Building a sample app that uses the Orion APIs

**Post 1: the application**

In this post we’ll talk about how we developed a small application that uses the Orion health FHIR API’s both as the source data, and also as a repository to store data that it creates. The application is an implementation of the Framingham Cardiovascular risk calculator ([reference](https://en.wikipedia.org/wiki/Framingham_Risk_Score)) which calculates the likelihood (as a percentage) of having some sort of cardiovascular event (like a Heart Attack) in the subsequent 10 years. (Please don’t rely on these figures from this app – this is a demonstration only!)

It works by taking a number of measures (blood lipid levels, blood pressure, smoking status and whether a diabetic) and assigning a number of ‘points’ to each one (there can be negative points for good results as well). Then the points are totaled and the overall risk calculated. We can save the risk assessment back to the platform, which means that it can be viewed directly from the Portal. (It would be quite feasible to completely automate this process – having a ‘background’ process that calculates the risk automatically, but that is outside the scope of this project).

The application is a web application, which means that it will run in a browser on any device – including smart phones, though the UI is not optimized for this. It runs fine on a tablet however.

We’ll discuss this project in a number of posts.

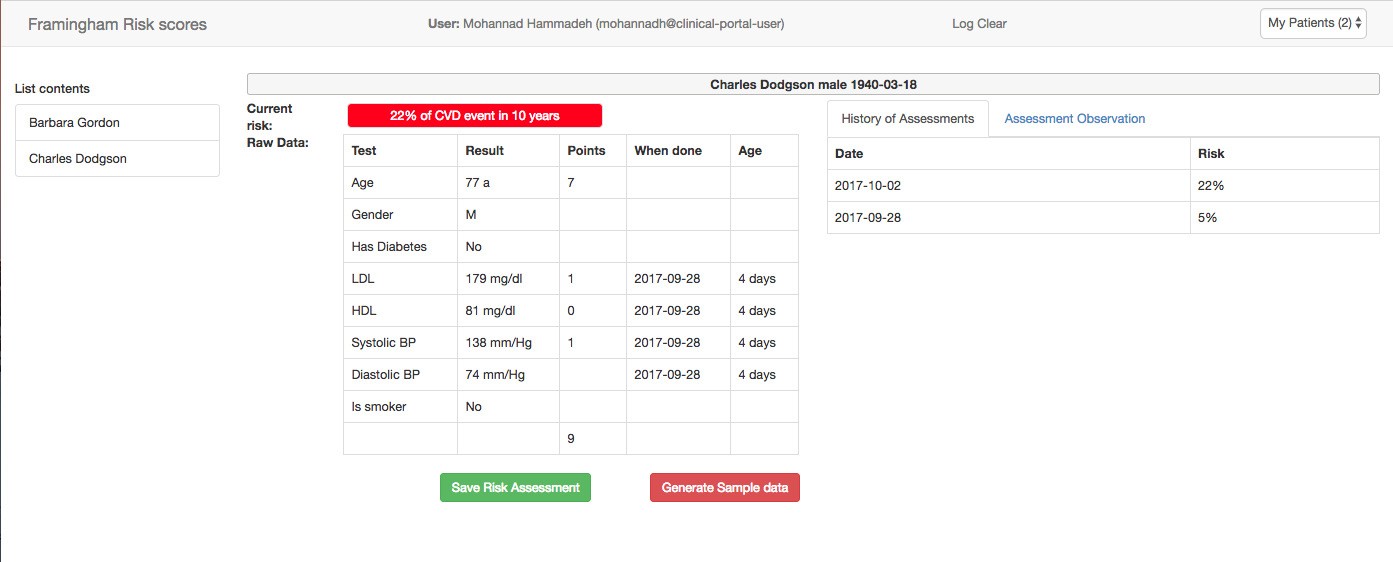
* First we’ll talk about the way the app works – how we meet the requirements of calculating the risk for an individual patient
* Next, we’ll consider what API’s we need from the platform, and how to call them
* Finally, we’ll dig into the security details – how we use OAuth to secure the application

The overall application flow is as follows.

First we log into the application. Actually, behind the scenes we are really logging into the Orion Health platform, so we need to have a portal login. Our user account also needs to have permissions to access the clinical data we need (we’ll talk about this is the last post in this series).

Once we’ve logged in, the app will retrieve the Worklists that have been defined for the logged in user. These worklists are lists of patients that can be maintained by the user. The lists are displayed in the app.

