The Generated Code for Database Tables

The Generated Code for Database Tables

1 INTRODUCTION

This topic will walk you through AspCoreGen 6.0 MVC's generated code.

1.1 READ THESE TUTORIALS IN ORDER

1. Database Settings Tab
2. Code Settings Tab
3. UI Settings Tab
4. App Settings Tab
5. Selected Tables Tab
6. Selected Views Tab
7. Generating Code

Then follow these step-by-step instructions.

1.2 GENERATED CODE FOR DATABASE TABLES

In the *Generating Code Tutorial* under the *Database Objects to Generate From*, there are four (4) database objects where we can generate code from. This tutorial will discuss the generated code for database tables only:

1. All Tables
2. Selected Tables Only

1.3 GENERATED PROJECTS

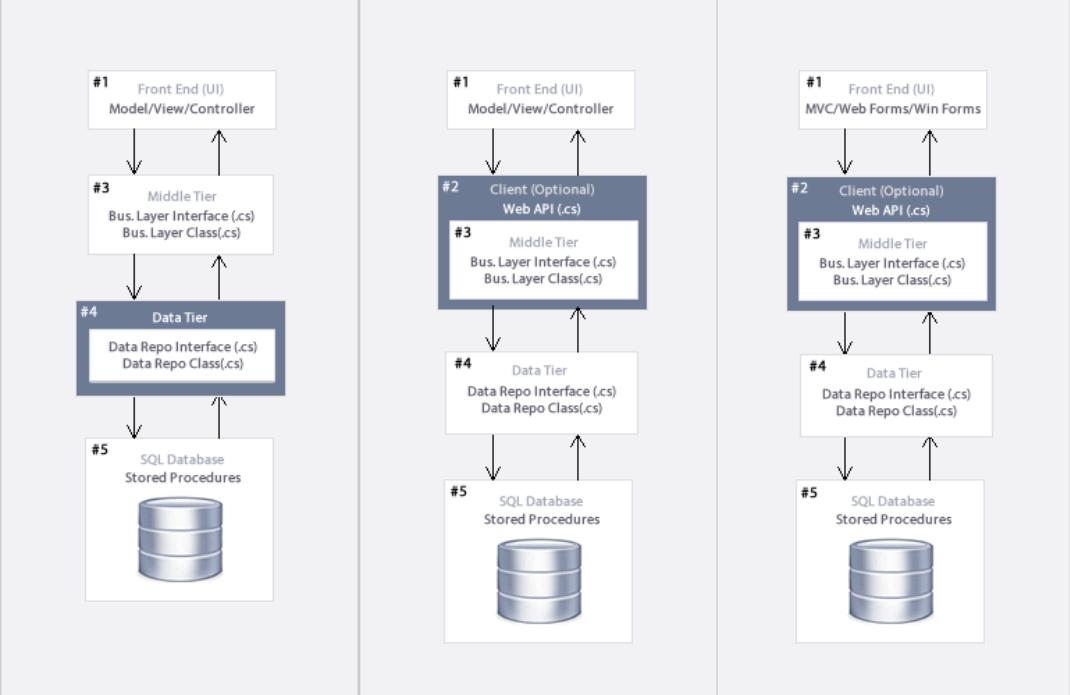
In the *App Settings Tutorial* there are 3 projects that can be generated in a solution:

- Web Application Project (User Interface)
- Middle Layer Project (Class Library Project – Business Layer, Data Repository, and Shared Libraries)
- Web API Project (Web Services - Optional)

We will be discussing these generated projects including the *Web API Project*.

2 N-TIER LAYERING

AspCoreGen 6.0 MVC generates code in an *n-tier* architecture. A presentation tier (the client), middle tier (business layer), data tier (data repository), and the database scripts such as stored procedures. Code are separated in different layers.



2.1 FRONT END

User Interface or Presentation Layer. Views, Controllers, JavaScript, CSS, JQuery, and more.

2.2 MIDDLE-TIER/ MIDDLE LAYER

1. Business Layer Interface and Class, Models, Views, View Models, etc. Or,
2. Web API (Optional). Optionally encapsulate calls to the Business Layer when generating Web API code.

2.3 DATA-TIER/ DATA LAYER

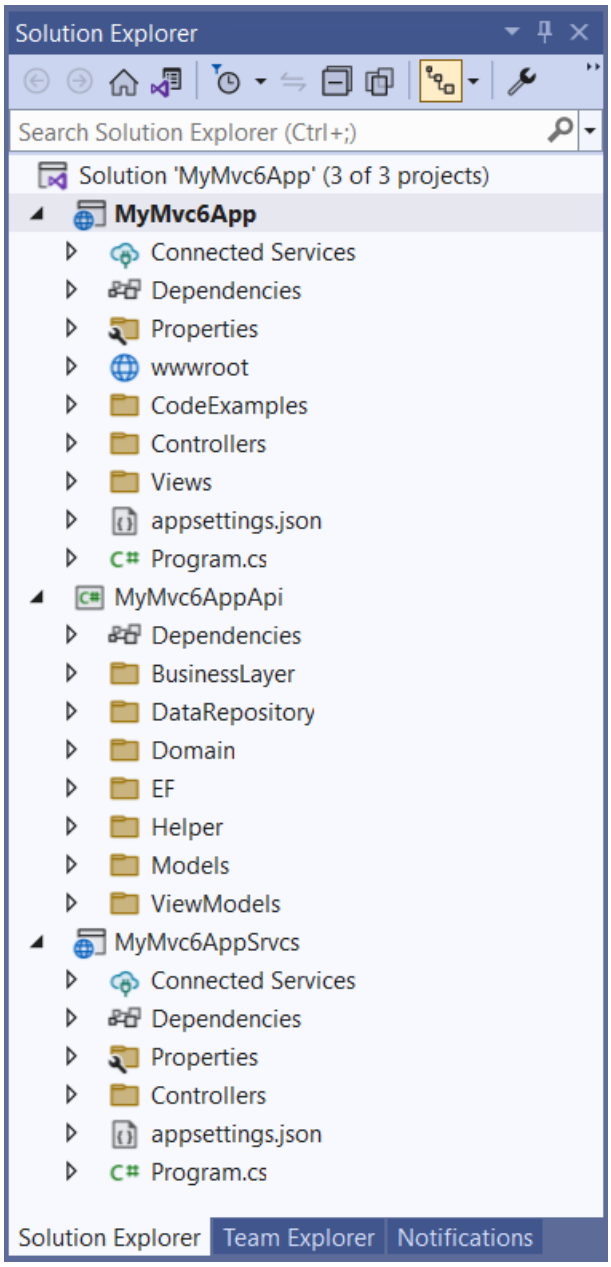
Data Repository Interface and Class, Class Files using Linq-to-Entities - Entity Framework Core or Ad-Hoc SQL.

2.4 SQL SCRIPTS

Stored Procedures.

3 GENERATED PROJECTS

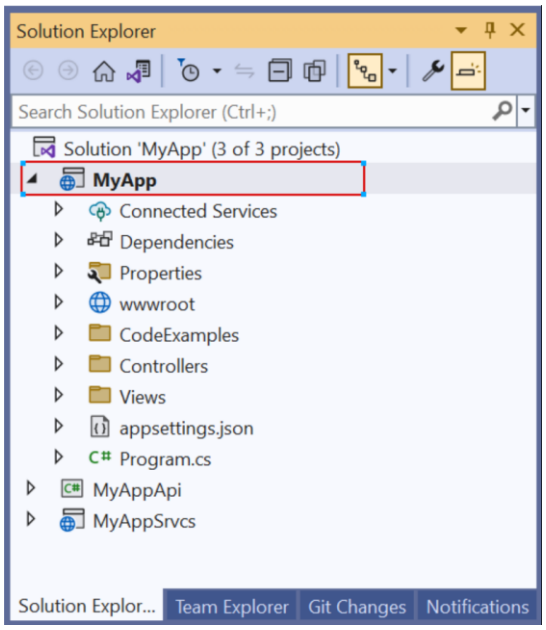
There are 3 projects that can be generated by AspCoreGen 6.0 MVC Professional Plus including the optional *Web API* project. When you chose *Stored Procedures* under the *Generated SQL Script* in the *Database Settings Tab*, these SQL scripts will be generated straight in your MS SQL Server Database's *Stored Procedures* folder.



