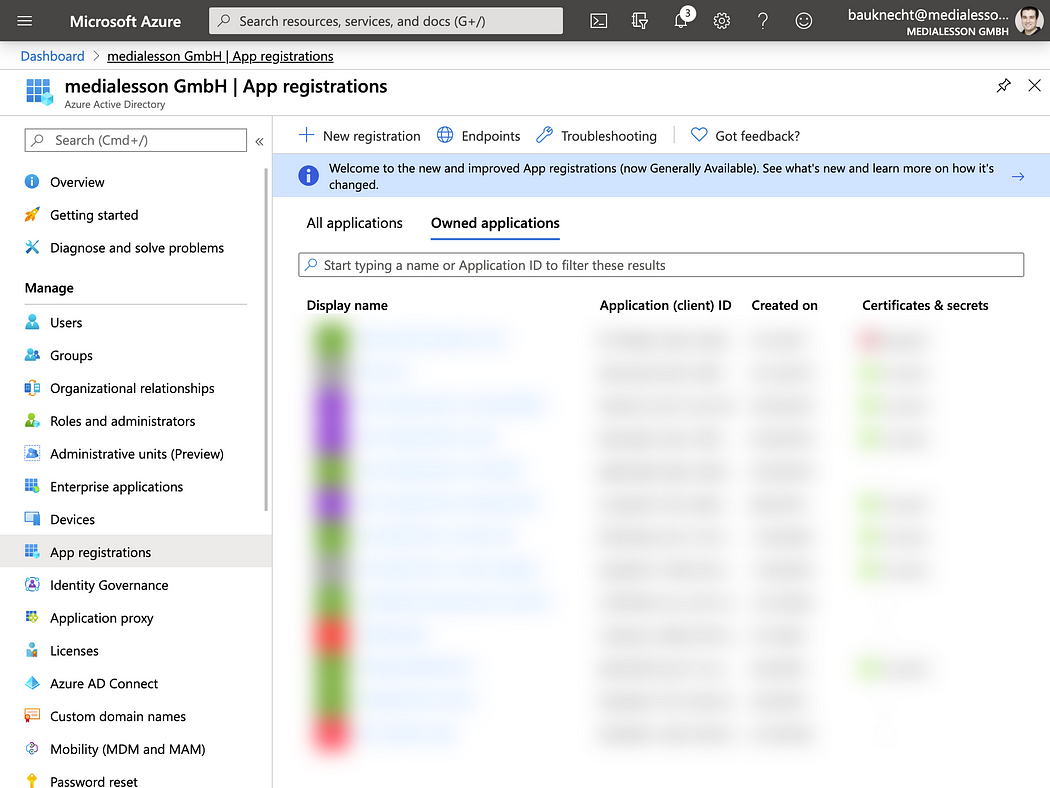
**Protecting Azure Function apps with Azure AD Authentication & Authorization**

How to enable Authentication/Authorization in Azure Function apps using configuration and get the signed in user in code.

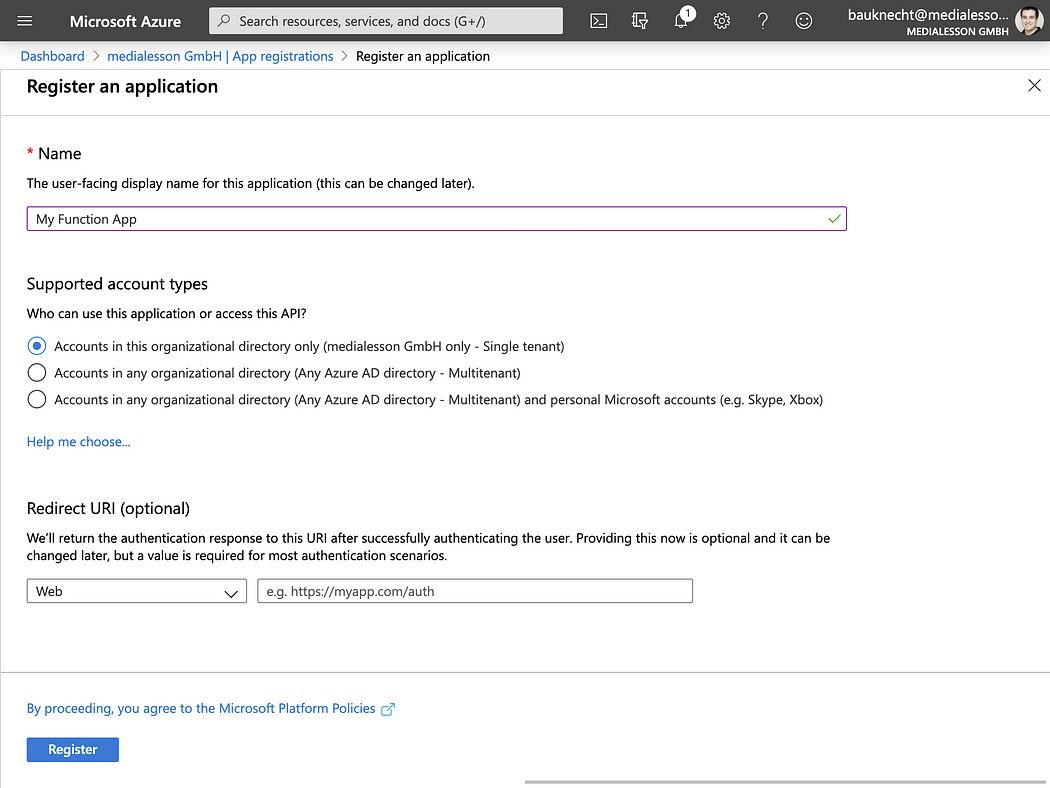
**Configure a new app registration in Azure AD**

To use Azure AD as an authentication provider in Angular we need to register a new app in the Azure portal: <https://portal.azure.com/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/RegisteredApps>, click on new registration:



