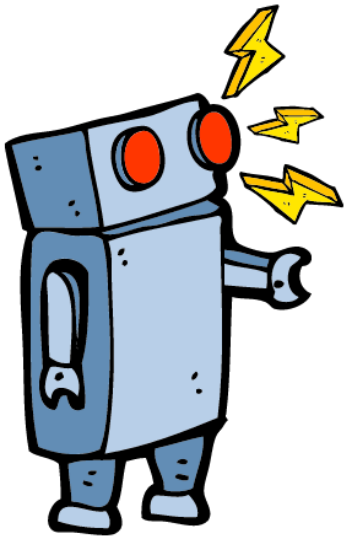


AZURE ACTIVE DIRECTORY HANDS-ON LAB

An Azure Active Directory integrated web single-page-app.

ABSTRACT

This lab document contains a step-by-step guide to create an Azure Active Directory integrated application, that allows you to map existing local users to AAD users.

About

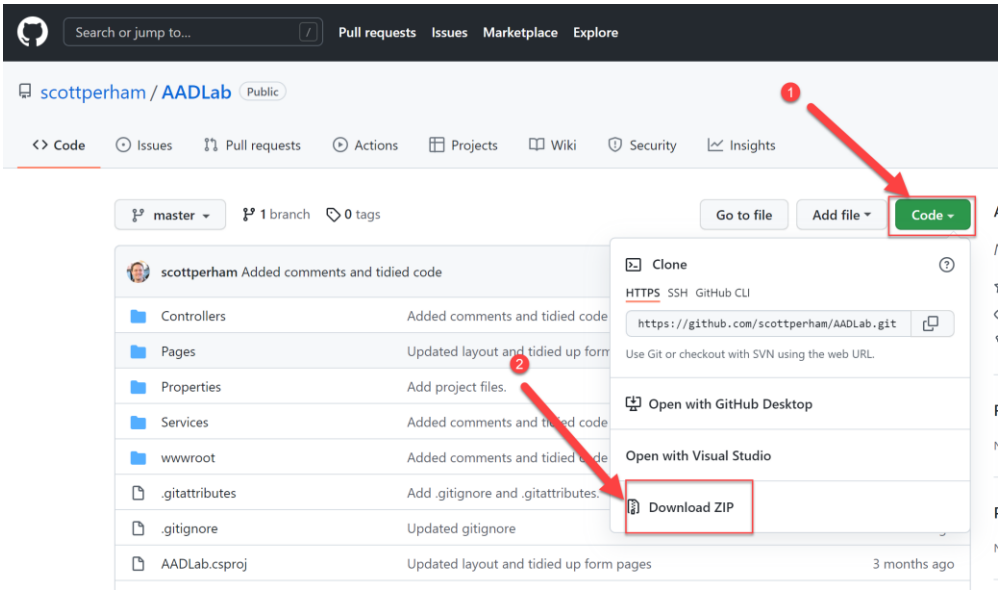
Written for the App Modernisation Series – a series of events for Microsoft Partners created by Jason Cabot, Mike Ormrod, Scott Perham, Luciana Blanchard and Jack Lewis.

Contents

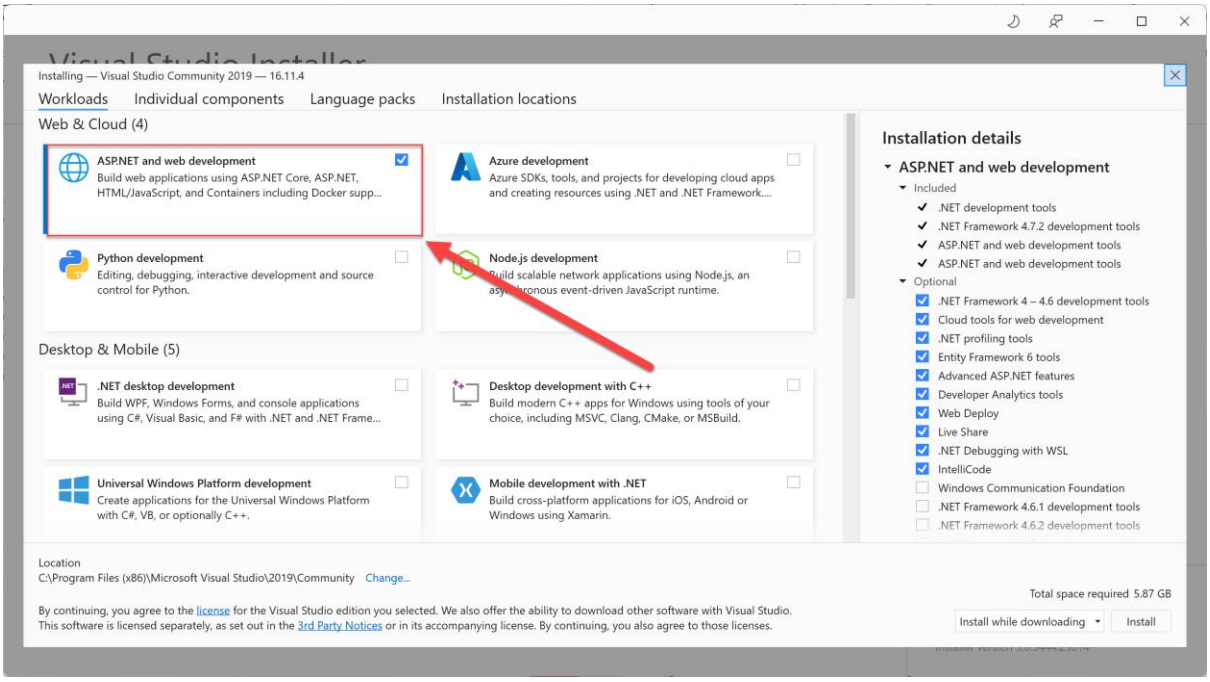
Getting Started

You will need the following to complete this lab:

1. **A Microsoft 365 E5 Tenant**, which can be obtained from https://cdx.transform.microsoft.com/ (for Microsoft Partners) or by signing up to the Microsoft 365 Developer Program – Welcome to the Microsoft 365 Developer Program | Microsoft Docs . Alternatively, a Microsoft 365 E5 Trial will also provide the functionality you need. We do **not** recommend using your production/live Tenant for this lab. We will refer to this Tenant as the **Test/Dev Tenant** in this documentation.
2. The **AADLab App**, which can be downloaded from here, click **Code** and select **Download ZIP**. You'll then need to extract that ZIP to a location on your computer.



3. **Visual Studio**, installed onto your local machine. If you do not have Visual Studio, the Community edition can be downloaded here. When installing Visual Studio, make sure to select **ASP.NET and web development**.



4. **.Net 5 SDK**, which is required to run the App, available here.

Azure AD Integration App Lab

Overview

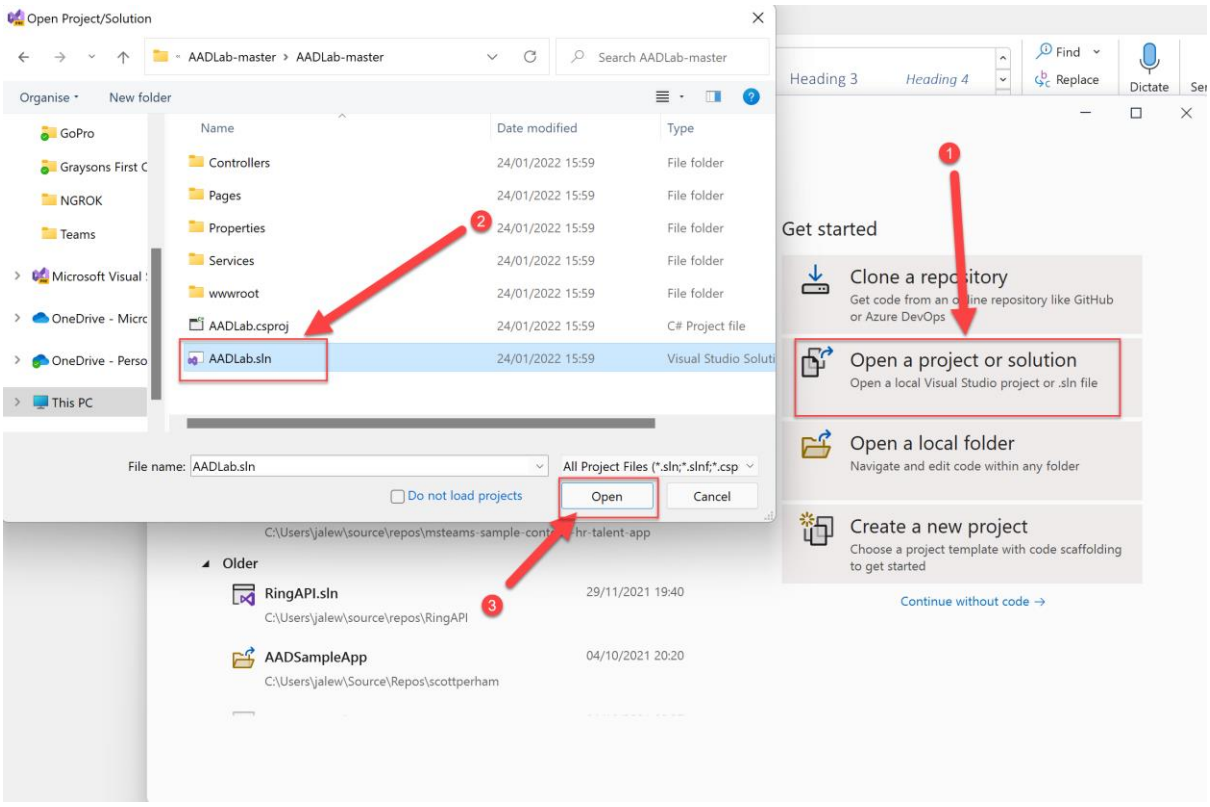
The following steps will walk you through the setup of the web-application, to allow it to run locally on your machine, the setup of the Azure AD App Registration, to enable SSO integration, and will then step you through the code to review how Azure AD SSO and identity mapping works with this application. The purpose of working through this lab is to inform and educate developers on how they can integrate Azure AD SSO into their applications, while still allowing users to retain access to resources they previously had access to when signing directly into your web-application with local usernames and passwords.

Lab Steps

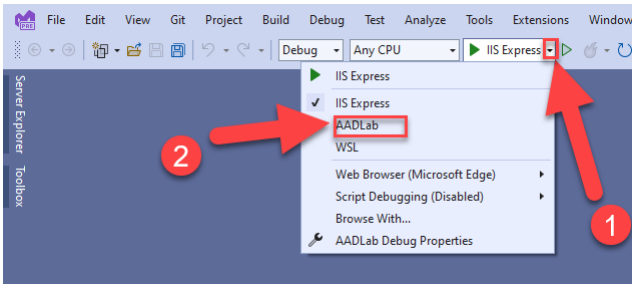
1) Setup the Web-App to run locally

In this step you will setup the web-app in Visual Studio and setup the Azure AD App Registration in Azure. You will then run (debug) the app locally, within Visual Studio, from your local machine.

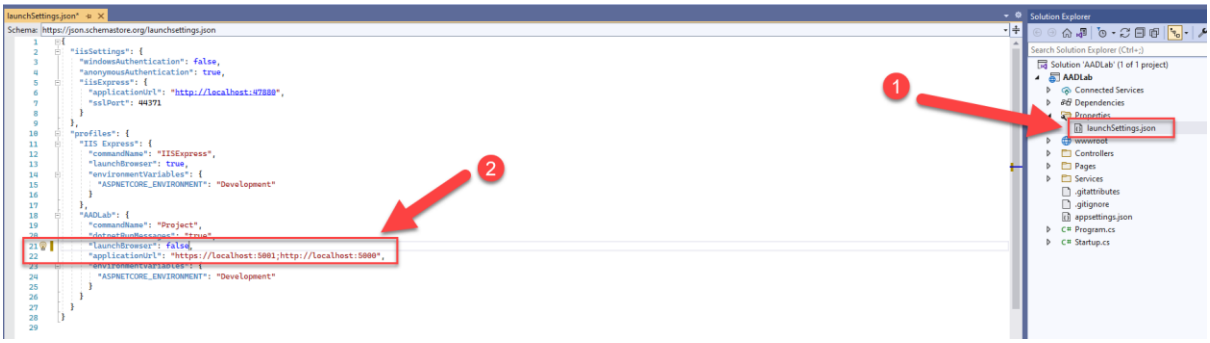
1. Open Visual Studio and open the **AADLab.sln** file found within the extracted Repo you downloaded from GitHub.