Generated Projects in Visual Studio

3.1 WEB APPLICATION PROJECT

The generated *Web Application Project* is the *User Interface*, *Front End,* or *Presentation Layer* part of the N-tier layer generated code. This is an ASP.NET MVC core project. The application's main purpose is to serve as a client's user interface. The *Presentation Layer* is what the users see, use, and interact with.

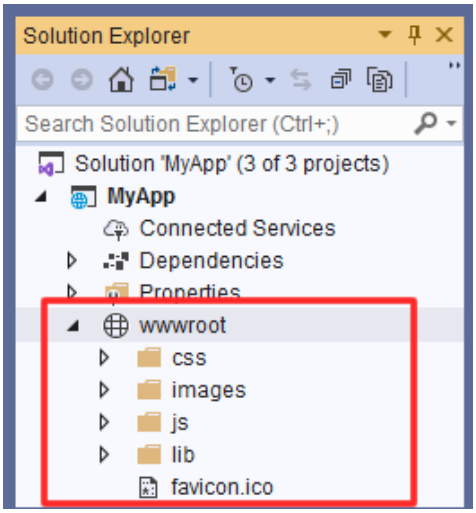


In this example, the *MyApp* project is the *Web Application Project* that was generated. Everything in this project are used to present users with an interface they can interact with, except the optional *CodeExamples* folder which contains *Class Files* for each of the database tables showing code examples on how to access the *Middle-Tier/Business Layers* to do CRUD* operations.

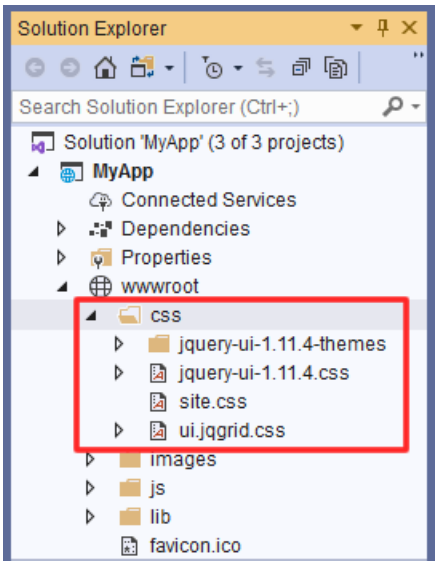
As shown in the *N-Tier Layering* above, the *Front End* (MVC view) accesses the *Middle Tier* (class) to do any kind of operation. Or it can also access the *Web API* instead of the *Middle Tier* (class).

3.1.1 wwwroot

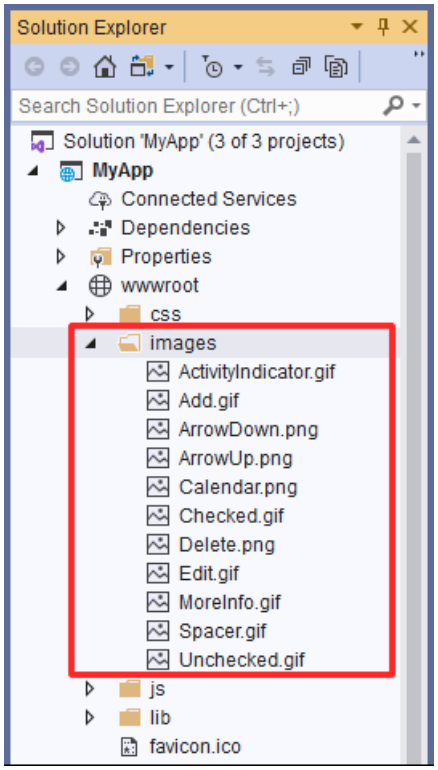
This folder is generally needed by ASP.NET Core MVC as the *Web Root* of the project by default. You can place static files needed by the ASP.NET Core MVC project here. You can add folders and files and name them to whatever you like.



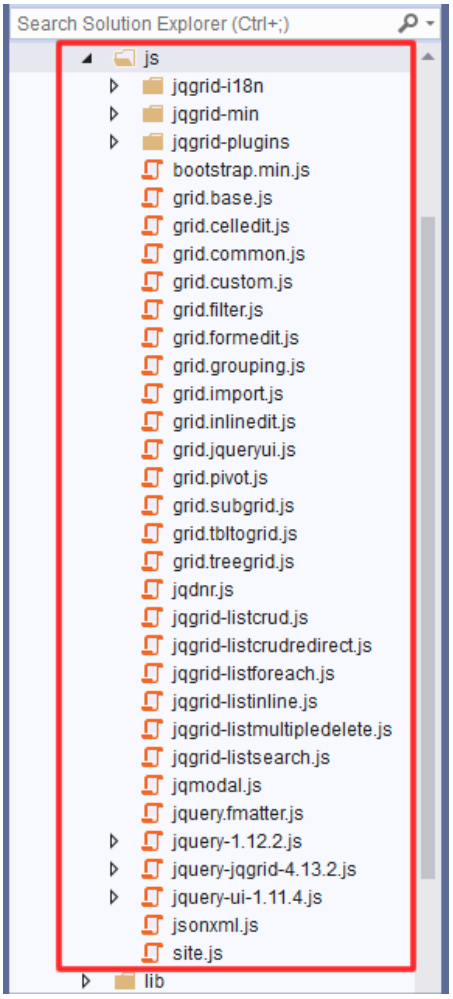
1. **css (folder):** Contains styles including 24 different JQuery-UI themes used by the project. You can add your own stylesheets here. You can also add and updates styles in the *site.css* stylesheet.



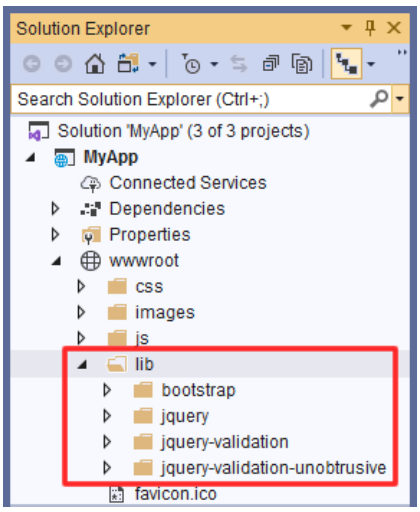
2. **images (folder):** Contains images used by the project. You can add your own images here.



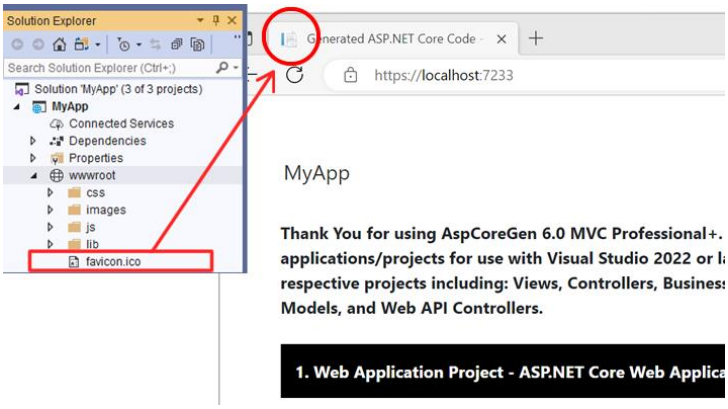
3. **js (folder):** Contains javascript files including JQGrid and JQuery plugins used by the project. You can add your own scripts here.



