Developing with Power BI Embedding using .NET 5

In this lab, you will create a new .NET 5 project for an MVC web application and then you will go through the steps required to implement Power BI embedding. You will use the new Microsoft authentication library named *Microsoft.Identity.Web* to provide an interactive login experience for users and to acquire app-only access tokens which you will need to call the Power BI Service API. After that, you will write the server-side C# code and the client-side JavaScript code to embed a simple Power BI report on a custom Web page. In the later exercises of this lab, you will add project-level support for Node.js, TypeScript and webpack so that you can migrate the client-side code from JavaScript to TypeScript so your code receives the benefits of strong typing, IntelliSense and compile-time type checks.

These lab exercises require that you have a Microsoft 365 tenant in which you have Power BI administrator permissions and the ability to create new Azure AD groups and applications. If you need a new Azure AD tenant for a development environment, you can follow the steps in Create Power BI Development Environment.pdf.

To complete this lab, your developer PC must be configured to allow the execution of PowerShell scripts with Windows PowerShell 5. Your developer workstation must also have the following software and developer tools installed.

1) **PowerShell cmdlet library for AzureAD** – [download]
2) **.NET 5 SDK** – [download]
3) **Visual Studio Code** – [download]
4) **Node.js** – [download]
5) **Visual Studio 2019** (optional) – [download]

Please refer to this setup document if you need more detail on how to configure your developer workstation to work on this tutorial.

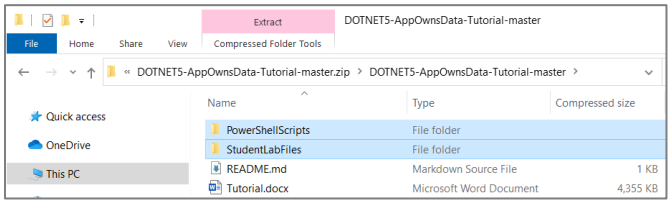
Exercise 1: Create a New Azure AD Application

In this exercise, you will begin by copying the student files into a local folder on your student workstation. After that, you will use the .NET 5 CLI to create a new .NET 5 project for an MVC web application with support for authentication.

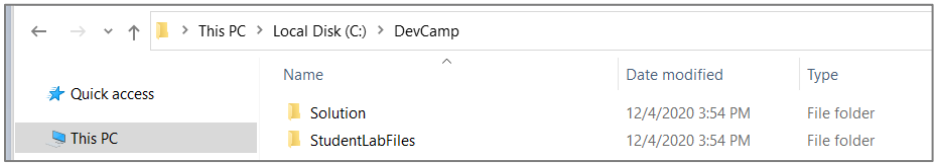
1. Download the student lab files to a local folder on your developer workstation.
   a) Create a new top-level folder on your workstation named **DevCamp** at a location such as **c:\DevCamp**.
   b) Download the ZIP archive with the student lab files from GitHub by clicking the following link.

      https://github.com/PowerBiDevCamp/DOTNET5-AppOwnsData-Tutorial/archive/master.zip

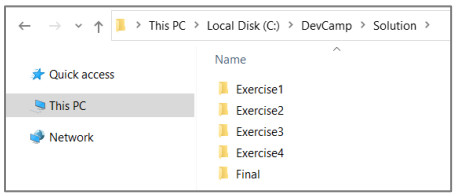
   c) From inside the ZIP file, extract the **Solutions** folder and **StudentLabFiles** folder into a local folder such as **c:\DevCamp**.



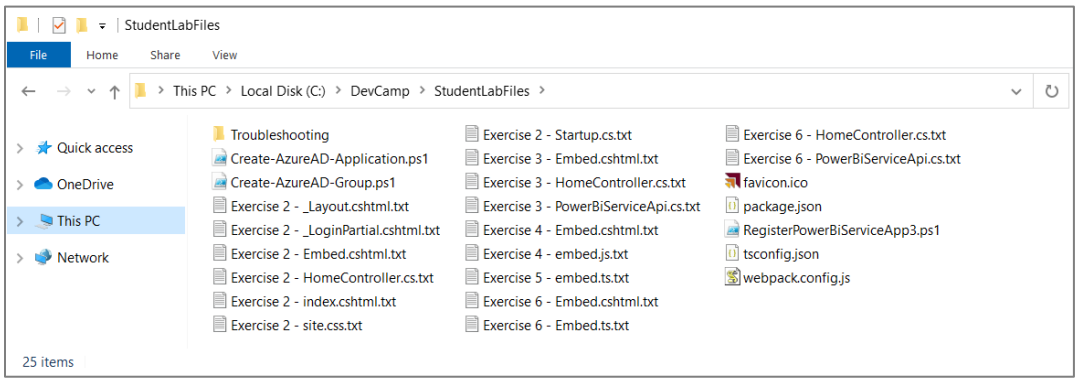
   d) The **DevCamp** folder should now contain a Solutions folder and **StudentLabFiles** folder as shown in the following screenshot.



   e) The **Solutions** folder contains child folders with the completed .NET 5 project solution for each of the exercises.



f) The **StudentLabFiles** folder contains files that you will need as you move through the exercises in this lab.



You will be required to write quite a bit code as you complete the exercises in this lab. Many of the files in the **StudentLabFiles** folder are provided to you as a convenience in case you'd like to copy-and-paste the code required during the various exercise of this lab.

2. Execute the PowerShell script named **Create-AzureAD-Group.ps1** to create a new Azure AD group named **Power BI Apps**.

a) Using Windows Explorer, look in the **StudentLabFiles** folder and locate the script named **Create-AzureAD-Group.ps1.**

b) Open **Create-AzureAD-Group.ps1** in the Windows PowerShell ISE.

c) The script begins by calling **Connect-AzureAD** to establish a connection with Azure AD.

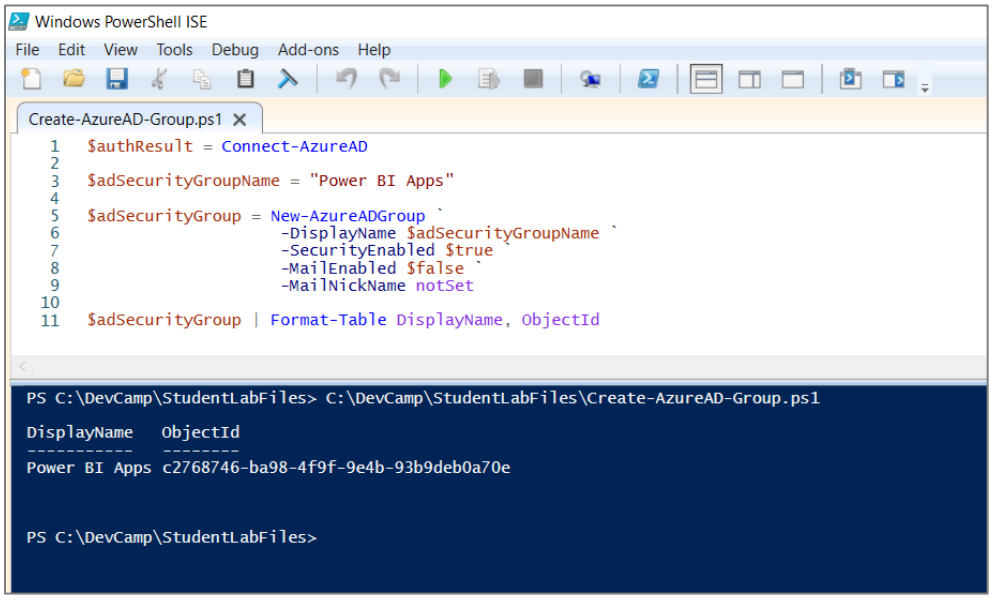
```
$authResult = Connect-AzureAD
```

d) After that, the script creates a new Azure AD security group in your Azure AD tenant named **Power BI Apps**.

```
$adSecurityGroupName = "Power BI Apps"

$adSecurityGroup = New-AzureADGroup `
                    -DisplayName $adSecurityGroupName `
                    -SecurityEnabled $true `
                    -MailEnabled $false `
                    -MailNickName notSet

$adSecurityGroup | Format-Table DisplayName, ObjectId
```

e) Execute **Create-AzureAD-Group.ps1**.using the Windows PowerShell ISE or the Windows PowerShell command line.

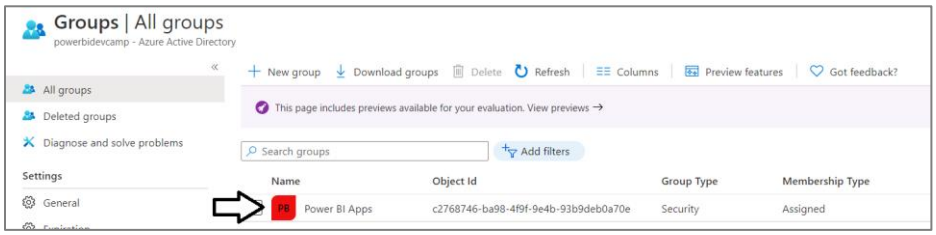
f) When prompted for credentials, log in with an Azure AD user account in the same tenant where you are using Power BI.



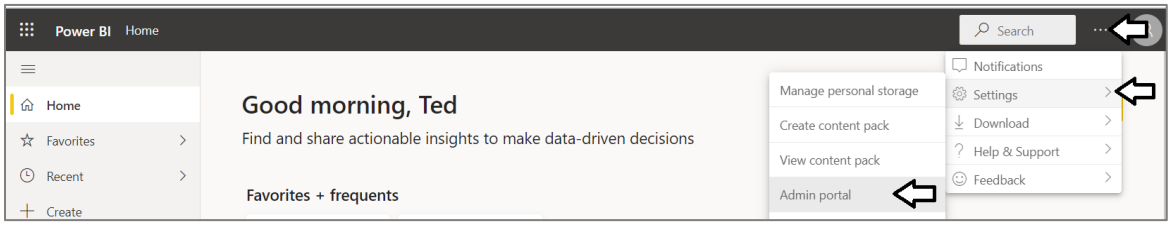
3. Use the Azure portal to verify the new Azure AD group named Power BI Apps has been created.
   a) Navigate to the **All groups** blade of the Azure AD portal using the following URL.

   ```
   https://portal.azure.com/#blade/Microsoft_AAD_IAM/GroupsManagementMenuBlade/AllGroups
   ```

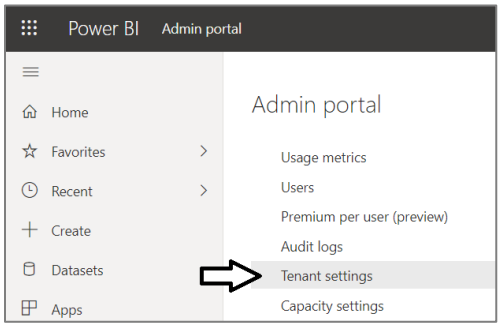
   b) Verify that you can see the new Azure AD security group named **Power BI Apps**.



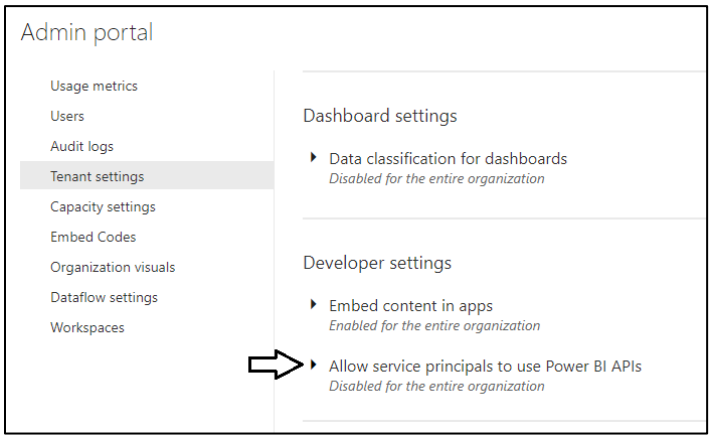
4. Enable the **Allow service principals to use Power BI APIs** setting and configure it with the **Power BI Apps** security group.
   a) Navigate to the Power BI portal at https://app.powerbi.com.
   b) Drop down the **Settings** menu and select the navigation command for the **Admin portal**.



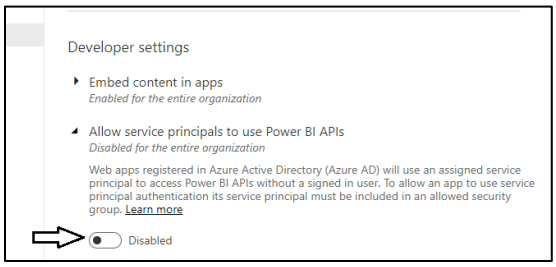
   c) In the Power BI Admin portal, click the **Tenant settings** link on the left.



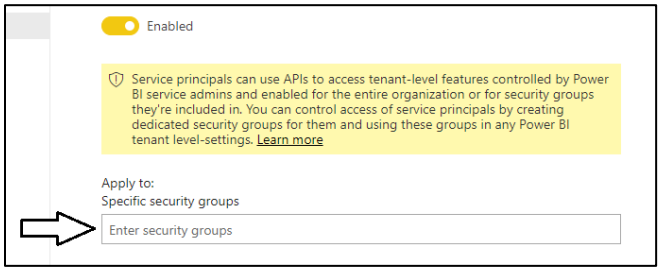
   d) Move down in the **Developer settings** section and expand the **Allow service principals to use Power BI APIs** section.



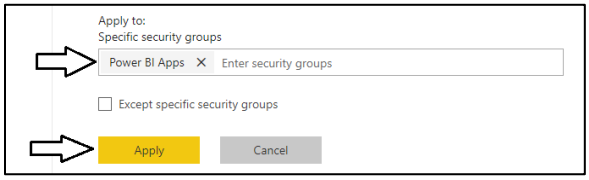
e) Note that the **Allow service principals to use Power BI APIs** setting is initially set to **Disabled**.



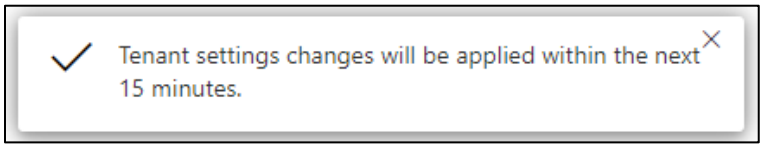
f) Change the setting to **Enabled** and place your cursor inside the **Apply to: Specific security groups** textbox.



g) Type in **Power BI Apps** to resolve the Azure AD group and then click **Apply** to save your configuration changes.



h) You will see a notification indicating it may take up to 15 minutes until your tenant recognizes your configuration changes.



The previous step is required because you must configure the Power BI Service instance in your Azure AD tenant to support API access by a service principal which is used in app-owns-data embedding. By default, any code running as a service principal is not allowed to call the Power BI Service API. In the next step you will create a new Azure AD application and add its service principal to the Azure AD group named **Power BI Apps.** This is turn will make it possible for your application to call the Power BI Service API with an app-only identity.

5. Execute the PowerShell script named **Create-AzureAD-Application.ps1**

a) Using Windows Explorer, look in the **StudentLabFiles** folder and locate the script named **Create-AzureAD-Application.ps1.**

b) Open **Create-AzureAD-Application.ps1** in a text editor such as the Windows PowerShell ISE or Notepad.

c) The script begins by calling **Connect-AzureAD** to establish a connection with Azure AD.

```
$authResult = Connect-AzureAD
```

d) The script contains two variables to set the application display name and a reply URL of **https://localhost:5001/signin-oidc**.

```
$appDisplayName = "App-Owns-Data Sample App"
$replyUrl = "https://localhost:5001/signin-oidc"
```

When you register a reply URL with **localhost** with a port number such as **5001**, Azure AD will allow you to perform testing with reply URLs that use localhost and any other port number. For example, you can use a reply URL of **https://localhost:44300/signin-oidc**.

e) The script also contains the code below which creates a new **PasswordCredential** object for an app secret.

```
# create app secret
$newGuid = New-Guid
$appSecret = ([System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes(($newGuid))))+"="
$startDate = Get-Date
$passwordCredential = New-Object -TypeName Microsoft.Open.AzureAD.Model.PasswordCredential
$passwordCredential.StartDate = $startDate
$passwordCredential.EndDate = $startDate.AddYears(1)
$passwordCredential.KeyId = $newGuid
$passwordCredential.Value = $appSecret
```

f) Down below, you can see the call to the **New-AzureADApplication** cmdlet which creates a new Azure AD application.

```
# create Azure AD Application
$aadApplication = New-AzureADApplication `
                        -DisplayName $appDisplayName `
                        -PublicClient $false `
                        -AvailableToOtherTenants $false `
                        -ReplyUrls @($replyUrl) `
                        -Homepage $replyUrl `
                        -PasswordCredentials $passwordCredential
```

g) Moving down, you should see code which determines the Object ID for the service principal of the new application..

```
# create applicaiton's service principal
$appId = $aadApplication.AppId
$appObjectId = $aadApplication.ObjectId
$serviceServicePrincipal = New-AzureADServicePrincipal -AppId $appId
$serviceServicePrincipalObjectId = $serviceServicePrincipal.ObjectId
```

h) Below that code there is code which add the service principal to the Azure AD security group named **Power BI Apps**.

