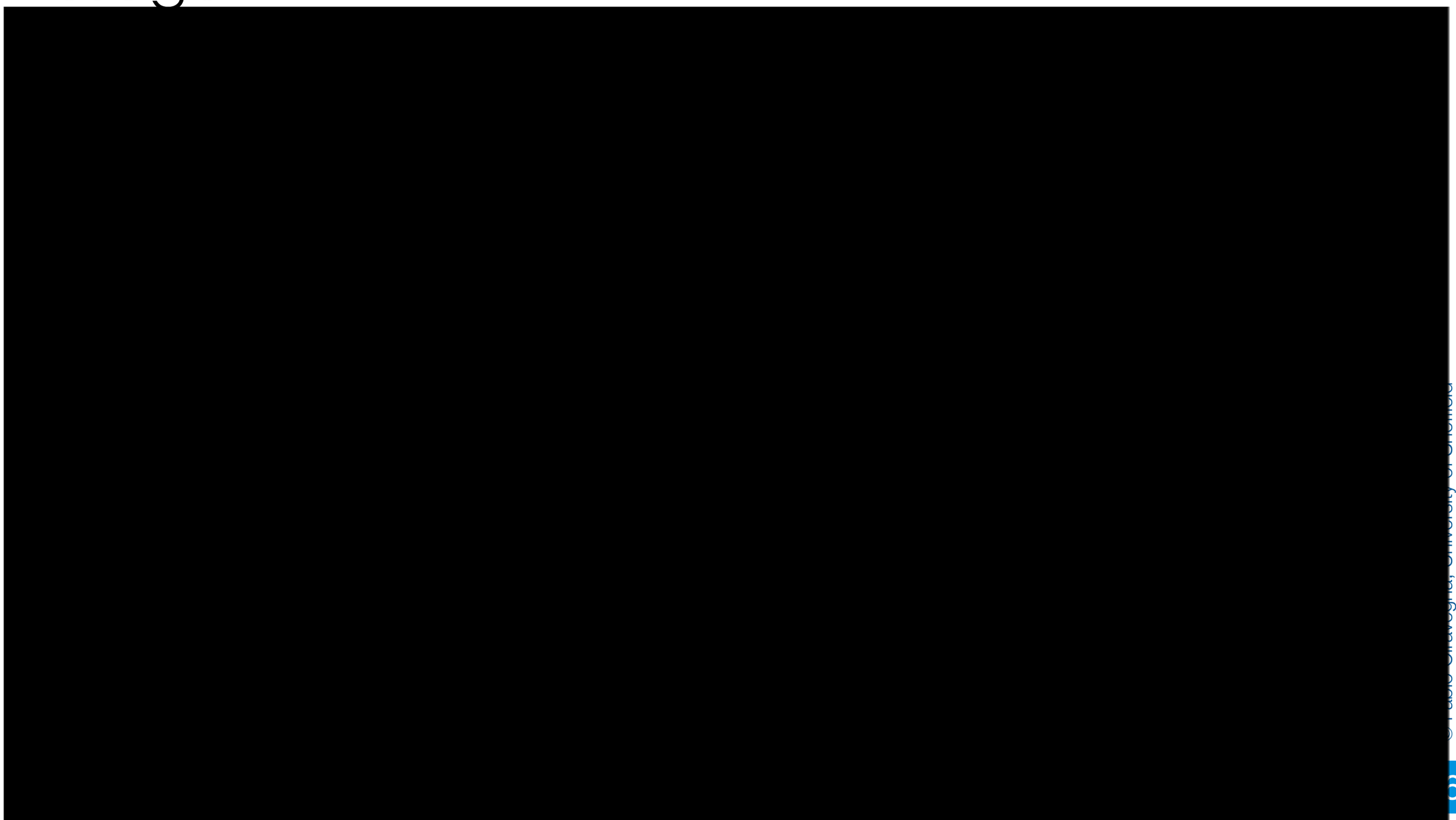
✓ **Improved conversions**

The ability to deliver an amazing user experience helped AliExpress improve conversions for new users across all browsers by 104% and on iOS by 82%.

The University Of Sheffield.

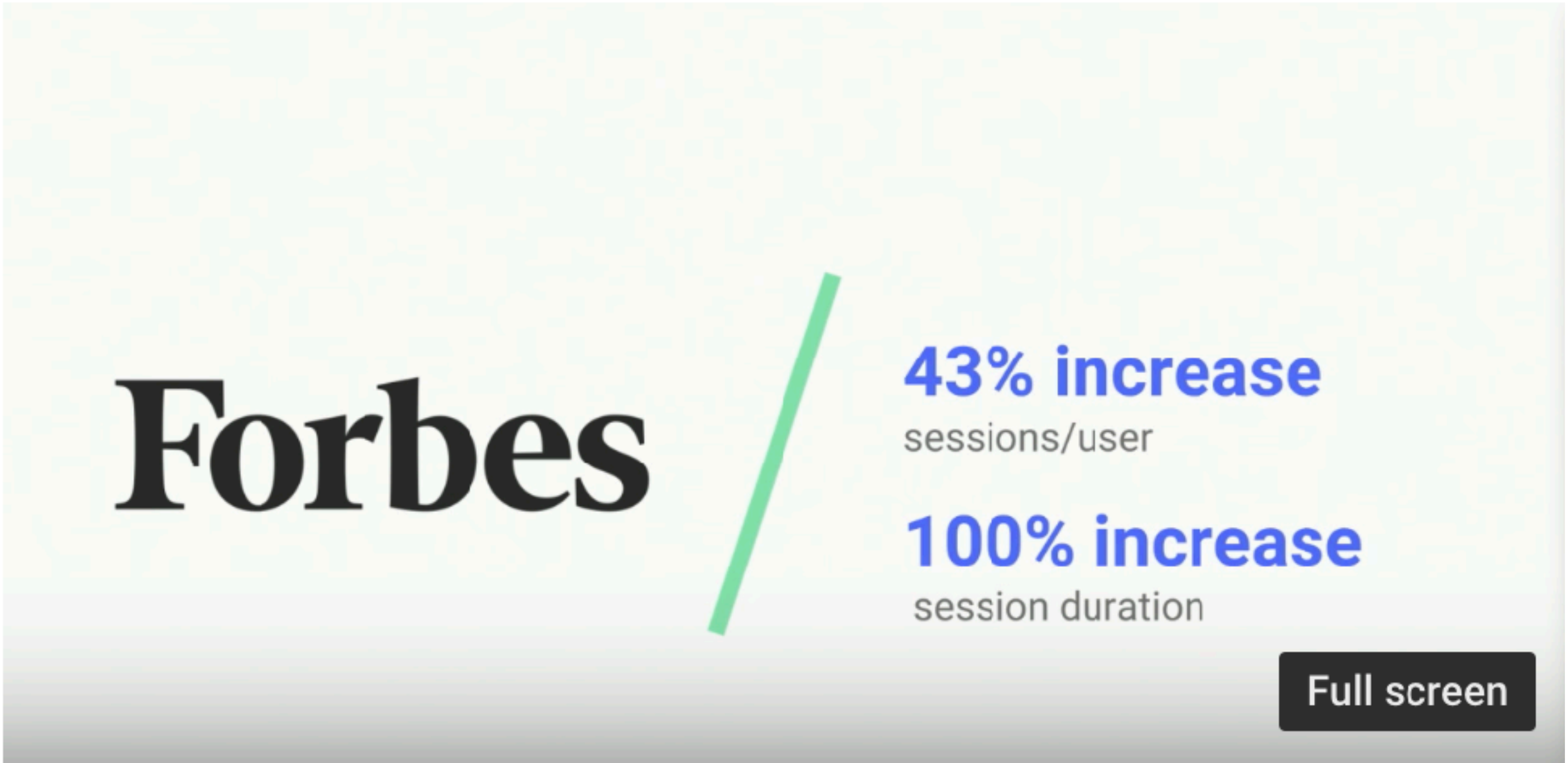
Not just a web site

- A fully fledged app with notifications and offline usage



Why?

- The point is to make sure people come and access your service (rather than website)
  - that is the whole point of having apps. With a progressive app:



Manifest file

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

- A manifest file is a file containing metadata for a group of accompanying files that are part of a set or coherent unit
  - For example, the files of a computer program may have a manifest describing the name, version number, license and the constituting files of the program
- This is a generic concept that is specialised by the Web Manifest
  - The web app manifest provides information about an application (such as name, author, icon, and description) in a JSON text file.
  - The purpose of the manifest is to install web applications to the homescreen of a device, providing users with quicker access and a richer experience.

The University Of Sheffield.

