Developing for Power BI using .NET Core

In this lab, you will create a new .NET Core project for a custom 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 and to acquire 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 that your code receives the benefits of strong typing, IntelliSense and compile-time type checks.

To complete this lab, your developer workstation must configure to allow the execution of PowerShell scripts. Your developer workstation must also have the following software and developer tools installed.

1) **PowerShell cmdlet library for AzureAD** – [download]
2) **DOTNET Core SDK 3.1 or later** – [download]
3) **Node.js** – [download]
4) **Visual Studio Code** – [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 .NET Core MVC Web Application Project

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

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/NetCore-UserOwnsData-Tutorial/raw/master/StudentLabFiles.zip

   c) Extract the **StudentLabFiles** folder from **StudentLabFiles.zip** into a to a local folder such as **c:\DevCamp\StudentLabFiles**.
   d) The **StudentLabFiles** folder should contain the set of files shown in the following screenshot.
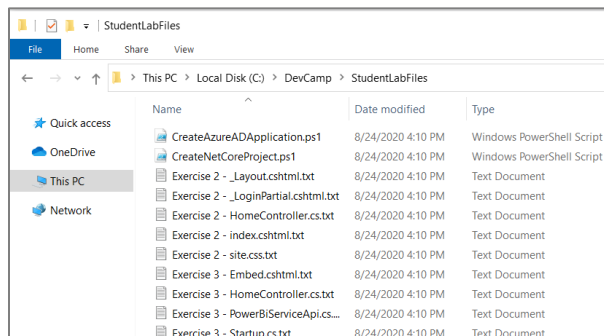
2. Create a new .NET Core project using the .NET Core CLI and a PowerShell script.
   a) Create a new folder on your local drive named **UserOwnsData** at a location such as **c:\DevCamp\UserOwnsData**.
   b) In the **StudentLabFiles** folder, locate the scripts named **CreateNetCoreProject.ps1** and **CreateAzureADApplication.ps1**.
   c) Copy **CreateNetCoreProject.ps1** and **CreateAzureADApplication.ps1** into the **UserOwnsData** folder.
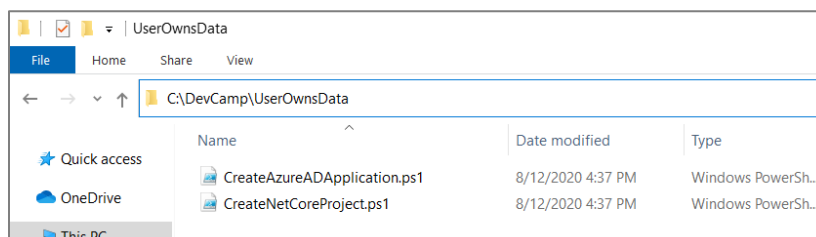
3. Review the PowerShell code in **CreateNetCoreProject.ps1**.

    a) Open **CreateNetCoreProject.ps1** in a text editor such as Notepad or the PowerShell Integrated Scripting Environment (ISE).

    b) Review the code in **CreateNetCoreProject.ps1** which creates a new .NET Core project and adds a few .NuGet packages.

```
dotnet new mvc --auth SingleOrg --framework netcoreapp3.1

dotnet remove package Microsoft.AspNetCore.Authentication.AzureAD.UI

# update to latest available version of Microsoft.Identity.Web
dotnet add package Microsoft.Identity.Web -v 0.2.3-preview
dotnet add package Microsoft.Identity.Web.UI -v 0.2.3-preview
dotnet add package Microsoft.PowerBi.Api
```
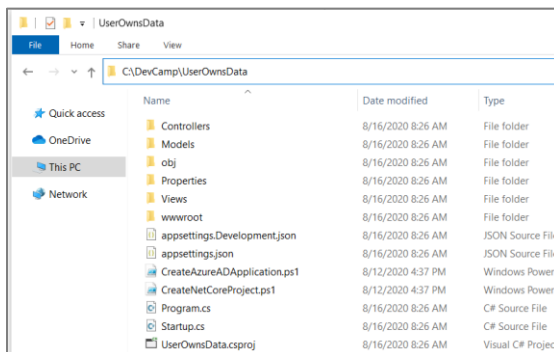
    c) Open up a PowerShell console and set the location of the command prompt to the **UserOwnsData** folder.

    d) Execute the script **CreateNetCoreProject.ps1** by invoking the command **.\CreateNetCoreProject.ps1**.
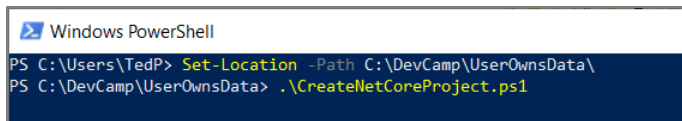
    e) Once the script has completed, you should see that the **UserOwnsData** folder has been populated with project files.

4. Open the **UserOwnsData** folder with Visual Studio Code

    a) Launch Visual Studio Code.

   b) Use the **Open Folder** command in Visual Studio Code to open the **UserOwnsData** folder.
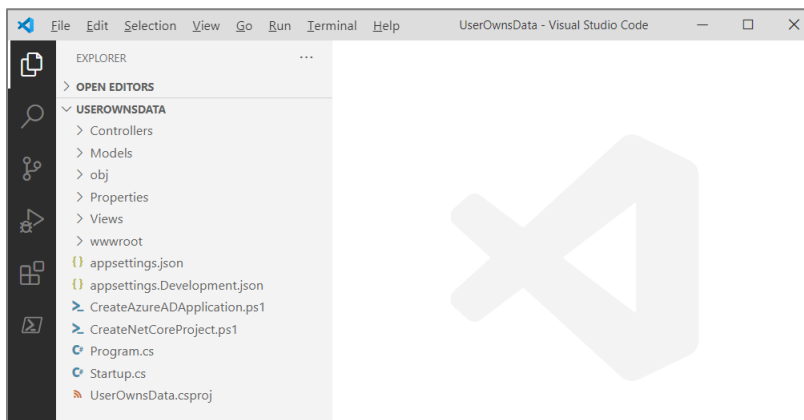
You will not be able to build the project yet. That is because the PowerShell script removed the .NuGet package for an older authentication library named **Microsoft.AspNetCore.Authentication.AzureAD.UI** and added new packages for the new Microsoft authentication library **Microsoft.Identity.Web**. You'll be required to modify some code in this project before it will build. But before that, you will install the Microsoft C# extension in Visual Studio Code to ensure you have full support for working with .NET Core projects.

5. Configure Visual Studio Code with the extensions needed for .NET Core development.

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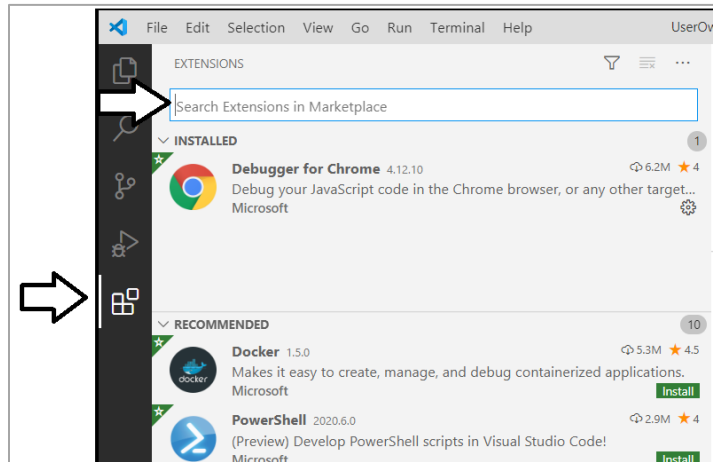
   d) Find and install the **C#** extension from Microsoft if it is not already installed.
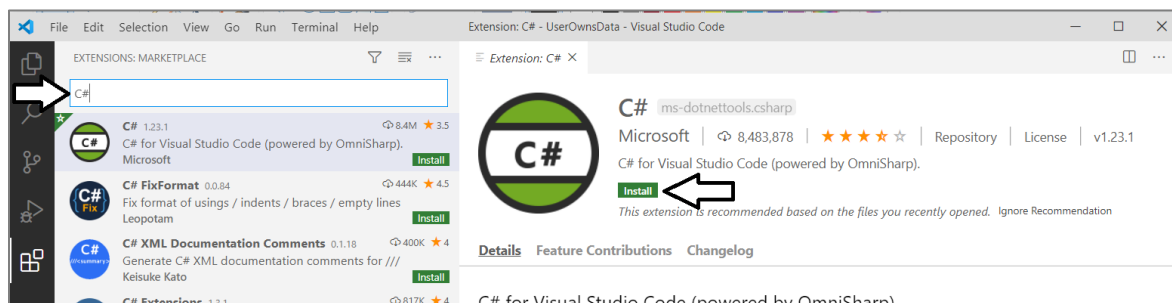
   e) Find and install the **Debugger for Chrome** extension from Microsoft if it is not already installed.

   f) You should be able to confirm that the **C#** extension and the **Debugger for Chrome** extensions are now installed.
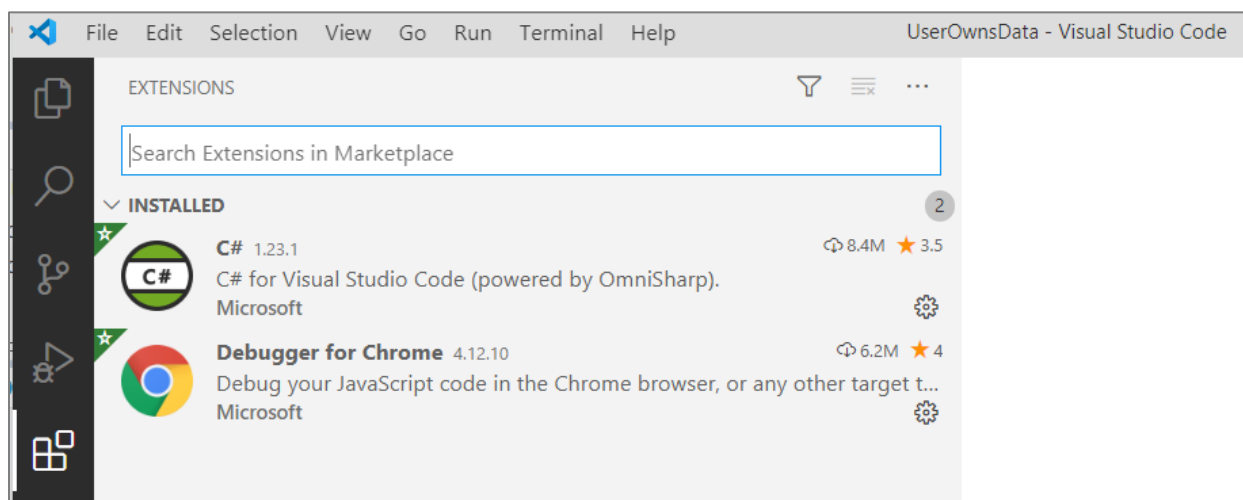
It is OK if you have other Visual Studio Code extensions installed as well. It's just important that you install these two extensions in addition to whatever other extensions you may have installed.

Exercise 2: Implement User Login using Microsoft.Identity.Web

In this exercise, you start by running a PowerShell script to create a new confidential client application in Azure AD. After that, you will configure your project to implement an interactive user login experience with Azure AD by using the **Microsoft.Identity.Web** library.

1. Create a new Azure AD application by running the PowerShell script named **CreateAzureADApplication.ps1**.

   a) Open the PowerShell script named **CreateAzureADApplication.ps1** in a text editor such as Notepad or the PowerShell ISE.