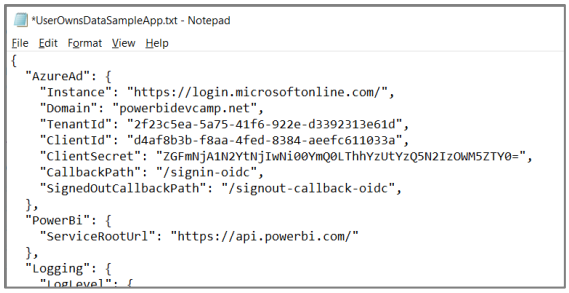
```
# add the service princiapl to Azure AD security group named 'Power BI Apps'
$adSecurityGroupName = "Power BI Apps"
$adSecurityGroup = Get-AzureADGroup -Filter "DisplayName eq '$adSecurityGroupName'"
if($adSecurityGroup) {
    # Add service principal of the new app as member of Power BI Apps group
    Add-AzureADGroupMember -ObjectId $($adSecurityGroup.ObjectId) `
                           -RefObjectId $($serviceServicePrincipalObjectId)

    # Display members of the Power BI Apps group
    Write-Host "Members of Azure AD group named $adSecurityGroupName"
    $members = Get-AzureADGroupMember -ObjectId $($adSecurityGroup.ObjectId)
    $members | Format-Table ObjectType, ObjectId, DisplayName
}
```

i) Execute **Create-Azure-ADApplication.ps1**.using the Windows PowerShell ISE or the Windows PowerShell command line.

Note you must execute this script using Windows PowerShell 5 and not PowerShell 7. This restriction is due to incompatibilities between PowerShell7 and the PowerShell module named **AzureAD**.

j) When prompted for credentials, log in with an Azure AD user account in the same tenant where you are using Power BI.

k) When the PowerShell script runs successfully, it will create and open a text file named **AppOwnsDataSampleApp.txt**.

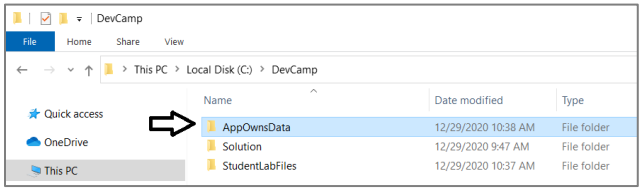


You should leave the text file **AppOwnsDataSampleApp.txt** open for now. This file contains JSON configuration data that you will copy and paste into the **appsettings.json**.file of the new .NET 5 project you will create the next exercise.

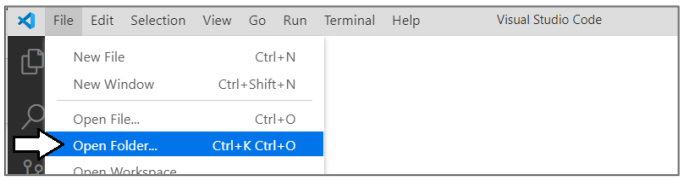
Exercise 2: Create a .NET 5 Project for a Secure Web Application

In this exercise, you will use the .NET CLI to create a new .NET 5 project for an MVC web application with support for authentication using the new Microsoft authentication library named Microsoft.Identity.Web..

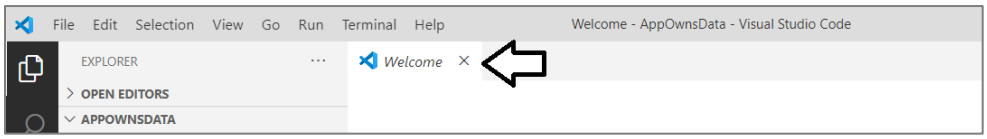
1. Create a new .NET 5 project for an ASP.NET MVC web application.
   a) Using Windows Explorer, create a child folder inside the **C:\DevCamp** folder named **AppOwnsData**.



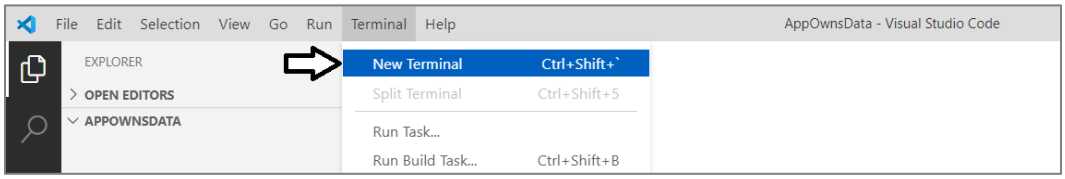
   b) Launch Visual Studio Code.
   c) Use the **Open Folder…** command in Visual Studio Code to open the **AppOwnsData** folder.



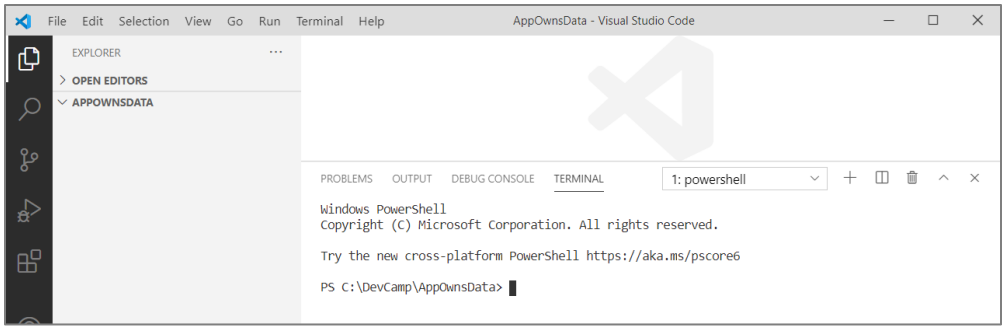
   d) Once you have open the **AppOwnsData** folder, close the **Welcome** page.



2. Use the Terminal console to verify the current version of .NET
   a) Use the **Terminal > New Terminal** command or the [**Ctrl+Shift+**`] keyboard shortcut to open the Terminal console.



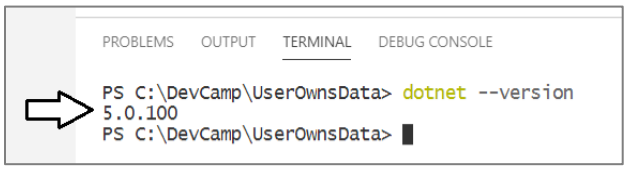
   b) You should now see a Terminal console with a cursor where you can type and execute command-line instructions.



   c) Type the following **dotnet** command-line instruction into the console and press **Enter** to execute it.

```
dotnet --version
```


d) When you run the command, the **dotnet** CLI should respond by display the .NET version number.



Make sure you .NET version number is **5.0.100** at a minimum. If you are working with .NET CORE version 3.1 or early, the project templates for creating new web applications are much different and these lab instructions will not work as expected. If you do not have .NET 5 installed, you must install the .NET 5 SDK before you can move past this point.

3. Run .NET CLI commands to register a self-signed certificate to enabled the .NET debugger to use SSL with https://localhost.
   a) In the Terminal console, type and execute the following command to generate a new .NET console application.

```
dotnet dev-certs https --trust
```

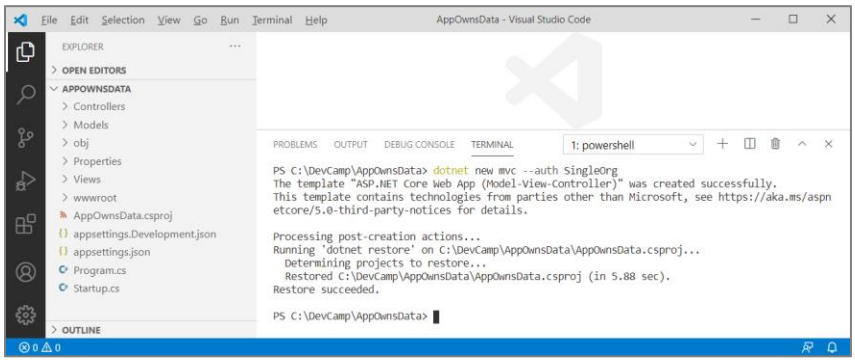
   b) Once you have run this command, you should now be able to run application in the debugger at **https://localhost:5001**.

4. Run .NET CLI commands to create a new ASP.NET MVC project.
   a) In the Terminal console, type and execute the following command to generate a new .NET console application.

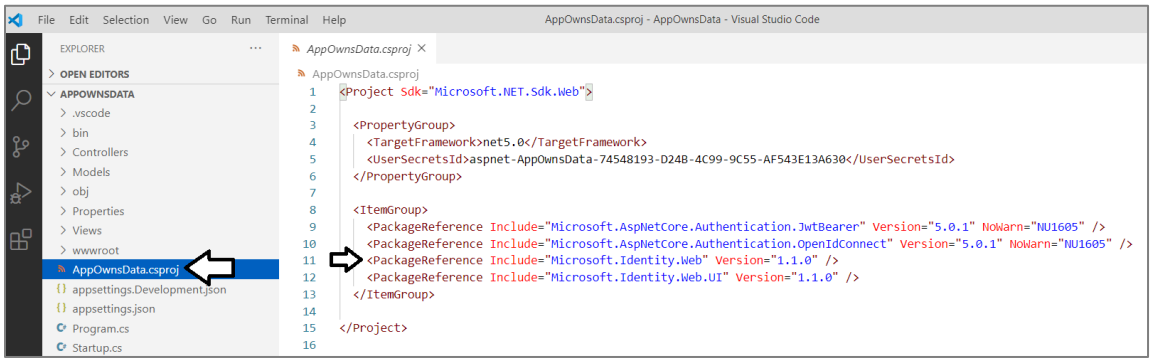
```
dotnet new mvc --auth SingleOrg
```

The **--auth SingleOrg** parameter instructs the .NET 5 CLI to generate the new web application with extra code with authentication support using Microsoft's new authentication library named Microsoft.Identity.Web.

   b) After running the **dotnet new** command, you should see that quite a few new folders and files have been added to the project.



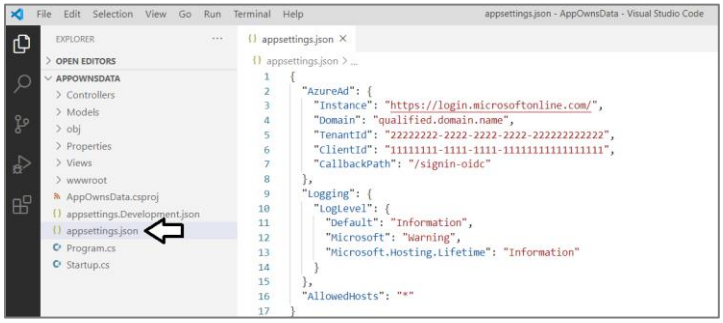
   c) Open the **AppOwnsData.csproj** file and locate the reference to **Microsoft.Idebtity.Web** and **Microsoft.Idebtity.Web.UI**.



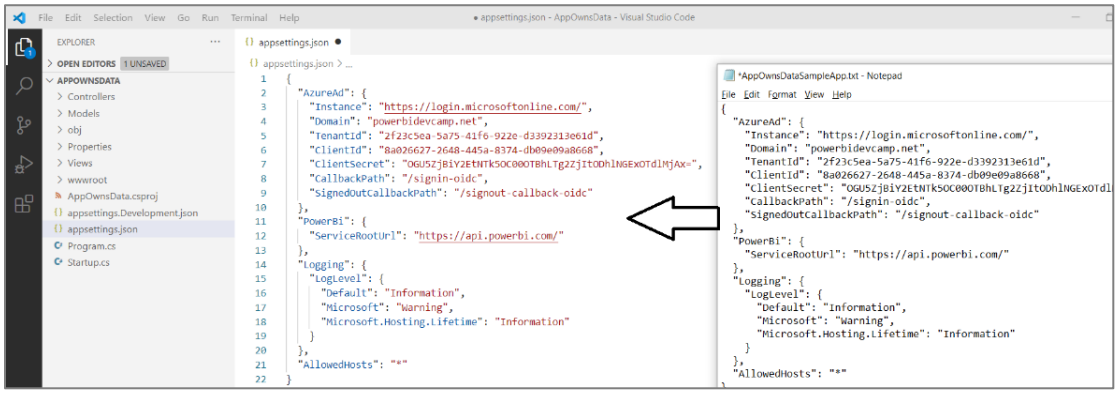
   d) Close the **AppOwnsData.csproj** without saving any changes.

Key point.Microsoft.Identity.Web has been added to project along with scaffolding code to authenticate user. All you need to do

1. Copy the JSON in **AppOwnsDataSampleApp.txt** into the **appsettings.json** file in your project.
   a) Return to the **AppOwnsData** project in Visual Studio Code and open the **appsettings.json** file.
   b) The **appsettings.json** file should initially appear like the screenshot below.

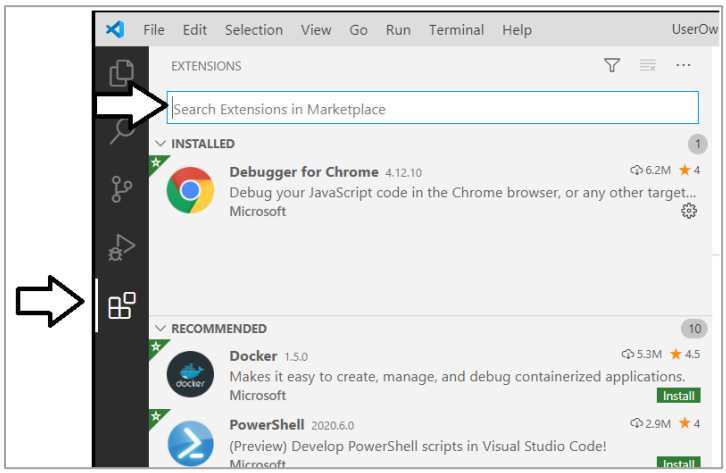


   c) Delete the contents of **appsettings.json** and replace it by copying and pasting the contents of **AppOwnsDataSampleApp.txt**

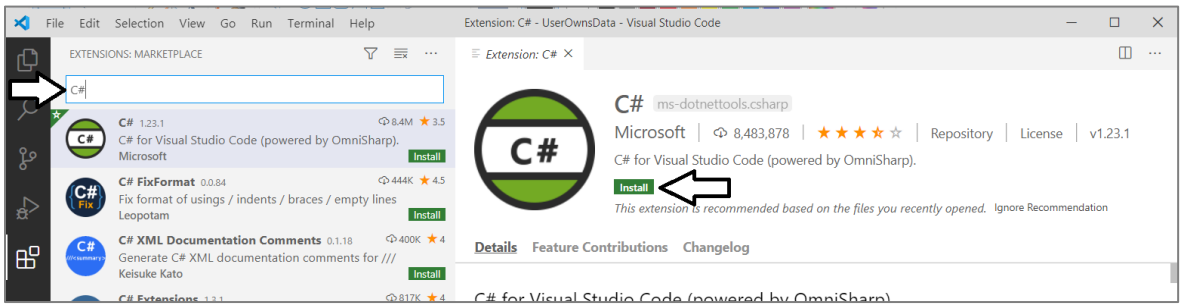


Note the **PowerBi:ServiceRootUrl** parameter has been added as a custom configuration value to track the base URL to the Power BI Service API. When programming against the Power BI Service in the Microsoft public cloud, the URL is https://api.powerbi.com/. However, the base URL for the Power BI Service API will be different in other clouds such as the government cloud. Therefore, this value will be stored as a project configuration value so it is easy to change if required. More info

2. Configure Visual Studio Code with the Omnisharp extensions needed for C# development with .NET 5.
   a) Click on the button at the bottom of the left navigation menu to display the **EXTENSION** pane.
   b) You should be able to see what extensions are currently installed.
   c) You should also be able to search to find new extensions you'd like to install.

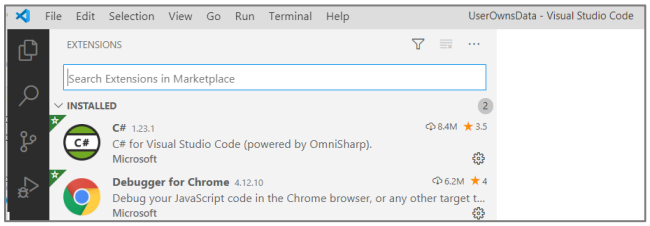


d) Find and install the **C#** extension from Microsoft if it is not installed. This extension is also known as the Omnisharp extension



e) Find and install the **Debugger for Chrome** extension from Microsoft if it is not already installed.

f) You should be able to confirm that the **C#** extension and the **Debugger for Chrome** extensions are now installed.



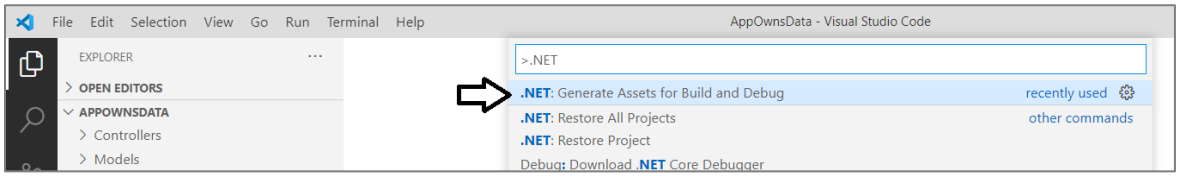
It is OK if you have other Visual Studio Code extensions installed as well. It's just important that you install these two extensions in addition to whatever other extensions you may have installed.