Example

```
{  "name": "HackerWeb",
  "short_name": "HackerWeb",
  "start_url": ".",
  "display": "standalone",
  "background_color": "#fff",
  "description": "A simply 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"
  }]
```

Minimum Requirements

- The name of the web application
- Links to the web app icons or image objects
- The preferred URL to launch or open the web app
- The web app configuration data for a number of characteristics
- Declaration for default orientation of the web app (e.g. portrait)
- Enables to set the display mode e.g. full screen

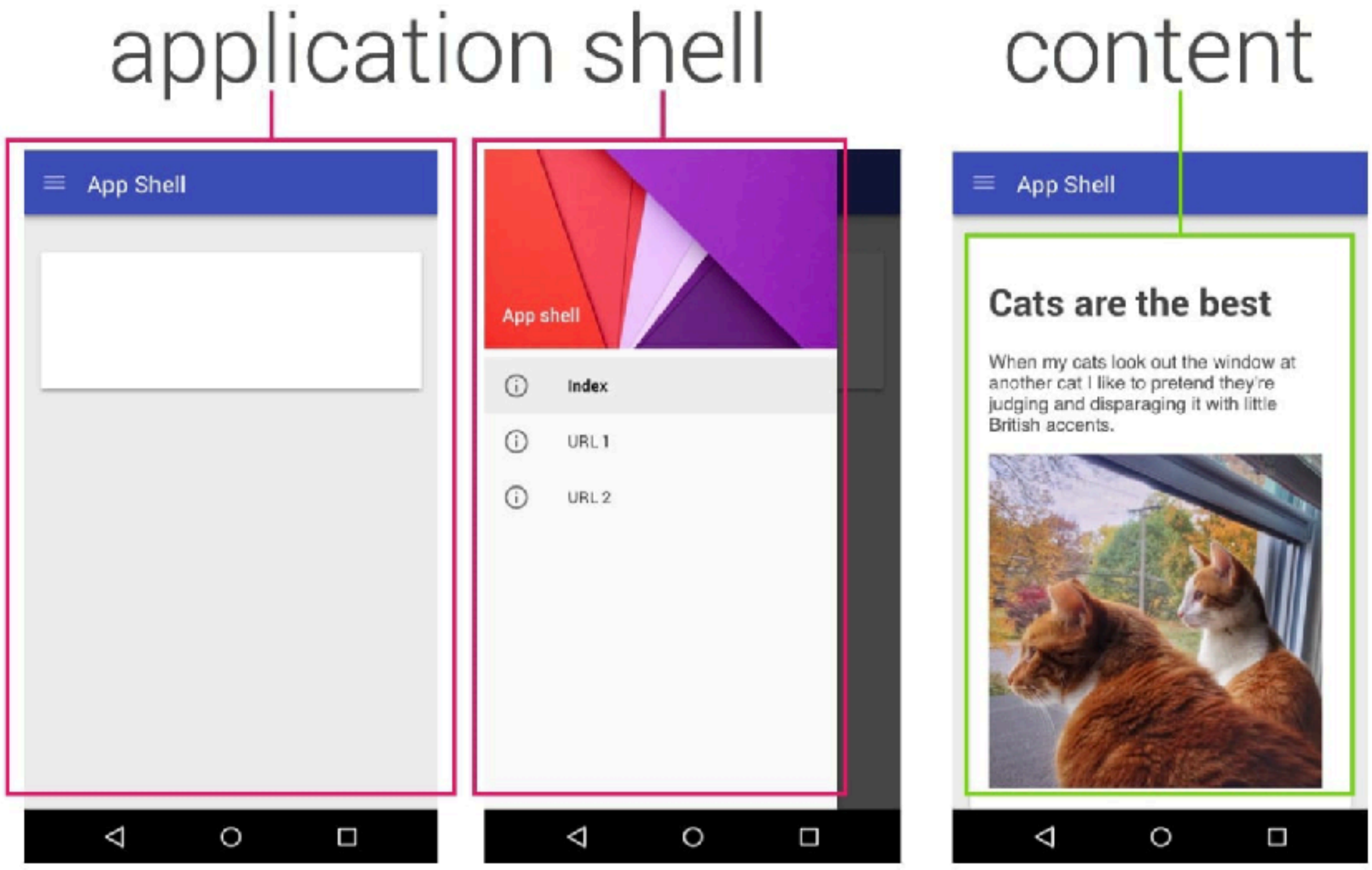
App Shell

- The app's shell is the minimal HTML, CSS, and JavaScript that is required to power the user interface of PWa
  - the components that ensures reliably good performance also in limited connectivity conditions
  - Its first load should be extremely quick and immediately cached
    - "Cached" means that the shell files are loaded once over the network and then saved to the local device.
  - Every subsequent time that the user opens the app, the shell files are loaded from the local device's cache, which results in blazing-fast startup times.

https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/

- App shell architecture separates the core application infrastructure and UI from the data
- All of the UI and infrastructure is cached locally using a service worker
  - so that on subsequent loads, the Progressive Web App only needs to retrieve the necessary data
- For example a Facebook like app will have
  - as shell the interface containing the news feeds and the different pages (profile, etc.
  - as cashed data the news you have already downloaded, your profile information etc.
  - as downloaded data, the updates of news, to profile, etc,

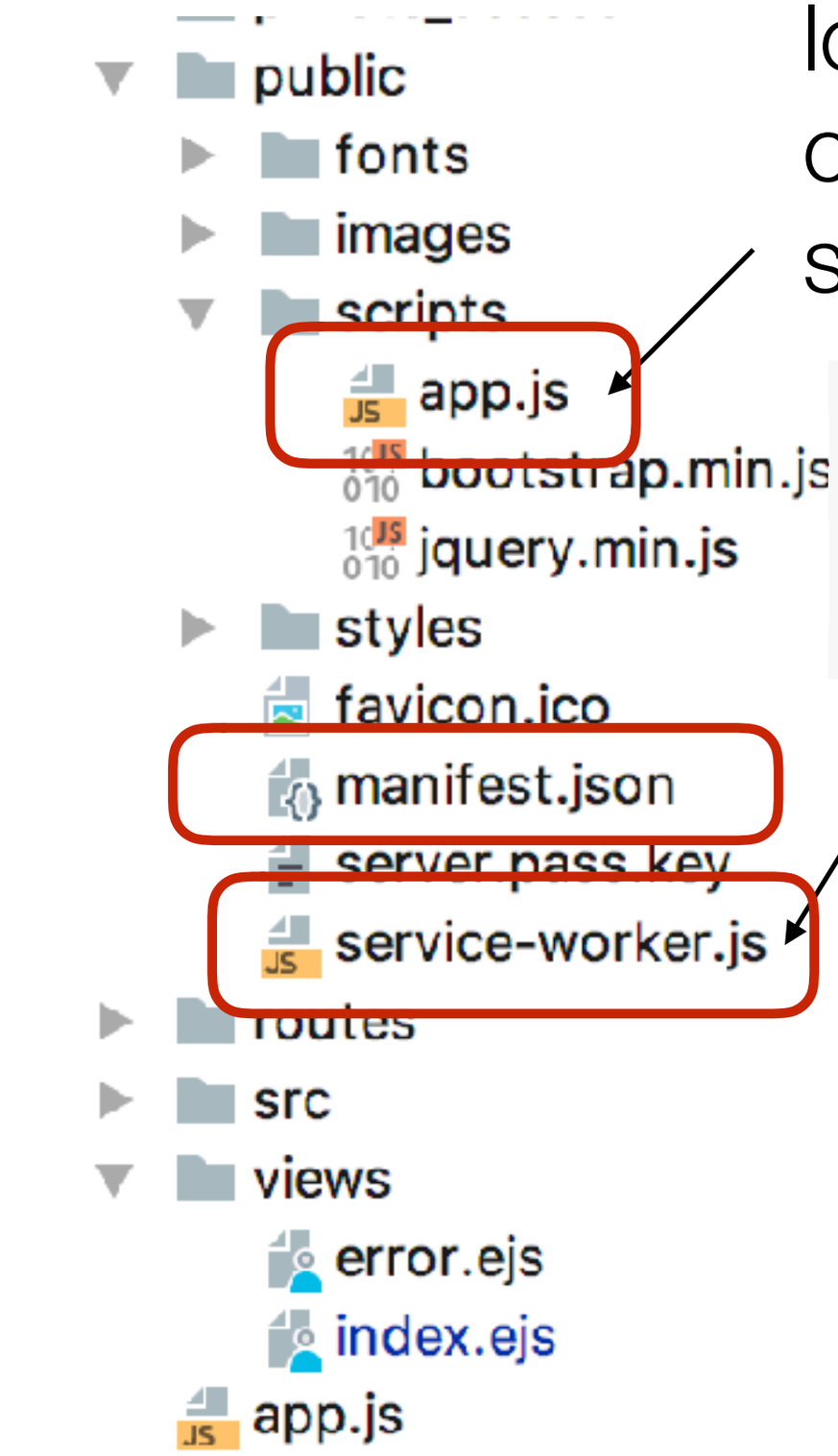
Example



Designing the shell

- The first step is to break the design down into its core components.
- Ask yourself:
- What needs to be on screen immediately?
- What other UI components are key to our app?
- What supporting resources are needed for the app shell?
  - For example images, JavaScript, styles, etc.

App organisation



loaded by index.ejs, it will contain the declaration of the service worker

```
if ('serviceWorker' in navigator) {
   navigator.serviceWorker
            .register('./service-worker.js')
            .then(function() {
                console.log('Service Worker Registered');
 }
```

The service worker: it contains the list of shell files and the install&fetch events capturing declarations

```
var cacheName = 'weatherPWA-step-6-1';
var filesToCache = [...];

self.addEventListener('install', function(e) {
  console.log('[ServiceWorker] Install');
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      console.log('[ServiceWorker] Caching app shell');
      return cache.addAll(filesToCache);
```

```
var cacheName = 'weatherPWA-step-8-1';
var filesToCache = [
  '/',
  '/scripts/app.js',
  '/styles/inline.css',
 'styles/bootstrap.min.css',
 'scripts/bootstrap.min.js',
 'scripts/jquery.min.js',
'/fonts/glyphicons-halflings-regular.woff2',
'/fonts/glyphicons-halflings-regular.woff',
'/fonts/glyphicons-halflings-regular.ttf',
];

self.addEventListener('install', function(e) {
  console.log('[ServiceWorker] Install');
  e.waitUntil(
    caches.open(cacheName).then(function(cache) {
      console.log('[ServiceWorker] Caching app shell');
      return cache.addAll(filesToCache);
    })
  );
});

self.addEventListener('activate', function(e) {
  console.log('[ServiceWorker] Activate');
  e.waitUntil(
    caches.keys().then(function(keyList) {
      return Promise.all(keyList.map(function(key) {
        if (key !== cacheName && key !== dataCacheName) {
          console.log('[ServiceWorker] Removing old cache', key);
          return caches.delete(key);
        }
      }));
```


```
self.addEventListener('fetch', function(e) {
  console.log('[Service Worker] Fetch', e.request.url);
      /*
      * The app is asking for app shell files. In this scenario the app uses
the
      * "Cache, falling back to the network" offline strategy:
      * https://jakearchibald.com/2014/offline-cookbook/#cache-falling-back-
to-network
      */
  e.respondWith(
    caches.match(e.request).then(function(response) {
      return response
          || fetch(e.request)
              .then(function(response) {
               // note if network error happens, fetch does not return
              // an error. it just returns response not ok
             // https://www.tjvantoll.com/2015/09/13/fetch-and-errors/
                  if (!response.ok) {
                      console.log("error: " + err);
                  }
               }) // here we capture HTTP errors such as 404 file not found
               .catch(function (e) {
                  console.log("error: " + err);
               })
    })
```