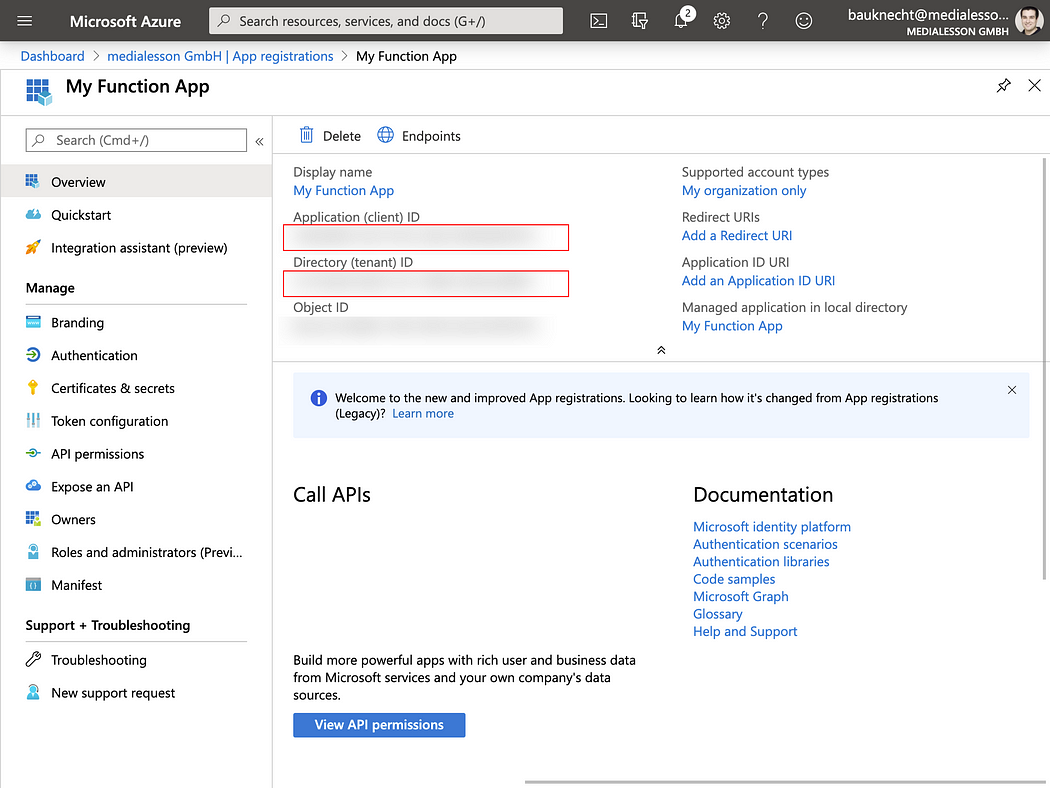
Azure AD App Registrations

Enter as display name and save it.



Register an application

On the overview page make sure to copy the Application (client) ID and your Directory (tenant) ID:

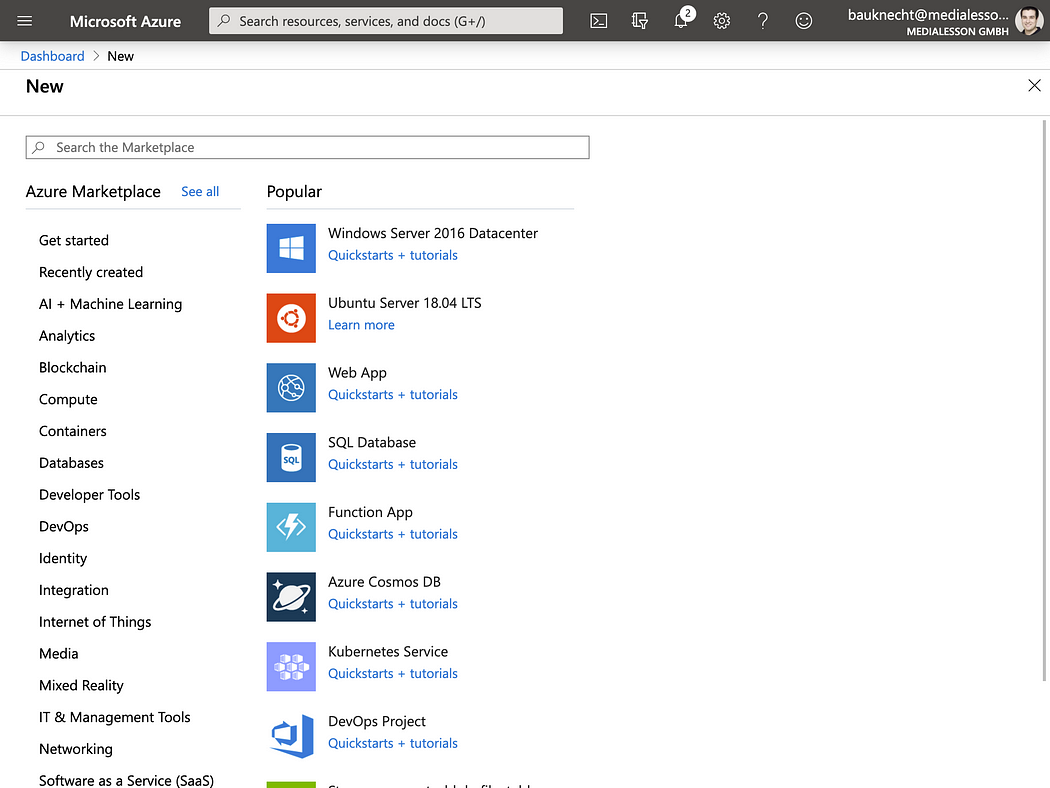


Client Id & Tenant Id

**Create a new Function app**

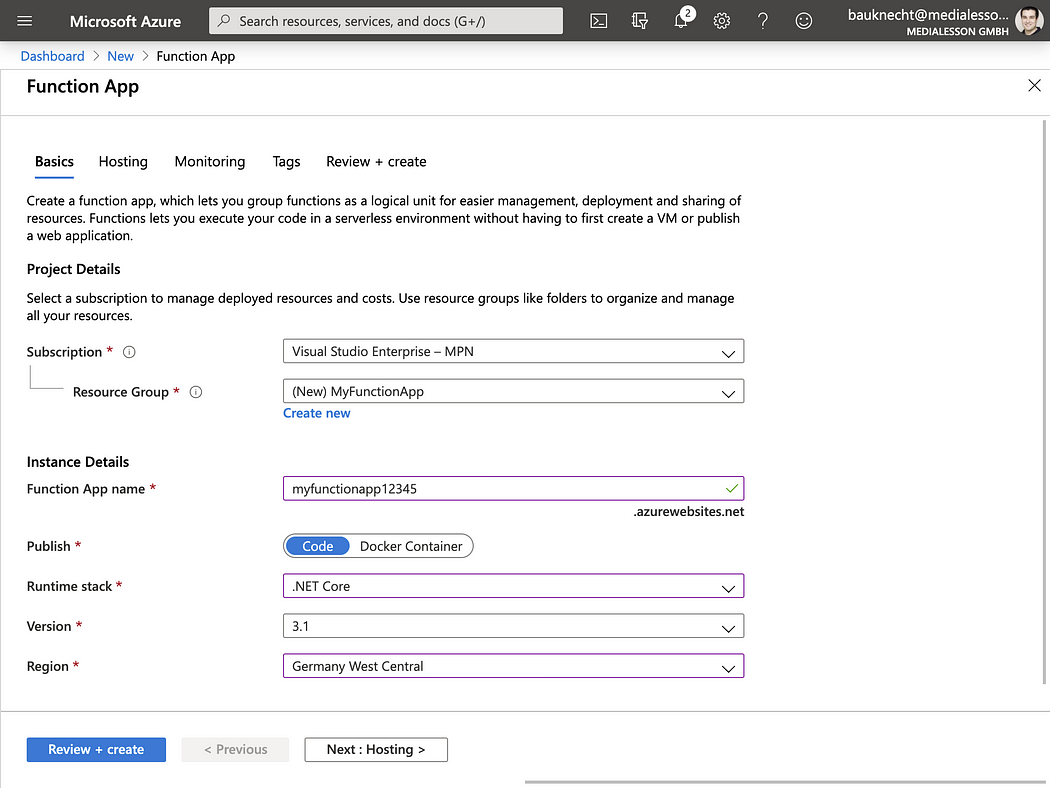
**Create Function app in Azure Portal**

Let’s start by creating a new Function app in den Azure Portal, <https://portal.azure.com/#create/hub>