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

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

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

   ```
   $appDisplayName = "User-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**.

   d) 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
   ```

   e) 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
   ```

   f) Execute the PowerShell script named **CreateAzureADApplication.ps1**.

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

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

> The text file **UserOwnsDataSampleApp.txt** contains JSON configuration data that you will copy and paste into **appsettings.json**.

2. Copy the JSON in **UserOwnsDataSampleApp.txt** into the **appsettings.json** file in your project.

    a) Return to the **UserOwnsData** project in Visual Studio Code and open the **appsettings.json** file.

   b) The **appsettings.json** file should initially appear like the screenshot below.
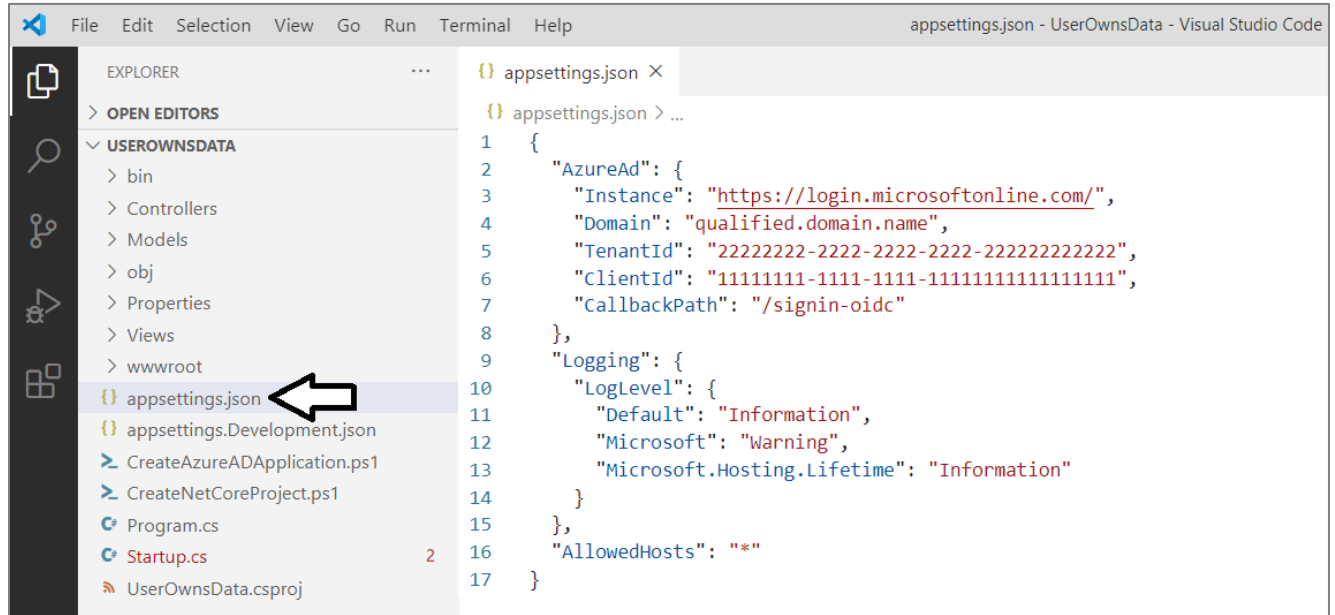
c) Delete the contents of **appsettings.json** and replace it by copying and pasting the contents of **UserOwnsDataSampleApp.txt**
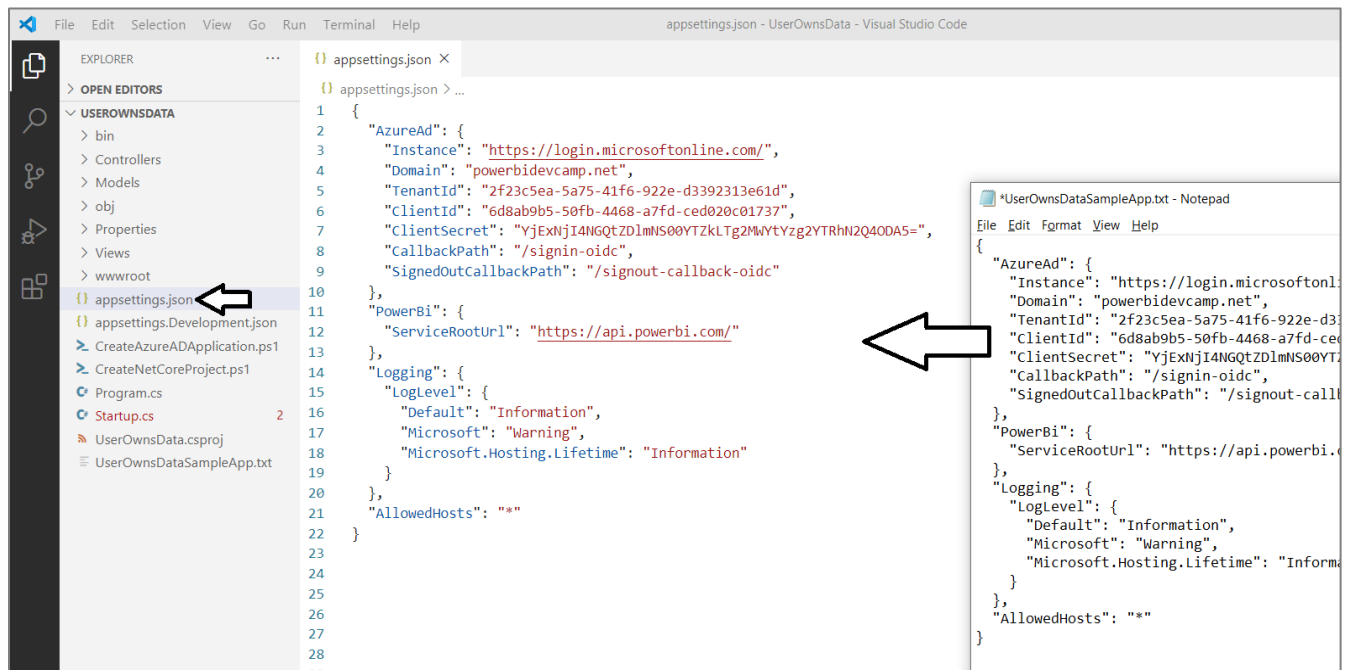
Note the **PowerBi:ServiceRootUrl** parameter has been added as a custom configuration value to track the base URL to the Power BI Service. When you are programming against the Power BI Service in Microsoft public cloud, the URL is https://api.powerbi.com/. However, the root URL for the Power BI Service 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 whenever required.

3. Modify the code in **Startup.cs** to properly initialize the authentication service provided by **Microsoft.Identity.Web**.

    a) Open the **Startup.cs** file in an editor window.

    b) Remove line that imports **Microsoft.AspnetCore.Authentication.AzureAD.UI** which is causing a build error.
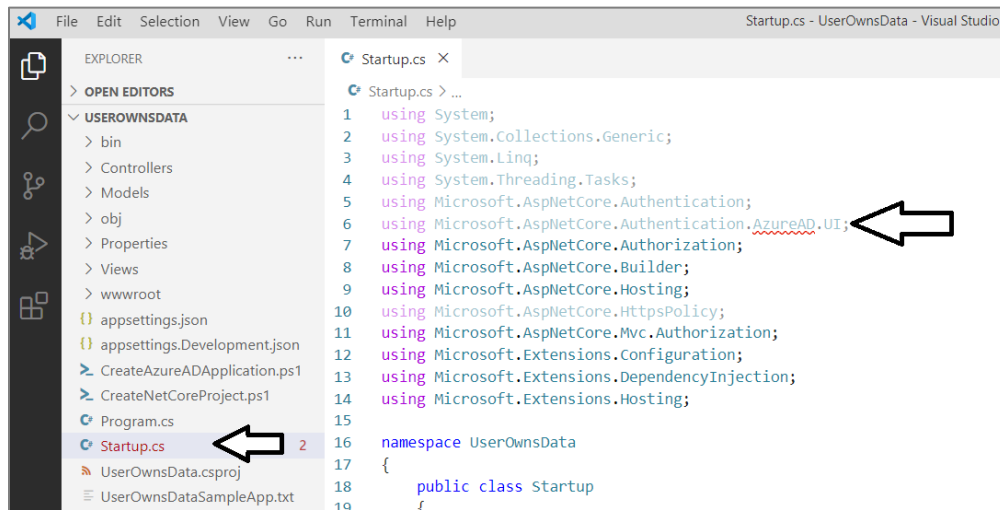
    c) Place your cursor below the existing **using** statements and add the following **using** statements.

```
using Microsoft.Identity.Web;
using Microsoft.Identity.Web.UI;
using Microsoft.Identity.Web.TokenCacheProviders;
using Microsoft.Identity.Web.TokenCacheProviders.InMemory;
```

    d) Move down into the **ConfigureServices** method. Locate and delete the code which calls **services.AddAuthentication**.

    e) Replace the call you removed with a call to **services.AddMicrosoftWebAppAuthentication**.as shown in the following listing.

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

  services.AddMicrosoftWebAppAuthentication(Configuration);
```

    f) Move down in the **ConfigureServices** method and locate the code which calls **services.AddControllersWithViews**.

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

    g) Replace this code with the code shown in the following listing.

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

mvcBuilder.AddMicrosoftIdentityUI();
```

h) At this point, the **ConfigureServices** method in your project should match what is shown in the following code listing.

```
public void ConfigureServices(IServiceCollection services) {

    services.AddMicrosoftWebAppAuthentication(Configuration);

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

    mvcBuilder.AddMicrosoftIdentityUI();

    services.AddRazorPages();
}
```
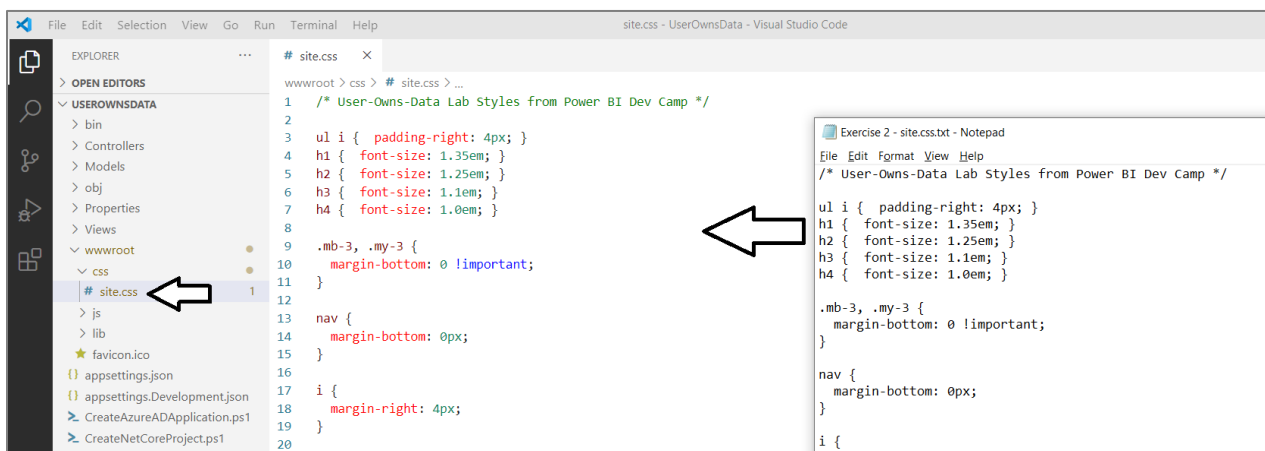
i) Save your changes and close **Startup.cs**.

4. Copy a pre-written set of CSS styles into the **UserOwnsData** 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.
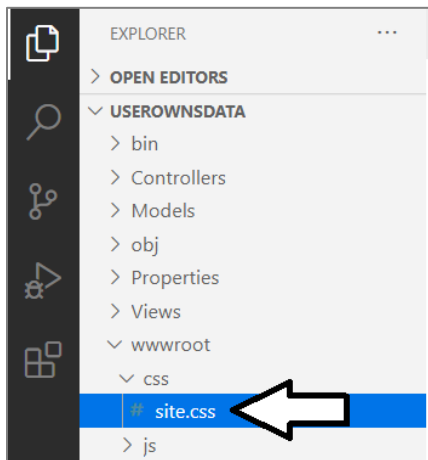
c) Using the Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **Exercise 2 - site.css.txt**.

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 **Exercise 2 - site.css.txt** into **sites.css**.

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

5. 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.
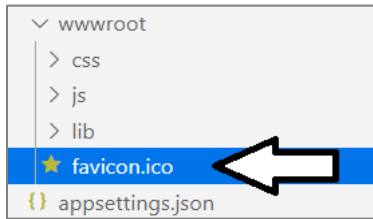
> Any file you add the **wwwroot** folder will appear at the root folder of the website created by the **UserOwnsData** project. By adding the **favicon.ico** file, this web application will now display a custom **favicon.ico** in the browser page tab.

6. Modify the partial razor view file named **_LoginPartial.cshtml** to integrate with the **Microsoft.Identity.Web** authentication library..

   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, you can see that there are two links which have an **asp-area** tag with a value of **AzureAD**.

> The **asp-area** value of **AzureAD** is used by the older library named **Microsoft.AspNetCore.Authentication.AzureAD.UI**. Since we are moving to the **Microsoft.Identity.Web** authentication library, the value must be changed from **AzureAD** to **MicrosoftIdentity**.

   d) Delete all the code **_LoginPartial.cshtml** and replace it with the code shown in the following code listing.

