Xavier Hahn · Follow
Sep 26, 2017 · 7 min read

ASP.Net Core & Angular OpenID Connect using Keycloak

How to setup user's authentication for an ASP.Net Core web API and an Angular front end using Keycloak server for user federation and OpenID Connect.

Keycloak

Keycloak is an open source identity provider owned by Red Hat. It allows to easily add authentication to any application and offers very interesting features such as user federation, identity brokering and social login. In the .Net ecosystem, one of its "competitor" would be Identity Server or OpenIddict with Asp.Net Identity.

The main difference between Keycloak and the two other mentioned solution is that it's a standalone product and not a library. This means that it can run "on its own". It's good if you have an ecosystem of application (maybe built with different technologies) and don't want to make one of the application the "master" that does all the user management. It's also a bit more complete, with a fully fledged management user interface and support for multi tenancy.

My goal in this article is to describe how to install a simple instance of Keycloak, secure the API side of an application developed with ASP.Net Core WebApi and secure an Angular client side application that will use the WebApi as a backend.

Enough talking, let's get started

Installing Keycloak

Keycloak has been built with Java on top of the Wildfly application server. There is a quite long installation procedure described in the documentation. In our case, we want to have a simple test server to be able to develop our sample application and will do so using Docker.

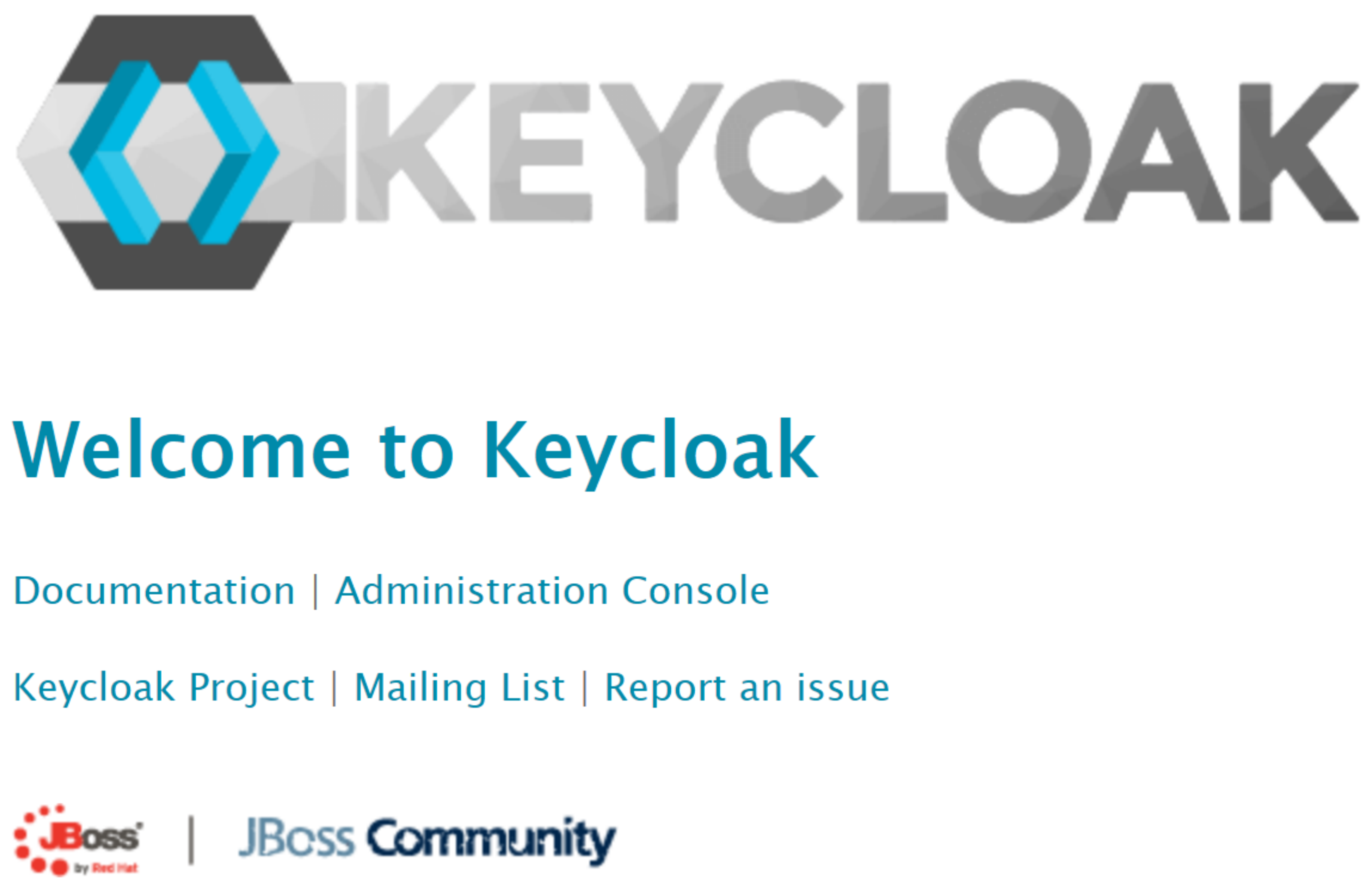
I won't explain here how to install docker, it's pretty straightforward, head to the docker homepage if you need guidance. I'll just assume you already have docker installed on your machine and it's working correctly.