Select Function App

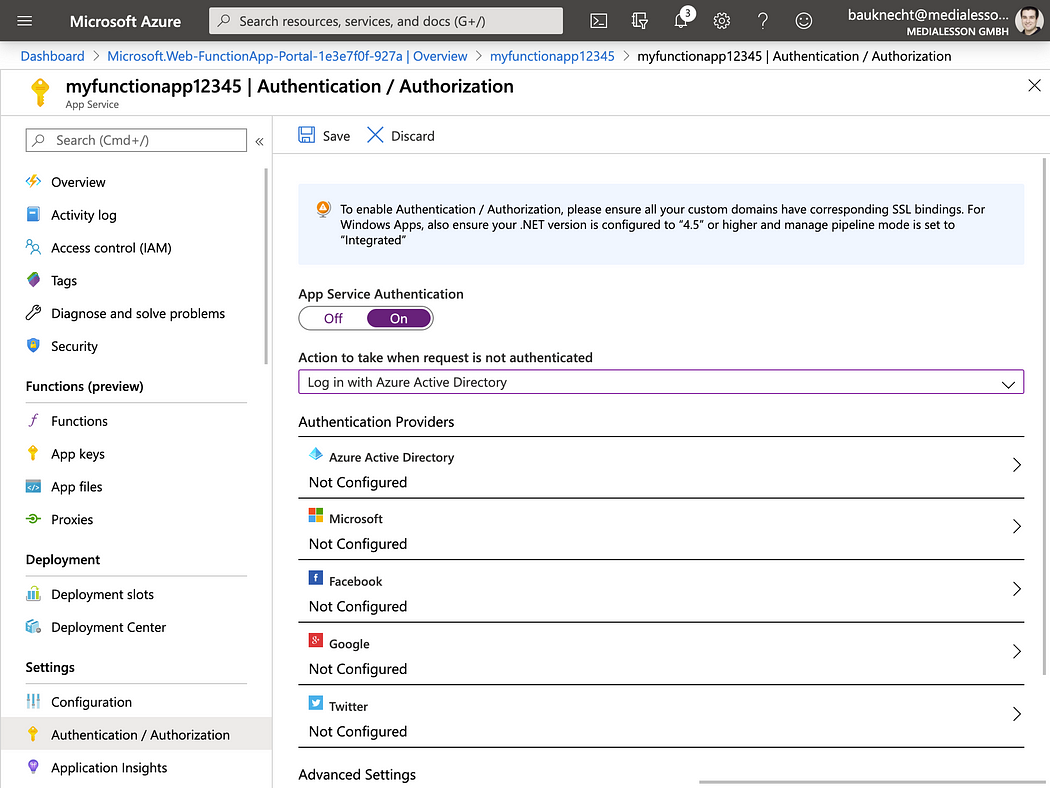
Create a new resource group, pick a name, select .NET Core 3.1 as runtime stack and create the app.



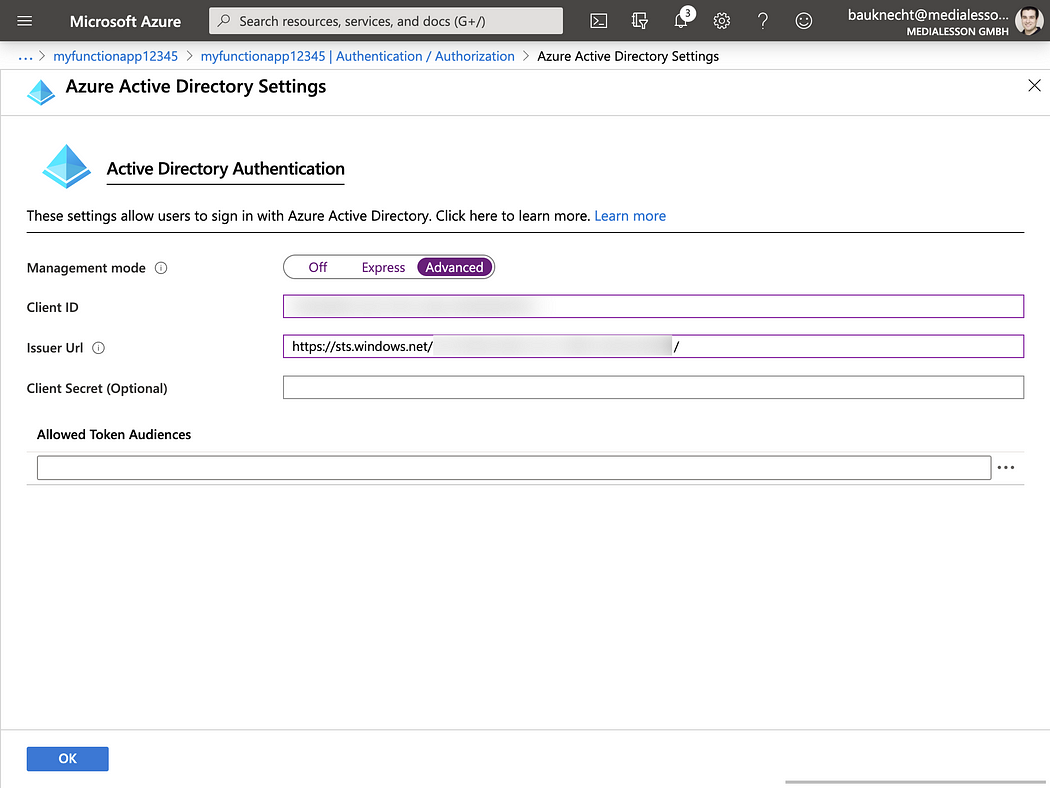
Once the app is created got to Authentication/Authorization and set App Service Authentication to On. Also select Log in with Azure Active Directory as Action to take when request is not authenticated.

*At this point a bit of context how this authentication actually works: The Authentication middleware in Azure Functions validates incoming access tokens and checks if they are meant for the provided audience. The audience is represented by the configured Azure AD app registration that we will provide in the next step. Also this middleware extracts all claims included in the access tokens and makes them accessible to the Function’s code via input binding/method parameters.*

Click on Azure Active Directory to configure the authentication provider:

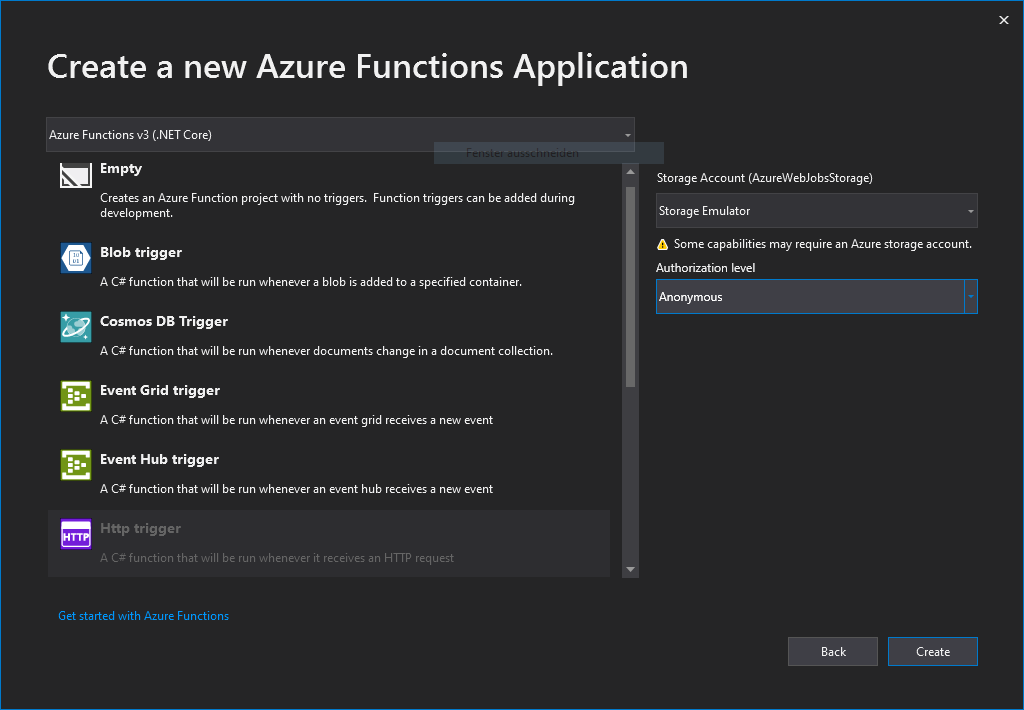


Next up paste the client id of the Azure AD app registration and also add the issuer url. The issuer url is in the form of https://sts.windows.net/YOUR\_TENANT\_ID/