```
@using System.Security.Principal

<ul class="navbar-nav">
  @if (User.Identity.IsAuthenticated) {
    <li class="nav-item">
      <span class="navbar-text text-dark">Hello @User.FindFirst("name").Value</span>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignOut">
          Sign out
      </a>
    </li>
  }
  else {
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="MicrosoftIdentity" asp-controller="Account" asp-action="SignIn">
          Sign in
      </a>
    </li>
  }
</ul>
```

> If it's easier, you can copy and paste this code from file named **Exercise 2 - _LoginPartial.cshtml.txt** in the **StudentLabFiles** folder.

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

7. Modify the HTML in **Index.cshtml** to display differently 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 User-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**.
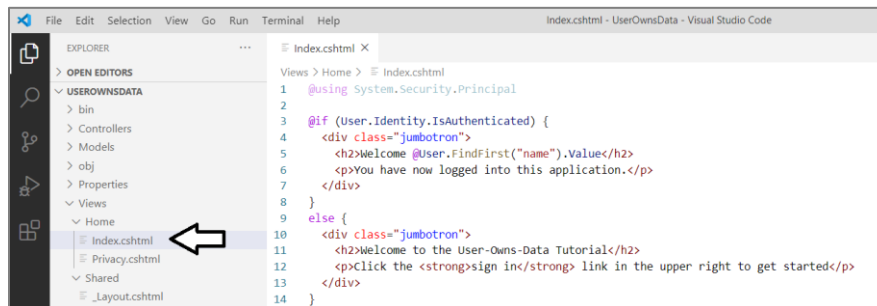
> When you create a new .NET Core project which supports authentication, the underlying project template creates a home page that requires authentication. To support a more natural log in experience, it often makes sense to configure your web application so that an anonymous user access the home page. In the next step you will modify the **Home** controller so the home page is accessible to the anonymous user.

8. 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.
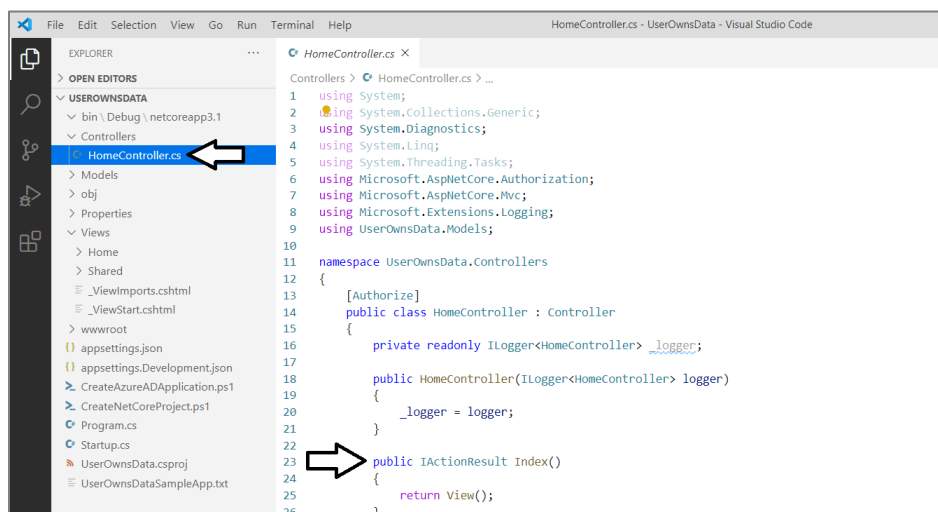
c) Add the **[AllowAnonymous]** attribute to the **Index** method as shown in the following code listing.

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

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

> You have now modified the project to the point where you can run the web application in the .NET Core debugger. In the next step, you will start the debugger so you can test your web application as it runs in the browser.

9. Test the **UserOwnsData** project by running it in the .NET Core debugging environment.

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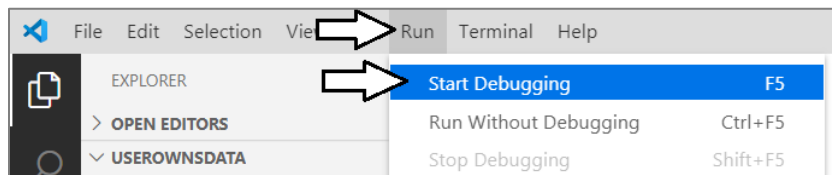

> This will be the first time that you run this project in the Visual Studio Code debugger. However, running your project in the Visual Studio Code debugger requires a special file named launch.json. When you start the debugger in a project that currently has no **launch.json** file, Visual Studio Code will prompt you to select a debugging environment. When you select a debugging environment based on .NET Core, Visual Studio Code will then automatically create the **launch.json** file for you behind the scenes and use it to start a debugging session.

b) When prompted to **Select Environment**, select **.NET Core**.
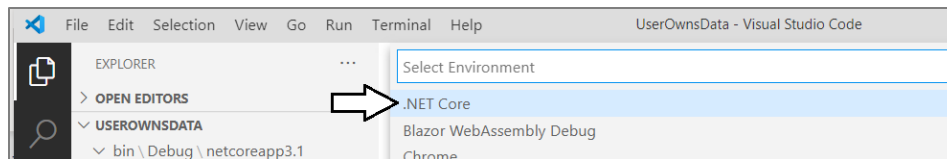
c) Behind the scenes, Visual Studio Code has created a folder named **.vscode** with two files **launch.json** and **tasks.json**.
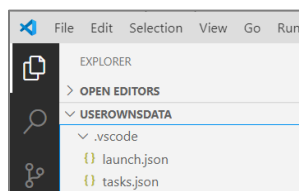

d) Once the debugging session has initialized, the browser should display the home page using anonymous access.

e) Click the **Sign in** link to test put the user experience when authenticating with Azure AD.
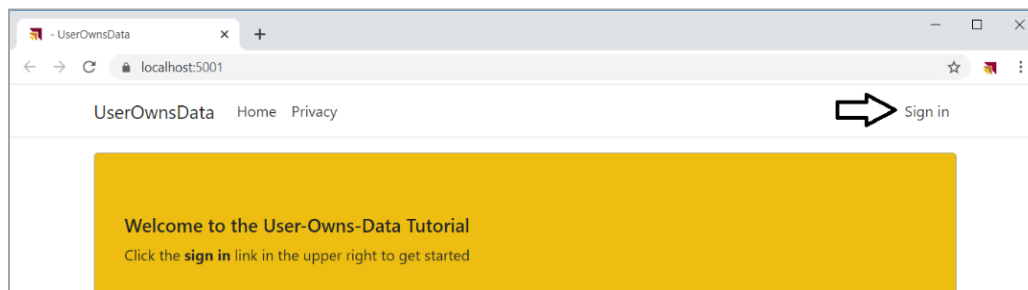

f) When prompted to **Sign in**, enter your user account and password to authenticate.

> You should now be prompted by the **Permissions requested** dialog. Azure AD wants you to consent to the delegated permissions that have been requested by the **User-Owns-Data Sample App**.

g) Click the **Accept** button to grant the **User-Owns-Data Sample App** the required permissions it has requested.
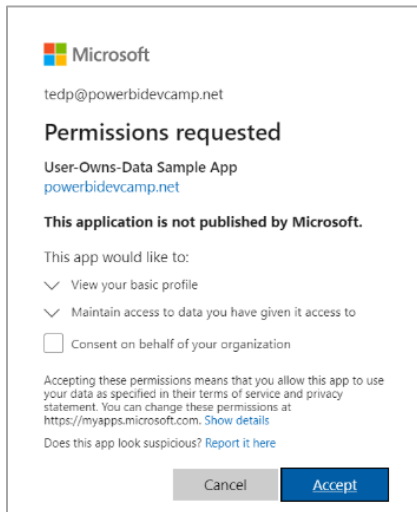
h) After clicking the **Accept** button, you should be redirected back to the home page of the web application.
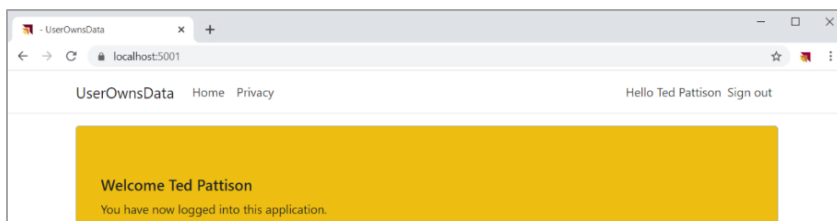

> At this point, the user should be authenticated. For example, you should see the logged in user name to the left of the **Sign out** link in the top right corner. You should also see that the home page displays text that welcomes the user by name.

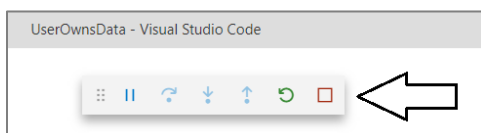10. Test the user experience for logging out.

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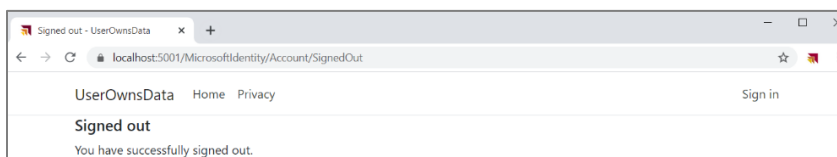
c) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

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.

11. 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 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**.

12. 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**.
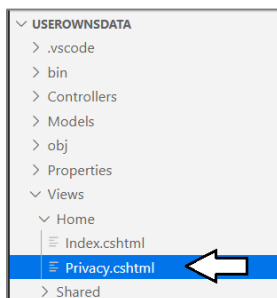
    b) Rename the **Privacy.cshtml** razor file to **Embed.cshtml**..
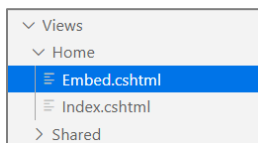
    c) Open **Embed.cshtml** in a code editor.

    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 Core 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.

13. 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.
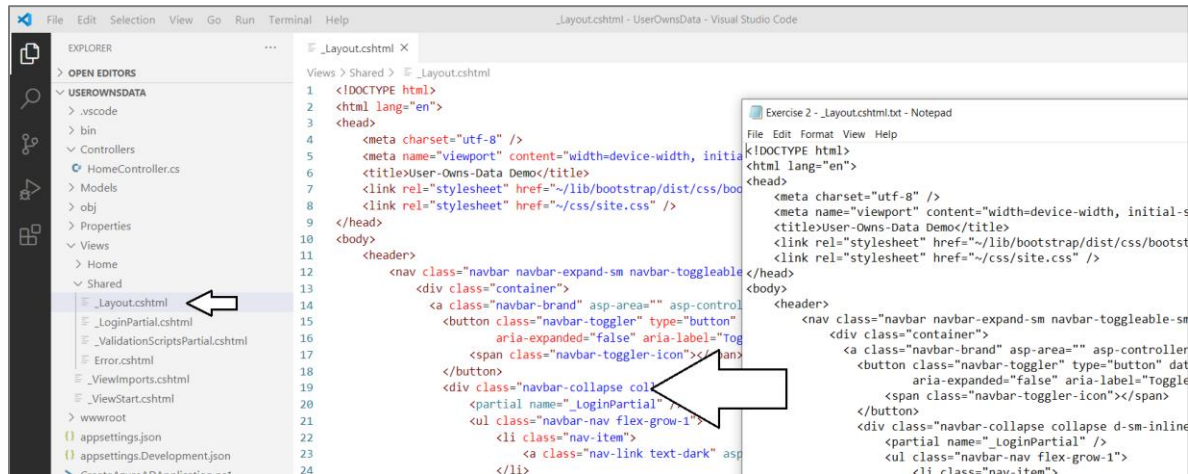

   e) Save your changes and close **_Layouts.cshtml**

14. 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 **UserOwnsData** 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.
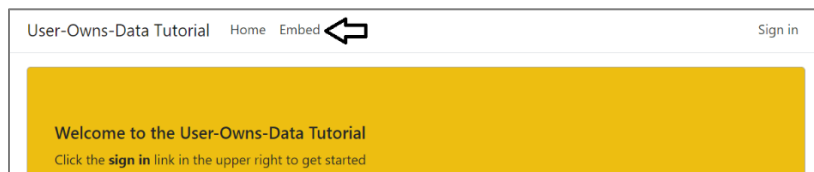
   d) When you attempt to navigate to the **Embed** page as an anonymous user, you'll be prompted to pick an account and log in.

   e) Log in using your user name and password.
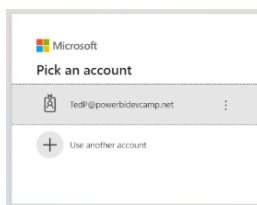
   f) Once you have logged in, you should be automatically redirected to the **Embed** page.
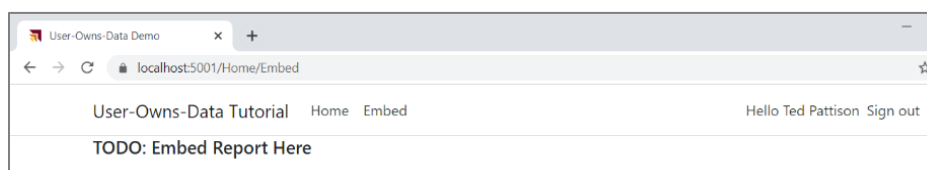
   g) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

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 Core projects using Visual Studio 2019, you can skip the next step and move ahead to *Exercise 3: Call the Power BI Service API*.

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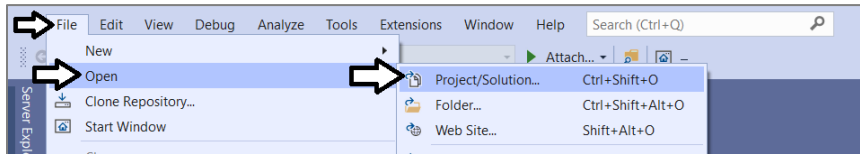
   c) In the **Open Project/Solution** dialog, select the **UserOwnsData.csproj** file in the **UserOwnsData** folder and click **Open**.

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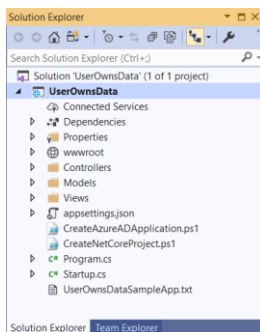