There are quite a few docker images available on Docker Hub. We're going to get the official "standard" package that is basically a ready to use Keycloak installation, complete with the database and everything. You can pass a couple of environment variables to directly create the admin account because none is created by default.

Remark: I wouldn't recommend using that installation of Keycloak in production as there is no redundancy and no backup of the data. I would recommend that you read on Keycloak's high availability setup on the official documentation.

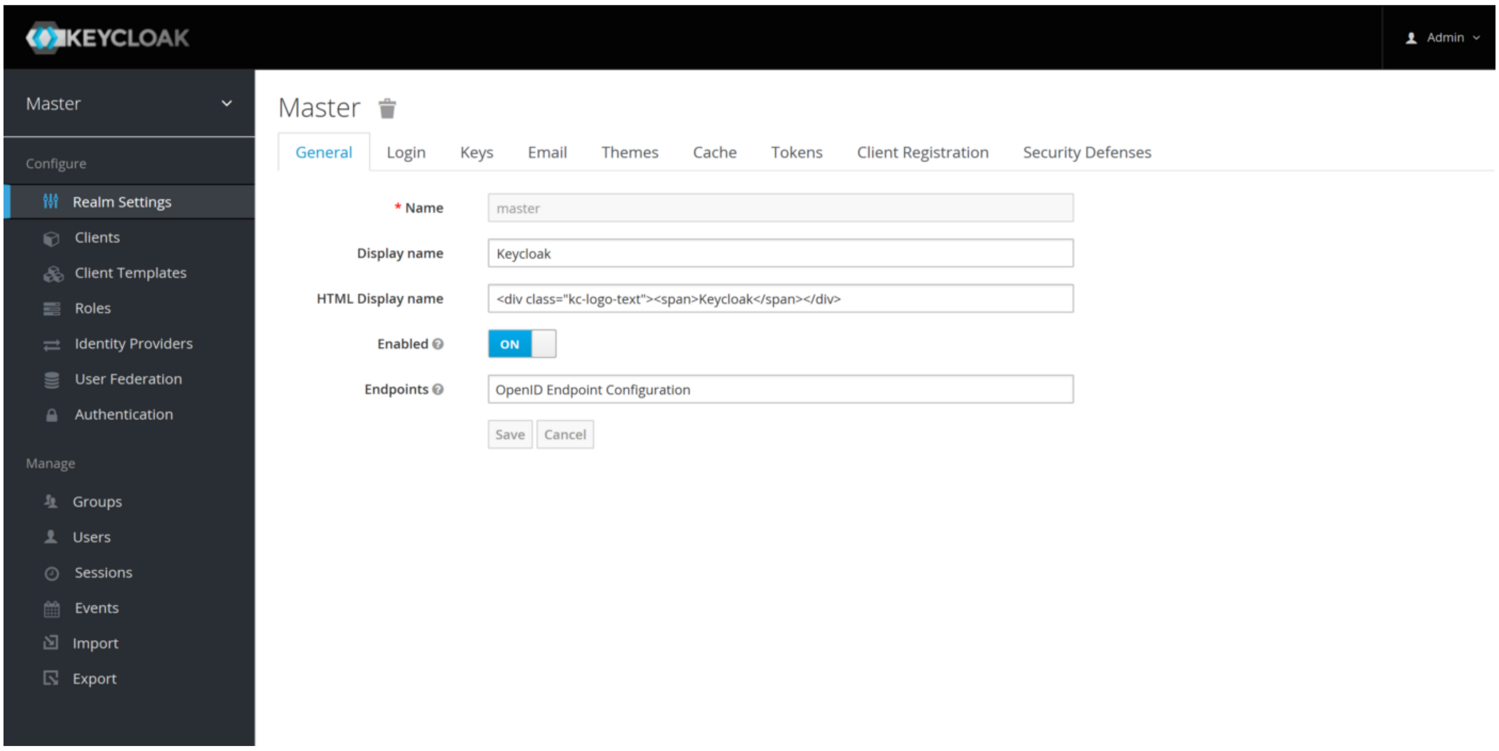
```
docker run --name keycloak -p 8080:8080 -e KEYCLOAK_USER=<USERNAME> -e
KEYCLOAK_PASSWORD=<PASSWORD> jboss/keycloak
```

