## Developing for User-Owns-Data Embedding using .NET Core

In this lab, you will create a new .NET Core project and go through all 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. Then you will work through creating 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 exercise of the lab, you will support for npm, TypeScript and webpack so that you can move the client-side code from JavaScript to TypeScript where 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](https://docs.microsoft.com/en-us/powershell/azure/active-directory/install-adv2?view=azureadps-2.0)]

2) **DOTNET Core SDK 3.1 or later** – [[download](https://dotnet.microsoft.com/download)]

3) **Node.js** – [[download](https://nodejs.org/en/download/)]

4) **Visual Studio Code** – [[download](https://code.visualstudio.com/Download)]

5) **Visual Studio 2019 (optional)** – [[download](https://visualstudio.microsoft.com/downloads/)]

Please refer to the setup document for this lab if you need more detail on how to configure your developer workstation.

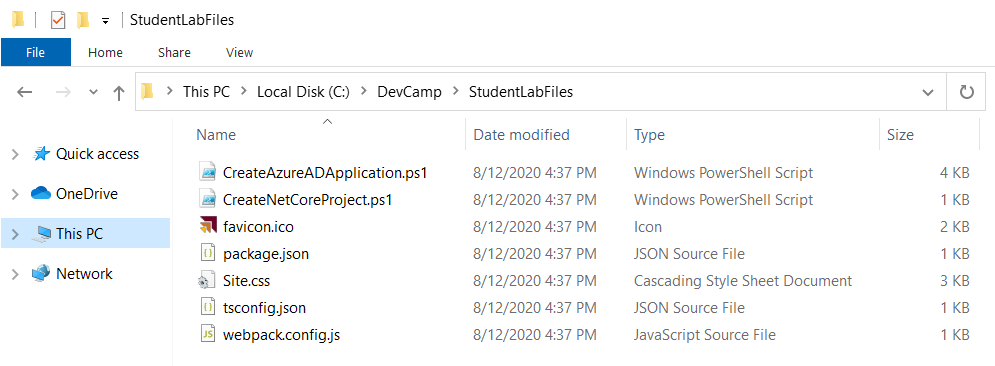
### Exercise 1: Create a New .NET Core MVC Web Application Project

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.

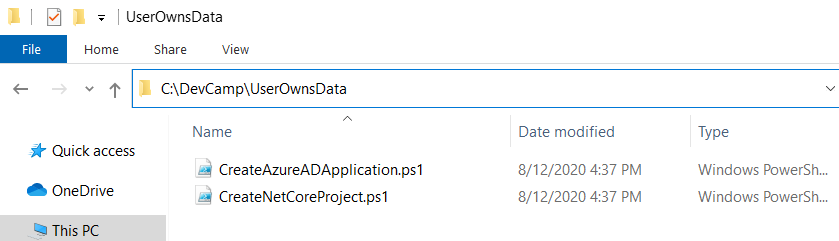
1. Download the student lab files to a local folder on your developer workstation.
   1. Create a new top-level folder on your workstation named **DevCamp** at a location such as **c:\DevCamp**.
   2. Download the ZIP archive with the student lab files from GitHub by clicking the following link.

<https://github.com/TedPattison/NetCore-UserOwnsData-Tutorial/raw/master/StudentLabFiles.zip>

* 1. Extract the **StudentLabFiles** folder from **StudentLabFiles.zip** into a to a local folder such as **c:\DevCamp\StudentLabFiles**.
  2. The **StudentLabFiles** folder should contain the set of files shown in the following screenshot.



1. Create a new .NET Core project using the .NET Core CLI and a PowerShell script.
   1. Create a new folder on your local drive named **UserOwnsData** at a location such as **c:\DevCamp\UserOwnsData**.
   2. In the **StudentLabFiles** folder, locate the scripts named **CreateNetCoreProject.ps1** and **CreateAzureADApplication.ps1**.
   3. Copy **CreateNetCoreProject.ps1** and **CreateAzureADApplication.ps1** into the **UserOwnsData** folder.



1. Review the PowerShell code in **CreateNetCoreProject.ps1**.
   1. Open **CreateNetCoreProject.ps1** in a text editor such asNotepad or the PowerShell Integrated Scripting Environment (ISE).
   2. Review the code in **CreateNetCoreProject.ps1** which creates a new .NET Core project and add a few .NuGet packages.

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

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

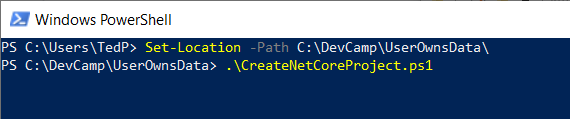
# update to latest available version of Microsoft.Identity.Web

dotnet add package Microsoft.Identity.Web -v 0.2.3-preview

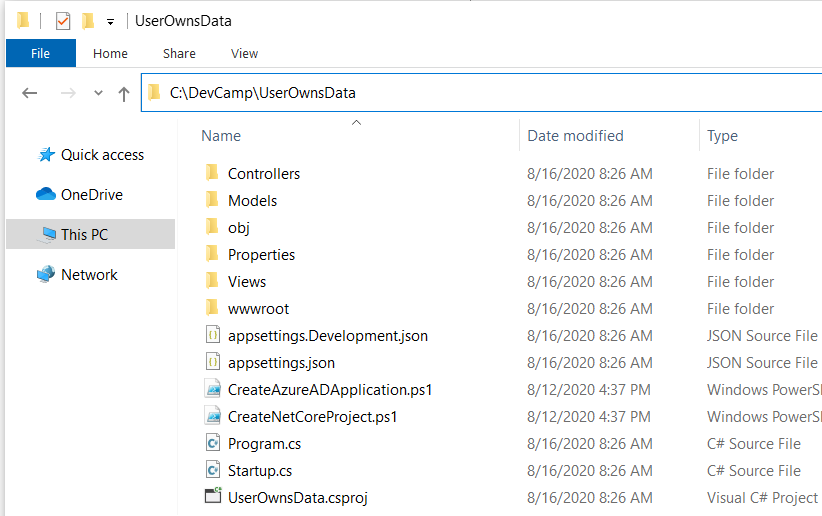
dotnet add package Microsoft.Identity.Web.UI -v 0.2.3-preview

dotnet add package Microsoft.PowerBi.Api

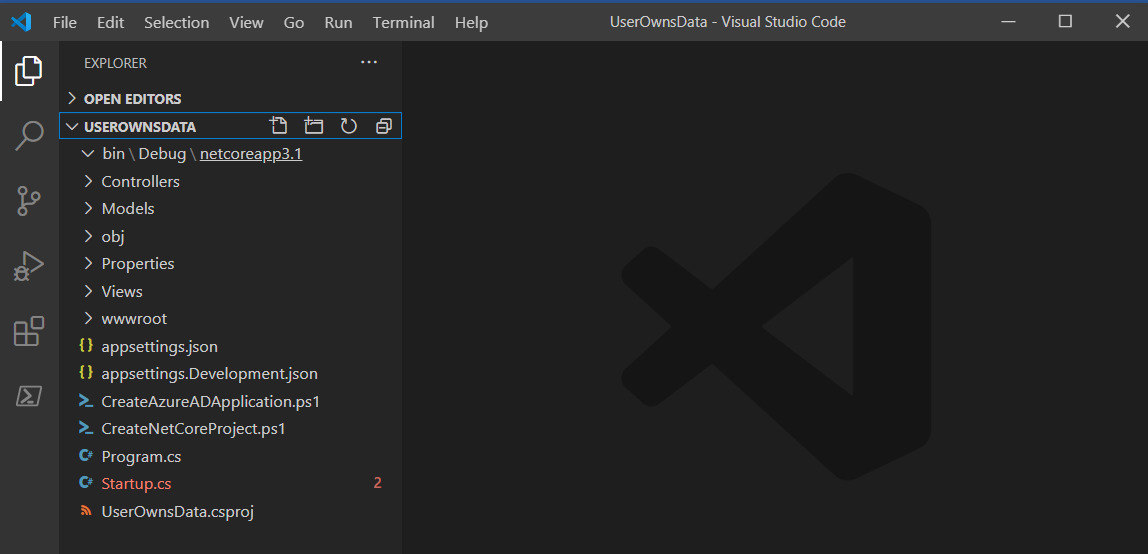
* 1. Open up a PowerShell console and set the location of the command prompt to the **UserOwnsData** folder.
  2. Execute the script **CreateNetCoreProject.ps1** and issuing the command **.\CreateNetCoreProject.ps1**.



* 1. Once the script has completed, you should see that the **UserOwnsData** folder has been populated with project files.

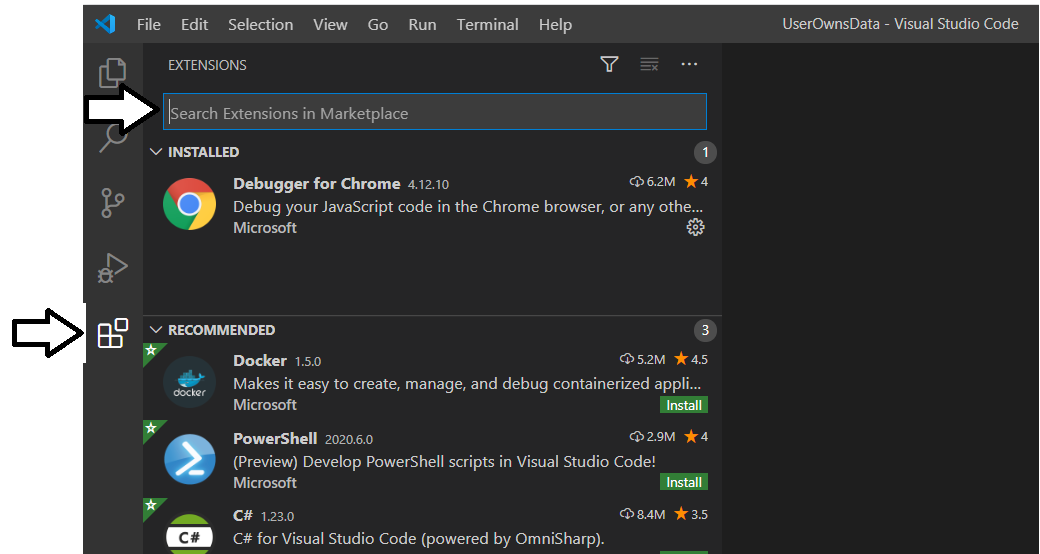


1. Open the **UserOwnsData** folder with Visual Studio Code
   1. Launch Visual Studio Code.
   2. Use the **Open Folder** command in Visual Studio Code to open the **UserOwnsData** folder.