Working with HTTPS

https

- HTTPS protects the integrity of your website
  - HTTPS helps prevent intruders from tampering with the communications between your websites and your users' browsers.
    - Intruders include intentionally malicious attackers, and legitimate but intrusive companies, such as ISPs or hotels that inject ads into pages
- HTTPS protects the privacy and security of your users
  - HTTPS prevents intruders from being able to passively listen to communications between your websites and your users.
- HTTPS is the future of the web
  - Powerful, new web platform features require explicit permission from the user before executing
    - e.g. taking pictures or recording audio with getUserMedia(), (see lecture 6)
    - enabling offline app experiences with service workers, or building progressive web apps
- To add HTTPS to your server you'll need to get a TLS certificate and set it up for your server
  - This varies depending on your setup, so check your server's documentation and be sure to check out Mozilla's SSL config generator for best practices.

https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https

Add https to nodeJs

- in bin/www
  - change

```
var http = require('http');
var server = http.createServer(app);
```

  - into:

```
var https = require('https');
var options = {
        // the private key
    key: fs.readFileSync('./private/key.pem'),
        // The Certificate Signing Request (csr)
    cert: fs.readFileSync('./private/bundle.crt')
};
/**
 * Create HTTPs server using the options
 */
var server = https.createServer(options, app);
```

SSL connection

- An SSL connection between a client and server is set up by a handshake, the goals of which are:
  - To satisfy the client that it is talking to the right server (and optionally visa versa)
  - For the parties to have agreed on a "cipher suite", which includes which encryption algorithm they will use to exchange data
  - For the parties to have agreed on any necessary keys for this algorithm
- Once the connection is established, both parties can use the agreed algorithm and keys to securely send messages to each other

https://robertheaton.com/2014/03/27/how-does-https-actually-work/

- 3 main phases
  - Hello - The handshake begins with the client sending a ClientHello message.
    - This contains all the information the server needs in order to connect to the client via SSL, including the various cipher suites and maximum SSL version that it supports.
    - The server responds with a ServerHello, which contains similar information required by the client
  - Certificate Exchange - Now that contact has been established, the server has to prove its identity to the client
    - This is achieved using its SSL certificate, which is a very tiny bit like its passport
    - An SSL certificate contains various pieces of data, including the name of the owner, the property (eg. domain) it is attached to, the certificate's public key, the digital signature and information about the certificate's validity dates
    - The client checks that it either implicitly trusts the certificate, or that it is verified and trusted by one of several Certificate Authorities (CAs) that it also implicitly trusts.

- Key Exchange -
  - The encryption of the actual message data exchanged by the client and server will be done using a symmetric algorithm agreed during the Hello phase.
  - A symmetric algorithm uses a single key for both encryption and decryption, in contrast to asymmetric algorithms that require a public/private key pair.
  - Both parties need to agree on this single, symmetric key, a process that is accomplished securely using asymmetric encryption and the server's public/ private keys.
- The client generates a random key to be used for the main, symmetric algorithm.
  - It encrypts it using an algorithm also agreed upon during the Hello phase, and the server's public key (found on its SSL certificate).
  - It sends this encrypted key to the server, where it is decrypted using the server's private key, and the interesting parts of the handshake are complete.
- The parties are sufficiently happy that they are talking to the right person, and have secretly agreed on a key to symmetrically encrypt the data that they are about to send each other. HTTP requests and responses can now be sent by forming a plaintext message and then encrypting and sending it.
- The other party is the only one who knows how to decrypt this message, and so Man In The Middle Attackers are unable to read or modify any requests that they may intercept.

Self signed certificates

- For debugging purposes you can generate a self signed certificate
  - NOT for public use
  - will not be trusted by the browsers
- Get an official certificate by a trusted provider for production (e.g. goDaddy)

Self Signed Certificate

https://www.mobilefish.com/services/ssl_certificates/ssl_certificates.php

Certificate signing request, certificate and private key settings:

Country Name [C]*:	United Kingdom
State or Province Name (full name) [ST]*:	England
Locality Name (eg, city) [L]*:	Sheffield
Organization Name (eg, company) [O]:	The University of Sheffield
Organizational Unit Name (eg, section) [OU]:	Computer Science
Common Name (eg, your name, domain name or ip address) [CN]*:	localhost
Email Address:	f.ciravegna@shef.ac.uk
Protect the private key with a passphrase. Enter the passphrase:	whateverworksforyou
Characters entered:	19
RSA key bit length *:	2048
Number of days the certificate is valid *:	365
To prevent automated submissions an Access Code has been implemented for this tool.	Y0i
Please enter the Access Code as displayed above *:	Y0i

* = required

Generate | Clear all | Demo Info

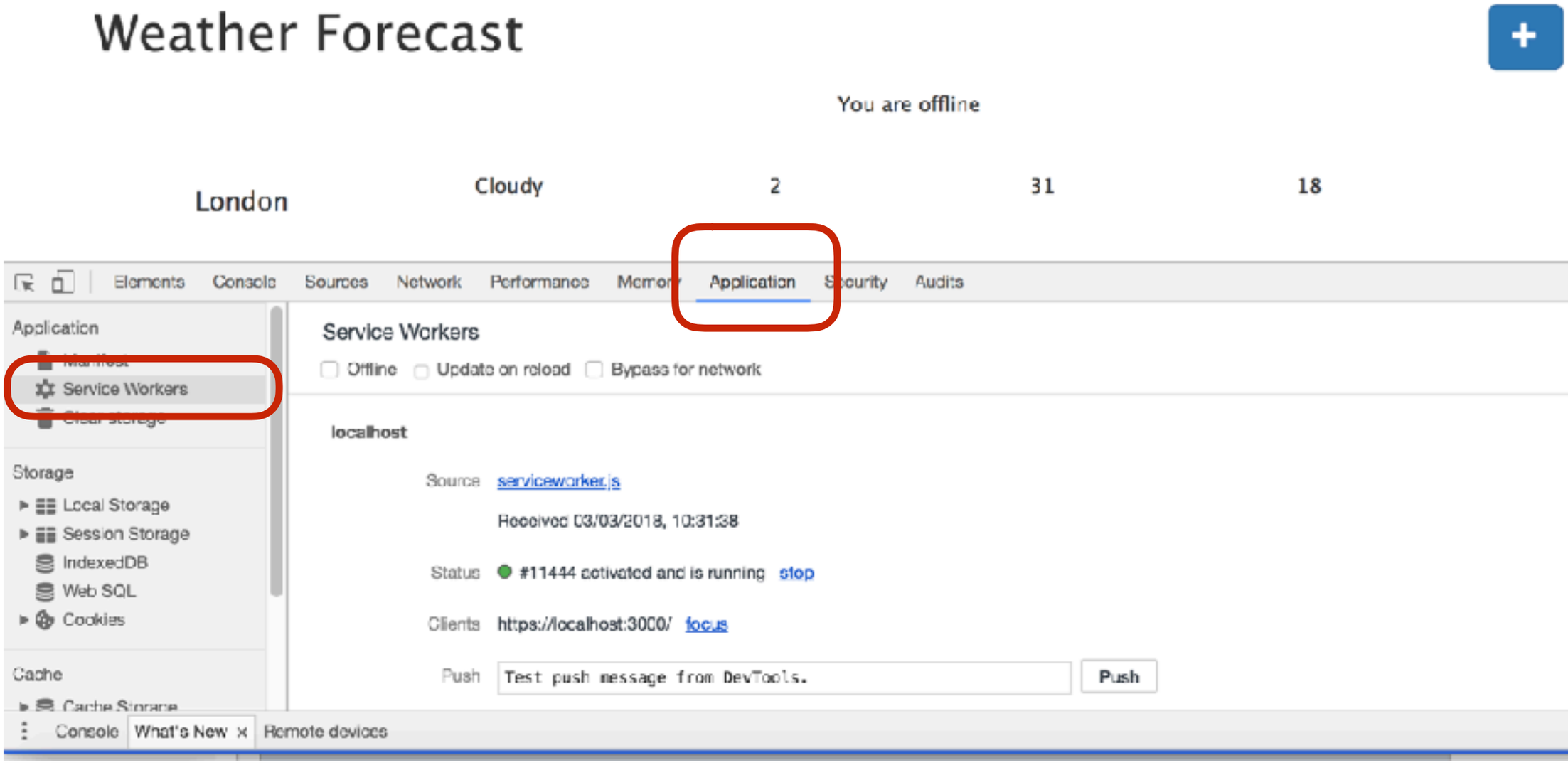
testing with self certificate

- http://deanhume.com/home/blogpost/testing-service-workers-locally-with-self-signed-certificates/10155
- Launch a new version of Chrome with limited security
- on a MAC:
  - /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome --user-data-dir=/tmp/foo --ignore-certificate-errors --unsafely-treat-insecure-origin-as-secure=https://localhost:3000
- on Windows
  - path to Chrome plus --ignore-certificate-errors --unsafely-treat-insecure-origin-as-secure=https://=https://localhost:3000
  - e.g.
  - C:\Program Files (x86)\Google\Chrome\Application\chrome.exe --ignore-certificate-errors --unsafely-treat-insecure-origin-as-secure=https://=https://localhost:3000

