Building a Demo Frontend

Introduction

At our client's offices, as part of the agile development practise, the lifecycle of a story usually concludes with a demonstration to your service's Product Owner. Working with various stakeholders, the Product Owner is responsible for understanding the client's requirements, setting a roadmap towards meeting these, and working with developers to deliver features along the way. They assume accountability for the success of the service being designed and built, and so every story the team works on is only complete when the Product Owner, with their high level view of things, gives the final sign off.

The process of demonstrating a new feature to the product owner usually involves showing a lot of code, running it, and explaining as fluently as possible how this meets the needs of the client. It was suggested that this process could be improved for the Access Management team, if there was a front end of sorts through which new functionality could be demonstrated. Building such a front end is quite non-routine work for us, as front end work is normally handled by a separate team. I saw this then as a great opportunity for my personal development, as well as an excellent way to advance the quality of Software Engineer and Product Owner communication for our team.

The Development Process

First steps

I created a wireframe (figure 1) using a simple online tool, so as to create a more concrete idea in my mind of what the final product should look like, more or less, and to help me ensure I stayed on track while working towards this.

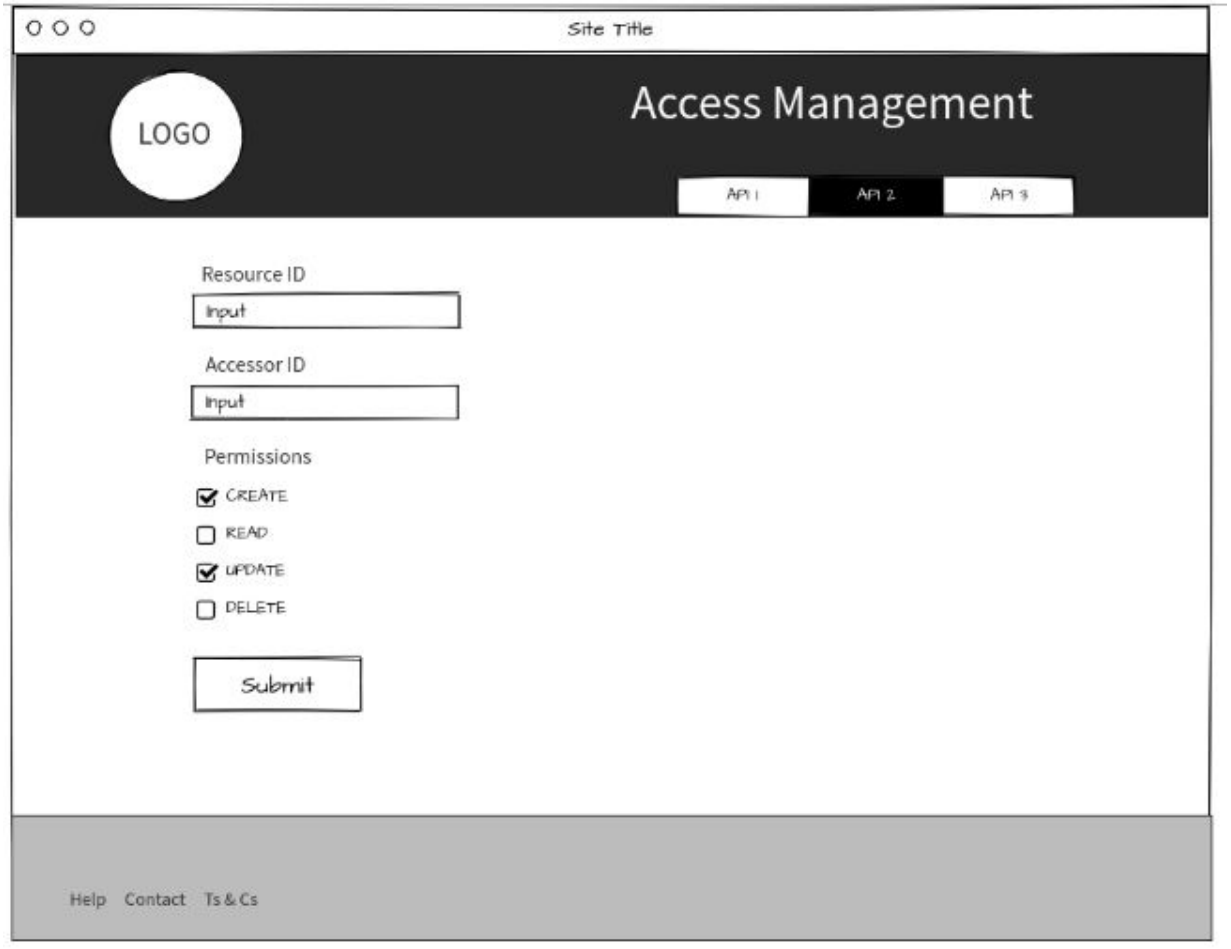


Figure 1 - Wireframe of the demo UI

I was aware that teams working on the frontend for client were using the Angular framework, so I decided to do the same. This is because though it was purely for internal purposes, I wanted the frontend to adhere to client

standards, and resemble the user interfaces created for the public as much as possible. Angular is a Typescript based client-side web framework that enjoys considerable popularity, so I was able to find plenty of resources with which I could start acquainting myself with it and with Typescript. I also found a tutorial for building a web application with Spring Boot (our backend framework) and Angular. I decided to follow the example to familiarise myself with the way the two would interact while simultaneously retooling it to suit the Access Management service. Starting with the backend, the only additional code required was a cross origin annotation at the top of the controller class (see figure 2).

```
@RestController
@CrossOrigin(origins = "http://localhost:4200")
@RequestMapping("api/${version:v1}")
/PMD.AvoidDuplicateLiterals, PMD.ConfusingTernary/
public class AccessManagementController {
```

Figure 2 – the cross origin annotation.

Because the Angular frontend would be deployed to http://localhost:4200 and the backend to http://localhost:3703, this annotation enabling Cross-Origin Resource Sharing (CORS), is required to prevent the browser denying requests from one to the other.