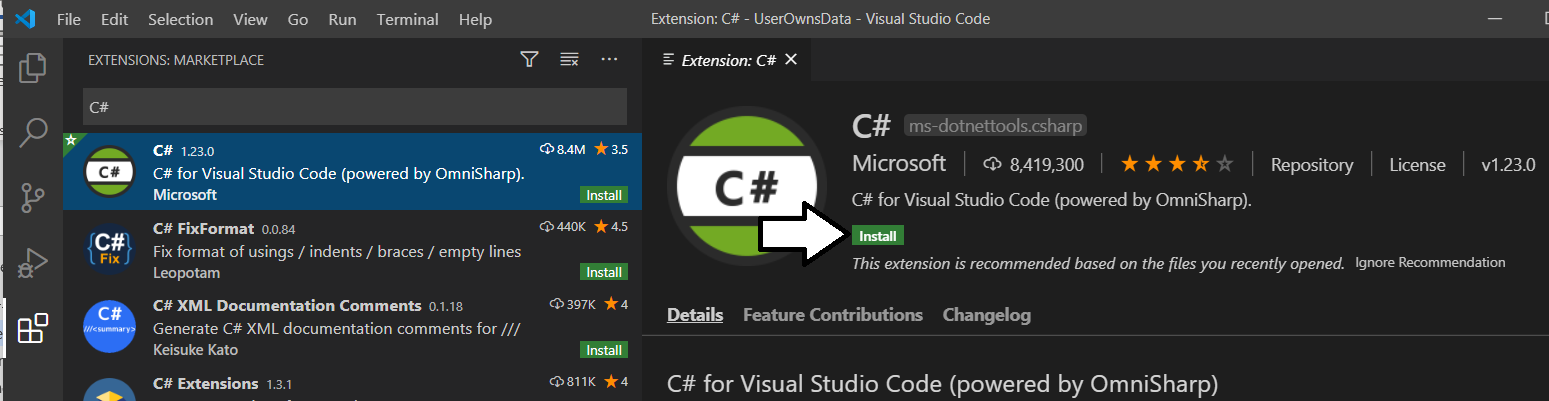


You will not be able to build the project yet. That is because the PowerShell script removed the .NuGet package for an older authentication library named **Microsoft.AspNetCore.Authentication.AzureAD.UI** and added new packages for the new Microsoft authentication library **Microsoft.Identity.Web**. 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.

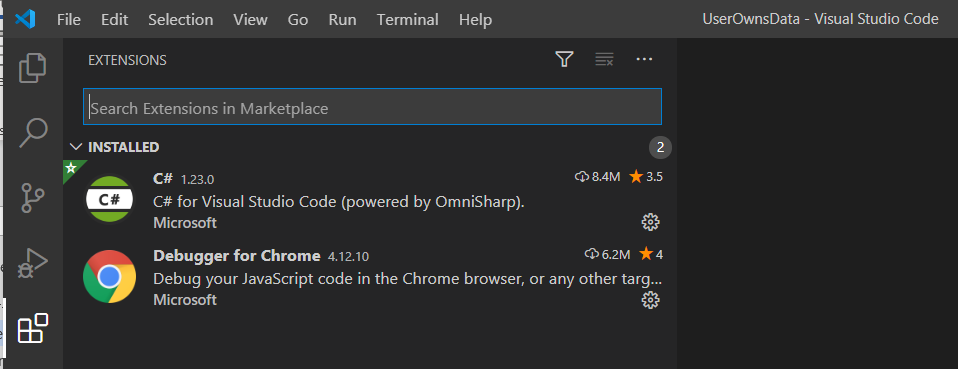
1. Configure Visual Studio Code with the extensions needed for .NET Core development.
   1. Click on the button at the bottom of the left navigation menu to display the **EXTENSION** pane.
   2. You should be able to see what extensions are currently installed.
   3. You should also be able to search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



* 1. Find and install the **C#** extension from Microsoft if it is not already installed.



* 1. Find and install the **Debugger for Chrome** extension from Microsoft if it is not already installed.
  2. 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 to 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**.
   1. Open the PowerShell script named **CreateAzureADApplication.ps1** in a text editor such asNotepad or the PowerShell ISE.
   2. The script begin by **calling Connect-AzureAD** to establish a connection with Azure AD.

$authResult = Connect-AzureAD

* 1. 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**.

* 1. 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

* 1. 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 `

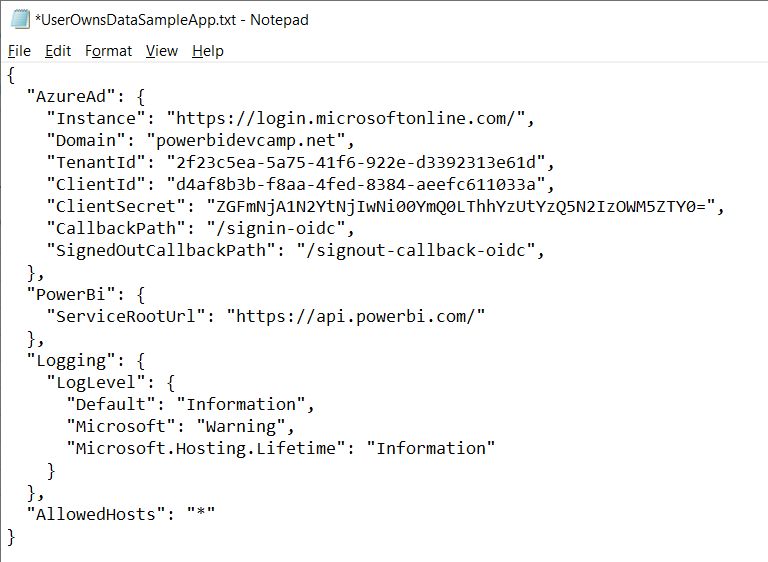
-AvailableToOtherTenants $false `

-ReplyUrls @($replyUrl) `

-Homepage $replyUrl `

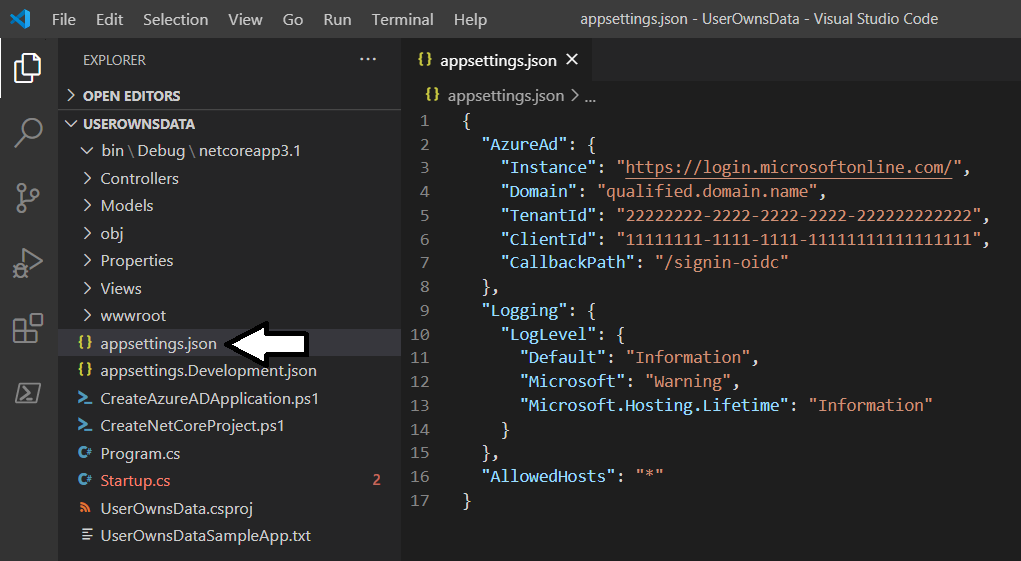
-PasswordCredentials $passwordCredential

* 1. Execute the PowerShell script named **CreateAzureADApplication.ps1**.
  2. When the PowerShell scriptruns successfully, it will create and open a text file named **UserOwnsDataSampleApp.txt**.

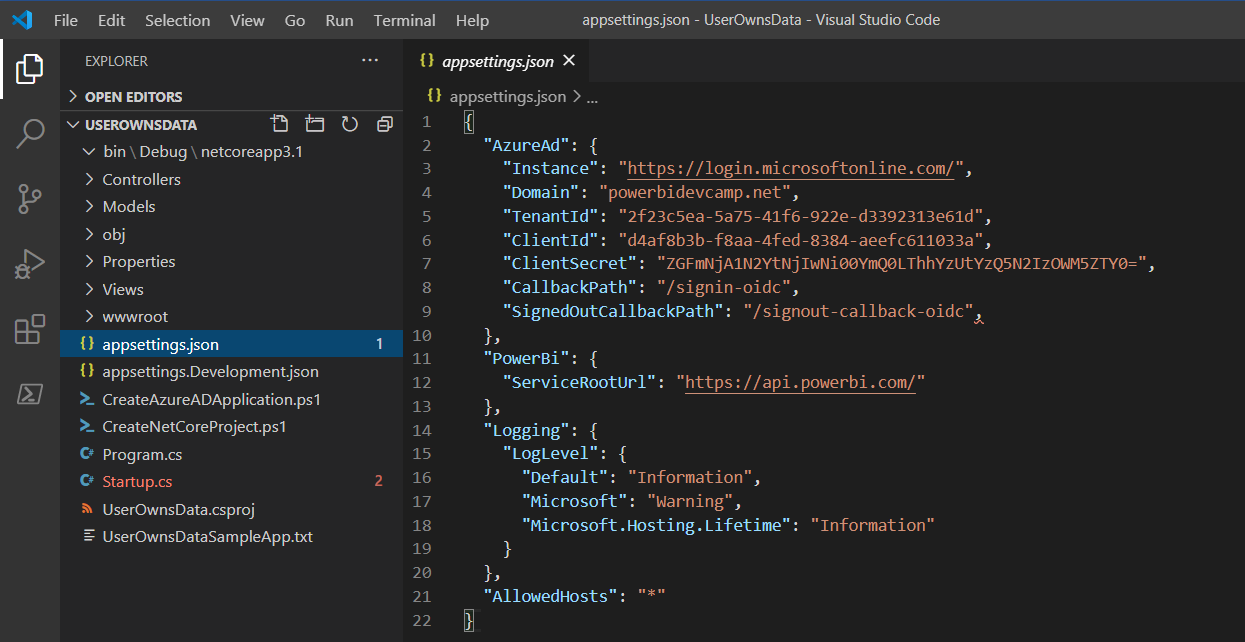


The text file **UserOwnsDataSampleApp.txt** contains JSON configuration that you will copy and paste into **appsettings.json**.