Debugging PWAs

Debugging

- Open Chrome development tools for your web app



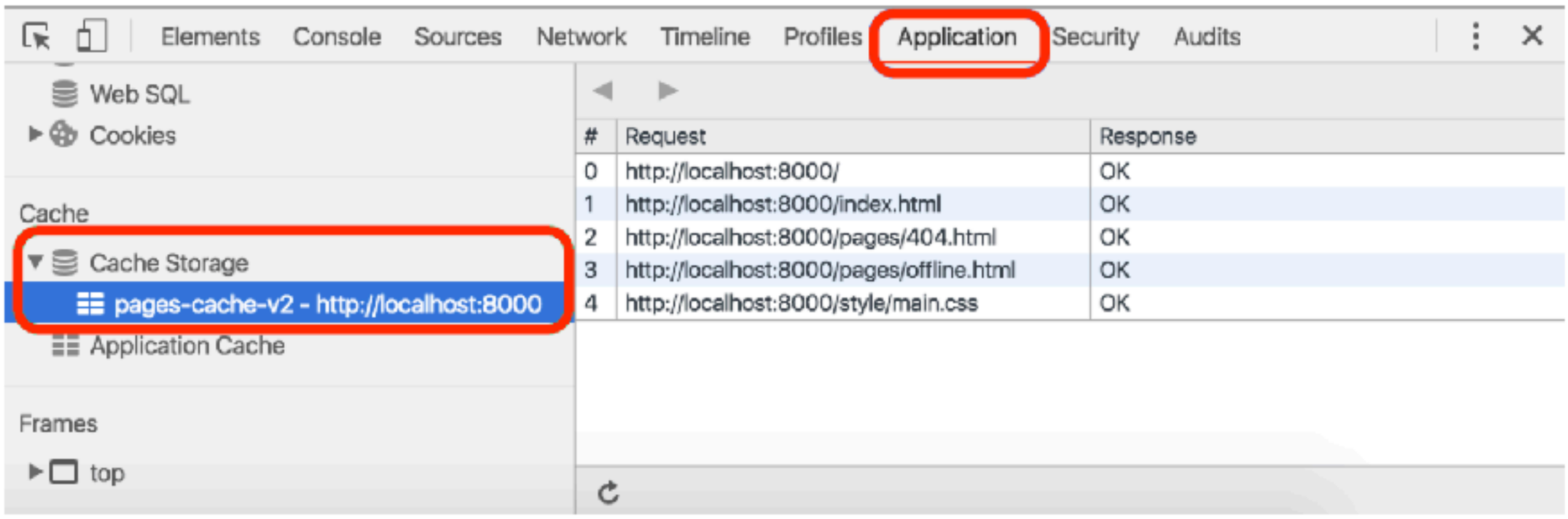
Inspecting cache

Inspect cache storage

Check the service worker cache

Chrome

Open **DevTools** and select the **Application** panel. In the navigation bar click **Cache Storage** to see a list of caches. Click a specific cache to see the resources it contains.

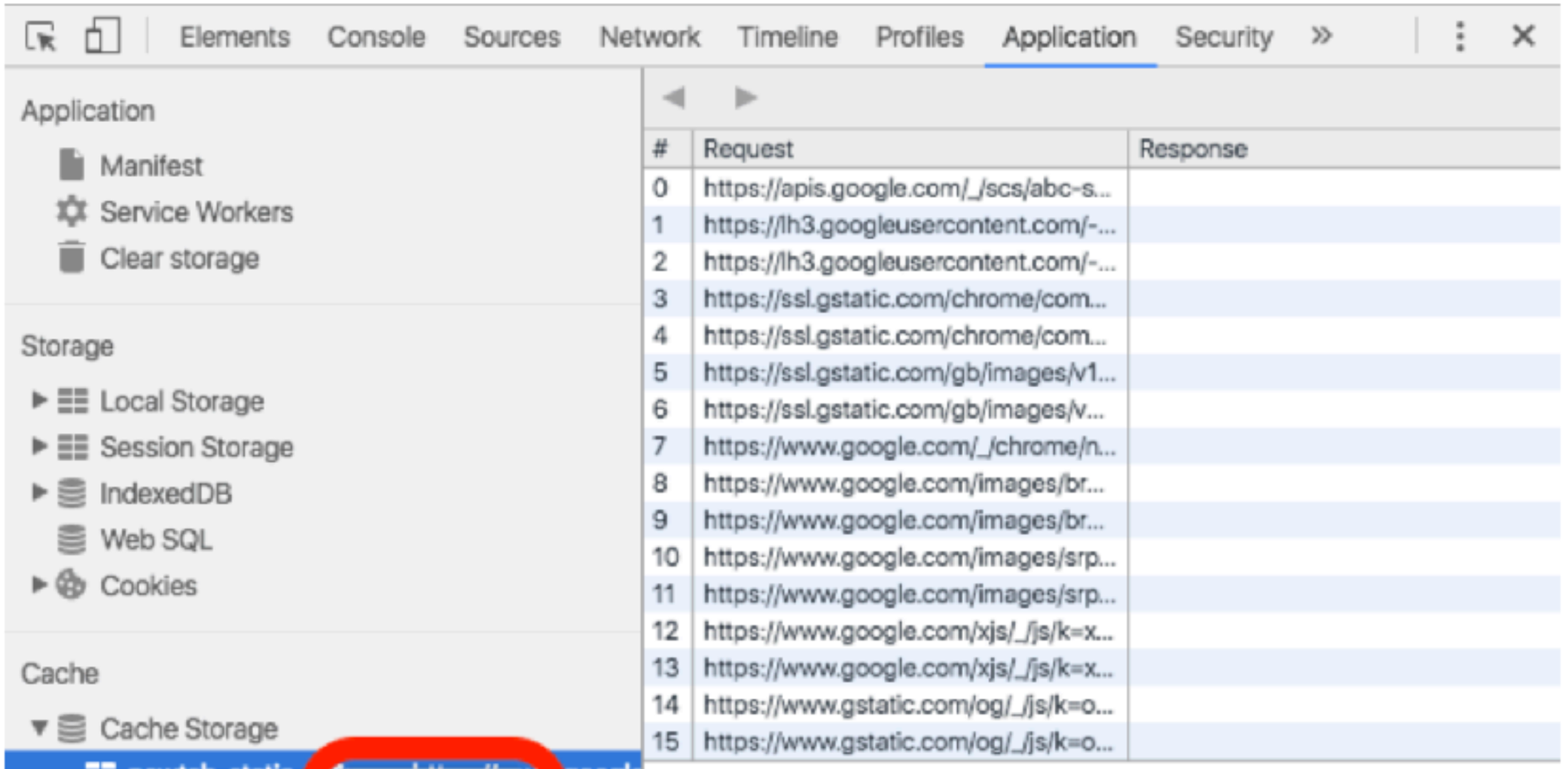


Clear Cache

Clear the service worker cache

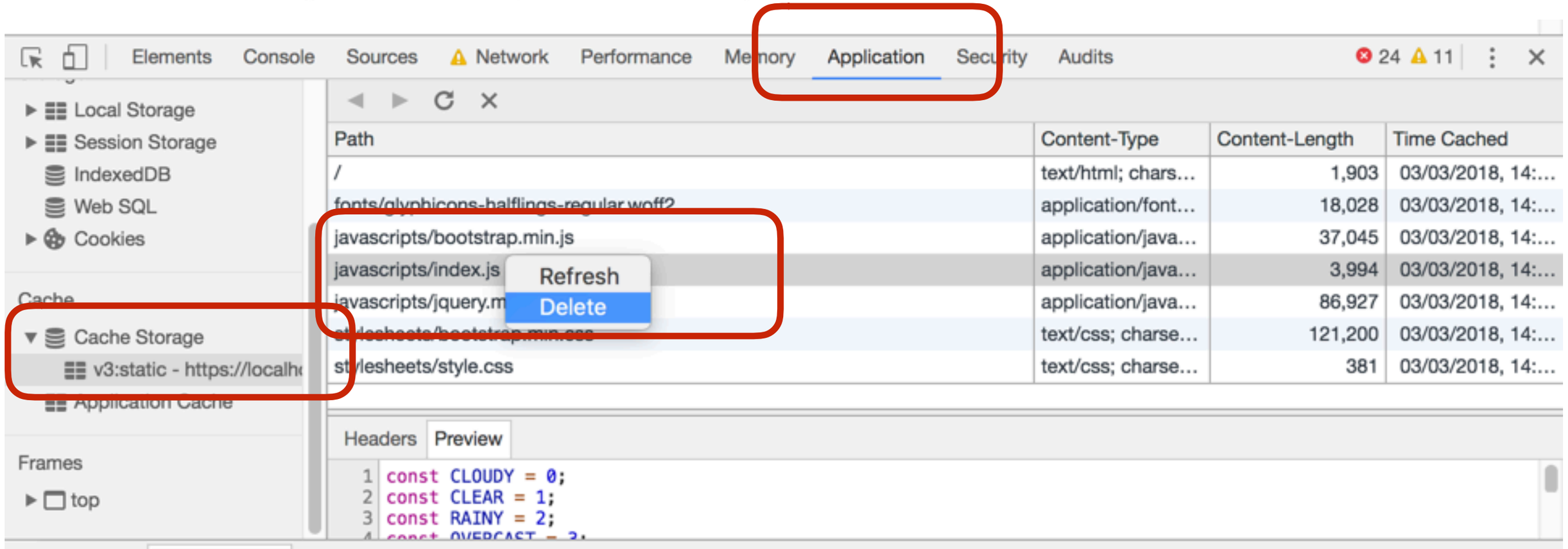
Chrome

Go to **Cache Storage** in **DevTools**. In the **Application** panel, expand **Cache Storage**. Right-click the cache name and then select **Delete**.