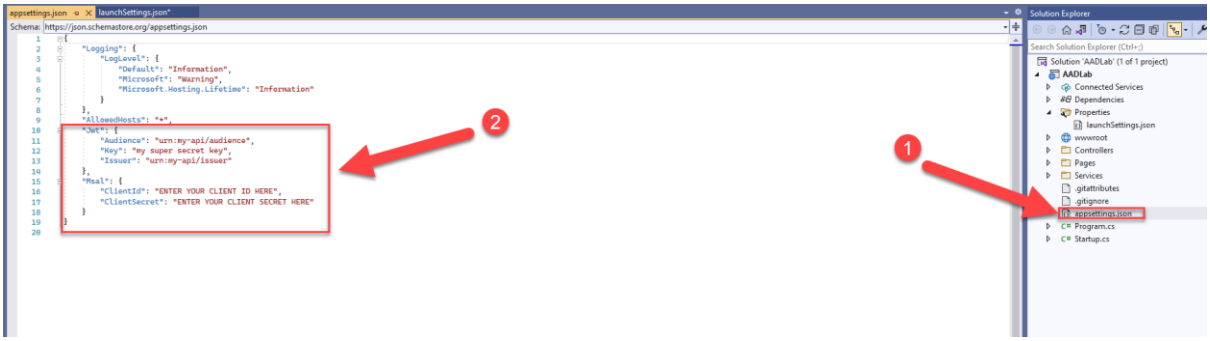
2. When Visual Studio has loaded the solution and associated projects, we need to set the debug properties correctly. In the Visual Studio toolbar, select the **IIS Express dropdown menu** and select **AADLab**.



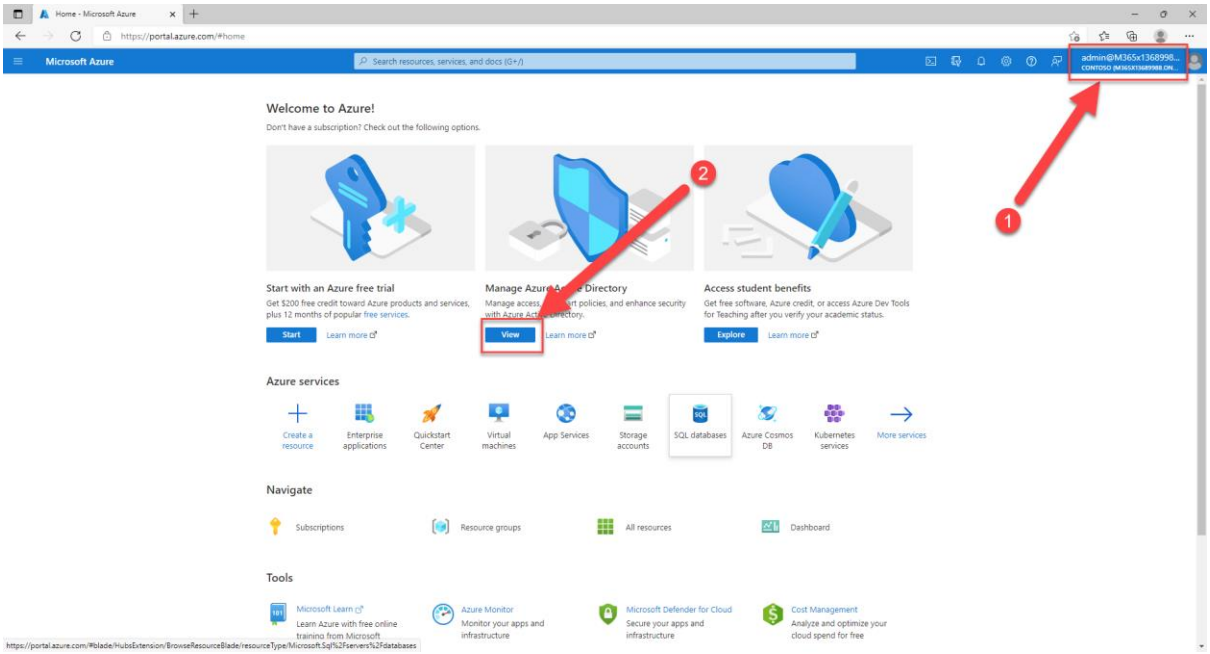
3. On the right-side of Visual Studio, within Solution Explorer, expand **AADLab, Properties** and select **launchSettings.json.** This file contains the properties that are used when we run (debug) the app locally, in Visual Studio. Change the value of **launchBrowser** (line 21) to **false**. Review the **applicationUrl** values (line 22) – these should use port 5001 and 5000 (for HTTPS and HTTP respectively). Hit **CTRL+S** to save or click the Save icon in the toolbar.



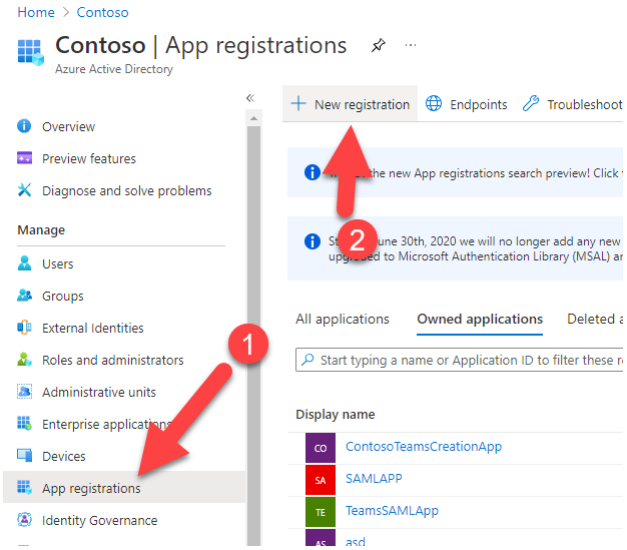
4. Select **appsettings.json**. This file contains placeholder values for **Audience, Key, Issuer, ClientId & ClientSecret**. Before we can run this app, we need to replace these with actual values. We will now configure the application in Azure to discover the ClientId and ClientSecret values.



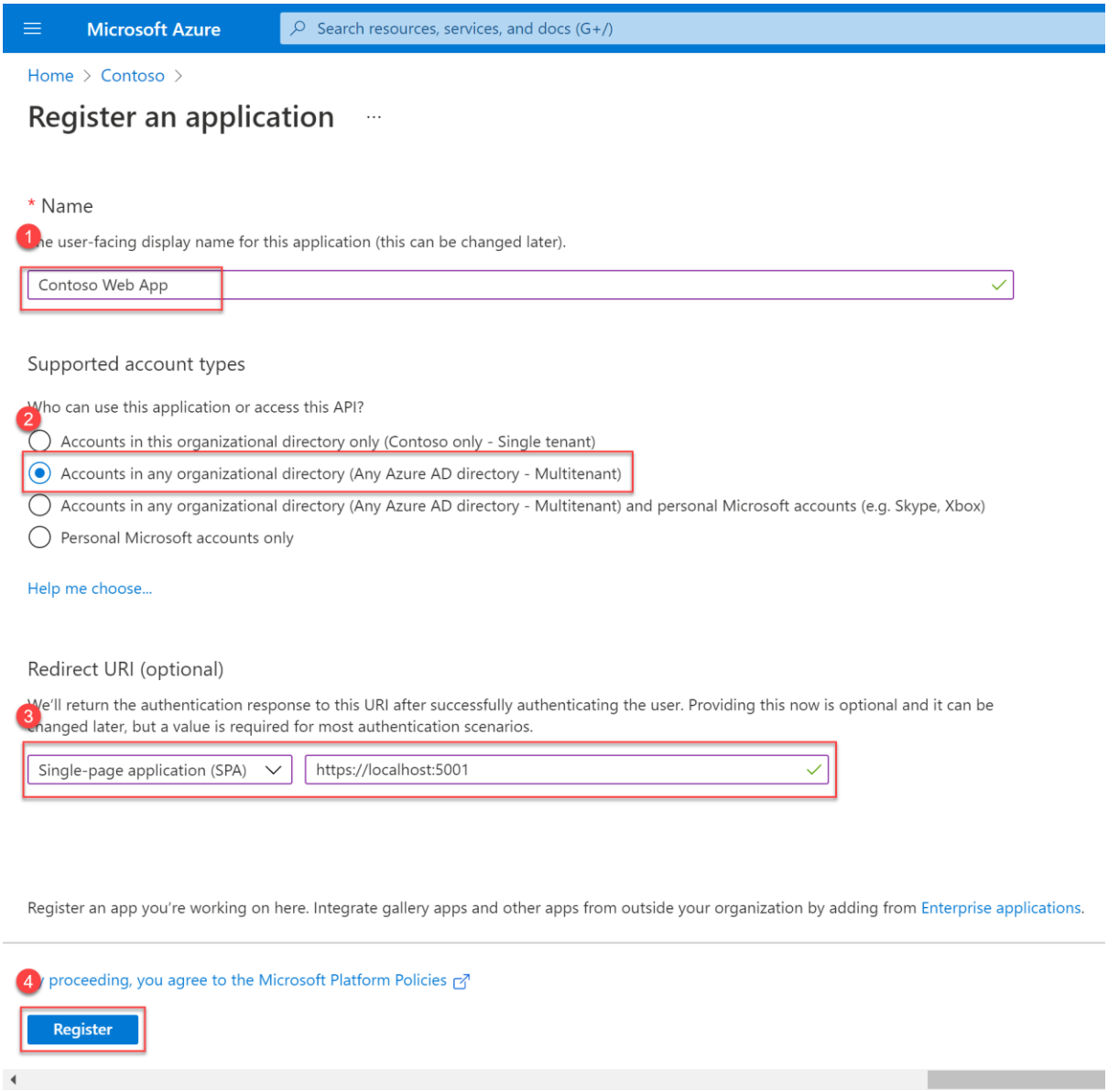
5. We now need to setup the Azure AD App Registration in Azure. This will provide us with the values for the **appsettings.json** file. Navigate to https://portal.azure.com/ and **sign in with the Administrator account from your Microsoft 365 Developer Subscription**. Once you are signed in and have access to an Azure Subscription on that account, Select **Azure Active Directory**.



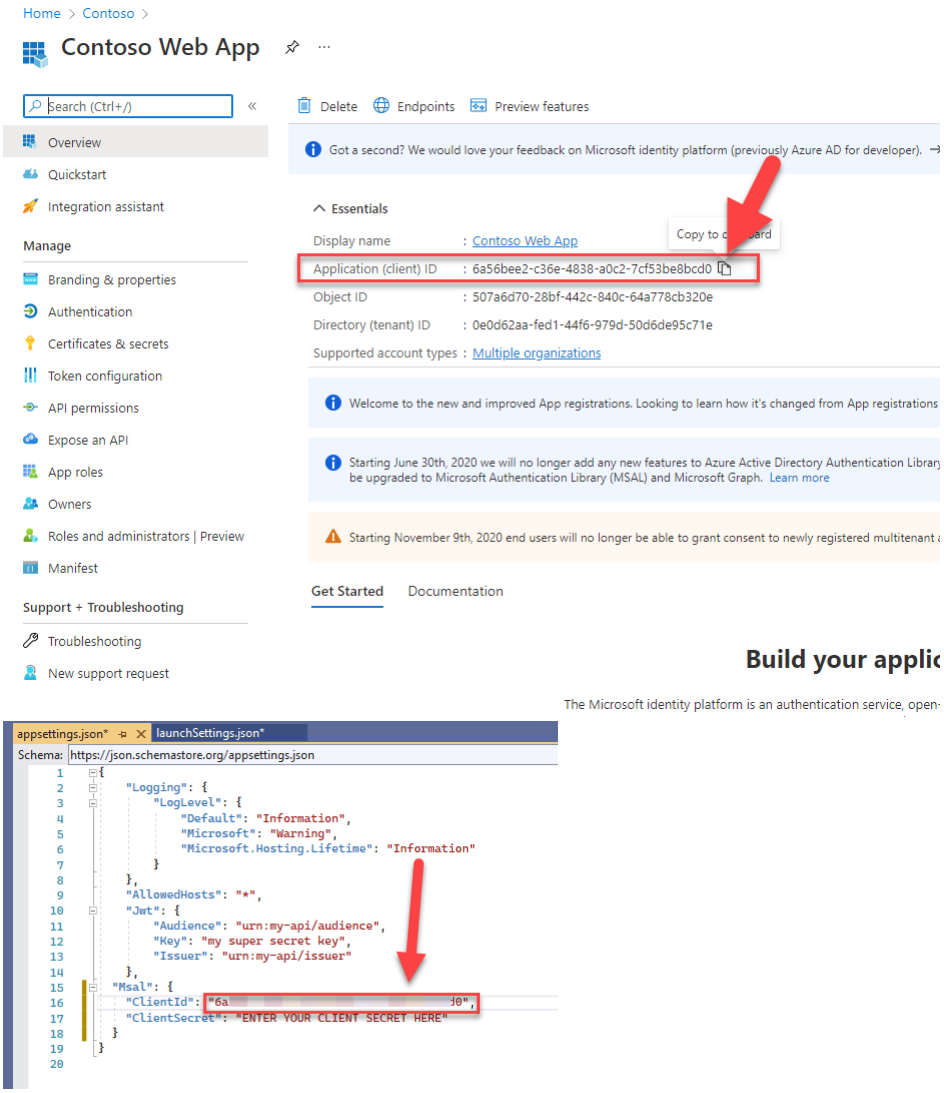
6. Select **App Registrations** and then click **New Registration**