The client-side

Having done some research I understood that I'd need certain tools to make building the client-side easier, such as Node Package Manager (npm) which is distributed with Node.js. So I installed Node.js, opened the command line and then using npm installed Angular CLI (command line interface). Angular CLI makes project scaffolding quite easy. It takes a single command to create an entire new project, and similarly with single commands I would later generate a service and components (see figures 3, 4 and 5).

```
C:\Users\764108\Documents\demo_ui>ng new demoUI
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? SCSS   [ https://sass-lang.com/documentation/syntax#scss
CREATE demoUI/angular.json (3681 bytes)
CREATE demoUI/package.json (1293 bytes)
CREATE demoUI/README.md (1024 bytes)
CREATE demoUI/tsconfig.json (543 bytes)
CREATE demoUI/tslint.json (1953 bytes)
CREATE demoUI/.editorconfig (246 bytes)
CREATE demoUI/.gitignore (631 bytes)
CREATE demoUI/browserslist (429 bytes)
CREATE demoUI/karma.conf.js (1018 bytes)
CREATE demoUI/tsconfig.app.json (270 bytes)
CREATE demoUI/tsconfig.spec.json (270 bytes)
CREATE demoUI/src/favicon.ico (948 bytes)
CREATE demoUI/src/index.html (292 bytes)
CREATE demoUI/src/main.ts (372 bytes)
CREATE demoUI/src/polyfills.ts (2838 bytes)
CREATE demoUI/src/styles.scss (80 bytes)
CREATE demoUI/src/test.ts (642 bytes)
CREATE demoUI/src/assets/.gitkeep (0 bytes)
CREATE demoUI/src/environments/environment.prod.ts (51 bytes)
CREATE demoUI/src/environments/environment.ts (662 bytes)
CREATE demoUI/src/app/app-routing.module.ts (246 bytes)
CREATE demoUI/src/app/app.module.ts (393 bytes)
CREATE demoUI/src/app/app.component.html (25530 bytes)
CREATE demoUI/src/app/app.component.spec.ts (1098 bytes)
CREATE demoUI/src/app/app.component.ts (211 bytes)
CREATE demoUI/src/app/app.component.scss (0 bytes)
CREATE demoUI/e2e/protractor.conf.js (808 bytes)
CREATE demoUI/e2e/tsconfig.json (214 bytes)
CREATE demoUI/e2e/src/app.e2e-spec.ts (639 bytes)
CREATE demoUI/e2e/src/app.po.ts (262 bytes)
```

Figure 3 – Using the 'ng new <project name>' command to set up client-side

```
C:\Users\764108\Documents\demo_ui\demoUI\src\app\service>ng generate service token
CREATE src/app/service/token.service.spec.ts (328 bytes)
CREATE src/app/service/token.service.ts (134 bytes)
```

Figure 4 – Using 'ng generate service <service name> to create a service

```
C:\Users\764108\Documents\demo_ui\demoUI\src\app\test>ng generate component test
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
CREATE src/app/test/test.component.scss (0 bytes)
UPDATE src/app/app.module.ts (472 bytes)
```

Figure 5 - Using 'ng generate component <component name> to create a component

As you can see, these commands create a number of files and most of these files are not empty. The command to create the new project also sets up the entire folder structure for you, and the command to generate subsequent components also ensure the new files are housed in their own folders. Essentially it ensures that wherever possible all aspects of the project are set up according to Angular best practices, saving time and avoiding human error.

Having the basic architecture in place, one of the first things I wanted to do was ensure I could create a landing page with the layout and styling that was standard for all our client's UIs. Our public sector client has quite thorough guidelines and specifications for this. A design system website is available to developers detailing appropriate styles, and listing reusable components. Throughout, guidance is provided as to when it is appropriate to use a component and how they should be used to optimise user experience- with notes on user feedback for further clarity. It also provides page templates (see figure 6).

Page template

Use this template to keep your pages consistent with the rest of [redacted]

This page template combines the boilerplate markup and components needed for a basic [redacted] page. It includes:

- JavaScript that adds a `.js-enabled` class, which is required by components with JavaScript behaviour
- the skip link, header and footer components
- the favicon, and other related theme icons

Figure 6 – guidance on the basic page template provided from the design system site.

The boilerplate template was pasted into the app.component.html file that was created with the project. This root level file informs the appearance of the entire project, so the template markup would not need to be repeated elsewhere. For the styling, the below lines were similarly put into root level file styles.scss (figure 7).

```
1  /* You can add global styles to this file,
2  and also import other style files */
3  $[redacted]-global-styles: true;
4
5  @import "node_modules/[redacted]-frontend/[redacted]/all";
6  @import "node_modules/@[redacted]/frontend/all";
```

Figure 7 – Applying global styles.

The client's styles are provided in a package that can be installed using npm. When done this way, it is stored in the node_modules directory alongside other libraries. The last two lines of figure 7 essentially bring in all of the client's

styles from there. At this point, with all files saved, when the application is launched from the command line, I can now see an appropriately branded landing page (figure 8).

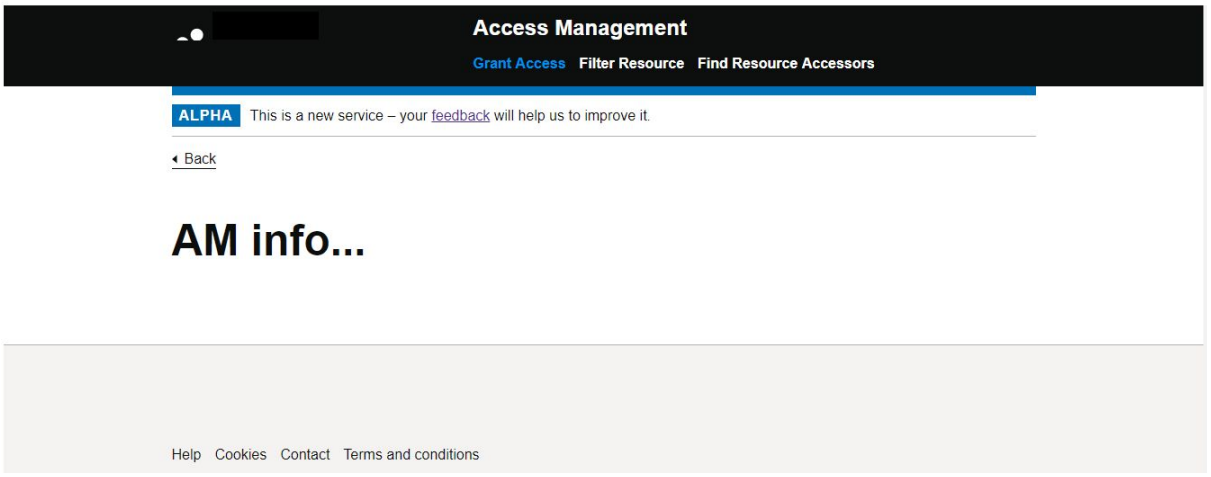


Figure 8 – The basic template.