4. **lib (folder):** Contains libraries, both styles and javascript used by the project. By default, these libraries are included even if you don't use AspCoreGen 6.0 MVC to generate the code. You can add your own libraries here, however, **we recommend that you don't**.



5. **favicon.ico:** An icon used by the browser as the default icon for your project. You can change this to your own icon (brand).

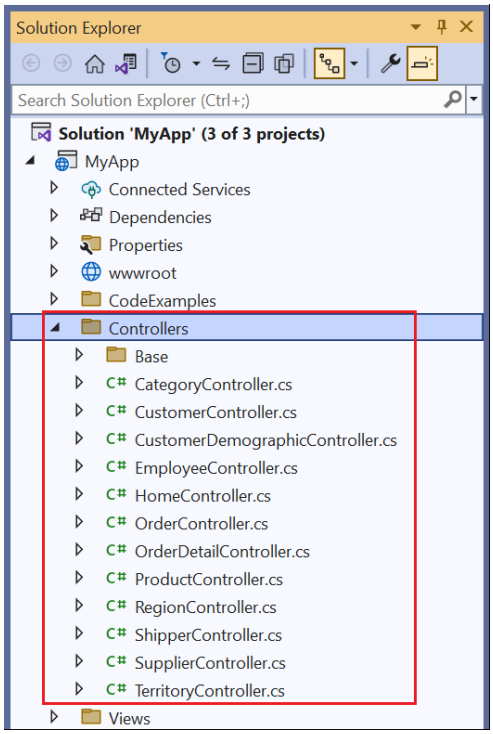


3.1.2 Controllers

This folder is generally needed by ASP.NET Core MVC by default. It houses *Controllers* used by the MVC *Views*. You can add your own *Controllers* here, and you don't need to copy the same layout such as that the *Controllers* generated by AspCoreGen 6.0 MVC. There are **2 *Controllers* with the same name (partial class)** per database table, one directly under the Controllers folder, and the other under the Controllers/Base folder.

You can add your own *Methods* and or *Actions* in any existing *Controller* generated by AspCoreGen 6.0 MVC found directly beneath the Controller folder, these partial classes **will not be overwritten** when you regenerate code for the same project (in this example - *MyApp*). Please see the *AppSettingsTab Tutorial*, page 4 (1.1.1 *Files That Will Be Written Once*) for more information.

Note: Do not add any code in any of the *Controller* generated in the Controller/Base folder. Please see the *AppSettingsTab Tutorial*, page 4 (1.1.2 *Files That Will Always Be Overwritten*) for more information.

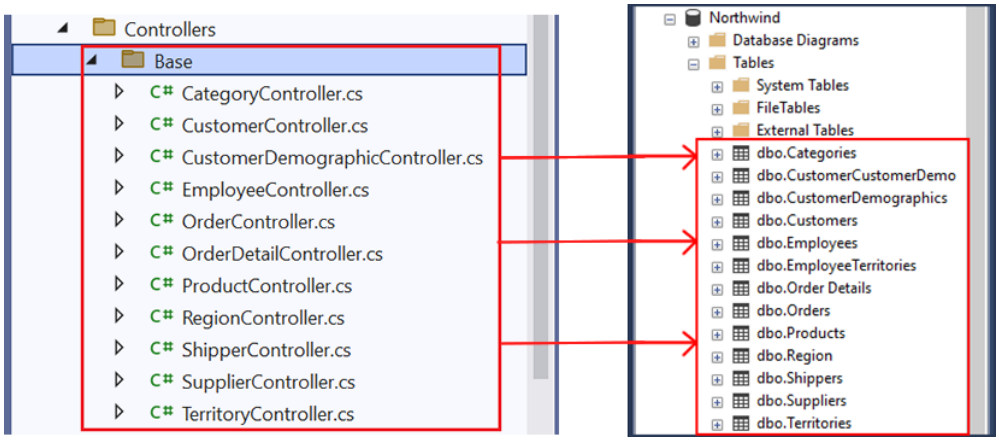


3.1.2.1 The Controller - Used Like A Base Class

Note: Not a base class. The code needed by the Controller are generated in these partial classes. These are the partial class files generated in the *Controllers\Base* folder. The naming convention used is: ***TableName****Controller.cs*.

Do not add any code in these *Partial Class* files.

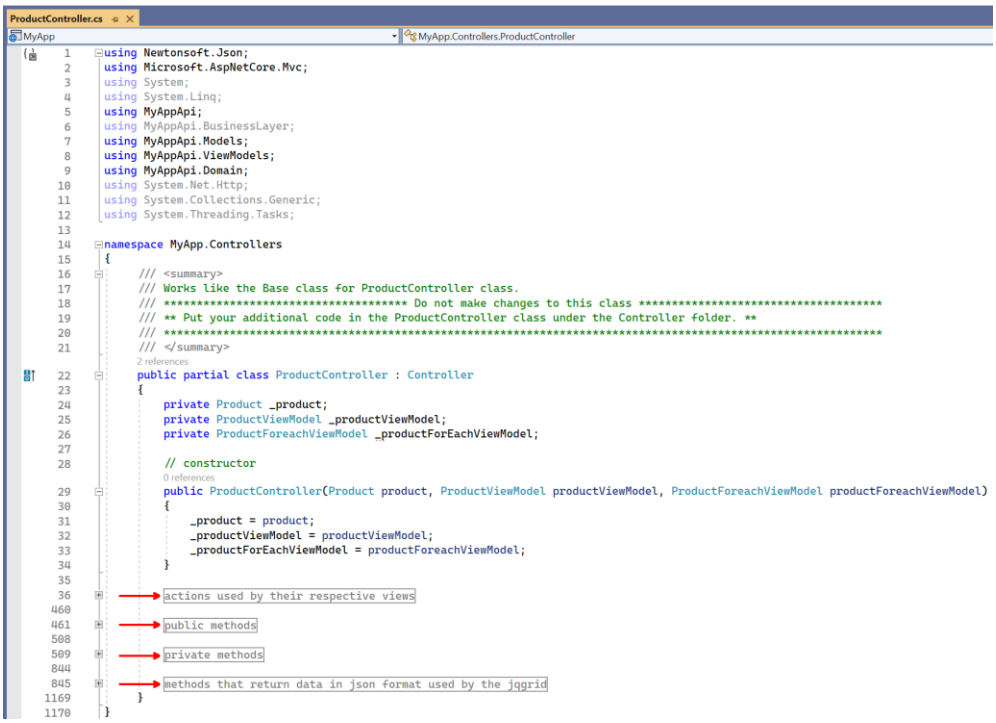
One *Partial Class* (in the Controllers\Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Controllers (Partial Classes) in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