There is one big difference between developing with Visual Studio Code and Visual Studio 2019. Visual Studio Code only requires project files (*.csproj). However, Visual Studio 2019 requires that you work with both project files and solution files (*.sln). In the next step you will save a new project file for the **UserOwnsData** 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 in the tree with the caption **Solution "UserOwnsData"**.

   f) From the **File** menu, select the **Save UserOwnsData.sln** menu command.
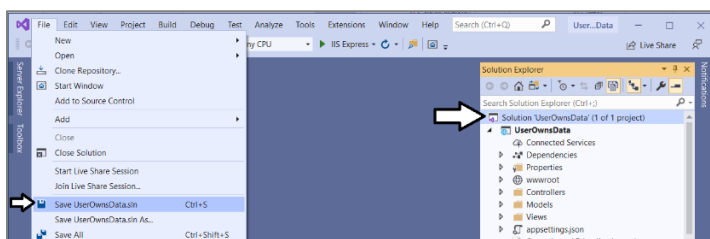
   g) Save the solution file **UserOwnsData.sln** in the **UserOwnsData** project folder
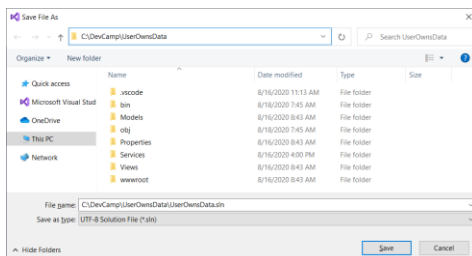
Remember that the **UserOwnsData.sln** file is only used by Visual Studio 2019 and it not used at all in Visual Studio Code.

16. Test the **UserOwnsData** web application in Visual Studio 2019.

    a) On the Visual Studio 2019 toolbar, locate the **Start** button with the green arrow.

    b) If you drop down the **Start** button menu, you should see the project configuration of **IIS Express** and **UserOwnsData**.

    c) Leave the **Start** button menu set for **IIS Express** and click the **Start** button to start run the project in the debugger.
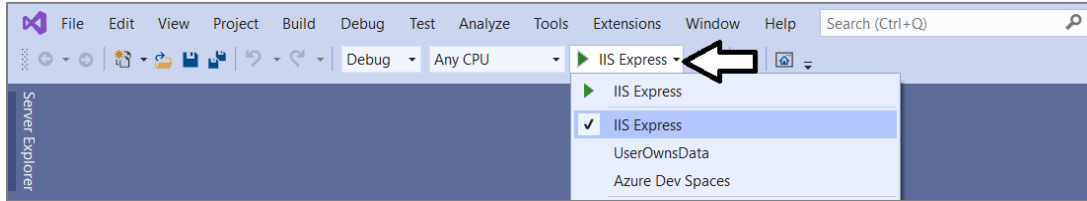

    d) The **UserOwnsData** application should run and serve up the home page using localhost and a port number around 44300.
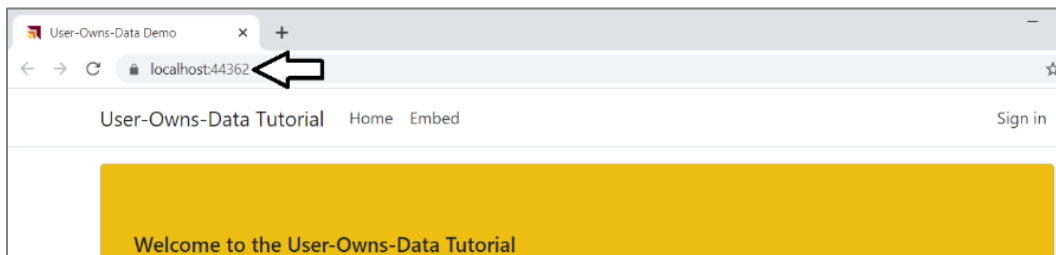

You should be able to test the **UserOwnsData** application by logging in and navigating to the embed page.

    e) After you are done testing the application, close the browser, return to Visual Studio and stop the current debugging session.

    f) Drop down the **Start** button menu and switch the debug profile to **UserOwnsData**.
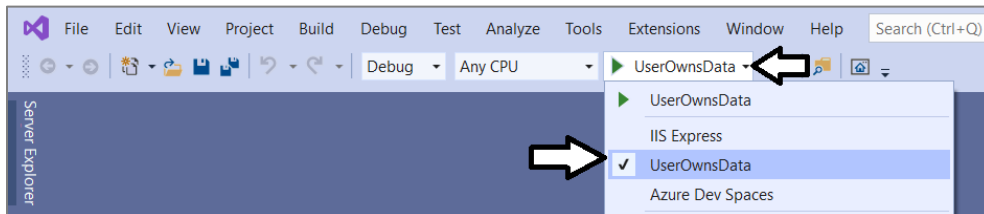
By switching the debug configuration from **IIS Express** to **UserOwnsData**, you are effectively changing the environment where you will debug your application. When you switch the configuration to **UserOwnsData**, Visual Studio 2019 will use the .NET Core web server process named **Kestrel** for debugging sessions. The base URL of the web site will change to **https://localhost:5001/**.

    a) With the **Start** button menu set for **UserOwnsData,** click the **Start** button to start run the project in the debugger.

    b) Note when you are debugging with **Kestrel**, you will see more activity output in the console window.
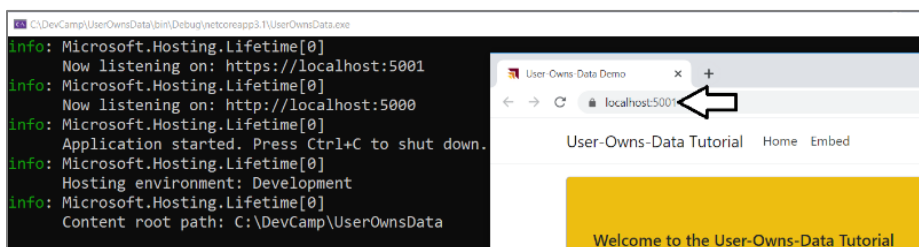

    c) The application should run and behave just as it has before.

At this point, the lab instructions and screenshots will return to using Visual Studio Code. However, if you'd like to continue using Visual Studio 2019, you should be able to complete all the steps in the following exercises.

Exercise 3: Call the Power BI Service API

In this exercise, you will begin by ensuring you have a Power BI app workspace and a report for testing. After that, you will add support to the **UserOwnsData** web application to acquire 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 with the same user account you used to create the Azure AD application earlier.
    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 Demos**.
   a) Click the **Workspace** flyout menu in the left navigation.
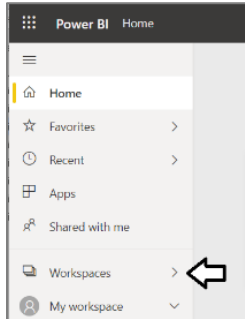
    b) Click the **Create app workspace** button to display the **Create an app workspace** dialog.
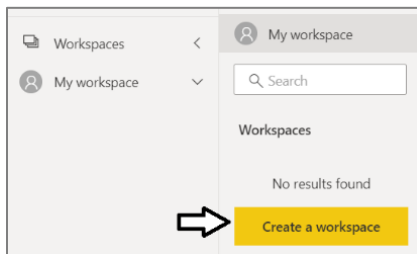
    c) In the **Create an app workspace** pane, enter a workspace name such as **Dev Camp Demos**.
   d) Click the **Save** button to create the new app workspace.
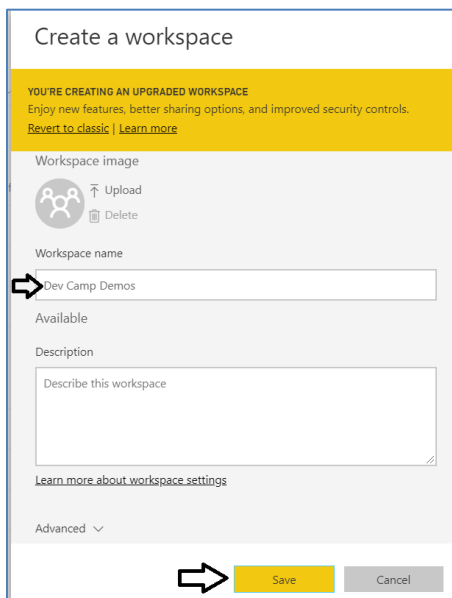
e) When you click **Save**, the Power BI service should create the new app workspace and then switch your current Power BI session to be running within the context of the new **Dev Camp Demos** workspace.
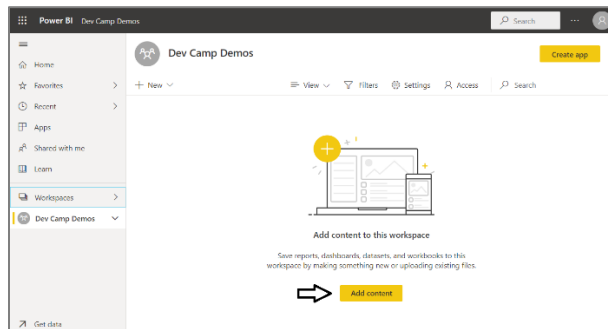

Now that you have created the new app workspace, the next step is to upload a PBIX project file created with Power BI Desktop. You are free to use you own PBIX file as long as the PBIX file does not have row-level security (RLS) enabled. If you don't have your own PBIX file, you can download the sample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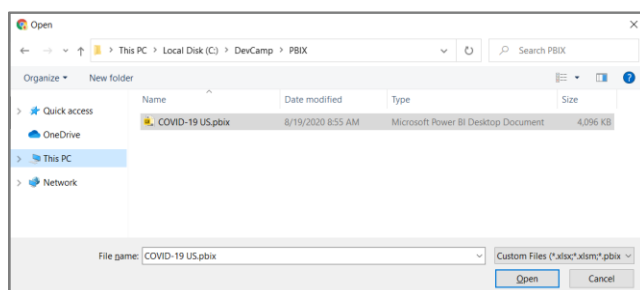
3. Upload a PBIX file to create a new report and dataset.

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

   b) Click the **Get** button in the **Files** section.
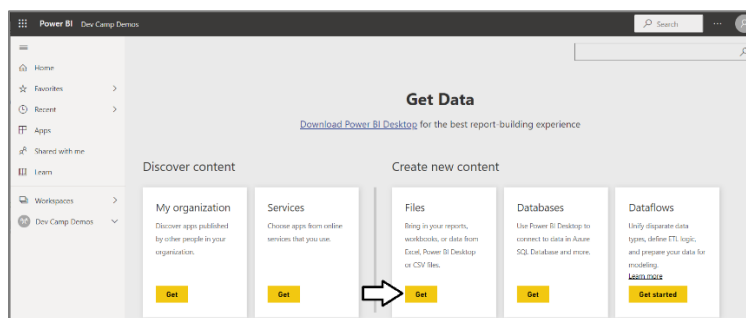
   c) Click on **Local File** in order to select a PBIX file that you have on your local hard drive.
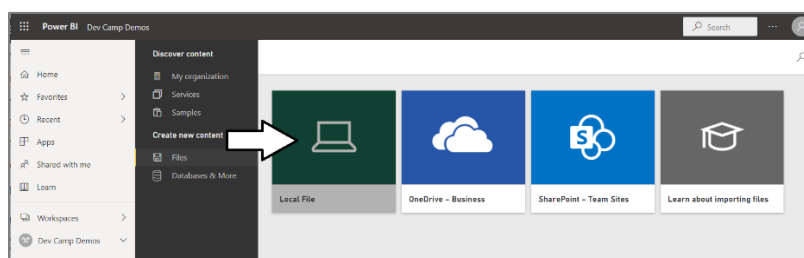
   d) Select the PBIX file and click the **Open** button to upload it to the Power BI Service.

e) The Power BI Service should have created a report and a dashboard 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.
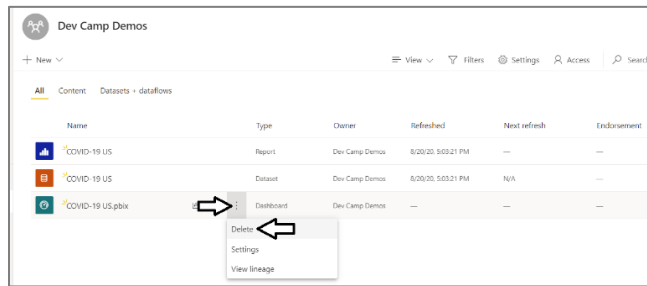
4. Open the report to see what it looks like when displayed in the Power BI Service.

a) Click on the report to open it.
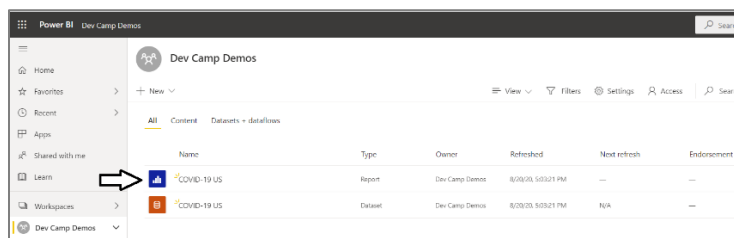
b) You should now be able to see the report.
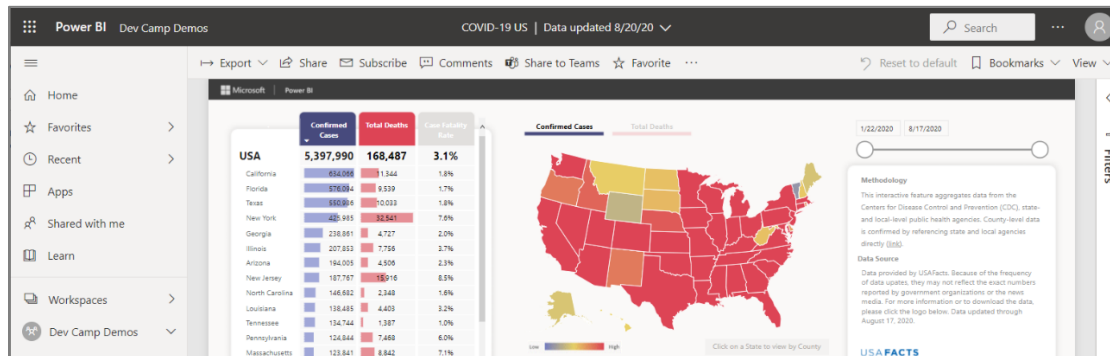
In the next step, you will find and record the GUID-based IDs for the report and its hosting workspace. You will then use these IDs later in this exercises when you first write the code to embed a report in the **UserOwnsData** web application.

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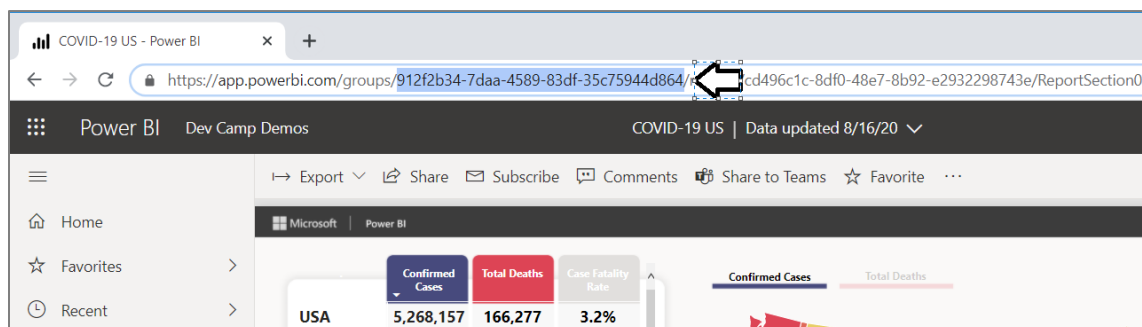
b) Open up a new text file in a program such as Notepad and paste in the value of the workspace ID.

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.
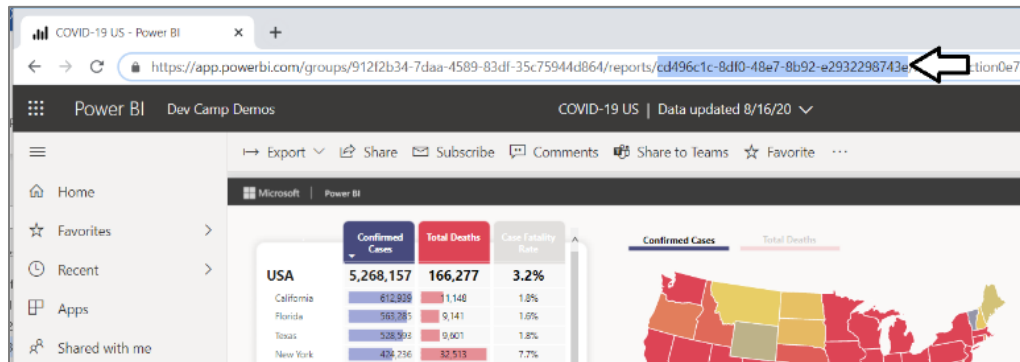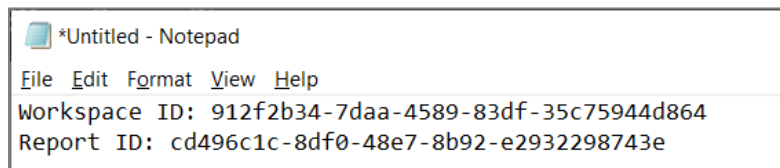

Leave the text file open for now. In a step later in this exercise, you will copy and paste these IDs into your C# code.

6. Create a new C# class named **PowerBiServiceApi** in which you will add code for calling the Power BI Service API.

a) Return to the **UserOwnsData** project in Visual Studio Code.

b) Create a new top-level folder in the **UserOwnsData** project named **Services**.
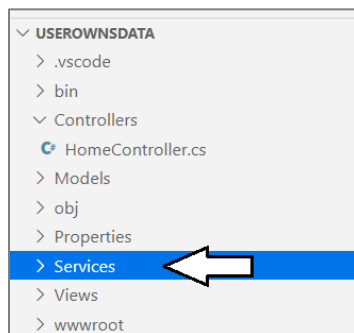
c) Inside the **Services** folder, create a new C# source file named **PowerBiServiceApi.cs**.
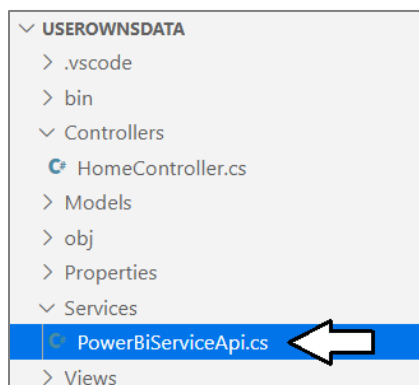
d) Copy and paste the following code into **PowerBiServiceApi.cs** to provide a starting point.

