SECURITY

Secure a .NET Core API using Bearer Authentication

LES JACKSON 23RD FEB '20 0

In this step by step tutorial, we secure a .NET Core API using Bearer authentication, JSON Web Tokens, (JWT), and Azure Active Directory (AAD).

What You'll Learn

By following the steps in this article, you'll learn about:

- The Bearer Authentication Scheme and JSON Web Tokens

- The code steps required to secure your API End Points
- How to write a client app to authenticate and access the secured API

Ingredients

The following ingredients are required to follow along:

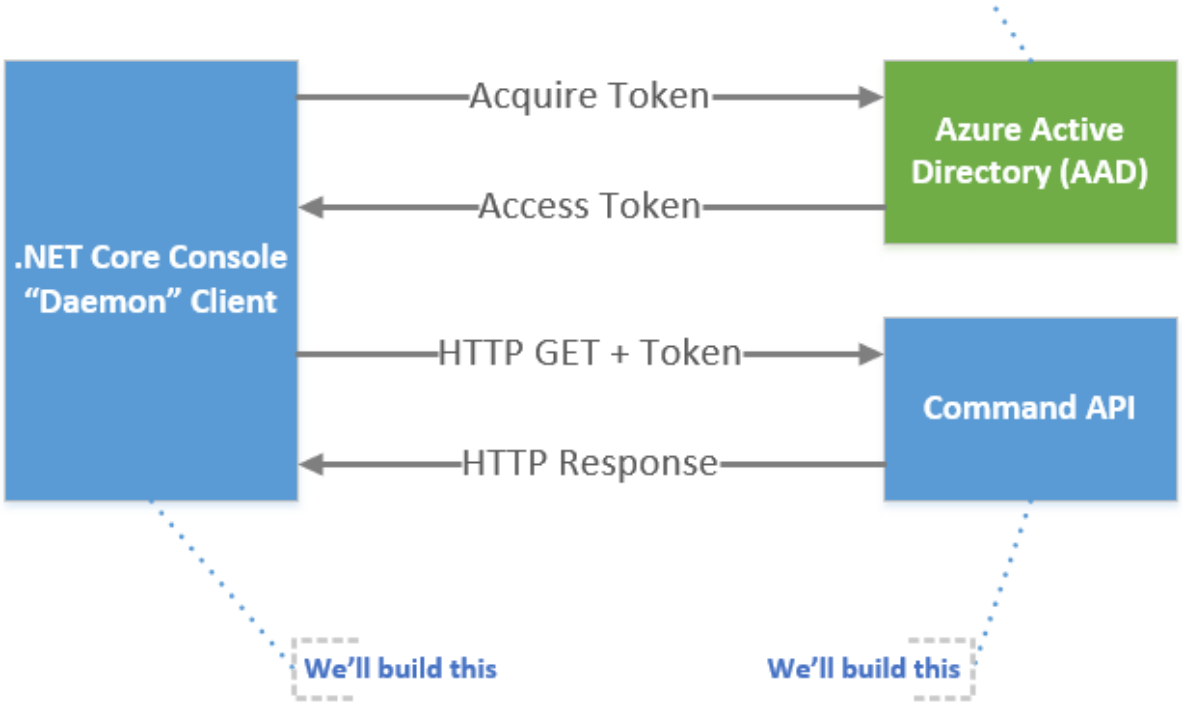
- .NET Core SDK 3.1
- Text Editor (I suggest VS Code which is awesome and free)
- An Account on Azure ("Free" but does require sign up)
- Postman (Free API test client – **available here**)

Source Code

The complete source code for both the API and Client projects can be **found here on GitHub**.

Our Authentication Use Case

Before delving into the technicalities of our chosen authentication scheme, I just wanted to cover our *authentication use case*. For this example we are going to "secure" our API by using Azure Active Directory, (AAD), and then create and configure a client, (or daemon), app with the necessary privileges to authenticate through and use the API. We are not going to leverage "interactive" user-entered User Ids and passwords. This use case is depicted below:



Overview of Bearer Authentication

There are a number of authentication schemes that we could have used, a non-exhaustive list is provided below:

Basic Authentication

A common, relatively simple authentication scheme. Requires the supply of a user name and password that's then encoded as a Base64 string, this is then added to the authorisation header of a HTTP request. Natively this is not encrypted so is not that secure, unless you opt so make requests over HTTPS, in which case the transport is encrypted.

Digest

Bearer

Token based authentication scheme where anyone in possession of a valid “token” can gain access to the associated secured resources, in this case our API. Considered secure, it is widely adopted in industry and is the scheme, (specified in **RFC 6750**), we’ll use to secure our API.

NTLM

Microsoft-specific authentication scheme, using Windows credentials to authenticate. Perfectly decent, secure scheme but as it’s somewhat “proprietary”, (and I’m trying to avoid that), we’ll leave our discussion there for now.

Bearer Tokens Vs JSON Web Tokens

The use of “tokens” in Bearer authentication is a central concept. A token is issued to a requestor, (in this case a daemon client), and the client, (or “bearer of the token”), then presents it to a secure resource in order to gain access.

So what’s JWT?

JWT, (or JSON Web Tokens), is an encoding standard, (specified in **RFC 7519**), for tokens that contain a JSON payload. JWT’s can be used across a number of applications, however in this instance we’re going to use JWT as our encoded token through our use of Bearer authentication.

In short:

- JWT is a specific implementation of bearer tokens, in particular those with a JSON payload.

Build Steps

OK so that's enough theory, we now move on to the build part of our tutorial, I've listed all the steps we need to perform below:

1. Create our API
2. Register our API in Azure Active Directory
3. Expose our API in Azure Active Directory
4. Update our API Manifest
5. Revisit and "Secure" an API Endpoint
6. Register our client application in Azure Active Directory
7. Create a Client Secret in Azure Active Directory (for our client)
8. Configure API permissions (for our client)
9. Create our client app
10. Test it altogether

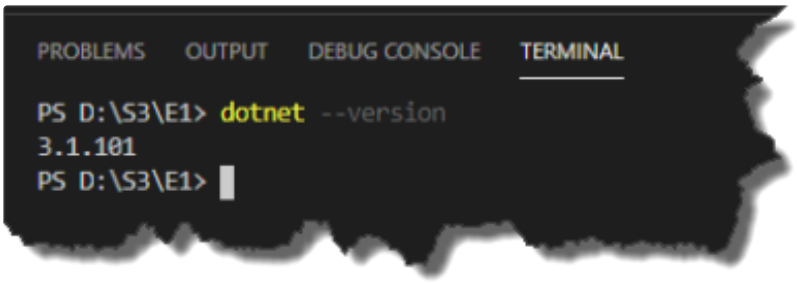
Step 1 – Create Our API

for us, (except of course making the necessary changes to secure it!). If you'd like a step by step on how to create a full-featured ASP .NET Core API, please **read my tutorial** on the subject.

Open an command line – I'll be using the integrated terminal in my VS Code editor, but you can use whatever you like. First ensure we have the .NET Core SDK installed by typing:

```
dotnet --version
```

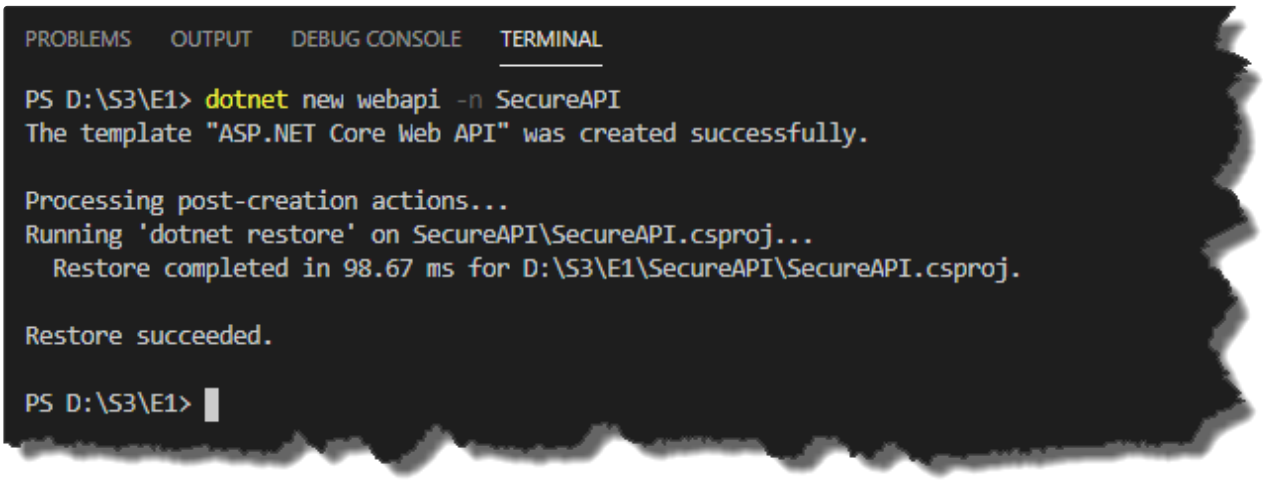
You should see output similar to that shown below:



If you get an error, or your version of the framework is significantly behind what I'm using here, I'd suggest installing / updating **.NET Core SDK**.

Move into your "*working directory*", this is just a fancy term for where you want to create you project files and type:

This will create a new *webapi* template project for us with the name *SecureAPI*, you should see output similar to this:

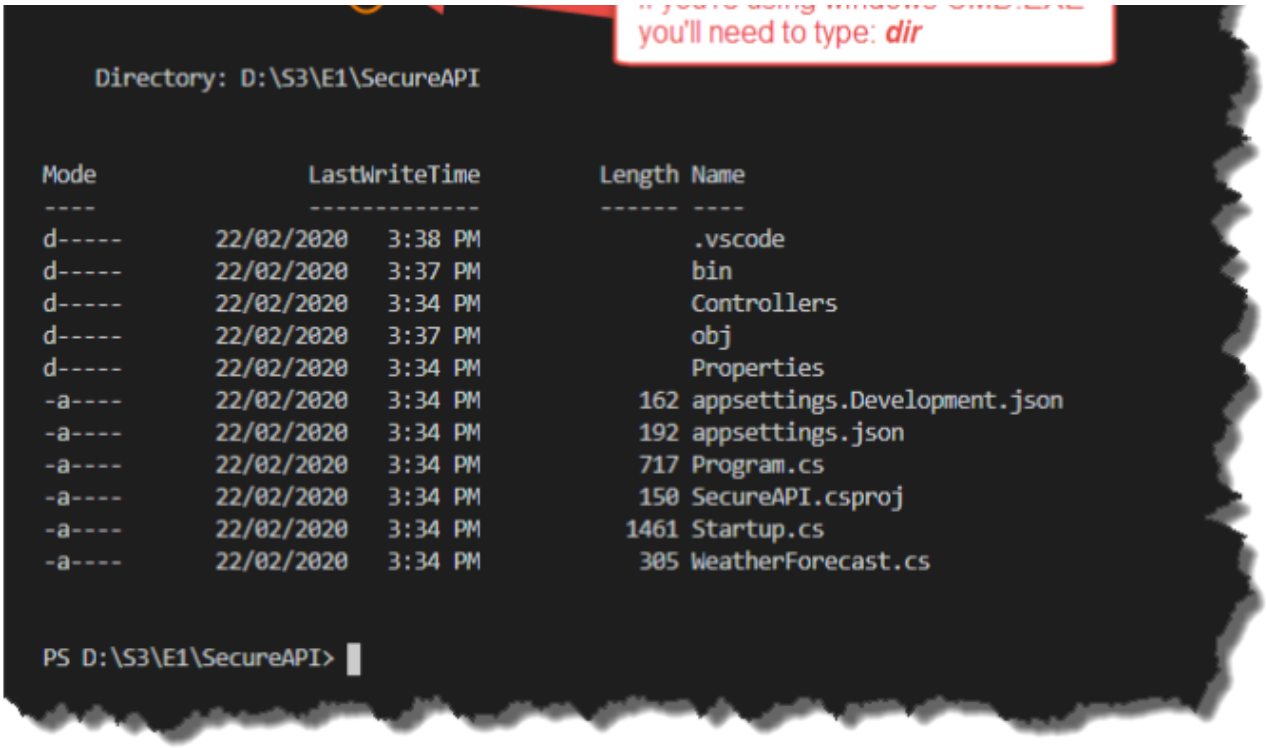


If you're using VS Code you can now type:

```
code -r SecureAPI
```

This will "recursively" open the project folder in VS Code. If you're using some other editor, just open the *SecureAPI* project folder for editing.