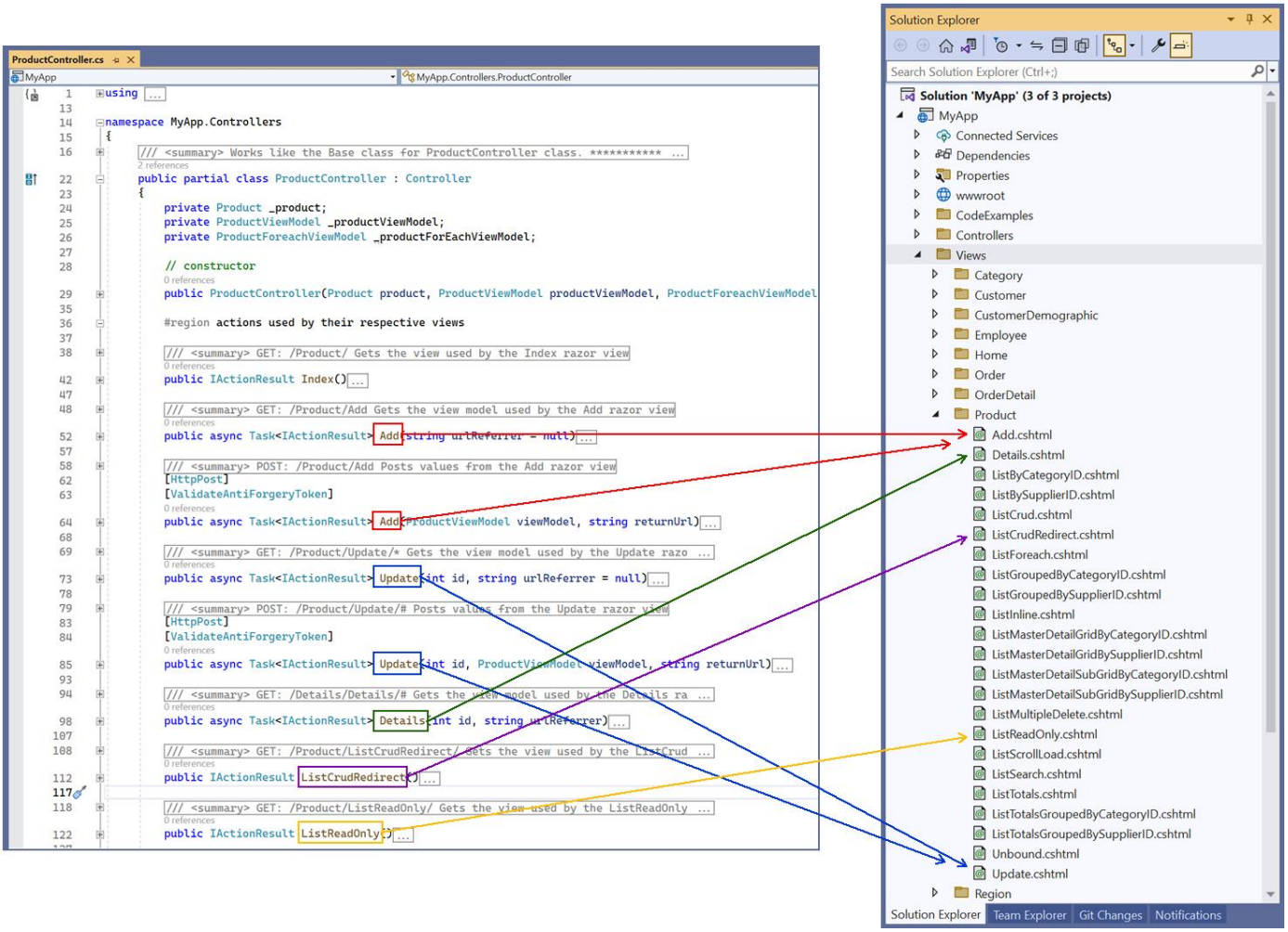
3.1.2.1.1 Code Separated By Regions

Because so much code is generated (depending on the number of *Database Tables* you have), the generated code is separated by *Regions* to classify the type of *Methods* that were generated.



3.1.2.1.1.1 Actions Used by Their Respective Views

These are the *Action* methods used by their respective MVC *Views* under the *Views* folder. The *ProductController* partial class shown below is located in the *Controller**Base* folder.



The ***ProductController*** looks for the ***Product*** folder under *Views* folder by default. The same goes for the MVC *Views* in the *Views* folder under the ***Product*** folder, it will look for the respective action in the ***ProductController***.

For example, the ***Add***.*cshtml View* under the ***Product*** folder will look for an ***Add*** *Action* method in the ***ProductController***. In the same way, the ***ListCrudRedirect***.*cshtml View* under the ***Product*** folder will look for a ***ListCrudRedirect*** *Action* method in the ***ProductController***. So you see the pattern here.

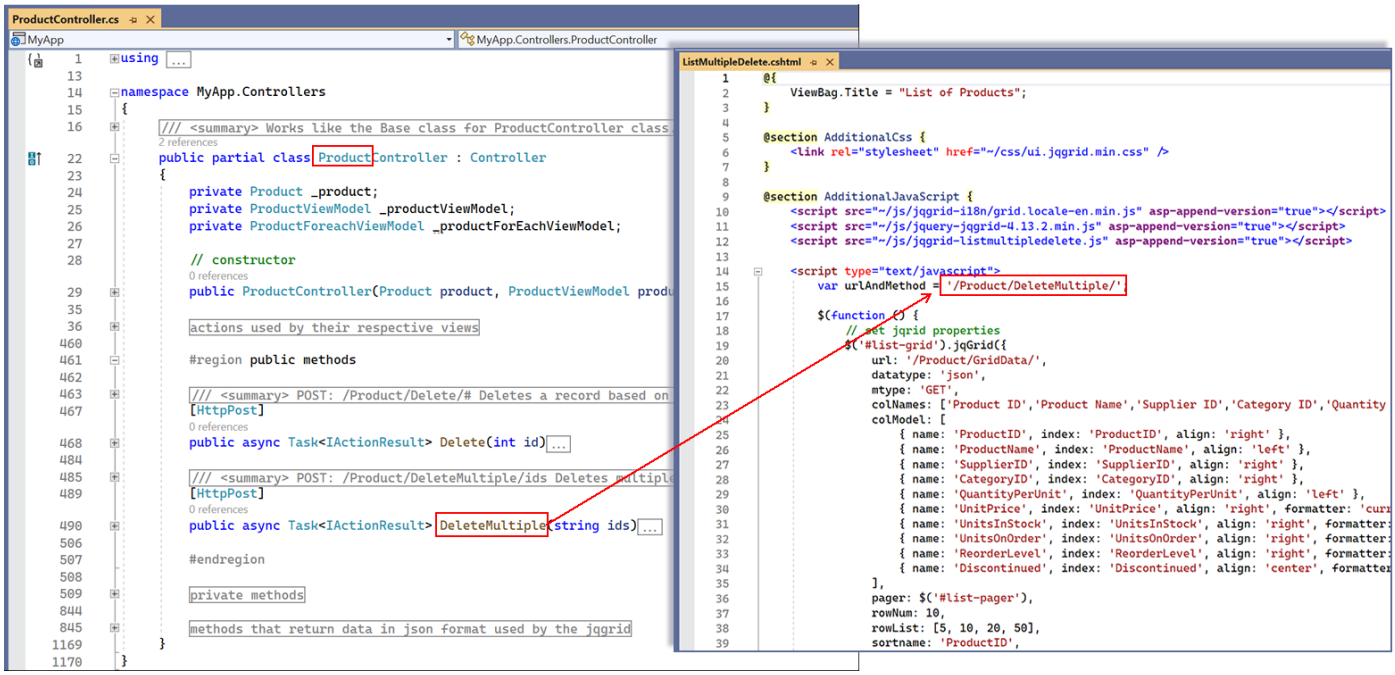
So why does the *Add* and *Update* have 2 *Action* methods each, while the *Details*, *ListCrudRedirect*, *ListReadOnly*, etc. only have 1 *Action* method each? The *Add* and *Update* MVC *Views* both requires a *Get* and *Post Action* methods, while the *Details*, *ListCrudRedirect*, *ListReadOnly* MVC Views only require a *Get Action* method. Each ASP.NET Core MVC *View* require a *Get Action* method minimum by default.

Note: Since both the *ProductController* (in the *Controller\Base* folder) and the *ProductController* (directly under the *Controller* folder) are partial classes with the same name, the MVC View will look at both *ProductController(s)* to find its respective action. If you need to add new code (Actions, methods, etc) that is not generated by AspCoreGen 6.0 MVC, you can **add them in the *ProductController* directly under the *Controller* folder**.