```
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 Newtonsoft.Json;

namespace UserOwnsData.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 class to pass the data needed to embed a single report. You will use this class later in this lab to pass report embedding data 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 Core dependency injection pattern. When your class needs to use a service, you can simply add a constructor parameter for that service and the .NET Core runtime takes care of passing the service instance at run time. In this case, the constructor is injecting an instance of the .NET Core 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 static readonly string[] RequiredScopes = new string[] {
  "https://analysis.windows.net/powerbi/api/Group.Read.All",
  "https://analysis.windows.net/powerbi/api/Report.ReadWrite.All",
  "https://analysis.windows.net/powerbi/api/Dataset.ReadWrite.All",
  "https://analysis.windows.net/powerbi/api/Content.Create",
  "https://analysis.windows.net/powerbi/api/Workspace.ReadWrite.All"
};
```

The **RequiredScopes** field holds a string array containing a set of delegated permissions supported by the Power BI Service API. When your application calls across the network to acquire an access token from Azure AD, it will pass this set of delegated permissions so that Azure AD knows which delegated permissions should be included with the access token it returns.

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

```
public string GetAccessToken() {
  return this.tokenAcquisition.GetAccessTokenForUserAsync(RequiredScopes).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);

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

}
```

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

Note that **Exercise 3 - PowerBiServiceApi.cs.txt** in the **StudentLanFiles** folder contains the final code for **PowerBiServiceApi.cs** in exercises 3 in case you want to copy-and-paste the code or verify what you have in your project is correct.

7. 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 UserOwnsData.Services;
```

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

```
public void ConfigureServices(IServiceCollection services) {

  services.AddMicrosoftWebAppAuthentication(Configuration);
```

d) Replace the call to **services.AddMicrosoftWebAppAuthentication** with the following code.

```
services
  .AddMicrosoftWebAppAuthentication(Configuration)
  .AddMicrosoftWebAppCallsWebApi(Configuration, PowerBiServiceApi.RequiredScopes)
  .AddInMemoryTokenCaches();
```

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

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

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

```
public void ConfigureServices(IServiceCollection services) {

  services
    .AddMicrosoftWebAppAuthentication(Configuration)
    .AddMicrosoftWebAppCallsWebApi(Configuration, PowerBiServiceApi.RequiredScopes)
    .AddInMemoryTokenCaches();

  services.AddScoped(typeof(PowerBiServiceApi));

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

  mvcBuilder.AddMicrosoftIdentityUI();

  services.AddRazorPages();

}
```

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 access tokens and refresh 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 dependency injection.

