Developing a Custom Web Service using Web API

Lab Time: 60 minutes

Lab Folder: C:\Student\Modules\06_WebAPI\Lab

Lab Overview: In this module, you will execute a PowerShell script to create an Azure SQL database named **ProductsDB**. After that, you will extend an ASP.NET MVC application named **ProductManagerSQL** with an Entity Framework model and a strongly-typed controller class to read and write product data in the **ProductsDB** database.

Exercise 1: Developing a Custom Web Service using Web API

In this exercise, you will begin with a pre-provided Visual Studio project named **ProductManagerWebAPI** which is similar to the project you worked on in the previous lab named **ProductManagerAngular1**. You will extend the **ProductManagerWebAPI** project by adding an Entity Framework model and a strongly-typed ODATA controller class to provide read/write access to the **ProductsDB** database through a custom Web service. After that, you will write client-side TypeScript code to access your new Web service.

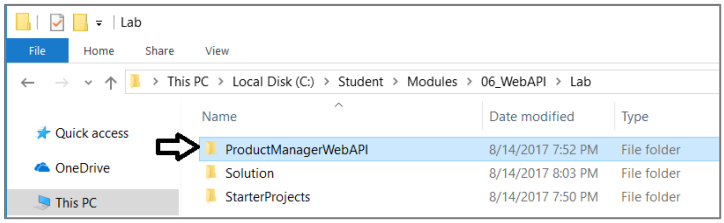
1. Copy the starter project named **ProductManagerWebAPI**.
   a) In Windows Explorer, navigate to the following folder.

   ```
   C:\Student\Modules\06_WebAPI\Lab\StarterProjects
   ```

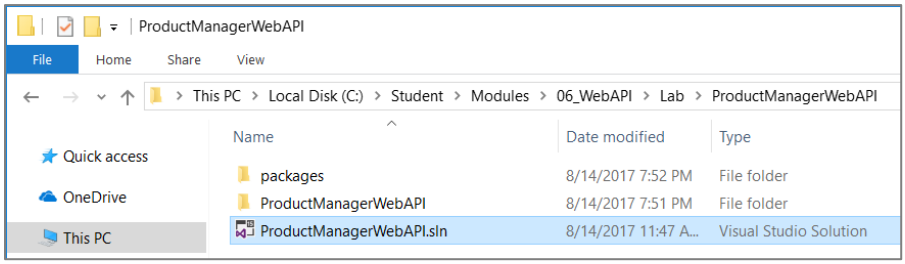
   b) Select the folder named **ProductManagerWebAPI** and copy it to the windows clipboard.
   c) In Windows Explorer, move out one level to the **Lab** folder at the following location.

   ```
   C:\Student\Modules\06_WebAPI\Lab
   ```

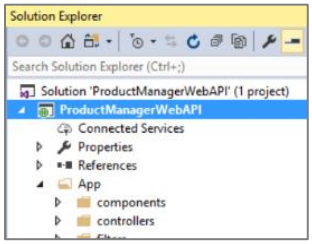
   d) Paste the **ProductManagerWebAPI** folder to make a copy of it inside the **Lab** folder.



   e) Navigate inside the **ProductMangerWebAPI** folder and double-click on the solution file named **ProductManagerWebAPI.sln** to open the **ProductManagerWebAPI** project in Visual Studio.