3. Generate the project assets required for building your project and running it in the .NET debugger.

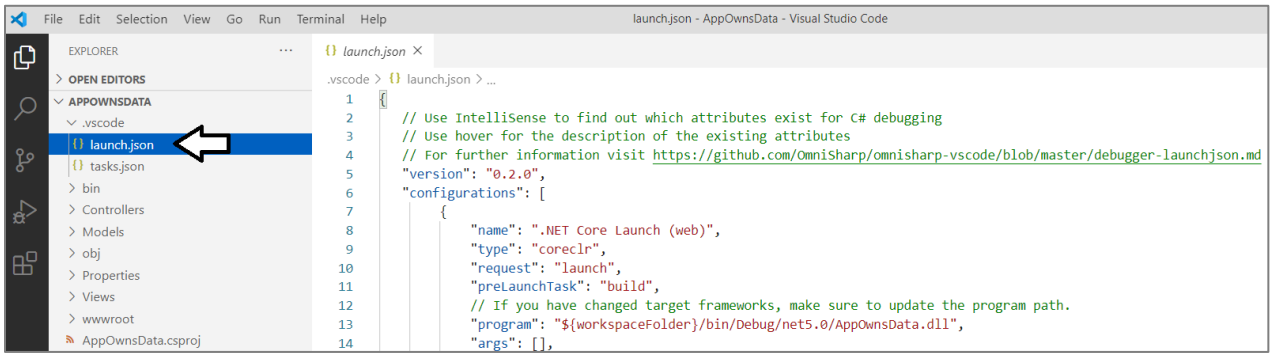
a) Open the Visual Studio Code Command Palette by using the **Ctrl**+**Shift**+**P** keyboard combination.

b) Type .NET into the Command Palette search box.

c) Select the command titled **.NET: Generate Assets for Build and Debug**.

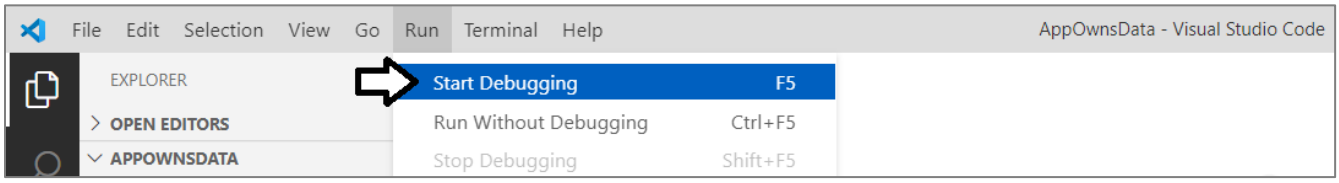


d) When the command runs successfully, in generates the files **launch.json** and **tasks.json** a new folder named **.vscode**.



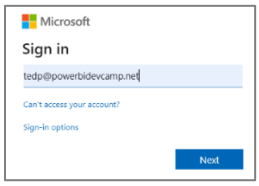
It's not uncommon for the configuration of the C# extension (i.e. Omnisharp) in Visual Studio Code to cause errors when running the command to generate the assets for build and debug. If this is the case, it can be tricky to fix this problem. If the previous step to generate build and debug assets did not successfully create the **launch.json** file and the **tasks.json** file in the **.vscode** folder, you can copy these two essential files from **Troubleshooting/.vscode** folder located inside the **StudentFiles** folder.

4. Run and test the **AppOwnsData** web application using Visual Studio Code and the .NET 5 debugger.
   a) Start the .NET 5 debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

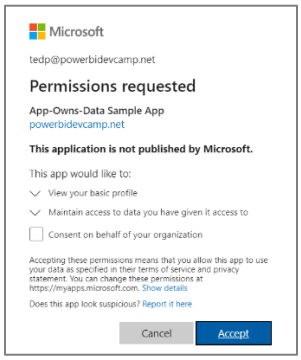


The AppOwnsData web application is currently configured authenticate the user before the user is able to view the home page. Therefore, you will be prompted to log in as soon as you launch the application in the .NET debugger.

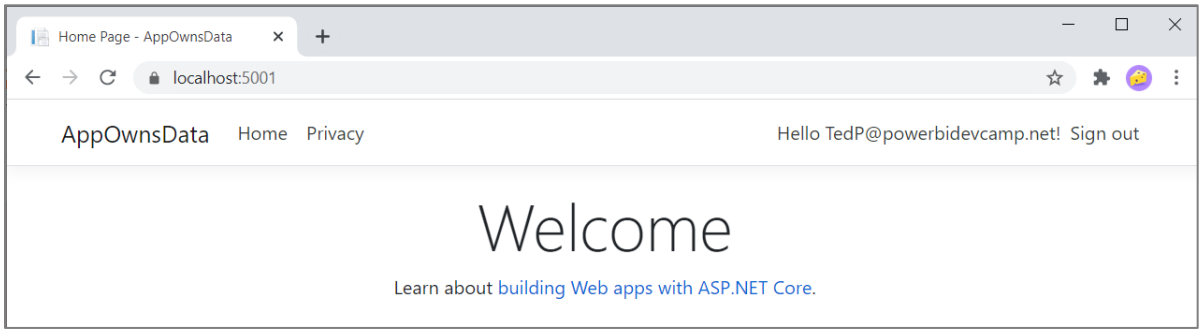
   b) When prompted to Sign in, log in using your organizational account.



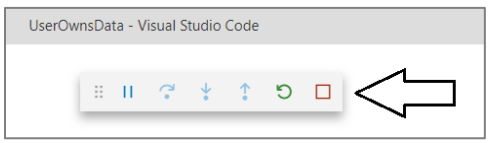
   c) Once you have signed in, you will be prompted by the **Permissions requested** dialog to grant consent to the application.
   d) Click the **Accept** button to continue.



   e) You should now see the home page for the **AppOwnsData** web application which should match the following screenshot.

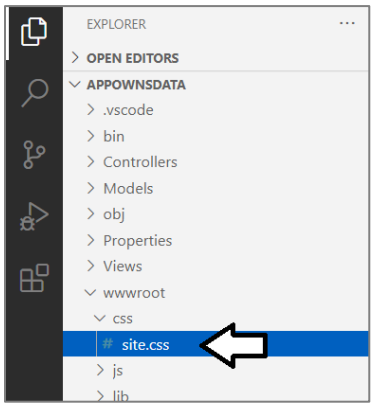


   f) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

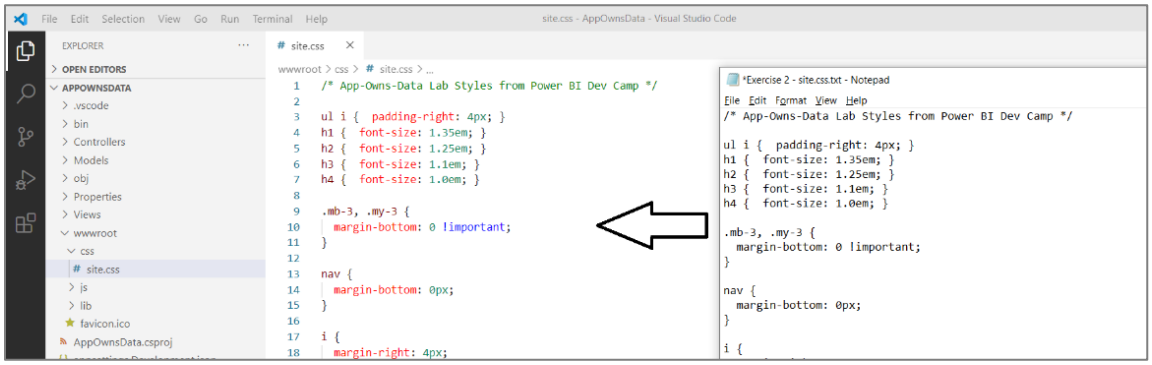


You now got to the point where the **AppOwnsData** web application is up and running and it can successfully authenticate the user. Over the next few steps you will add some custom HTML and CSS styles to make the application a bit more stylish. You will also configure the home page to allow for anonymous access in order to create a better login experience for the application's users.

5. Copy and paste a set of pre-written CSS styles into the **AppOwnsData** project's **site.css** file.
   a) Expand the **wwwroot** folder and then expand the **css** folder inside to examine the contents of the **wwwroot/css** folder.
   b) Open the CSS file named **site.css** and delete any existing content inside.

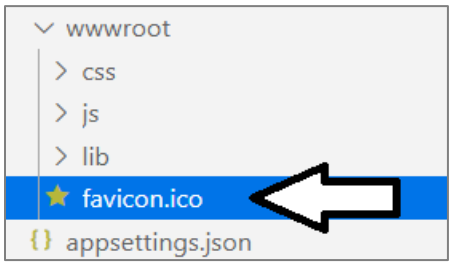


   c) Using the Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **Exercise 2 - site.css.txt**.
   d) Open **Exercise 2 - site.css.txt** up in a text editor and copy all of its contents into the Windows clipboard.
   e) Return to Visual Studio Code and paste the contents of the Windows clipboard into **sites.css**.



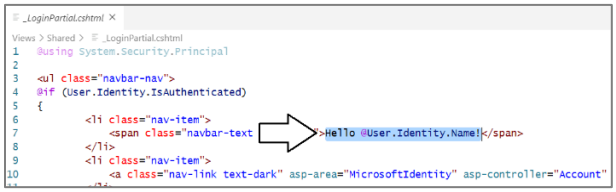
   f) Save your changes and close **site.css**.

6. Copy a custom **favicon.ico** file to the **wwwroot** folder.
   a) Using the Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **favicon.ico**.
   b) Copy the **favicon.ico** file into the **wwwroot** folder of your project.



Any file you add the **wwwroot** folder will appear at the root folder of the website created by the **AppOwnsData** project. By adding the **favicon.ico** file, this web application will now display a custom **favicon.ico** in the browser page tab.

7. Modify the partial razor view file named **_LoginPartial.cshtml** to display the user display name instead of the user principal name.
   a) Expand the **Views > Shared** folder and locate the partial view named **_LoginPartial.cshtml**.
   b) Open **_LoginPartial.cshtml** in an editor window.
   c) In the existing code, locate the code **Hello @User.Identity.Name!** which displays a message with the user principal name.



   d) Replace **Hello @User.Identity.Name!** with **Hello @User.FindFirst("name").Value** as shown in the following screenshot.

```
<li class="nav-item">
    <span class="navbar-text text-dark">Hello @User.FindFirst("name").Value</span>
</li>
```

With this update, the application will display the user's display name in the welcome message instead of using the user principal name. In other words, the user greeting will now display a message like **Hello Betty White** instead of **Hello BettyW@Contoso.com!**.

   e) Save your changes and close **_LoginPartial.cshtml**.

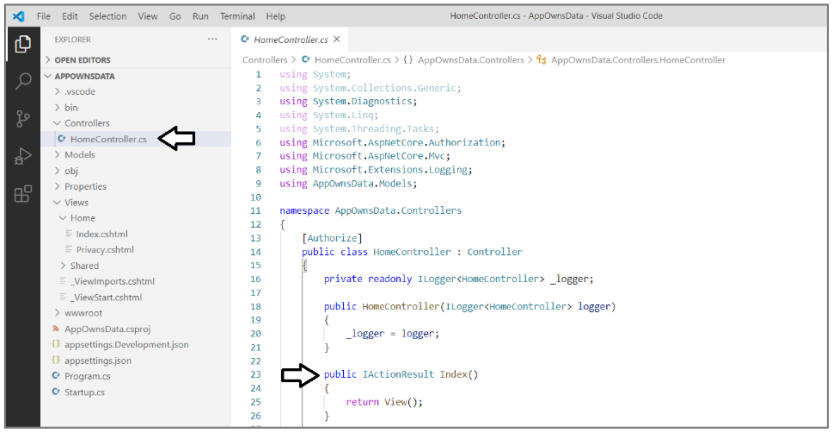
8. Modify **Index.cshtml** to display different HTML output depending on whether the user has logged in or not.
   a) Expand the **Views > Home** folder and locate the view file named **Index.cshtml**.
   b) Open **Index.cshtml** in an editor window.
   c) Delete the contents of **Index.cshtml** and replace it with the code shown in the following code listing.

```
@using System.Security.Principal

@if (User.Identity.IsAuthenticated) {
  <div class="jumbotron">
    <h2>Welcome @User.FindFirst("name").Value</h2>
    <p>You have now logged into this application.</p>
  </div>
}
else {
  <div class="jumbotron">
    <h2>Welcome to the App-Owns-Data Tutorial</h2>
    <p>Click the <strong>sign in</strong> link in the upper right to get started</p>
  </div>
}
```

   d) Once you have copied the code from above, save your changes and close **Index.cshtml**.

When you create a new .NET 5 project which supports authentication, the underlying project template creates a home page that requires authentication. To support a more natural login experience for the user, it often makes sense to configure your application so that an anonymous user can access the home page. In the next step you will modify the **Home** controller in order to make the home page accessible to anonymous users.

9. Modify the **Index** action method in **HomeController.cs** to support anonymous access.
   a) Inside the **Controllers** folder, locate **HomeControllers.cs** and open this file in an editor window.
   b) Locate the **Index** method inside the **HomeController** class.



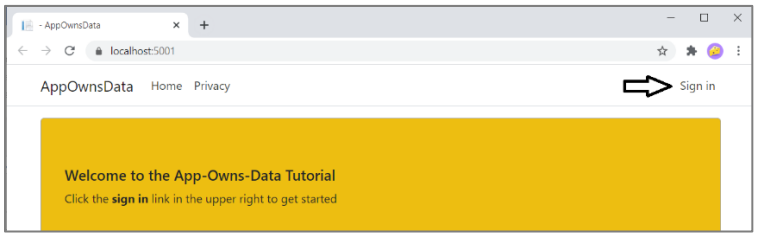
   c) Add the **[AllowAnonymous]** attribute to the **Index** method as shown in the following code listing.

```
[AllowAnonymous]
public IActionResult Index()
{
    return View();
}
```

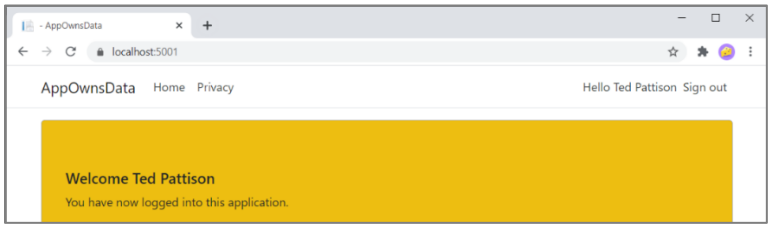
   d) Save your changes and close **HomeController.cs**.

It's time again to test this web application in the .NET debugger so you can see the effects of your changes. In the next step, you will start the debugger so you can start up the web application and test the user authentication experience in the browser.

10. Test the **AppOwnsData** project by running it in the .NET debugging environment.
   a) Start the .NET debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.
   b) Once the debugging session has initialized, the browser should display the home page using anonymous access.
   c) Click the **Sign in** link to test the user experience when authenticating with Azure AD.



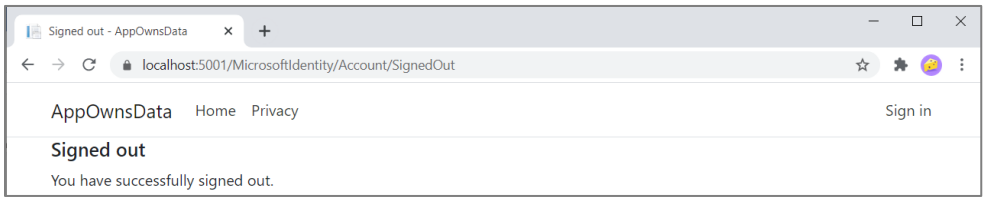
   d) Once you've signed in, you should be able to see the user display name in the welcome message in the upper right corner.



If the web page does not appear with a yellow background as shown in the screenshot above, it's possible your browser has cached the original version of the **site.css** file. If this is the case, you must clear the browser cache so it loads the latest version of **site.css**.

11. Test the user experience for logging out.
    a) Click the **Sign out** link to begin the logout experience.

    b) After logging out, you'll be directed to the **Microsoft.Identity.Web** logout page at **/MicrosoftIdentity/Account/SignedOut**.



    c) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

In the next step, you will add a new controller action and view named **Embed**. However, instead of creating a new controller action and view, you will simply the rename the controller action and view named **Privacy** that were automatically added by the project template.

12. Create a new controller action named **Embed**.
    a) Locate the **HomeController.cs** file in the **Controllers** folder and open it in an editor window.
    b) Look inside the **HomeController** class and locate the method named **Privacy**.

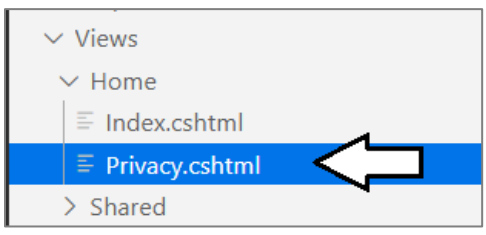
```
[AllowAnonymous]
public IActionResult Index() {
    return View();
}

public IActionResult Privacy() {
    return View();
}
```

    c) Rename of the **Privacy** method to **Embed**. No changes to the method body are required.

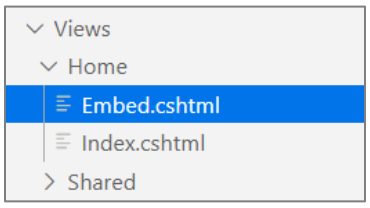
```
[AllowAnonymous]
public IActionResult Index() {
    return View();
}

public IActionResult Embed() {
    return View();
}
```

Note that, unlike the **Index** method, the **Embed** method does not have the **AllowAnonymous** attribute. That means only authenticated users will be able to navigate to this page. One really nice aspect of the ASP.NET MVC architecture is that it will automatically trigger an interactive login whenever an anonymous user attempts to navigate to a secured page such as **Embed**.