8. 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 **UserOwnsData.Services** namespace.

```
using UserOwnsData.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 Core 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() {

  Guid workspaceId = new Guid("912f2b34-7daa-4589-83df-35c75944d864");
  Guid reportId = new Guid("cd496c1c-8df0-48e7-8b92-e2932298743e");

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

  return View(viewModel);
}
```

9. 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.
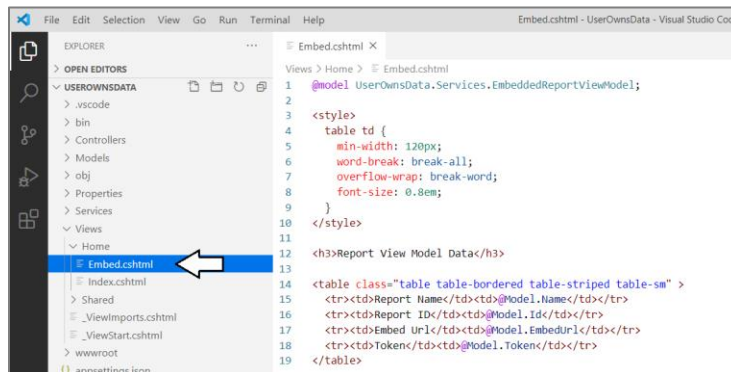
```
@model UserOwnsData.Services.EmbeddedReportViewModel;

<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>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**.

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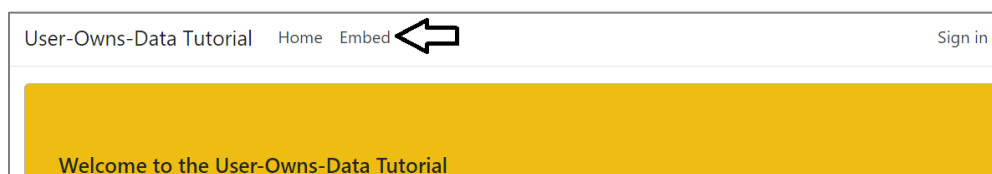
d) If you are prompted to enter your credentials, enter your user name and password and log in.

e) After you have authenticated for the first time, you should be prompted with a **Permissions Requested** dialog.

f) Click the **Accept** button to consent to the application using the requested delegated permissions on your behalf.

The **Permissions requested** dialog is only shown to each user during the first successful login. Once a user clicks **Accept**, Azure AD remembers that the user has consented to the required permissions and does not need to prompt the user about permission requests.

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

You are now half way in your development efforts to embed a Power BI report. You have added to code to call the Power BI Service API which has made it possible to retrieve the data required to embed a report and to make this report embedding data available behind a web page. In the next exercise, you will complete the implementation by adding client-side code which programs against the Power BI JavaScript API to actually embed the report in this web page.

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

Exercise 4: Embedding 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 UserOwnsData.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 UserOwnsData.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**.
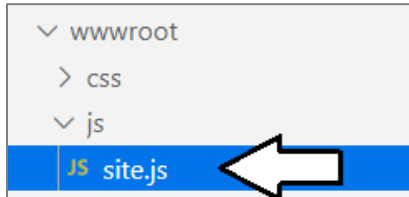
   d)   Rename **site.js** to **embed.js**.
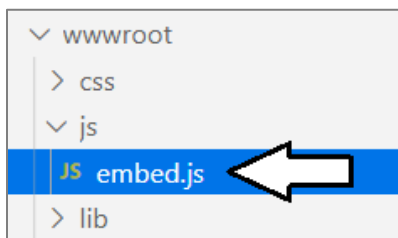
3.   Add the JavaScript code to **embed.js** to embed a report.
   a)   Open **embed.js** in an editor window.
   b)   Delete whatever content exists inside **embed.js**.
   c)   Paste the following code into **embed.js** to provide a starting point.

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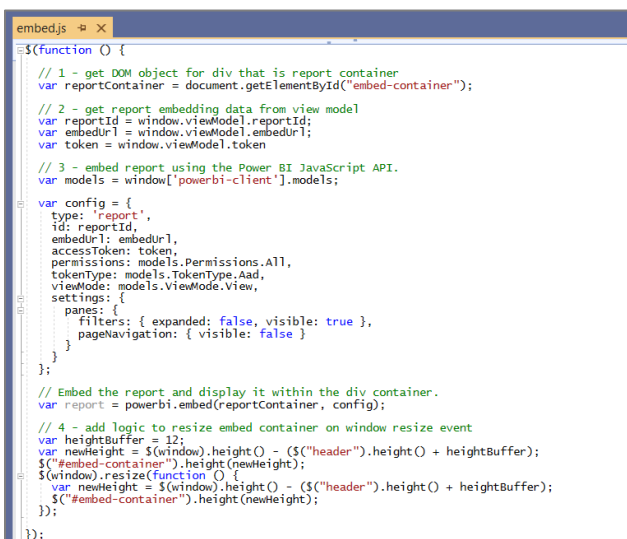
Note that the variable named **models** is initialized using a call to **window['powerbi-client'].models**. The **models** variable is used to set configuration values such as **models.Permissions.All**, **models.TokenType.Aad** and **models.ViewMode.View**.

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 **UserOwnsData** 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.
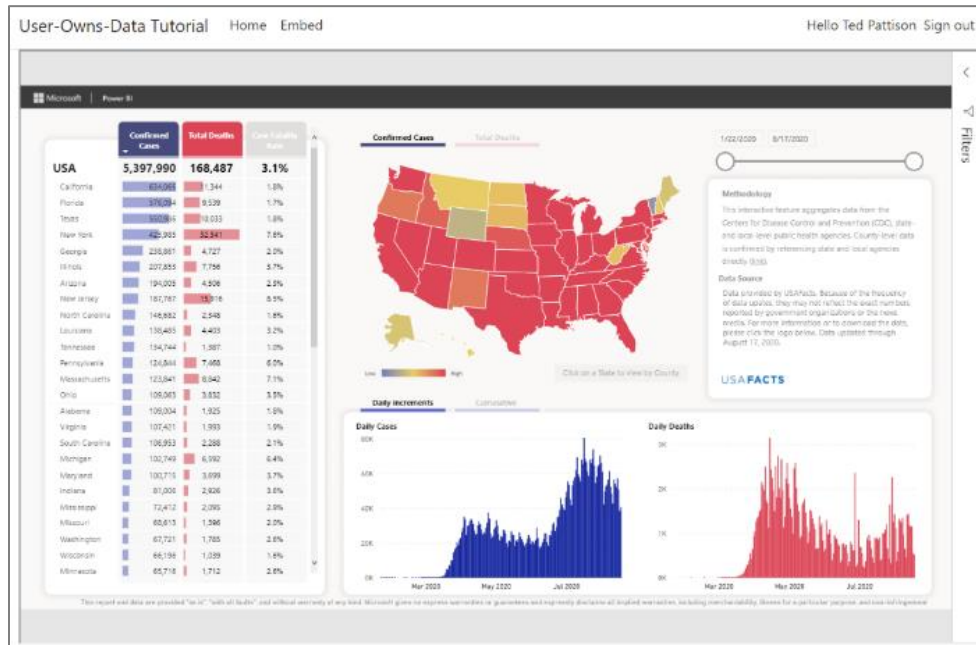
   e) Try resizing the browser window. The embedded report should continually adapt to the size of the window.
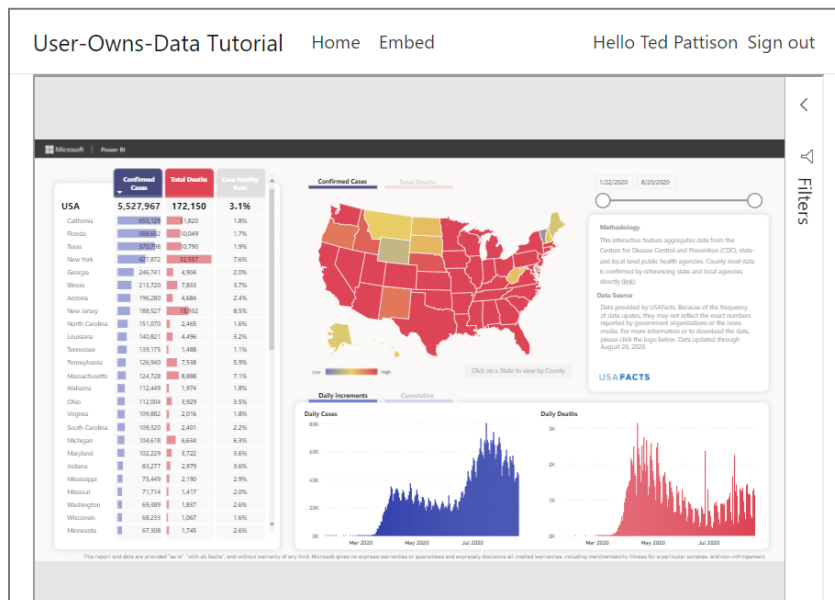
    f) You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

> 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 so you can receive the benefits of strong typing, compile-time type checks and much better IntelliSense.

## Exercise 5: Adding TypeScript Support to a .NET Core 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 and that the Node Package Manager application named **npm** is available at the commend line. You will begin by adding several Node.js configuration files to the root folder of the **UserOwnsData** 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 **UserOwnsData** 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 **UserOwnsData** project.
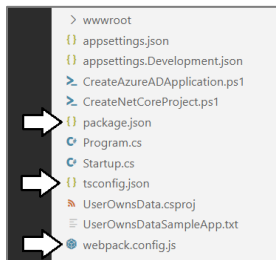

Visual Studio Code makes it difficult to add existing files to a project folder. You can use the Windows Explorer to copy these three files from the **StudentLabFiles** folder to the **UserOwnsData** project folder.

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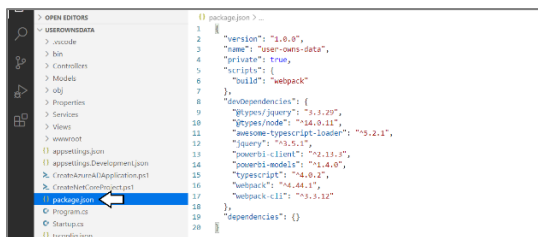
    b) Open the Visual Studio Code terminal by clicking the **View > Terminal** menu command or by using **Ctrl+`** keyboard shortcut.

    c) Run the **npm install** command to restore the list of Node.js packages.
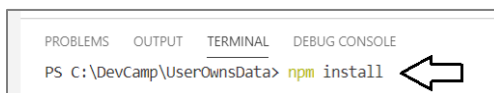
    d) When you run the **npm install** command, **npm** will download all the Node.js packages into the **node_modules** folder.
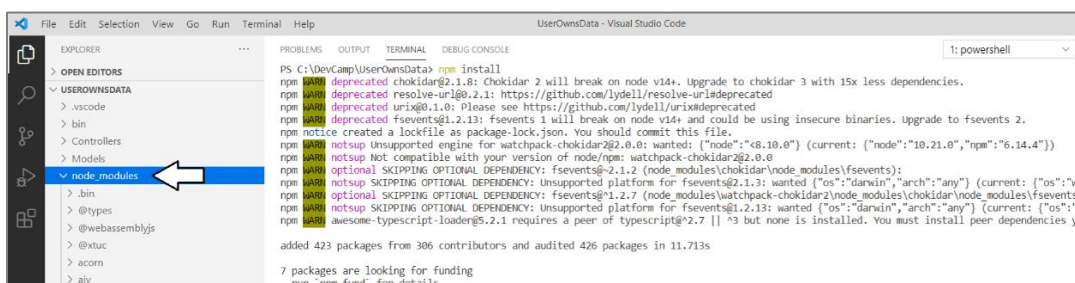
3. Take a quick look at the **tsconfig.json** file.

   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 **UserOwnsData** project folder, create a new top-level folder named **Scripts**.

   b) Create a new file inside the **Scripts** folder named **embed.ts**.
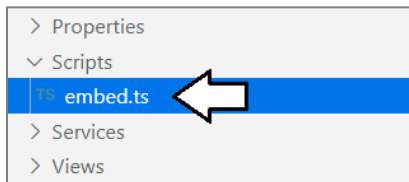
   c) In Windows Explorer, locate the **Exercise 5 - embed.ts.txt** file in the **StudentLabFiles** folder.

   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 content of **Exercise 5 - embed.ts.txt** 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.
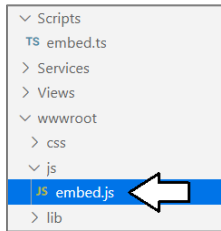
   b) Open the Visual Studio Code terminal by clicking the **View > Terminal** menu command or by using **Ctrl+`** keyboard shortcut.

   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.
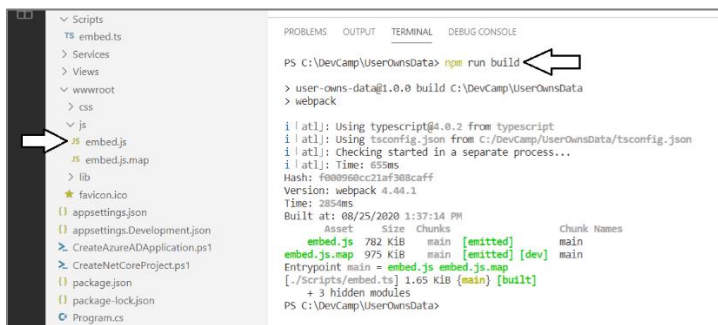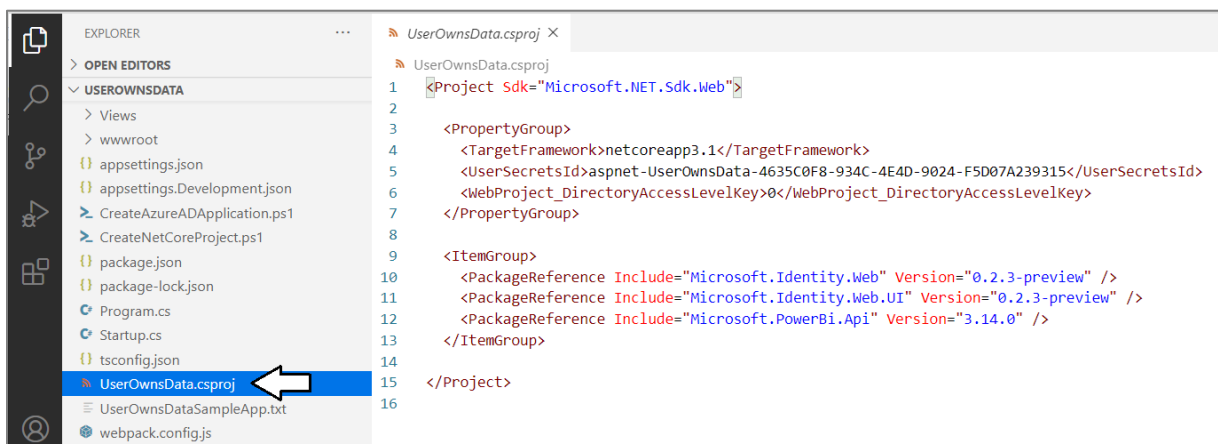

   e) Open the new version of **embed.js**. You should see it is a source file generated by the webpack utility.
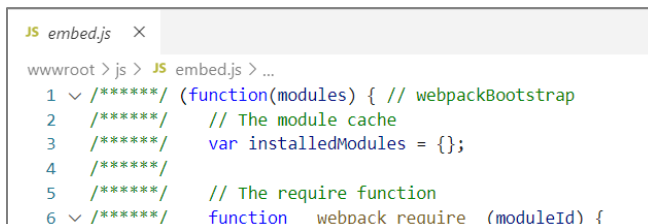
   f) Close **embed.js** without saving any changes.

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

   a) Open the .NET Core project file **UserOwnsData.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.

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <UserSecretsId>aspnet-UserOwnsData-87660A42-54AC-4CF9-8583-B31608FED004</UserSecretsId>
    <WebProject_DirectoryAccessLevelKey>0</WebProject_DirectoryAccessLevelKey>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Identity.Web" Version="0.2.3-preview" />
    <PackageReference Include="Microsoft.Identity.Web.UI" Version="0.2.3-preview" />
    <PackageReference Include="Microsoft.PowerBi.Api" Version="3.14.0" />
  </ItemGroup>

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

</Project>
```
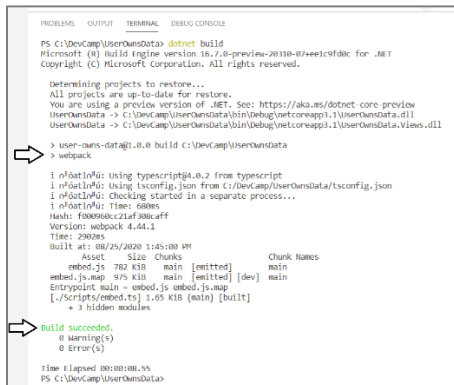
c) Save your changes and close **UserOwnsData.csproj**.