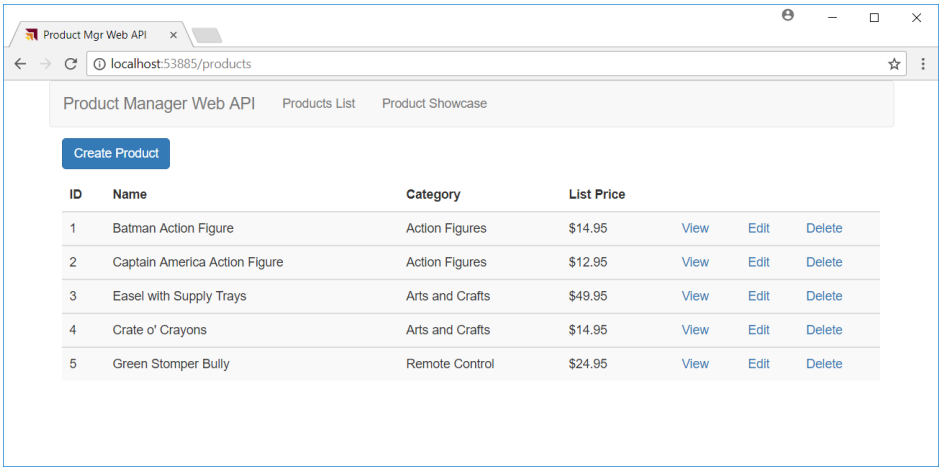


   f) Take a moment to review the project structure of the **ProductManagerWebAPI** project.



g) Press the **{F5}** key to start a debugging session in Visual Studio.

h) The project should start and display its home page in the browser as shown in the following screenshot.

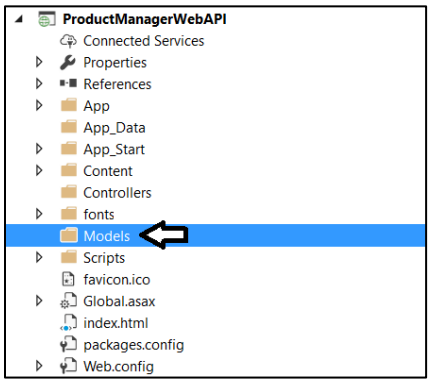


Currently the project is using product data that is hard-coded into a TypeScript class named **AsyncInMemoryProductDataService** which is include inside a source file named **services.ts** which is located inside the **App/services** folder. By the end of this lab, your application will be retrieving and updating product data in the Azure SQL database you created in Lab 4.

i) Close the browser and return to Visual Studio and terminate the debugging session.

2. Add a new Entity Framework model to provide access to the **ProductsDB** database.

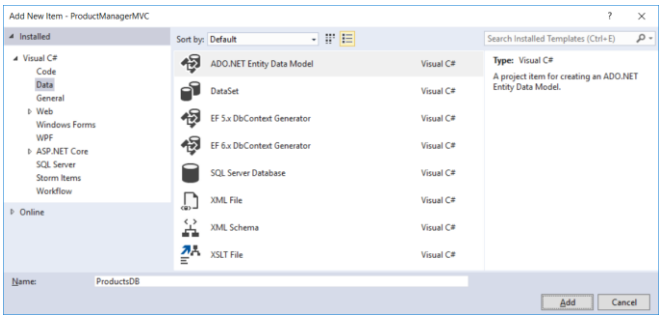
a) In Solution Explorer, locate the top-level folder named **Models**.



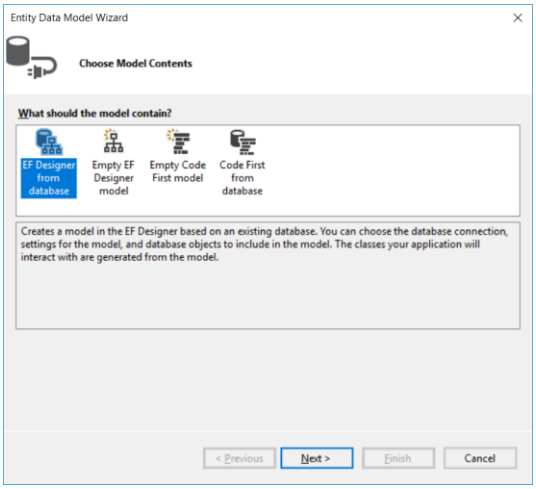
b) Right-click the **Models** folder and select the **Add > New Item…** command.

c) In the **Add New Item** dialog, select Visual C# > Data on the left and then select **ADO.NET Entity Data Model**.

d) In the **Name** textbox, enter a name of **ProductsDB** and then click the **Add** button.