3.1.2.1.1.2 Public Methods

These are *Public* ***HttpPost*** *Web Methods* used by the generated MVC *Views*. These methods are called from a JavaScript client code. You can say that calls to these methods cross from a client (javascript) code to a server code (C#), some calls this AJAX functionality.

For example, the *Delete Multiple* functionality can be found in the generated MVC *View* and related *Controller (*technically the *Controller Base* Class*)* as shown below. When a user deletes multiple items, the generated *ListMultipleDelete.cshtml* MVC *View* looks for the ***ProductController*** (remember this inherits from the *ProductController****Base***) with a *Public DeleteMultiple Method* as highlighted in the MVC *View*'s code below: *'****Product****/DeleteMultiple'*. Code inside the *Controller*'s *DeleteMultiple* method is executed and the control flow is returned back to the calling *ListMultipleDelete.cshtml* MVC *View*.



3.1.2.1.1.3 Private Methods

These are reusable *Private Methods* called by other methods in the *Controller*.

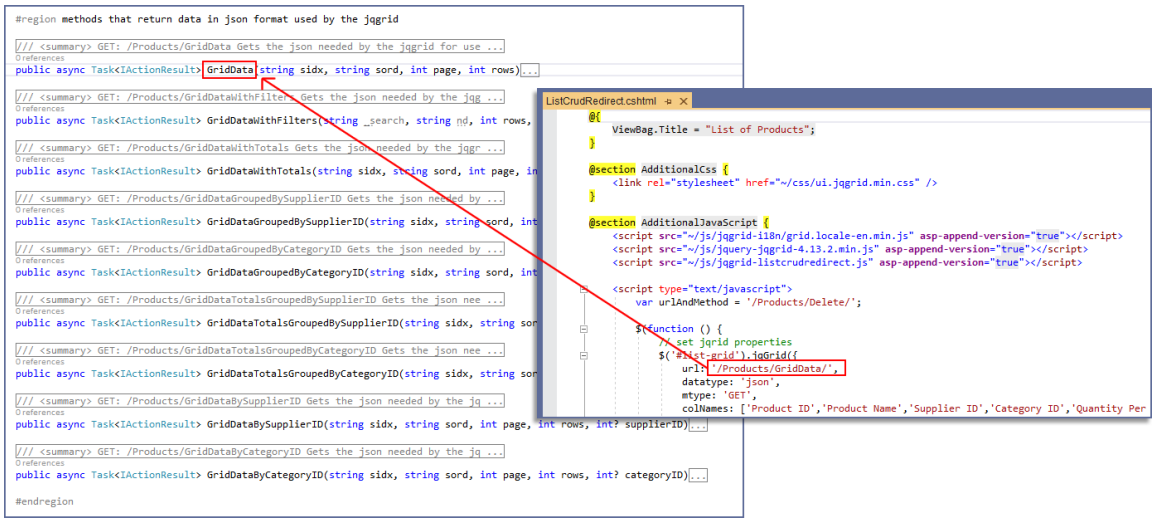
3.1.2.1.1.4 Methods that Return Data in JSON Format Use by the JQGrid

These are *Public* ***HttpGet*** *Web Methods* used by the generated MVC *Views*. These methods are called from a JavaScript client code. You can say that calls to these methods cross from a client (javascript) code to a server code (C#), some calls this AJAX functionality.

HttpGet Web Methods returns data to the calling client. In this instance, the client is a *JQGrid* plugin.

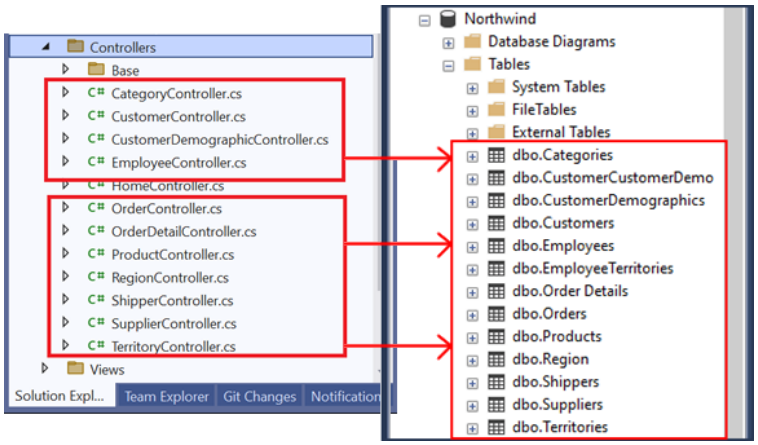
Note: These *Public HttpGet Web Methods* are not exclusively for use with a *JQGrid* client, any client can call them. So you can write your own custom code and call any of these *Public HttpGet Web Methods*.

For example, the *ListCrudRedirect.cshtml* MVC *View* uses the *JQGrid* plugin to pull data from the ***GridData***, a *Public Web Method* in the ***ProductController***.



3.1.2.2 The Controller - Empty

These are the *Partial Classes* generated directly under the *Controllers* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName**Controller.cs*. ASP.NET Core MVC recognizes this as a *Controller* by default because of the suffix "*Controller*" in the name. One *Controller* is generated per *Database Table*. **You can add code in these *Partial Class* files.**

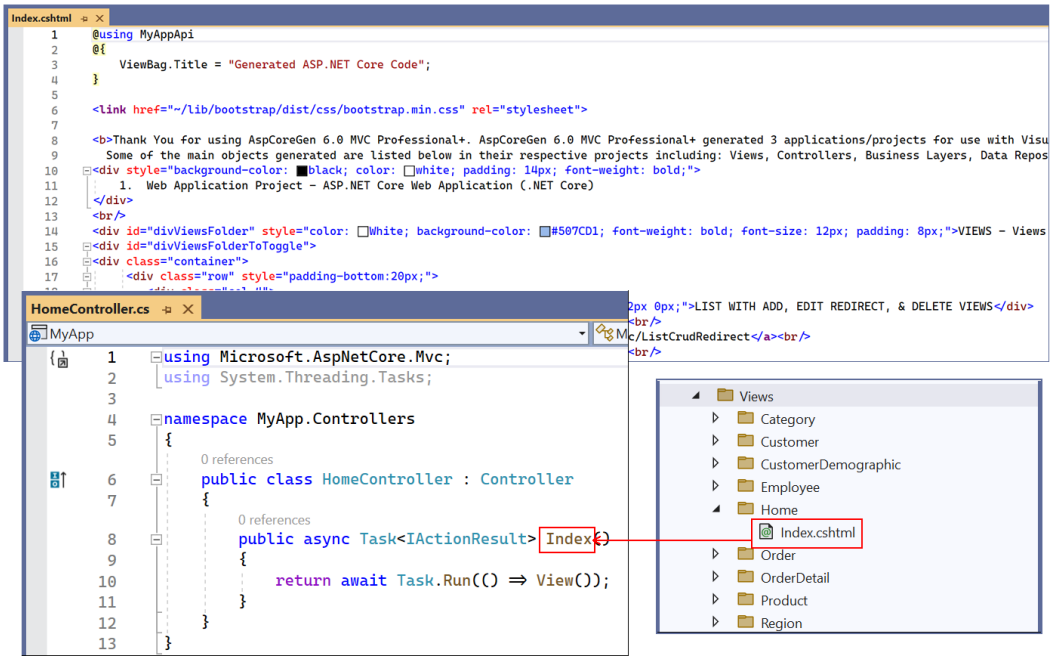


Controllers in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

3.1.2.2.1.1 HomeController

The *HomeController.cs* is unlike the other *Controllers*. It is not generated for a database table, instead, it is used to host the *Index Action Method*.

As shown in the example below, *Index.cshtml View* looks for the related *Index Action Method* in the *HomeController*.