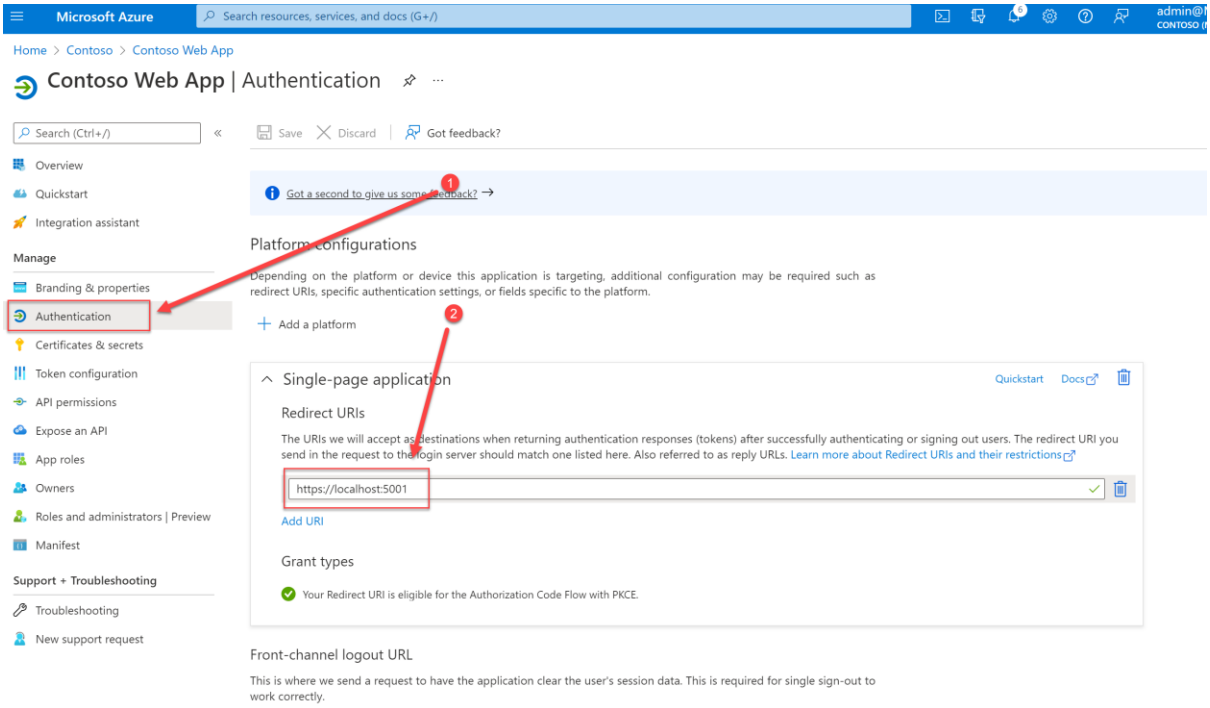
7. Enter a name for your App, such as **Contoso Web App**, select **Accounts in any organizational directory (Any Azure AD directory – Multitenant)**, select **Single-page application (SPA)** from the dropdown menu and then enter your Redirect URI as **https://localhost:5001** – click **Register.**



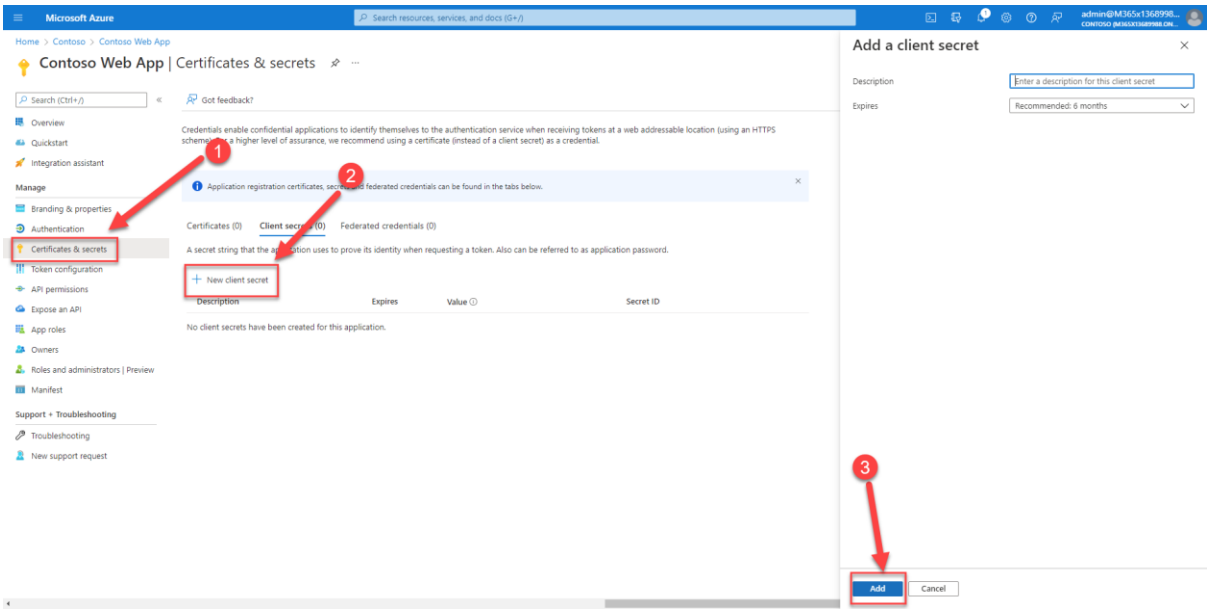
8. On the Overview page, copy the **Application (client) ID** and enter this as the **MicrosoftAppId** value in **appsettings.json** (line 16).



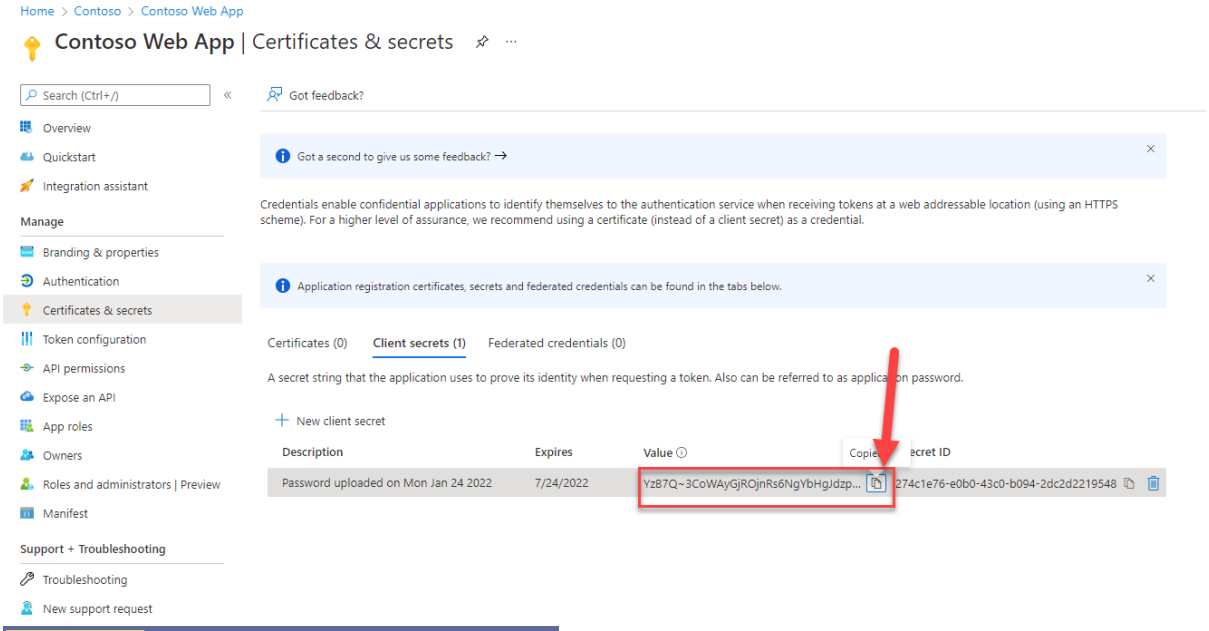
9. Select **Authentication** and review the configuration. You should see a Redirect URI of **https://localhost:5001**. For context, these Redirect URIs are the allowed locations that Azure AD will redirect clients to when an access token has been received via a successful authentication. The SPA URI is used to redirect browsers back to your app, which will then process the Azure AD access token, and then store it for use when calling the APIs.



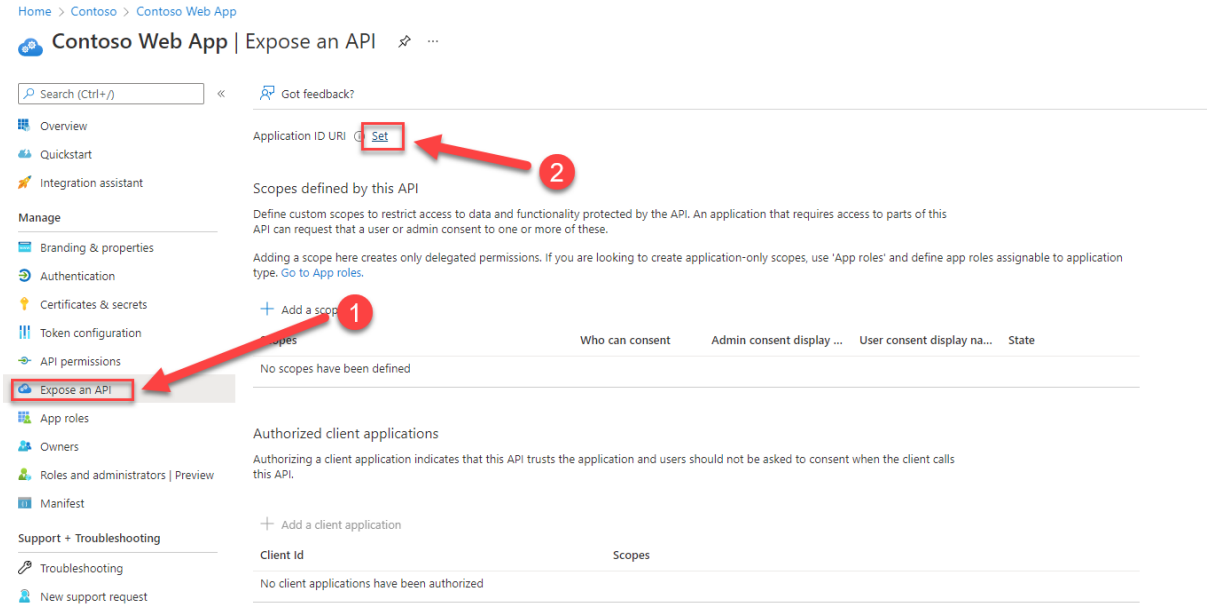
10. Select **Certificates & secrets,** select **New client secret** and click **Add**