This will attach Keycloak to the port 8080 of the host machine. Once the machine has started, open your browser and go to localhost:8080. It should display Keycloak's landing page.


KEYCLOAK
Welcome to Keycloak
Documentation | Administration Console
Keycloak Project | Mailing List | Report an issue
JBoss by Red Hat | JBoss Community

Keycloak's default landing page

Click the *Administration Console* link and connect using the username and password combination that you have set in the Docker run command. You should now see Keycloak's administration page.



Master

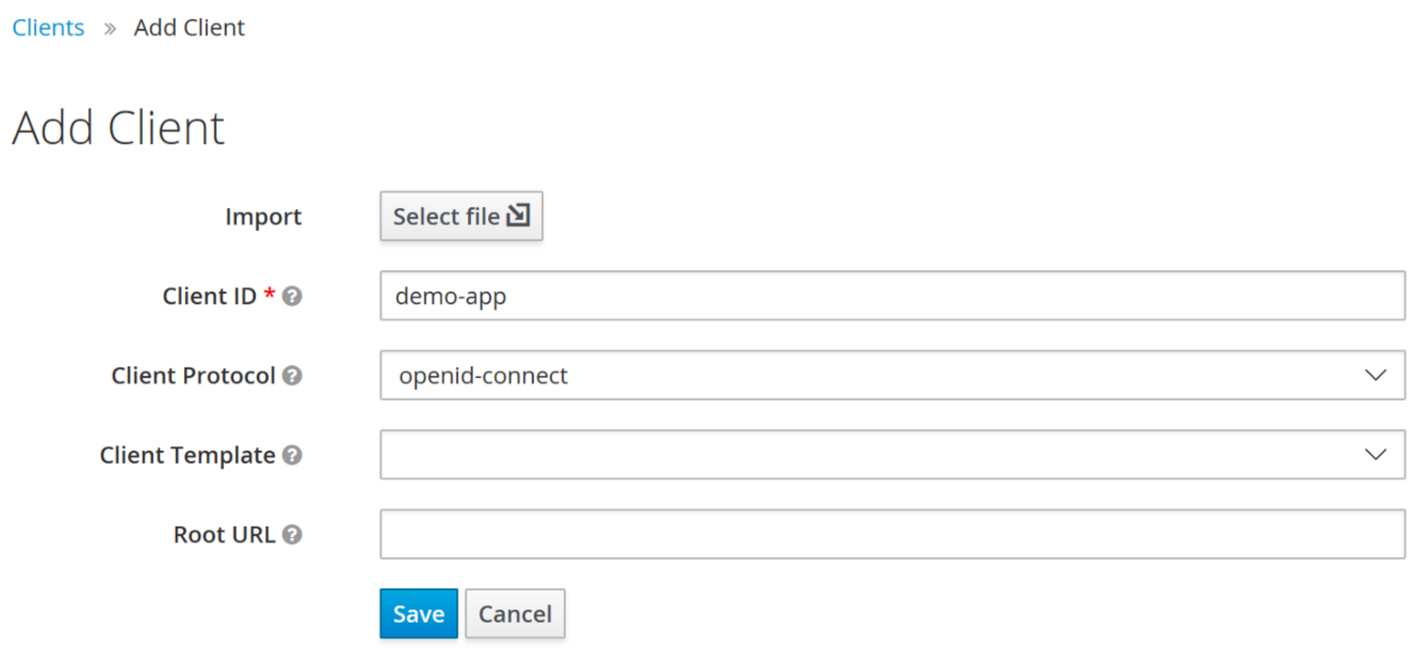
* Name	master
Display name	Keycloak
HTML Display name	`<div class="kc-logo-text"><span>Keycloak</span></div>`
Enabled	ON
Endpoints	OpenID Endpoint Configuration