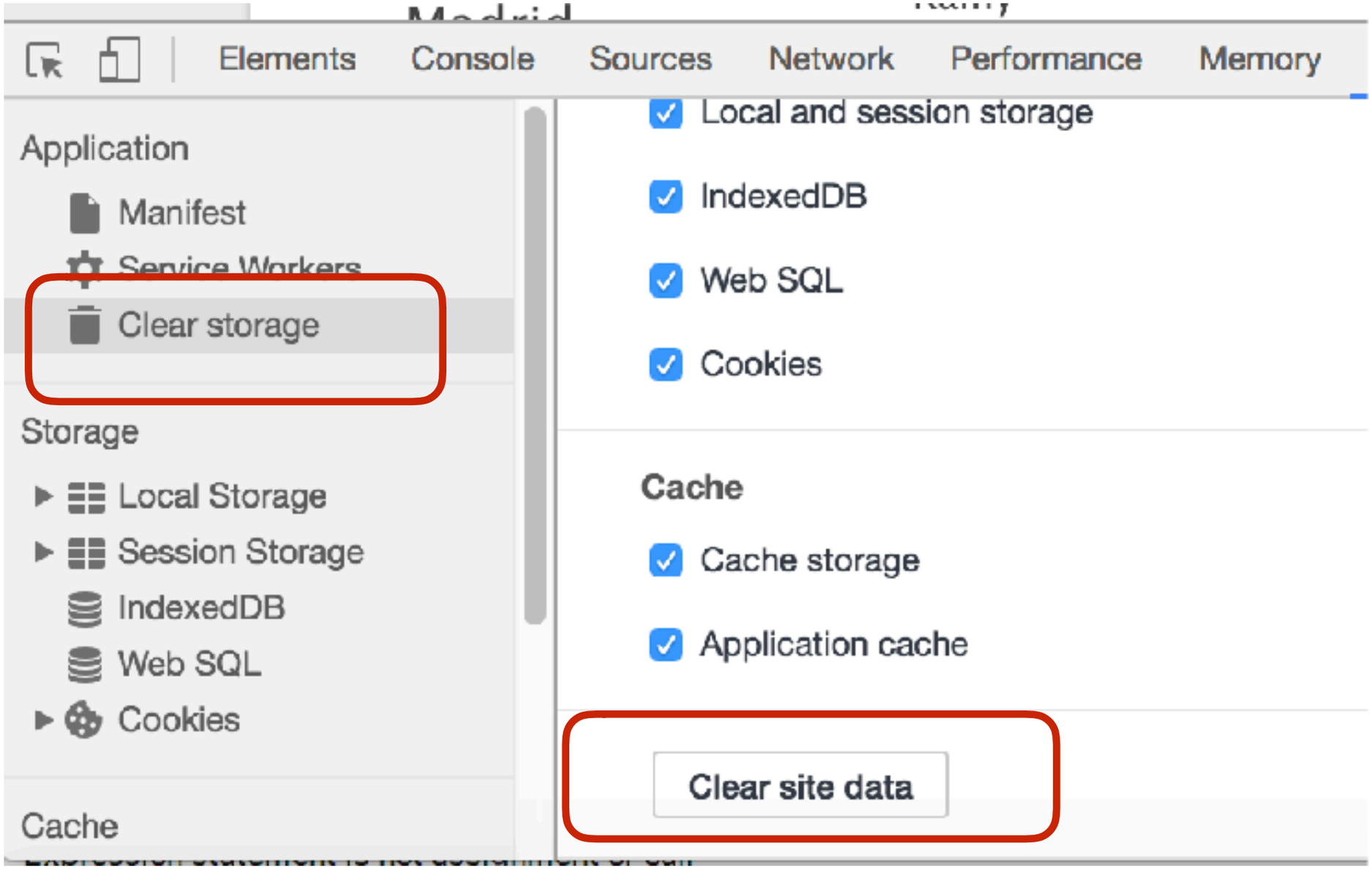


Modifying cached files

- To modify a cached file during development
  - modify the file
  - open chrome tools, choose the cache and right click on file to reload
  - then set the worker online again (application - Service worker - untick offline - otherwise you will get an error on loading the file you have just removed from the cache

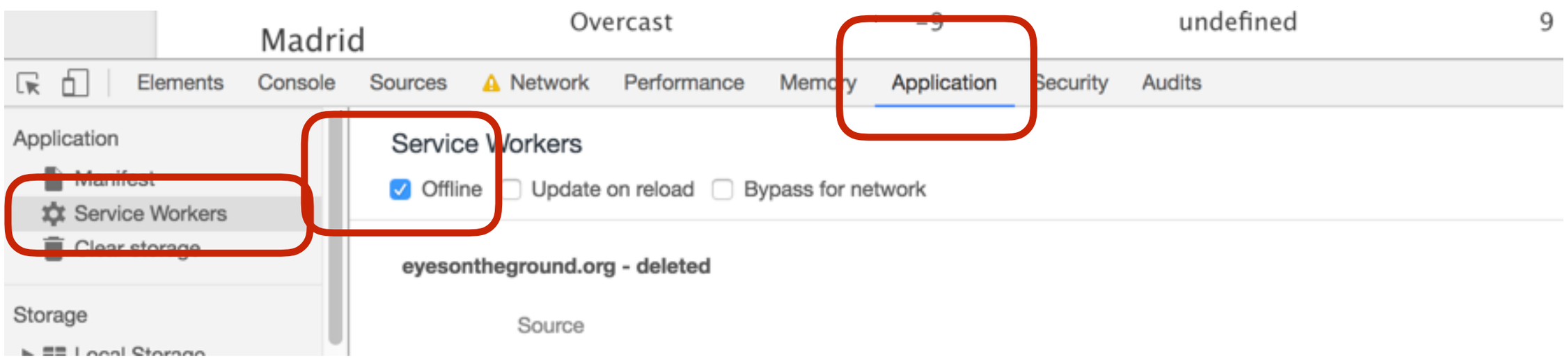


clear all your data



always check for errors

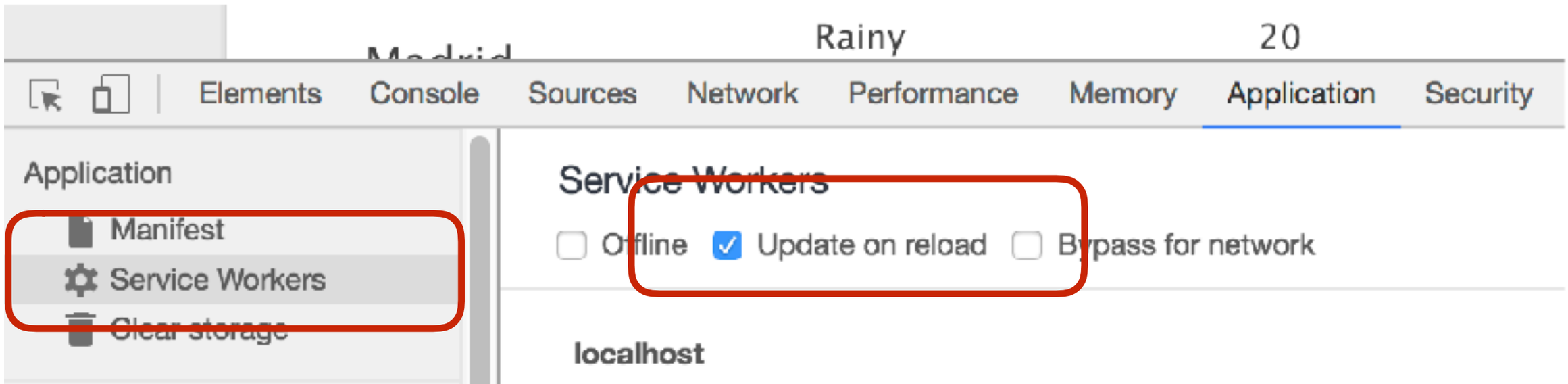
- if your service worker reports an error at installation time, it will not ACTIVATE
  - if it does not install **and** activate, it is not working, so you are effectively getting the data from the network
    - to test:
      - make sure to check the output of the console showing the activation
      - try putting some break points - it must stop when you reload the page
      - test the site office via the dev tools