Service Authorisation

My first attempt at writing a http request to the access management service returned a failed authentication message in the command line where access management was running. I realised then that the next thing I needed to establish was a way to acquire tokens to authenticate my requests. All http requests made to the access management service- in fact this applies to requests to all of the services developed for our client- require a service authorisation header with an accompanying token. Figure 9 shows a request for a token in Postman, an app for interacting with http APIs we frequently use for manual testing of endpoints. And figure 10 shows the response yielding a token.

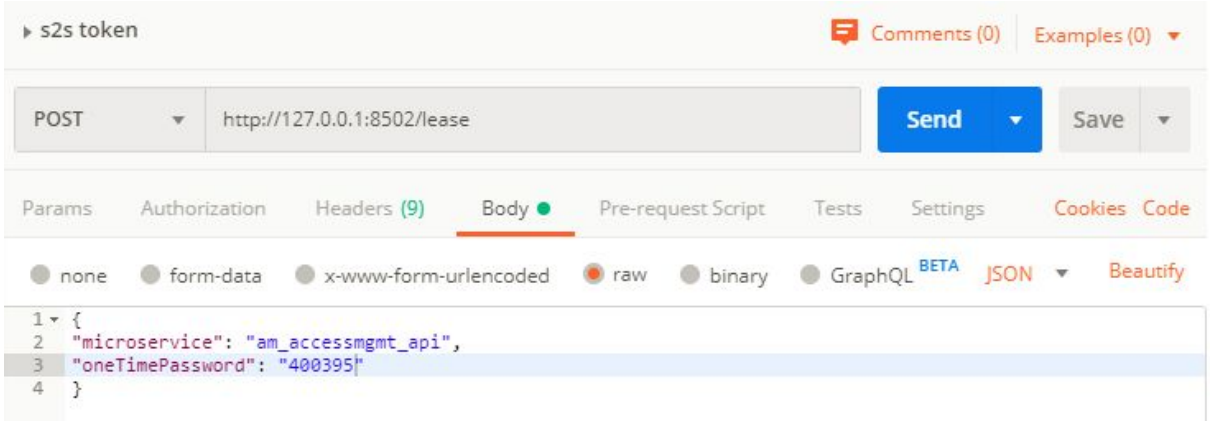


Figure 9 – A service to service token request in Postman.

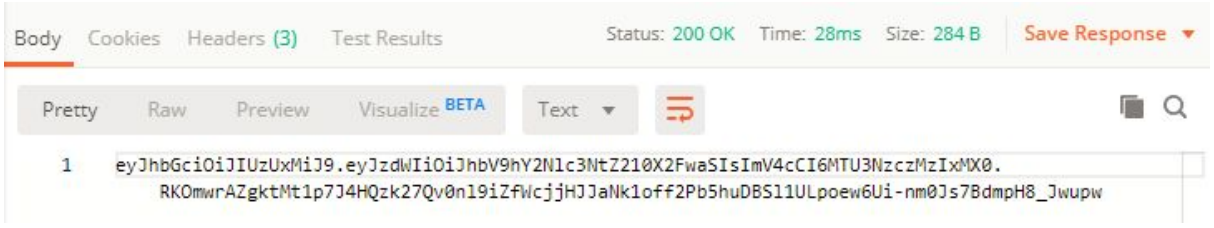


Figure 10 – The token in the response body.

This process is essentially what I needed to figure out how to do programmatically in my project, so that I could then take the token and attach it to a header in any given request subsequently made to an Access Management endpoint. This was further complicated by the one-time password in the request body being something that requires a secret to generate and expires every thirty seconds.

In the first instance I wrote a http request in the token service I'd created with a view to later inject this service into other components once my method worked correctly and yielded a token. This http request sent the request body with the microservice name and one time password to the url as shown in figure 9 above. For the one time password I installed a package called otp with npm and imported it into the service, so that I could use it's method for generating one-time passwords. For the sake of brevity I won't include a screenshot of this first attempt here, as it didn't work. Instead I found myself dealing with a series of errors in the console. The first was "Uncaught ReferenceError: global is not defined", then "Uncaught ReferenceError: Buffer is not defined", and then "Uncaught ReferenceError: process is not defined". For each of these I found solutions suggested online that I could implement in polyfills.ts (one of the auto generated files in Angular housing code that ensures the application is compatible for different browsers) one after the other (figure 11).

```
65  (window as any)['global'] = window;
66  (window as any).global.Buffer = (window as any).global.Buffer || require('buffer').Buffer;
67  (window as any).process = {
68    env: { DEBUG: undefined },
69    version: []
70  };
```

Figure 11 – Solutions to Uncaught Reference Errors in polyfills.ts file

I then arrived at a fourth error, and could no longer find any reasonable solutions. I researched this problem for a considerable amount of time. For a while I experimented with the possibility of including the generation of a time-based one-time password (TOTP) within the project's remit rather than delegating to a third party (see figures 12 and 13).

```
var TOTP = function() {

  var dec2hex = function(s) {
      return (s < 15.5 ? "0" : "") + Math.round(s).toString(16);
  };
  var hex2dec = function(s) {
      return parseInt(s, 16);
  };
  var leftpad = function(s, l, p) {
      if(l + 1 >= s.length) {
          s = Array(l + 1 - s.length).join(p) + s;
      }
      return s;
  };
  var base32tohex = function(base32) {
      var base32chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ234567";
      var bits = "";
      var hex = "";
      for(var i = 0; i < base32.length; i++) {
          var val = base32chars.indexOf(base32.charAt(i).toUpperCase());
          bits += leftpad(val.toString(2), 5, '0');
      }
      for(var i = 0; i + 4 <= bits.length; i+=4) {
          var chunk = bits.substr(i, 4);
          hex = hex + parseInt(chunk, 2).toString(16) ;
      }
      return hex;
  };
```

Figure 12 – Functions converting between decimal numbers, hexadecimal numbers and base 32 notation.