The *Index.cshtml* MVC View is the *Default* page for the generated *Web Application Project*. It is the first page that is launched when you run the generated *Web Application Project* in Visual Studio. It lists all the main

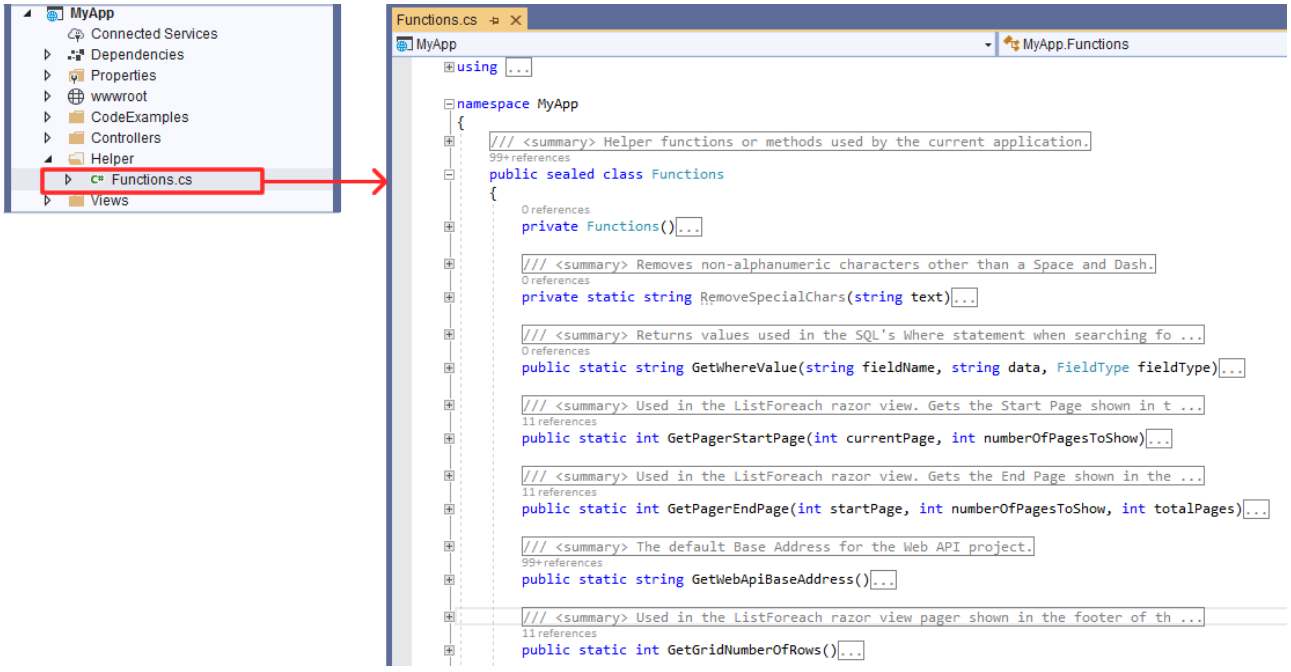
objects generated by AspCoreGen 6.0 MVC. ASP.NET Core MVC looks for an *Index.cshtml View* in the *HomeController* to run as the default page as set up by the generated code in the *Program* class as shown below.

```
Program.cs
MyApp
1   using MyAppApi;
2
3   var builder = WebApplication.CreateBuilder(args);
4
5   // Add services to the container.
6   builder.Services.AddControllersWithViews();
7
8   // register services for dependency injection (di)
9   Functions.AddModelServices(builder.Services);
10  Functions.AddViewModelServices(builder.Services);
11
12  var app = builder.Build();
13
14  // Configure the HTTP request pipeline.
15  if (!app.Environment.IsDevelopment())
16  {
17      app.UseExceptionHandler("/Home/Error");
18      // The default HSTS value is 30 days. You may want to change this
19      app.UseHsts();
20  }
21
22  app.UseHttpsRedirection();
23  app.UseStaticFiles();
24
25  app.UseRouting();
26
27  app.UseAuthorization();
28
29  app.MapControllerRoute(
30      name: "default",
31      pattern: "{controller=Home}/{action=Index}/{id?}");
32
33  app.Run();
```

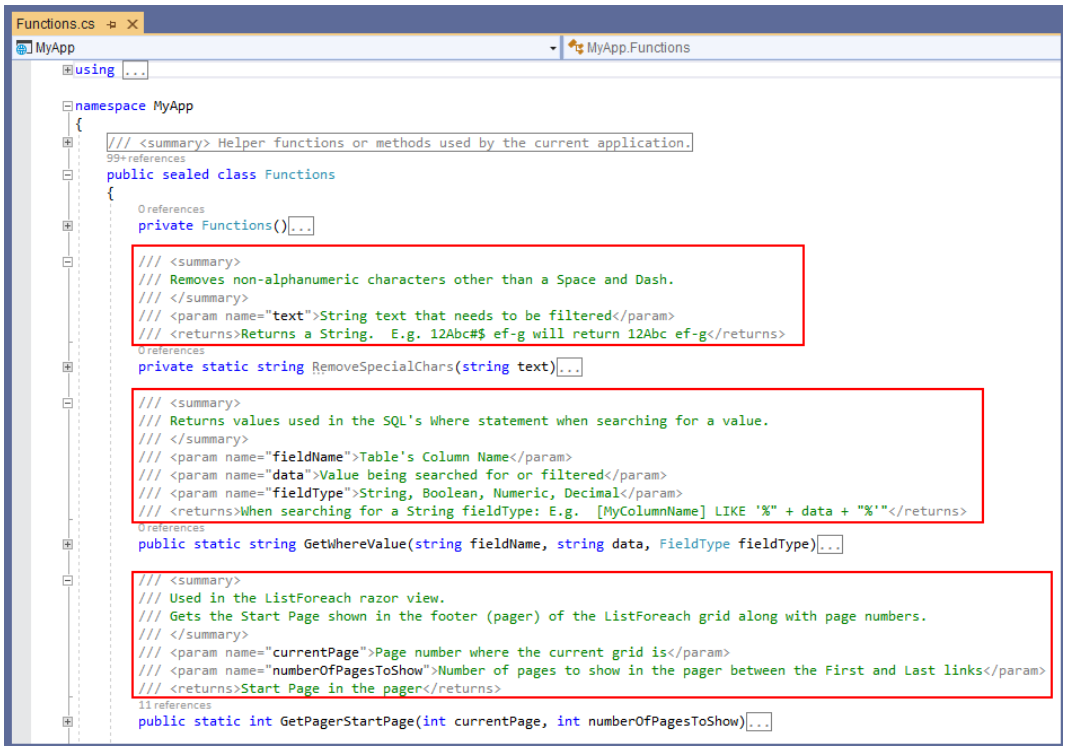
3.1.3 Helper

This folder houses helper *Class*(es).

1. **Functions.cs:** Reusable *Functions* or *Methods* used by the *Front-End* application. **You can add your own code here.**



Read the documentation comments on each one of the methods to learn about their respective functionalities.

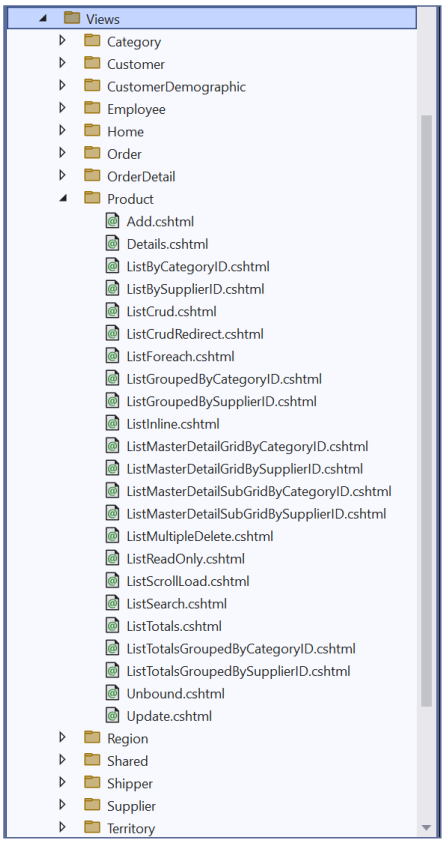


3.1.4 Views

This folder is generally needed by ASP.NET Core MVC by default. It houses MVC *Views*. **You can add your own MVC *Views* here.** All the MVC *Views* generated by AspCoreGen 6.0 MVC will be overwritten the next time you generate code for the same project.