Keycloak's administration page

Your Keycloak instance is now functional and ready to be used.

Configuring Keycloak

I'm not going to dive into too much details on how to configure Keycloak in this article, the only thing that we need to do to secure our application is creating a *Client*. To do this, click on the *Clients* option of the left menu. It will display a list of some pre-defined clients. Click on the *Create* button on the top right. In the displayed menu, select a name, for example *demo-app* and keep the client protocol as *openid-connect*.



Add Client

Import	Select file
Client ID *	demo-app
Client Protocol	openid-connect
Client Template	
Root URL	

The Add Client menu

Then, make sure to configure a valid redirect URI. To simplify, add * to allow any redirect URI. Enable also the implicit flow by selecting the checkbox.

Creating the Asp.Net Core + Angular app

To bootstrap the creation of the Asp.Net Core + Angular app, since .Net Core 2.0, there is now a generator that creates a Single Page App with Angular directly from the `dotnet` command line.

```
mkdir demo-app
cd demo-app
dotnet new angular
npm install
```

It's important to call the `npm install` command after the end of the generator, it's not run automatically.

Run the application using `dotnet run` and check that everything is working as expected.

Securing the API

Now we are going to setup the WebApi side to be able to secure it based on authentication and roles. That part was the most confusing for me, at first I thought that the application itself would have to manage the tokens and I couldn't wrap my head on how that would be done. After a few trying and error and finally understood that it was Keycloak's role to generate and manage the authentication Tokens and that the application should simply receive that token and "check" with Keycloak that it is indeed correct and coming from him. Implementing that part is very simple.

First, install the Microsoft.AspNetCore.Authentication.JwtBearer Nuget package into the solution.

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Now, open the `Startup.cs` file and add the following to the `ConfigureServices` method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.SpaServices.Webpack;
using Microsoft.Extensions.Configuration;
```

```csharp
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Http;

namespace demo-app
{
    public class Startup
    {
        public Startup(IHostingEnvironment env, IConfiguration configuration)
        {
            Configuration = configuration;
            Environment = env;
        }

        public IConfiguration Configuration { get; }
        public IHostingEnvironment Environment { get; }

        // This method gets called by the runtime. Use this method to add services to the contain
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();

            services.AddAuthentication(options =>
            {
                options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
                options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
            }).AddJwtBearer(o =>
            {
                o.Authority = Configuration["Jwt:Authority"];
                o.Audience = Configuration["Jwt:Audience"];
                o.Events = new JwtBearerEvents()
                {
                    OnAuthenticationFailed = c =>
                    {
                        c.NoResult();

                        c.Response.StatusCode = 500;
                        c.Response.ContentType = "text/plain";
                        if (Environment.IsDevelopment())
                        {
                            return c.Response.WriteAsync(c.Exception.ToString());
                        }
                        return c.Response.WriteAsync("An error occured processing your authentica
                    }
                };
            });
        }