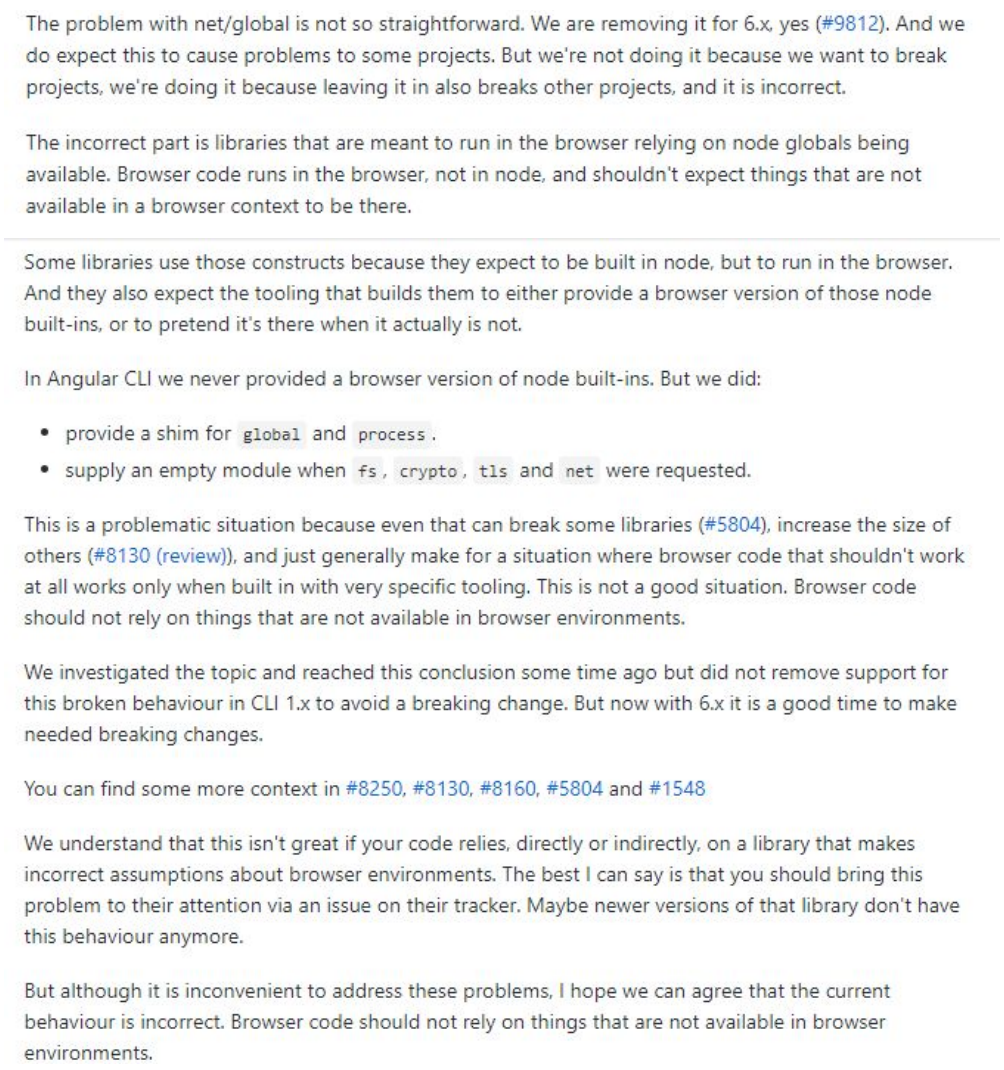
```
  this.getOTP = function(secret) {
      try {
          var epoch = Math.round(new Date().getTime() / 1000.0);
          var time = leftpad(dec2hex(Math.floor(epoch / 30)), 16, "0");
          var hmacObj = new jsSHA(time, "HEX");
          var hmac = hmacObj.getHMAC(base32tohex(secret), "HEX", "SHA-1", "HEX");
          var offset = hex2dec(hmac.substring(hmac.length - 1));
          var otp = (hex2dec(hmac.substr(offset * 2, 8)) & hex2dec("7fffffff")) + "";
          otp = (otp).substr(otp.length - 6, 6);
      } catch (error) {
          throw error;
      }
      return otp;
  };
```

Figure 13 – Function for generating a one-time password in javascript.

In brief, TOTP is a variant of the HMAC-Based One-Time Password (HOTP) algorithm. HMAC (hash-based message authentication code) being a message authentication code that uses a secret cryptographic key, as well as a hash function (an algorithm that maps data of an arbitrary size to a bit string of a fixed size), to verify the data integrity

and authenticity of a message. Both HOTP and TOTP generate one time passwords using a secret key and a moving part. In the case of the former, the moving part is an incrementing counter, while in the case of the latter, the moving part is 30 second increments of time. This is all new to me, and while it was fairly interesting to go down the tangential rabbit hole of learning cryptography- this felt like an inelegant solution. Adding so many lines of code just to achieve a six-digit password, where a third-party library should have worked more simply. And in any case this too relied on a third party library as there is no native Javascript HMAC function.

I kept researching then, and ultimately discovered the message in figure 14.



The problem with net/global is not so straightforward. We are removing it for 6.x, yes (#9812). And we do expect this to cause problems to some projects. But we're not doing it because we want to break projects, we're doing it because leaving it in also breaks other projects, and it is incorrect.

The incorrect part is libraries that are meant to run in the browser relying on node globals being available. Browser code runs in the browser, not in node, and shouldn't expect things that are not available in a browser context to be there.

Some libraries use those constructs because they expect to be built in node, but to run in the browser. And they also expect the tooling that builds them to either provide a browser version of those node built-ins, or to pretend it's there when it actually is not.

In Angular CLI we never provided a browser version of node built-ins. But we did:

- provide a shim for `global` and `process`.
- supply an empty module when `fs`, `crypto`, `tls` and `net` were requested.