At the command line again, (to open the integrated command line in VS Code type Ctrl + `), ensure you're "in" the project folder by listing the contents, you should see something like:



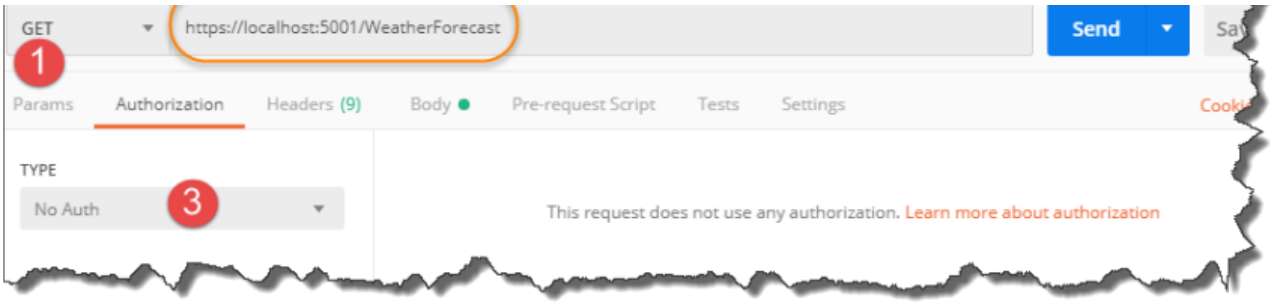
Assuming you're in the correct place, let's run our API to ensure it's working by typing the following at the command line:

```
dotnet run
```

This should run up our API, you'll see that it's listening for requests on:

- http://localhost:5000
- https://localhost:5001

To test that the API endpoint is responding to requests, open Postman, and configure a GET request as follows:



1. Ensure the verb is set to “GET”
2. Enter one of the listening URLs here, (make sure the port number is correct for either HTTP or HTTPS requests)
3. Make a note that we are not specifying any authorisation type, (our API is currently unsecured)
4. When ready make the request

Note: If you’ve just downloaded Postman and never used it before, **take a look at my tutorial on creating an API** to see how to configure it for 1st time use.

The request should be successful, and you should get the following results in Postman:

1. 200 OK Response code

At the command line hit CTRL + C to stop the API running, and in your text editor open the *WeatherForecastController.cs* file:


```
namespace SecureAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
        };

        private readonly ILogger<WeatherForecastController> _logger;

        public WeatherForecastController(ILogger<WeatherForecastController> logger)
        {
            _logger = logger;
        }

        [HttpGet]
        public IEnumerable<WeatherForecast> Get()
        {
            var rng = new Random();
            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateTime.Now.AddDays(index),
                TemperatureC = rng.Next(-20, 55),
                Summary = Summaries[rng.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
}
```

Quickly reviewing what we have:

1. Our *WeatherForcastController* class is decorated with the *[Apicontroller]* attribute. This just allows us some out the box behaviors, such as the rendering of the API call output as JSON.

2. The definition of our API “route”. In this case our API will be called with the base pattern: *[server address]/WeatherForecast, e.g. http://localhost/WeatherForecast*

4. We have 1 API endpoint defined, decorated with *[HttpGet]*, which I think is quite obvious what that does, and returning our array of weather elements rendered as JSON

Personally, I don't like the way the API end point is written, but it's perfectly operational and fit for our purposes. This is the end point that we are going to "secure".

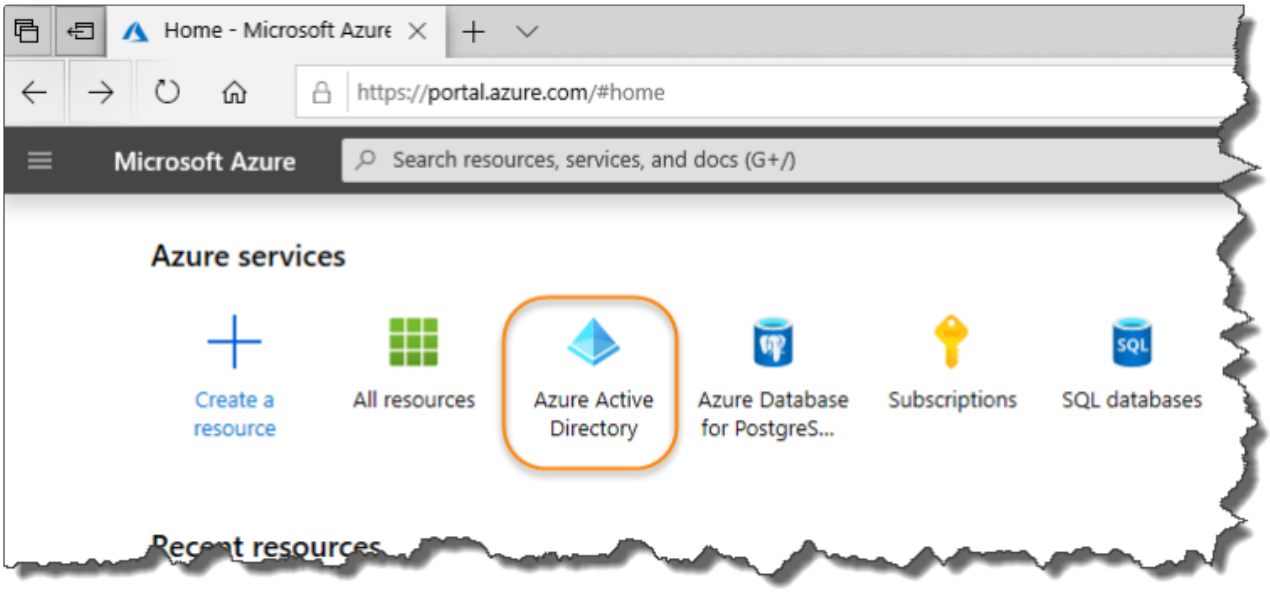
Looking back at how we called the API from Postman, you'll notice that there was no authentication type specified, and we could access the end point without having to supply any further details, our API is unsecured...

We're going to leave our API there for now, but we'll come back to it later as we have to make some changes to it in order to secure it.

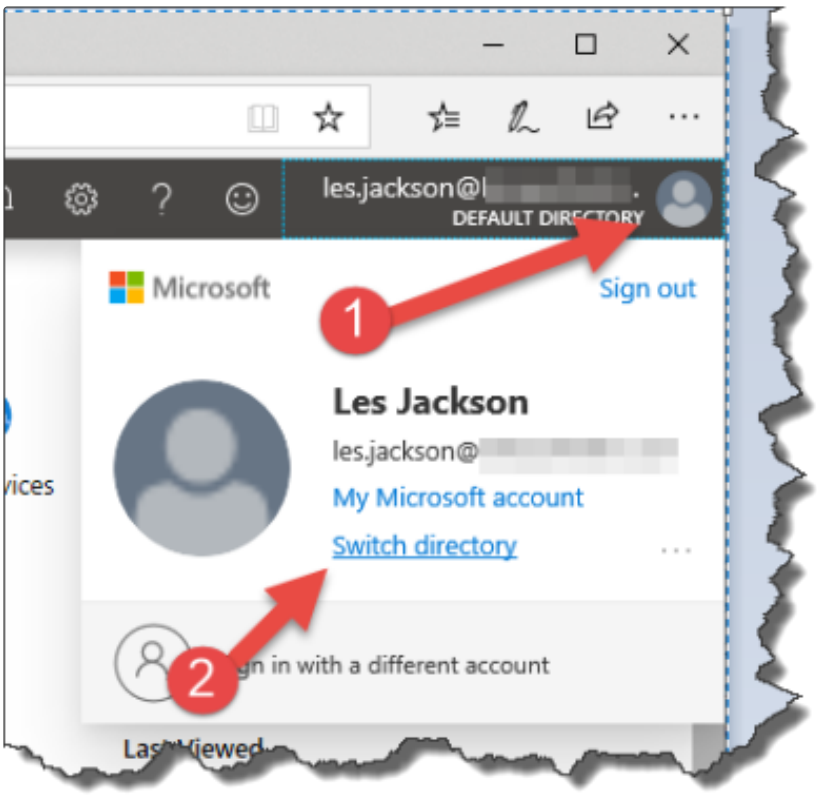
Step 2 – Register Our API in Azure Active Directory

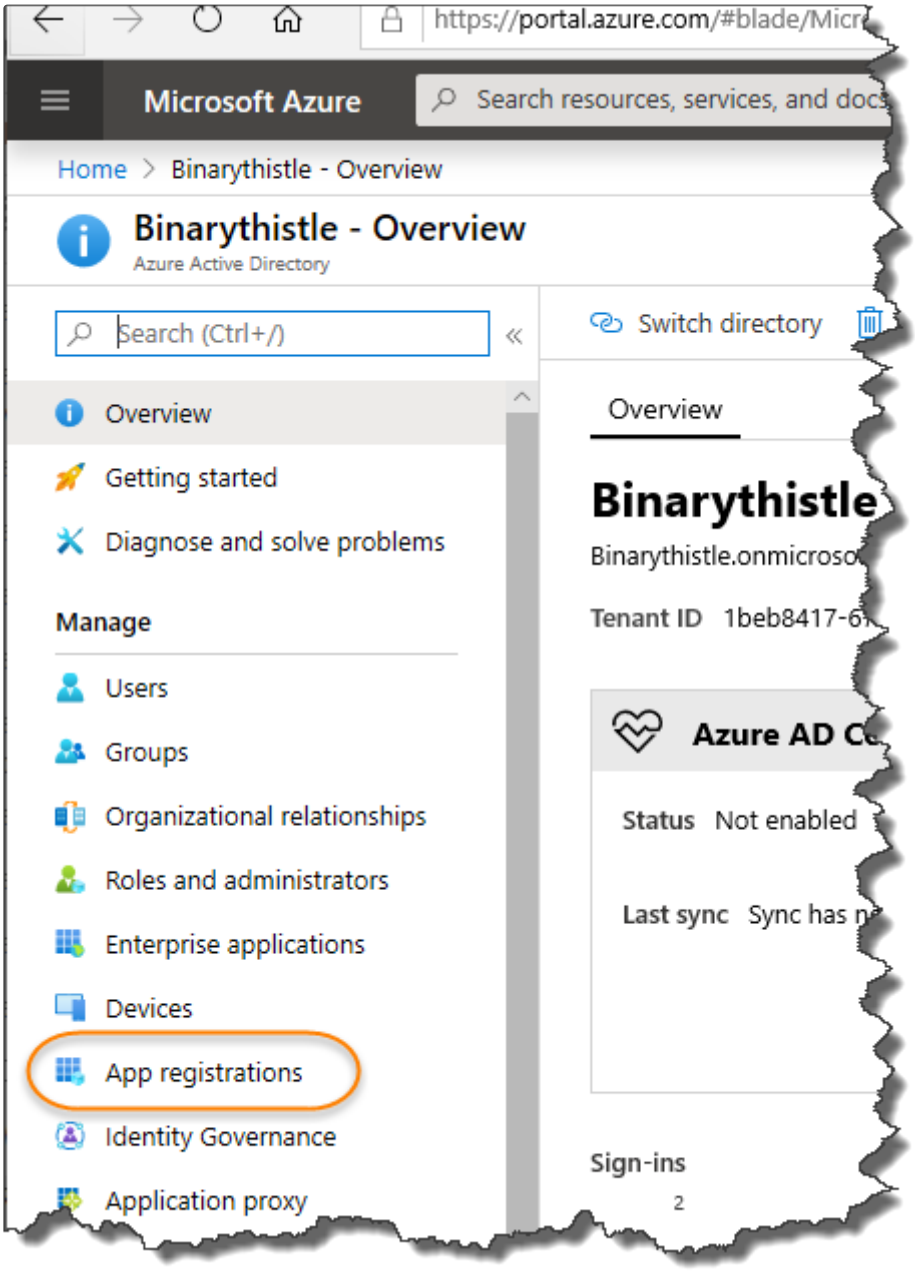
The next step is to register our API in Azure Active Directory, (AAD from now on), as we'll be using this as our Identity and Access Management service, a couple of points to note:

- We could write our own native code to provide the services that AAD is going to supply, however for me personally I'd rather concentrate on unique, value-add features in my API as opposed to writing my own custom Identity and Access Management layer. The AAD product would also be better than anything I could come up with anyway!
- Just because we're using AAD to provide Identity and Access Management to our API, we do not need to have our API running on Azure, (indeed in this tutorial we're just leaving our API on our local development machine)

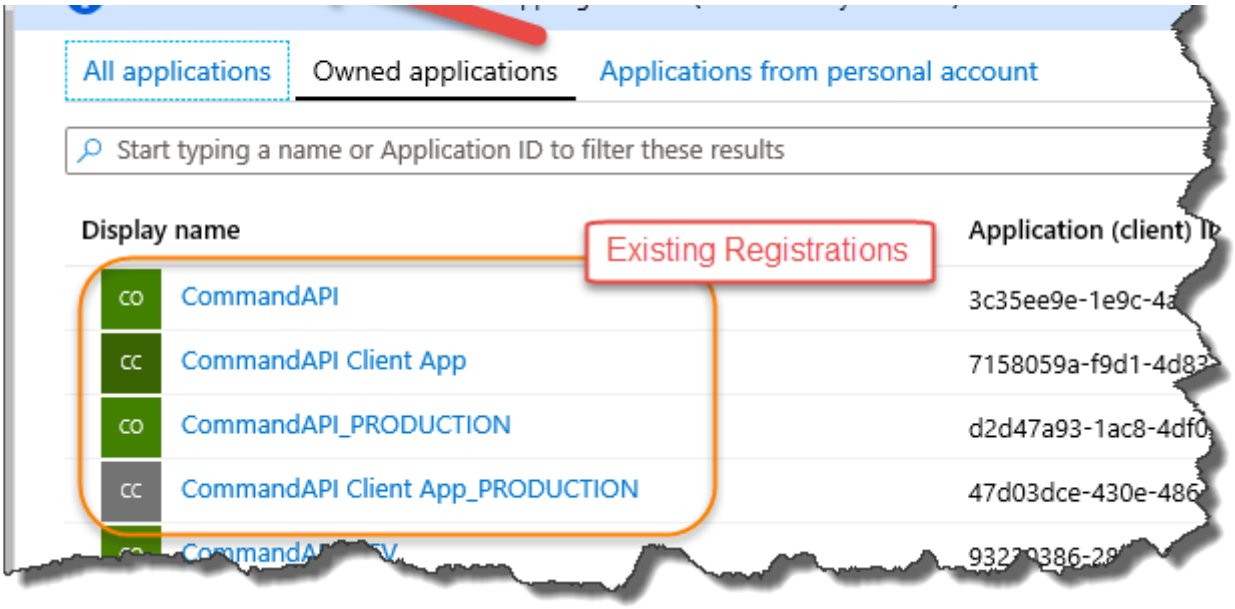


Note: you can have multiple AADs as part of your Azure subscription, and you create a new one in much the same was as you create any new resource. You can then switch between AADs as shown below:

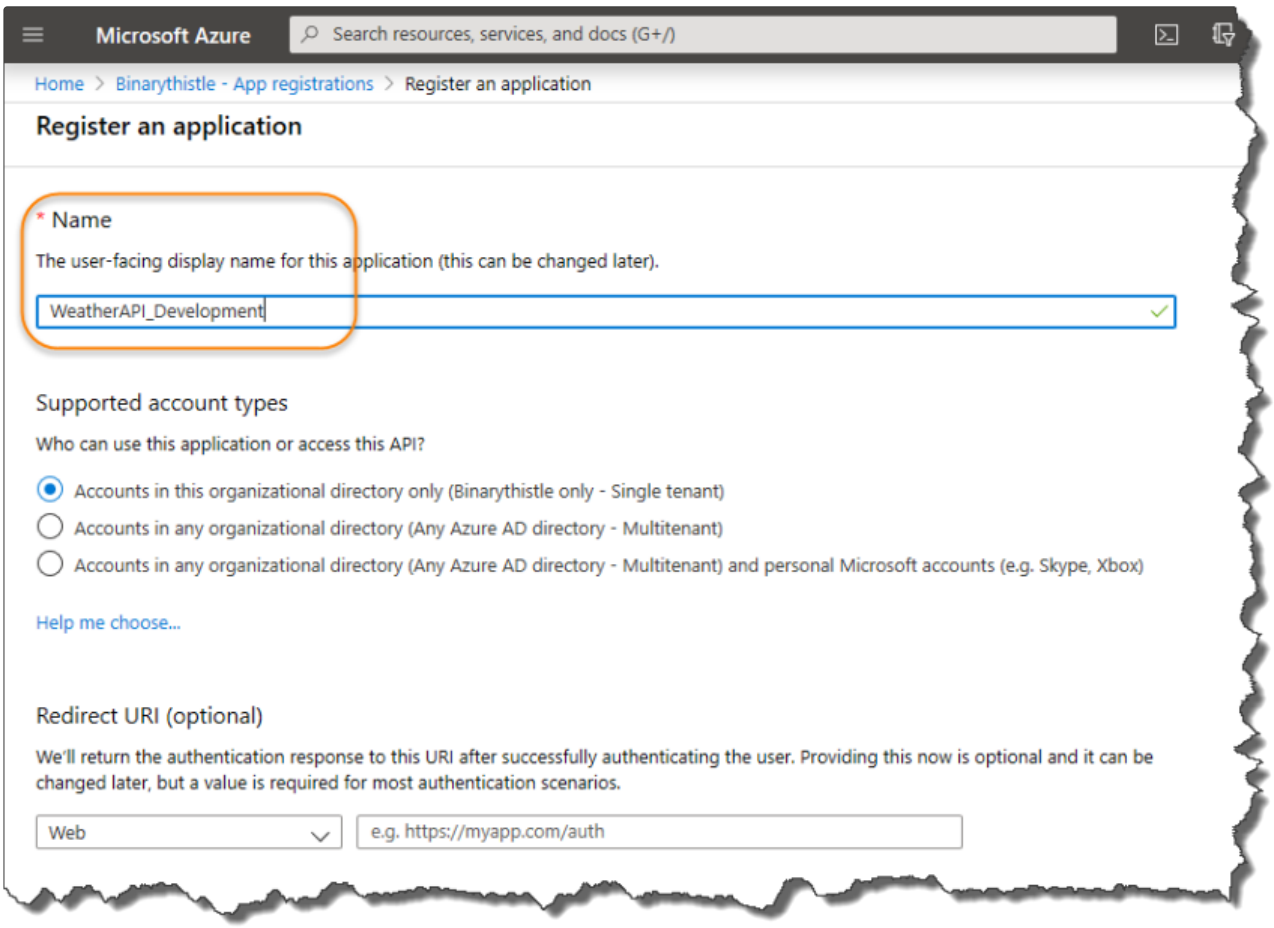




You can see from the example below I already have a few apps registered on my AAD, but we're going to create a new one for our WeatherAPI.

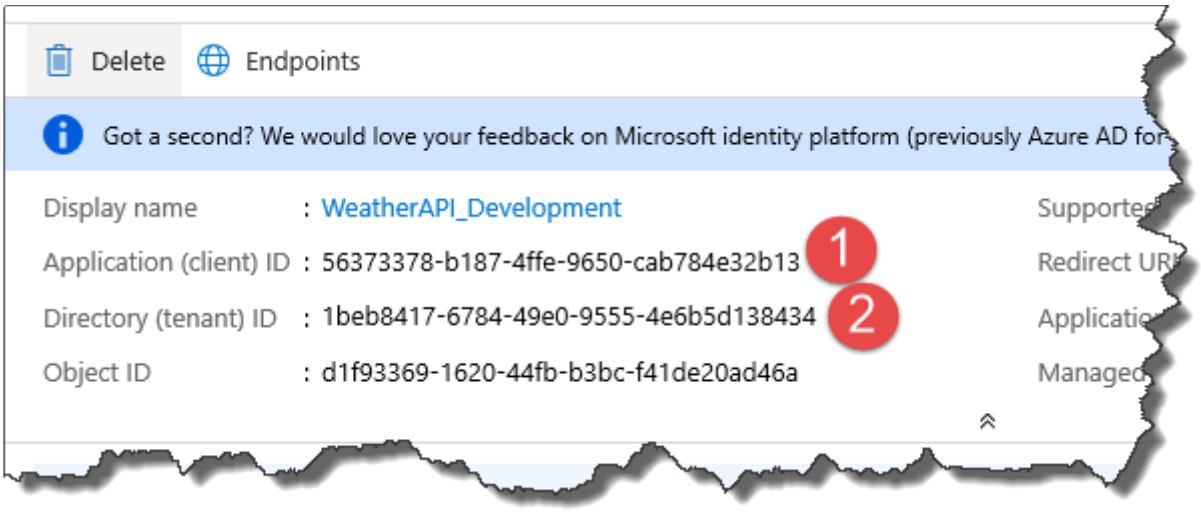


Select “New registration”, and you’ll see:



registrations we may choose to create for different environments). Also ensure that “Accounts in this organization directory only ([Your AAD Name] only – Single tenant)”, is selected.

We don’t need a Redirect URI, so click “register” to complete the initial registration, after which you’ll be taken to the overview screen:



Here we are introduced to the first 2 important bits of information that we need to be aware of:

1. Application (client) ID
2. Directory (tenant) ID

Going forward I’m going to use the terms Client ID and Tenant ID, but what are they?

CLIENT ID

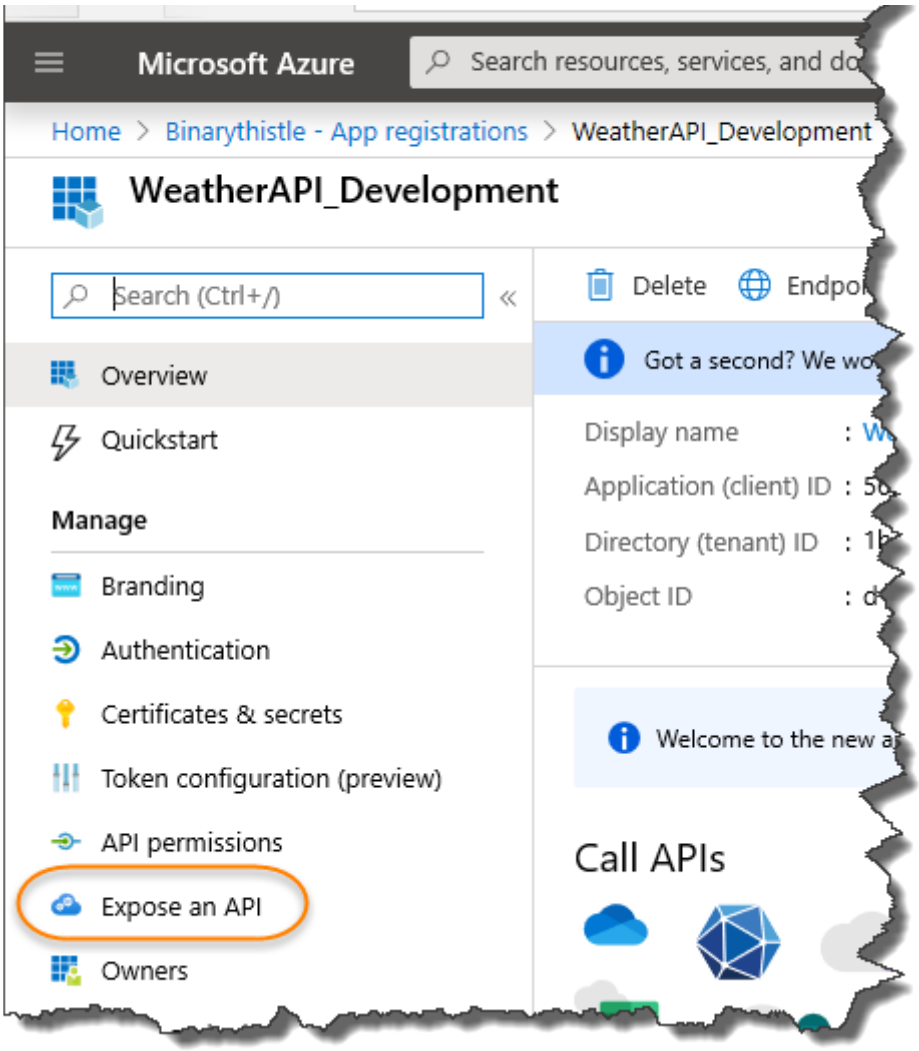
The client ID is essentially just a unique identifier that we can refer to the Weather API in reference to our AAD.

A unique id relating to the AAD we're using, remembering that we can have multiple, (i.e. multi-tenant), AAD's at our disposal.

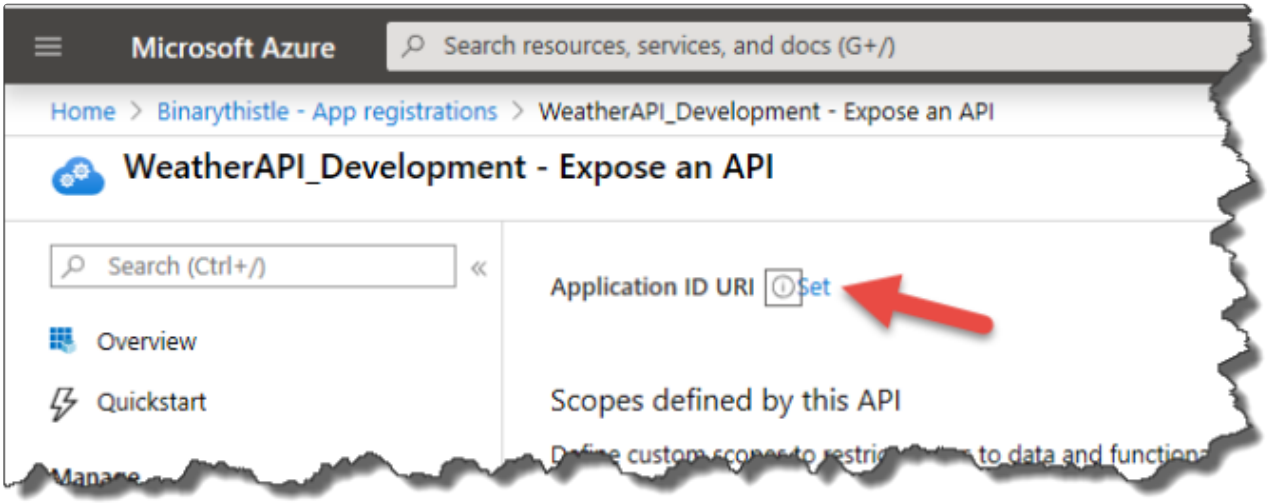
We'll come back to these items later when we come to configuring things at the application end, for now we need to move on to the next step.