1. Copy the JSON in **UserOwnsDataSampleApp.txt** into the **appsettings.json** file in your project.
   1. Return to the **UserOwnsData** project in Visual Studio Code and open the **appsettings.json** file.
   2. The **appsettings.json** file should initially appear like the screenshot below.

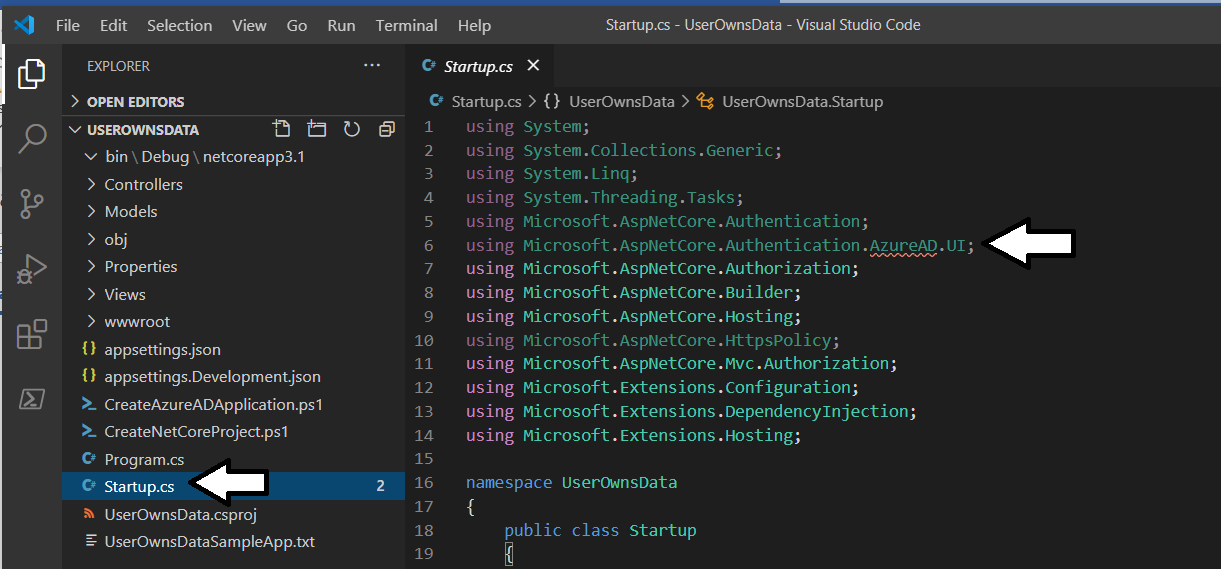


* 1. Delete the contents of **appsettings.json** and replace it by copying and pasting the contents of **UserOwnsDataSampleApp.txt**

****

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..

1. Modify the code in **Startup.cs** to properly initialize the authentication service provided by **Microsoft.Identoty.Web**.
   1. Open the **Startup.cs** file in an editor window.
   2. Remove line that imports **Microsoft.AspnetCore.Authentication.AzureAD.UI** which is causing a build error.



* 1. Please your cursor below the existing **using** statements and add the following **using** statements.

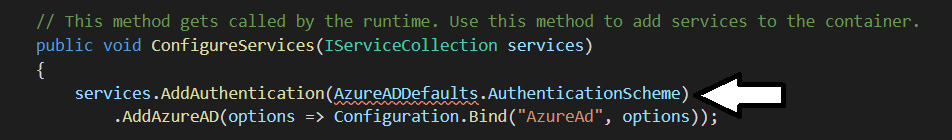
using Microsoft.Identity.Web;

using Microsoft.Identity.Web.UI;

using Microsoft.Identity.Web.TokenCacheProviders;

using Microsoft.Identity.Web.TokenCacheProviders.InMemory;

* 1. Move down into the **ConfigureServices** method and locate and delete the code which calls **services.AddAuthentication**.



* 1. 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);

* 1. 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));

});

* 1. 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();

* 1. 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));

});

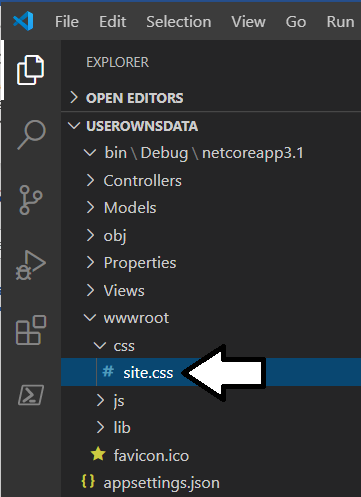
mvcBuilder.AddMicrosoftIdentityUI();

services.AddRazorPages();

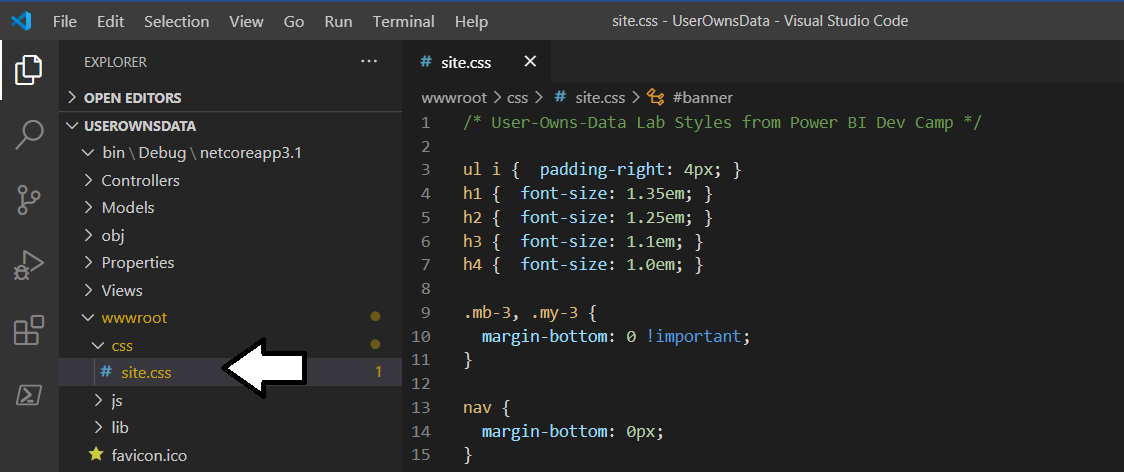
}

* 1. Save your changes and close **Startup.cs**.

1. Copy a pre-written set of CSS styles into the **UserOwnsData** project's **Site.css** file.
   1. Expand the **wwwroot** folder and then inside that expand the **css** folder to examine the contents of the **wwwroot/css** folder.
   2. Locate and open the CSS file that is automatically included in new .NET Core projects named **site.css**.

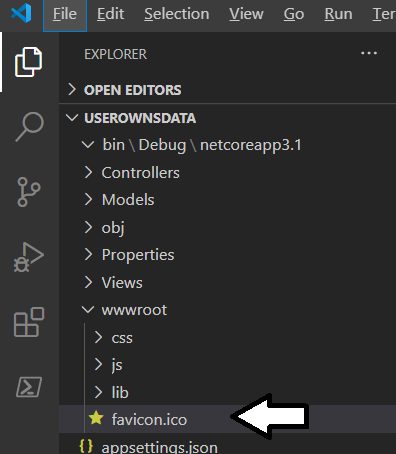


* 1. Using the Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **site.css**.
  2. Open the **site.css** file in the **StudentLabFiles** folder in a text editor and copy its entire contents into the Windows clipboard.
  3. Copy the CSS styles from the **site.css** file in the **StudentLabFiles** folder into the **site.css** file in your project.



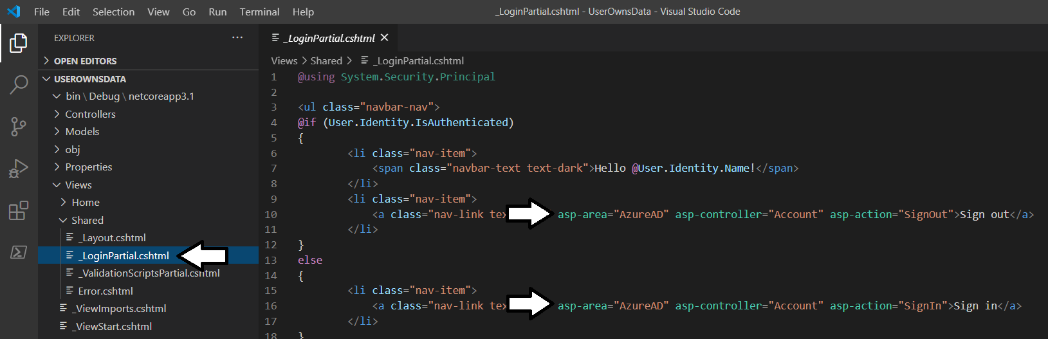
* 1. Save your changes and close **site.css**.

1. Copy a custom **favicon.ico** file to the **wwwroot** folder.
   1. Using the Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **favicon.ico**.
   2. 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.

1. Modify the partial razor view file named **\_LoginPartial.cshtml** to integrate with the **Microsoft.Identity.Web** authentication library..
   1. Expand the **Views > Shared** folder and locate the partial view named **\_LoginPartial.cshtml**.
   2. Open **\_LoginPartial.cshtml** in an editor window.
   3. 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**.

* 1. 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>

* 1. Save your changes and close **\_LoginPartial.cshtml**.