This is a problematic situation because even that can break some libraries (#5804), increase the size of others (#8130 (review)), and just generally make for a situation where browser code that shouldn't work at all works only when built in with very specific tooling. This is not a good situation. Browser code should not rely on things that are not available in browser environments.

We investigated the topic and reached this conclusion some time ago but did not remove support for this broken behaviour in CLI 1.x to avoid a breaking change. But now with 6.x it is a good time to make needed breaking changes.

You can find some more context in #8250, #8130, #8160, #5804 and #1548

We understand that this isn't great if your code relies, directly or indirectly, on a library that makes incorrect assumptions about browser environments. The best I can say is that you should bring this problem to their attention via an issue on their tracker. Maybe newer versions of that library don't have this behaviour anymore.

But although it is inconvenient to address these problems, I hope we can agree that the current behaviour is incorrect. Browser code should not rely on things that are not available in browser environments.

Figure 14 – The root cause and Angular 6 breaking change.

Having understood why this was happening, I set about trying to find a solution. At first I hoped that I could find a different one-time password generating library- one that might not rely on node globals. I tried three others and got the same result. So I took my research to github, and found a frontend in current development for another service for our client. I saw that it used the ‘otp’ package- the first one I had tried using- and I realised that this frontend project was a full-stack service unto itself with it’s own node.js backend with express server. The otp package functioning correctly in this project made sense then, but the concept of a frontend, supported by it’s own backend, which is designed to be the frontend for another service that is entirely backend- was completely new to me, and at first glance seemed unnecessarily complicated. So I researched a little further and discovered that this architectural choice, decoupling frontend and backend then connecting them with APIs, conferred a number of benefits. Having separate teams work on the two in parallel optimises the speed of the work, it eliminates restraints on technology choices that each might have imposed on each other otherwise, and it is easier hire specialists in single domains.

While this discovery inspired a lot of curiosity and a lot questions, I was quite pressed for time and now cognizant of the need to learn node.js and express in order to finish this project.

Building a backend with Node.js and Express

First I created a server directory and in there created package.json and tsconfig.json files specifically for the backend (as opposed to the ones that were automatically created with the angular project) by running 'npm init' and 'tsc –init' in the command line respectively. I then added dependencies: ts-node, ts-node-dev, typescript, express and @types/express via the command line which could all then be seen in the package.json, and added a server.ts file in the server folder. To check I was on track at this point I wrote a line to log hello world to the console in server.ts, and in the command line executed the command 'ts-node server.ts'. My 'hello world' appeared in the command line as a result.

Next, following advice I'd found during the course of my research, I installed 'concurrently' with npm so I could run two scripts at the same time using node. Here I used it to launch the client and the server altogether. In the parent package.json (the original Angular one), under "scripts" I wrote the key value pair that would enable this (figure 15).

```
{
  "name": "demo-ui",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
    "serve": "concurrently \"ng serve\" \"src\\server\\node_modules\\.bin\\ts-node-dev --respawn src\\server\\server.ts\""
  },
```

Figure 15 – The concurrently script in the parent package.json

Each of the two commands must be enclosed in quotes, escaped with back slashes. The first command, ng serve, launches the Angular project and so far I'd been using it exclusively. The second command uses ts-node-dev to run the server.ts file. For development purposes, ts-node-dev creates a live watcher of your typescript files and automatically reloads as you make changes, which proved incredibly useful. I found I needed to include the file paths for both ts-node-dev and server.ts, and I also needed to replace the usual forward slashes of the file paths with double back slashes on account of being a windows user- but once I'd gotten the script right, I could run 'npm run serve' and launch the frontend and backend simultaneously.

My next concern was setting up the server (figure 16).

```
const routes = require('./routes');
const express = require('express');
const app = express();

app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
  res.header('Access-Control-Allow-Methods', 'OPTIONS, GET, POST, PUT, DELETE');
  if('OPTIONS' == req.method) {
    res.sendStatus(200);
  } else {
    console.log(`${req.ip} ${req.method} ${req.url}`); //For debugging
    next();
  }
})

app.use(express.json());

app.use('/', routes);

app.listen(4201, 'localhost', function() {
  console.log('Server now listening on 4201');
})
```

Figure 16 – The server.ts file.

Lines 2 and 3 import Express then create an Express application- the object is named ‘app’ as per convention. This object has methods for modifying application settings and routing http requests, among other things.

The code on lines 5 to 15 serves to address the issue of Cross Origin Resource Sharing (CORS). Once again this is because we’re requesting data from an API that is not on the same origin or host as the server. The first header, with the wildcard value, allows any origin. The second header allows a variety of headers. Finally the third header, and the if-block immediately below it, allows for various RESTful methods.

Line 17 is a built in express method that is a type of middleware- meaning it’s called between processing a request and sending a response- which recognises an incoming request object as a JSON object.

Lines 21 to 23 sets the host and port, and logs to the console.

Lines 1 and 19 import and make use of a separate routes file I created, in which a call to Express’ Router() method creates a sub router that I can delegate to. To begin with I populated this file with just the basic ‘hello world’ route, as well as basic pair of get and post routes to an experimental ‘/thing’ path (figure 17) that I used to test that it works as expected (it did), and later use as a syntax model.

```
router.get('/', (req, res) => res.send('Hello World'));
router.get('/thing', (req, res) => res.send([]));
router.post('/thing', (req, res) => res.send({body: req.body}));

module.exports = router;
```

Figure 17 – basic sub router set up.

Returning to the token issue

I made sure to add otp as a dependency on this level in my newly established backend’s package.json. I then created a token.ts file in the server folder and moved much of the code I had been experimenting with there (figure 18).

```
// Attempt 1 - kind of works
  function retrieveServiceToken () {
    let oneTimePassword = otp({ secret: totpSecret }).totp()
    console.log(oneTimePassword)

    let options = {
      method: 'POST',
      uri: S2sUrl,
      body: {"microservice": microserviceName, "oneTimePassword": oneTimePassword},
      //body: new ServiceAuthRequest(microserviceName, oneTimePassword),
      json: true
    }
    console.log(options)
    return rp(options)
    .then(function (parsedBody) {
      console.log("TOKEN::::::::::" + parsedBody);
      return parsedBody;
    })
    .catch(err => {
      console.log(err);
      return null;
    })
  }
```

Figure 18 – First attempt at retrieving token from backend.