13. Create a new MVC view for the **Home** controller named **Embed.cshtml**.
    a) Look inside the **Views > Home** folder and locate the razor view file named **Privacy.cshtml**.



b) Rename the **Privacy.cshtml** razor file to **Embed.cshtml**..



c) Open **Embed.cshtml** in a code editor.

d) Delete the existing contents of **Embed.cshtml** and replace it with the follow line of HTML code.

```
<h2>TODO: Embed Report Here</h2>
```

e) Save your changes and close **Embed.cshtml**.

In a standard .NET 5 web application that uses MVC, there is a shared page layout defined in a file named **_Layouts.cshtml** which is located in the **Views > Shared** folder. In the next step you will modify the shared layout in the **_Layouts.cshtml** file so that you can add a link to the **Embed** page into the top navigation menu.

14. Modify the shared layout in **_Layout.cshtml** to include a link to the **Embed** page.

a) Inside the **Views > Shared** folder, locate **_Layouts.cshtml** and open this shared view file in an editor window.

b) Using Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **_Layout.cshtml**.

c) Open **Exercise 2 - _Layout.cshtml.txt** in the **StudentLabFiles** folder copy its contents to the Windows clipboard.

d) Return to Visual Studio Code and paste the contents of the Windows clipboard into the **_Layouts.cshtml** file.

e) Save your changes and close **_Layouts.cshtml**

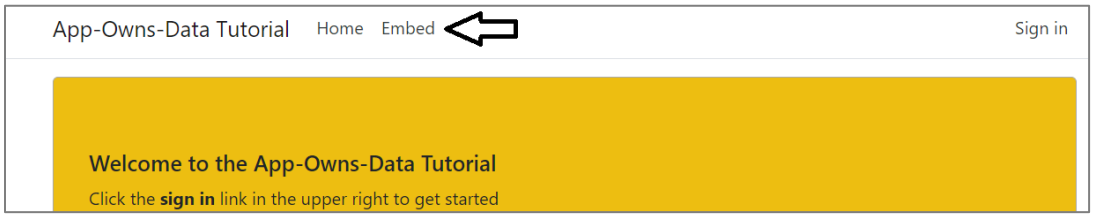
15. Run the web application in the Visual Studio Code debugger to test the new **Embed** page.

a) Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

b) The **AppOwnsData** web application should display the home page as shown to an anonymous user.

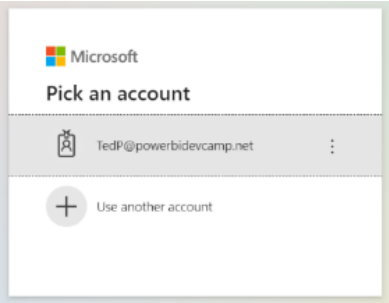
If you are automatically signed in by the application, then you should sign out and then click **Home**.

c) Click on the **Embed** link in the top nav menu to navigate to the **Embed** page.

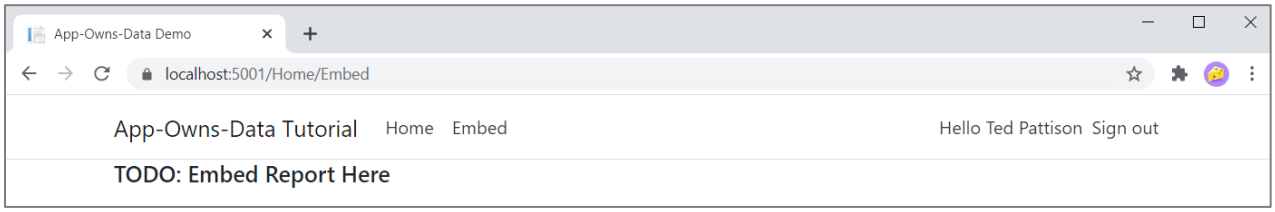


When you attempt to navigate to the **Embed** page as an anonymous user, you'll be automatically prompted to log in.

d) Log in using your user name and password.



e) Once you have logged in, you should be automatically redirected to the **Embed** page.



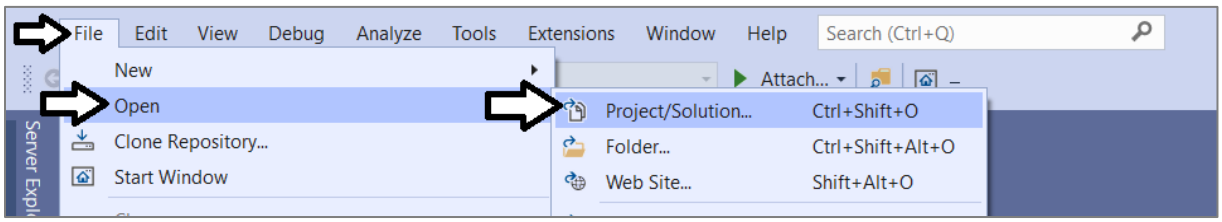
f) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

The next step is an *optional step* for those campers that prefer developing with Visual Studio 2019 instead of Visual Studio Code. If you are happy developing with Visual Studio Code and are not interested in developing .NET 5 projects using Visual Studio 2019, you can skip the next step and move ahead to *Exercise 3: Call the Power BI Service API*.

16. Open and test the **AppOwnsData** project using Visual Studio 2019.

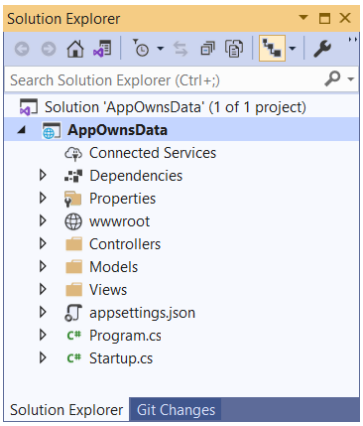
a) Launch Visual Studio 2019 – You can use any edition including the Enterprise edition, Pro edition or Community edition.

b) From the **File** menu, select the **Open > Project/Solution…** command.



c) In the **Open Project/Solution** dialog, select the **AppOwnsData.csproj** file in the **AppOwnsData** folder and click **Open**.

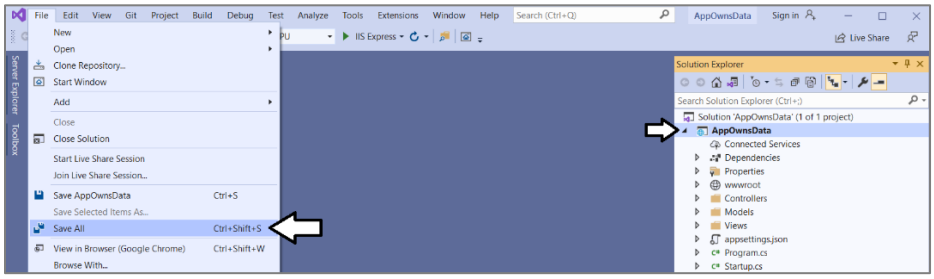
d) The **AppOwnsData** project should now be open in Visual Studio 2019 as shown in the following screenshot.



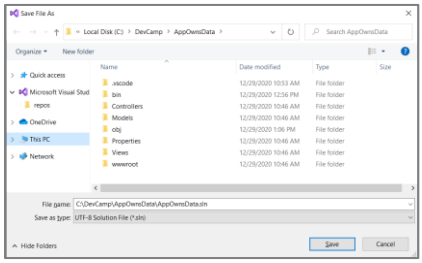
There is one big difference between developing with Visual Studio Code and Visual Studio 2019. Visual Studio Code only requires project files (*.csproj). However, Visual Studio 2019 requires that you work with both project files and solution files (*.sln). In the next step you will save a new project file for the **AppOwnsData** solution to make it easier to develop this project with Visual Studio 2019.

e) In the **Solution Explorer** on the right, select the top node for the **AppOwnsData** project.

f) From the **File** menu, select the **Save All** menu command.



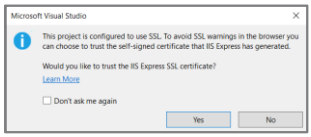
g) Save the solution file **AppOwnsData.sln** in the **AppOwnsData** project folder



17. Run the AppOwnsData application in the Visual Studio debugger.

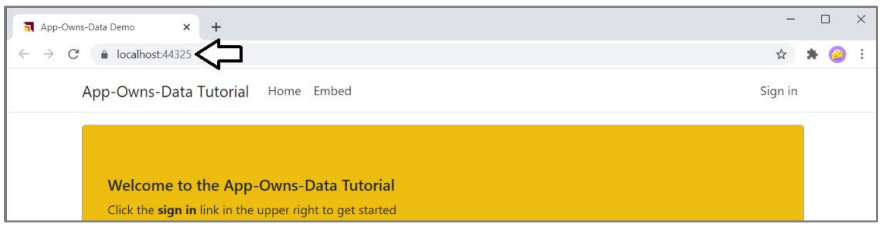
a) Press {F5} to launch the Visual Studio debugger.

b) If you are prompted with a dialog prompting you to trust an SSL certificate, click **Yes** to continue.



c) If you are prompted with a **Security Warning** dialog, click **Yes** to trust the certificate.

d) sssss



e) Test the application by signing in, signing out and clicking the **Embed** link.

f) Once you have tested the application, close the browser, return to Visual Studio and terminate the debugging session.

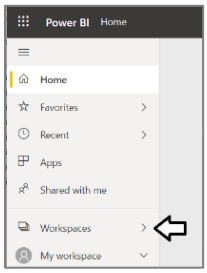
Remember that the **AppOwnsData.sln** file is only used by Visual Studio 2019 and it not used at all in Visual Studio Code.

The following lab exercises will go back to developing with Visual Studio Code. However, you can use whichever IDE you like better. Also, you can and open and test all the lab solutions using either Visual Studio Code or Visual Studio 2019.

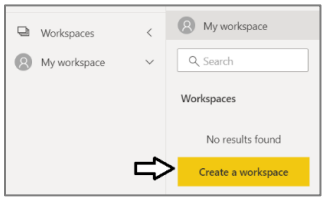
Exercise 3: Call the Power BI Service API

In this exercise, you will begin your work by creating a new Power BI workspace and importing a PBIX file to that you have a report for testing purposes. After that, you will add support to the **AppOwnsData** web application to acquire app-only access tokens from Azure AD and to call the Power BI Service API. By the end of this exercise, your code will be able to call to the Power BI Service API to retrieve data about a report required for embedding.

1. Using the browser, log into the Power BI Service in the Microsoft 365 tenant serving as your Power BI development environment.
   a) Navigate the Power BI portal at https://app.powerbi.com and if prompted, log in using your credentials.
2. Create a new app workspace named **Dev Camp Lab**.
   a) Click the **Workspace** flyout menu in the left navigation.

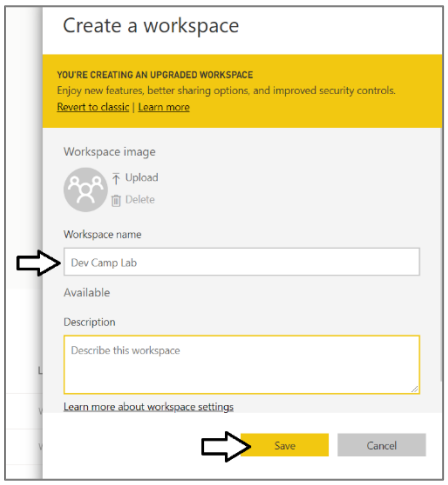


b) Click the **Create app workspace** button to display the **Create an app workspace** dialog.

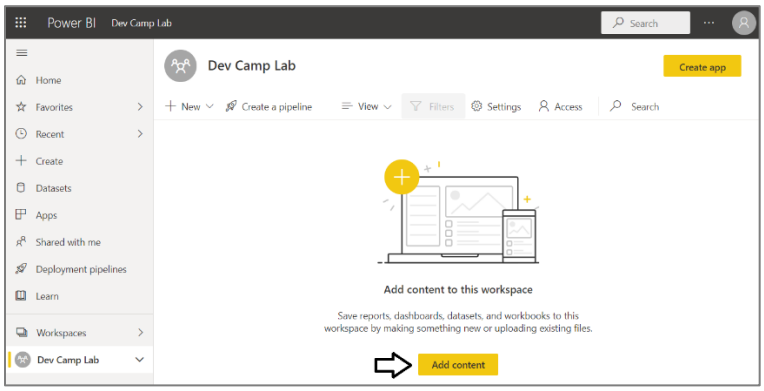


c) In the **Create an app workspace** pane, enter a workspace name such as **Dev Camp Lab**.

d) Click the **Save** button to create the new app workspace.



e) When you click **Save**, the Power BI service should create the new app workspace and then switch your current Power BI session to be running within the context of the new **Dev Camp Lab** workspace.

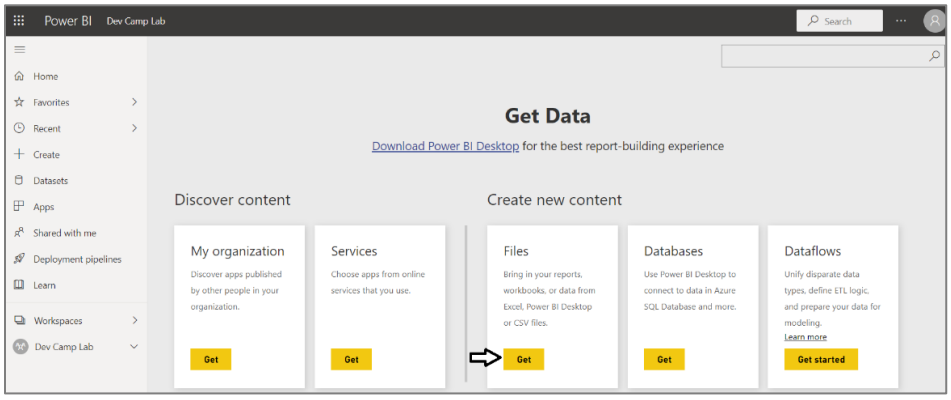


Now that you have created the new app workspace, the next step is to upload a PBIX project file created with Power BI Desktop. You are free to use your own PBIX file as long as the PBIX file does not have row-level security (RLS) enabled. If you don't have your own PBIX file, you can download the sample PBIX file named **COVID-19 US.pbix** and use that instead.

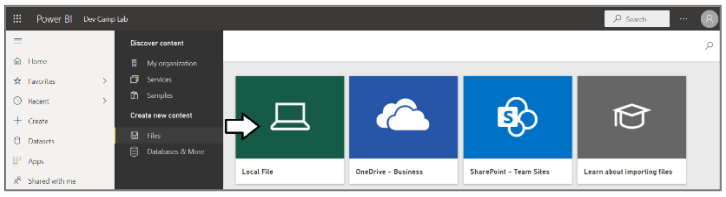
3. Upload a PBIX file to create a new report and dataset.

a) Click **Add content** to navigate to the **Get Data** page.

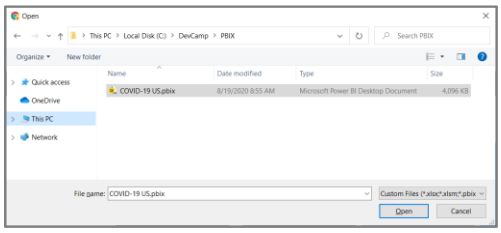
b) Click the **Get** button in the **Files** section.



c) Click on **Local File** in order to select a PBIX file that you have on your local hard drive.

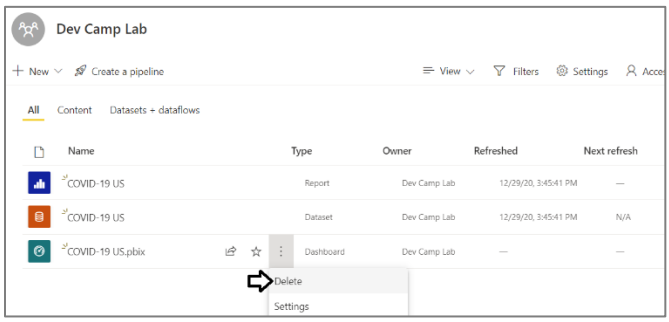


d) Select the PBIX file and click the **Open** button to upload it to the Power BI Service.



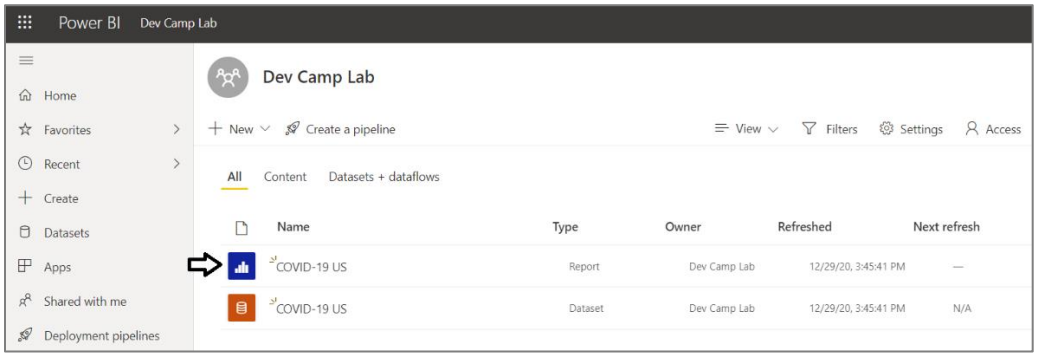
e) The Power BI Service should have created a report and a dataset from the PBIX file you uploaded.

f) If the Power BI Service created a dashboard as well, delete this dashboard as you will not need it.