        // This method gets called by the runtime. Use this method to configure the HTTP request
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions
                {
                    HotModuleReplacement = true
                });
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
            }

            app.UseStaticFiles();

            app.UseAuthentication();

            app.UseMvc(routes =>
            {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Home}/{action=Index}/{id?}");

                routes.MapSpaFallbackRoute(
                    name: "spa-fallback",
                    defaults: new { controller = "Home", action = "Index" });
            });
        }
    }
}
```

Startup.cs hosted with ❤ by **GitHub** **view raw**

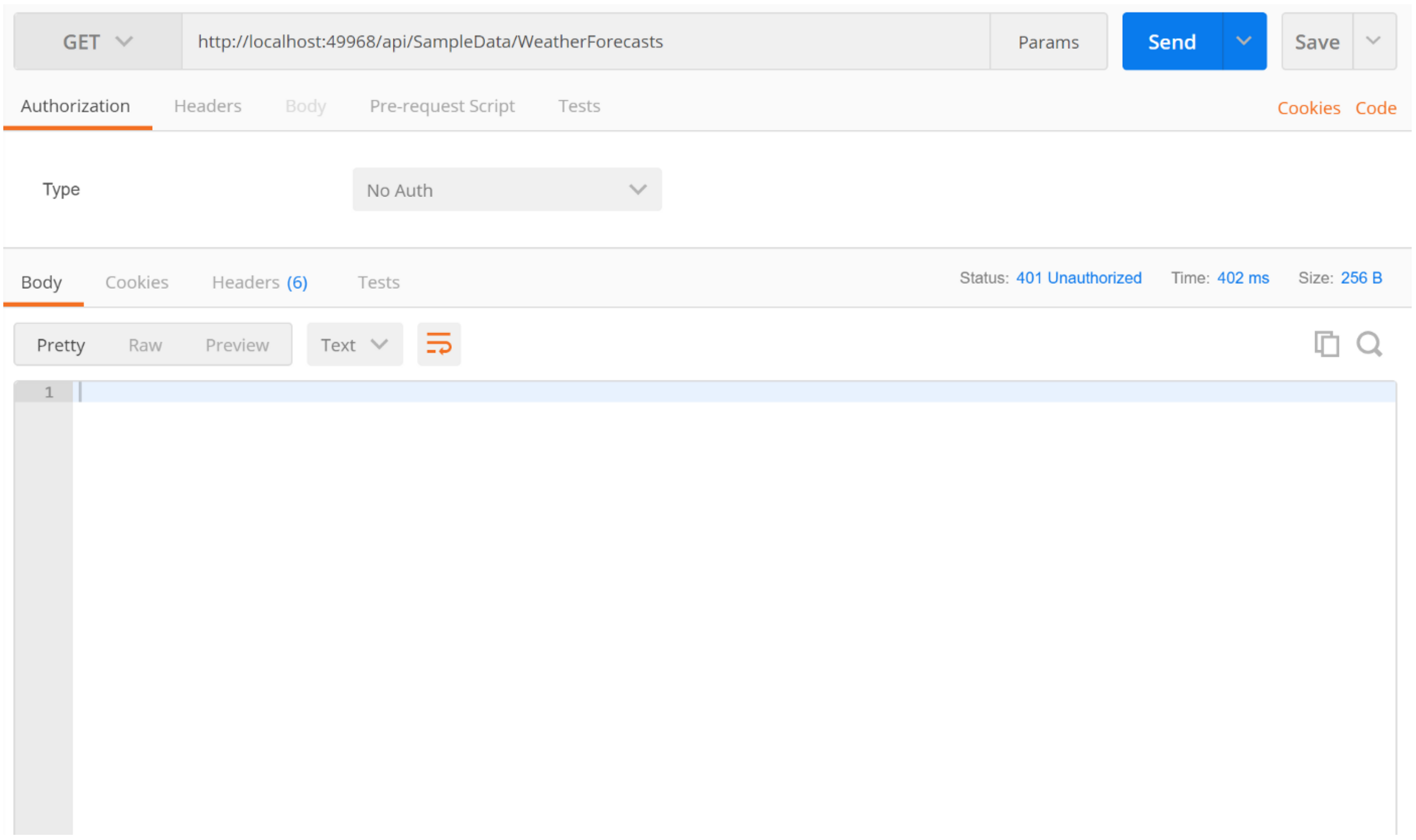
You should then add the Authority and Audience in your configuration. Authority is the URL of your Keycloak instance and realm, in our case `https://localhost:8080/auth/realms/Master`, and the Audience is the name of the Client ID of the client that we created earlier: demo-app. Add this to your configuration file for example, or to environment variables.

Now, all we have to do to secure an API endpoint is to add an `Authorize` attribute to the endpoint method. This will add a check that the method was called with an Authentication header containing a valid JWT token. Let's leverage the existing demo code and add an attribute to the `SampleDataController` created by the scaffolding.