Remember to reload worker

If you modify the code of the service worker, always remember to

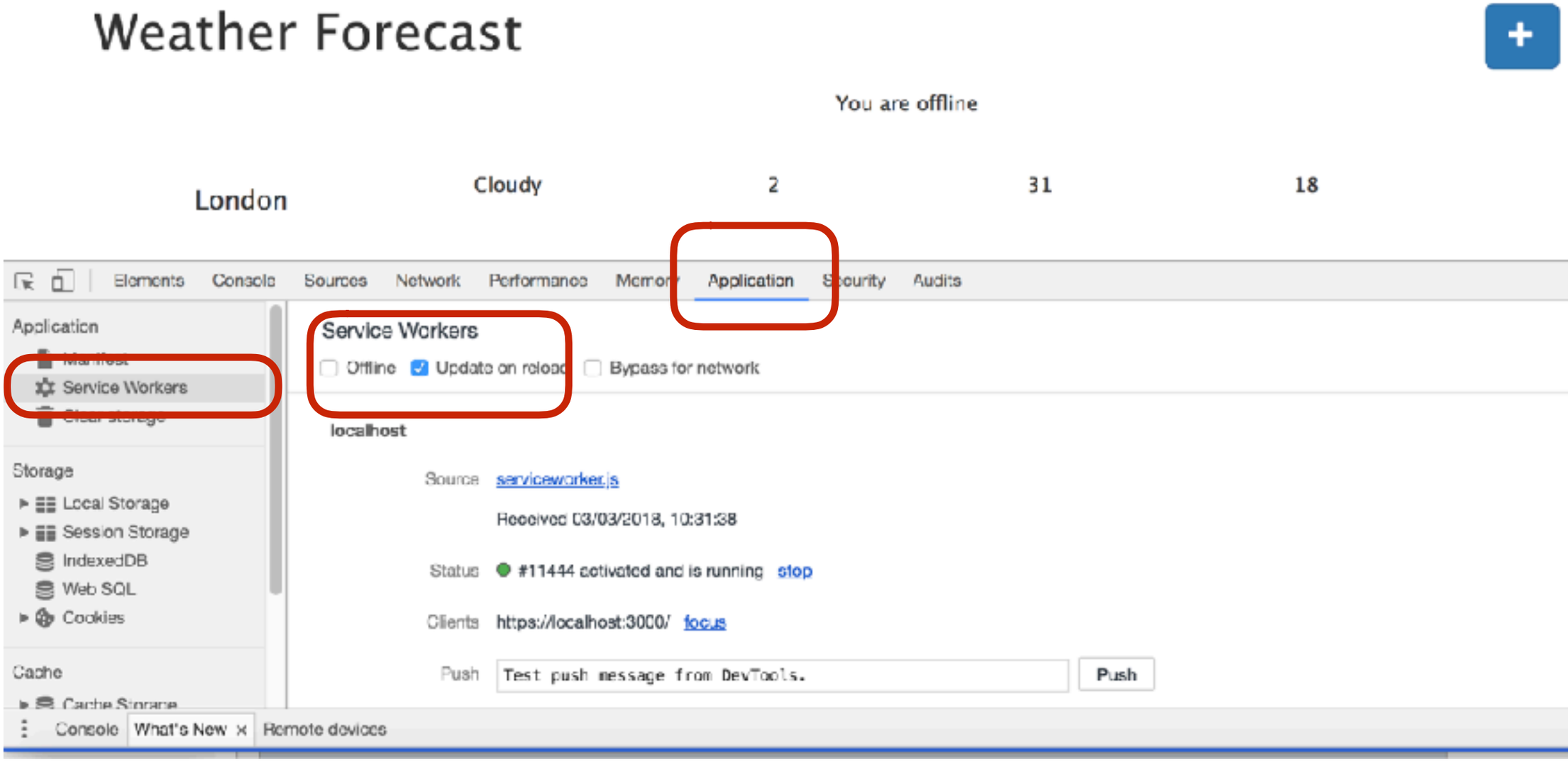
1. tick update on reload (after loading remember to untick! !important)
2. close all the tabs pointing to any of the pages controlled by the worker (!important) or the old web worker will be alive
3. reload the page



Updating service workers

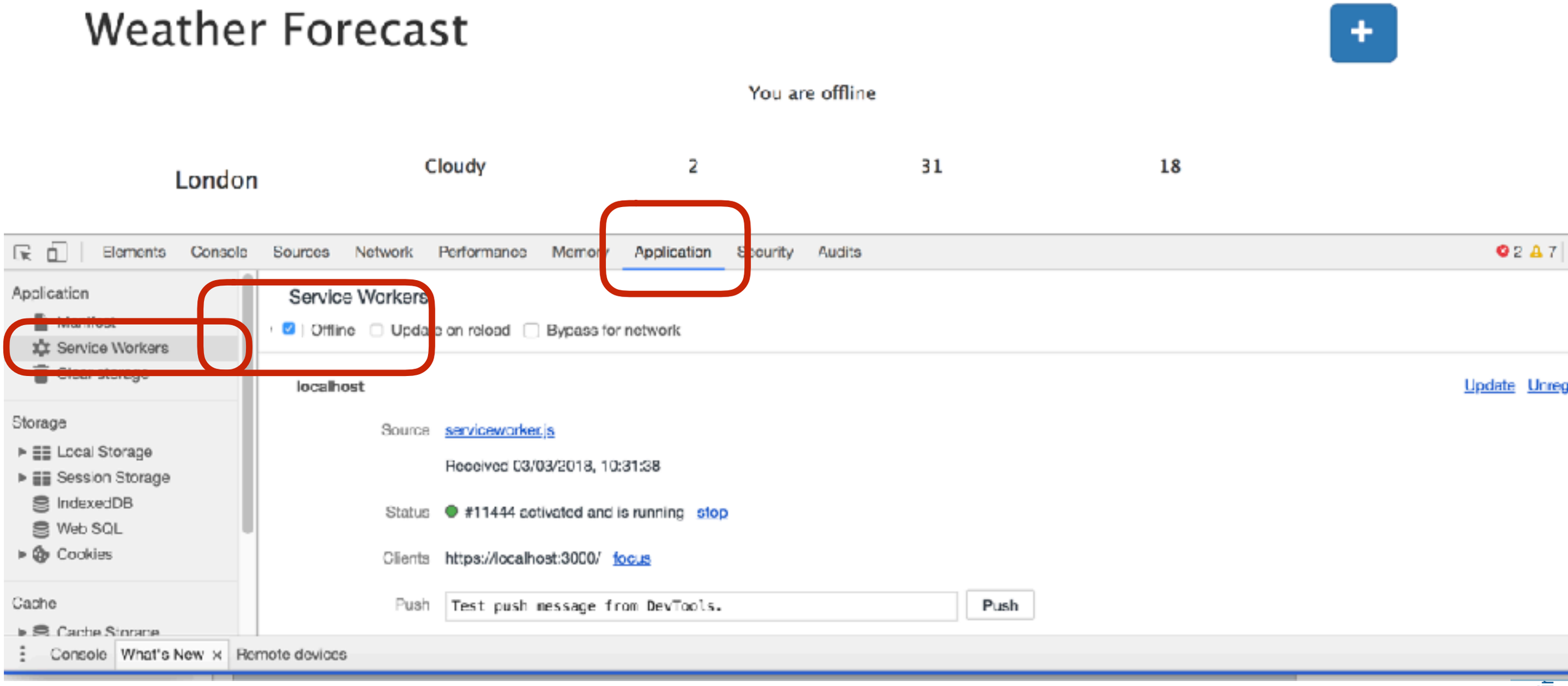
https://developers.google.com/web/ilt/pwa/tools-for-pwa-developers#update

---



The University Of Sheffield.

Testing offline



NOTE

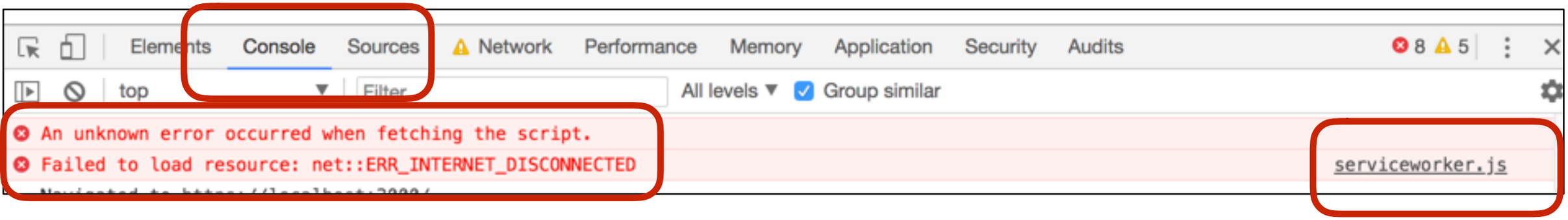
- When you re offline, Chrome will print out errors for your attempts to go to the network
  - it is one if you have a fallback position in AJAX