We select a worklist, and the app then displays all the patients in that worklist. We can select any patient, and the app will retrieve the data from the platform (via the API) and calculate the risk. Both risk and the data used to calculate the risk are displayed on the screen. Here’s a screen shot:



We can then save the assessment back to the platform using the green ‘Save Risk Assessment’ button, after which it can be viewed in Portal – or retrieved from the API again. In fact, we display the previous assessments on the screen over on the right hand side of the app.

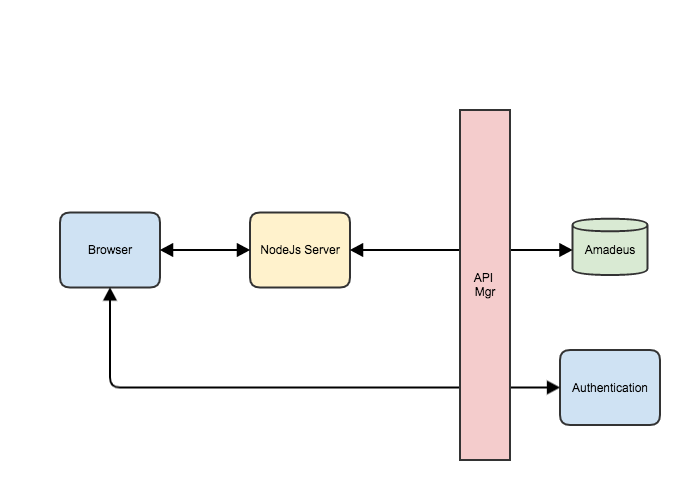
Here’s a screen shot of the assessment being viewed from portal:

>>> picture to come.

And because this is a demo app, we’ve also provided the ability to generate random test data (using the red button).

And that’s really all the app is! There’s a lot more we could do – for example we could allow the user to modify the data on screen and re-generate the risk to show someone how they could reduce their risk, or display a chart showing the risk changing over time, but this is only a demo after all!

Here is the architecture of the app.



It’s a single page application (SPA) using Angular on the client, communication with a nodejs application on the server. The node app makes the actual API calls (via the API Manager) and returns the results to the client. It also has the assessment algorithm.

This approach was taken to avoid any CORS (Cross Origin) issues that can occur when the call is made directly from the browser to the server.

In the next post, we’ll dig into the API in more detail.

========== end of post 1 ===========

**Post 2: API requirements**

In the previous post we described the overall operation of our sample application. Now, let’s dig into the actual API’s that we’re going to need. Here’s a list:

|  |  |  |
| --- | --- | --- |
| **Call** | **Description** | **Type** |
| Get User details | Get the details of the currently logged in user. | Orion |
| Get worklists for user | Retrieve the worklists that for the current user | Orion |
| Get details of worklist | Get the worklist details – the patient who are on the list. This is used to display the list of patients | Orion |
| Get patient details | Needed when we select patient from the list to display details like name, Date of Birth, gender (We also need this as part of the risk calculation). | FHIR |
| Get Observations for a patient | Actually, we only need some specific observations – we’ll discuss this more in a moment. | FHIR |
| Get Conditions for a patient. | Like Observations, we only need specific conditions – in this case diabetes | FHIR |
| Write Observation | We use this to write back the Risk Assessment Observation to the platform. | FHIR |

We’ll consider each of these in turn. Note that these calls are made from the Nodejs server to the platform – not the Browser app. The server itself exposes API endpoints that the client app calls from the browser. For this reason you won’t see the calls to the platform in the browser logs.

Note also that any calls to the platform must include the access token (as an http header) that is issued by the platform after logging in. So you must log in first to be able to access any data (this process is described in the next post)

Here are the platform calls.

## Get User Details

Currently this is not a FHIR API (And there are no plans to create one as user management is considered outside the scope of FHIR). Orion supplies the following query to retrieve the details of the currently logged in user

GET [API endpoint]/Actor/current

The actual response is specific to Orion – but straightforward to understand.

## Get worklists for User

Again, this is specific to Orion:

GET [API endpoint]/Actor/{user identifier}/patientlist/watchlist

The user identifier comes from the previous API call – get user details.

If this were to be ‘FHIRified’ then it would most likely be a ‘List of Lists’

## Get worklist details

GET [API endpoint]/Actor/{user identifier}/patientlist/watchlist/{listIdentifier]

The list identifier comes from the ‘get worklist’ call. In this app, the server transforms this into a FHIR bundle of Patient resources, so that the client does not need to understand the Orion-specific nature of the contents.