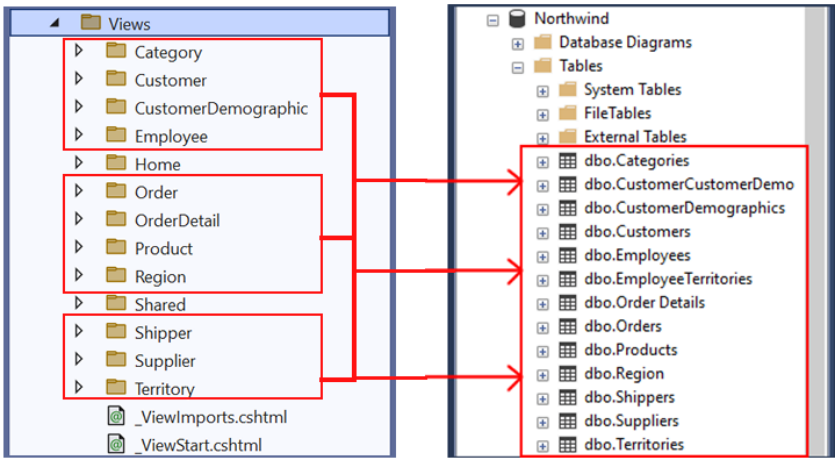
Note: Do not add any code in any of the generated MVC *Views*. Please see the *AppSettingsTab Tutorial*, page 5 (1.1.2 *Files That Will Always Be Overwritten*) for more information.

For more information on the different kinds of MVC *Views* generated by AspCoreGen 6.0 MVC, please see the *UISettingsTab Tutorial* on *Views to Generate*, starting in page 5.



3.1.4.1 Views Generated for Database Tables

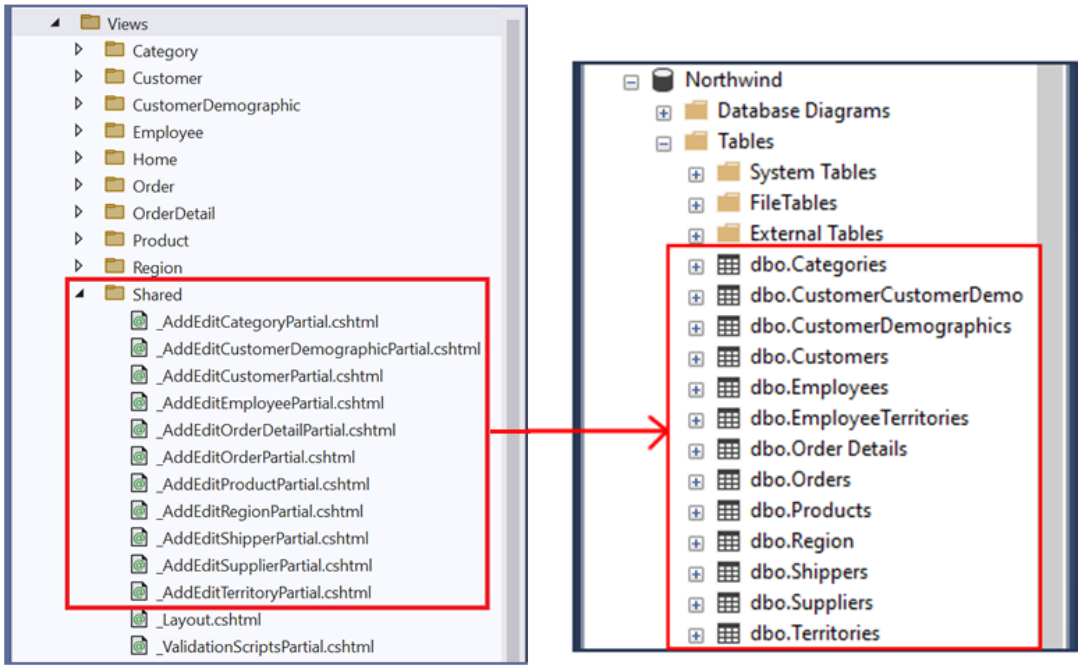
These MVC *Views* are generated based on the *Database Tables* you chose to generate code for. Each *Folder* as shown below is directly related to a *Database Table*.



Views in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

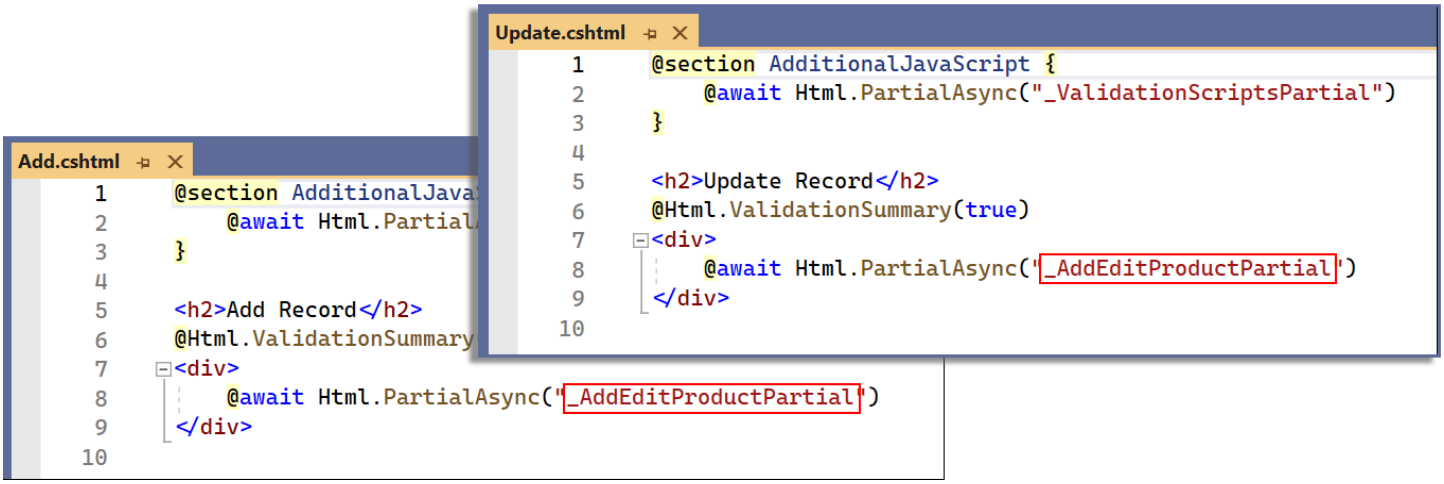
3.1.4.2 Partial Views for Database Tables

These *Partial Views* are generated based on the *Database Tables* you chose to generate code for. Each *Partial View* is directly related to the respective *Database Table* as shown below and has a prefix *"_AddEdit"*. *Partial Views* are located in the *Views/Shared Folder*. The ASP.NET Core MVC naming convention for *Partial Views* starts with an *Underscore* "_" prefix.