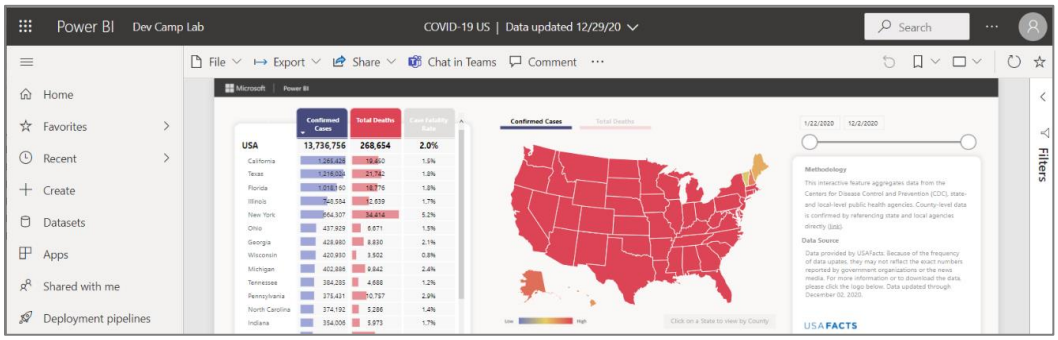


4. Open the report to see what it looks like when displayed in the Power BI Service.

a) Click on the report to open it.



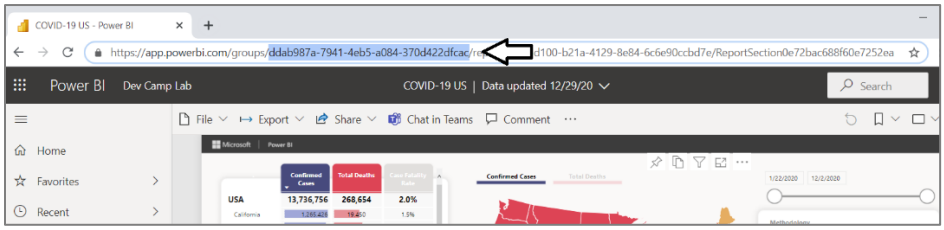
b) You should now be able to see the report.



In the next step, you will find and record the GUID-based IDs for the report and its hosting workspace. You will then use these IDs later in this exercises when you first write the code to embed a report in the **AppOwnsData** web application.

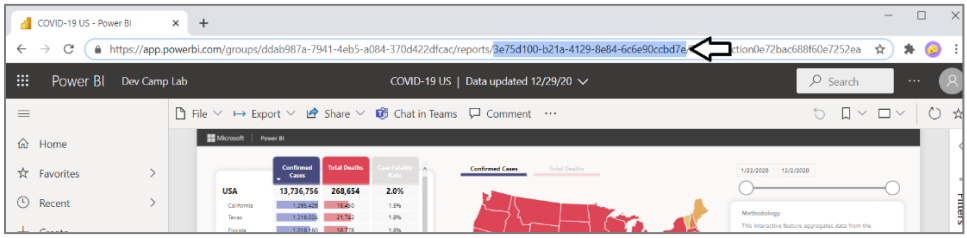
5. Get the Workspace ID and the Report ID from the report URL.

a) Locate and copy the app workspace ID from the report URL by copying the GUID that comes after ***/groups/***.

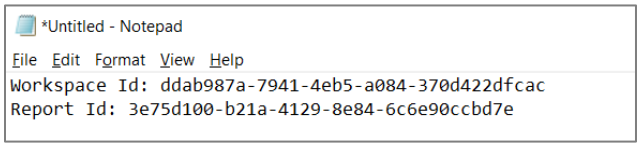


b) Open up a new text file in a program such as Notepad and paste in the value of the workspace ID.

c) Locate and copy the report ID from the URL by copying the GUID that comes after **/reports/**.



d) Copy the report ID into the text file Notepad.

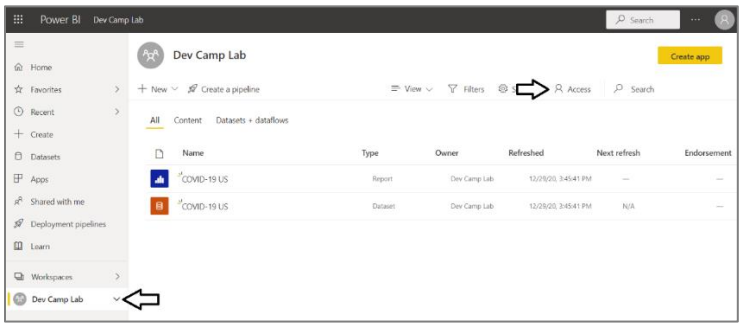


Leave the text file open for now. In a step later in this exercise, you will copy and paste these IDs into your C# code.

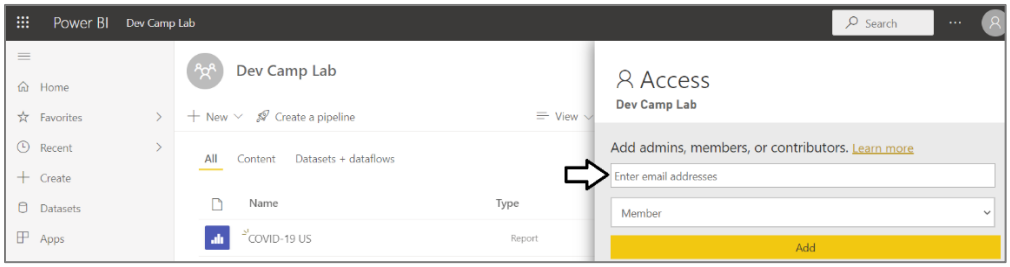
6. Add the Azure AD service principal for the **AppOwnsData** application to the **Dev Camp Lab** workspace as an administrator.

a) Click the **Dev Camp Lab** workspace in the left navigation to display the workspace summary page.

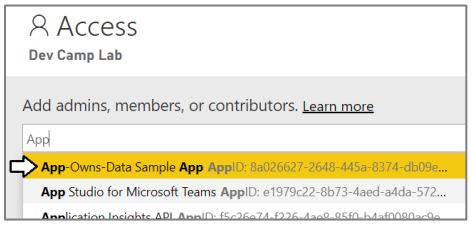
b) Click the **Access** link to open the **Access** pane where you can configure who has access to workspace resources.



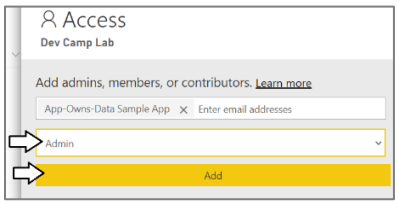
c) In the search box with the caption of **Enter email address**, type **App-Owns-Data** to find the Azure AD application.



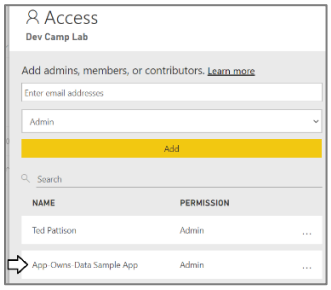
d) Select the Azure AD application named **App-Owns-Data Sample App**.



e) Select **Admin** in the dropdown menu to specify the level of access and then click the **Add** button.



f) You should now be able to confirm that the **App-Owns-Data Sample App** has been configured as a workspace admin.



g) Close the **Access** pane.

Now you will move from working with Power BI in the browser back to the AppOwnsData project in Visual Studio Code.

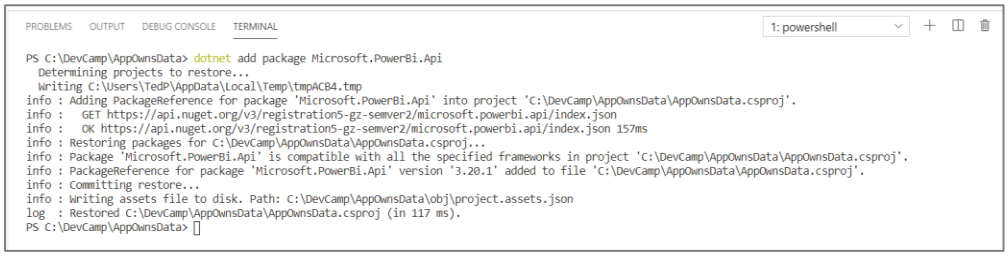
7. Add the NuGet package for the **Power BI .NET SDK**.

a) Return to the instance of Visual Studio Code where you are working on the **AppOwnsData** project.

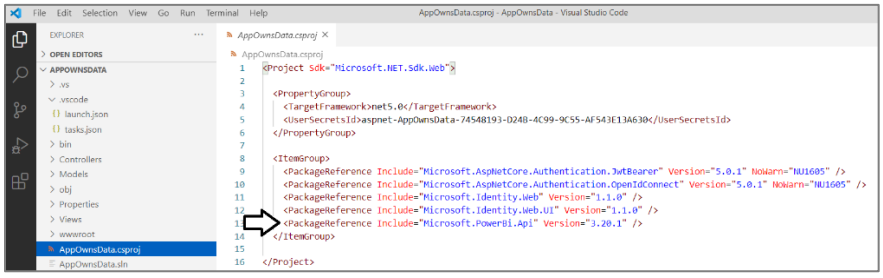
b) Move back to the Terminal so you can execute another dotnet CLI command.

c) Type and execute the following **dotnet add package** command to add the NuGet package for the **Power BI .NET SDK**.

```
dotnet add package Microsoft.PowerBi.Api
```



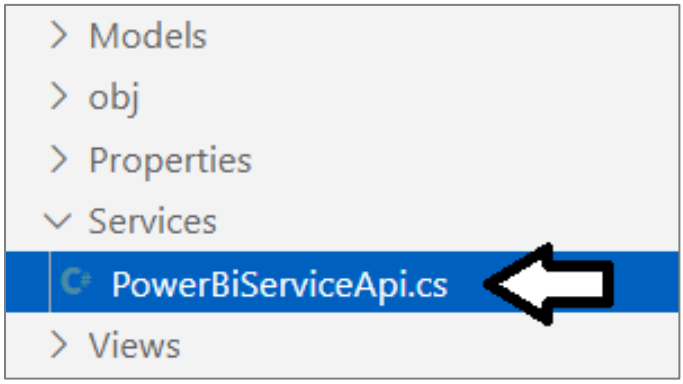
d) Open the **AppOwnsData.csproj** file. You should now see this file contains a package reference to **Microsoft.PowerBi.Api**.



e) Close the the **AppOwnsData.csproj** file without saving any changes.

Your project now includes the NuGet package for the Power BI .NET SDK so you can begin to program against the classes from this package in the **Microsoft.PowerBI.Api** namespace.

8. Create a new C# class named **PowerBiServiceApi** in which you will add code for calling the Power BI Service API.
   a) Return to the **AppOwnsData** project in Visual Studio Code.
   b) Create a new top-level folder in the **AppOwnsData** project named **Services**.
   c) Inside the **Services** folder, create a new C# source file named **PowerBiServiceApi.cs**.



   d) Copy and paste the following code into **PowerBiServiceApi.cs** to provide a starting point.

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Identity.Web;
using Microsoft.Rest;
using Microsoft.PowerBI.Api;
using Microsoft.PowerBI.Api.Models
using Newtonsoft.Json;

namespace AppOwnsData.Services {

  public class EmbeddedReportViewModel {
    //TODO: implement this class
  }

  public class PowerBiServiceApi {
    //TODO: implement this class
  }

}
```

   e) Implement the **EmbeddedReportViewModel** class using the following code.

```
public class EmbeddedReportViewModel {
  public string Id;
  public string Name;
  public string EmbedUrl;
  public string Token;
}
```

The **EmbeddedReportViewModel** class is designed as a view model which defines the structure the embedding data required for a Power BI report. You will use this class later in this lab to pass embedding data for a report from an MVC controller to an MVC view.

   f) Begin your implementation by adding two private fields named **tokenAcquisition** and **urlPowerBiServiceApiRoot**.

```
public class PowerBiServiceApi {

  private ITokenAcquisition tokenAcquisition { get; }
  private string urlPowerBiServiceApiRoot { get; }

}
```


g) Add the following constructor to initialize the two private fields named **tokenAcquisition** and **urlPowerBiServiceApiRoot**.

```
public class PowerBiServiceApi {

  private ITokenAcquisition tokenAcquisition { get; }
  private string urlPowerBiServiceApiRoot { get; }

  public PowerBiServiceApi(IConfiguration configuration, ITokenAcquisition tokenAcquisition) {
    this.urlPowerBiServiceApiRoot = configuration["PowerBi:ServiceRootUrl"];
    this.tokenAcquisition = tokenAcquisition;
  }

}
```

This code uses the .NET dependency injection (DI) pattern. When your class needs to use a service, you can simply add a constructor parameter based on the type for that service and the .NET runtime takes care of passing the service instance at run time. In this case, the constructor is injecting an instance of the .NET configuration service using the **IConfiguration** parameter which is used to retrieve the **PowerBi:ServiceRootUrl** configuration value from **appsettings.json**. The **ITokenAcquisition** parameter which is named **tokenAcquisition** holds a reference to the Microsoft authentication service provided by the **Microsoft.Identity.Web** library and will be used to acquire access tokens from Azure AD.

h) Place your cursor at the bottom of the **PowerBiServiceApi** class and add another new line so you can add more members.

i) At the bottom off the **PowerBiServiceApi** class, add the following static read-only field named **RequiredScopes**.

```
public const string powerbiApiDefaultScope = "https://analysis.windows.net/powerbi/api/.default";
```

The **powerbiApiDefaultScope** constant is a string with the default permissions scope for the Power BI Service API. Your application will pass this permission scope when it calls to Azure AD to acquire an app-only access token.

j) Move down in the **PowerBiServiceApi** class below the **RequiredScopes** field and add the **GetAccessToken** method.

```
public string GetAccessToken() {
  return this.tokenAcquisition.GetAccessTokenForAppAsync(powerbiApiDefaultScope).Result;
}
```

k) Move down below the **GetAccessToken** method and add the **GetPowerBiClient** method.

```
public PowerBIClient GetPowerBiClient()     {
  var tokenCredentials = new TokenCredentials(GetAccessToken(), "Bearer");
  return new PowerBIClient(new Uri(urlPowerBiServiceApiRoot), tokenCredentials);
}
```

l) Move down below the **GetPowerBiClient** method and add the **GetReport** method.

```
public async Task<EmbeddedReportViewModel> GetReport(Guid WorkspaceId, Guid ReportId) {

    PowerBIClient pbiClient = GetPowerBiClient();

    // call to Power BI Service API to get embedding data
    var report = await pbiClient.Reports.GetReportInGroupAsync(WorkspaceId, ReportId);

    // generate read-only embed token for the report
    var datasetId = report.DatasetId;
    var tokenRequest = new GenerateTokenRequest(TokenAccessLevel.View, datasetId);
    var embedTokenResponse =
                    await pbiClient.Reports.GenerateTokenAsync(WorkspaceId, ReportId, tokenRequest);
    var embedToken = embedTokenResponse.Token;

    // return report embedding data to caller
    return new EmbeddedReportViewModel {
        Id = report.Id.ToString(),
        EmbedUrl = report.EmbedUrl,
        Name = report.Name,
        Token = embedToken
    };
}
```

m) Save and close **PowerBIServiceApi.cs**.

Note that **Exercise 3 - PowerBiServiceApi.cs.txt** in the **StudentLabFiles** folder contains the final code for **PowerBiServiceApi.cs**.

9. Modify the code in **Startup.cs** to properly register the services required for user authentication and access token acquisition.
   a) Open the **Startup.cs** file in an editor window.
   b) Underneath the existing **using** statements, add the following **using** statement;

```
using AppOwnsData.Services;
```

   c) Look inside the **ConfigureServices** method and locate the following line of code.

```
public void ConfigureServices(IServiceCollection services) {

    services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
        .AddMicrosoftIdentityWebApp(Configuration.GetSection("AzureAd"));
```

   d) Remove the code that calls **AddAuthentication** and **AddMicrosoftIdentityWebApp** on the services object.
   e) Add the following code to the top of the **ConfigureServices** method.

```
public void ConfigureServices(IServiceCollection services) {

  services
   .AddMicrosoftIdentityWebAppAuthentication(Configuration)
   .EnableTokenAcquisitionToCallDownstreamApi()
   .AddInMemoryTokenCaches();
```

   f) Move below the call to **AddInMemoryTokenCaches** and add the following code.

```
services.AddScoped(typeof(PowerBiServiceApi));
```

   g) At this point, the **ConfigureService** method in **Startup.cs** should match the following code listing.

```
public void ConfigureServices(IServiceCollection services) {

    services
        .AddMicrosoftIdentityWebAppAuthentication(Configuration)
        .EnableTokenAcquisitionToCallDownstreamApi()
        .AddInMemoryTokenCaches();

    services.AddScoped(typeof(PowerBiServiceApi));

    services.AddControllersWithViews(options => {
        var policy = new AuthorizationPolicyBuilder()
            .RequireAuthenticatedUser()
            .Build();
        options.Filters.Add(new AuthorizeFilter(policy));
    });
    services.AddRazorPages()
            .AddMicrosoftIdentityUI();
}
```

The code in **ConfigureServices** accomplishes several important things. The call to **AddMicrosoftWebAppCallsWebApi** configures the Microsoft authentication library to acquire access tokens. Next, the call to **AddInMemoryTokenCaches** configures a token cache that the Microsoft authentication library will use to cache app-only access tokens behind the scenes. Finally, the call to **services.AddScoped(typeof(PowerBiServiceApi))** configures the **PowerBiServiceApi** class as a service class that can be added to other classes using the dependency injection (DI) pattern.

10. Modify the **HomeController** class to program against the **PowerBiServiceApi** class.
   a) Inside the **Controllers** folder, locate **HomeController.cs** and open it in an editor window.
   b) Underneath the existing **using** statements, add a **using** statement to import the **AppOwnsData.Services** namespace.