**Create Function app in Visual Studio**

Now that we have the app setup in Azure we also need to create some code. Therefore we need create a new Function app using C# in Visual Studio:

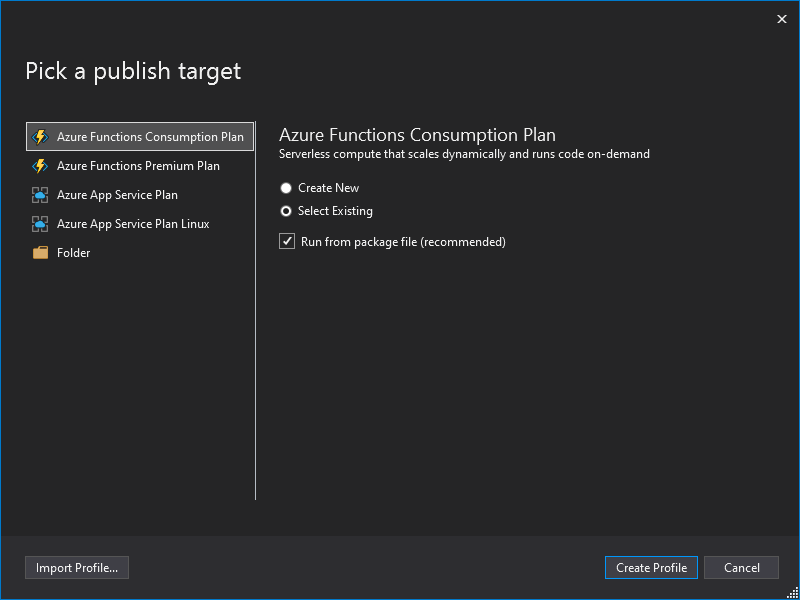


Select Http trigger so we have a sample function to test authentication with.

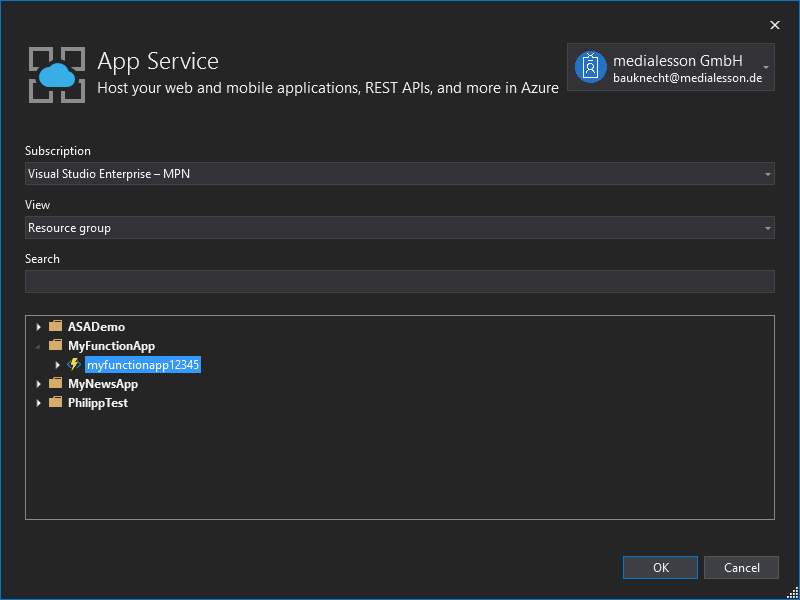
**Working with Claims**

As mentioned before the authentication middleware will extract the claims from the incoming authentication token. This allows us e.g. to get the username and other relevant information about the user. To do this we need to add a ClaimsPrincipal method parameter to our function. Also let’s just return the username as http response, so we can test if authentication and claims work:

So with this simple test function, let’s deploy the app to Azure so we can test it. Right click the project and select publish and pick Select Existing:



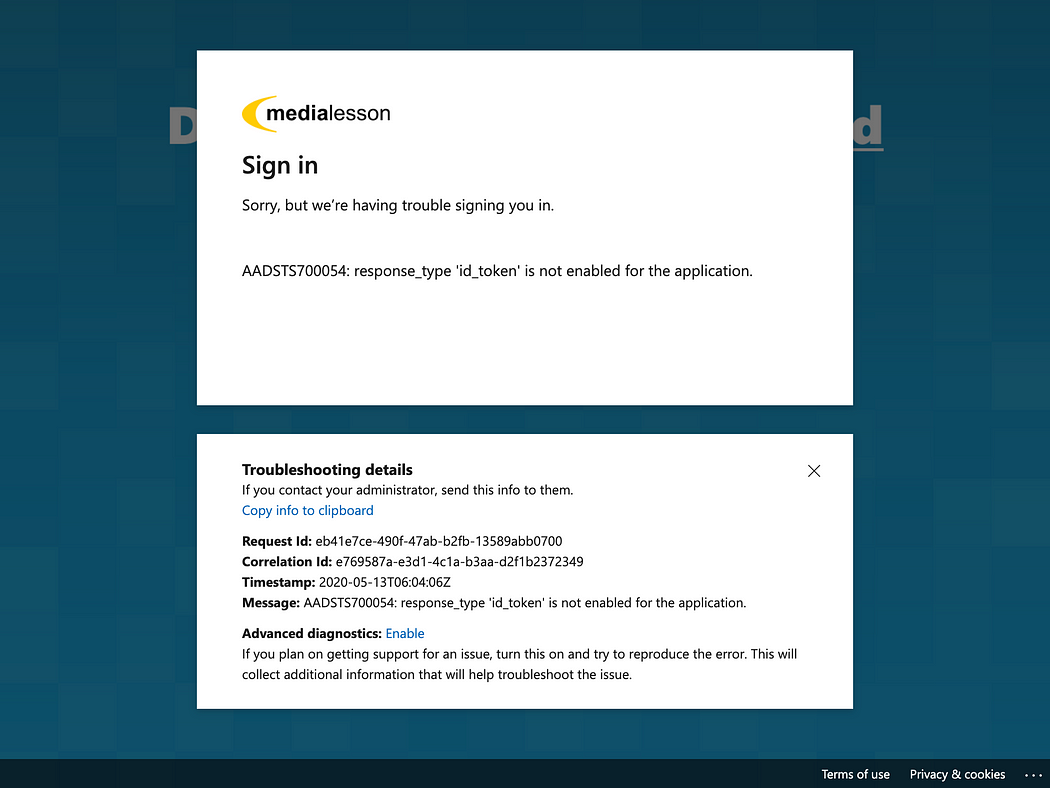
Login to your Azure account and select the Azure Function app we created before:



*Note: I have yet to find a way to test authentication locally. Happy for any ideas…*

**Test with Postman**

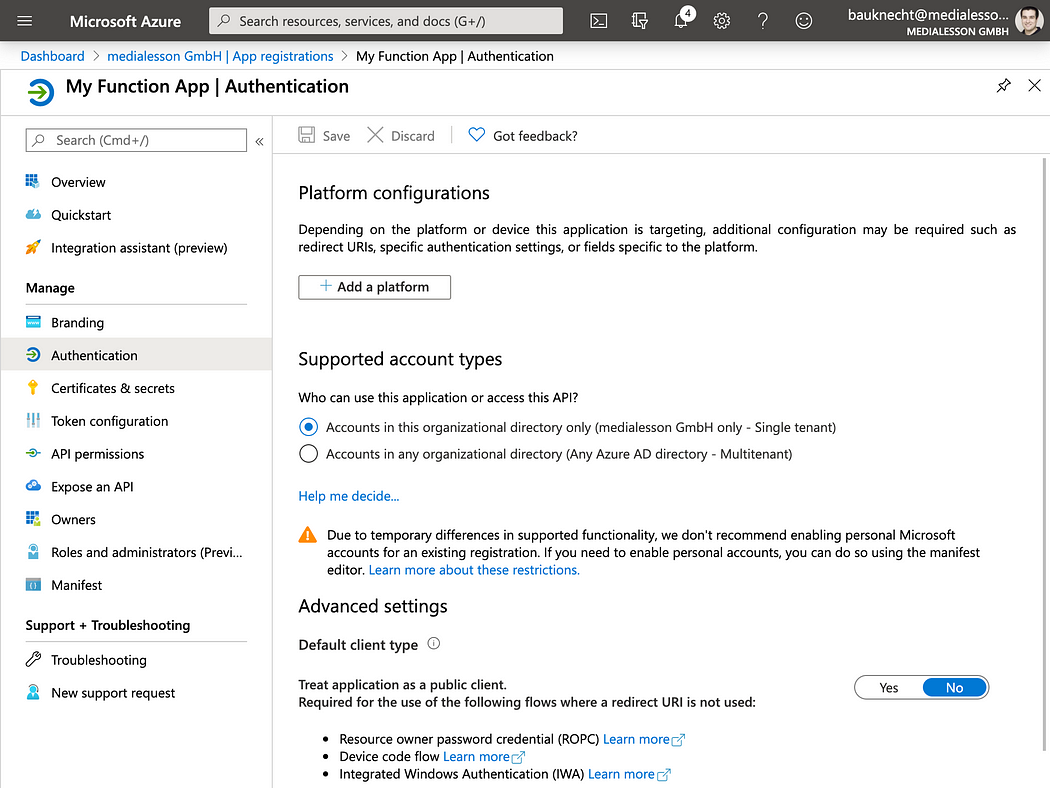
With authentication setup we now want to test this. Let’s call the function’s url in the browser to test it:



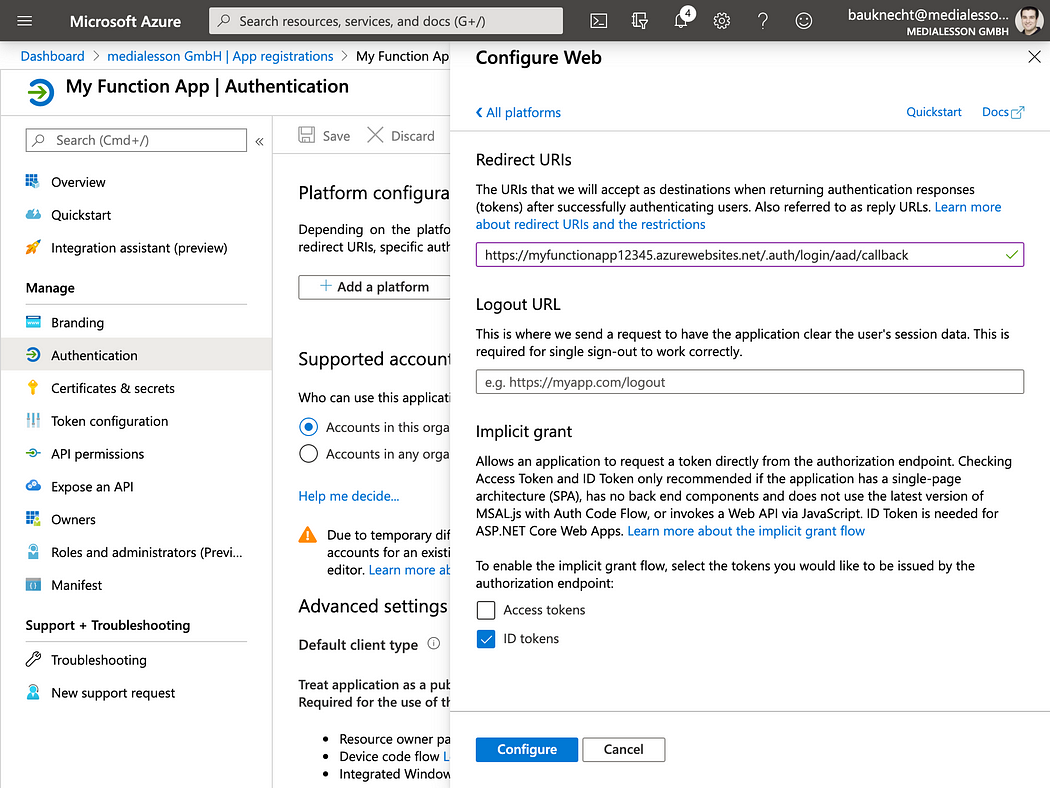
So we are being redirected to the login, but after successfully signing in, we get this nice little error. Don’t worry, it actually makes sense.

In real world scenarios our API will be called by some client, e.g. a web app. So the token is generated by a different app (e.g. an Angular app) and also by a different app registration. Since we don’t have a web app yet to create a token we will need to modify our app registration in Azure AD to create at least an ID token to test the endpoint temporarily. Please don’t forget to undo the following changes, once you move to production.