1. Modify the HTML in **Index.cshtml** to display differently depending on whether the user has logged in or not.
   1. Expand the **Views > Home** folder and locate the view file named **Index.cshtml**.
   2. Open **Index.cshtml** in an editor window.
   3. 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>ed

}

else {

<div class="jumbotron">

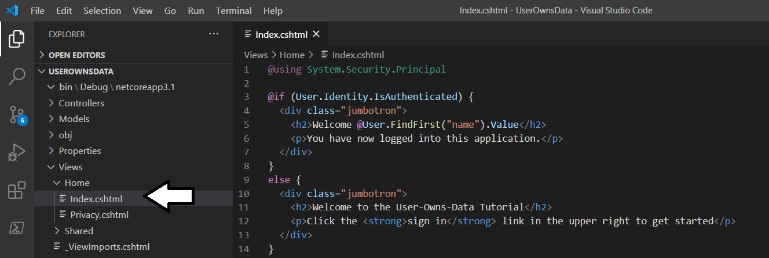
<h2>Welcome to the User-Owns-Data Tutorial</h2>

<p>Click the <strong>sign in</strong> link in the upper right to get started</p>

</div>

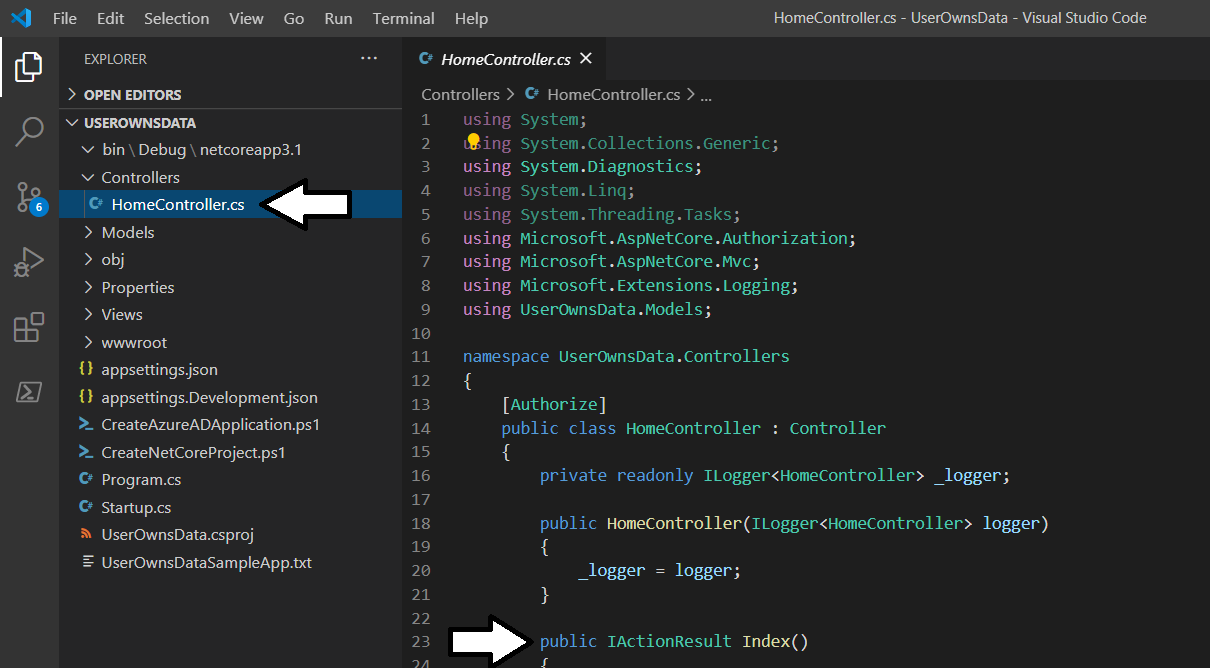
}

* 1. 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.

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



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

[AllowAnonymous]

public IActionResult Index()

{

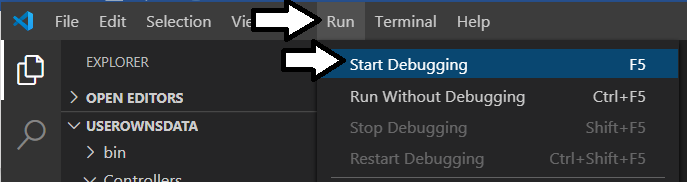
return View();

}

* 1. Save your changes and close **HomeController.cs**.

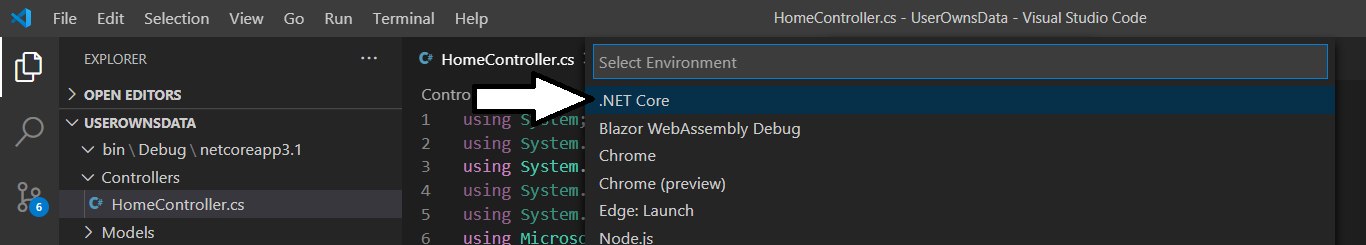
You have now modified the project to the point where you can now run the web application in the .NET Core debugger. In the next step, you will start the debugger so you can test your web application as it runs in the browser.

1. Test the **UserOwnsData** project by running it in the .NET Core debugging environment.
   1. Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.

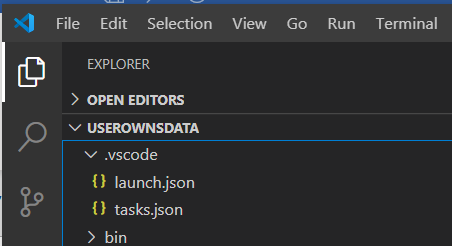


This will be the first time that you run this project in the Visual Studio Code debugger. However, running your project in the Visual Studio Code debugger requires a special file named launch.json. When you start the debugger in a project that currently has no launch.json file, Visual Studio Code will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

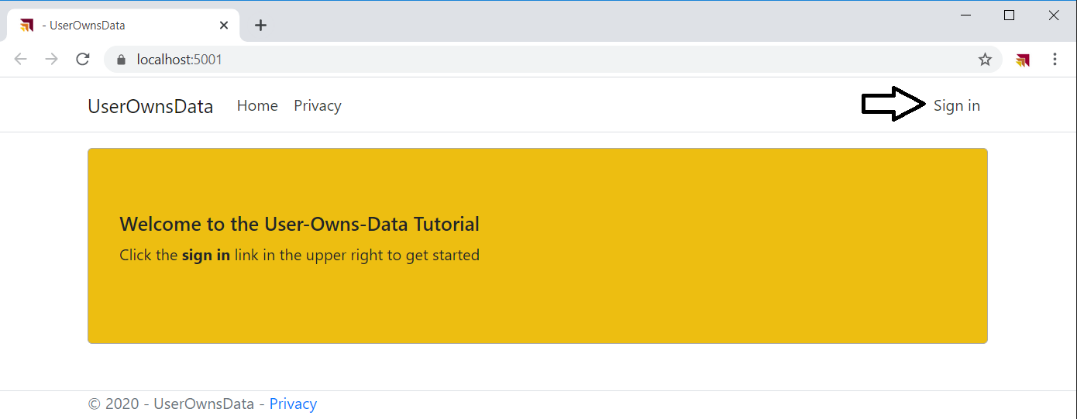
* 1. When prompted to **Select Environment**, select **.NET Core**.



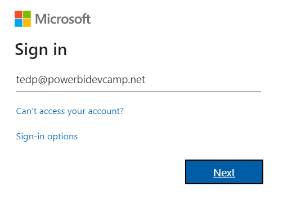
* 1. Behind the scenes, Visual Studio Code has created a folder named **.vscode** with two files **launch.json** and **tasks.json**.



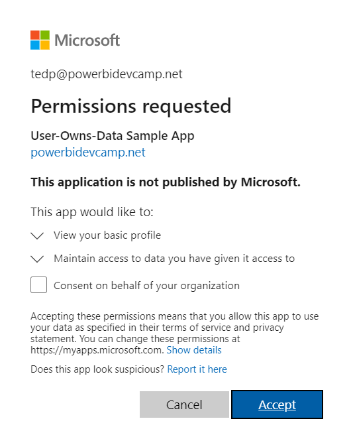
* 1. Once the debugging session has initialized, the browser should display the home page using anonymous access.
  2. Click the **Sign in** link to test put the user experience when authenticating with Azure AD.



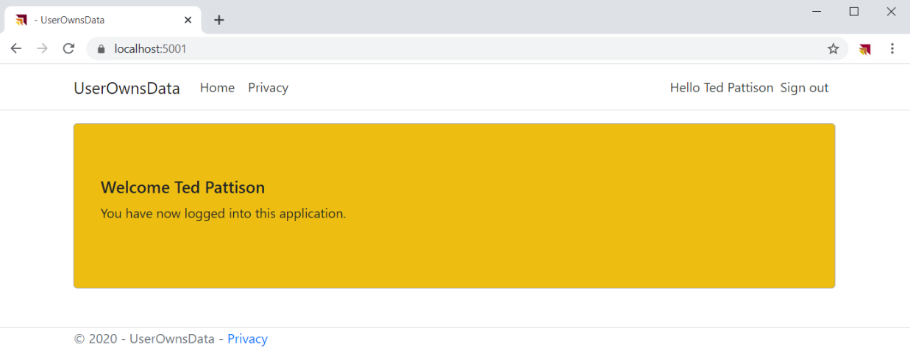
* 1. When prompted to **Sign in**, enter your user account and password to authenticate.



* 1. You should now be prompted by the **Permissions requested** dialog.
  2. Click the **Accept** button to grant the **User-Owns-Data Sample App** the required permissions it has requested.



* 1. After clicking the Accept button, you should be redirected back to the home page of the web application.

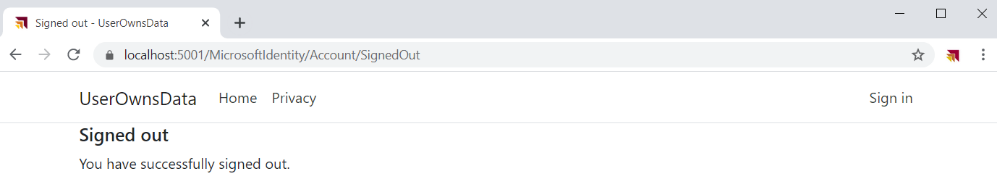


At this point, the user should be authenticated. For example, you should see the logged in user name to the left of the **Sign out** link in the top right corner. You should also see that the home page displays text that welcomes the user by name.

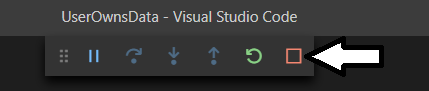
1. Test the user experience for logging out.
   1. Click the **Sign out** link to begin the logout experience.



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



* 1. 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.