```
using AppOwnsData.Services;
```


c) At the top of the **HomeController** class locate the **_logger** field and the constructor as shown in the following code listing.

```
[Authorize]
public class HomeController : Controller {

  private readonly ILogger<HomeController> _logger;

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

d) Remove the **_logger** field and the existing constructor and replace them with the following code.

```
[Authorize]
public class HomeController : Controller {

  private PowerBiServiceApi powerBiServiceApi;

  public HomeController(PowerBiServiceApi powerBiServiceApi) {
    this.powerBiServiceApi = powerBiServiceApi;
  }
```

This is another example of using dependency injection. Since you registered the **PowerBiServiceApi** class as a service by calling **services.AddScoped** in the **ConfigureServices** method, you can simply add a **PowerBiServiceApi** parameter to the constructor and the .NET 5 runtime will take care of creating a **PowerBiServiceApi** instance and passing it to the constructor.

e) Locate the **Embed** method implementation in the **HomeController** class and replace it with the following code.

```
public async Task<IActionResult> Embed() {

  // replace these two GUIDs with the workspace ID and report ID you recorded earlier
  Guid workspaceId = new Guid("11111111-1111-1111-1111-111111111111");
  Guid reportId = new Guid("22222222-2222-2222-2222-222222222222");

  var viewModel = await powerBiServiceApi.GetReport(workspaceId, reportId);

  return View(viewModel);
}
```

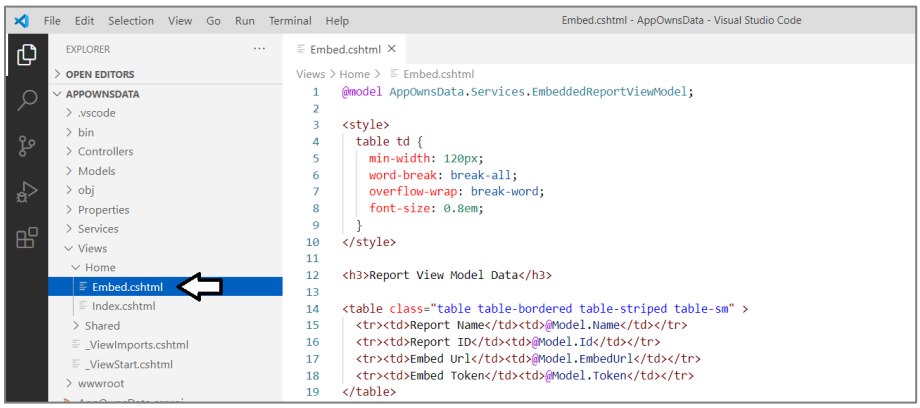
11. Modify the HTML and razor code in the view file named **Embed.cshtml**.

a) Locate the **Embed.cshtml** razor file inside the **Views > Home** folder and open this file in an editor window.

b) Delete the contents of **Embed.cshtml** and replace it with the following code which creates a table to display report data.

```
@model AppOwnsData.Services.EmbeddedReportViewModel;

<style>
  table td {
    min-width: 120px;
    word-break: break-all;
    overflow-wrap: break-word;
    font-size: 0.8em;
  }
</style>

<h3>Report View Model Data</h3>

<table class="table table-bordered table-striped table-sm" >
  <tr><td>Report Name</td><td>@Model.Name</td></tr>
  <tr><td>Report ID</td><td>@Model.Id</td></tr>
  <tr><td>Embed Url</td><td>@Model.EmbedUrl</td></tr>
  <tr><td>Embed Token</td><td>@Model.Token</td></tr>
</table>
```


c) The code in **Embed.cshtml** should now match the following screenshot..



d) Save your changes and close **Embed.cshtml**.

12. Run the web application in the Visual Studio Code debugger to test the new **Embed** page.

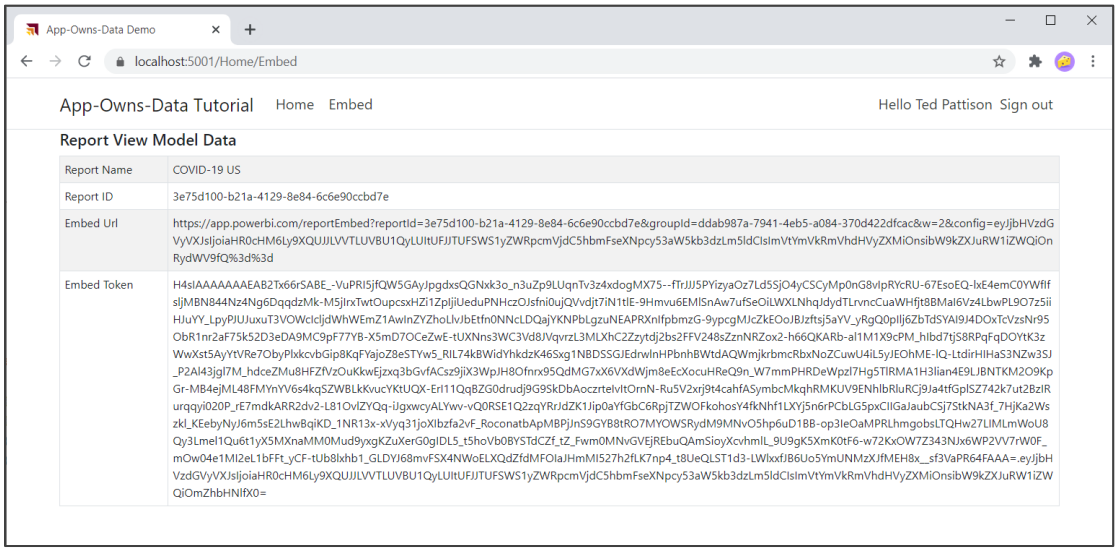
a) Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

b) The **AppOwnsData** web application should display the home page as shown to an anonymous user.

c) Click on the **Embed** link in the top nav menu to navigate to the **Embed** page.

d) If you are prompted to enter your credentials, enter your user name and password and log in.

e) Once you navigate to the **Embed** page, it should display a table containing the embedding data for your Power BI report.



f) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

You are now half way in your development efforts to embed a Power BI report. You have written the server-side code to call the Power BI Service API and retrieve the data required to embed a report. In the next exercise, you will complete the Power BI embedding implementation by adding client-side JavaScript code which programs against the Power BI JavaScript API.

Exercise 4: Embed a Report using powerbi.js

In this exercise, you will modify the view named **Embed.cshtml** to embed a Power BI report on a web page. Your work will involve adding a new a JavaScript file named **embed.js** in which you will write the minimal client-side code required to embed a report.

1. Modify the razor view file named **Embed.cshtml**.
   a) Inside the **Views > Home** folder, locate and open **Embed.cshtml** in an editor window.
   b) Replace the contents of **Embed.cshtml** with the following code.

```
@model AppOwnsData.Services.EmbeddedReportViewModel;

<div id="embed-container" style="height:800px;"></div>

@section Scripts {

}
```

Note that the div element with the ID of **embed-container** will be used as the embed container.

Over the next few steps, you will add three **script** tags into the **Scrips** section. The benefit of adding script tags into the **Scripts** section is that they will load after the JavaScript libraries such as jquery which are loaded from the shared view **_Layout.cshtml**.

   c) Place your cursor inside the **Scripts** section and paste in the following **script** tag to import **powerbi.min.js** from a CDN.

```
<script src="https://cdn.jsdelivr.net/npm/powerbi-client@2.13.3/dist/powerbi.min.js"></script>
```

powerbi.min.js is the JavaScript file that loads the client-side library named the **Power BI JavaScript API**.

   d) Underneath the **script** tag for the Power BI JavaScript API, add a second **script** tag using the following code.

```
<script>
  var viewModel = {
    reportId: "@Model.Id",
    embedUrl: "@Model.EmbedUrl",
    token: "@Model.Token"
  };
</script>
```

This **script** tag is creates a JavaScript object named **viewModel** which is accessible to the JavaScript code you'll write later in this lab.

   e) Underneath the other two **script** tags, add a third **script** tag to load a custom JavaScript file named **embed.js**.

```
<script src="~/js/embed.js"></script>
```

Note that the JavaScript file named **embed.js** does not exist yet. You will create the **embed.js** file in the next step.

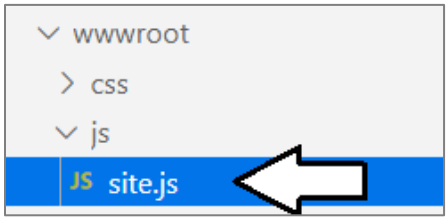
   f) When you are done, the contents you have in **Embed.cshtml** should match the following code listing.

```
@model AppOwnsData.Services.EmbeddedReportViewModel;

<div id="embed-container" style="height:800px;"></div>

@section Scripts {
  <script src="https://cdn.jsdelivr.net/npm/powerbi-client@2.13.3/dist/powerbi.min.js"></script>
  <script>
    var viewModel = {
      reportId: "@Model.Id",
      embedUrl: "@Model.EmbedUrl",
      token: "@Model.Token"
    };
  </script>
  <script src="~/js/embed.js"></script>
}
```

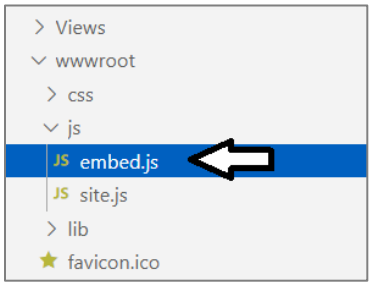
   g) Save your changes and close **Embed.cshtml**.

The final step is to add a new JavaScript file named **embed.js** with the code required to embed a report.

2. Add a new JavaScript file named **embed.js**.
   a) Locate the top-level folder named **wwwroot** and expand it.
   b) Locate the **js** folder inside the **wwwroot** folder and expand that.
   c) Currently, there should be one file inside the **wwwroot > js** folder named **site.js**.



   d) Add a new JavaScript file named **embed.js**.



3. Add the JavaScript code to **embed.js** to embed a report.
   a) Open **embed.js** in an editor window.
   b) Delete whatever content exists inside **embed.js**.
   c) Paste the following code into **embed.js** to provide a starting point.

```
$(function(){

  // 1 - get DOM object for div that is report container

  // 2 - get report embedding data from view model

  // 3 - embed report using the Power BI JavaScript API.

  // 4 - add logic to resize embed container on window resize event

});
```

You will now copy and paste four sections of JavaScript code into **embed.js** to complete the implementation. Note that you can copy and paste all the code at once by copying the contents of **Exercise 4 - embed.js.txt** in the **StudentLabFiles** folder.

   d) Add the following JavaScript code to create a variable named **reportContainer** which holds a reference to **embed-container**.

```
// 1 - get DOM object for div that is report container
var reportContainer = document.getElementById("embed-container");
```

   e) Add code to create 3 variables named **reportId**, **embedUrl** and **token** which are initialized from the global **viewModel** object.

```
// 2 - get report embedding data from view model
var reportId = window.viewModel.reportId;
var embedUrl = window.viewModel.embedUrl;
var token = window.viewModel.token
```

Now this JavaScript code has retrieved the three essential pieces of data from **window.viewModel** to embed a Power BI report.

f) Add the following code to embed a report by calling the **powerbi.embed** function provided by the Power BI JavaScript API.

```
// 3 - embed report using the Power BI JavaScript API.
var models = window['powerbi-client'].models;

var config = {
  type: 'report',
  id: reportId,
  embedUrl: embedUrl,
  accessToken: token,
  permissions: models.Permissions.All,
  tokenType: models.TokenType.Embed,
  viewMode: models.ViewMode.View,
  settings: {
    panes: {
      filters: { expanded: false, visible: true },
      pageNavigation: { visible: false }
    }
  }
};

// Embed the report and display it within the div container.
var report = powerbi.embed(reportContainer, config);
```

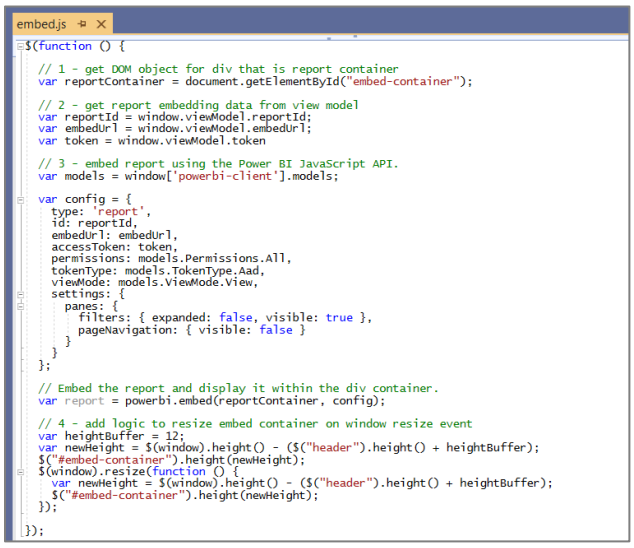
Note that the variable named **models** is initialized using a call to **window['powerbi-client'].models**. The **models** variable is used to set configuration values such as **models.Permissions.All**, **models.TokenType.Aad** and **models.ViewMode.View**.

A key point is that you need to create a configuration object in order to call the **powerbi.embed** function. You can learn a great deal more about creating the configuration object for Power BI embedding in this wiki for the Power BI JavaScript API.

g) Add the following JavaScript code to resize the **embed-container** div element whenever the window resize event fires.

```
// 4 - add logic to resize embed container on window resize event
var heightBuffer = 12;
var newHeight = $(window).height() - ($("header").height() + heightBuffer);
$("#embed-container").height(newHeight);
$(window).resize(function () {
  var newHeight = $(window).height() - ($("header").height() + heightBuffer);
  $("#embed-container").height(newHeight);
});
```

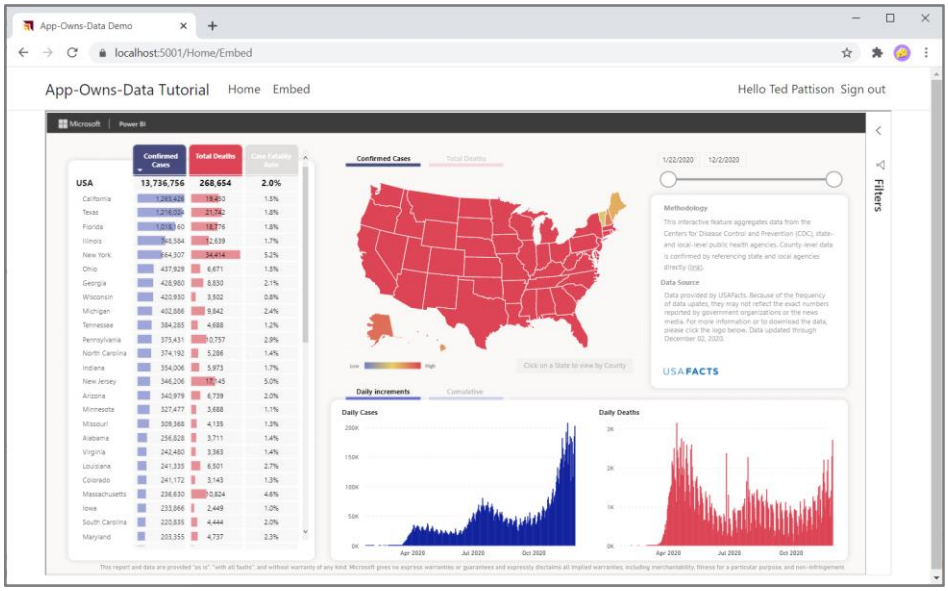
h) Your code in **embed.js** should match the following screenshot.



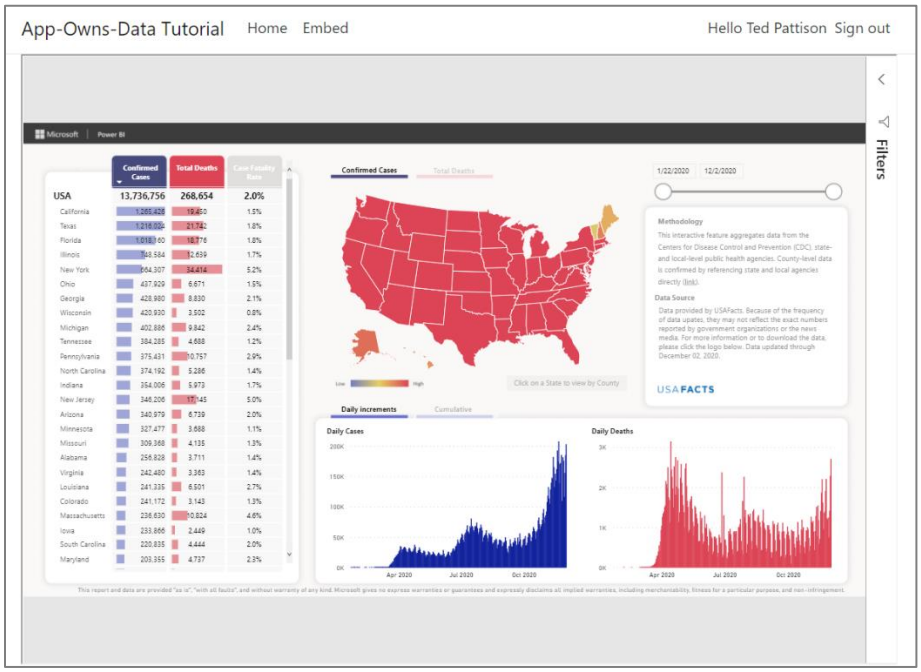
Remember you can copy and paste all the code at once by using the text in **Exercise 4 - embed.js.txt** in the **StudentLabFiles** folder.

i) Save your changes and close **embed.js**.