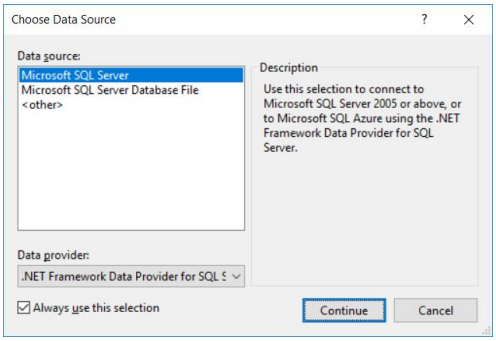


e) On the first page of the **Entity Data Model Wizard**, select **EF Designer from database** and then click **Next**.



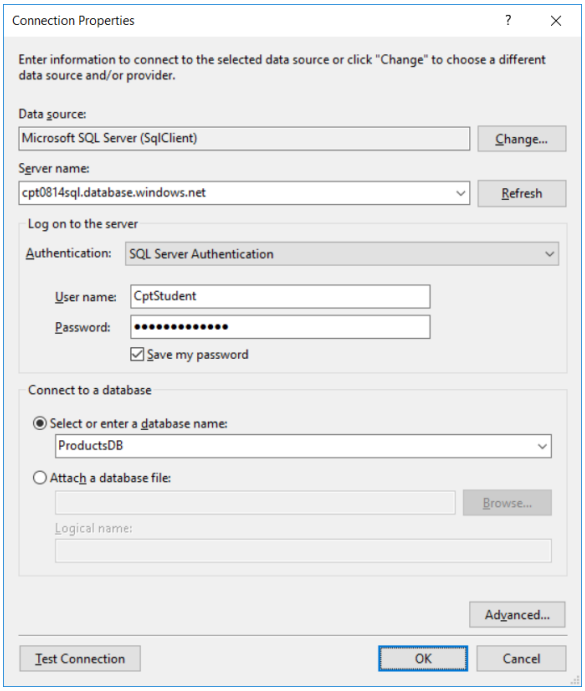
f) On the **Choose Your Data Connection** page of the **Entity Data Model Wizard**, click **New Connection…**.

g) In the **Choose Data Source** dialog, select **Microsoft SQL Server** and click **Continue**.

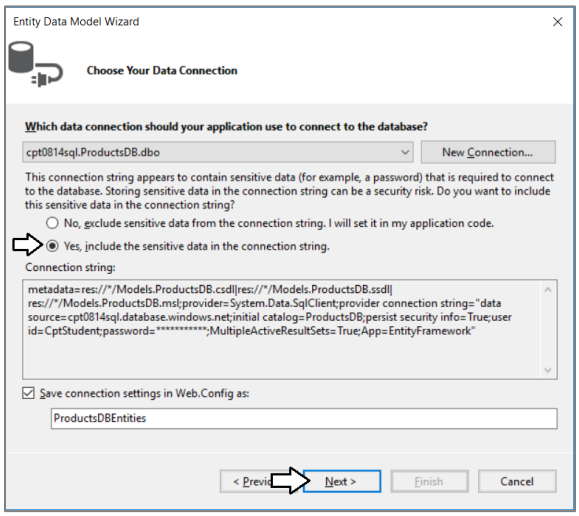


h) In the **Connection Properties** dialog, enter the following information.
   i) For **Server name**, add the name of your SQL Server instance including the suffix of **database.windows.net**.
   ii) For **Authentication**, set the value to **SQL Server Authentication**.
   iii) Enter a **User name** of **CptStudent**.
   iv) Enter a **Password** of **pass@word1234**.
   v) Check to **Save my password** checkbox.
   vi) In the **Select or enter a database name** dropdown list, select **ProductsDB**.