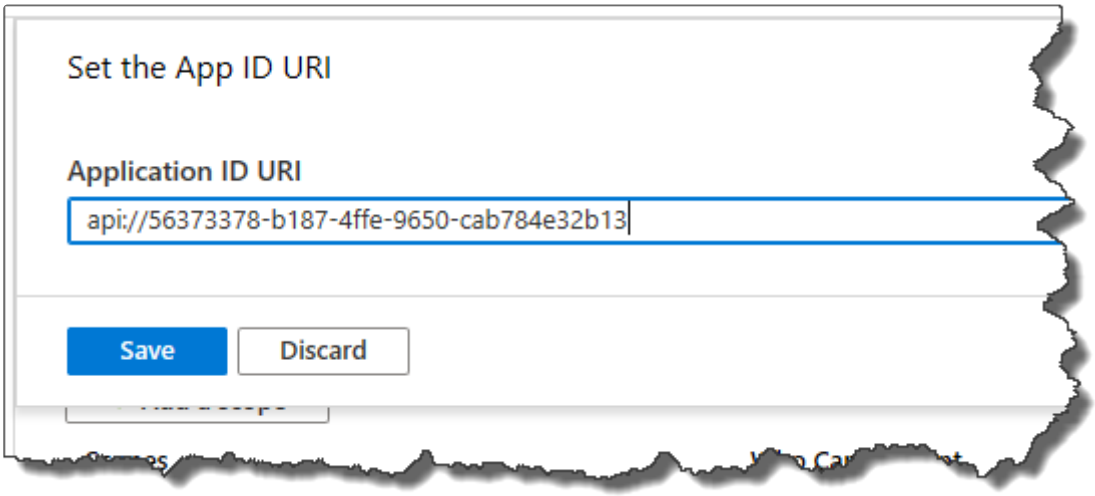
Step 3 – Expose our API in Azure

So far we've merely registered our API, we now need to *expose* it for use, so click on "Expose an API" from our left hand menu options on our *WeatherAPI_Development* registration page:

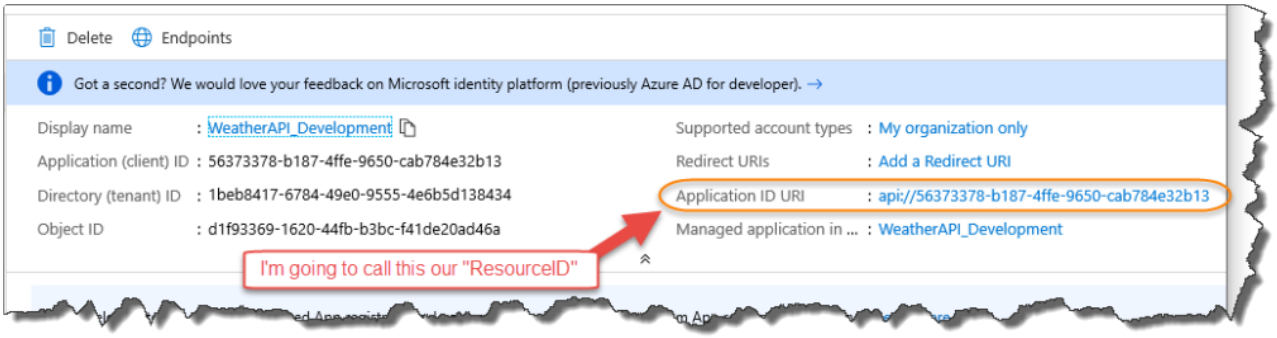


What we need to do here is create an “Application ID URI”, (sometimes referred to as a “Resource ID”), so click on “Set” as shown below:





Click Save and you're done. Clicking back into the overview of the registration and you should see this reflected here too:



We're almost finished with our API configuration in AAD, but have one more bit of configuration to complete, so let's move onto the next step.

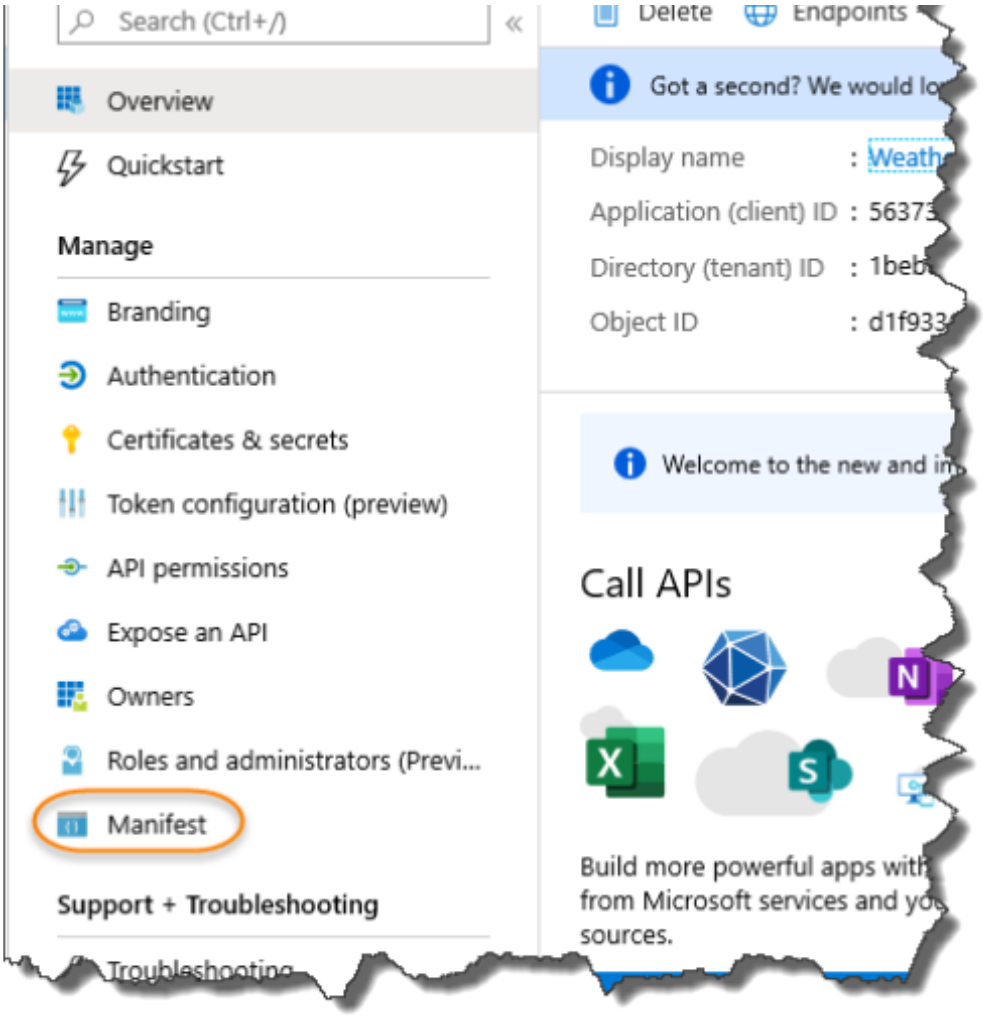
Step 4 – Update our API Manifest

Here we update the *appRoles* section of our application manifest which specifies the type of application role(s) that can access the API. In our case we need to specify a

Anyway, back to the task at hand, we need to insert the following JSON snippet at the appRoles section of our manifest:

```
.
.
.
"appRoles": [
  {
    "allowedMemberTypes": [
      "Application"
    ],
    "description": "Daemon apps in this role can consume the web ap
    "displayName": "DaemonAppRole",
    "id": "6543b78e-0f43-4fe9-bf84-0ce8b74c06a3",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "DaemonAppRole"
  }
],
.
.
.
```

So, click on “Manifest” in the left-hand window of our App Registration config page:



And insert the json above into the correct spot, (essentially updating the existing empty appRoles section):

Make sure you keep the integrity of the json and don't omit or introduce any additional commas, (for example). You can always use something like https://jsoneditoronline.org/ to check.

You can add multiple appRoles to this section, we need only one, although if you do decide to add some additional roles you'll need to ensure that the "id" attribute is a unique GUID.
When completed, don't forget to save the file.

That's it for our API registration in Azure, we need to move over to our API now and make some config and code changes so it can make use of AAD for authorisation.

Step 5 – Revisit & Secure Our API Endpoint

Update AppSettings.JSON

In order for our API to work with the AAD registration we created in the last step, we need to supply the API configuration layer with a few of the elements we just discussed, specifically:

- ResourceId
- Instance (we've not actually discusses this, more on it below)
- TenantId

The "*instance*" is really just a login URL for AAD, and unlike the other 2 config elements, the value of Instance will be the same for everyone, (*ResourceId* and *TenantId* will be unique to you).

So in my case the values I'll have are:

- ResourceId: api://56373378-b187-4ffe-9650-ccb784e32b13
- Instance: https://login.microsoftonline.com/
- TenantId: 1beb8417-6784-49e0-9555-4e6b5d238434

You'll need to obtain your own values for ResourceId and TenantId from the overview section of your API Registration.

Next we need to put these values into our *appsettings.json* file.

purposes of simplicty and brevity, I've decided just to put them in *appsettings.json*.

So, the JSON you'll need to add to appsettings.json is, (again make sure you use your own values for ResouceId and TenantId):

```
.
.
.
"AAD" : {
    "ResourceId" : "api://56373378-b187-4ffe-9650-cab784e32b13",
    "Instance" : "https://login.microsoftonline.com/",
    "TenantId" : "1beb8417-6784-49e0-9555-4e6b5d138434"
  }
.
.
.
```

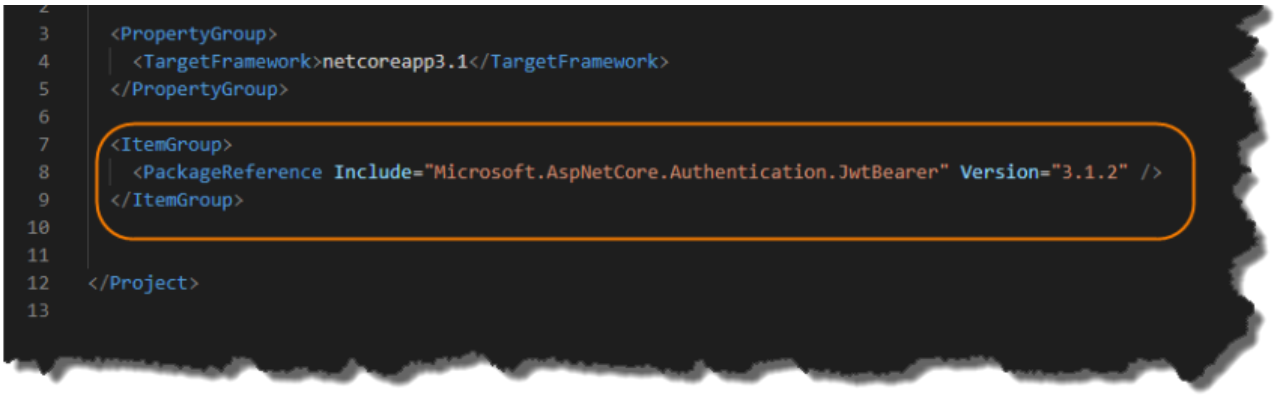
So overall, you're appsettings.json file should look like this:

Update Our Project Packages

Before we start coding, we need to add a new package that will be required to support the code we're going to introduce, so at a command prompt "inside" the API project type:

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

This should successfully add the following package reference to the .csproj file:



Update Our Startup Class

Over in the startup class of our API project we need to update both our *ConfigureServices* and *Configure* methods. First though, add the following using directive to the top of the *startup* class file:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

Update Configure Services

We need to set up bearer authentication in the ConfigureServices method, to do so add the following code, (new code is highlighted in **bold**):

```
        services.AddControllers();
    }
```

to put it in context the code will look like this:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(opt =>
        {
            opt.Audience = Configuration["AAD:ResourceId"];
            opt.Authority = $"{Configuration["AAD:Instance"]}{Configuration["AAD:TenantId"]}";
        });

    services.AddControllers();
}
```

The code above adds authentication to our API, specifically Bearer authentication using JWT Tokens. We then configure 2 options:

- Audience: We set this to the ResourceID of our App Registration in Azure
- Authority: Our AAD Instance that is the token issuing authority

Update Configure

All we need to do now is add authentication & authorization to our request pipeline via the Configure method, (authorization is probably already there by default – if not add it):

```
app.UseAuthorization();
```

As shown below:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Authentication Vs Authorisation

As we've added both Authentication and Authorisation to our request pipeline, I just want to quickly outline the difference between these two concepts before we move on.

- Authentication (the "who"): Verifies who you are, essentially it checks your identity is valid
- Authorisation (the "what"): Grants the permissions / level of access that you have

This is where the rubber hits the road in terms of securing our single API endpoint...

First off add the following using directive at the top of our WeatherForecastController class:

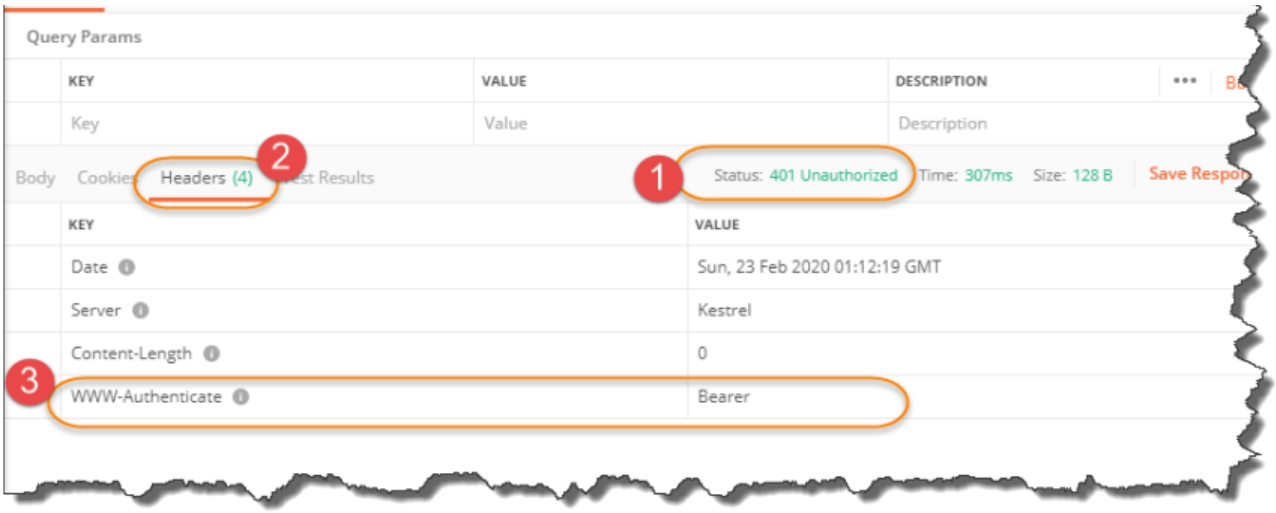
```
using Microsoft.AspNetCore.Authorization;
```

The we simply decorate our API Endpoint code with the *[Authorize]* attribute:

```
[Authorize]
[HttpGet]
public IEnumerable<WeatherForecast> Get()
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

Note: You can choose to either decorate individual endpoints with the *[Authorize]* attribute, (as we've done here), or decorate the entire class, which would lockdown all end points. We only have 1 endpoint in out API so both approaches would have the same effect in this instance!