```
[Authorize]
[Route(“api/[controller]”)]
public class SampleDataController : Controller {...}
```

You can add the attribute to either the class (like I did here), in which case it will secure all the methods in the controller, or only on a method to secure just an API endpoint.

Let's run the application again and try to call the `WeatherForecast` method of the `SampleDataController`. I'm using Postman to do the REST API calls.


GET http://localhost:49968/api/SampleData/WeatherForecasts

Authorization — Type: No Auth

Status: 401 Unauthorized Time: 402 ms Size: 256 B

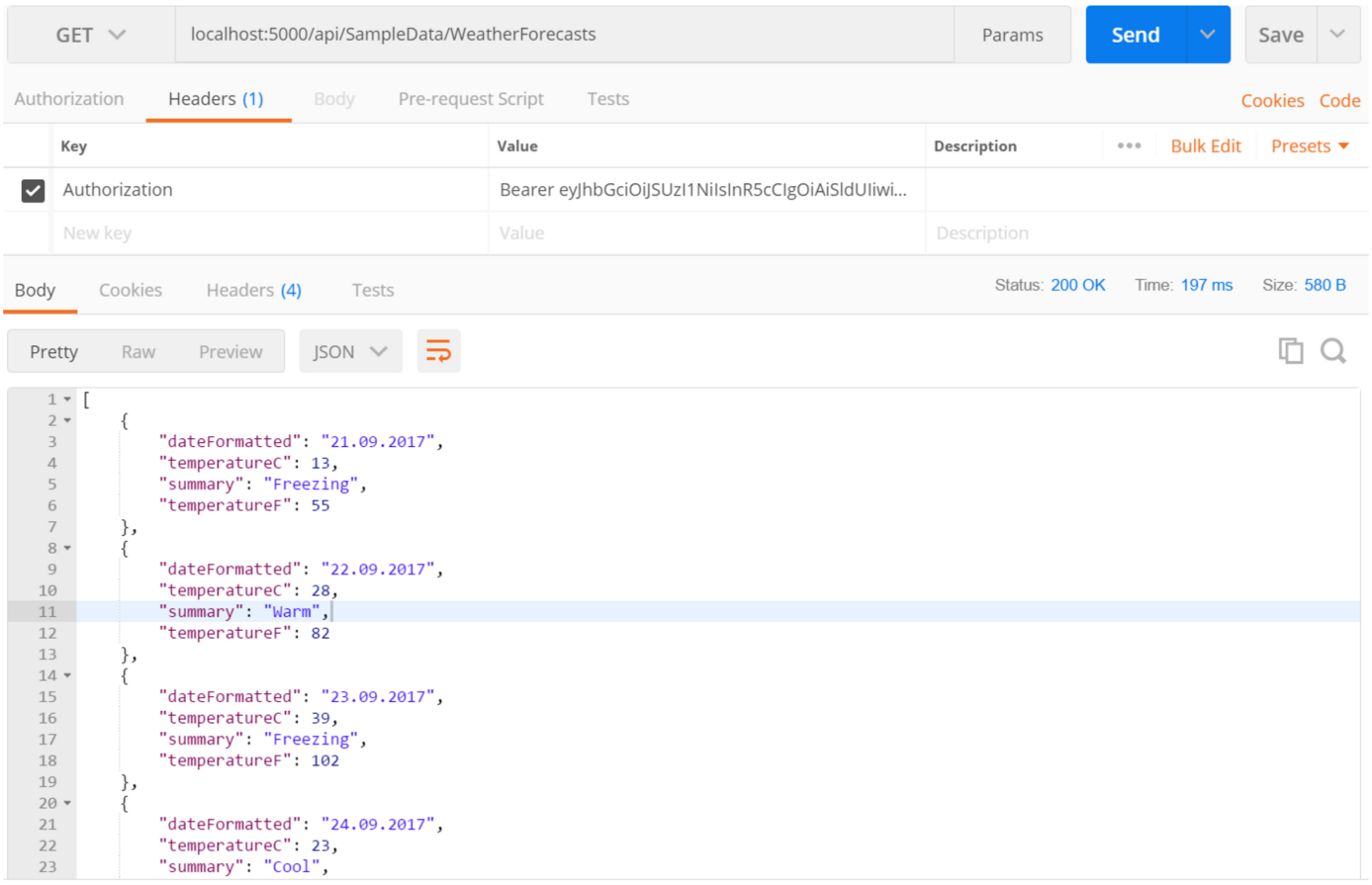
Result of the call with the [Authorize] attribute (401 Unauthorized response)

We can continue our test with Postman by requesting Keycloak a token and adding it to our request. Click the Authorization tab (under the “GET” drop down) and select OAuth 2.0. Click the *Get new access token* button and enter the following options:

Token name: demo
Auth URL: http://localhost:8080/auth/realms/master/protocol/openid-connect/auth
Token URL: http://localhost:8080/auth/realms/master/protocol/openid-connect/token
Client ID: demo-app

Then, click the *Request token* button. There should be a browser window that displays with Keycloak's login. Enter your credentials and click Login. Keycloak will display the newly created token in the list of *Existing token*. Select it, select *Add token to: header* and click *Use token*. It will be added to the header in an Authorization parameter.

Let's call the method again and verify that it's working as expected.



GET localhost:5000/api/SampleData/WeatherForecasts

	Key	Value	Description
✓	Authorization	Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwi...	
	New key	Value	Description

Status: 200 OK Time: 197 ms Size: 580 B