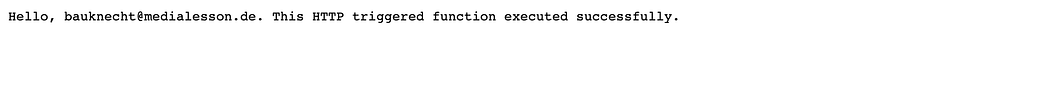
In the app registration in Azure AD we need to configure Authentication and add a platform:



Select web since we want to login in the browser. The Redirect URI is important to match with what the Function app will use. The correct setup is [https://YOUR\_APP.azurewebsites.net/.auth/login/aad/callcack.](https://your_app.azurewebsites.net/.auth/login/aad/callcack.) Make sure to also select ID token:



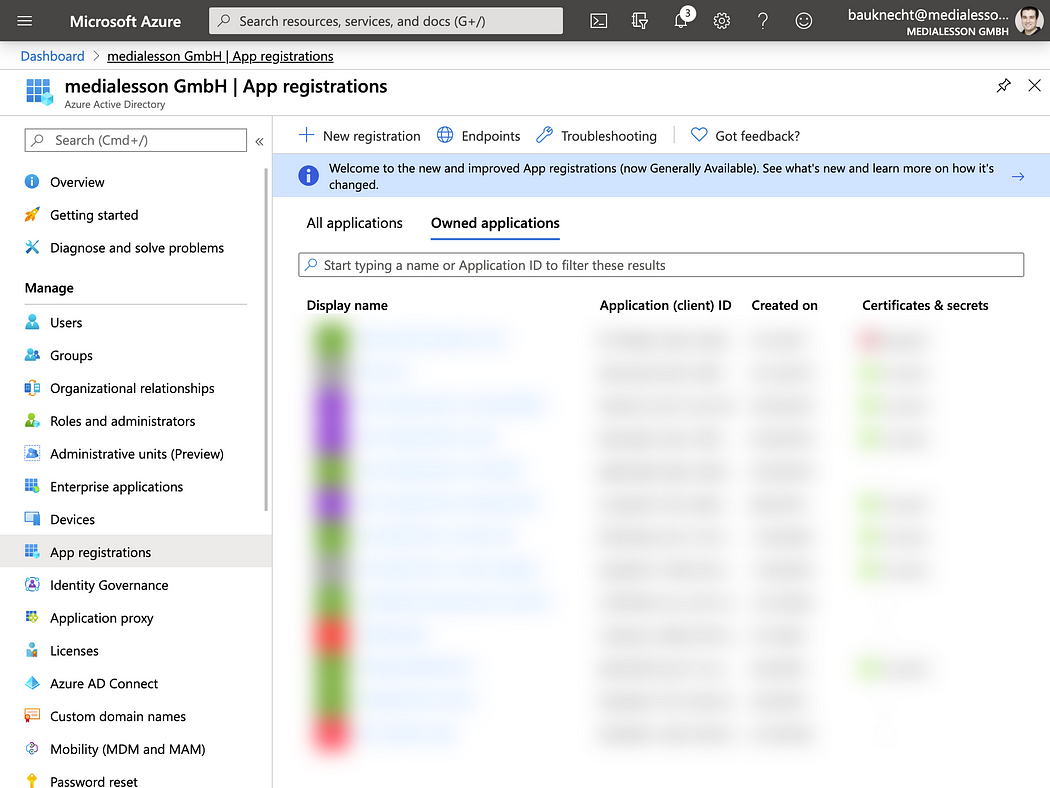
Let’s try again with the function url. This time we should be able to login and get our function’s response with the username:



**Summary**

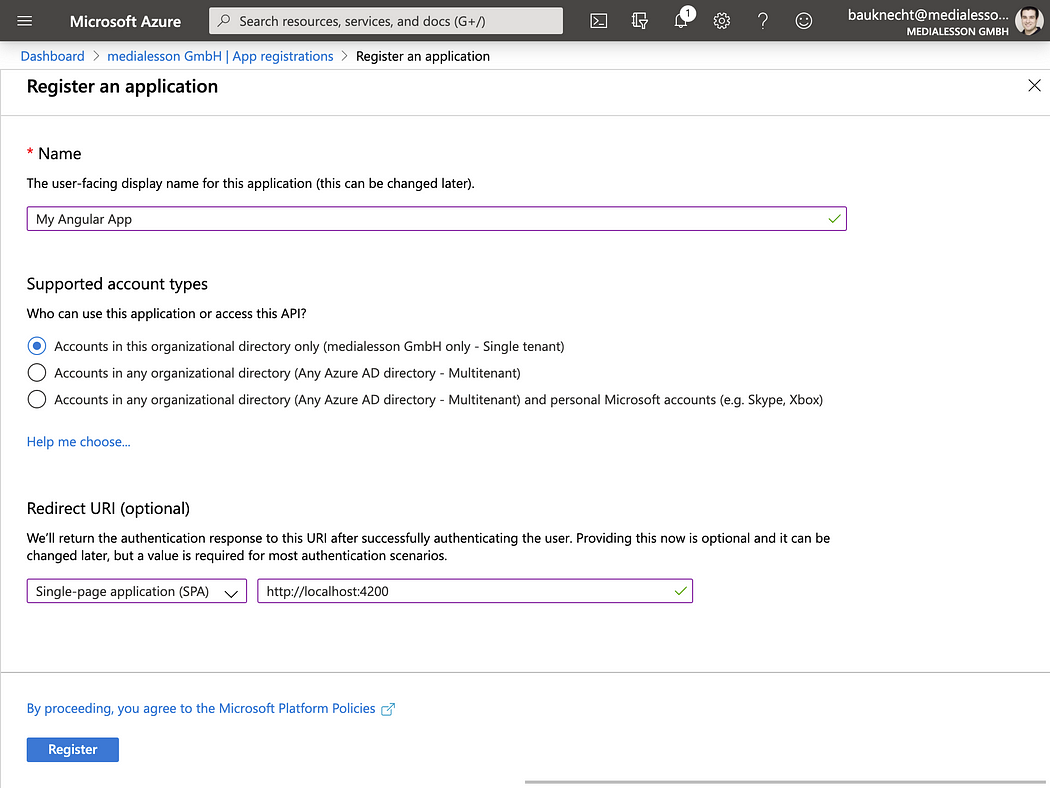
**Configure a new app registration in Azure AD**

To use Azure AD as an authentication provider in Angular we need to register a new app in the Azure portal: <https://portal.azure.com/#blade/Microsoft_AAD_IAM/ActiveDirectoryMenuBlade/RegisteredApps>, click on new registration:



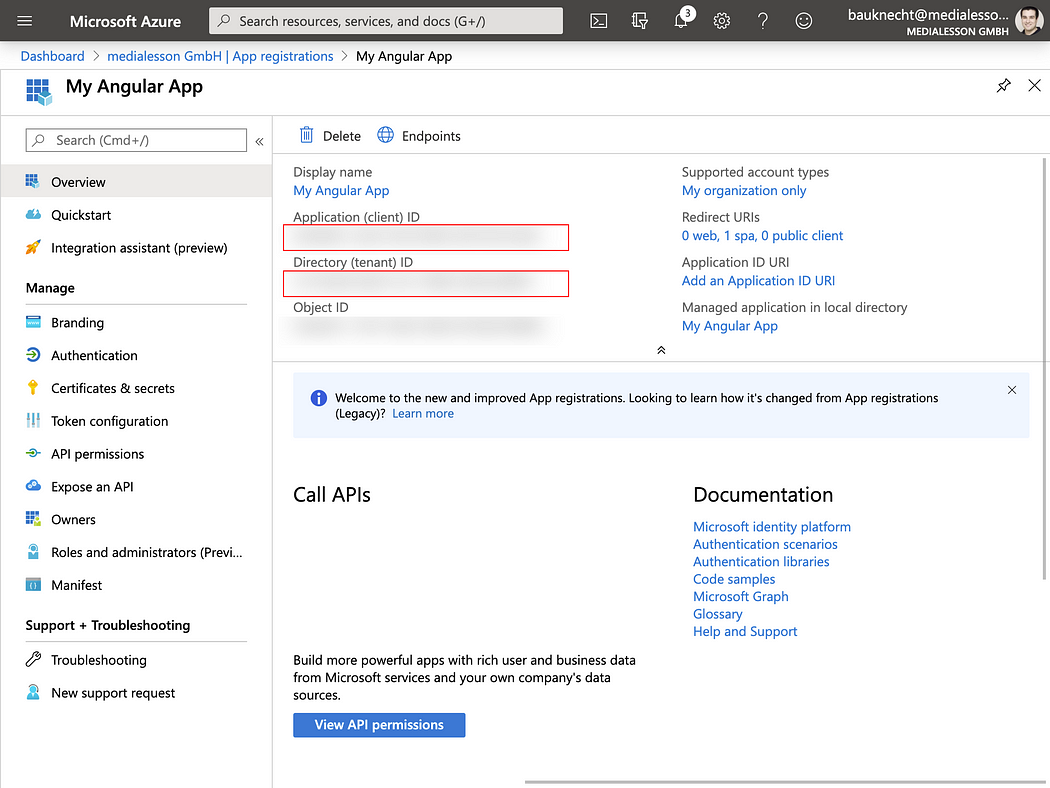
Azure AD App Registrations

Enter a display name, select Single-page application (SPA) and enter [http://localhost:4200](http://localhost:4200/) under Redirect URI. This is a local testing URI of our angular app. Save the registration.  
*Note: later we will need to also add the published URI of our app.*



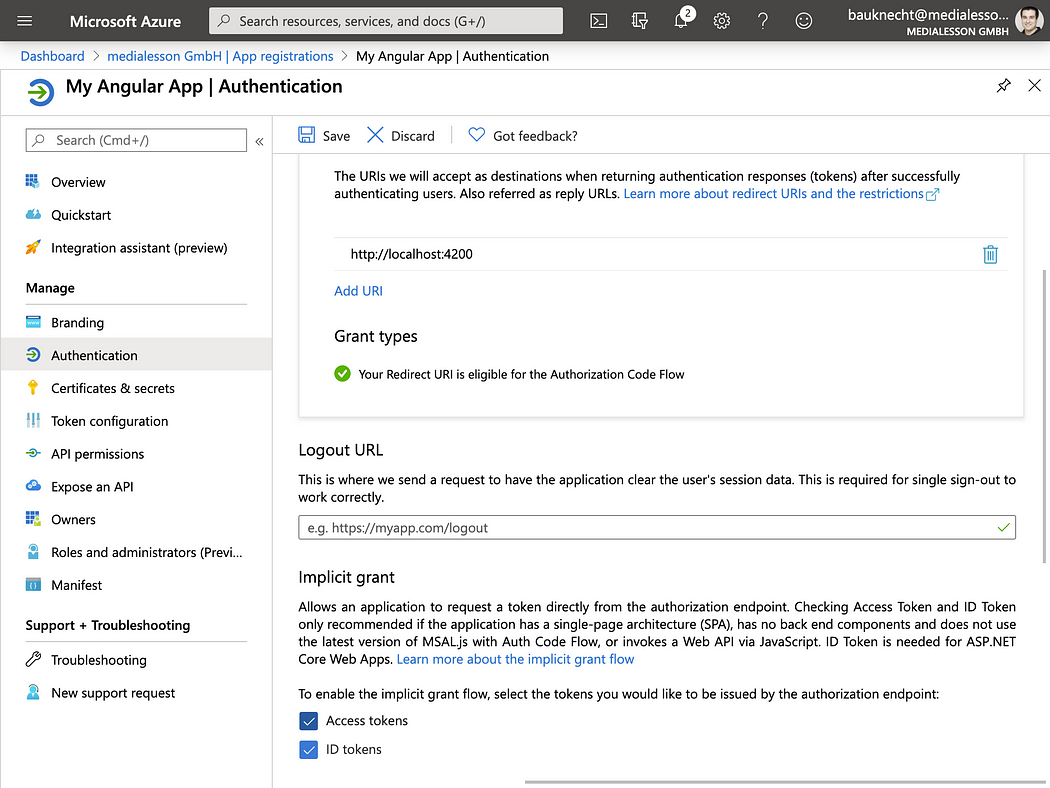
Register an application

On the overview page make sure to copy the Application (client) ID and your Directory (tenant) ID:



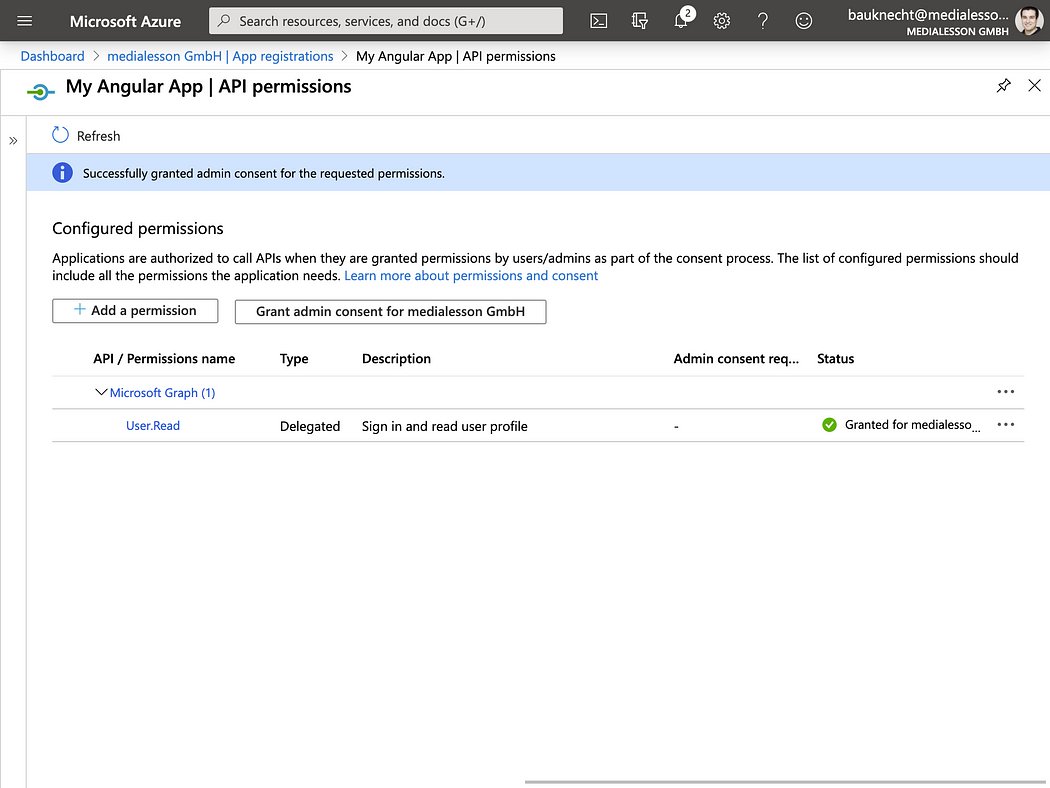
Client Id & Tenant Id

Select Authentication on the left panel and check Access tokens and ID tokens. This is needed so we can get information about the logged in user (ID token) as well as request access tokens to authenticate to our backend. Make sure to also save these changes.



Enable access and ID tokens

Finally grant admin constent to the app to sign in and read user profile under API permissions:



Now we have all the information we need to setup our Angular app.