Then before tinkering with it too much I imported the token.ts file in the server and added my first attempt at custom middleware (figure 19).

```
app.use(function (req, res, next) {
  const auth = token.retrieveServiceToken()
  console.log(auth)
  res.header('Authorization', auth);
  next()
})
```

Figure 19 – my first attempt at custom middleware.

Now when I refreshed http://localhost:4200/, in the command line I was finally seeing the one time password and the token being generated (figure 20).

```
ng serve
[1] 127.0.0.1 GET /
[1] TOKEN::::::::::eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhbV9hY2Nlc3NtZ210X2FwaSIsImV4cCI6MTU3NzYzMzQ0Mn0.C0-rVDcfwIDTFQRQeyEL
182Hf29HCu_VkJwD9SknwxxZnv2NrFLHUxnP8O9QxFPdwKIAfjbDQFv-j6ACGMeh5g
[1] 598323
[1] { method: 'POST',
[1]   uri: 'http://localhost:8502/lease',
[1]   body:
[1]    { microservice: 'am_accessmgmt_api', oneTimePassword: '598323' },
[1]   json: true }
```

Figure 20 – One-time password and token now generating.

However this was followed by an error and accompanying stacktrace- and not every refresh yielded a new token. Progress for sure, but something was still not right. After some research I realised that I needed to take a proverbial step back, and with consideration of everything I'd now learned, make a new plan for completing this project. The process of going from attempting to build a simple views layer with angular to building an entirely separate API had

led me to create a fledgling Frankenstein's monster- it's parts harvested from various tutorials, documentations, and other repositories. However now that I broadly understood what I was building and the tech stack I was using, it seemed that now would be a good time to try and move forward with a greater sense of deliberation and forethought. In the first instance, I needed to understand asynchronous operations and Promise chaining.

An asynchronous function operates in a separate order to the rest of the code, and a Promise is an object representing the eventual completion or failure of an asynchronous operation. I realised that understanding and applying these two concepts would get past this hurdle, because I didn't just want to get a token- I wanted to trigger a request for a token every time a call to an access management endpoint was made, and I wanted to ensure the token was retrieved, and attached to the http request header, before the call continued to execute. I needed these events to happen in a particular order, one after the other each time.

Through trial and error, and copious use of the console.log method to identify problems to solve and progress made, I wrote a series of async functions. These functions essentially return a promise, fulfil that promise and attach the returned token, as part of a custom header, to my post request to the 'grant access' endpoint (see figures 21 and 22).

```
47  function createAuthRequest() {
48    let oneTimePassword = otp({ secret: totpSecret }).totp()
49    console.log(oneTimePassword)
50    return new ServiceAuthRequest(microserviceName, oneTimePassword)
51  }
52
53  function getToken() {
54    let data = createAuthRequest();
55    console.log(data)
56    let response = axios.post(S2sUrl, data)
57    console.log(response)
58    return response
59  }
60
61  async function currentToken() {
62    if (authToken === undefined) {
63      authToken = await getToken()
64      console.log("In if block - authToken generated::::" + authToken);
65      return authToken;
66    } else {
67      console.log("In else block - authToken::::" + authToken);
68      return authToken;
69    }
70  }
```

Figure 21 – retrieving a token with async functions part 1

The function 'createAuthRequest()' generates a one-time password and prepares the request body for retrieving a token. The function 'getToken()' calls 'createAuthRequest()', stores the return value in a variable and then passes that variable as the second argument in an axios post request, with the first argument being the destination url. The outcome of this post request is returned, which I had thought would be the response, and named it accordingly, but is in fact at this point a pending Promise. Apropos, axios is a Promise based HTTP client that I found recommended on stack overflow. I installed it using npm and imported it into the file, and it proved particularly useful for appending an authorisation http header later. The last function in figure 21, 'currentToken()' uses a conditional statement to check whether the authToken variable, which I had declared at the top of the file, is currently undefined. Undefined is a primitive type in javascript and is the value returned if a variable has not been assigned a value. Here, if authToken is undefined, then 'getToken()' is called. The keyword 'await', which is only valid in async

functions, pauses the execution of the function to wait for the passed Promise, then resumes the function's execution. At this point the variable authToken holds a response object.

```
async function getLatestToken() {
  let latestToken = await currentToken()
  console.log(JSON.stringify(latestToken.data))
  return latestToken.data;
}

async function grantAccess(requestBody) {
  let serviceAuth = await getLatestToken()
  const options = {
    headers: {'ServiceAuthorization': `${serviceAuth}`}
  }
  axios.post('http://localhost:3704/api/v1/access-resource', requestBody, options)
       .then((response) => {
        console.log(response);
      })
      .catch((error) => {
        console.log(error)
      });
}
```

Figure 22 – retrieving a token with async functions part 2

I realised that the object being returned contained headers, http status code, http status text, etc. So the 'getLatestToken()' function waits for the object to be retrieved then returns just the response body. I could see from the console log that I finally had a token being returned, so I called 'getLatestToken()' to trigger the series of events that I knew would retrieve it and stored the returned value in the variable serviceAuth. I then created another variable, options, which held the ServiceAuthorization header and interpolated the serviceAuth variable into the value. Following that an axios post request is made to the access management endpoint, with the options variable passed as the last argument. Finally I wrapped these last few steps in the process in the async function 'grantAccess()', which I then called a couple of lines later and ran 'npm run serve' to see the output.

At this point the 403 forbidden error I'd been receiving was replaced with 400 bad request, which let me know I'd finally gotten past the authentication issue. The 400 status code (figure 23) made sense as I was passing an empty object as the request body. My next step then was to build the form and accompanying logic that would capture the data comprising the request body, in the client side.

```
[1]       { errorCode: 400,
[1]         status: 'BAD_REQUEST',
[1]         errorMessage: 'Malformed Input Request',
[1]         errorDescription:
[1]          'Cannot deserialize instance of `java.util.HashSet<java.lang.Object>` out of VALUE_STRING token\n at [Sourc
e: (PushbackInputStream); line: 1, column: 36] (through reference chain: uk.gov.hmcts.reform.amlib.models.ExplicitAccess
Grant$ExplicitAccessGrantBuilder["accessorIds"])',
```

Figure 23 – Error code 400 due to an empty request body.