```
[
    {
        "dateFormatted": "21.09.2017",
        "temperatureC": 13,
        "summary": "Freezing",
        "temperatureF": 55
    },
    {
        "dateFormatted": "22.09.2017",
        "temperatureC": 28,
        "summary": "Warm",
        "temperatureF": 82
    },
    {
        "dateFormatted": "23.09.2017",
        "temperatureC": 39,
        "summary": "Freezing",
        "temperatureF": 102
    },
    {
        "dateFormatted": "24.09.2017",
        "temperatureC": 23,
        "summary": "Cool",
```

The result of the call with the valid JWT token passed

Now that the API is secured, let's do the Angular part.

Implementing implicit flow in Angular

OpenId connect's implicit flow is based on redirecting the user to the identity provider's login page and uses redirect URIs to verify the client's identity. It is the only "flow" that doesn't necessitate the use of a secret. This means that it's the only valid flow that can be used by a purely client-side application like we are writing with Angular that can't really keep a secret.

To implement the implicit flow, I'm going to use an open source library. There are two libraries that I found and that are certified by the OpenId foundation, and they have a very similar name: angular-auth-oidc-client by Damien Bowden and angular-oauth2-oidc by Manfred Steyer. Both are very similar in their implementation and both projects are well supported. Here, I'll use the angular-oauth2-oidc library. A lot of the explanation and code samples are coming from the library's Readme.

Let's start by installing the library using NPM

```
npm install angular-oauth2-oidc --save
```

Then, you need to import the module in your `app.module` in our case, we will add it to the `app.module.shared.ts` file. Simply add the import of the `OAuthModule` and add it to the `NgModule` import property.

```
import { OAuthModule } from 'angular-oauth2-oidc';
[...]

@NgModule({
  imports: [
    [...]
    HttpModule,
    OAuthModule.forRoot()
  ],
  declarations: [
    AppComponent,
    HomeComponent,
    [...]
  ],
  bootstrap: [
    AppComponent
  ]
})
export class AppModule {
}
```

app.module.shared.ts hosted with ❤ by **GitHub** **view raw**

Then, we'll create a file in the `ClientApp/app` folder and call it `auth.config.ts.` It will export the configuration for the Oidc.