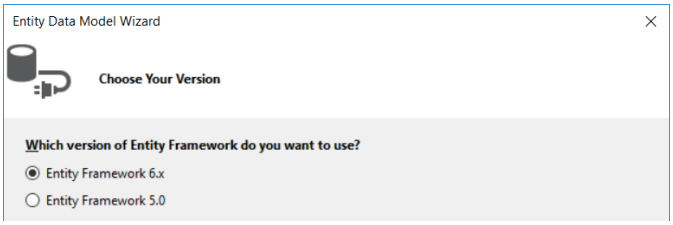
i) When the **Connection Properties** dialog matches the following screenshot (except for Server name), click **OK** to continue.



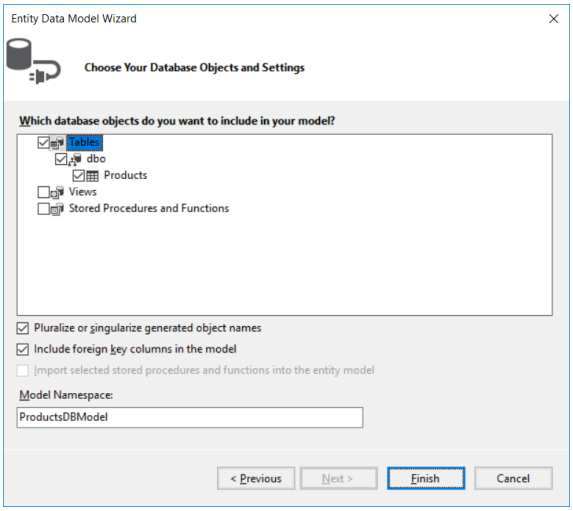
j) On the **Choose Your Data Connection** page, select the option **Yes, include the sensitive data in the connection string**. and then click **Next** button to continue.



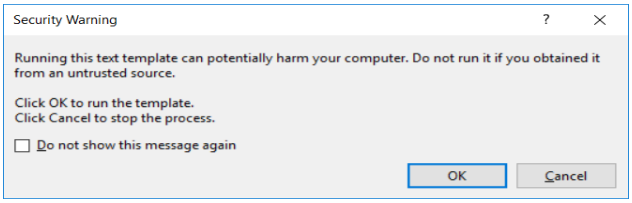
k) On the **Choose Your Version** dialog, select **Entity Framework 6.x** and click **Next** to continue.



l) In the **Choose Your Database Objects and Setting** page, select the **Products** table and click **Next**.



m) If you are prompted with a **Security Warning** dialog shown in the following screenshot, click **OK** to continue.

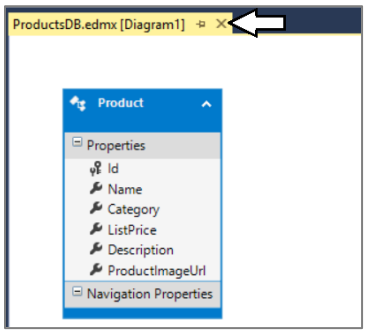


It will usually take Visual Studio 30-60 seconds to complete its work building a small Entity Framework model.

n) Wait until Visual Studio completes its work creating the files for the new Entity Framework model.

o) When Visual Studio finishes, it will display a visual designer of the model in a file named **ProductsDB.edmx**.

p) Click the **x** button to close the viewer for **ProductsDB.edmx**.



When you create an Entity Framework data model like this, the full set of generated code is not built out until you run the project for the first time. In the next step you will run and build the project one time to ensure all the Entity Framework code is fully generated.

3. Run the project one time to fully build out the Entity Framework model code.

a) Press the **{F5}** key to start a debugging session.