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

[AllowAnonymous]

public IActionResult Index() {

return View();

}

public IActionResult Privacy() {

return View();

}

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

[AllowAnonymous]

public IActionResult Index() {

return View();

}

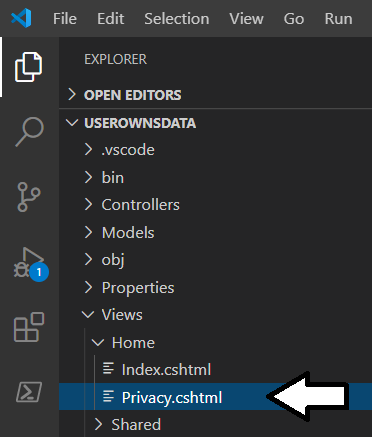
public IActionResult Embed() {

return View();

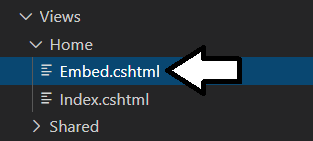
}

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**.

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



* 1. Rename the **Privacy.cshtml** razor file to **Embed.cshtml**..



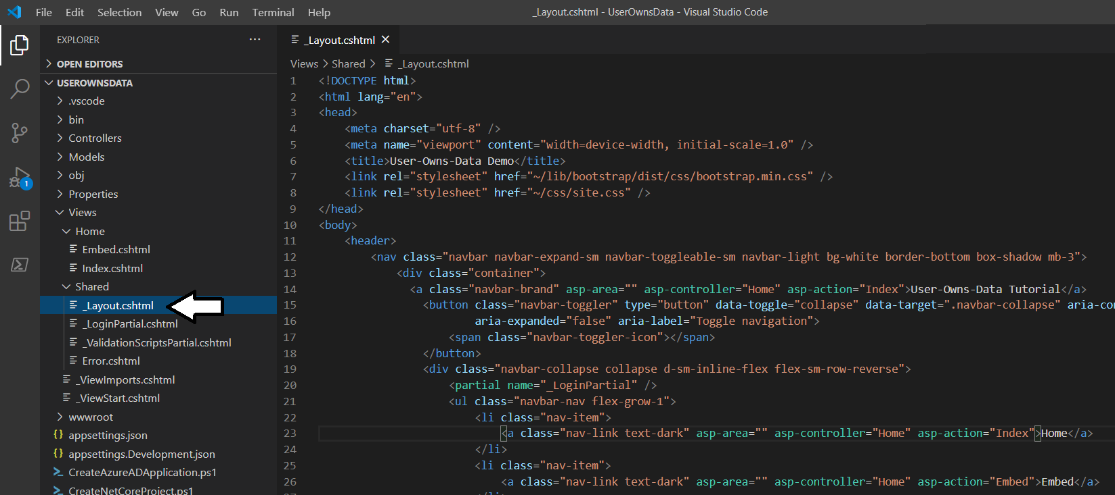
* 1. Open **Embed.cshtml** in a code editor.
  2. Delete the existing contents of **Embed.cshtml** and replace it with the follow line of HTML code.

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

* 1. Save your changes and close **Embed.cshtml**.

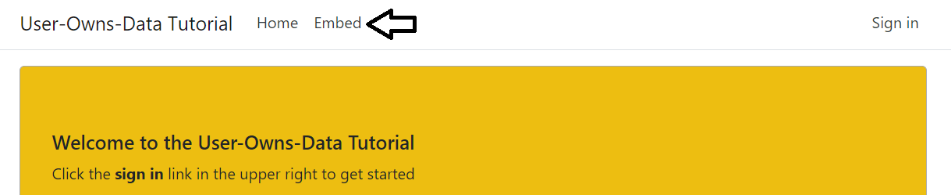
In a standard .NET Core web application that uses MVC, there is a shared page layout defined in a file named **\_Layouts.cshtml** which is located in the **Views > Shared** folder. In the next step you will modify the shared layout in the **\_Layouts.cshtml** file so that you can add the Embed page into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

1. Modify the shared layout in **\_Layout.cshtml** to include a link to the **Embed** page.
   1. Inside the **Views > Shared** folder, locate \_Layouts.cshtml and open this shared view file in an editor window.
   2. Using Windows Explorer, look inside the **StudentLabFiles** folder and locate the file named **\_Layout.cshtml**.
   3. Open the **\_Layout.cshtml** file in the **StudentLabFiles** folder in a text editor and copy its contents to the Windows clipboard.
   4. Return to Visual Studio Code and paste the contents of the Windws clipboard into the **\_Layouts.cshtml** file.

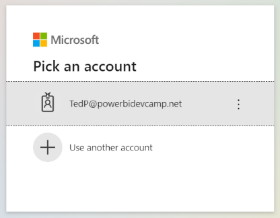


* 1. Save your changes and close **\_Layouts.cshtml**

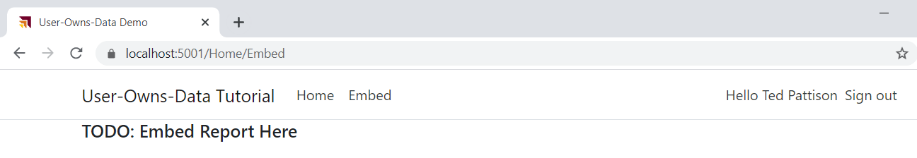
1. Run the web application in the Visual Studio Code debugger to test the new **Embed** page.
   1. Start the Visual Studio Code debugger by selecting **Run > Start Debugging** or by pressing the **{F5}** keyboard shortcut.
   2. The **UserOwnsData** web application should display the home page as shown to an anonymous user.
   3. Click on the **Embed** link in the top nav menu to navigate to the **Embed** page.



* 1. When you attempt to navigate to the Embed page as an anonymous user, you'll be prompted to pick an account and log in.
  2. Log in using your user name and password.



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



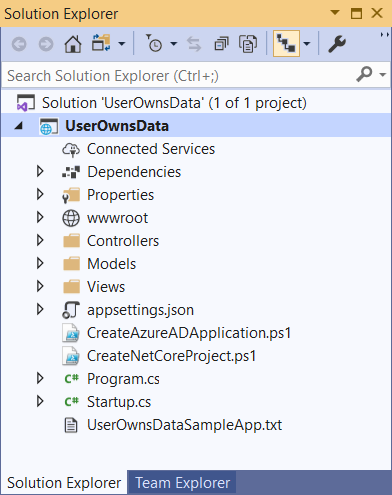
* 1. You're done testing. Close the browser, return to Visual Studio Code and stop the debug session using the debug toolbar.

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*.

1. Open and test the **UserOwnsData** project using Visual Studio 2019.
   1. Launch Visual Studio 2019 – You can use any edition including the Enterprise edition, Pro edition or Community edition.
   2. From the **File** menu, select the **Open > Project/Solution…** command.

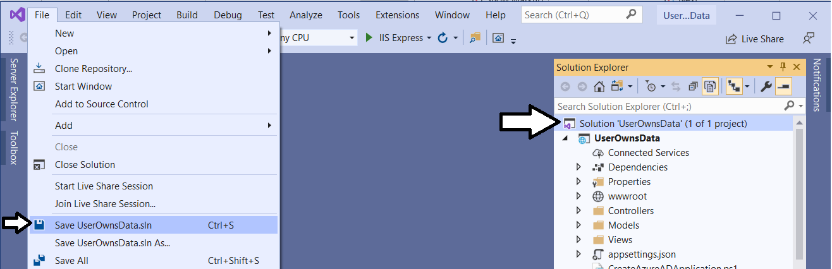


* 1. In the **Open Project/Solution** dialog, select the **UserOwnsData.csproj** file in the **UserOwnsData** folder and click **Open**.
  2. The **UserOwnsData** project should now be open in Visual Studio 2019 as shown in the following screenshot.

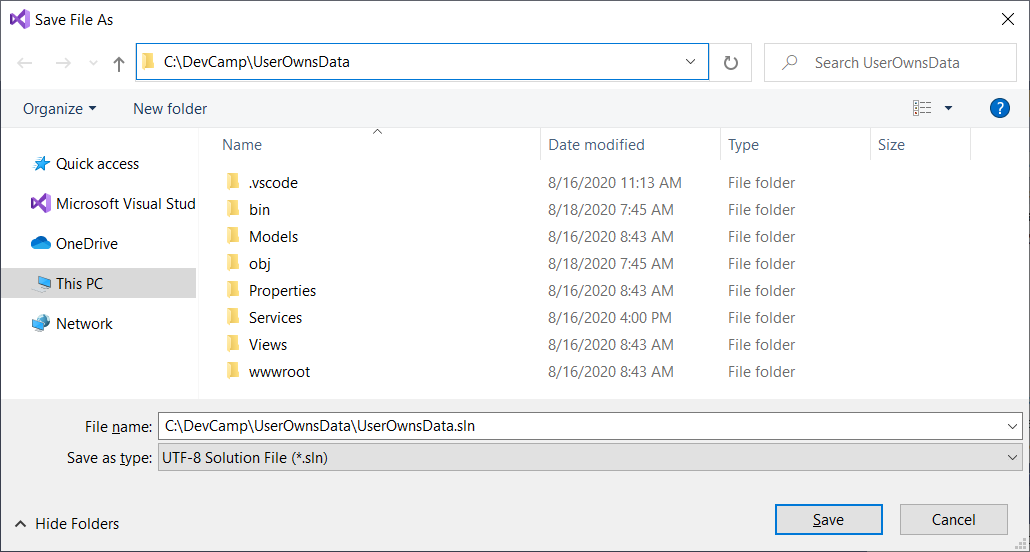


There is one big difference between developing with Visual Studio Code and Visual Studio 2019. Visual Studio Code only requires project files (\*.csproj). However, Visual Studio 2019 requires that you work with both project files and solution files (\*.sln). In the next step you will save a new project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

* 1. In the **Solution Explorer** on the right, select the top node in the tree with the caption **Solution "UserOwnsData"**.
  2. From the **File** menu, select the **Save UserOwnsData.sln** menu command.