```
import { AuthConfig } from 'angular-oauth2-oidc';
```

```

export const authConfig: AuthConfig = {

  // Url of the Identity Provider
  issuer: 'http://localhost:8080/auth/realms/master',

  // URL of the SPA to redirect the user to after login
  redirectUri: window.location.origin,

  // The SPA's id.
  // The SPA is registerd with this id at the auth-server
  clientId: 'demo-app',

  // set the scope for the permissions the client should request
  // The first three are defined by OIDC.
  scope: 'openid profile email',
  // Remove the requirement of using Https to simplify the demo
  // THIS SHOULD NOT BE USED IN PRODUCTION
  // USE A CERTIFICATE FOR YOUR IDP
  // IN PRODUCTION
  requireHttps: false
}
```

auth.config.ts hosted with ❤ by **GitHub** **view raw**

Then, we need to configure the oauth service when the application starts. We'll modify the `app.component.ts` file. We just inject a `OAuthService` from the constructor and call a method to setup the configuration that we created above and try the login.

```
import { Component } from '@angular/core';
import {
  OAuthService,
  JwksValidationHandler } from 'angular-oauth2-oidc';
import { authConfig } from '../../auth.config';
@Component({
  selector: 'app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(private oauthService: OAuthService) {
    this.oauthService.configure(authConfig);
    this.oauthService.tokenValidationHandler =
      new JwksValidationHandler();
    this.oauthService.loadDiscoveryDocumentAndTryLogin();
  }
}
```

app.component.ts hosted with ❤ by **GitHub** **view raw**

Next step is to add the login menu to the navigation bar. Let's edit the `navmenu.component.ts` file and create login and logout methods.

```
export class NavMenuComponent {
   constructor(private oauthService: OAuthService) { }

   login(){ this.oauthService.initImplicitFlow(); }
   logout(){ this.oauthService.logOut(); }

   get givenName() {
      let claims = this.oauthService.getIdentityClaims();
      if(!claims) return null;
      return claims.given_name;
   }
}
```

navmenu.component.ts hosted with ❤ by **GitHub** **view raw**

and the html, add this to the end of the navigation menu.

```
<li *ngIf="!givenName">
   <button class="btn btn-default" (click)="login()">
      Login
   </button>
</li>
<li *ngIf="givenName">
   <h1>Hello, {{givenName}}</h1>
</li>
<li *ngIf="givenName">
   <button class="btn btn-default" (click)="logout()">
      Logout
   </button>
</li>
```

navmenu.component.html hosted with ❤ by **GitHub** **view raw**

Finally, let's add the authentication token to the headers of the call to the API. I also took the opportunity to move the call to the API out of the constructor to an OnInit method, it's a pretty bad idea to have it in the constructor anyway.

```
import { Component, Inject, OnInit } from '@angular/core';
import { Http, Headers } from '@angular/http';
import { OAuthService } from 'angular-oauth2-oidc';

@Component({
    selector: 'fetchdata',
    templateUrl: './fetchdata.component.html'
})
export class FetchDataComponent implements OnInit {
    public forecasts: WeatherForecast[];

    constructor(
        private http: Http,
        @Inject('BASE_URL') private baseUrl: string,
        private oauthService: OAuthService) { }

    ngOnInit(): void {
        var headers = new Headers({
            "Authorization": "Bearer " + this.oauthService.getAccessToken()
        });

        this.http.get(this.baseUrl + 'api/SampleData/WeatherForecasts',
            { headers: headers })
            .subscribe(result => {
                this.forecasts = result.json() as WeatherForecast[];
            }, error => console.error(error));
    }
}

interface WeatherForecast {
    dateFormatted: string;
    temperatureC: number;
    temperatureF: number;
    summary: string;
}
```

As you can see, it’s just a matter of getting the access token using the oauth library and adding it to the header in an Authorization property.

Conclusion

We’re done with this basic sample, we went around all the important things and shown how to use Keycloak with an Asp.Net Core application.

I have put the full project available on GitHub here:
https://github.com/Gimly/SampleNetCoreAngularKeycloak

I have also a follow-up article that will guide you on adding authorization using Keycloak’s groups.
https://medium.com/@xavier.hahn/adding-authorization-to-asp-net-core-app-using-keycloak-c6c96ee0e655