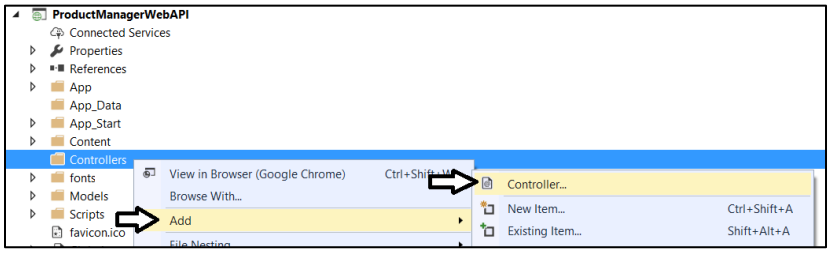
b) Once the browser starts and display the home page, close the browser.

c) Return to Visual Studio and terminate the debugging session.

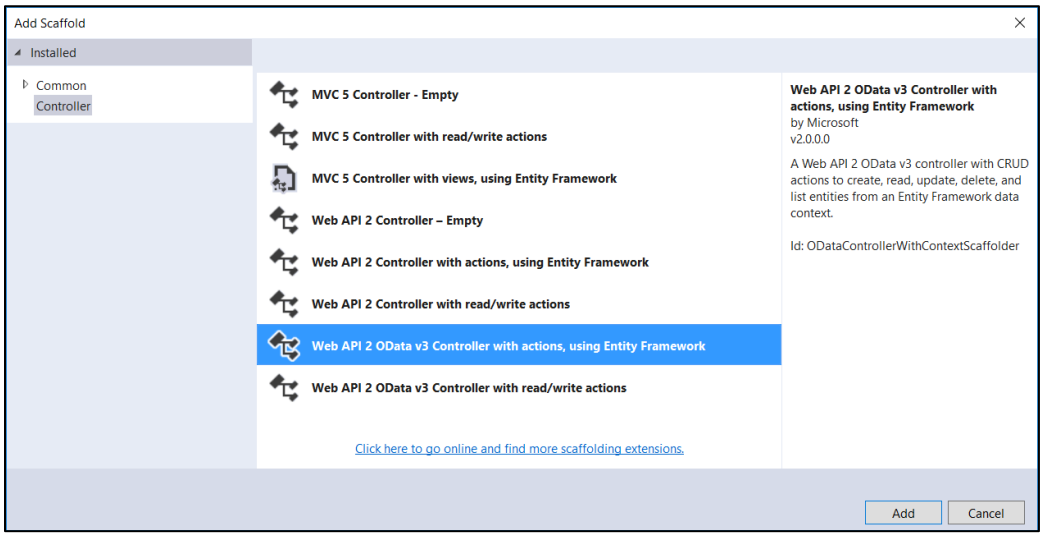
Exercise 3: Extend an MVC Web Application with CRUD Behavior

In this exercise, you will create a strongly-typed controller class using the Entity Framework model you created in an earlier exercise.

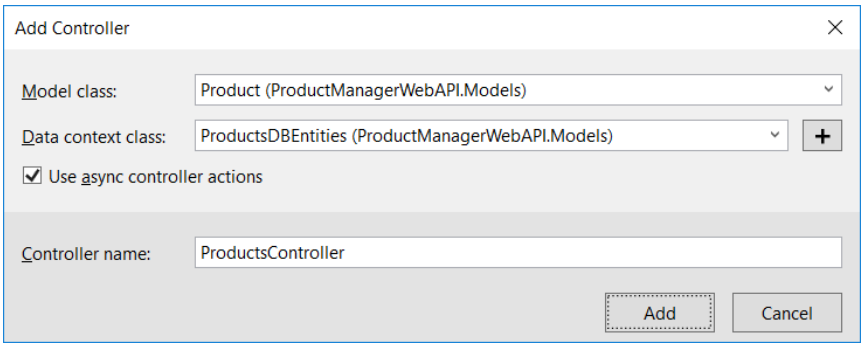
1. Add a new strongly-typed OData controller class named **Products**.
   a) Inside **Solution Explorer**, right-click the **Controllers** folder and then select the **Add > Controller…** menu command.