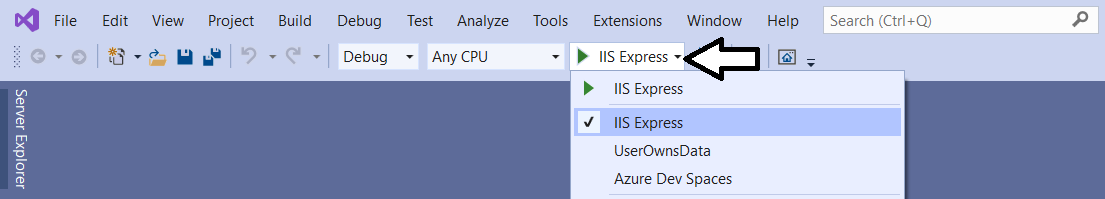


* 1. Save the solution file **UserOwnsData.sln** in the **UserOwnsData** project folder

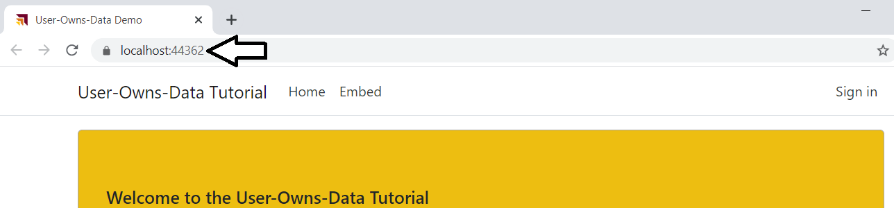


Remember that the **UserOwnsData.sln** file is only used by Visual Studio 2019 and it not used at all in Visual Studio Code.

1. Test the **UserOwnsData** web application in Visual Studio 2019.
   1. On the Visual Studio 2019 toolbar, locate the **Start** button with the green arrow.
   2. If you drop down the **Start** button menu, you should see the project configuration of **IIS Express** and **UserOwnsData**.
   3. Leave the **Start** button menu set for **IIS Express** and click the **Start** button to start run the project in the debugger.

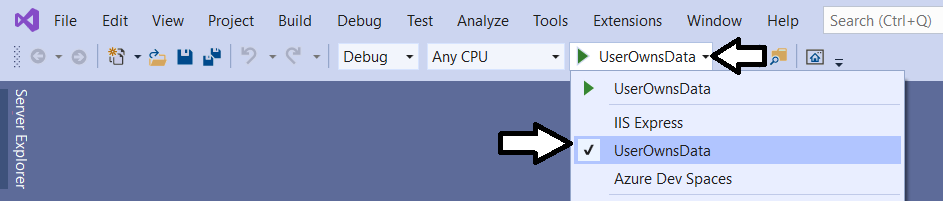


* 1. The **UserOwnsData** application should run and serve up the home page using localhost and a port number around 44300.



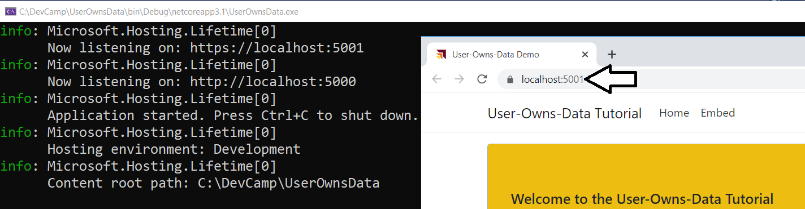
You should be able to test the **UserOwnsData** application by logging in and navigating to the embed page.

* 1. After you are done testing the application, close the browser, return to Visual Studio and stop the current debugging session.
  2. 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/**.

* 1. With the **Start** button menu set for **UserOwnsData,** click the **Start** button to start run the project in the debugger.
  2. Note when you are debugging with **Kestrel**, you will see more activity output in the console window.



* 1. 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.

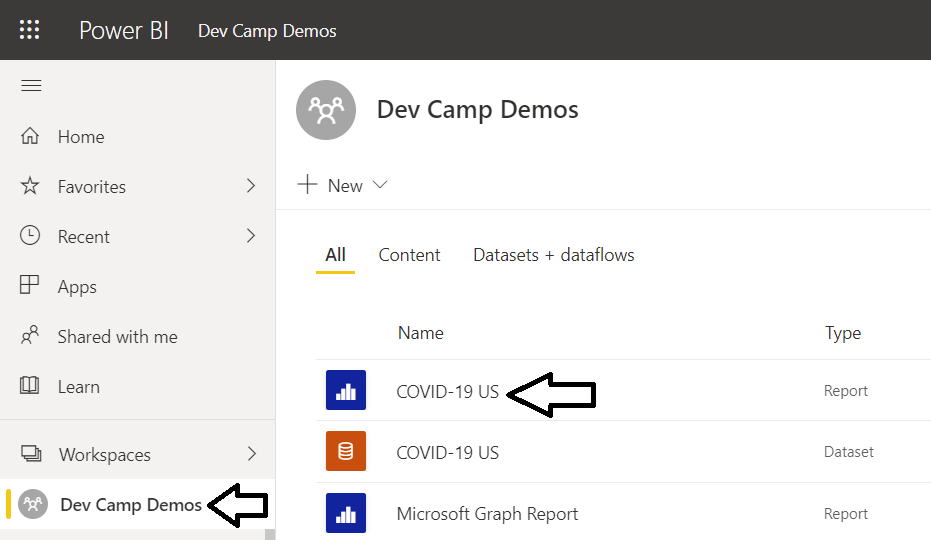
### Exercise 3: Call the Power BI Service API

In this exercise, you will 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, you will be able to call to the Power BI Service API to retrieve data about a report required for embedding. You will begin this exercise by ensuring you have a Power BI app workspace and a report for your testing.

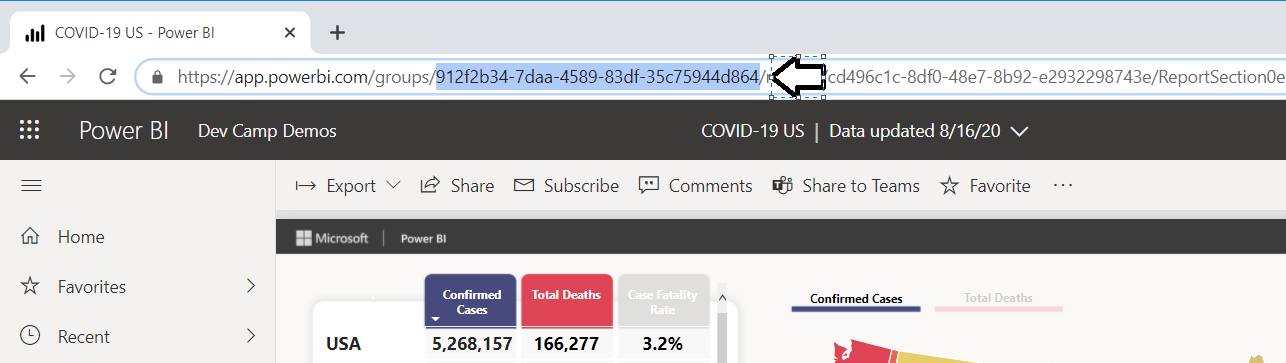
1. Navigate to the Power BI Service.
   1. In the browser, navigate to the Power BI Service at <https://app.powerbi.com>.
   2. Log in with the same account that you used when running the PowerShell script to create the Azure AD application earlier.

Note that the user account you use and the Azure AD application you created earlier must existing within the same Azure AD tenant.

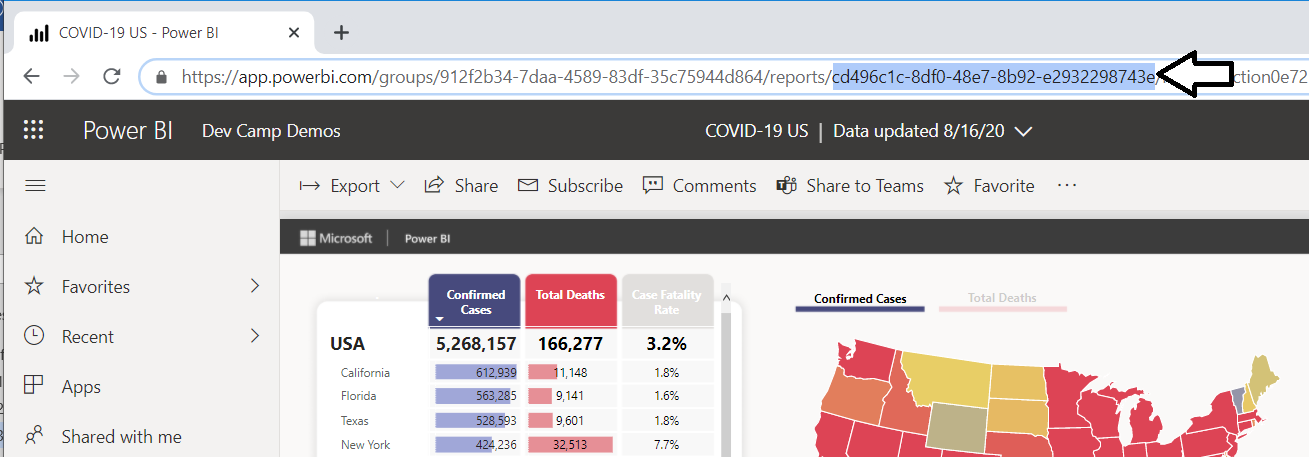
1. In Power BI Service, create a new workspace and add a report.
   1. If you need a report for testing, you can download the report named **COVID-19 US.pbix** from this [download link](https://github.com/TedPattison/NetCore-UserOwnsData-Tutorial/raw/master/PBIX/COVID-19%20US.pbix).
   2. Get GUIDs for these



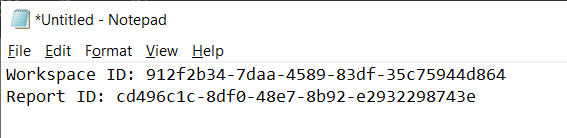
* 1. X



* 1. X

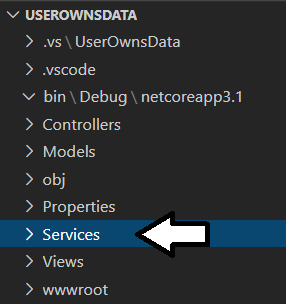


* 1. X

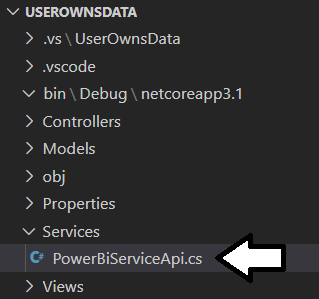


* 1. x

1. Create a new service class named **PowerBiServiceApi** in which you will add the code for calling the Power BI Service API.
   1. Create a new top-level folder in the **UserOwnsData** project named **Services**.



* 1. Inside the **Services** folder, create a new C# source file named **PowerBiServiceApi.cs**.



* 1. Copy and paste the following to provide a starting point for the code in **PowerBiServiceApi.cs**.