From a FHIR perspective this is a simple List (of Patient). And the List resource has an identifier – so implementing it should be straightforward…

## Get Patient details

This is a true FHIR endpoint exposed by the platform. Because the actual patient information that we have is their identifier, it is actually a search against Patient with the identifier as a parameter, and returning a Bundle with (hopefully) a single Patient. Here’s the call:

GET [API endpoint]/Patient?identifier={}

And note that the identifier is in the FHIR format – {system}|{value}. Note that like all platform calls involving PHI (Personal Health Information), the data is only returned if the user has the right to view that data. For this demo we aren’t checking (and, presumably the patient wouldn’t be on the list in the first place if the user didn’t have access to their records), but in real-world app it would be a good idea to check (there are various bundle.link elements that carry this information).

The server extracts the Patient from the bundle and returns to the client.

## Get Observations for a client

This is another FHIR endpoint exposed by the platform (again, subject to privacy). We have a couple of ways of using it in this scenario. The endpoint does support query for specific values, so we could just query for the values that we want – and that would be the best solution. However, there are 5 different codes that we are after, so to be performant we’d need to ‘parallelize’ the calls to the platform.

Given that this is just a demo, we went for the easier approach of retrieving all the Observations for the patient from the platform, and filtering them in the nodejs server. If this were a ‘real’ server, we’d make the separate parallel calls.

Here’s the call:

GET [API endpoint]/Observation?subject.identifier = {}

Note that this is what is technically called a ‘chained’ request – we want Observations where the subject of the Observation has specific characteristics (a given identifier). See <http://hl7.org/fhir/search.html#chaining> for more details about chaining. As a side note, the spec also defines a ‘patient’ search parameter which queries on subject where the subject is a Patient – so an equivalent query would be Observation?patient.identifier={}. However this is not currently implemented in the platform.

## Get Conditions for Patient

Very similar to getting observations – and the same comments about searching for a specific code or filtering on the server apply.

GET [API endpoint]/Condition?subject.identifier = {}

An interesting question here is determining which condition.code represents diabetes, as there are a multitude of possible codes depending on the terminology being used and the level of detail of the Condition. This is a separate discussion – for the purposes of this app we just chose an example one.

## Write Observation

This is the only write API that we’re using in this example (and, at the item of writing, the only write operation that the platform supports). It’s a straightforward POST operation, the only tricky bit being that we need to get the reference to the patient using the Patient.id element of the GET Patient operation described above.

POST [API endpoint]/Observation

We actually use it in a couple of places:

1. When writing back the Assessment Observation.
2. When creating the demo data

So that all the API’s we need!

--------- end of post 2 ---------

**Post 3: Security using OAuth**

So far we haven’t talked about the security aspects of this interaction, so it’s time to take a look at OAuth2. A couple of comments before we get started:

* The calls must be using TLS – Transport Level Security – ie HTTPS and not HTTP.
* This is not quite the same as SMART. SMART is a ‘profile’ on OAuth2 which we are in the process of implementing. This is ‘vanilla’ OAuth2

OAuth2 is a commonly used framework that manages user access to data. It can be used for a number of different scenarios – one of which is to control access by an application (and a user of that application) to data provided by an EHR over an API (Application Program Interface) – exactly what we want to do.

The overall flow of OAUth2 is described in some detail [here](https://fhirblog.com/2016/03/13/implementing-smart-on-fhir-in-an-ehr/) (with links to the formal specification), but the following is a high level description from the perspective of the user of an external application wishing to access data in an EHR. Feel free to skip to the actual steps in the process described below.

There are a number of key actors in the process:

* An application that can access data from the EHR (read or write)
* A user of that application who has an account in the EHR (i.e. is allowed to access data to some level)
* The server that has the data to be accessed (the Resource Server, in this case the EHR)
* The server that can identify that the person is who they say they are (the Authorization (Auth) server) – e.g. by supplying a login & password. This may be the same as the EHR, but doesn’t have to be.

(These aren’t quite the same as in the OAuth2 spec – but hopefully a bit easier to understand in the context of this discussion)

The overall idea is that the Resource server doesn’t have to actually authorize the user directly (which is really hard to do properly) – it can delegate that task to someone that it trusts, rather than having to do that itself (that’s how you can ‘login with google’ to a completely separate web application).