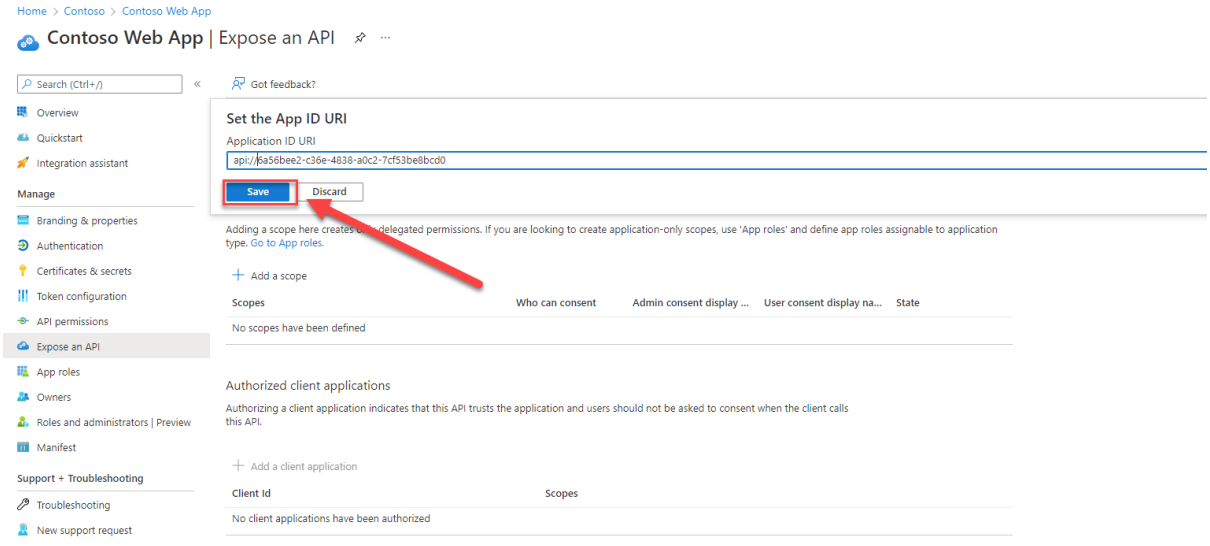
11. Once generated, copy the **Value** of the Client Secret and enter this as the **ClientSecret** value in **appsettings.json** (line 17).



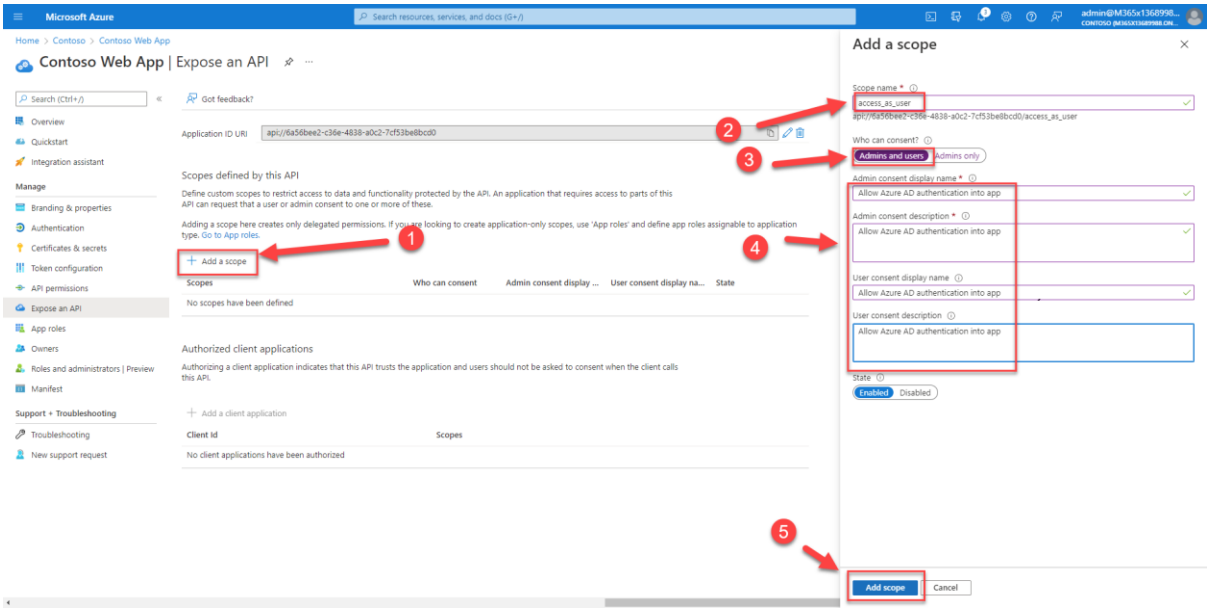
12. Select **Expose an API** and **Set** the Application ID URI.



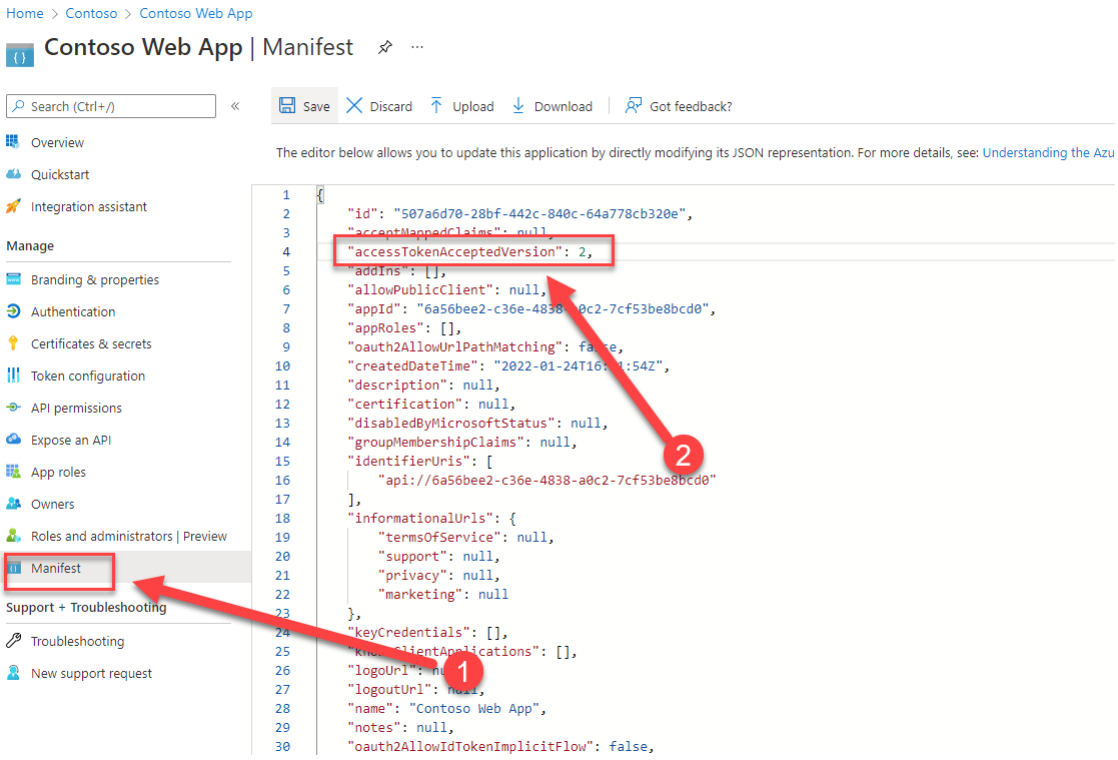
13. For this application, the default URI is fine, click **Save.** *Note: If you plan on making this app capable of doing Microsoft Teams SSO in the future, you must stick to a strict format for your App ID URI, for more details see - [Single sign-on support for tabs - Teams | Microsoft Docs]*



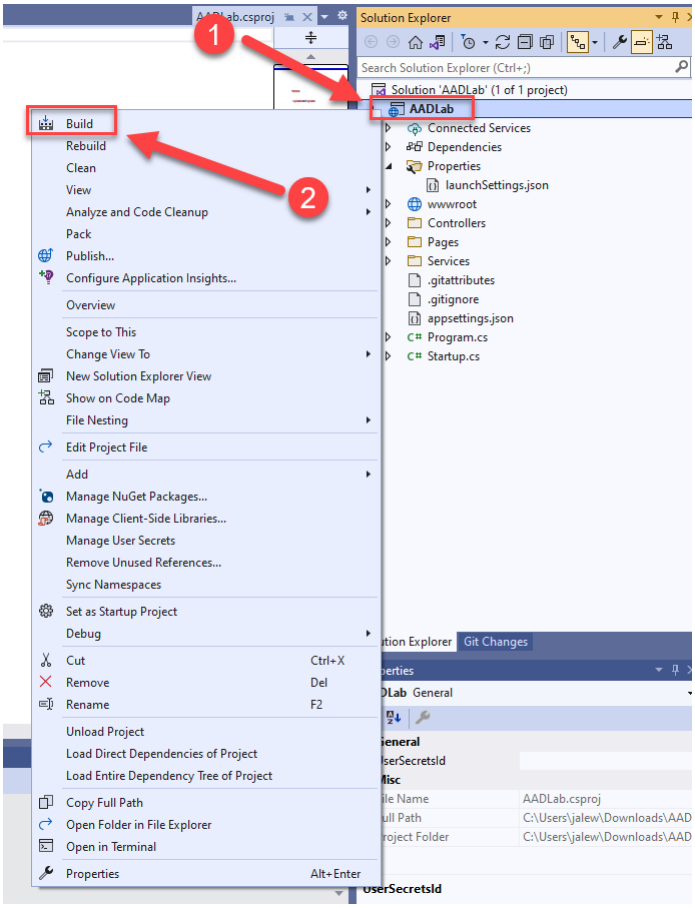
14. Select **Add a scope**. We now need to configure the API scope that will be issued when Azure AD presents your application with an Access Token. Enter the Scope name **access_as_user** select **Admins and users**. For the display names and descriptions enter **Allow Azure AD authentication into app** (note: in a real-world scenario you would populate these with more information). Select **Add scope**



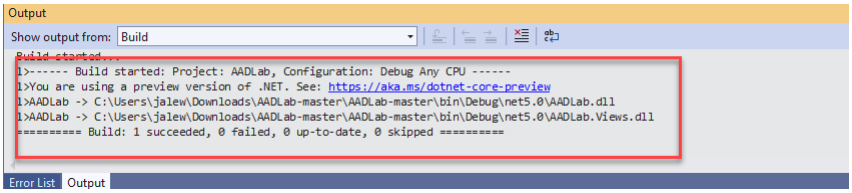
15. The final step we need to do, is to configure the App Registration manifest to only use Access Tokens v2. Select **Manifest**, change the value of **accessTokenAcceptedVersion** to **2** (line 4). Select **Save**.



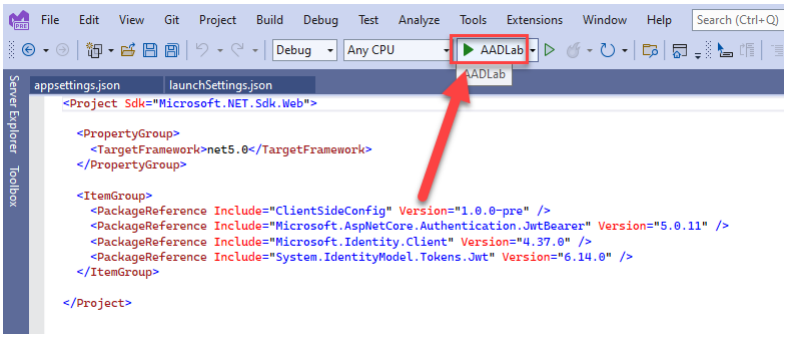
16. The Azure AD App Registration has now successfully been configured. We now need to build and run the app in Visual Studio. **Right click the Solution** and click **Build Solution**.



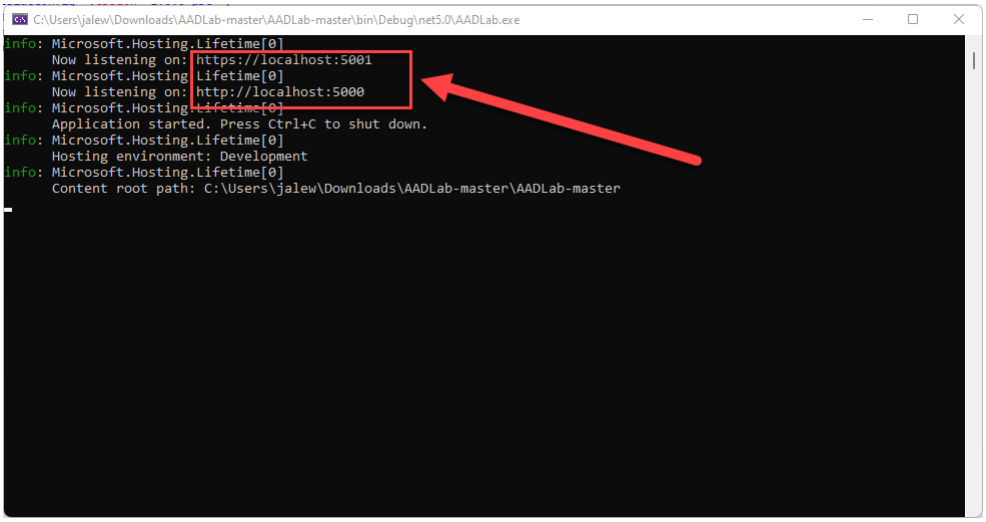
17. At the bottom of Visual Studio, the Output section will begin populating with text. Once the Output confirms that the build succeeds, move onto the next step.