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 {

}

public class PowerBiServiceApi {

}

}

* 1. Sss

public class EmbeddedReportViewModel {

public string Id;

public string Name;

public string EmbedUrl;

public string Token;

}

* 1. Sss

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;

}

}

* 1. Xxx

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"

};

* 1. Xxx

public string GetAccessToken() {

return this.tokenAcquisition.GetAccessTokenForUserAsync(RequiredScopes).Result;

}

* 1. ddd

public PowerBIClient GetPowerBiClient() {

var tokenCredentials = new TokenCredentials(GetAccessToken(), "Bearer");

return new PowerBIClient(new Uri(urlPowerBiServiceApiRoot), tokenCredentials);

}

* 1. ssss

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()

};

}

* 1. Save and close **PowerBIServiceApi.cs**.

1. Modify **Startup.cs** to register service
   1. Underneath the existing import statements, add the following import statement;

using UserOwnsData.Services;

* 1. Dddd

public void ConfigureServices(IServiceCollection services) {

services.AddMicrosoftWebAppAuthentication(Configuration);

* 1. Replace with this.

services

.AddMicrosoftWebAppAuthentication(Configuration)

.AddMicrosoftWebAppCallsWebApi(Configuration, PowerBiServiceApi.RequiredScopes)

.AddInMemoryTokenCaches();

* 1. Now this.

services.AddScoped(typeof(PowerBiServiceApi));

* 1. The whole this

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();

}

1. Modify Home controller to inject PowerBiApi.
   1. Open HomeController.cs.
   2. Add import

using UserOwnsData.Services;

* 1. Locate

[Authorize]

public class HomeController : Controller {

private readonly ILogger<HomeController> \_logger;

public HomeController(ILogger<HomeController> logger) {

\_logger = logger;

}

* 1. Replace with this

[Authorize]

public class HomeController : Controller {

private PowerBiServiceApi powerBiServiceApi;

public HomeController(PowerBiServiceApi powerBiServiceApi) {

this.powerBiServiceApi = powerBiServiceApi;

}

* 1. Replace

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);

}

1. Add HTML to Embed.cshtml to create a table.
   1. Sss

@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>

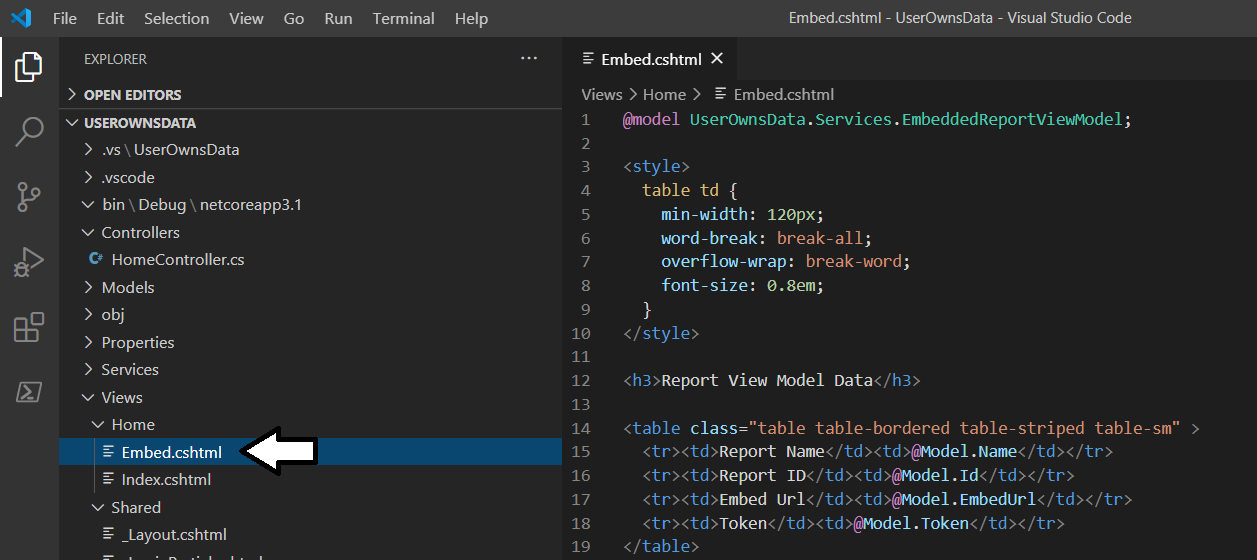
<tr><td>Report ID</td><td>@Model.Id</td></tr>

<tr><td>Embed Url</td><td>@Model.EmbedUrl</td></tr>

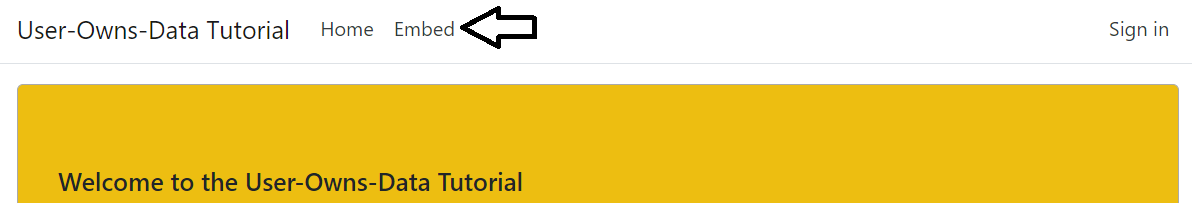
<tr><td>Token</td><td>@Model.Token</td></tr>

</table>

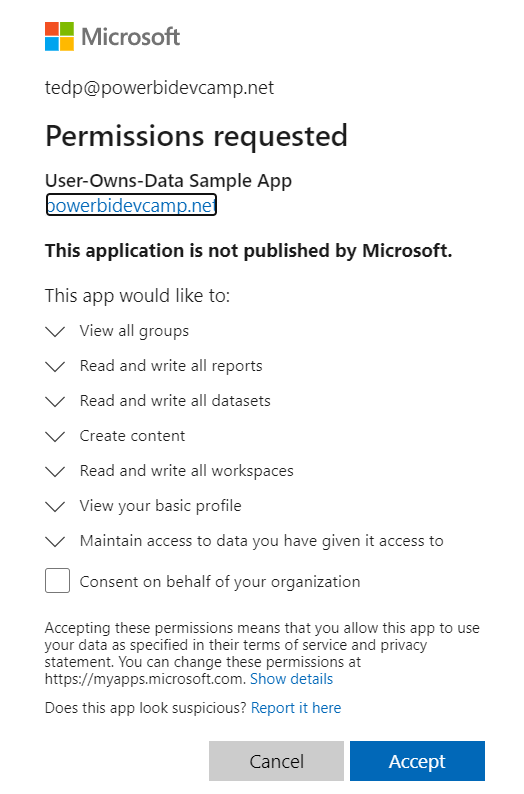
* 1. Xx



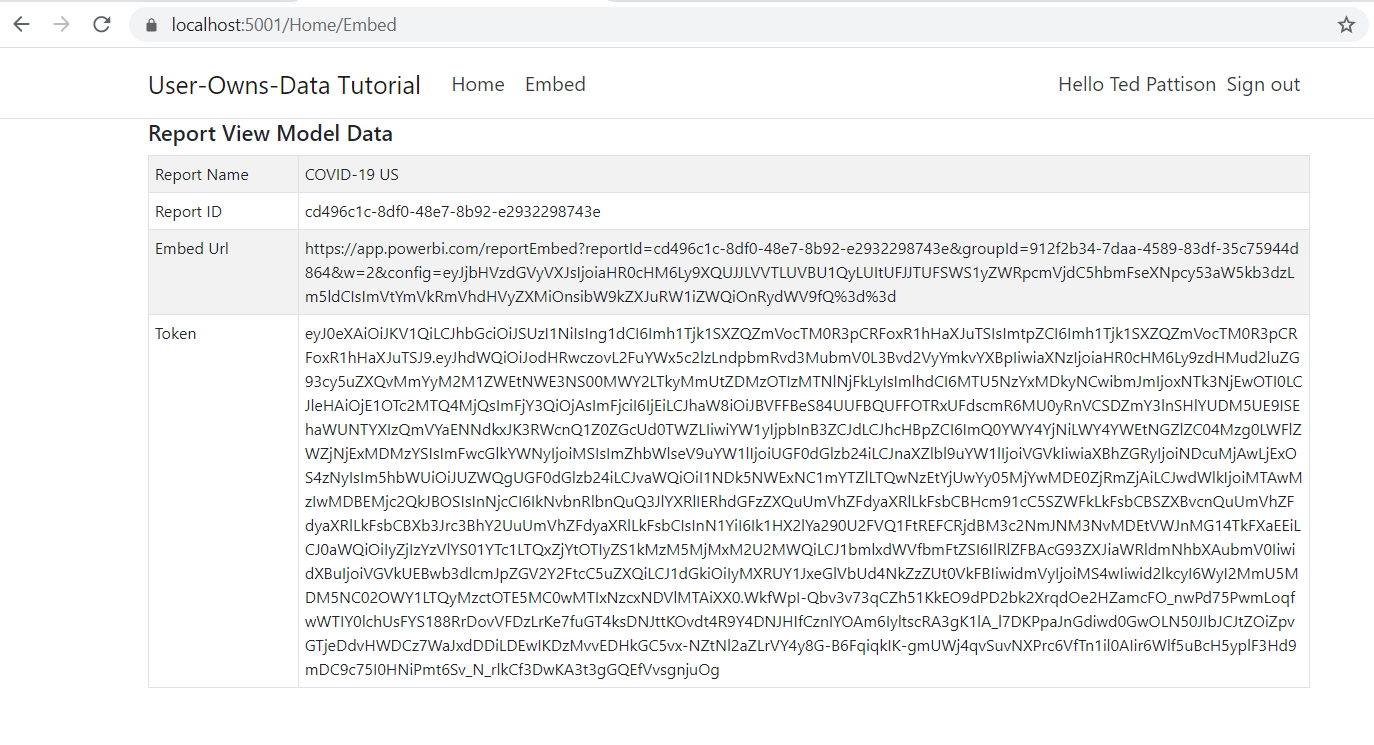
1. Test
   1. Ssss



* 1. S



* 1. X

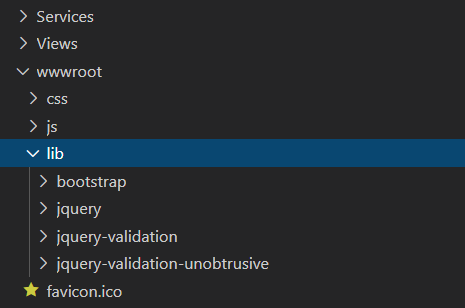


* 1. x

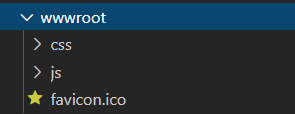
### Exercise 4: Embedding a Report using powerbi.js

In this exercise, you will create a new confidential client application in the Azure portal and you will configure the application’s required permissions to provide the access you need to call into the Power BI Service API.