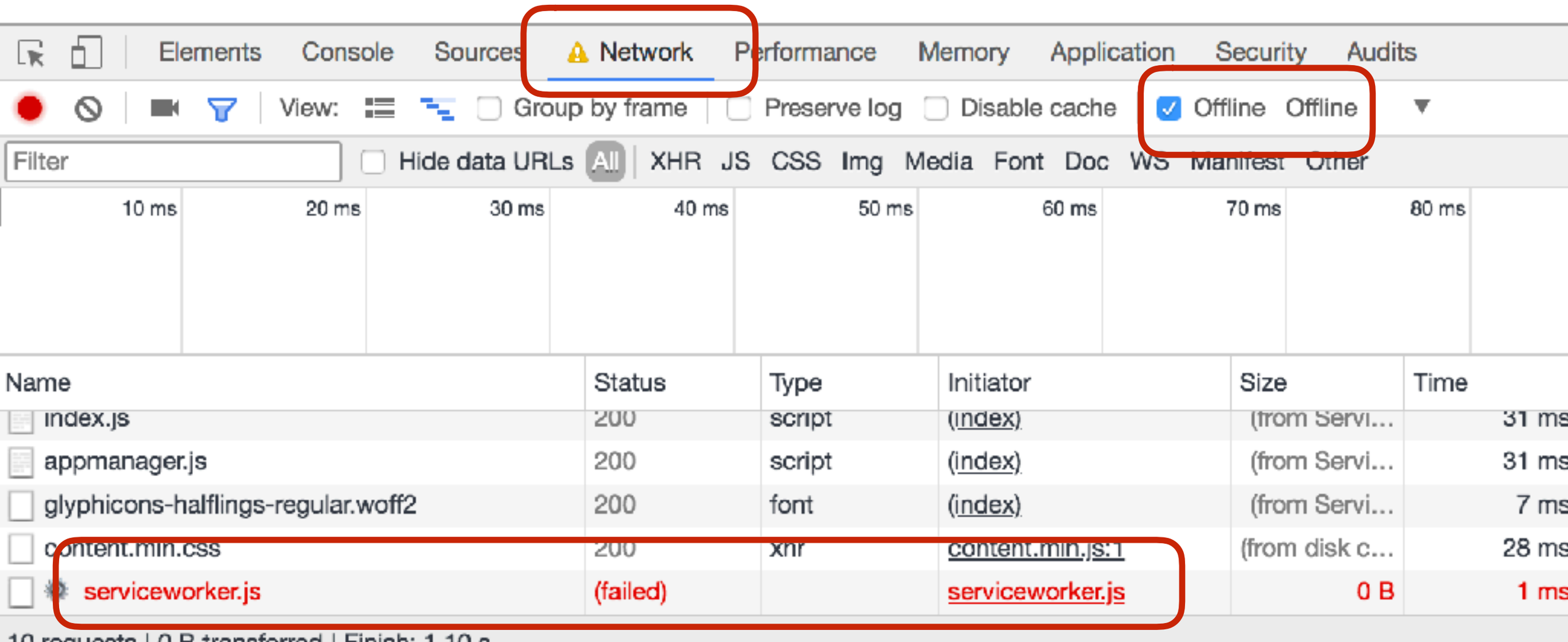
```
▶XHR finished loading: GET "chrome-extension://mgijmajocgfcbeboacabfgobmjgjcoja/conte
2 [Service Worker] Fetch https://eyesontheground.org:3000/weather_data
2 ▶Uncaught (in promise) TypeError: Failed to fetch
  ▶[Violation] Added synchronous DOM mutation listener to a 'DOMSubtreeModified' event.
  ▶POST https://eyesontheground.org:3000/weather_data net::ERR_INTERNET_DISCONNECTED
  ▶POST https://eyesontheground.org:3000/weather_data net::ERR_INTERNET_DISCONNECTED
  ▶Fetch failed loading: POST "https://eyesontheground.org:3000/weather_data".
```

```
const input = JSON.stringify({location: city, date: date});
$.ajax({
    url: '/weather_data',
    data: input,
    contentType: 'application/json',
    type: 'POST',
    success: function (dataR) {
        // no need to JSON parse the result, as we are using
        // dataType:json, so JQuery knows it and unpacks the
        // object for us before returning it
        addToResults(dataR);
        storeCachedData(dataR.location, dataR);
    },
    error: function (xhr, status, error) {
        showOfflineWarning();
        addToResults(getCachedData(city, date));
    }
});
```

Do not worry if…

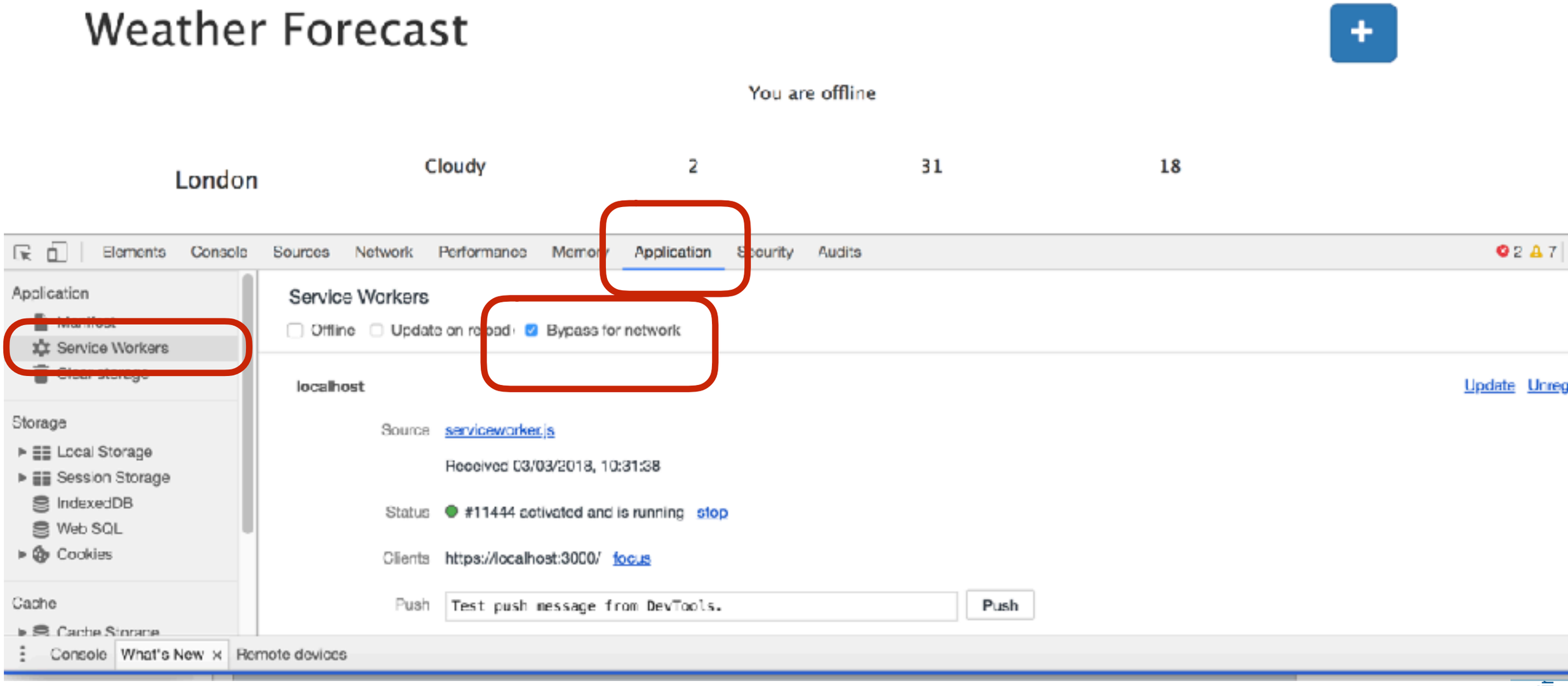
- When offline the page first tries to load a new version of the service worker from network and cannot fetch it. It is the correct behaviour





The University Of Sheffield.

Testing network only



The University Of Sheffield.

Storing Data with Mongo DB

MongoDB

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling

https://docs.mongodb.com/getting-started/shell/introduction/

Records

- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects.
- The values of fields may include other documents, arrays, and arrays of documents

Collections

- MongoDB stores documents in collections. Collections are analogous to tables in relational databases.
  - Unlike a table, however, a collection does not require its documents to have the same schema.
- In MongoDB, documents stored in a collection must have a unique _id field that acts as a primary key.


```
{
   "_id" : ObjectId("54c955492b7c8eb21818bd09"),
   "address" : {
      "street" : "2 Avenue",
      "zipcode" : "10075",
      "building" : "1480",
      "coord" : [ -73.9557413, 40.7720266 ]
   },
   "borough" : "Manhattan",
   "cuisine" : "Italian",
   "grades" : [
      {
         "date" : ISODate("2014-10-01T00:00:00Z"),
         "grade" : "A",
         "score" : 11
      },
      {
         "date" : ISODate("2014-01-16T00:00:00Z"),
         "grade" : "B",
         "score" : 17
      }
   ],
   "name" : "Vella",
   "restaurant_id" : "41704620"
}
```

BSON

- Mongo's documents are internally represented as binary JSON

https://www.slideshare.net/mdirolf/introduction-to-mongodb

Advantages

Faster process

Open Source

Sharding

mongoDB

Schemaless

Document based

No SQL Injection

https://www.studytonight.com/mongodb/advantages-of-mongodb

Clip slide

Best Use Cases

Scaling Out

Caching

The Web

High Volume

Clip slide

A Quick Aside

`_id`

special key

present in all documents

unique across a Collection

any type you want

https://www.slideshare.net/mdirolf/introduction-to-mongodb

Clip slide

Post

```
{author: “mike”,
 date: new Date(),
 text: “my blog post...”,
 tags: [“mongodb”, “intro”]}
```

Clip slide

Comment

```
{author: “eliot”,
 date: new Date(),
 text: “great post!”}
```

Clip slide

New Post

```
post = {author: “mike”,
   date: new Date(),
   text: “my blog post...”,
   tags: [“mongodb”, “intro”]}

db.posts.save(post)
```

posts is the name of the collection

Updating

```
c = {author: “eliot”,
   date: new Date(),
   text: “great post!”}

db.posts.update({_id: post._id},
                {$push: {comments: c}})
```

Clip slide

Retrieving or

```
db.posts.find({author: “mike”})
```

Clip slide

Last 10 Posts

```
db.posts.find()
        .sort({date: -1})
        .limit(10)
```

The University Of Sheffield.

Posts Since April 1

```
april_1 = new Date(2010, 3, 1)

db.posts.find({date: {$gt: april_1}})
```

Clip slide

Posts Ending With 'Tech'

```
db.posts.find({text: /Tech$/})
```

Clip slide

Posts With a Tag

```
db.posts.find({tags: "mongodb"})
```

...and Fast

(multi-key indexes)

```
db.posts.ensureIndex({tags: 1})
```

Clip slide

Indexing / Querying on Embedded Docs

(dot notation)

```
db.posts.ensureIndex({“comments.author”: 1})

db.posts.find({“comments.author”: “eliot”})
```

Clip slide

Counting Posts

```
db.posts.count()

db.posts.find({author: “mike”}).count()
```

- Paging is used to access several blocks of results
  - called pages

Clip slide

Basic Paging

```
page = 2
page_size = 15

db.posts.find().limit(page_size)
                .skip(page * page_size)
```

The University Of Sheffield.

mongoose public

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.

slack 10/386 | build passing | npm package 5.0.9

```
npm install mongoose
```

```
// Using Node.js `require()`
const mongoose = require('mongoose');
```

The University Of Sheffield.