Save all the new code, build then run the API locally. Once running make a call to our newly protected endpoint in Postman:



Here you will see:

1. We get a 401 Unauthorized response
2. Selecting the return headers we see...
3. That the authentication type is “Bearer”

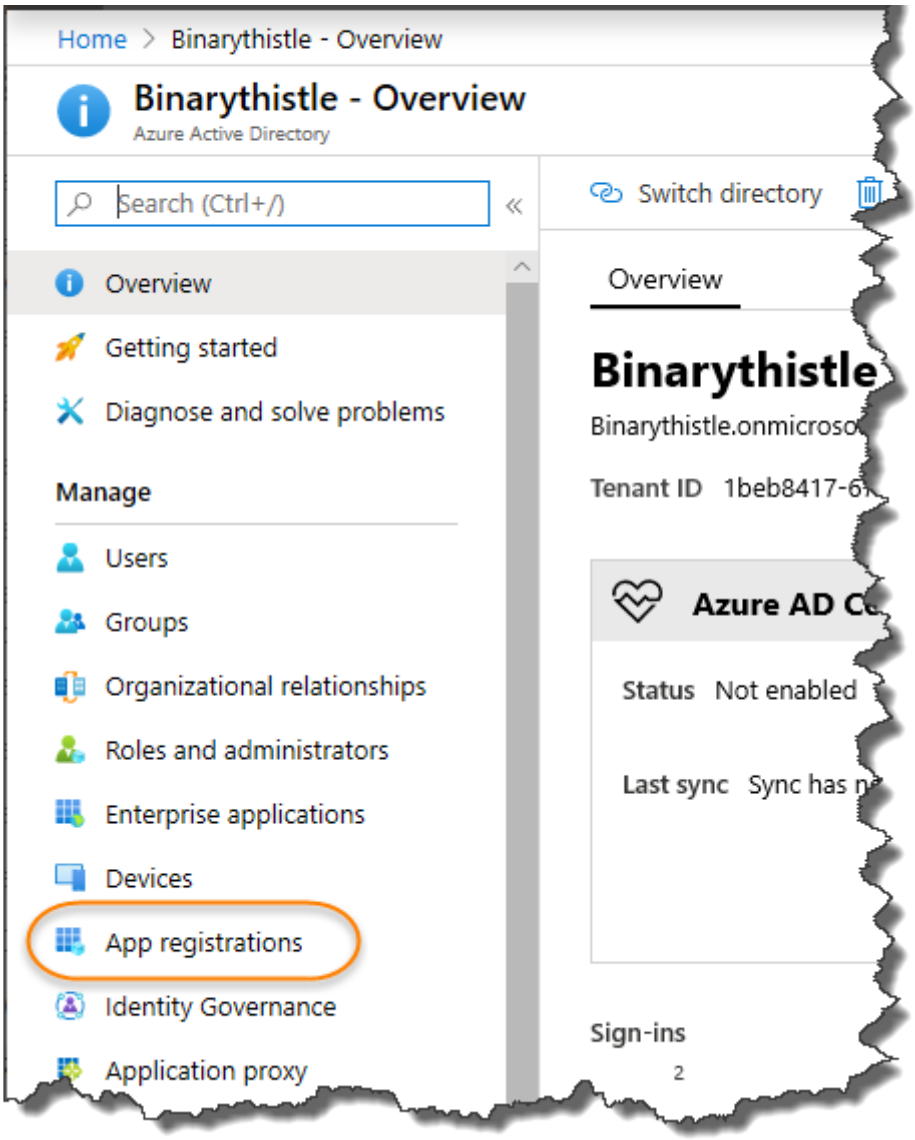
So with that our API is now locked down with Bearer Authenitcation, we now need to move on to creating a client app that is authorised to use API....

Step 6 – Register our Client App in Azure Active Directory

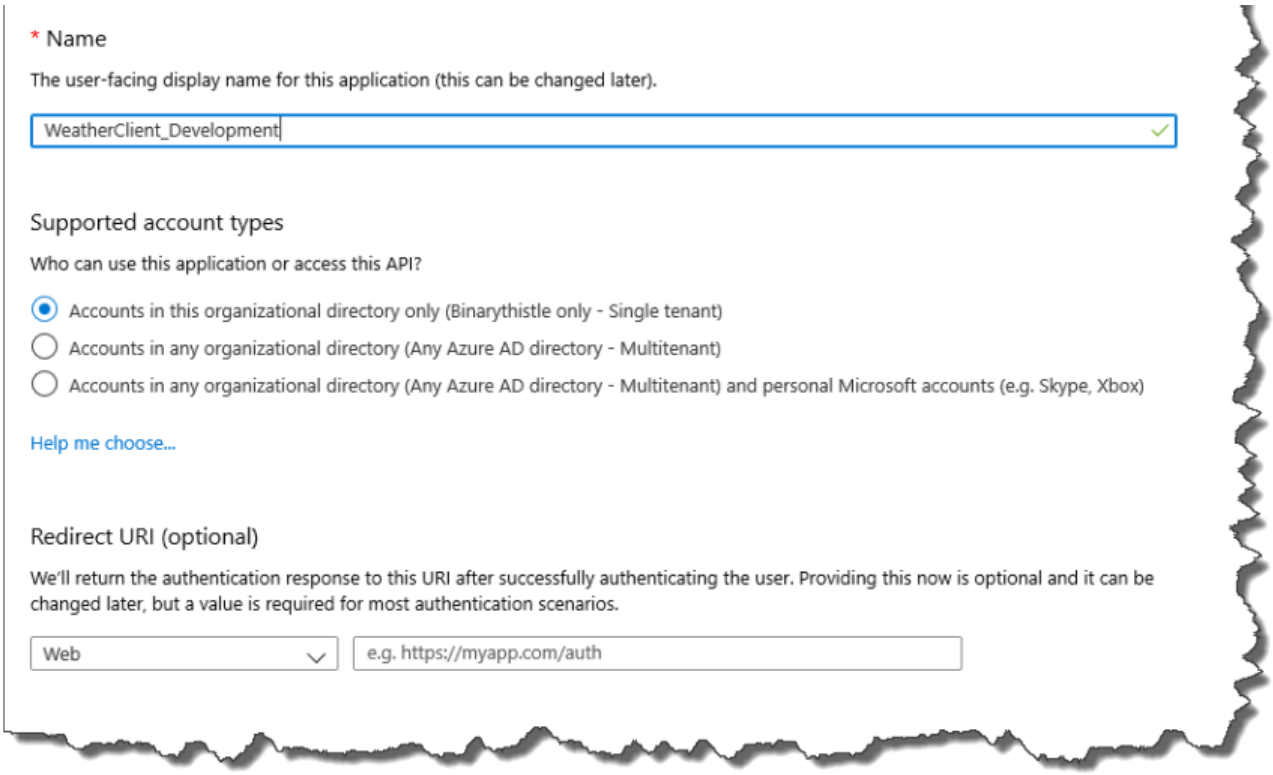
In the sections that follow we’re going to write a simple .NET Core Console application that will act as an authorised “client” of the API. As this is a “daemon app” it needs to run without user authentication interaction, so we need to configure it as such.

> could explore when it comes to consuming an API, for example a user authenticating against AAD, (username / password combo), to grant access to the API.
>
> The use-case I've decided to go with in this example, (a "daemon app"), resonated with me more in terms of a real-world use-case. You may of course disagree...

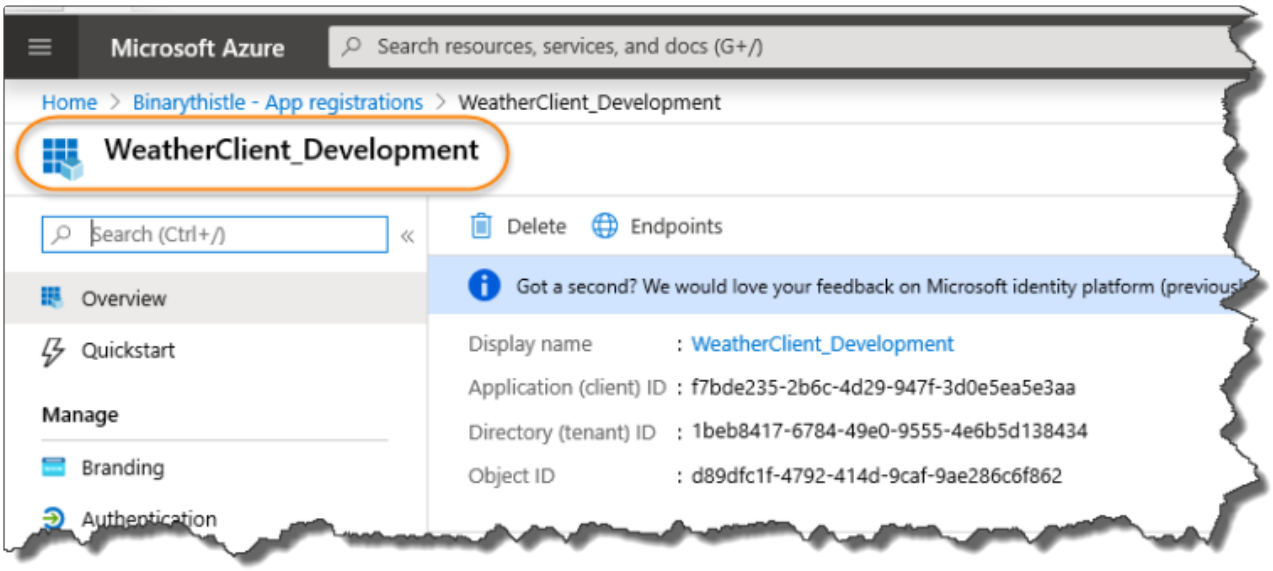
Back over in Azure, select the same AAD that you registered the API in, and select App Registrations once again:



Then select “+ New registration”, and on the resulting screen enter a suitable name for our client app as shown below:

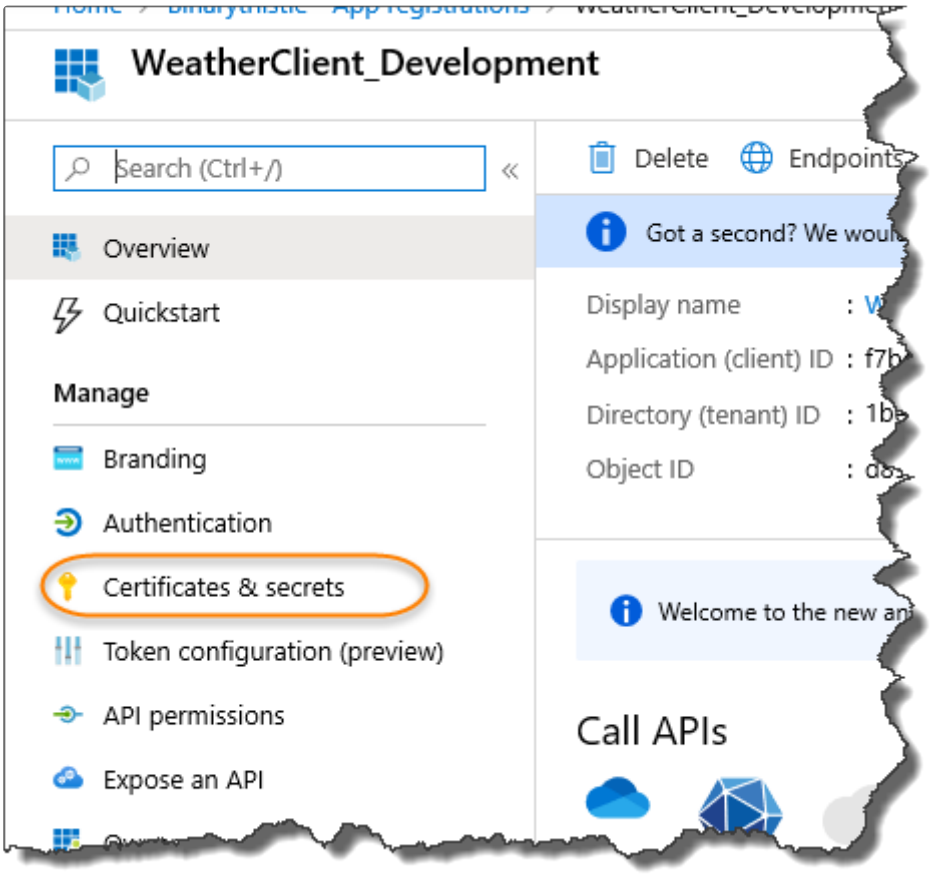


Again, select the Single tenant Supported account type option and click “Register”, this will take you to the overview screen of your new app registration:

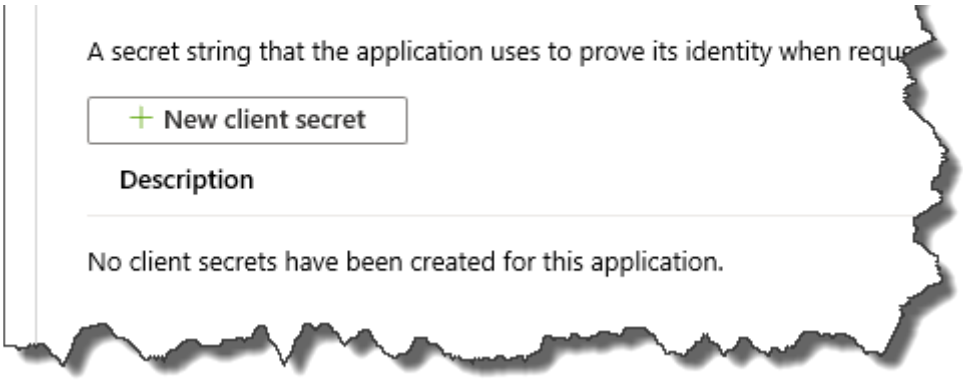


As before it’s prepopulated with certain attributes.