Returning to the client-side

A note on UI design

I put together the HTML that would capture the sum of the request body's parts. As I mentioned previously, our client provides very thorough guidelines and specifications for service UIs. It does not just concern itself with the colours and styles that comprise it's brand identity, but also accessibility. I found guidance on easing the user journey for those using screen magnifiers or screen readers, and published blogs on detailed field research that had been conducted such the passage below on updates made to checkboxes and radios:

The problem
Early in the development of […] we observed in research how a majority of users would click on radio button or checkbox controls rather than on their labels, despite the fact that the labels are much bigger and therefore easier to click (see Fitt's Law).
We reasoned that this was because users didn't know whether or not they could click on the labels. Many websites don't let you click on labels, so choosing to always click on the control is perfectly rational user behaviour.

The solution
Iteration 1 - grey boxes
First we thought we'd try to make it really obvious that you could click our labels, so we coloured them grey and made them respond to the mouse hovering over them.
We thought this would work, so we rolled out the design across services on […]. We saw again and again though in lab research that most users still clicked on the controls.

Iteration 2 - bigger controls
In our next iteration we decided to support the existing user behaviour, so we increased the size of the controls to make them easier to click.
Unfortunately it turns out that native browser support for bigger radios and checkboxes is very patchy. Nearly half the browsers we tested didn't support these bigger controls.
Still, we rolled out the change for those that did and raised bug reports for those that didn't.

Iteration 3 - custom controls
In our latest iteration we've replaced the native browser controls with custom ones, which all our supported browsers will get. We've also removed the grey background, as it did not have the effect on user behaviour that it was intended to.
The new controls are more in keeping with the rest of […] and are the same height as our text fields, which improves the vertical rhythm on form pages and allows for better horizontal alignment of form fields.

The results
The new styles have been piloted in a number of services, including high-volume ones like […]. So far they've tested really well. In research, people of all confidence levels are clicking these controls quickly and easily.
The code has been tested extensively and works across the full range of supported browsers, even if the user has customised their colour scheme.

This was another tangential research rabbit hole for me. That there was *so* much information, complete with eponymous laws, about every element of a user interface was interesting, and slightly overwhelming, but really helpful in informing the design of the form.

Client-side logic

Once the form had been put together, I created a grant service (figure 24) which housed a method containing a post request. The first parameter is the path, which here is the backend URL followed by a '/grant-access' route I added to the backend routes file (figure 25). The second parameter is the data to be sent, which will be passed as an argument whenever the method is called.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class GrantService {

  postUrl = 'http://localhost:4201/grant-access';

  constructor(private _http: HttpClient) { }

  grantAccess(grantAccessData) {
    return this._http.post<any>(this.postUrl, grantAccessData);
  }
}
```

Figure 24 – The grant service.

```
router.post('/grant-access', function(req, res){
  grantAccess(req.body);
  res.json();
});
```

Figure 25 – The corresponding express route containing the grantAccess async function.

I then generated a grant-access component, and in it's typescript file injected the service using the constructor (figure 26) as well as an Angular class useful for building complex forms.

```
export class GrantAccessComponent implements OnInit {

  grantAccessForm: FormGroup;

  constructor(private fb: FormBuilder, private _grantService: GrantService) { }

  ngOnInit() {
    this.grantAccessForm = this.fb.group({
        resourceId: [''],
        accessorIds: [["Reem"]],
        accessorType: ['USER'],
        resourceDefinition: this.fb.group({
          serviceName: [''],
          resourceType: [''],
          resourceName: ['']
        }),
        attributePermissions: this.fb.group({
          "/": [["READ"]]
        })
      })
  }
```

Figure 26 – The grant access component.

On line 13 I declare the variable grantAccessForm of type FormGroup- this will store the form data. Lines 18 to 30 show the form model. As you can see, I've created a couple of form group instances within the larger instance- this is so that the form data will be nested correctly when converted to JSON. Also it is possible to provide default values or leave fields empty, and an extra set of square brackets indicate an array. All of the keys declared within the form groups, such as resourceId and serviceName are called form controls which are then referenced in the html (figure 27).

```
3  <div>
4    <form [formGroup]="grantAccessForm" (ngSubmit)="onSubmit()">
5      <div class="[redacted]-form-group">
6        <label class="[redacted]-label" for="resource-id">
7          Resource ID
8        </label>
9        <span id="resource-id-hint" class="[redacted]-hint">
10         e.g. [redacted]
11       </span>
12       <input formControlName="resourceId" class="[redacted]-input [redacted]-
13     </div>
```

Figure 27 – html snippet.

The value given to the formControlName attribute on line 12 matches the form control declared in the typescript file. On line 4 I added the formGroup directive in the square brackets to the form html tag and bound the grantAccessForm to it. These create an association between the form groups and form controls in the typescript file and their corresponding html elements, enabling communication between them.

```
132      <button data-prevent-double-click="true" class=
133        Submit
134      </button>
135    </form>
136    {{ grantAccessForm.value | json }}
137  </div>
```

Figure 28 – Interpolating the form values as JSON.

To visualise the communication between the html and the model, I interpolated the form values on line 136 with the JSON pipe (figure 28). This would allow me to see the structure of the Json at the bottom of the form in the browser, enabling me to tinker with the model to get the values nested correctly, and also to see the values populate as I filled in the form (see figure 29).

Submit

{ "resourceId": "1234", "accessorIds": ["Reem"], "accessorType": "USER", "resourceDefinition": { "serviceName": "test service name", "resourceType": "test resource type", "resourceName": "test resource name" }, "attributePermissions": { "/": ["READ"] } }

Figure 29 – The interpolated form data with JSON pipe output.

In figure 27 above, just after the formGroup directive on line 4, I bound the form to the ngSubmit event which is emitted when the submit button is clicked, and assigned the event handler 'onSubmit()'. This method was then written in the grant access component (figure 30).