1. Remove all files from inside the wwwroot lib
   1. Sss



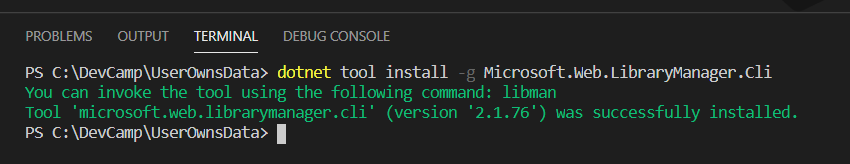
* 1. Ddd



1. Add client-side library support with Libman,json
   1. Install tool

dotnet tool install -g Microsoft.Web.LibraryManager.Cli

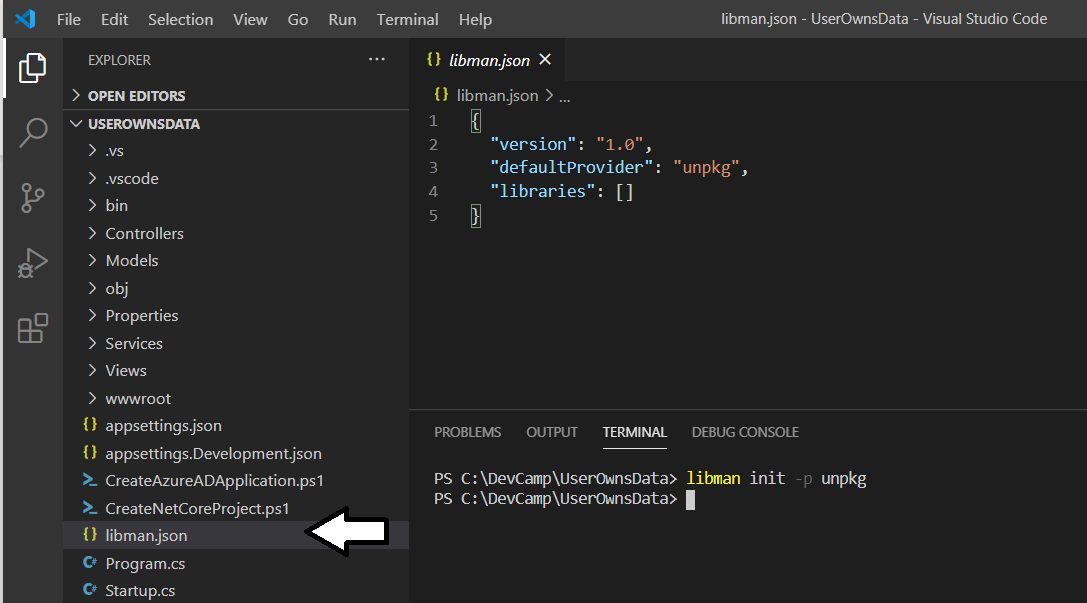
* 1. x



* 1. X

libman init -p unpkg

* 1. Sss



* 1. Sss

{

"version": "1.0",

"defaultProvider": "unpkg",

"libraries": [

{

"library": "jquery@3.5.1",

"destination": "wwwroot/lib/jquery/",

"files": [ "dist/jquery.min.js" ]

},

{

"library": "bootstrap@4.5.2",

"destination": "wwwroot/lib/bootstrap/",

"files": [ "dist/css/bootstrap.min.css", "dist/js/bootstrap.bundle.min.js" ]

},

{

"provider": "unpkg",

"library": "powerbi-client@2.13.3",

"destination": "wwwroot/lib/powerbi-client/",

"files": [ "dist/powerbi.min.js", "dist/powerbi.js" ]

}

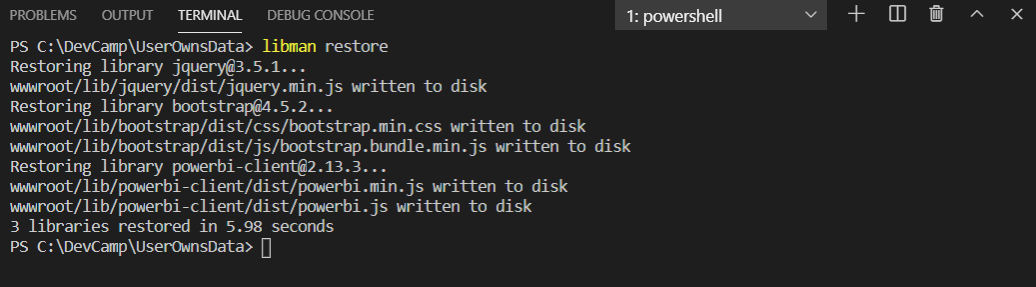
]

}

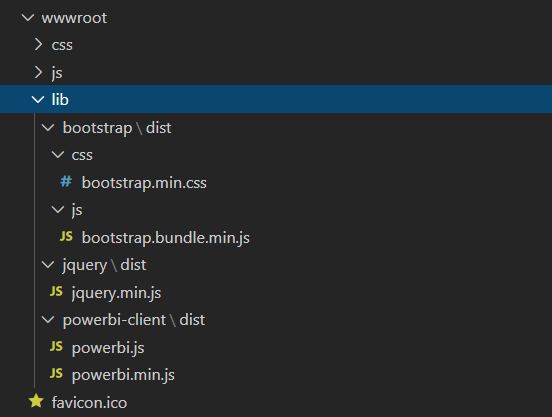
* 1. You should see this.

libman restore

* 1. X



* 1. X



1. Add view model to Embed.cshtml
   1. Open Embed.cshtml in an editor window.
   2. Replace the contents of Embed.cshtml with the following code.

@model UserOwnsData.Services.EmbeddedReportViewModel;

<style>

.mb-3, .my-3 { margin-bottom: 0 !important; }

</style>

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

@section Scripts {

<script src="~/lib/powerbi-client/dist/powerbi.js"></script>

<script>

var viewModel = {

reportId: "@Model.Id",

embedUrl: "@Model.EmbedUrl",

token: "@Model.Token"

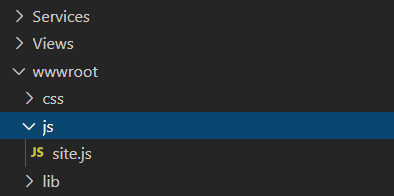
};

</script>

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

}

1. Add a new JavaScript file named embed.js.
   1. Sss



* 1. Sss

$(function(){

// get DOM object div for report container

var reportContainer = document.getElementById("embed-container");

var reportId = window.viewModel.reportId;

var embedUrl = window.viewModel.embedUrl;

var token = window.viewModel.token

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);

// add logic to resize embed container element on window rersize event

var heightBuffer = 12;

var newHeight = $(window).height() - ($("header").height() + heightBuffer);

$("#embed-container").height(newHeight);

$(window).resize(function () {

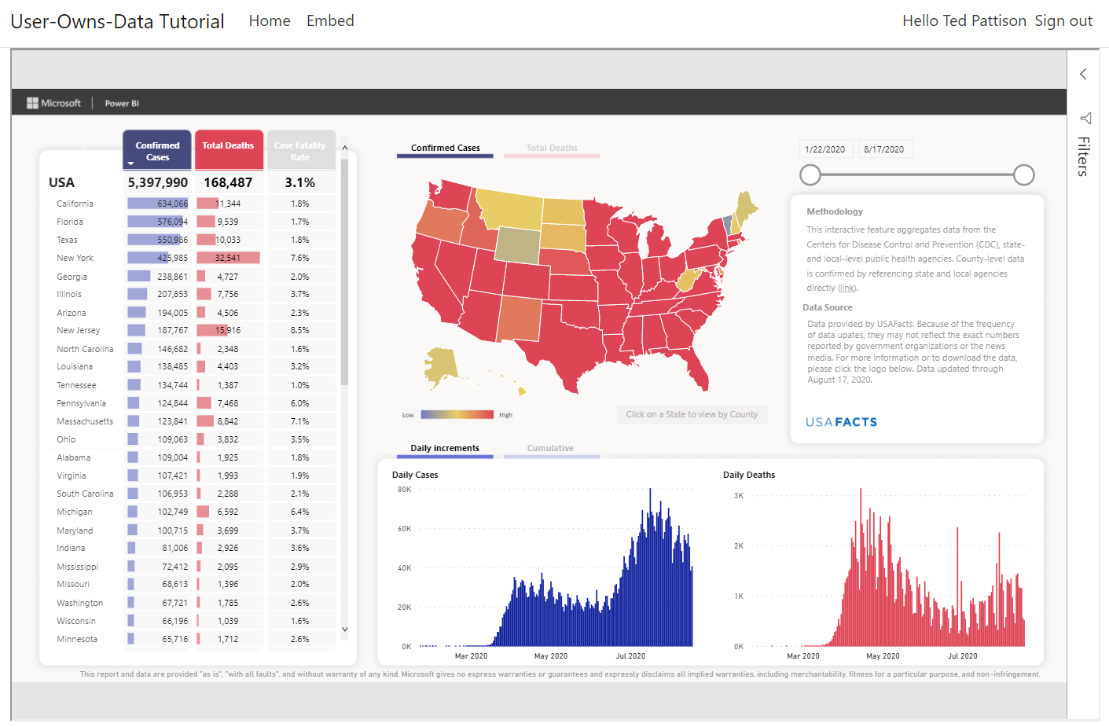
var newHeight = $(window).height() - ($("header").height() + heightBuffer);

$("#embed-container").height(newHeight);

});

});

1. Add Embed,js
2. Test



### Exercise 5: Adding TypeScript Support to a .NET Core Project

In this exercise, you will create a new confidential client application in the Azure portal and you will configure the application’s required permissions to provide the access you need to call into the Power BI Service API.

1. Add package.json – npm init
   1. sss
2. Add tsconfig.json
   1. sss
3. Add webpack.config.js
   1. Ssss
4. Add embed.js
   1. Create Script folder
   2. Add embed.js
   3. Compile from command line
5. Update UserOwnsData.csproj
   1. sss

### Exercise 6: Creating a View Model for App Workspaces

In this exercise, you will create a new confidential client application in the Azure portal and you will configure the application’s required permissions to provide the access you need to call into the Power BI Service API.

1. Add new method to PowerBiApi.cs
   1. Sss
2. Modify Embed method in HomeController to call new method.
   1. Pass view model as object not as string
3. Rewrite Embed.cshtml
4. Rewrite Embed.ts.