4. Run the web application in the Visual Studio Code debugger to test your work on the **Embed** page.
   a) Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.
   b) The **AppOwnsData** web application should display the home page as shown to an anonymous user.
   c) Click on the **Embed** link in the top nav menu to navigate to the **Embed** page and login when prompted.
   d) You should now be able to navigate to the **Embed** page and see the Power BI report displayed on the page.



   e) Try resizing the browser window. The embedded report should continually adapt to the size of the window.



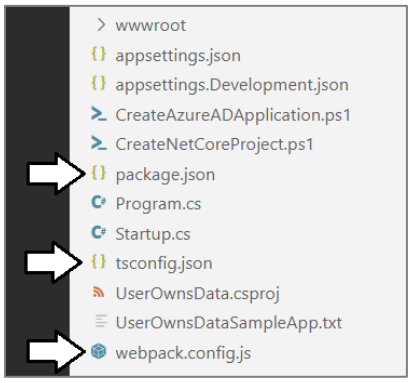
   f) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

You have now reached an important milestone. You can now tell all your peers that you have embedded a Power BI report. However, there is more that you can do to improve the developer experience for writing client-side code against the Power BI JavaScript API. In the next exercise, you will add support to your project so that you can program client-side code using TypeScript instead of JavaScript. By moving to TypeScript you can benefit from strongly-typed programming, compile-time type checking and much better IntelliSense.

Exercise 5: Add TypeScript Support to a .NET 5 Project

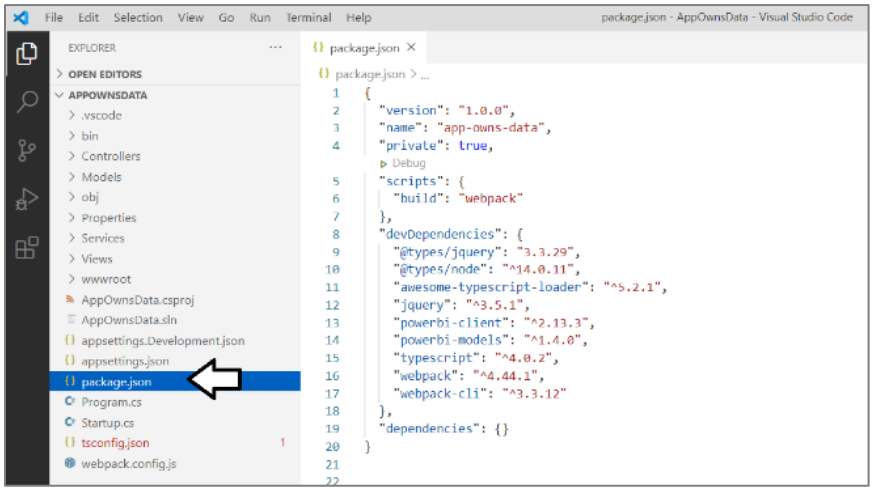
In this exercise, you will add support for developing your client-side code with TypeScript instead of JavaScript. It is assumed that you have already installed Node.js so that the Node Package Manager command-line tool (**npm)** is available at the commend line. You will begin by adding several Node.js configuration files to the root folder of the **AppOwnsData** project. After that you will restore a set of Node.js packages and use the webpack utility to compile TypeScript code into an output file named **embed.js**.

1. Copy three essential node.js development configuration files into the root folder of the **AppOwnsData** project.
   a) Locate these three files in the **StudentLabFiles** folder.
      i) **package.json** – the standard project file for all Node.js projects.
      ii) **tsconfig.json** – a configuration file used by the TypeScript compiler (TSC).
      iii) **webpack.config.js** – a configuration file used by the webpack utility.
   b) Copy **package.json**, **tsconfig.json** and **webpack.config.js** into the root folder of the **AppOwnsData** project.



Visual Studio Code makes it difficult to add existing files to a project folder. You can use the Windows Explorer to copy these three files from the **StudentLabFiles** folder to the **AppOwnsData** project folder.

2. Restore the Node.js packages which are referenced in **package.json**.
   a) Open **package.json** and review the Node.js packages referenced in **devDependencies** section.



   b) Open the Visual Studio Code terminal by clicking the **View > Terminal** menu command or by using **Ctrl+`** keyboard shortcut.
   c) Run the **npm install** command to restore the list of Node.js packages.

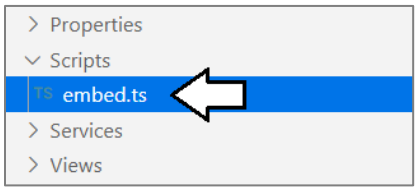
d) When you run the **npm install** command, **npm** will download all the Node.js packages into the **node_modules** folder.

3. Take a quick look at the **tsconfig.json** file.
   a) Open the **tsconfig.json** file in an editor window and examine the TypeScript compiler settings inside.
   b) When you are done, close **tsconfig.json** without saving any changes.
4. Take a quick look at the **webpack.config.js** file.
   a) Open the **webpack.config.js** file in an editor window and examine its content.

```
const path = require('path');

module.exports = {
  entry: './Scripts/embed.ts',
  output: {
    filename: 'embed.js',
    path: path.resolve(__dirname, 'wwwroot/js'),
  },
  resolve: {
    extensions: ['.js', '.ts']
  },
  module: {
    rules: [
      { test: /\.(ts)$/, loader: 'awesome-typescript-loader' }
    ],
  },
  mode: "development",
  devtool: 'source-map'
};
```

Note the **entry** property of **model.exports** object is set to **./Scripts/embed.ts**. The **path** and **filename** of the **output** object combine to a file path of **wwwroot/js/embed.js**. When the webpack utility runs, it will look for a file named **embed.ts** in the **Scripts** folder as its main entry point for the TypeScript compiler (tsc.exe) and produce an output file in named **embed.js** in the **wwwroot/js** folder.

   b) When you are done, close **webpack.config.js** without saving any changes.
5. Add a new TypeScript source file named **embed.ts**.
   a) In the **AppOwnsData** project folder, create a new top-level folder named **Scripts**.
   b) Create a new file inside the **Scripts** folder named **embed.ts**.



   c) In Windows Explorer, locate the **Exercise 5 - embed.ts.txt** file in the **StudentLabFiles** folder.
   d) Open **Exercise 5 - embed.ts.txt** in a text editor such as Notepad and copy all its contents to the Windows clipboard.

e) Return to Visual Studio Code and paste the contents of the Windows clipboard into **Embed.ts.**

f) Save your changes and close **embed.ts**.

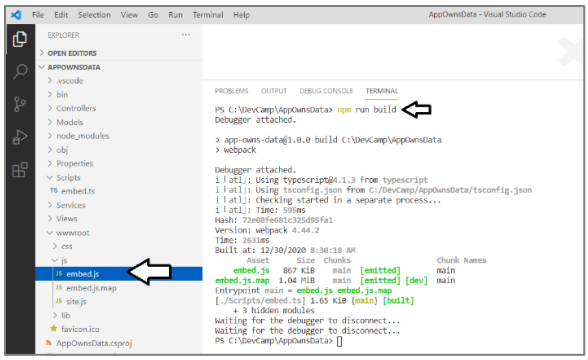
6. Use the webpack utility to compile **embed.ts** into **embed.js**.

a) Locate the original **embed.js** file in the **wwwroot/js** folder and delete it.

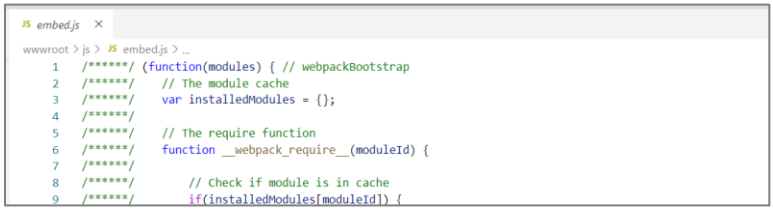
b) Open the Visual Studio Code terminal by clicking the **View > Terminal** menu command or by using **Ctrl+`** keyboard shortcut.

c) Run the **npm run build** command to run the webpack utility.

d) When you run **npm run build**, webpack should automatically generate a new version of **embed.js** in the **wwwroot/js** folder.



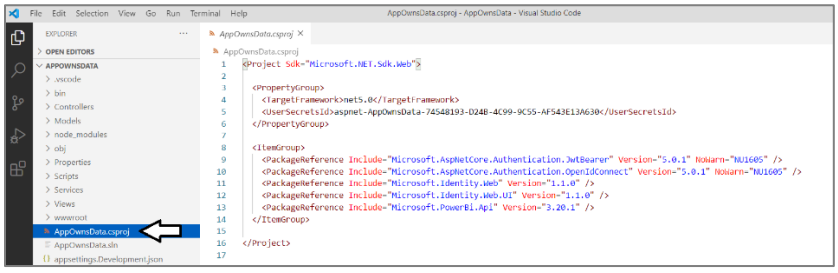
e) Open the new version of **embed.js**. You should see it is a source file generated by the webpack utility.



f) Close **embed.js** without saving any changes.

7. Update **AppOwnsData.csproj** to add the **npm run build** command as part of the MSBuild build process.

a) Open the project file named **AppOwnsData.csproj** in an editor window.



b) Add a new **Target** element named **PostBuild** to run the **npm run build** command as shown in the following code listing.

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <UserSecretsId>aspnet-AppOwnsData-74548193-D24B-4C99-9C55-AF543E13A630</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.OpenIdConnect" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Microsoft.Identity.Web" Version="1.1.0" />
    <PackageReference Include="Microsoft.Identity.Web.UI" Version="1.1.0" />
    <PackageReference Include="Microsoft.PowerBi.Api" Version="3.20.1" />
  </ItemGroup>

  <Target Name="PostBuild" AfterTargets="PostBuildEvent">
    <Exec Command="npm run build" />
  </Target>

</Project>
```

c) Save your changes and close **AppOwnsData.csproj**.

d) Return to the terminal and run the **dotnet build** command.

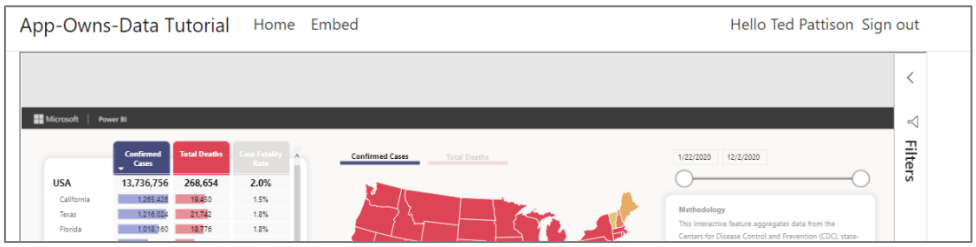
e) When you run the **dotnet build** command, the output window should show you that the webpack command is running.

Now whenever you start a debug session with the **{F5}** key, the TypeScript in **embed.ts** will be automatically compiled into **embed.js**.

8. Run the web application in the Visual Studio Code debugger to test your work on the **Embed** page.

a) Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

b) Click on the **Embed** link in the top nav menu to navigate to the **Embed** page and login when prompted to view the report.



c) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

When you test the **AppOwnsData** web application, it should behave just as it did when you tested it in Exercise 4. The difference is that now the client-side behavior is now implemented with TypeScript instead of JavaScript.

Exercise 6: Create a View Model for App Workspaces

Up to this point, you have implemented the **AppOwnsData** project to embed a single report by hard-coding the IDs of that report and its hosting workspace. In this exercise, you will remove the hard-coded IDs and extend the **Embed** page of the **AppOwnsData** project to dynamically discover what workspaces and reports are available to the current user.

1. Extend the **PowerBiServiceApi** class with a new method named **GetEmbeddedViewModel**.
   a) Locate the **PowerBiServiceApi.cs** in the **Service** folder and open it in an editor window.
   b) Add the following method named **GetEmbeddedViewModel** to the end of **PowerBiServiceApi** class.

```
public async Task<string> GetEmbeddedViewModel(string appWorkspaceId) {

    PowerBIClient pbiClient = GetPowerBiClient();

    Object viewModel;

    Guid workspaceId = new Guid(appWorkspaceId);
    var workspaces = (await pbiClient.Groups.GetGroupsAsync()).Value;
    var currentWorkspace = workspaces.First((workspace) => workspace.Id == workspaceId);
    var datasets = (await pbiClient.Datasets.GetDatasetsInGroupAsync(workspaceId)).Value;
    var reports = (await pbiClient.Reports.GetReportsInGroupAsync(workspaceId)).Value;

    IList<GenerateTokenRequestV2Dataset> datasetRequests = new List<GenerateTokenRequestV2Dataset>();
    foreach (var dataset in datasets) {
        datasetRequests.Add(new GenerateTokenRequestV2Dataset(dataset.Id));
    };

    IList<GenerateTokenRequestV2Report> reportRequests = new List<GenerateTokenRequestV2Report>();
    foreach (var report in reports) {
        reportRequests.Add(new GenerateTokenRequestV2Report(report.Id, allowEdit: true));
    };


    IList<GenerateTokenRequestV2TargetWorkspace> workspaceRequests =
        new GenerateTokenRequestV2TargetWorkspace[] {
            new GenerateTokenRequestV2TargetWorkspace(workspaceId)
    };


    GenerateTokenRequestV2 tokenRequest =
        new GenerateTokenRequestV2(datasets: datasetRequests,
                                   reports: reportRequests,
                                   targetWorkspaces: workspaceRequests);


    // call to Power BI Service API and pass GenerateTokenRequest object to generate embed token
    string embedToken = pbiClient.EmbedToken.GenerateToken(tokenRequest).Token;


    viewModel = new
    {
        workspaces = workspaces,
        currentWorkspace = currentWorkspace.Name,
        datasets = datasets,
        reports = reports,
        token = embedToken
    };

    return JsonConvert.SerializeObject(viewModel);
}
```

The **GetEmbeddedViewModel** method accepts an **appWorkspaceId** parameter and returns a string value with JSON-formatted data. If the **appWorkspaceId** parameter contains a GUID, the **GetEmbeddedViewModel** method returns a view model for the app workspace associated with that GUID.

You can copy and paste this method from the **Exercise 6 - PowerBiServiceApi.cs.txt** file in the **StudentLabFiles** folder.

c) Enhance your conceptual understanding of the data involved by examining the JSON returned by **GetEmbeddedViewModel**.


```
{
  "workspaces": [
    { "id": "912f2b34-7daa-4589-83df-35c75944d864", "name": "Operations", "isReadOnly": false
    { "id": "ddab987a-7941-4eb5-a084-370d422dfcac", "name": "Dev Camp Lab", "isReadOnly": fals
  ],
  "currentWorkspace": "Dev Camp Lab",
  "datasets": [
    { "id": "7760cec0-6048-4783-9de1-2ff810accd31", "name": "COVID-19 US", "configuredBy": "Te
  ],
  "reports": [
    { "id": "3e75d100-b21a-4129-8e84-6c6e90ccbd7e", "name": "COVID-19 US", "embedUrl": "https:
  ],
  "token": "H4sIAAAAAAAAEAB2Sx66rVgBF_-VOiUQ3EOkNMKba9M6MQ-9waIYo_56rjPcarb3--bHSq5_S_Ofvn2d1fK
}
```


d) Underneath the **GetEmbeddedViewModel** method, add another method named

```
public async Task<Group> GetFirstWorkspace() {
    PowerBIClient pbiClient = this.GetPowerBiClient();
    var workspaces = (await pbiClient.Groups.GetGroupsAsync()).Value;
    if (workspaces.Count > 0) {
        return workspaces.First();
    }
    Else {
        return null;
    }
}
```

e) Save your work and close **PowerBiServiceApi.cs**.

2. Modify **Embed** method in **HomeController** to call the **GetEmbeddedViewModel** method.

a) Locate the **HomeController.cs** file and open it in an editor window.

b) Locate the **Embed** method which should currently match this **Embed** method implementation.

c) Delete the **Embed** method implementation and replace it the following code.