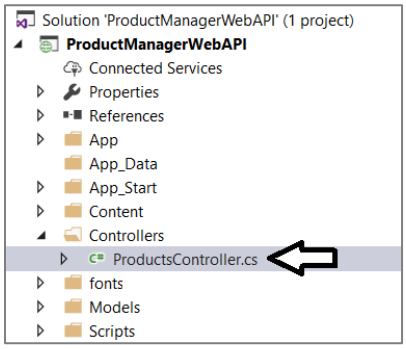
   b) In the **Add Scaffold** dialog, select **Web API 2 OData v3 Controller with actions, using Entity Framework** and click **Add**.



   c) In the **Add Controller** dialog, enter the following entries.
      i) Set **Model class** to **Product**.
      ii) Set **Data context class** to **ProductsDBEntities**.
      iii) Check the **Use async controller actions** checkbox.
      iv) Leave the default value of the **Controller name** which is **ProductsController**.
   d) When the **Add Controller** dialog matches the following screenshot, click **Add** to create the new controller class.



e) Visual Studio will create add a new source file named **ProductsController.cs**.

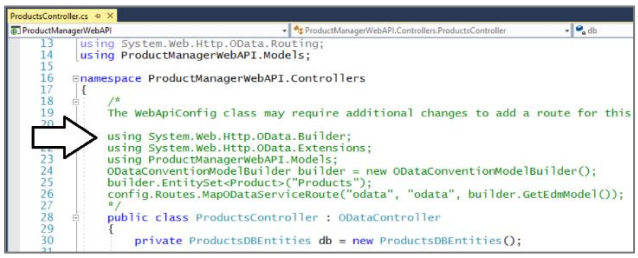


f) Take a moment and inspect the method that are defined inside the **ProductsController** class including **GetProducts**, **GetProduct**, **Put**, **Post**, **Patch** and **Delete**.

There is a big difference between generating an MVC controller class like you did in an earlier lab and generating a Web API controller class like you are doing in this lab. MVC controller classes are generated along with MVC view because they must generated HTML to return to the browser. However, Web API controller classes do not generated HTML and, therefore, are not created with associated MVC view files. The only thing that is generated is the classes named ProductsController.

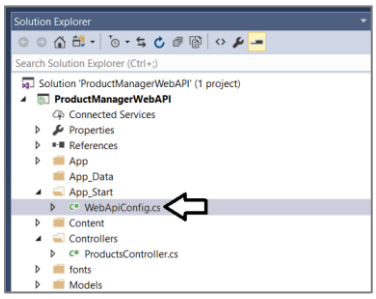
2. Update **WebApiConfig.cs** to add the new OData controller into the application's route map.

a) Look at the top of the source file named **ProductsController.cs** just above the **ProductsController** class and locate the three **using** statements that have been added inside a large multiline comment.



b) Copy the three **using** statements into the Windows clipboard.

c) Inside the **App_Start** folder, locate and open the source file named **WebApiConfig.cs** in an editor window.



d) Paste the **using** statements from the Windows clipboard into **WebApiConfig.cs** just under the other **using** statements.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

using System.Web.Http.OData.Builder;
using System.Web.Http.OData.Extensions;
using ProductManagerWebAPI.Models;
```

e) Return to the code window for **ProductsController.ts** and locate the following three lines in the comment section.

```
ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");
config.Routes.MapODataServiceRoute("odata", "odata", builder.GetEdmModel());
```

f) Copy those three lines from **ProductsController.ts** into the Windows clipboard.

g) Return to the code editor Window for **WebApiConfig.ts**.

h) Inside the **Register** method in **WebApiConfig.ts**, locate the comment which reads **// Web API configuration and services**.