The OAuth2 spec itself doesn’t dictate how this trust occurs, though does describe a number of patterns. One of these is the use of a ‘token’ – issued by the Auth Server, and used by the Resource Server to decide what services to offer.

The overall flow for a web based application using OAuth2 to access data via an API is as follows.

Pre-requisites

* The application is pre-registered with the Auth server, and given a code that identifies it.
* The location of the required end points on the Auth and Resource server are known (more on this later) to the application
* The User of the application is known to the Auth server, and has some way of identifying themselves (in this case a user name & password – the same that you’d use to log into Portal)
* The Auth server can issue a token that the Resource server can use to identify the user and application – ie there is a trust mechanism between the two

Usual Flow

1. The user starts the app, and initiates the ‘log in’ process.
2. The app contacts the Auth server, passing across its app code (so the Auth server knows who is contacting it). The Auth server then displays a login page. (Assuming that username/password are being used to identity the user)
3. The user enters their username and password which is validated by the Auth Server. The Auth server then ‘redirects’ back to the app, giving it an ‘authorization code’.
4. The app then calls the Auth server again, passing across the authorization code, and receiving in return an Access Token. Whenever it subsequently makes a call to the resource server, it will include the Access Token in the call, so that the Resource server can decide whether to honour the request (and know who is making it for audit purposes).

You may wonder why there are 2 calls required to the Auth server – one to get the Authorization code, and a second one to get the Access Token. The reason is that the Access Token usually only lasts for a short time (commonly 1 hour) before it expires. However in step 4 above the Auth server also returns another token – the Refresh Token - that the app can use to get another Access token when it has expired.

So, in effect, what is happening is that the user can continue to access the EHR API for as long as the Refresh token is valid (which is controlled by the Auth server), but every hour the app has to check that the user is still allowed to access the EHR API. This allows the Auth server to revoke access if needed, and the maximum time that the user can invoke the EHR post invalidation is one hour.

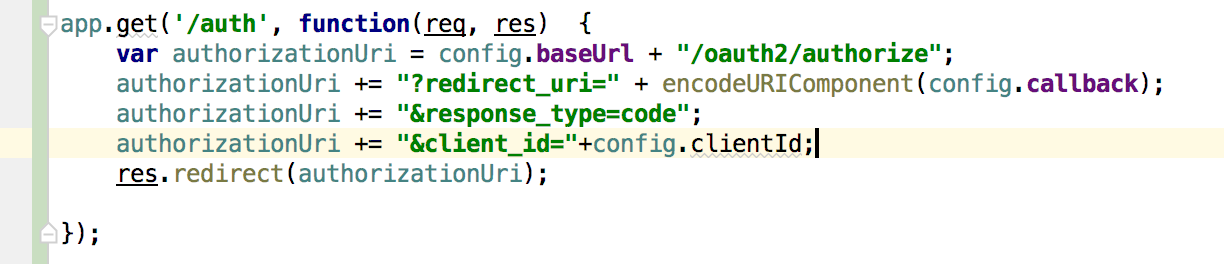
So with the theory behind us, let’s describe how we’ve used OAuth to login to the platform, and also to identify the user.

## Step 1

The server exposes a start page (actually named login.html in this app) which has some descriptive text, and a ‘login’ button that calls an ‘/auth’ endpoint on the server. This end-point constructs a url that will reference the authorization endpoint in the platform and issues a ‘redirect’ to the browser, indicating that it should load an html page at that location. The Url contains some specific attributes needed for the authorization – namely:

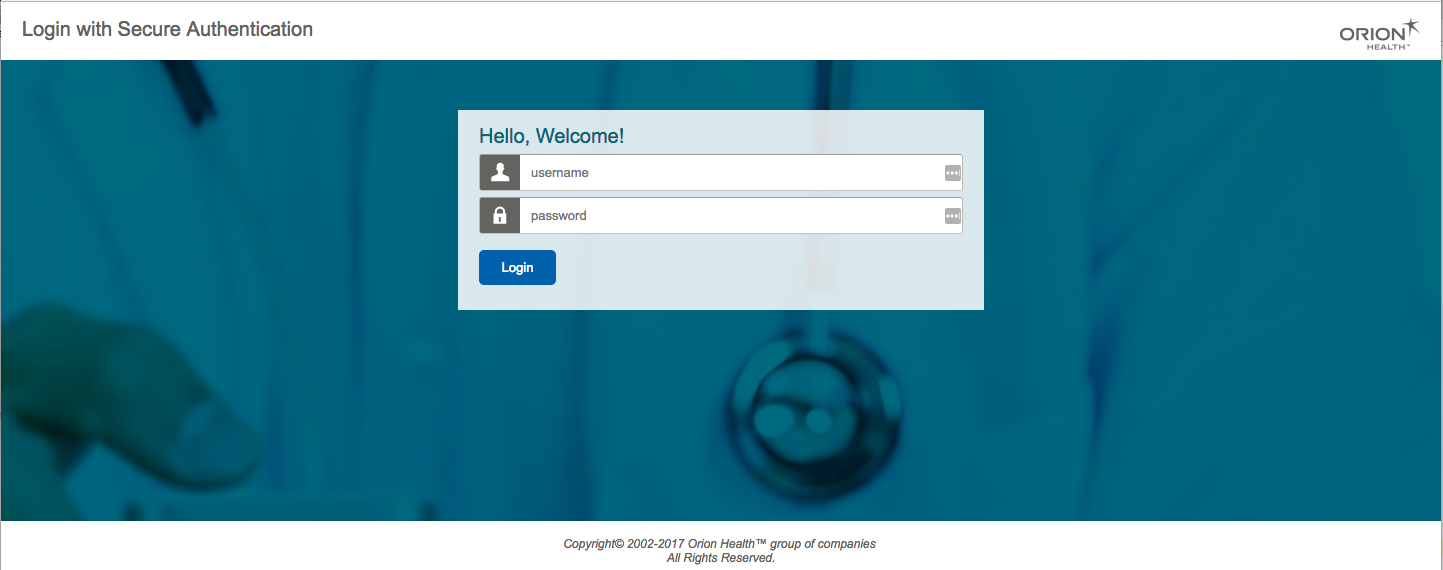
* The application id (client id)
* The callback url (where the browser should be re-directed after successful authentication
* The response type (fixed to ‘code’)

Here a screenshot of the server code:



Note that the actual values for the parameters are stored in a config file – for obvious reasons they aren’t shown here!

The redirect causes the Portal login page to be loaded.



## Step 2

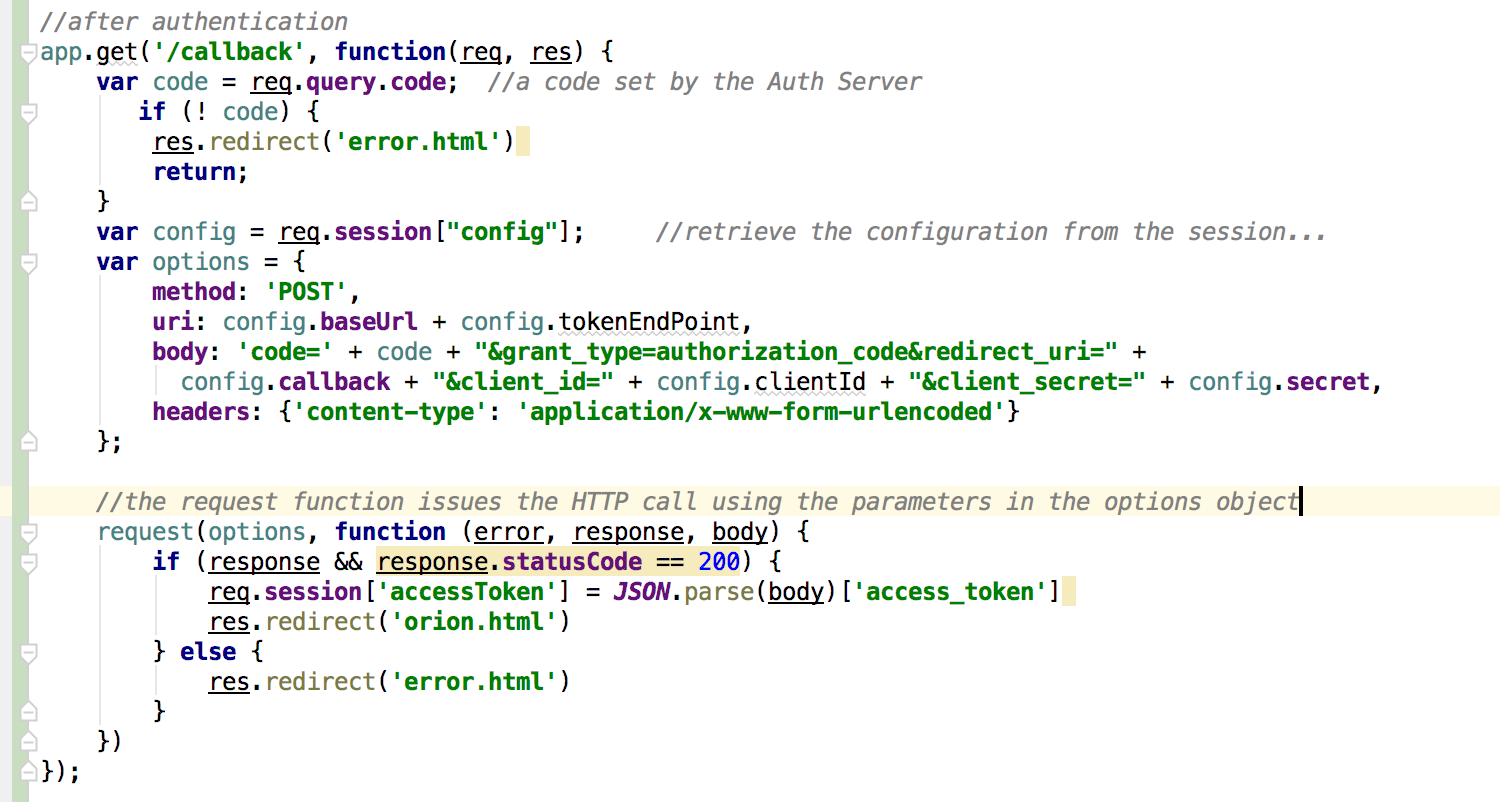
At this point you enter your username and password, and then press login.