```
public async Task<IActionResult> Embed(string workspaceId) {

    try {
        Guid guidTest = new Guid(workspaceId);
        var viewModel = await this.powerBiServiceApi.GetEmbeddedViewModel(workspaceId);
        return View(viewModel as object);
    }
    Catch {
        var firstWorkspace = await this.powerBiServiceApi.GetFirstWorkspace();
        if (firstWorkspace == null) {
            return RedirectToPage("/Error");
        }
        Else {
            return RedirectToPage("/Embed", null, new { workspaceId = firstWorkspace.Id });
        }
    }
}
```

d) Save your work and close **HomeController.cs**.

There are a few things to note about the new implementation of the **Embed** controller action method. First, the method now takes a string parameter named **workspaceId**. When this controller method is passed a workspace ID in the **workspaceId** query string parameter, it passes that workspace ID along to the **PowerBiServiceApi** class when it calls the **GetEmbeddedViewModel** method.

The second thing to note about this example if that the string-based **viewModel** variable is cast to a type of **object** in the **return** statement using the syntax **View(viewModel as object)**. This is a required workaround because passing a string parameter to **View()** would fail because the string value would be treated as a view name instead of a view model being passed to the underlying view.

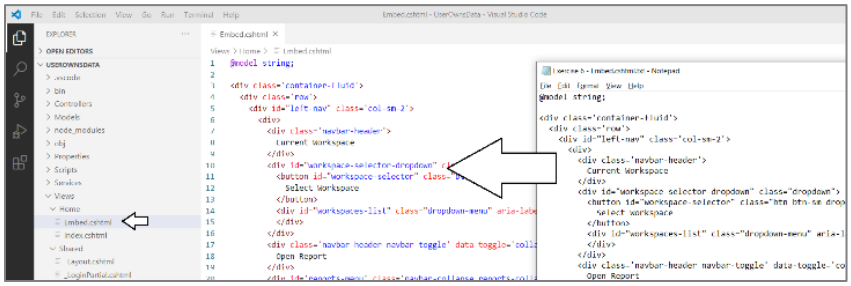
3. Replace the code in **Embed.cshtml** with a better implementation.

a) Locate **Embed.cshtml** file in the **Views > Home** folder, open it in an editor window and delete all the content inside.

b) In Windows Explorer, locate the **Exercise 6 - Embed.cshtml.txt** file in the **StudentLabFiles** folder.

c) Open **Exercise 6 - Embed.cshtml.txt** in a text editor such as Notepad and copy all its contents to the Windows clipboard.

d) Return to Visual Studio Code and paste the contents of to the Windows clipboard into **Embed.cshtml.**



e) Save your changes and close **Embed.cshtml.**

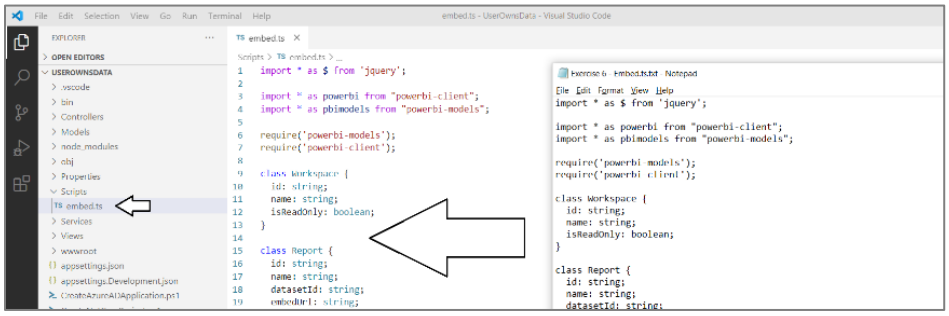
4. Replace the code in **Embed.ts** with a better implementation.

a) Locate **Embed.ts** file in the **Scripts** folder, open it in an editor window and delete all the content inside.

b) In Windows Explorer, locate the **Exercise 6 - Embed.ts.txt** file in the **StudentLabFiles** folder.

c) Open **Exercise 6 - Embed.ts.txt** in a text editor such as Notepad and copy all its contents to the Windows clipboard.

d) Return to Visual Studio Code and paste the content of **Exercise 6 - Embed.ts.txt** into **Embed.ts.**



e) Save your changes and close **Embed.ts.**

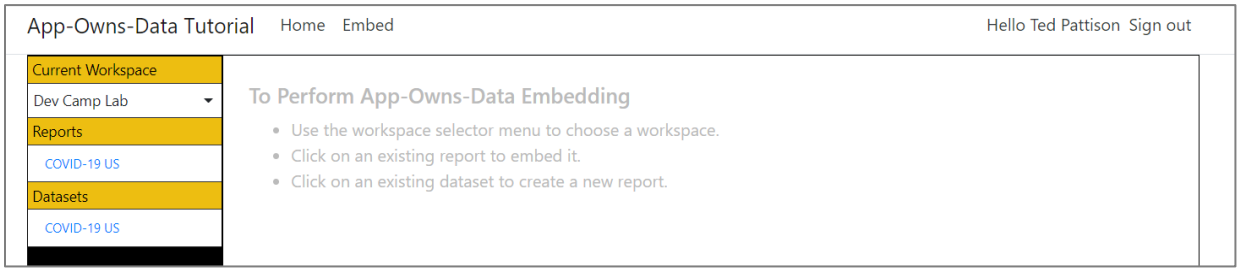
Revieing the client-side code in **Embed.ts** has been left as an exercise for the reader, After you run and test the **AppOwnsData** application at the end of this exercise and you have experienced the user interface from the user perspective, you might want to examine the code in **Embed.ts** to see how this application implements the functionality to navigate between workspace and to discover the reports and datasets in each workspace.

5. Run the web application in the .NET debugger to test your work on the **Embed** page.

a) Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

b) Click on the **Embed** link in the top nav menu to navigate to the **Embed** page and login when prompted.

c) The **Embed** page should appear much differently than before as shown in the following screenshot.

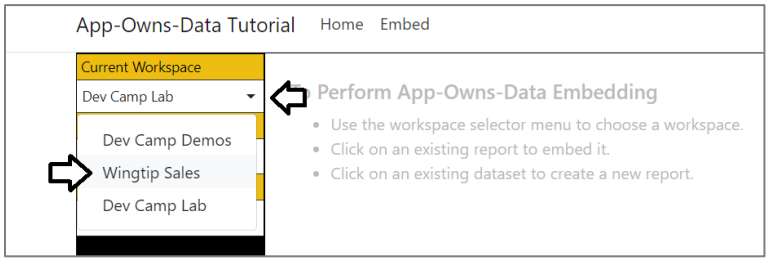


Note there is a dropdown list for the **Current Workspace** that you can use to navigate across workspaces.

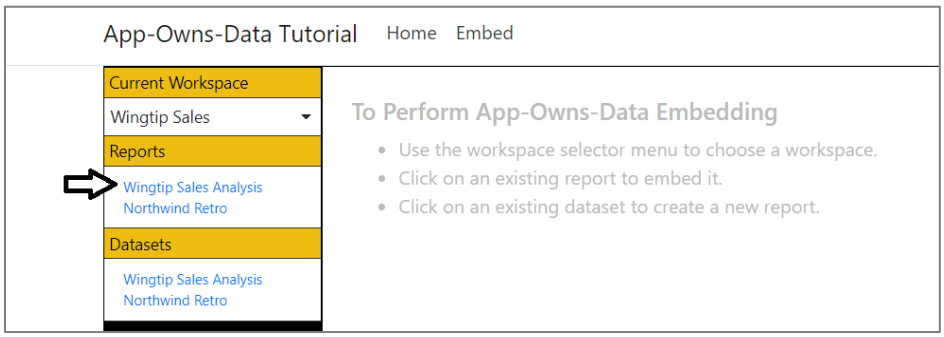
d) Return to the Power BI in the browser and add the App-Owns-Data Sample App as an admin on a few other workspaces.

e) Return back to the **AppOwnsData** app and refresh the page.

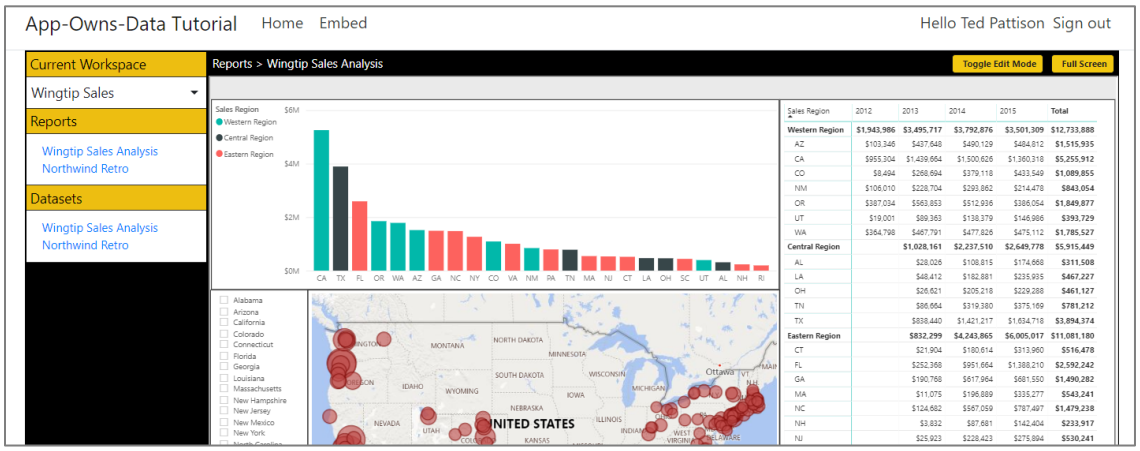
f) Use the **Current Workspace** dropdown menu to navigate to the workspace you created earlier in this lab.



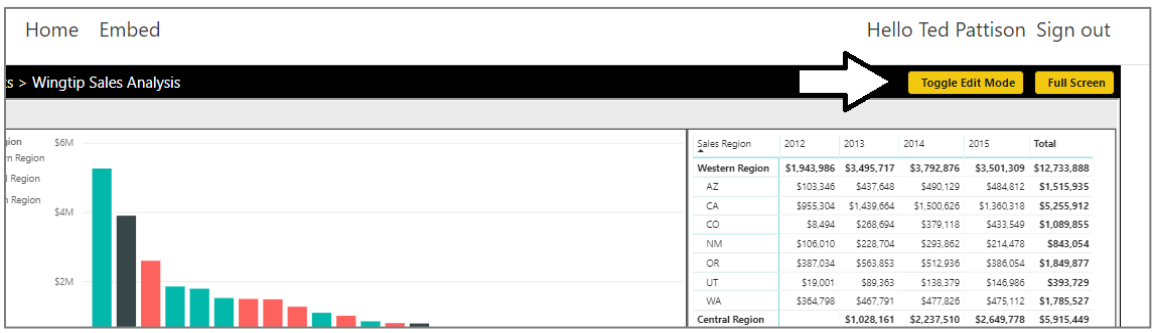
g) Click on a report in the **Open Report** section.



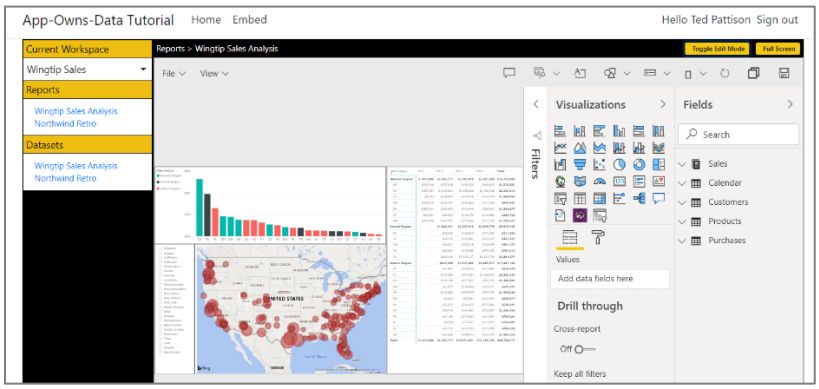
h) The report should open in read-only mode.



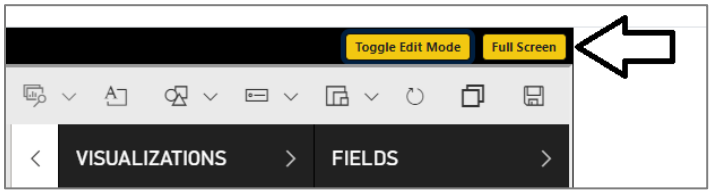
i) Click the **Toggle Edit Mode** button to move the report into edit mode.



j) Note that when the report goes into edit mode, there isn't much space to work on the report while editing.



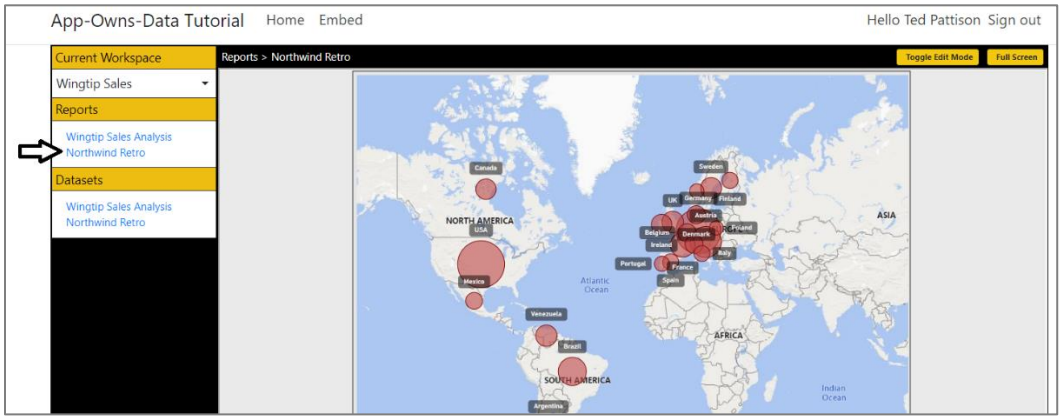
k) Click the **Full Screen** button to enter full screen mode



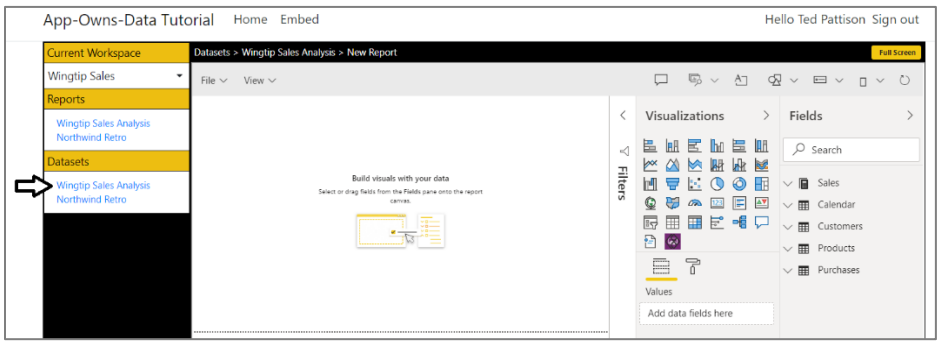
You can invoke the **File > Save** command in a report that is in edit mode to save your changes.

l) Press the **Esc** key in the keyboard to exit full screen mode.

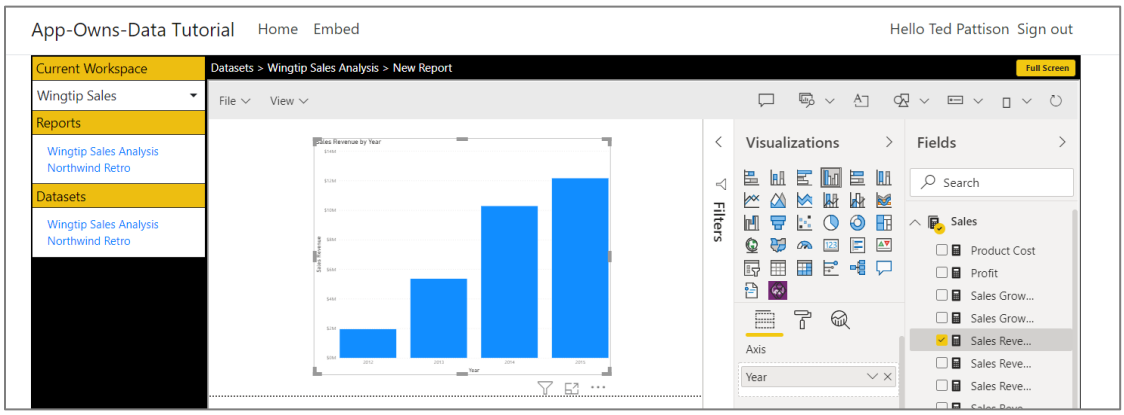
m) Click on a second report in the **Open Report** section to navigate between reports.



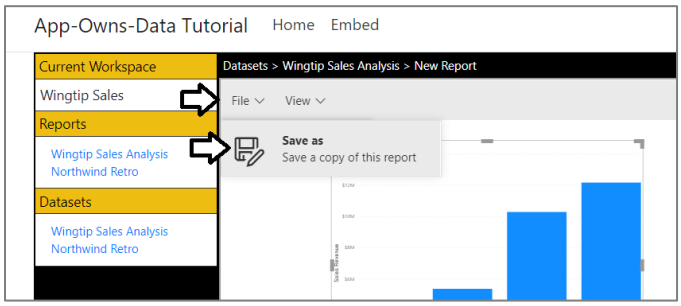
n) Create a new report by clicking on a dataset name in the **New Report** section.



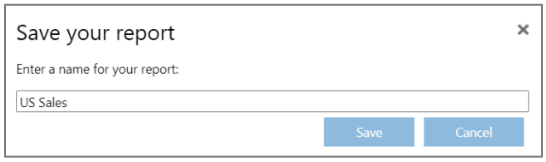
o) Add a simple visual of any type to the new report.



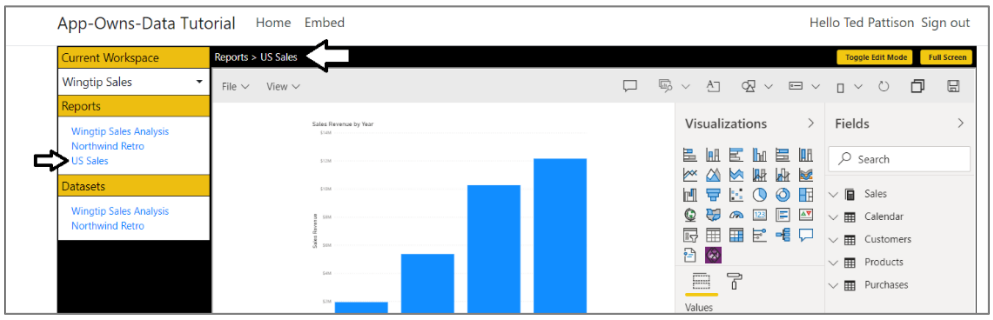
p) Save the new report using the **File > Save as** menu command.



q) Give your new report a name.



r) After you click save, the new report should show up in the Open Report section and be displayed in read-only mode.



s) When you're done testing, close the browser, return to Visual Studio Code and stop the debug session.

Congratulations. You have now completed this lab and have gained cutting-edge experience developing with Power BI embedding