```
// Web API configuration and services
```

i) Just under this comment, paste the three lines you copied into the Windows clipboard

```
// Web API configuration and services
ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Product>("Products");
config.Routes.MapODataServiceRoute("odata", "odata", builder.GetEdmModel());
```

j) At this point, the code you have added to in **WebApiConfig.ts** should match the following screenshot.

k) Save your changes and close **WebApiConfig.ts**

Now that you have created the OData controller and added to the application's route map, you can test it out o make sure it works correctly before you begin to write client-side code in TypeScript to access the new web service programmatically.

3. Run the application and test out the **Products** controller.

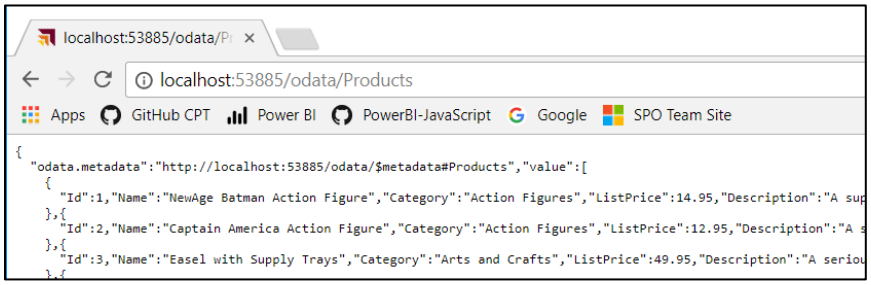
a) Press the **{F5}** key to begin a debugging sessions.

b) When the application starts, you will see the home page as you did before.

c) Place your cursor in the address bar and edit he URL to the base URL for the site and then **/odata/Products**.so that your URL looks like the following URL.

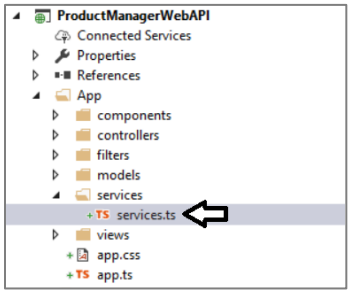
```
http://localhost:53885/odata/Products
```

d) You should see a JSON-formatted response with data from the **ProductsDB** database as shown in the following screenshot.

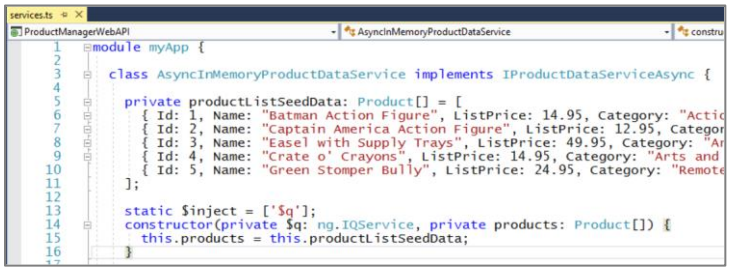


Now that you have created a functional OData web service which reads and writes data to the **ProductsDB** database, the next step is to add the required client-side TypeScript code into the Angular application to integrate the web service.

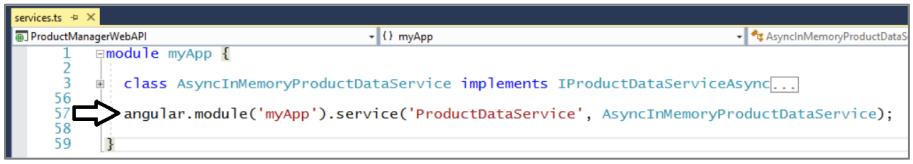
4. Add the TypeScript code to **services.ts** to integrate the new OData web service into the Angular application.
   a) Inside the **App/services** folder, locate the source file named **service.ts** and open it in an editor window.



   b) At the top of **services.ts**, there is a class definition named **AsyncInMemoryProductDataService**.



   c) Underneath the definition for the **AsyncInMemoryProductDataService** class, there is a line of code that calls the **service** method to register the **AsyncInMemoryProductDataService** class as a named service named **ProductDataService**.

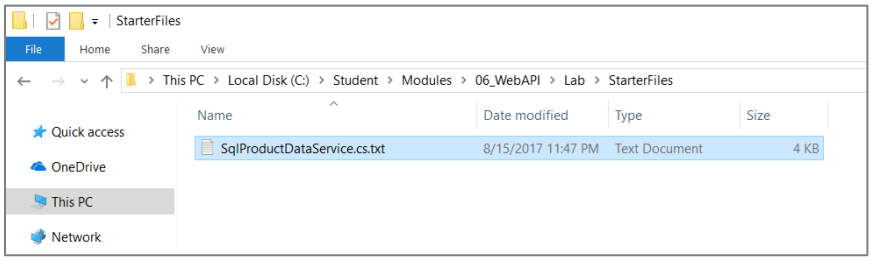


Over the next few steps you will add a new class named **SqlProductDataService** that implements the **IProductDataServiceAsync** interface just like the **AsyncInMemoryProductDataService** class. That will make it very easy to swap out the the **AsyncInMemoryProductDataService** class with the **SqlProductDataService** class.

   d) Using Windows Explorer, navigate to the following path.