Connecting to MongoDB

First, we need to define a connection. If your app uses only one database, you should use `mongoose.connect`. If you need to create additional connections, use `mongoose.createConnection`.

Both `connect` and `createConnection` take a `mongodb://` URI, or the parameters `host, database, port, options`.

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/my_database');
```

In the lab you will have mongo pre-installed
(just launch it)

Models (Schemas)

Defining a Model

Models are defined through the `Schema` interface.

```
const Schema = mongoose.Schema,
    ObjectId = Schema.ObjectId;

const BlogPost = new Schema({
 author: ObjectId,
 title: String,
 body: String,
 date: Date
});
```

Aside from defining the structure of your documents and the types of data you're storing, a Schema handles the definition of:

- **Validators** (async and sync)
- **Defaults**
- **Getters**
- **Setters**
- **Indexes**
- **Middleware**
- **Methods** definition
- **Statics** definition
- **Plugins**
- **pseudo-JOINs**

Example

```
const Comment = new Schema({
  name: { type: String, default: 'hahaha' },
  age: { type: Number, min: 18, index: true },
  bio: { type: String, match: /[a-z]/ },
  date: { type: Date, default: Date.now },
  buff: Buffer
});

// a setter
Comment.path('name').set(function (v) {
  return capitalize(v);
});

// middleware
Comment.pre('save', function (next) {
  notify(this.get('email'));
  next();
});
```

Used to perform operations before e.g. saving (for example to notify an administrator)

Accessing a Model

Once we define a model through `mongoose.model('ModelName', mySchema)`, we can access it through the same function

```
const myModel = mongoose.model('ModelName');
```

Or just do it all at once

```
const MyModel = mongoose.model('ModelName', mySchema);
```

The first argument is the *singular* name of the collection your model is for. **Mongoose automatically looks for the *plural* version of your model name.** For example, if you use

```
const MyModel = mongoose.model('Ticket', mySchema);
```

Then Mongoose will create the model for your **tickets** collection, not your **ticket** collection.

Once we have our model, we can then instantiate it, and save it:

```
const instance = new MyModel();
instance.my.key = 'hello';
instance.save(function (err) {
  //
});
```

Or we can find documents from the same collection

```
MyModel.find({}, function (err, docs) {
  // docs.forEach
});
```

You can also `findOne`, `findById`, `update`, etc. For more details check out **the docs**.

The University Of Sheffield.

Examples

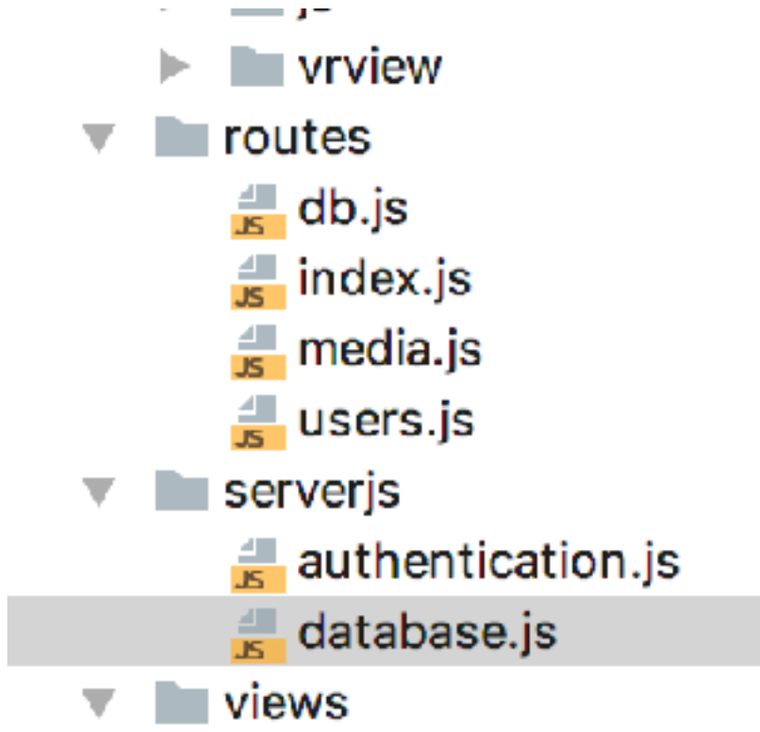
```
// retrieve my model
var BlogPost = mongoose.model('BlogPost');

// create a blog post
var post = new BlogPost();

// create a comment
post.comments.push({ title: 'My comment' });

post.save(function (err) {
  if (!err) console.log('Success!');
});
```

The University Of Sheffield.

Examples

```
BlogPost.findById(myId, function (err, post) {
  if (!err) {
    post.comments[0].remove();
    post.save(function (err) {
      // do something
    });
  }
});
```

In the lab



- create a js file called database.js

```
var mongoose = require('mongoose');
var assert = require('assert');
var ObjectId = require('mongodb').ObjectID;
var bcrypt = require('bcryptjs');

//The URL which will be queried. Run "mongod.exe" for this to connect
//var url = 'mongodb://eyesontheground.org:27017/test';
var url = 'mongodb://localhost:27019/test';
try {
    var connection = mongoose.createConnection(url);
    console.log("connection to mongodb worked!");
}catch (e) {console.log('error in db connection: ' +e.message)
}

//Create the Schema object
var Schema = mongoose.Schema
  , ObjectId = Schema.ObjectId;

 //Define a schema                 you can have as many as you want
var myClient = new Schema({
    clientId      : String
  , firstname     : String
  , surname       : String
  , address       : String
  , notes     : String
});
```

Create the connections

```
try {
    var Client = connection.model("Client", myClient);
    // add as many as necessary
  console.log("All connections to relations worked!");
}catch(e) {
    console.log('some connection to models did not work: ' +
e.message)
}
```

Create the access methods

```
//Public functions used to interact with the DB from the server
module.exports = {

   getClientWithoutIdFromDB: function (info, callback){
      console.log('client to find:' +info);
      var firstname = info.firstname;
      var surname = info.surname;
      var clientId = info.clientId;
      var options = {};
      if (clientId) options.clientId = clientId.toLowerCase();
      if (firstname) options.firstname = firstname.toLowerCase();
      if (surname) options.surname = surname.toLowerCase();
      console.log('client to find: '+ JSON.stringify(options));
      Client.find(options, callback);
   },
…
```

Note the callback to return the results (e.g. containing the response to contact the client)

Access the database

- create a route called db.js and include

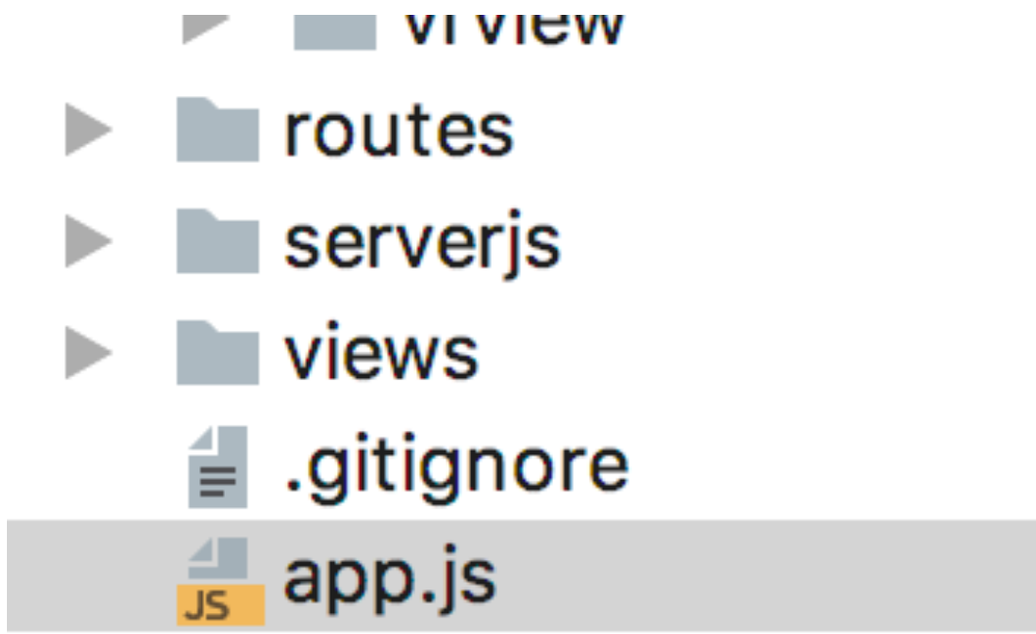
```
var express = require('express');
var router = express.Router();
var Database = require('../serverjs/database.js');
var bodyParser = require('body-parser');
```

```
router.use(bodyParser.json({limit: '50mb'}));


router.post('/find_clients_from_remote', function (req, res) {
    var body = req.body;
    Database.getClient(body, function (err, dataX) {
        if (err || dataX.length == 0) {
            res.writeHead(400);
            res.end(err || 'user not found');
        } else {
            dataX = dataX[0];
… do something
```

declare the route

- In your server's app.js

```
var db = require('./routes/db');
app.use('/db', db);
```



The University Of Sheffield.

Questions?