Your login details are sent back to the platform (over TLS) to the Authorization server and if they are correct, then the platform will issue another redirect – this time to the callback url that was configured when the app was registered with the platform, and sending a code that it has created.

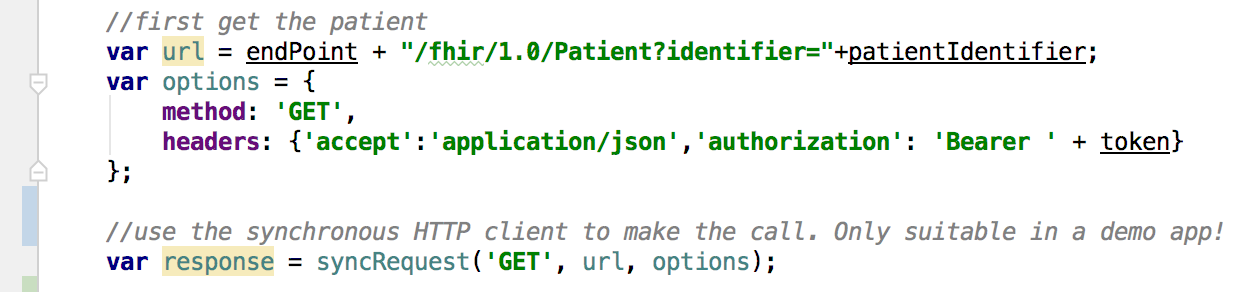
This url is exposed by the app server, and when invoked it:

1. Checks that the authorization server sent it a code
2. Calls the /token endpoint on Authorization server passing across the code and a previously agreed secret and receiving back an access token.
3. Saves the token in the session (so it doesn’t need to be sent back to the client)
4. Re-directs to the main application page.

Here’s the code:



The purpose of all of this is that the Access token (just called the token) is used by the platform to validate any subsequent calls. It is included in every call (in the http Authorization header) and effectively identifies the previously authenticated user – think of it like an electronic passport – so the platform knows who you are, and what you are allowed to access. Here is an example of a call to the platform including the token.



Note that we’ve used a different HTTP library to make the call – synRequest – which waits until the call has completed before continuing. This is OK in a demo, but you’d never do this in a real app.

The token is only valid for a short time – commonly an hour – after which it must be re-issued. Either the user can log in again (which is tedious) or it is possible to get a new one directly using the ‘/refresh’ endpoint on the authorization server. We haven’t implemented that here, though it would be straightforward to do so.

It should be noted that although we’ve implemented all the steps manually here, in real life you’d generally use a library to do so. Implementing security can be tricky and the consequences of failure can be devastating, so using well tested code is always a good idea.