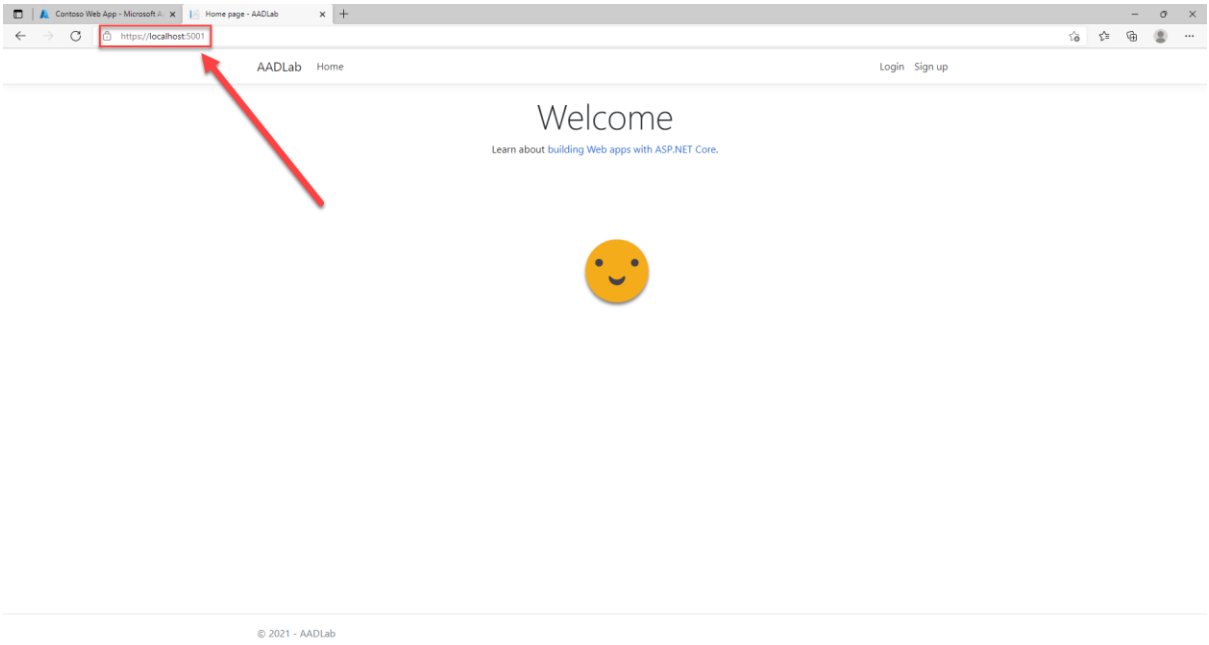
18. Click the **Play** button, to run (debug) the app locally.



19. All being well, you should receive a message in a console window, confirming that the application is running and listening on ports 5000 and 5001.



20. Open your web-browser and navigate to https://localhost:5001 to confirm the web-app is running.

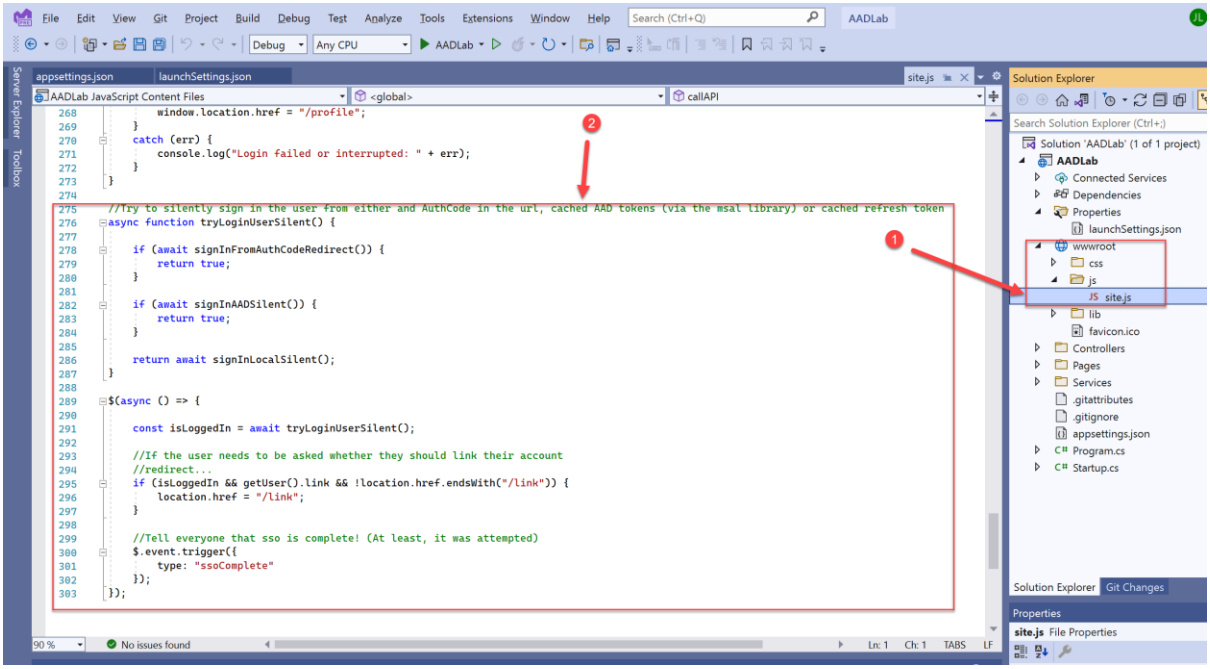


Nice job! You have setup the developer environment and Azure resources required to run this application locally. Now that the application is running, we can review how the application is configured to support both local and Azure Active Directory authentication!

2) Page load / SSO flow

Let's review how the page load flow works, as it includes some interesting points of reference for how Azure AD SSO works in this web-app:

1. Let's review the page load flow, as it includes a few interesting points of reference, in Visual Studio open **Visual Studio,** expand **wwwroot, js** and then select **site.js**. Scroll down to **line 275**:



2. This code is run every time a page is loaded by a client-app (browser):

a. **Line 279**: When using the msal-browser library and the Auth Code Flow (with PKCE), AAD will redirect the user back to our application with an auth code appended to the url. We would like the msal library to handle this code and complete the sign in process.
b. **Line 283**: See if the msal library can sign in the current user.
c. **Line 287**: At this point, we know that we aren't in the middle of a sign in flow and there is no AAD user signed in. Now we try to sign in the user using a local account through locally stored refresh tokens.

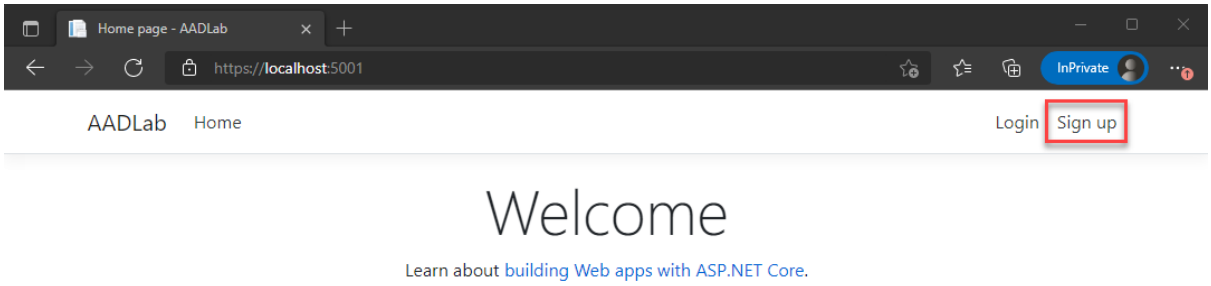
3. Review lines **290-305**, on every page load we run through the following process:
   a. Try to silently log in the user (or complete the sign in process if we are mid-flow)
   b. If the sign in was successful see if there is any reconciliation that needs to occur. In our case, we check to see if we should prompt the user to link identities.
   c. Continue the application. We do this using a custom jQuery event trigger called **ssoComplete**.

3) Create Local Account / Sign In

Let's review how the local login works for this web-application, by signing up for an account, authenticating and reviewing the profile information.

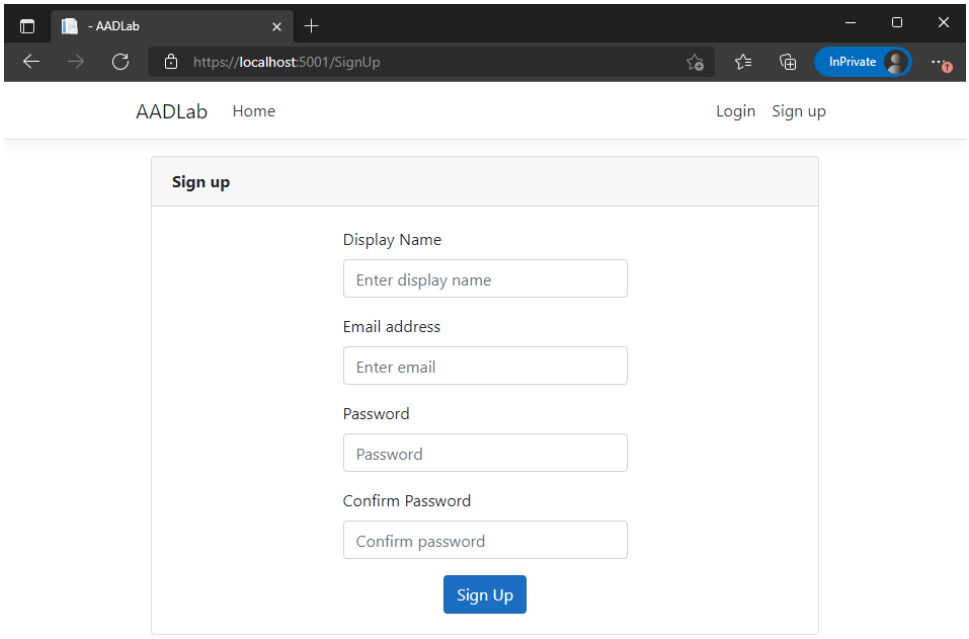
1. First, we will run through the create account / sign in process for a "local user" (That is a user account that only exists within this system).

   From the home page, click **Sign up**.

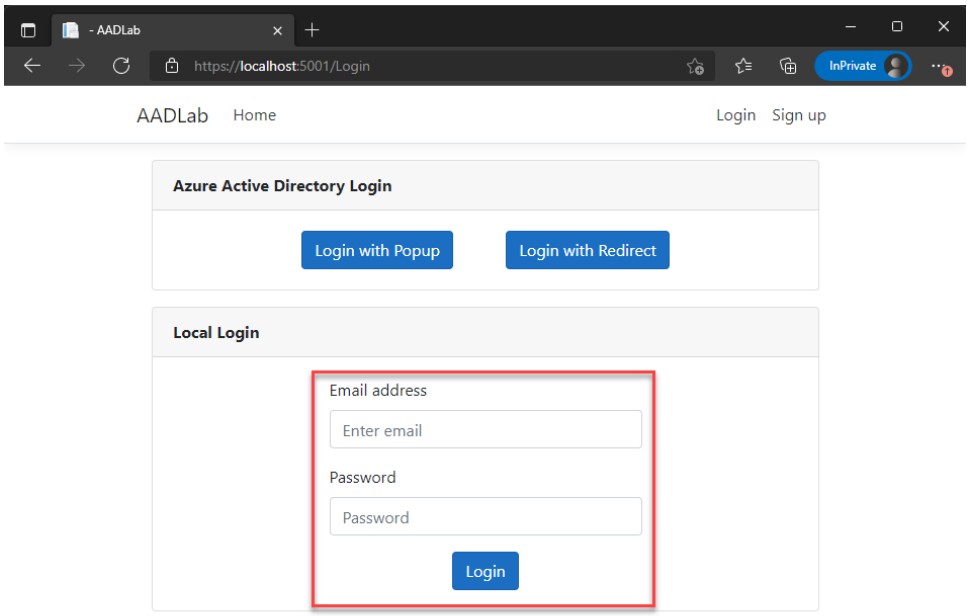


   This will navigate to the Sign up page where we can complete the form to create an account.