Next, click on “Certificates & secrets” in the left-hand menu:

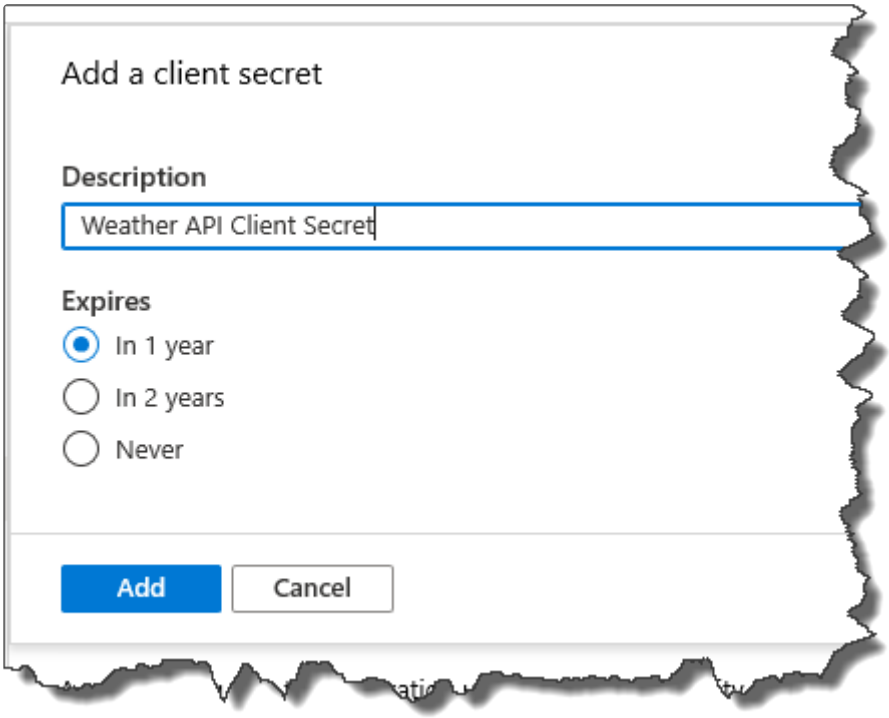


Here we are going to configure a “Client Secret”. This is a unique ID that we will use in combination with our other app registration attributes to identify and authenticate our client to our API. Click “+ New client secret”:



And on the resulting screen give it:

- A description (can be anything but make it meaningful)
- An expiry (you have a choice of options)



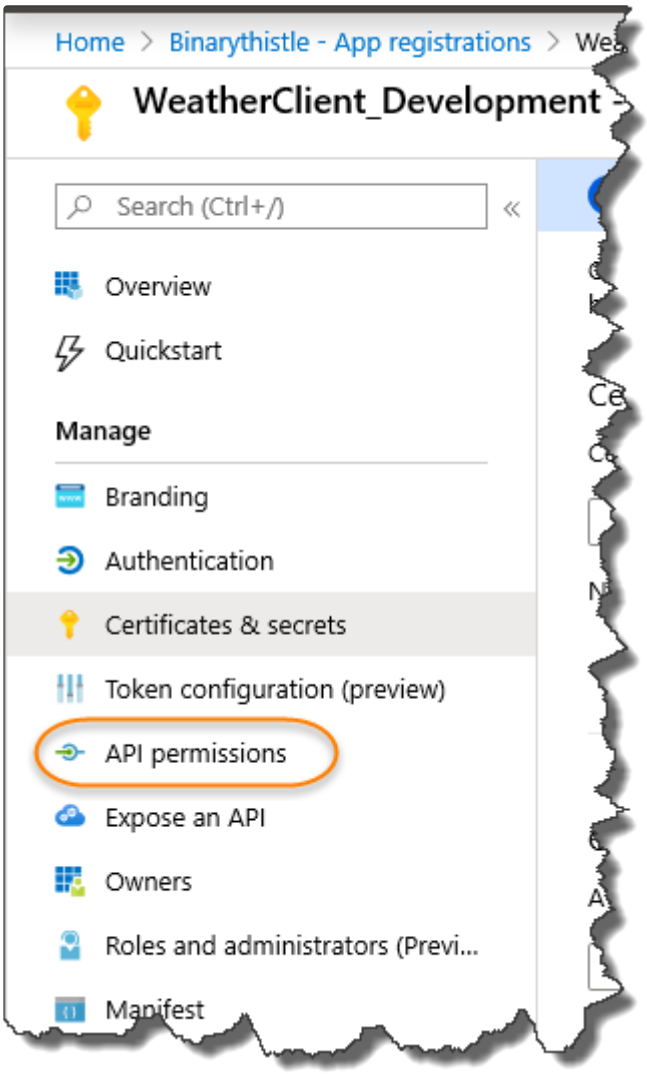
When you're happy click "add".

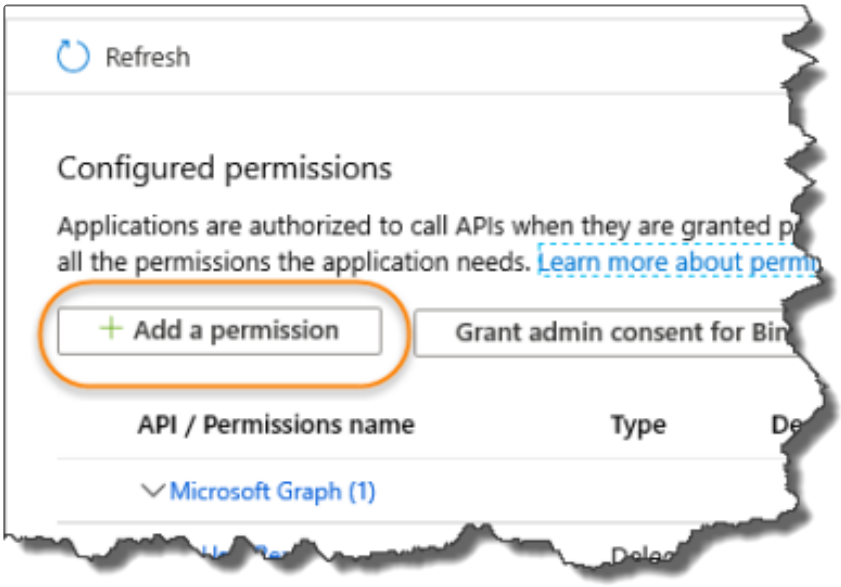
able to retrieve it unlike our other registration attributes.

Also note the ***client secret*** is akin in sensitivity to a UserId and Password, so it should be guarded closely. In the sections that follow I store this as plain-text in a *appsettings.json* file which I would not recommend outside of a tutorial / learning environment.

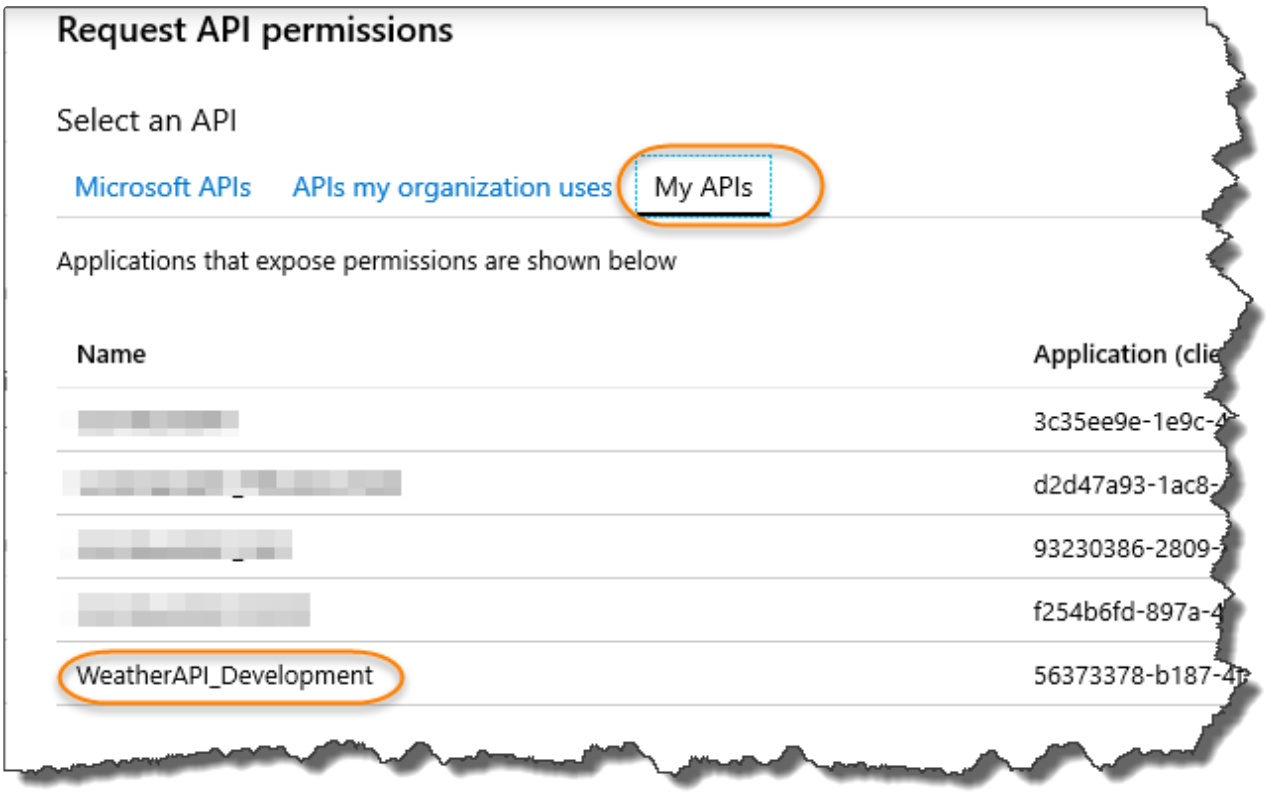
Step 8 – Configure API Permissions

Now click on "API Permissions", here we are going to, (drum role please), configure access to our command API:



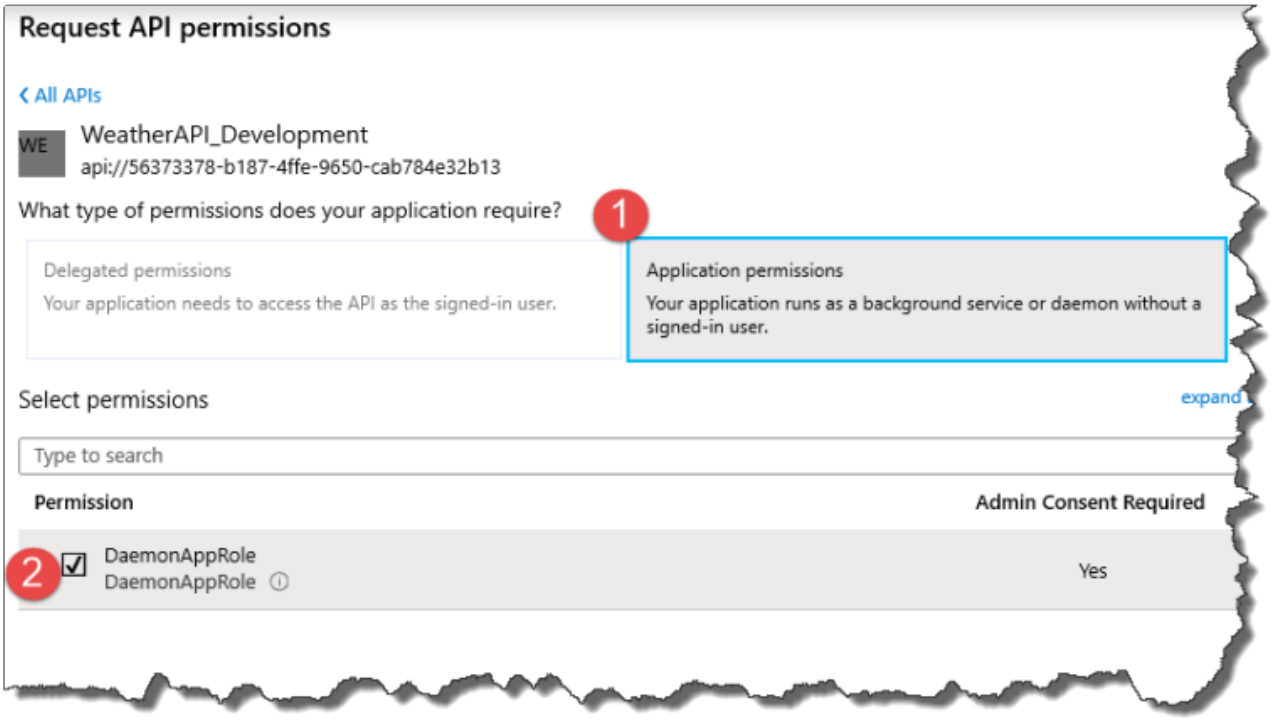


In the “Request API permissions” window that appears, select the “My APIs” tab:

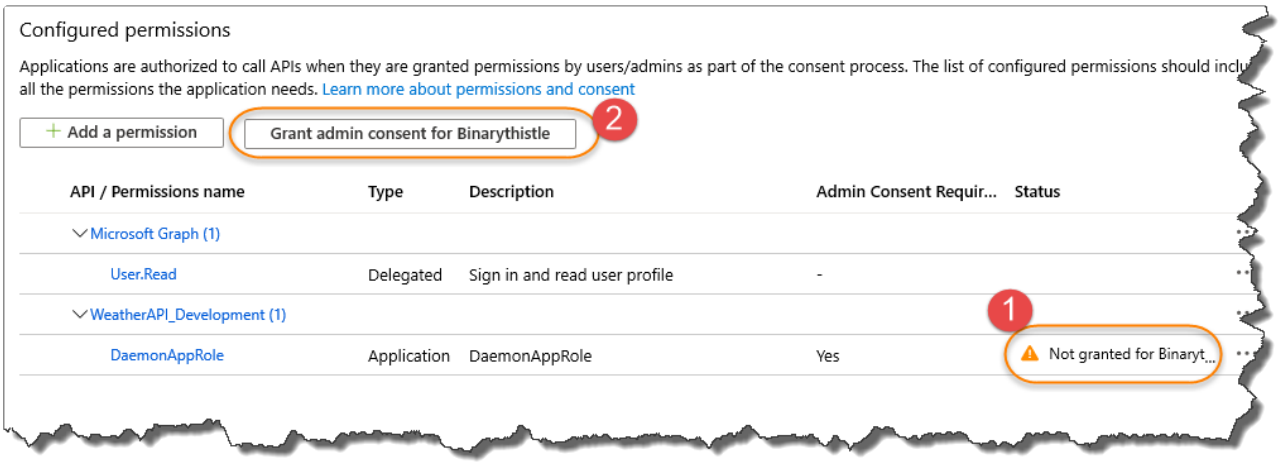


Select the API that you want your client to have permission to, (in my case WeatherAPI_Development), and on the resulting screen ensure that:

2. You “check” the DaemonAppRole Permission



When you’re happy, click “Add permission” and your permission will be added to the list:

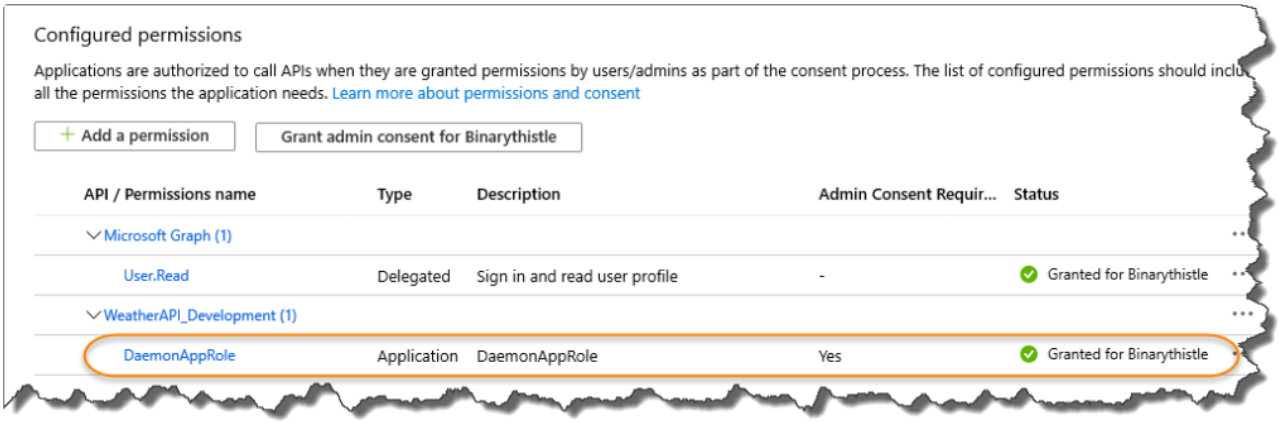


You’ll notice:

1. The permission has been “created” but not yet “granted

Clicking on the Grant Admin conset button may result in a confirmation pop up, answer in the affirmative if you do.

You'll be returned to the Configure permissions window, where after a short time, your newly created API Permission will have been granted access:

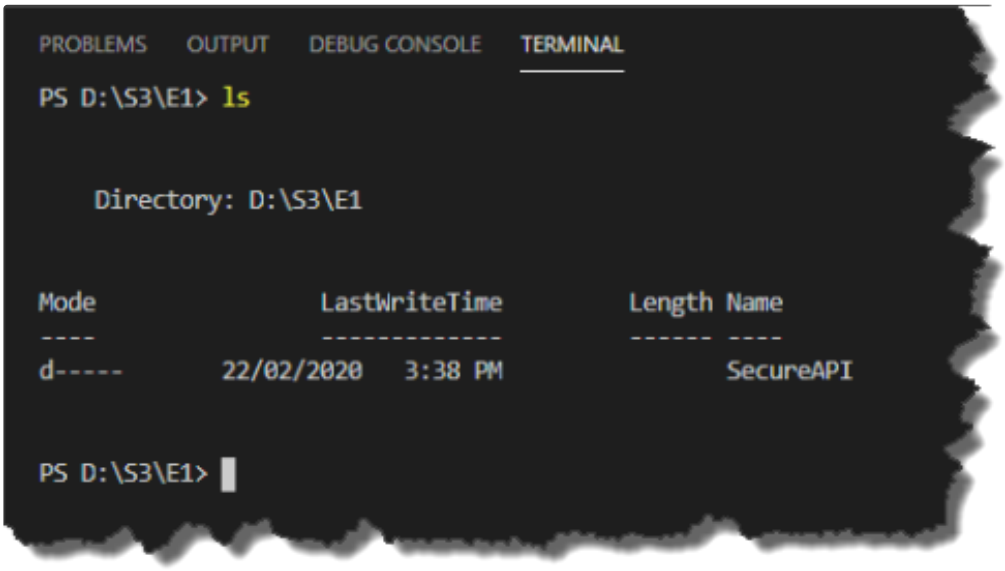


And with that the registration of our, (yet to be created), client app is complete. We create the client app in our next step.

Step 9 – Create our Client App

So now we're going to create the client app that's going to call our protected endpoint.

So at a command line ensure you're back in the "root" of your working directory, i.e. performing a directory listing you should see the project folder for our WeatherAPI as shown below:



Ensuring that you're in the right place, we're going to create a new console application by typing:

```
dotnet new console -n SecureAPIClient
```

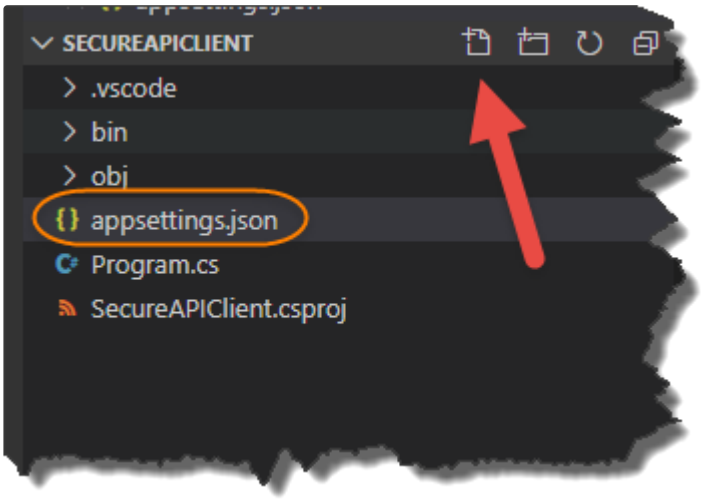
Once the project has been created open the project folder *SecureAPIClient* in your development environment, so if you're using VS Code you could type:

```
code -r SecureAPIClient
```

Client Configuration

As with our API, we need to supply some configuration elements to our Client app in order for it to use our AAD to authenticate through to our API. I'm going to use an *appsettings.json* file for this, (which we'll need to create), but again I'd call out that I would not use this approach for a production app as we'll be storing sensitive info in this file that you would not want to fall into the wrong hands.

Anyway, create an appsettings.json file in the root of your client project folder as shown below:



Note: you can use the built in "add file" functionality within VS Code to do this as indicated by the arrow above.

Into that file add the following JSON, making sure to populate the correct values for *your* client/daemon application registration, and in the case of the *ResourceId* & *BaseAddress*, your API application registration.

```
  "Instance": "https://login.microsoftonline.com/{0}",
  "TenantId": "[YOUR TENANT ID]",
  "ClientId": "[YOUR CLIENT ID]",
  "ClientSecret": "[YOUR CLIENT SECRET]",
  "BaseAddress": "https://localhost:5001/api/Commands/1",
  "ResourceId": "api://[YOUR API CLIENT ID]/.default"
}
```

So in my case my file looks like this:

A couple of points here:

- *BaseAddress*: This is just the local address of the command API. Note, I'm deliberately specifying the API Action Result that requires authorization.
- *ResourceId*: This is the *ResourceId* of our API App Registration

The other attributes are straightforward and can be retrieved from Azure, except the *ClientSecret* which you should have made a copy of when you created it.

Add Package References

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Binder
- Microsoft.Extensions.Configuration.Json
- Microsoft.Identity.Client

I prefer to do this by using the dotnet CLI as we've done previously so:

```
dotnet add package <package name>
```

So, for example issue the following command inside the *SecureAPIClient* app folder:

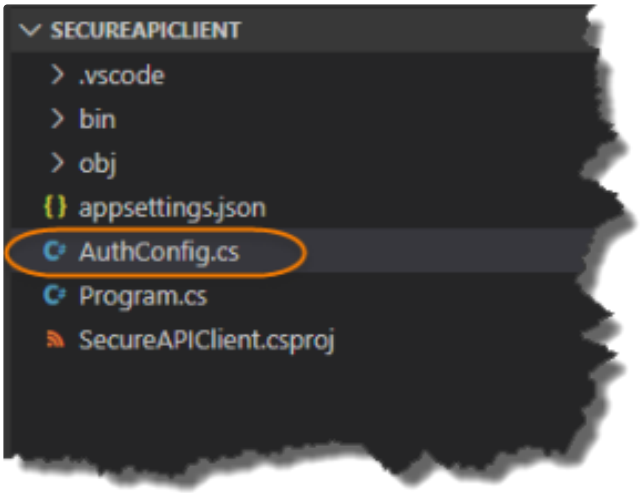
```
dotnet add package Microsoft.Extensions.Configuration
```

Repeat so you add all 4 packages, your project .csproj file should look like this when done:

```
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Configuration" Version="3.1.2" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="3.1.2" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="3.1.2" />
    <PackageReference Include="Microsoft.Identity.Client" Version="4.8.2" />
  </ItemGroup>

</Project>
```