```
33   onSubmit() {
34     this._grantService.grantAccess(this.grantAccessForm.value)
35     .subscribe(
36       response => console.log('Success!', response),
37       error => console.log('Error', error)
38     );
```

Figure 30 – The method triggered on submitting the form.

On line 34 it calls on the instance of the service that was injected earlier, then on the 'grantAccess()' method that was defined in the service. The form values are passed as an argument, then '.subscribe', which is a method on the observable type, looks out for the returned data in order to do something with it. Here, it logs to the console either the success response or error.

I received the 400 bad request error a few times while I tinkered with the form logic, until I finally managed to get the input data correctly transformed into the JSON structure required to make a successful request to the grant access endpoint. At which point I received a new error (figure 31).

```
[1]      data:
[1]       { errorCode: 500,
[1]         status: 'INTERNAL_SERVER_ERROR',
[1]         errorMessage:
[1]          'There is a problem with your request. Please check and try again',
[1]         errorDescription: 'The execution of the service failed',
[1]         timeStamp: '01-01-2020 19:54:09.136' } },
```

Figure 31 – 500 internal server error.

This didn't really give much information, and didn't make immediate sense like the 400 bad request error did, so I checked the command line where access management was running and found the issue (figure 32).

```
am-accessmgmt-api-db        | 2020-01-01 19:54:09.113 UTC [815] ERROR:  insert or update on table "access_management" violates foreign key constraint "access_management_resources_fkey"
am-accessmgmt-api-db        | 2020-01-01 19:54:09.113 UTC [815] DETAIL:  Key (service_name, resource_type, resource_name)=(cmc, case, case name) is not present in table "resources".
am-accessmgmt-api-db        | 2020-01-01 19:54:09.113 UTC [815] STATEMENT:  insert into access_management (resource_id, accessor_id, permissions, accessor_type, service_name, resource_type, r
esource_name, attribute, relationship, last_update, calling_service_name) values ($1, $2, $3, cast($4 as accessor_type), $5, $6, $7, $8, $9, now() at time zone 'utc', $10) on conflict (resourc
e_id, accessor_id, accessor_type, attribute, resource_type, service_name, resource_name) where relationship is null do update set permissions = $11,  last_update = now() at time zone 'utc', ca
lling_service_name = $12
am-accessmgmt-api-db        |  RETURNING "access_management_id"
```

Figure 32 – Foreign key constraint violated.

So the error here, '*insert or update on table "access_management" violates foreign key constraint "access_management_resources_fkey"*', is caused by the three fields that comprise the resource definition not already being present in the resources table. The final hurdle then, before using my UI to demonstrate a successful post request to the grant access endpoint, I needed to ensure the test data was loaded in the database ahead of any demonstration. So through the command line I got the database up and running with the command, 'winpty docker exec -it am-accessmgmt-api-db psql am -U amuser', then ran insert statements I'd prepared (see figure 33). Executing 'select * from resources;' afterwards showed my new resource definition row had been successfully inserted (figure 34).

```
1  insert into services (service_name, service_description) values ('test service name', null)
2   on conflict on constraint services_pkey do nothing;
3
4  insert into resources (service_name, resource_type, resource_name) values ('test service name', 'test resource type', 'test resource name')
5   on conflict on constraint resources_pkey do nothing;
```

Figure 33 – Insert statements to prevent foreign key constraint violation in preparation for demo.

```
am=# select * from resources;
   service_name    |   resource_type    |   resource_name    |        last_update
-------------------+--------------------+--------------------+----------------------------
                   |                    |                    | 2020-01-06 16:54:40.485488
                   |                    |                    | 2020-01-06 16:54:40.485488
 test service name | test resource type | test resource name | 2020-01-06 17:26:12.819309
(3 rows)
```

Figure 34 – Resource definition in database ahead of post request demo.

Now when I launched my application, navigated to the grant access form, filled it in and clicked submit- I could see in the node command line a 201 success code for the first time (figure 35) and I could see the request body looked exactly as it should (figure 36).

```
[1] 127.0.0.1 POST /grant-access
[1] In else block - authToken::::[object Object]
[1] "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhbV9hY2Nlc3NtZ210X2
JIDXuzeSvkSTbyhu0-8kQT6GtoN7FiobXpAXtu6Br9wKSpQImiUA"
[1] { status: 201,
```

Figure 35 – Status 201 created.

```
[1]   data:
[1]    { resourceId: '1234',
[1]      resourceDefinition:
[1]       { serviceName: 'test service name',
[1]         resourceType: 'test resource type',
[1]         resourceName: 'test resource name' },
[1]      accessorIds: [ 'Reem' ],
[1]      accessorType: 'USER',
[1]      attributePermissions: { '/': [Array] },
[1]      relationship: null } }
```

Figure 36 – Request body.

At this stage, having at last worked out the data's entire journey from input field to persistence in the access management service, I asked my line manager to test it and provide me with feedback. He was very happy with it overall, suggesting that given time it would be worth expanding the UI to include the other access management APIs, particularly 'filter resource' as it would help demonstrate the fine-grained access control capabilities of the service.

Reflective Statement

This was the most challenging project to date. Given that the project's entire tech stack was completely new to me, I found the learning curve pretty steep. Throughout the course of building this frontend, progress was often frustrated by the need for me to stop and learn the basics of the tools and frameworks I was using. That said, this is possibly the most rewarding project to have committed myself to and I regret I didn't have more time. If possible I may tinker with it further in my own time. Owing to this project I now have a foundational knowledge of Typescript, the Angular framework, Node.js, Express, npm, and a handful of useful libraries like axios and otp that I utilised here. I'm keen to build on this, but also to circle back to some of the questions that arose along the way that I didn't have the time to satisfy. For example, in researching the choice to decouple frontend and backend I found discussions around micro-frontends which I'm curious to know more about. Also the language around cryptography and security was fairly opaque and difficult to understand, but I'm curious to learn more about it nonetheless, as I recognise how important it is. I appreciate now, perhaps more than ever, what a broad church software engineering is.

The other very salient value of doing this project was of course the benefits it yields for our team's demonstrations to the Product owner. These demonstrations are often an exercise in communication skills for both parties. I had observed in these meetings a tendency for software engineers to occasionally relapse into overly technical language, and for protracted discussions in which the Product Owner would have to ask several questions in the vein of, 'what if the end user tries/wants/does this?'. While this is a fairly normal part of the demonstration process as far as I'm

aware, I can also see how such questions could be satisfied with greater ease via a user interface- rather than blocks of code that could seem pretty opaque to a non-technical colleague.