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.

   c) You should now be able to navigate to the **Embed** page and see the Power BI report displayed on the page.
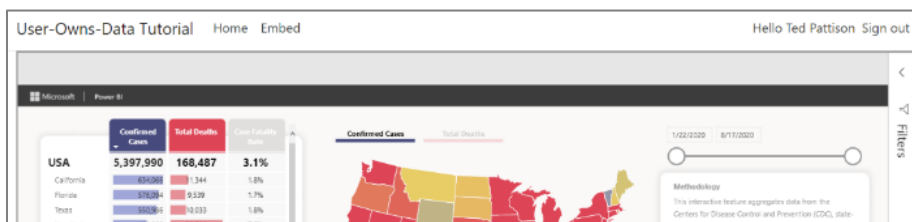
> When you test the **UserOwnsData** web application, it should behave just as it did when you tested it in Exercise 4. The difference is that now the client-side behavior is now implemented with TypeScript instead of JavaScript.

Exercise 6: Creating a View Model for App Workspaces

Up to this point, you have implemented the **UserOwnsData** 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 **UserOwnsData** 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 = "") {

  var accessToken = this.tokenAcquisition.GetAccessTokenForUserAsync(RequiredScopes).Result;
  var tokenCredentials = new TokenCredentials (accessToken, "Bearer");
  PowerBIClient pbiClient = new PowerBIClient (new Uri (urlPowerBiServiceApiRoot), tokenCredentials);

  Object viewModel;
  if (string.IsNullOrEmpty (appWorkspaceId)) {
    viewModel = new {
      currentWorkspace = "My Workspace",
      workspaces = ( await pbiClient.Groups.GetGroupsAsync() ).Value,
      datasets = ( await pbiClient.Datasets.GetDatasetsAsync() ).Value,
      reports = ( await pbiClient.Reports.GetReportsAsync() ).Value,
      token = accessToken
    };
  } else {
    Guid workspaceId = new Guid (appWorkspaceId);
    var workspaces = (await pbiClient.Groups.GetGroupsAsync ()).Value;
    var currentWorkspace = workspaces.First ((workspace) => workspace.Id == workspaceId);
    viewModel = new {
      workspaces = workspaces,
        currentWorkspace = currentWorkspace.Name,
        currentWorkspaceIsReadOnly = currentWorkspace.IsReadOnly,
        datasets = (await pbiClient.Datasets.GetDatasetsInGroupAsync (workspaceId)).Value,
        reports = (await pbiClient.Reports.GetReportsInGroupAsync (workspaceId)).Value,
        token = accessToken
    };
  }

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

The **GetEmbeddedViewModel** method accepts an **appWorkspaceId** parameter and returns a string value with JSON-formatted data. If the **appWorkspaceId** parameter is blank, the **GetEmbeddedViewModel** method returns a view model for the current user's personal workspace. 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) To enhance your conceptual understanding, examine a sample of JSON returned by the **GetEmbeddedViewModel** method.

```
{
  "workspaces": [
    { "id": "6679bd47-5b5f-4be0-ac6d-7a7ab1ba16f8", "name": "All Company", "isReadOn
    { "id": "912f2b34-7daa-4589-83df-35c75944d864", "name": "Dev Camp Demos", "isRea
  ],
  "currentWorkspace": "Dev Camp Demos",
  "currentWorkspaceIsReadOnly": false,
  "datasets": [
    { "id": "94e863ea-9117-445c-ac9e-db9373bdb8ea", "name": "COVID-19 US" },
    { "id": "89795189-0f43-4057-9af7-ab4838082a3b", "name": "Microsoft Graph Report"
    { "id": "bfa46105-5d6e-4270-88d8-eb6be5bf57ad", "name": "Wingtip Sales Analysis"
  ],
  "reports": [
    { "id": "cd496c1c-8df0-48e7-8b92-e2932298743e", "name": "COVID-19 US", "webUrl":
    { "id": "23d56559-af3c-4c51-9175-2904ee76bbae", "name": "Microsoft Graph Report"
    { "id": "fa89b3ae-c6a6-4e0c-8e4b-be7057cf583c", "name": "Wingtip Sales Analysis"
  ],
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6ImppYk5ia0ZTU2JteFBZck45Q0Zx
};
```

    d) 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.

```
public async Task<IActionResult> Embed() {
  Guid workspaceId = new Guid("912f2b34-7daa-4589-83df-35c75944d864");
  Guid reportId = new Guid("cd496c1c-8df0-48e7-8b92-e2932298743e");
  var viewModel = await powerBiServiceApi.GetReport(workspaceId, reportId);
  return View(viewModel);
}
```

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

```
public async Task<IActionResult> Embed(string workspaceId) {
  var viewModel = await powerBiServiceApi.GetEmbeddedViewModel(workspaceId);
  // cast string value to object type in order to pass string value as MVC view model
  return View(viewModel as object);
}
```

   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 content of **Exercise 6 - Embed.cshtml.txt** into **Embed.cshtml.**
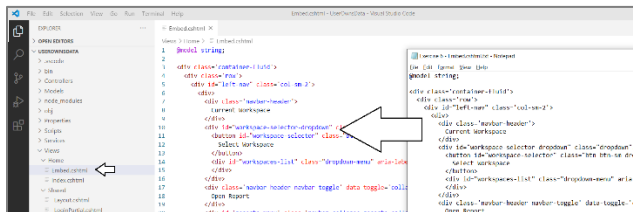

   e) Save your changes and close **Embed.cshtml.**

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.**
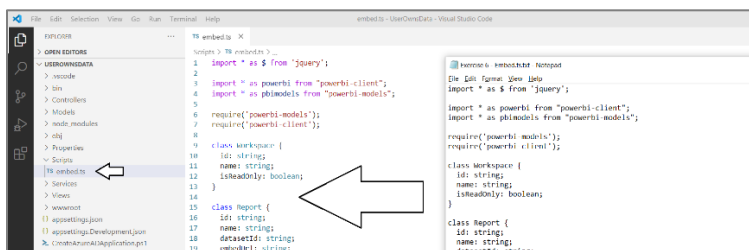

   e) Save your changes and close **Embed.cshtml.**

5. 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.