Client Configuration Class

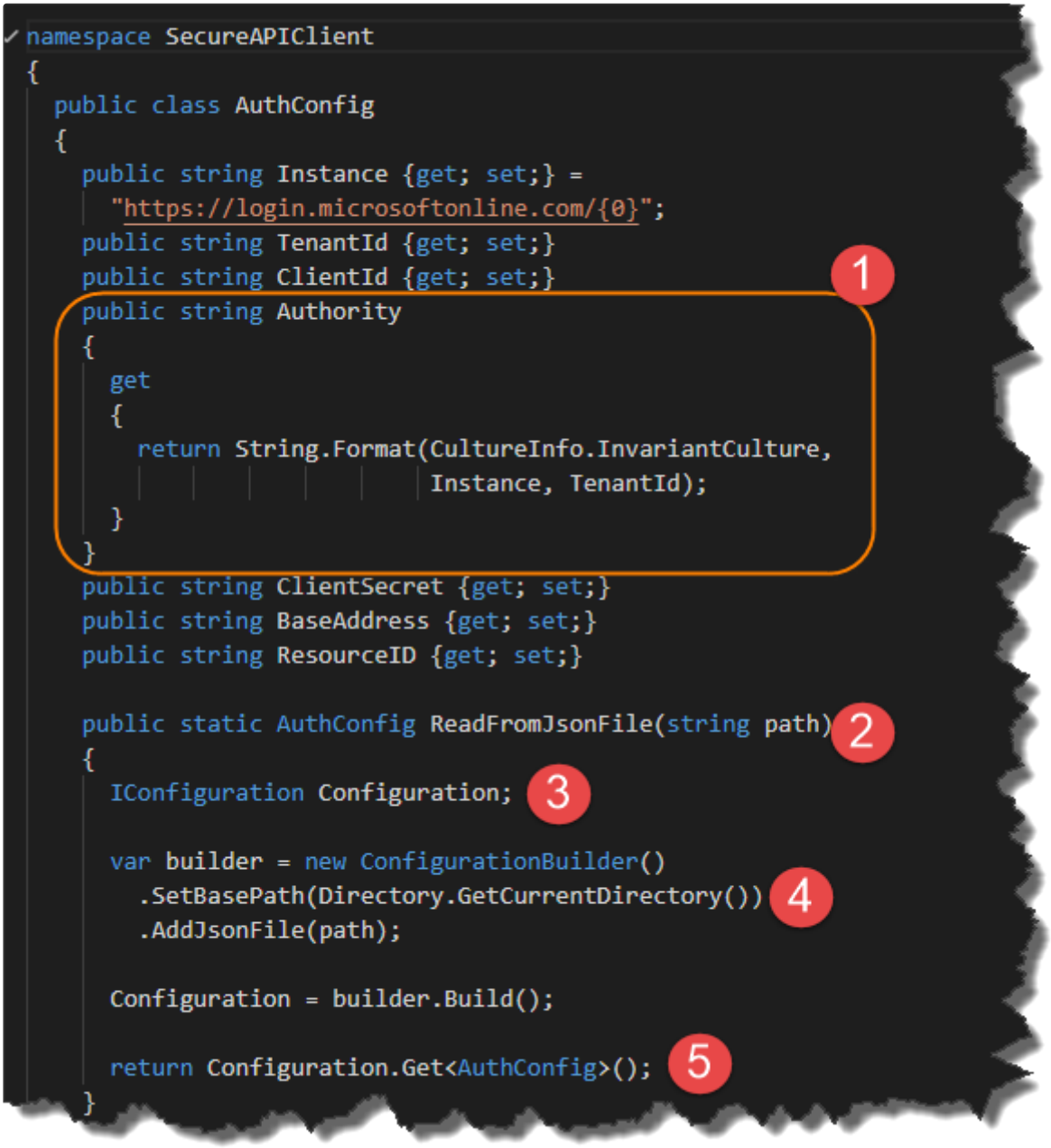
For ease of use we're going to create a custom class that will allow us to read in our appsettings.json file and then access those config elements as class attributes. In the client project create a new class file in the root of the project and call it AuthConfig.cs as shown below:



The enter the following code:

```
using System.IO;
using System.Globalization;
using Microsoft.Extensions.Configuration;

namespace SecureAPIClient
{
  public class AuthConfig
  {
    public string Instance {get; set;} =
      "https://login.microsoftonline.com/{0}";
    public string TenantId {get; set;}
    public string ClientId {get; set;}
    public string Authority
    {
      get
      {
        return String.Format(CultureInfo.InvariantCulture,
                             Instance, TenantId);
      }
    }
    public string ClientSecret {get; set;}
    public string BaseAddress {get; set;}
    public string ResourceID {get; set;}

    public static AuthConfig ReadFromJsonFile(string path)
    {
      IConfiguration Configuration;

      var builder = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile(path);

      Configuration = builder.Build();

      return Configuration.Get<AuthConfig>();
    }
```

When complete your AuthConfig class should look like this:



Noteable code listed below:

1. We combine the Instance and our AAD Tenant to create something called the “Authority”, this is required when we come to attempting to connect our

2. Our class has 1 static method that allows us to specify the name of our json config file

3. We create an instance of the .NET Core Configuration subsystem

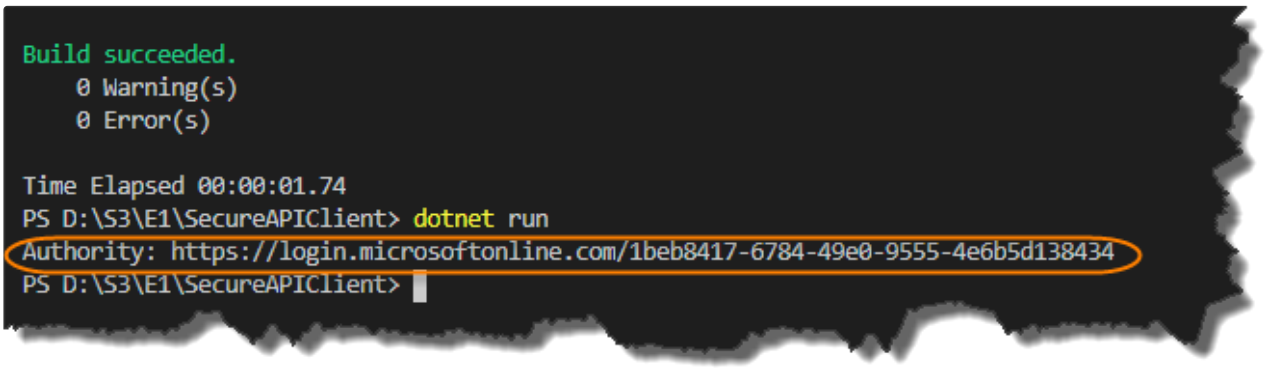
4. Using ConfigurationBuilder we read the contents of our json config file

5. We pass back our read-in config bound to our AuthConfig class

To quickly test that this all works, perform a build, and assuming we have no errors, move over to our *Program* class and edit the Main method so it looks like this:

```
static void Main(string[] args)
{
  AuthConfig config = AuthConfig.ReadFromJsonFile("appsettings.json

  Console.WriteLine($"Authority: {config.Authority}");
}
```

Build your code again then run it, assuming all is well you should get output similar to this:



As mentioned previously the first thing our client will have to do is obtain a JWT token that it will then attach to all subsequent requests in order to get access to the resources it needs, so let's focus in on that.

Still in our *Program* class we're going to create a new static asynchronous method called *RunAsync*, the code for our reworked *Program* class is shown below, (noting new or changed code is bold & highlighted):

```
            AuthenticationResult result = null;
            try
            {
              result = await app.AcquireTokenForClient(ResourceIds).Execu
              Console.ForegroundColor = ConsoleColor.Green;
              Console.WriteLine("Token acquired \n");
              Console.WriteLine(result.AccessToken);
              Console.ResetColor();
            }
            catch (MsalClientException ex)
            {
              Console.ForegroundColor = ConsoleColor.Red;
              Console.WriteLine(ex.Message);
              Console.ResetColor();
            }
          }
        }
      }
```

I've tagged the points of interest below:

```
}

private static async Task RunAsync()
{
    AuthConfig config = AuthConfig.ReadFromJsonFile("appsettings.json");

    IConfidentialClientApplication app;

    app = ConfidentialClientApplicationBuilder.Create(config.ClientId)
        .WithClientSecret(config.ClientSecret)
        .WithAuthority(new Uri(config.Authority))
        .Build();

    string[] ResourceIds = new string[] { config.ResourceID };

    AuthenticationResult result = null;
    try
    {
        result = await app.AcquireTokenForClient(ResourceIds).ExecuteAsync();
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine("Token acquired \n");
        Console.WriteLine(result.AccessToken);
        Console.ResetColor();
    }
    catch (MsalClientException ex)
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(ex.Message);
        Console.ResetColor();
    }
}
```

1. Our RunAsync method is asynchronous and returns a result we're interested in, so we chain the GetAwaiter and GetResult methods to ensure the console app does not quit before a result is processed and returned.

2. ConfidentialClientApplication is a specific class type for our use case, we use this in conjunction with the ConfidentialClientApplicationBuilder to construct a "client" with our config attributes.

3. We set up our app with the values derived from our AuthConfig class

5. The AuthenticationResult contains, (drum roll), the result of a token acquisition

6. Finally we make an asynchronous AquireTokenForClient call to, (hopefully!), return a JWT Bearer token from AAD using our authentication config

Save the file, build your code and assuming all's well, run it too, should see:

We move onto the 2nd and final part of our RunAsync method, and that is to call our protected API endpoint with the token we just obtained in the previous step, so directly after the catch statement in our RunAsync method, add the following code, (take note of the 3 additional using statements too):

```
if (!String.IsNullOrEmpty(result.AccessToken))
{
  var httpClient = new HttpClient();
  var defaultRequestHeaders = httpClient.DefaultRequestHeaders;

  if(defaultRequestHeaders.Accept ==null ||
     !defaultRequestHeaders.Accept.Any(m => m.MediaType == "applica
  {
    httpClient.DefaultRequestHeaders.Accept.Add(new
      MediaTypeWithQualityHeaderValue("application/json"));
  }
  defaultRequestHeaders.Authorization =
    new AuthenticationHeaderValue("bearer", result.AccessToken);

  HttpResponseMessage response = await httpClient.GetAsync(config.Ba
  if (response.IsSuccessStatusCode)
  {
    Console.ForegroundColor = ConsoleColor.Green;
    string json = await response.Content.ReadAsStringAsync();
    Console.WriteLine(json);
  }
  else
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine($"Failed to call the Web Api: {response.Statu
    string content = await response.Content.ReadAsStringAsync();
    Console.WriteLine($"Content: {content}");
  }
  Console.ResetColor();
}
```

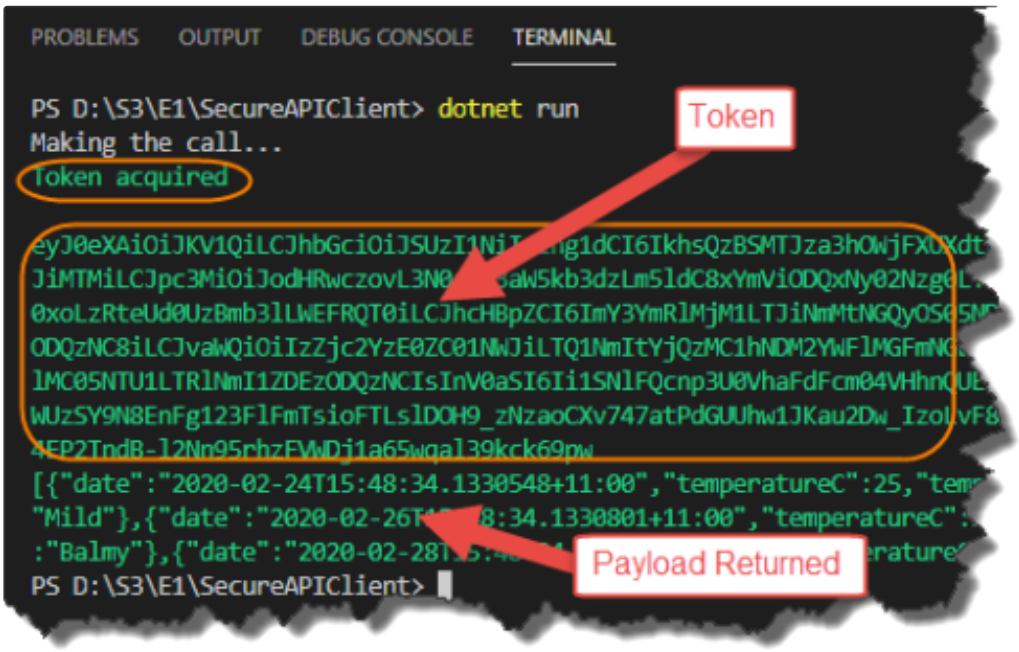
I've highlighted some interesting code sections below:

```
    Console.ResetColor();
}

if (!string.IsNullOrEmpty(result.AccessToken))
{
    var httpClient = new HttpClient();                                    1
    var defaultRequestHeaders = httpClient.DefaultRequestHeaders;

    if (defaultRequestHeaders.Accept == null ||                                2
       !defaultRequestHeaders.Accept.Any(m => m.MediaType == "application/json"))
    {
        httpClient.DefaultRequestHeaders.Accept.Add(new
          MediaTypeWithQualityHeaderValue("application/json"));
    }
    defaultRequestHeaders.Authorization =
      new AuthenticationHeaderValue("bearer", result.AccessToken);        3

    HttpResponseMessage response = await httpClient.GetAsync(config.BaseAddress);  4
    if (response.IsSuccessStatusCode)                                      5
    {
        Console.ForegroundColor = ConsoleColor.Green;
        string json = await response.Content.ReadAsStringAsync();
        Console.WriteLine(json);
    }
    else
    {
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"Failed to call the Web Api: {response.StatusCode}");
        string content = await response.Content.ReadAsStringAsync();
        Console.WriteLine($"Content: {content}");
    }
    Console.ResetColor();
}
```

1. We use a HttpClient object as the primary vehicle to make the request
2. We ensure that we set the media type in our request headers appropriately
3. We set out authorisation header to “bearer” as well as attaching our token received in the last step
4. Make an asynchronous request to our protected API address
5. Check for success and display

Save the code and build it, there should be no errors.

We reach that last step, simply to run our code, so issue a dotnet run, and you should see:



And we're done!

We've successfully:

- Configured AAD to be our Identify and Access Management Service
- Secure our API Endpount
- Created a Secure Client that will be issued a token that it uses to access our secured API

Les Jackson

Les enjoys understanding how things work, proving concepts then telling people about it! He lives and works in Melbourne, Australia but is originally from Glasgow, Scotland. He's just obtained an MCSD accreditation after almost a year, so now has more time for writing this blog, making YouTube videos, as well as enjoying the fantastic beer, wine, coffee and food Melbourne has to offer.

RELATED ARTICLES

SECURITY

Custom Local Domain using HTTPS, Kestrel & ASP.NET Core

PREVIOUS POST
Azure Functions in C#

NEXT POST

Dotnet Playbook

Pragmatic hints, tips, step by step tutorials on how to get the most out of the .Net Framework.

CATEGORIES

META

- Log in
- Entries feed
- Comments feed
- WordPress.org