**Setup a new Angular app to use authentication**

So let’s get started and create a new Angular app using the Angular CLI with routing enabled:

Next up we need MSAL and @Azure/msal-angular libraries installed intro our project:

Open the project in Visual Studio Code to start developing:

Since I like to keep my configuration central and some values like the redirect url will change in production let’s put all our authentication configuration in the environment.ts:

To configure MSAL authentication we need to create a factory method and let it return a new configuration object including our configuration parameters in our app.module.ts:

Then we can provide the MSAL\_CONFIG with this factory method:

We also need to create a factory method for the angular specific configuration of MSAL:

This needs also to be provided this time as MSAL\_ANGULAR\_CONFIG and we also need to provide MsalService (which will be our API to interact with MSAL) and the MsalModule which includes the MsalGuard to protect routes:

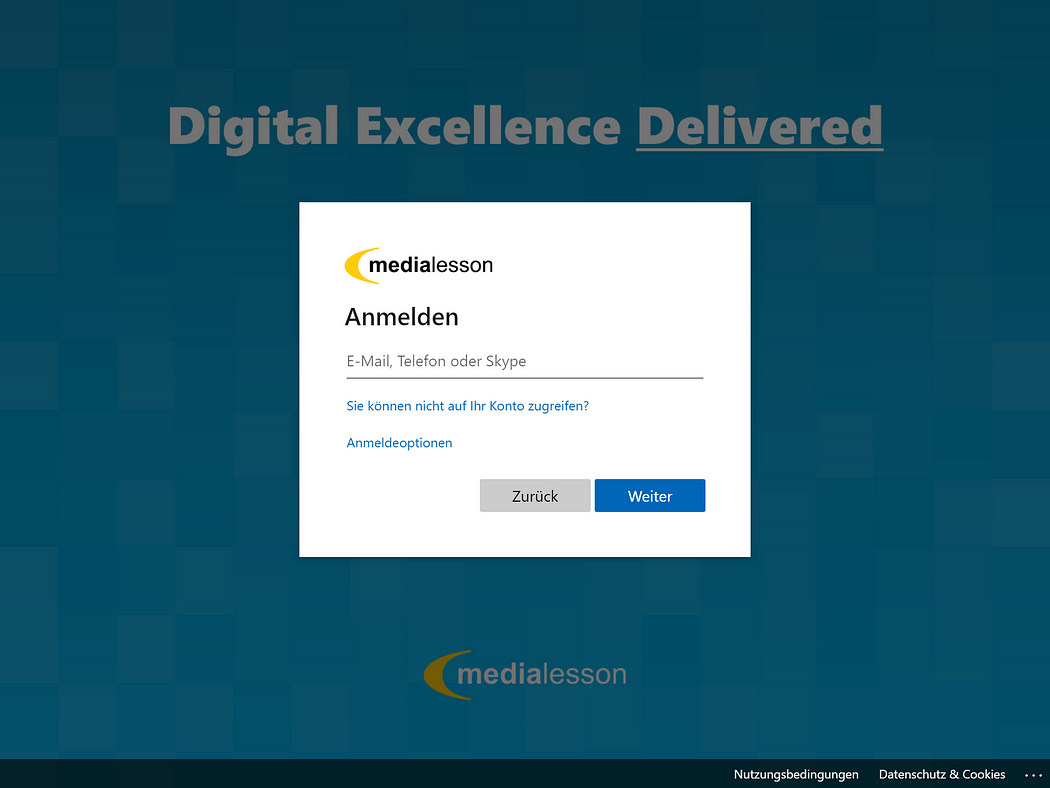
**Protect a route and content using Msal**

To test the authentication we need to create an area of our app that needs to be login protected. So let’s generate a new component:

The newly generated ProfileComponent will be accessible via a route. So we need to setup a router-outlet in our AppComponent:

And also create a route in app-routing.module.ts. This route needs to be protected and shall only be activated when we have successfully authenticated as a user in Azure AD. Therefore the route uses the MsalGuard:

Since this route is using an empty path it will be our initial starting point of the app. Starting the app we should be redirected to the Azure AD login right away:



As we have provided the tenant id we also get a branded login experience. After signing in we will be redirected back into our app and we can see the content of the profile component:



Now we can use the MsalService to get some information (from the ID token) about the logged in user in our profile component:

Let’s bind these two properties in our template:

Et voilá!



**Using access tokens to authenticate API calls**

More important limiting our user interface to signed in users it’s to limit access to sensible information stored in our backend systems. Therefore access tokens are being used to authenticate requests. The MSAL library also helps doing this. As a simple example lets try to load the signed in user’s profile picture. To do this we can use the Microsoft Graph, a set of RESTful APIs to interact with data stored in Microsoft 365.

In many cases apps use multiple APIs, e.g. Microsoft Graph and a custom backend, and every endpoint will typically require its own access token that needs to be acquired with a specific scope.

The first step is to create a mapping table of endpoints and scopes that we want to create access tokens for. The idea here is that every time a request is made to one of the provided protected resources in that mapping table, the MSAL library is acquiring a matching access token in the background and passes it into the request’s authorization header. This protected resource map will be part of the MSALAngularConfigFactory method:

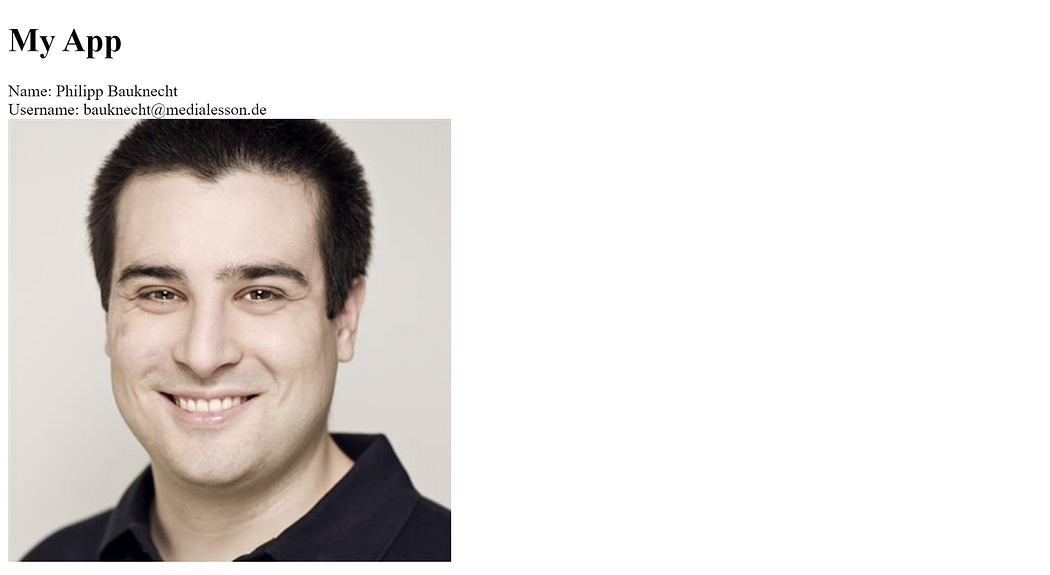
The second part of the configuration requires to provide the MsalInterceptor for the HTTP\_INTERCEPTORS. This basically is the plumbing to intercept http requests and sneak in the access tokens:

So let’s create a new service to call the Microsoft Graph and download the user’s profile photo:

We use this service to fetch the photo in the profile component:

And render it in the template:

That’s it! The user photo is shown:



**Summary**