```
C:\Student\Modules\06_WebAPI\Lab\StarterFiles
```

   e) Double-click on the file named **SqlProductDataService.cs.txt** to open the file in **NOTEPAD.EXE**.

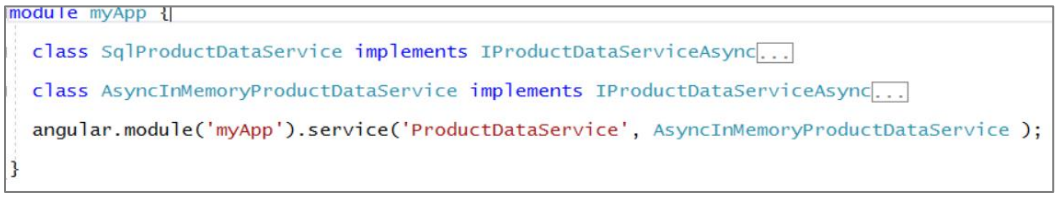


f) Select all the content inside **SqlProductDataService.cs.txt** and copy it to the Windows clipboard.

g) Return to Visual Studio and the code editor window for **service.ts**.

h) Place the cursor at the top of the **myApp** module definition before the **AsyncInMemoryProductDataService** class.

i) Paste in the content of the Windows clipboard to add the **SqlProductDataService** class to services.tx.



j) Go to the bottom of **services.ts** and find the line of code the registers the serviced named **ProductDataService**.

k) At this point it creates an instance of **AsyncInMemoryProductDataService** to when instantiating the servie.

```
angular.module('myApp').service('ProductDataService', AsyncInMemoryProductDataService );
```

l) Modify the code to use the **SqlProductDataService** class instead of the **AsyncInMemoryProductDataService** class.

```
angular.module('myApp').service('ProductDataService', SqlProductDataService);
```

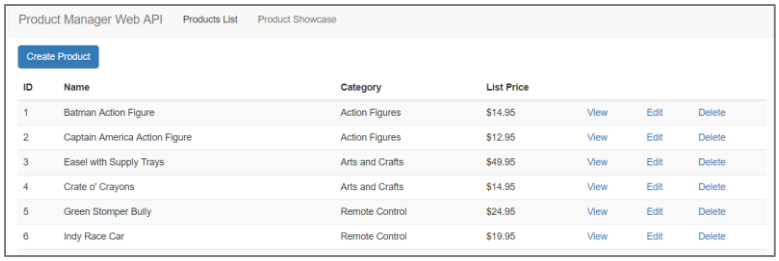
m) The code you have modified in **services.ts** should match the following screenshot.

n) Save your changes to **services.ts**.

5. Test the application.

a) Press the **{F5}** key to start a debugging session in Visual Studio.

b) The project should start and display its home page in the browser.



Congratulations. You have now completed this lab and the application you have created is now using a custom Web service to access product data in an Azure SQL database.