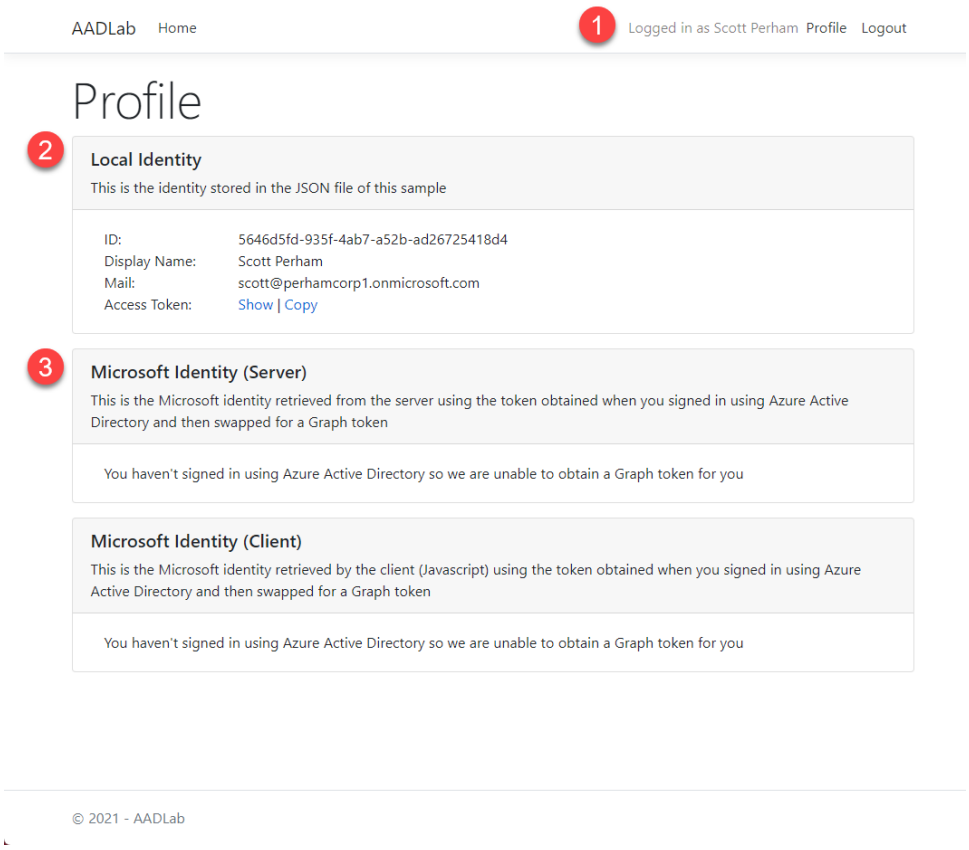
   N.B. As no emails are sent from the application, the email address *does not* have to be real.



2. Once you have created your account you will be redirected to the Login page. From here, enter your email address and password used to create an account in the **Local Login** section of the page.



3. If successful, you will be redirected to the Profile page. This page is protected and will only display data if you have a valid “local token”, that is, a token that can be used to call our APIs.



4. A few points of interest:
   a. You are signed in as denoted by the "Logged in as…" message at the top of the screen
   b. Your "local identity" is displayed in the topmost box on the page
   c. As you have only signed in using a local identity, there is no Microsoft profile for you

5. From the snippet below from **Login.cshtml**. The login page uses a simple jQuery event hook to submit the form fields to the backend API.

```
<script type="text/javascript">

    // When the DOM is ready...
    $(() => {
        // Hook the submit event of the login form
        $("#loginForm").submit(async (e) => {

            // The default behaviour of an HTML form submit event is to POST to the current URL
            // we want to prevent that because we are handling the submit from the client
            e.preventDefault();

            // Grab the contents of the email and password input boxes
            const email = $("#email").val();
            const password = $("#password").val();

            // Attempt to log in using the entered crendentials
            const [success, error] = await signInLocal(email, password);

            if (!success) {
                // There was a problem, show the error
                $("#invalidEmail,#invalidPassword").html(error).show();
            }
            else {
                // Success! Redirect to the profile page
                window.location.href = "/Profile";
            }
        });
    });

</script>
```

6. This is done via the **signInLocal** method in ***site.js***:

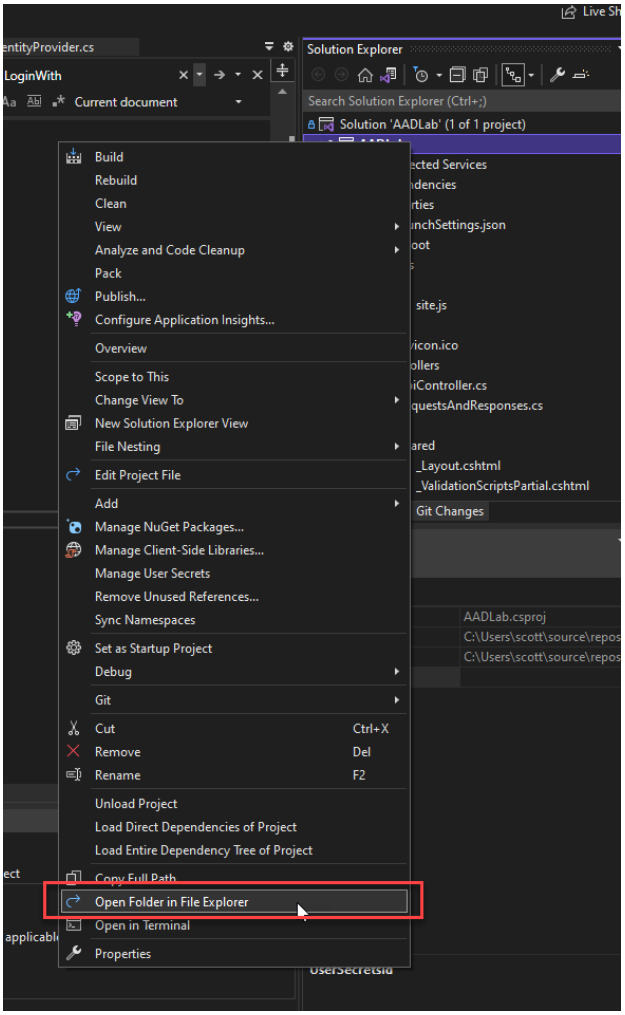
```
108   //Try to sign in the user with a locally defined email and password
109   async function signInLocal(email, password) {
110       const { success, error, result } = await callAPI("/api/loginLocal", {
111           email: email,
112           password: password
113       });
114
115       if (success) {
116           //Save the user info
117           setUserChanged(result.displayName, null, result.accessToken, null, false);
118           //Save the refresh token info
119           cacheServerToken(result.refreshToken, result.tokenExpiry);
120       }
121
122       return [success, error];
123   }
```

7. As you can see, this method calls the API */api/loginlocal* and if successful caches the users identity information and refresh token. This is then used on subsequent page reloads to silently log in the user without them having to manually sign in until the refresh token expires.

4) Application data

This application uses a dummy identity provider backed by a JSON file. This means that it is persistent between runs, we can easily see what data is being stored against a user and we can easily reset the application. <u>This file will be created the first time an account is created or signed in.</u>

The data file is located in a Data folder in the runtime directory of the dotnet application. The easiest way to locate this is to right click on the project in Visual Studio and select Open Folder in File Explorer.



Then navigate to **bin\Debug\.net5.0\Data**. The identities.json file contains the identities that have been created in the application.

If you want to reset the data, you can simply delete this file, restart the application in Visual Studio and close and reopen your in-private browsing session.

5) Page and API protection

Let's review how the pages/routes of this single-page-app are protected, and how authentication against our APIs, to pull data from the server, works:

1. In this application, the Profile page is protected. It will only display information about the current user and only if there is a user currently logged into the application.

   This protection is performed by checking the existence of the cached user information obtained during login. If the cached identity doesn't exist, then we simply redirect the user back to the home page as seen in the following snippet from **Profile.cshtml**:

```
// When the SSO process has completed...
$(document).on("ssoComplete", async () => {
    // Get the cached user
    const user = getUser();

    // If there is no cached user then no one is logged in...
    if (!user) {
        // ...so redirect to the home page
        window.location.href = "/";
        return;
    }

    // Call the API endpoint to retrieve profile information
    const { success, error, result } = await callAPI("/api/profile", {
        accessToken: user.aadToken
    }, user.apiToken);

    const profile = result;

    //Populate the UI
    populateLocalIdentity(result.localIdentity, user.apiToken);
    populateMicrosoftIdentity(result.microsoftIdentity, user.aadToken);
    await populateLocalMicrosoftIdentity(user.graphToken);

});
```

2. To protect the APIs, we expect a bearer token to be passed in the Authorization header of any protected request. This token validation is performed by the dotnet Framework and configured in **Startup.cs**:

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<ITokenService>(new TokenService(Configuration));
    services.AddSingleton<IIdentityProvider>(new JsonFileIdentityProvider());

    services.AddControllersWithViews();

    services.AddRazorPages();

    services.AddAuthentication(x =>
    {
        x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

    }).AddJwtBearer(x =>
    {
        x.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(Configuration["Jwt:Key"])),
        };
    });
}
```

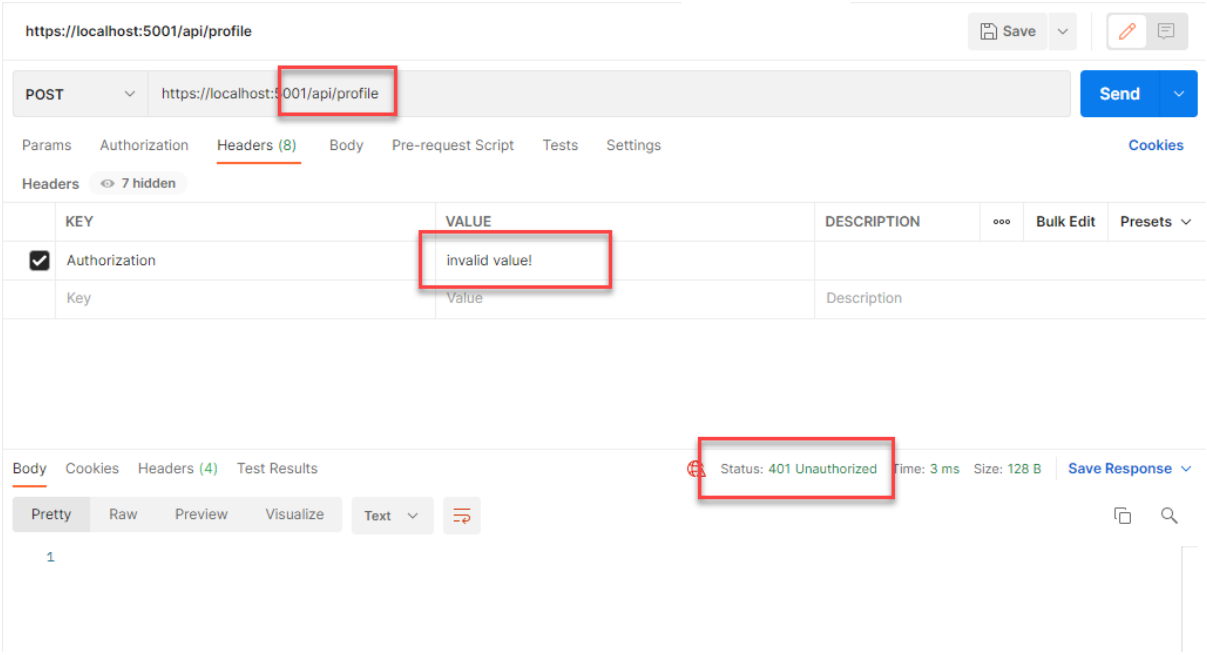
3. And then enabled further down the same .cs file:

```
79          app.UseAuthentication();
80          app.UseRouting();
81          app.UseAuthorization();
```

4. In order to protect a route, we use the Authorize attribute on any API calls we want to protect as seen in **ApiController.cs**:

```
252         [HttpPost("Profile")]
253         [Authorize]
254         public async Task<IActionResult> GetProfile([FromBody] GetProfileRequest request)
255         {
```

5. Failing to provide a valid bearer token when making calls to the Profile API will result in a 401 Unauthorized response being returned as depicted by the following from Postman.

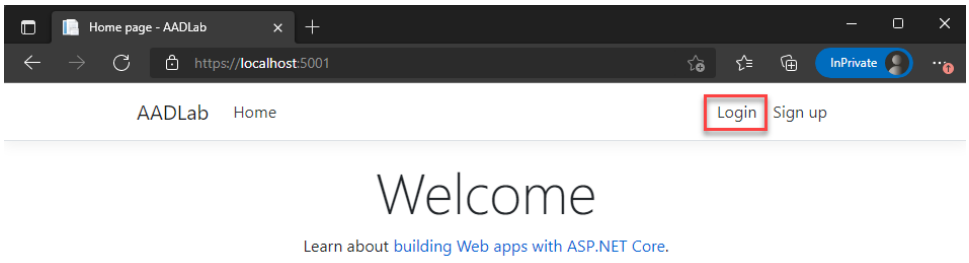


6) MSAL Login

Let's review how the MSAL SDK and Azure AD SSO works with this Single-Page App:

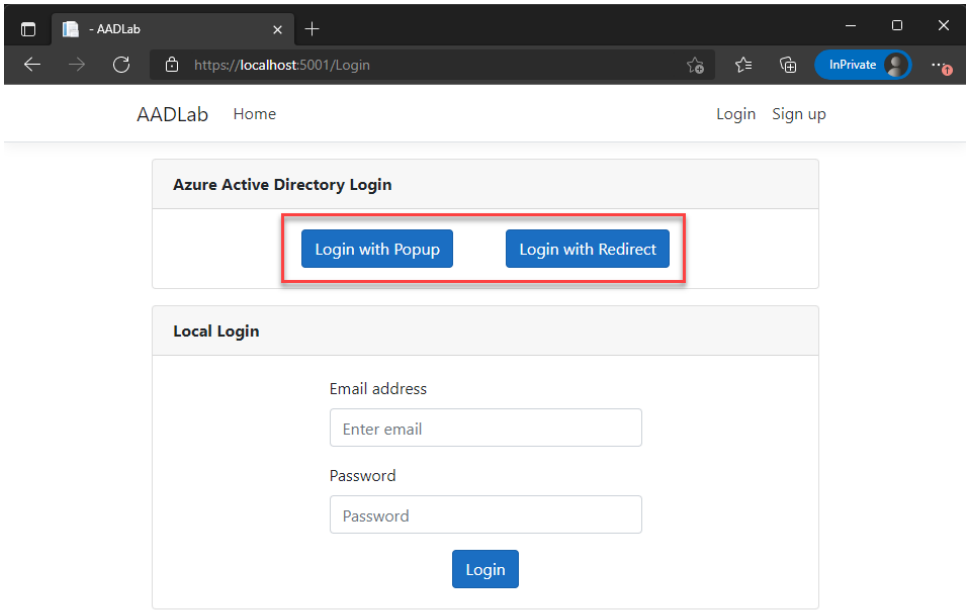
1. Next, we will sign in using a Microsoft organizational account.

   From the home page click **Login**.

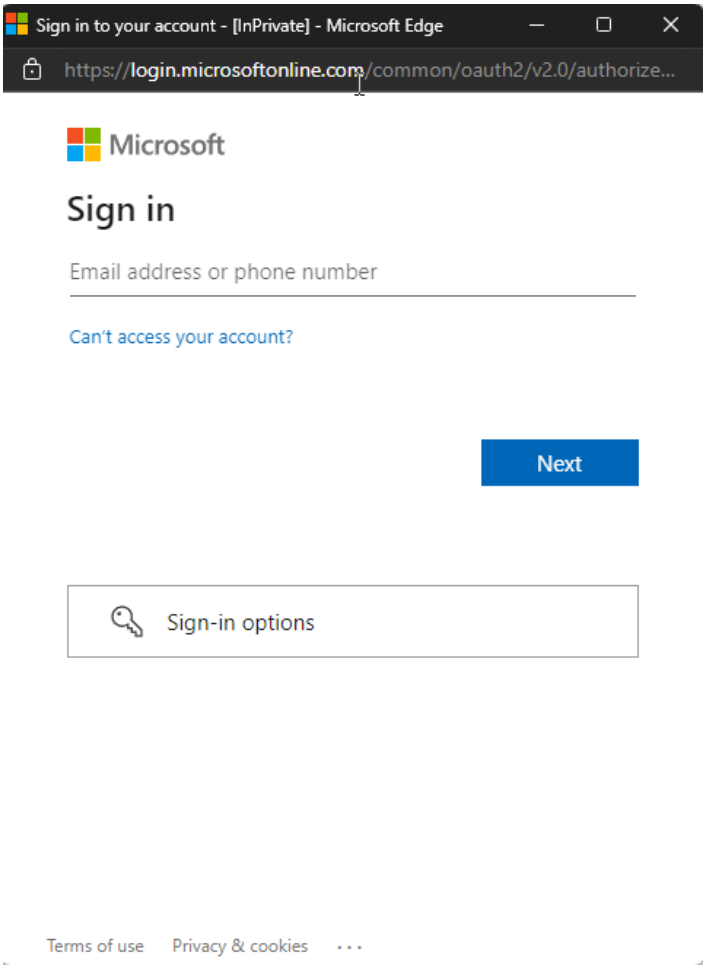


   This will display the Login page. This time we will login using a Microsoft organizational account.

2. Click **Login with Popup** or **Login with Redirect**.

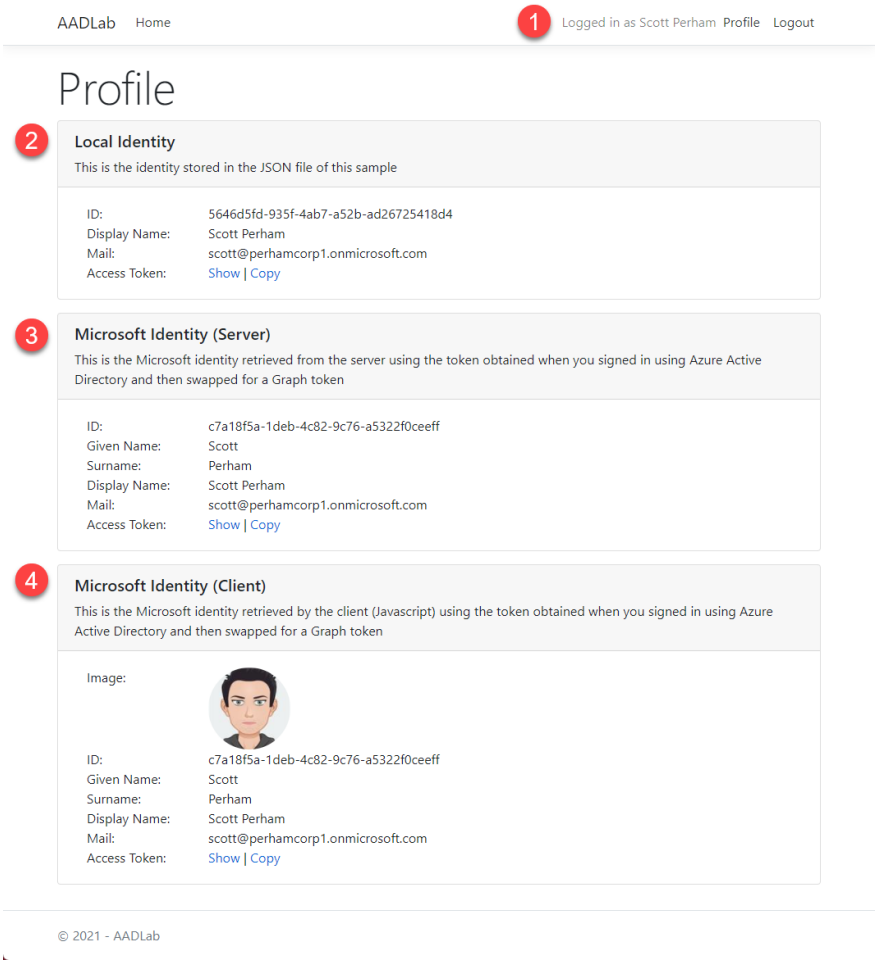


3. You will either be redirected to the Microsoft login page or be presented with a popup depending on which button was clicked, but the process is the same.



4. After successfully signing in with your Microsoft account you will be automatically redirected to the profile page.

You will see this time that there is a lot more information shown!



a. There is a user signed into the application
b. Even though the user signed in using AAD, we have created a local account for that user as well. This represents the fact that a lot of systems have their own Identity Provider and require a local identity in order to generate a valid token for future API calls.
c. We now have a Microsoft Identity displayed. The first is the identity that has been obtained through server-side code as seen in **ApiController.cs**:

```
252         [HttpPost("Profile")]
253         [Authorize]
254         public async Task<IActionResult> GetProfile([FromBody] GetProfileRequest request)
255         {
256             //Pull out the nameidentifier claim from the token
257             var idClaim = User.FindFirst("nameidentifier");
258
259             //Find the identity
260             var identity = await _identityProvider.GetUserById(idClaim.Value);
261
262             GraphMeResult microsoftIdentity = null;
263
264             //If we've passed an AAD access token
265             if (!string.IsNullOrEmpty(request.AccessToken))
266             {
267                 //Swap the token
268                 var graphAccessToken = await _graph.GetOnBehalfOfToken(request.AccessToken);
269
270                 //Get user info from Graph
271                 microsoftIdentity = await _graph.GetMe(graphAccessToken);
272             }
273
274             return Ok(new
275             {
276                 LocalIdentity = identity,
277                 MicrosoftIdentity = microsoftIdentity
278             });
279         }
```

5. The second Microsoft Identity is the same identity but obtained through client-side code as seen in **Profile.cshtml**, line **70**:

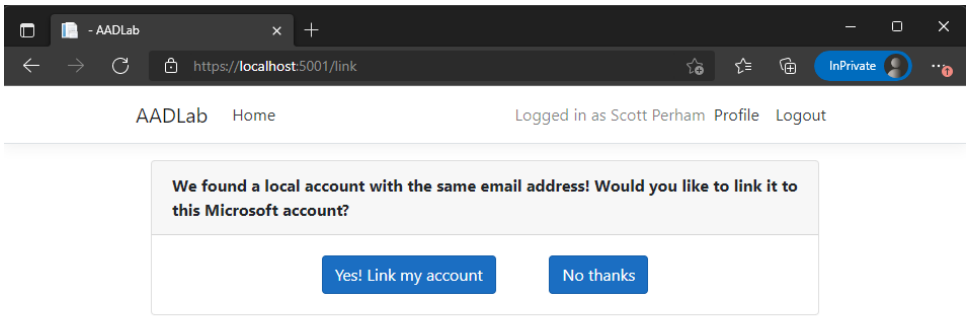
```
64      async function populateLocalMicrosoftIdentity(graphToken) {
65          if (!graphToken) {
66              $("#localMsIdLoading").html("You haven't signed in using Azure Active Directory so we are u
67              return;
68          }
69
70          const { success, err, result } = await callAPI("https://graph.microsoft.com/v1.0/me", null, gra
71
72          if (success) {
73
74              $("#localMsId_Id").html(result.id);
75              $("#localMsId_givenName").html(result.givenName);
76              $("#localMsId_surname").html(result.surname);
77              $("#localMsId_displayName").html(result.displayName);
78              $("#localMsId_mail").html(result.mail);
79              $("#localMsId_accessToken").html(graphToken);
80
81              $("#localMsId").show();
82              $("#localMsIdLoading").hide();
83
84              await downloadProfilePicture(graphToken);
85          }
86      }
```

7) Link AAD account to existing local account

Now let's review how SSO with Azure AD works, and how the 'linking' of accounts works so that regardless of how you sign-in (local or Azure AD) you are essentially the same user when calling the web-app's APIs:

1. There is also the scenario where you have created a local account with your organizational address and subsequently sign in using Azure Active Directory. In this instance, the application will determine that you already have a local account with the same email address and prompt you to make a choice whether to link these two accounts or not.

N.B. It may be easier to reset your data file to see this in action. Follow the steps above to locate the Data folder and remove it. Remember to restart your project in visual studio and restart your in-private browsing session too!



2. When you login using your Microsoft Account, the code checks to see whether there is an existing, local account linked. This is done by looking for a local identity that has an AAD Object Id and Tenant Id associated with it. If it can't be found it then checks to see if it can find a local account with the same email address. In real world scenarios, this might require additional checks, but for the purpose of the example we are just checking the email address of the user. Consider the below code snippet from **ApiController.cs**:

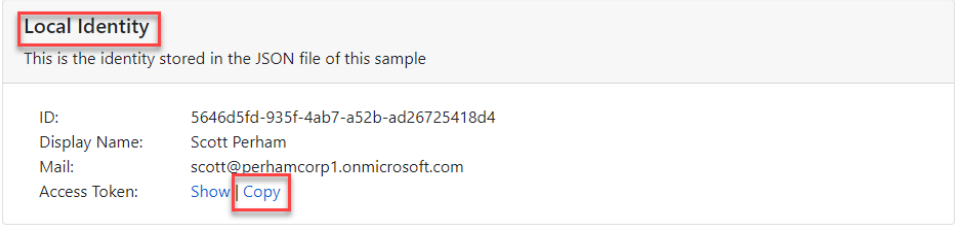
```
156             //Find the identity based on the AAD object ID and tenant ID
157             var identity = await _identityProvider.GetUserByOidAndTid(me.Id, org.Value[0].Id);
158
159             //If there isn't one...
160             if (identity == null)
161             {
162                 //Find the user by email
163                 identity = await _identityProvider.GetLocalUserByEmail(me.Mail);
164
165                 if (identity != null)
166                 {
167                     if (!shouldLink)
168                     {
169                         //Do link flow...
170                         return GetLoginResult(identity, null, graphAccessToken, true);
171                     }
172
173                     if (saveLink)
174                     {
175                         //Link!
176                         identity.AADOID = me.Id;
177                         identity.AADTID = org.Value[0].Id;
178                     }
179                     else
180                     {
181                         identity = null;
182                     }
183                 }
```

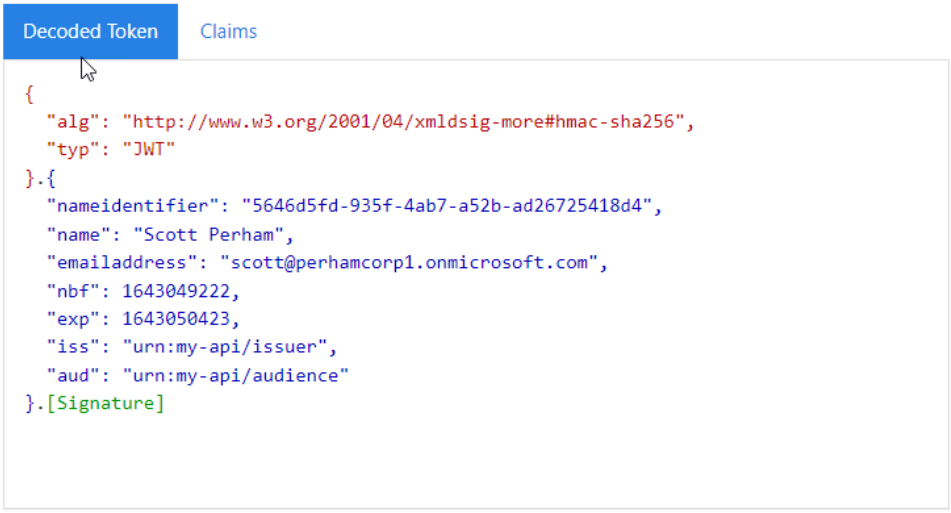
a. **Line 178**: After finding a linkable account (that is, a local account with the same email address) the method returns a login result back to the client that suggests they should ask the user whether they want to link the accounts. This in turn will display the Link Account interface as seen above.
b. **Lines 176 & 177**: This is the code that saves the relevant information to the current identity so that future login attempts will immediately see that the account is linked. We use Object Id and Tenant Id because these values will never change.

8) Let's look at the tokens

In this step you will review the tokens that the application utilises and presents back to the users in the profile page.

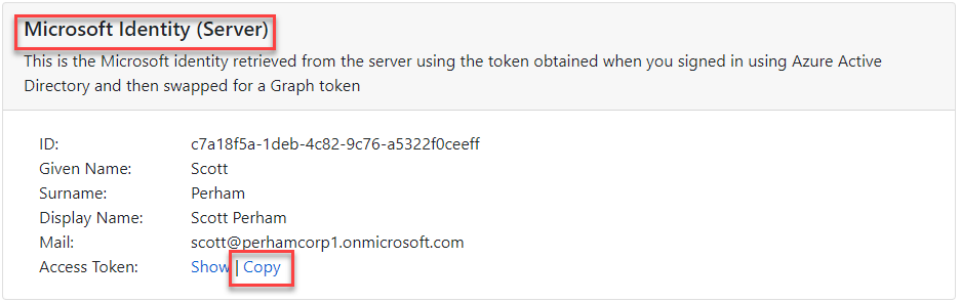
1. First, let's review the Local Identity (The token issued by your application that authorizes you to call our own APIs).
2. Select **Copy**, and then paste the decoded access token into **https://jwt.ms/** . This will decode the token, and expose the claims that are contained within the access token. This access token does not contain anything hugely interesting, it is generated server-side by the application, during the sign-in process, and passed back down to the client to be used in subsequent API calls, to pull information from the server. If you check the signature, you will find that the token is signed and is trusted by the server.

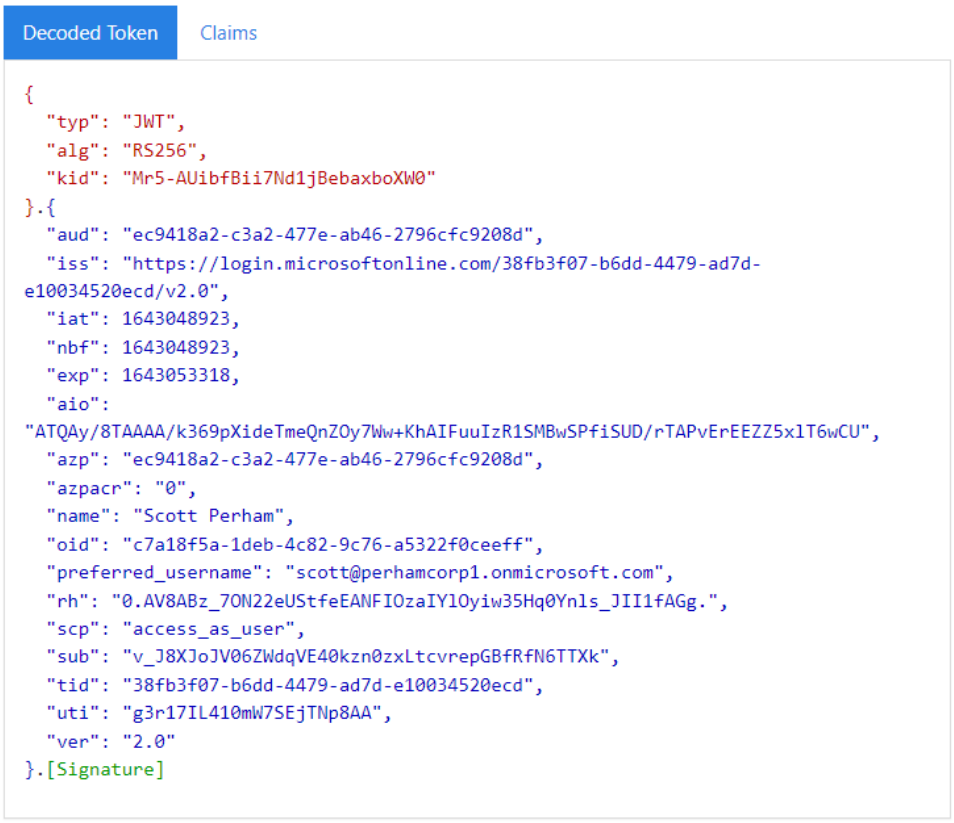




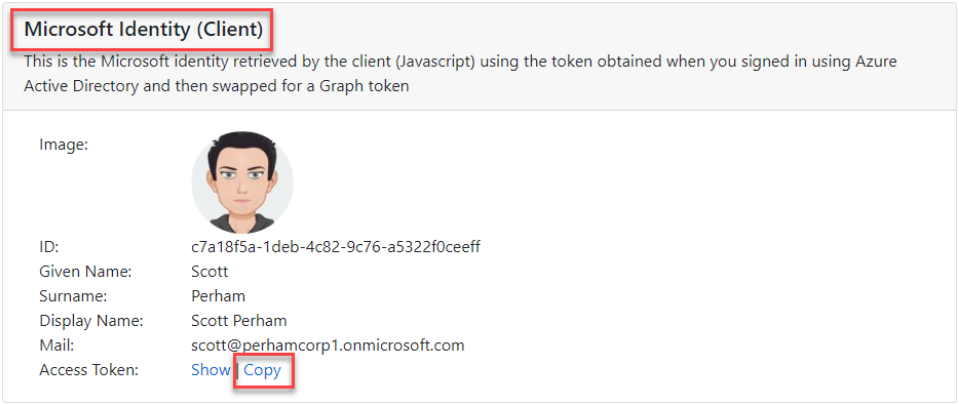
3. You can see that this is indeed the token generated by this application (and not AAD) because the issuer (**iss**) and audience (**aud**) claims are those defined in our **appsettings.json** file:

4. The second token **Microsoft Identity (Server)** is slightly more interesting, as it is not issued by the web-app server, instead it is issued by Azure Active Directory. **Copy** the token and paste it into **https://jwt.ms/** to expose the claims and decode the access token. This access token contains some interesting claims, firstly the **aud** claim should match your Azure AD application ID, and the **iss** claim signifies that the access token was issued by the Azure Active Directory service. Further claims of interest include:
    a. **Name:** the name of the user in Azure AD
    b. **Oid:** the object ID of the user in Azure AD, guaranteed unique when paired with the TID
    c. **Tid**: the tenant ID that the user belongs to, this can be used to make decisions about which organisation the user belongs to, which is useful for applications that promote collaboration, allowing you to group users in the same organisation together.
    d. **Scp**: this is the scope claim. As the audience is our own application, the scope is **access_as_user**, although you could create more claims, and issue them as you deem appropriate to allow you to authorise access to certain resources.
    e. **Exp**: when the token will expire and should no longer be accepted by your server for accessing resources
    f. **Preferred_username:** this is the claim we are using to match Azure AD users to existing users that have only ever signed into our app locally.
5. Alongside the claims included in this token, you can also configure additional claims to be included in the token alongside these default claims, these can include; group membership, MFA/security information, Azure AD roles (such as if the user is a global admin), email, UPN and other information.





6. Finally, and possibly most interestingly, the server returns to the client an access token that can be used to make API calls to the Microsoft Graph Service. Microsoft Graph API allows you to create, read, update and delete information that resides in Microsoft 365, this includes in SharePoint Online/OneDrive, Exchange Online, Microsoft Teams, Azure AD and other services. This token is gathered by the server, using an on-behalf-of authentication flow, to swap the Azure AD Access Token where the audience is your application, for an Azure AD Access Token where the audience is Graph API. This token, then allows the client to pull information from Graph API, and in this sample application, is used to pull the user profile image (if one exists), the application then processes this and shows it in the profile page. To view more details about this token **copy** it and paste it into **https://jwt.ms**



7. Let's review the claims included in the Graph API access token.
    a. Firstly, the **aud** is a well known GUID, this is Microsoft Graph API
    b. The **amr** claim details how the user authenticated, in this case, it was just password and not an MFA
    c. **Scp** contains the customer approved scopes that dictate what you can and can't do with the Graph API, in this case **openid** profile **User.Read** and email are minimal scopes and only let you pull information about this user. Other scopes, such as Mail.Read allow you to read the user's mail. You can also request offline_access, to be presented with a refresh token, so that you can continue to make calls to Graph API when the user is no longer present in your application - useful for processing information, etc…

And that’s it for the tokens! Importantly, you will have learned that local access tokens, used to authenticate against the web-apps APIs, Azure AD Access Tokens and Graph API Access Tokens can all co-exist together, to provide customers with freedom of choice in regards to how the login to your app, but also provide your app with additional functionality in the form of Graph API and allowing your app to integrate with Microsoft 365.

Congratulations! You have now successfully completed this lab. We hope that you found this lab, and the associated lab materials useful. We look forward to seeing what Apps you build as a result of attending this lab!

Lab Complete.