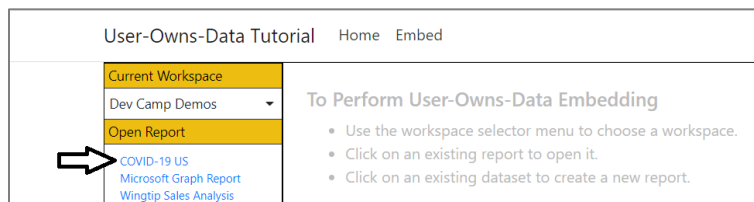   c) The **Embed** page should appear much differently than before as shown in the following screenshot.
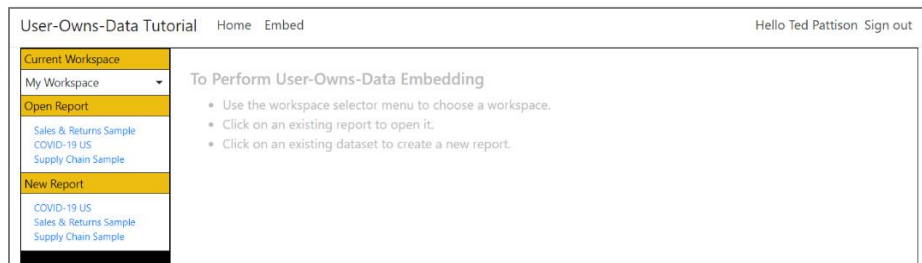
> Note there is a dropdown list for the **Current Workspace** that you can use to navigate across workspaces.

   d) Navigate to the workspace you created earlier in this lab.
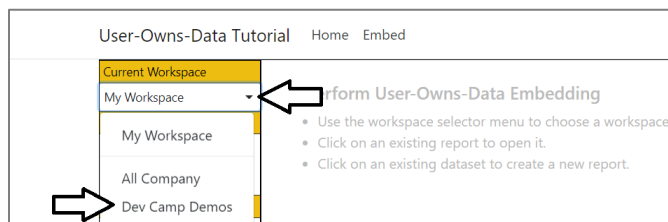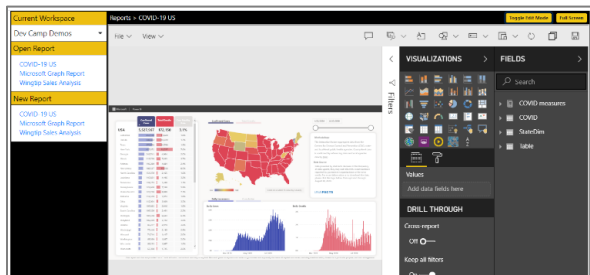

   e) Click on a report in the **Open Report** section.
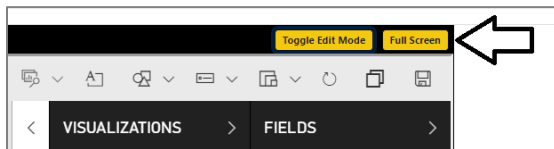

   f) The report should open in read-only mode.
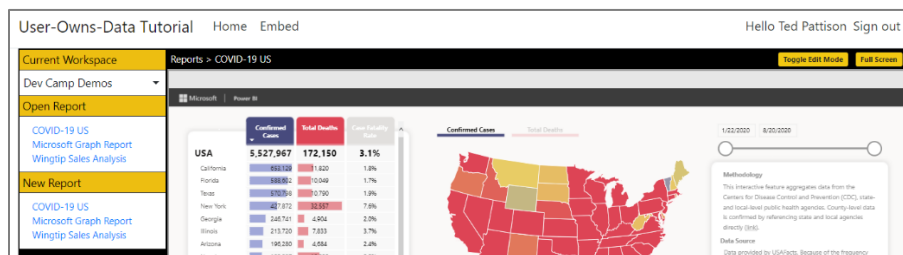
   g) Click the **Toggle Edit Mode** button to move the report into edit mode.
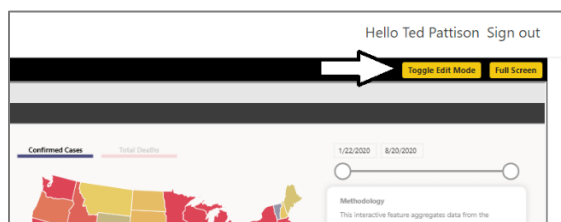
h) Note that when the report goes into edit mode, there isn't much space to work on the report while editing.
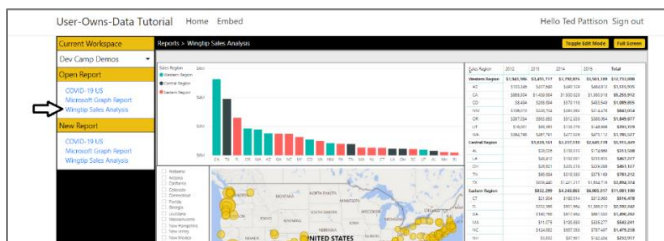

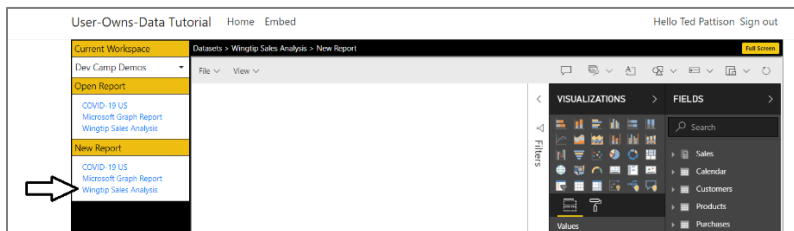i) Click the Full Screen button to enter full screen mode

You can invoke the **File > Save** command in a report that is in edit mode to save your changes.
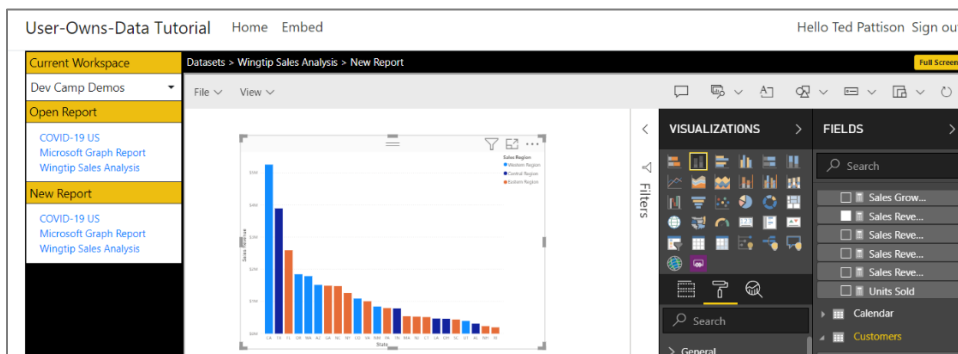
j) Press the **Esc** key in the keyboard to exit full screen mode.

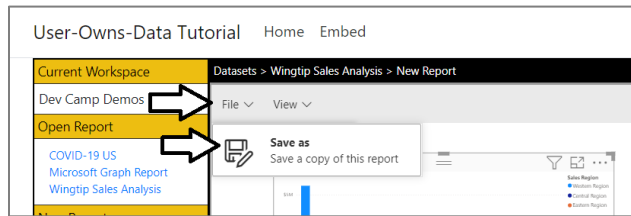k) Click on a second report in the **Open Report** section to navigate between reports.

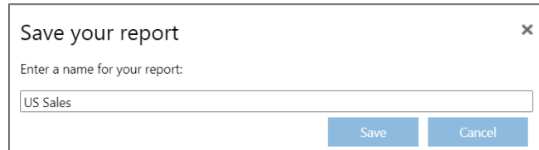l) Create a new report by clicking on a dataset name in the **New Report** section.
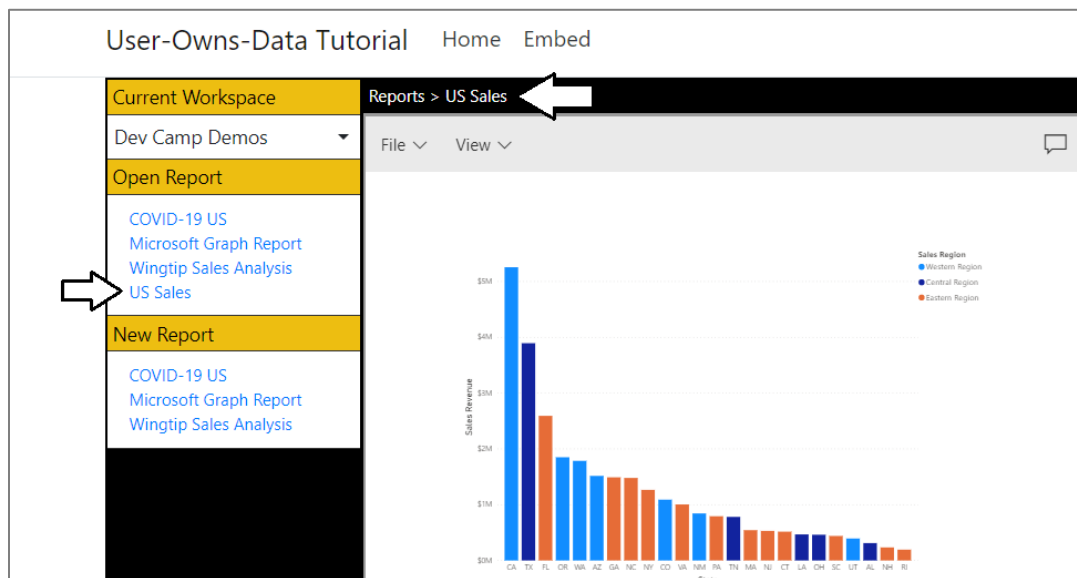

m) Add a simple visual of any type to the new report.

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

o) Give your new report a name.

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

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

> If you have time, please fill out the post tutorial survey. Navigate to this survey form and provide whatever feedback you are willing to give us on how we can make Power BI Dev Camp better.

Campers Challenge: Extend the UserOwnsData Project with Extra Functionality

> At Power BI Dev Camp, we like to run a competition after each session to let our campers showcase their talents to their peers. In this month's student challenge, we like you to use the final solution to the **UserOwnsData** web application as your starting point and further customize this project with whatever features and functionality that you can dream up.

1. Competition rules

    a) Extend the final solution of the **UserOwnsData** web application with your own set of features.

    b) Create a 3-5 minute youtube video of you demonstrating the app and discussing what you did.

    c) Fill out the competition entry form and submit it before the deadline of Friday September 18th.

    d) We will pick a winner and runner up by the next Power BI Dev Camp session on Thursday, Sept 24th.

    e) Good luck and may .NET Core be with you.