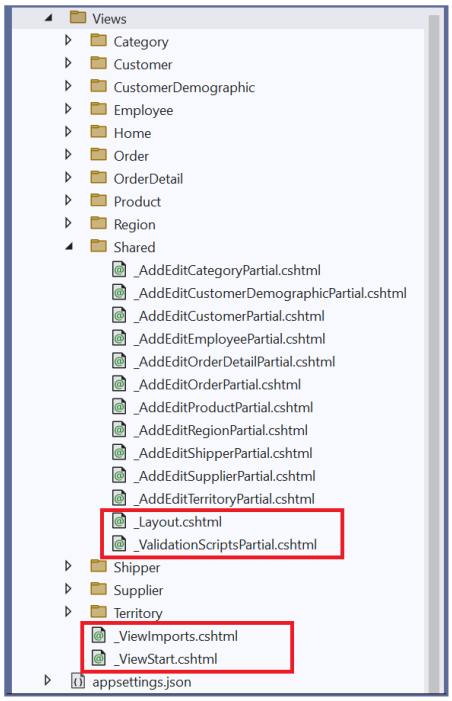
Partial Views in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

Each *Partial View* is used by the *Add.cshtml* and *Update.cshtml* MVC *Views*.



3.1.4.3 Other Partial Views

These are mainly ASP.NET Core MVC default *Partial Views*. The ASP.NET Core MVC naming convention for *Partial Views* starts with an *Underscore* "_" prefix.



3.1.4.3.1 _Layout.cshtml

The *_Layout.cshtml* is a *Partial View* that is the default overall design or master page for all the MVC *Views* that incorporates it. MVC *Views* that incorporate the *_Layout.cshtml* starts it's code base where it shows the *@RenderBody()* code shown below. **You can change the overall design of all the generated MVC *Views* by changing all or a few code here.**

_Layout.cshtml

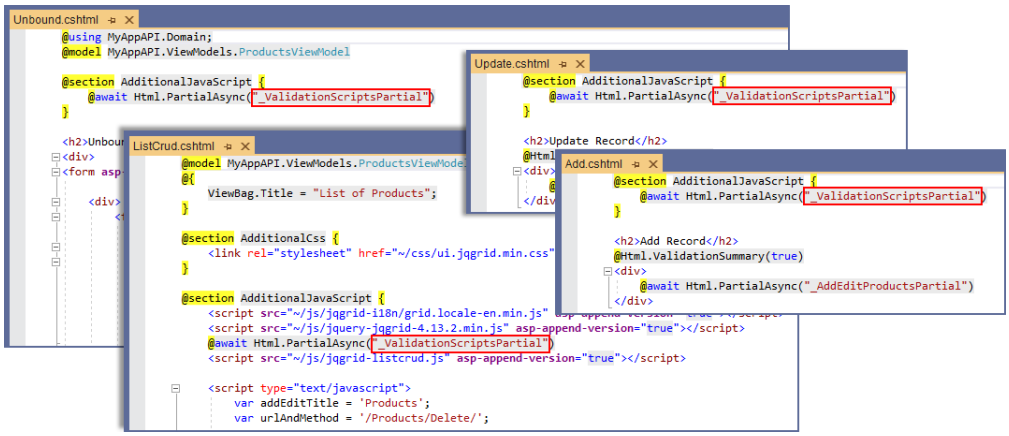
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>@ViewData["Title"] - MyApp</title>
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
        <link rel="stylesheet" href="~/css/site.css" />
        <link rel="stylesheet" href="~/css/jquery-ui-1.11.4-themes/redmond/jquery-ui.min.css" />
        <link rel="stylesheet" href="~/css/jquery-ui-1.11.4-themes/redmond/theme.css" />
        @RenderSection("AdditionalCss", required: false)
    </head>
    <body>
        <div class="navbar navbar-inverse navbar-fixed-top">
            <div class="container">
                <div class="navbar-header">
                    <a asp-controller="Home" asp-action="Index" class="navbar-brand">MyApp</a>
                </div>
                <div class="navbar-collapse collapse">
                </div>
            </div>
        </div>
        <br />
        <div class="container body-content">
            @RenderBody()
            <hr />
            <footer>
                <p>&copy; @DateTime.Now.Year - MyApp</p>
            </footer>
        </div>

        <script src="~/js/jquery-1.12.2.min.js"></script>
        <script src="~/lib/bootstrap/dist/js/bootstrap.min.js"></script>
        <script src="~/js/jquery-ui-1.11.4.min.js" asp-append-version="true"></script>
        @RenderSection("AdditionalJavaScript", required: false)
    </body>
</html>
```

3.1.4.3.2 _ValidationScriptPartial.cshtml

The *_ValidationScriptPartial.cshtml* is a *Partial View* that references javascript (jQuery) libraries for use when validating controls for errors. **You can add your own code here**.

```
_ValidationScriptsPartial.cshtml
<environment names="Development">
    <script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.14.0/jquery.validate.min.js"
            asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
            asp-fallback-test="window.jQuery && window.jQuery.validator">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unobtrusive.min.js"
            asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"
            asp-fallback-test="window.jQuery && window.jQuery.validator && window.jQuery.validator.unobtrusive">
    </script>
</environment>
```

It is used by MVC *Views*: *Add.cshtml, Update.cshtml, Unbound.cshtml,* and *ListCrud.cshtml* as shown below.



3.1.4.3.3 _ViewImports.cshtml

This *Partial View* imports directives that can be shared throughout all the generated MVC *Views*. **You can add your own code here**.

```
_ViewImports.cshtml
@using MyApp
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

3.1.4.3.4 _ViewStart.cshtml

By default, this *Partial View* is ran before any MVC *View*. **You can add your own code here**.

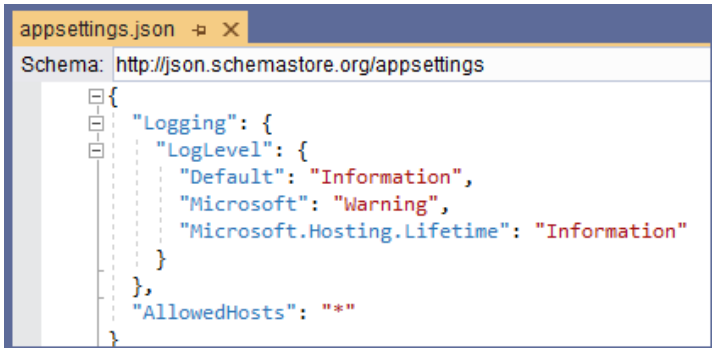
```
_ViewStart.cshtml
@{
    Layout = "_Layout";
}
```

3.1.4.4 Index.cshtml View

This MVC *View* is the default page of the *Web Application Project*. The *Index Action Method* can be found in the *HomeController*. Please read about the *HomeController* in page 14 for more information on the *Index View*.

3.1.5 appsettings.json

This is a settings json file used by the ASP.NET MVC Core *Web Application Project* by default. **You can add your own code here**.



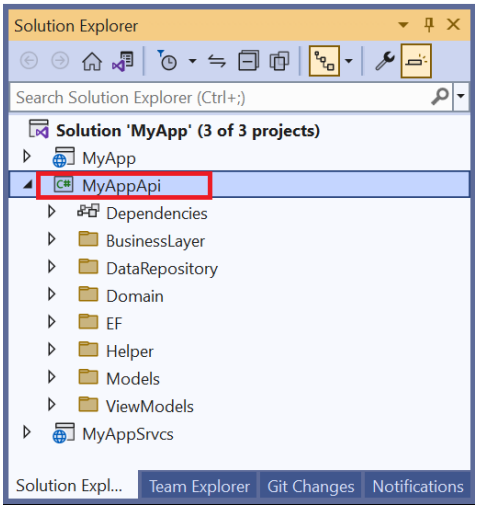
3.1.6 Program.cs

The *Program.cs Class* is the entry point to the ASP.NET MVC Core *Web Application Project* by default. An ASP.NET MVC Core web application project is technically a *Console* app. Just like any *Console* app, execution of the app starts at the *Program Class's* ***Main()*** *Method.* **You can add your own code here**.

3.2 Middle Layer Project (Business Layer, Data Repository, Shared Libraries)

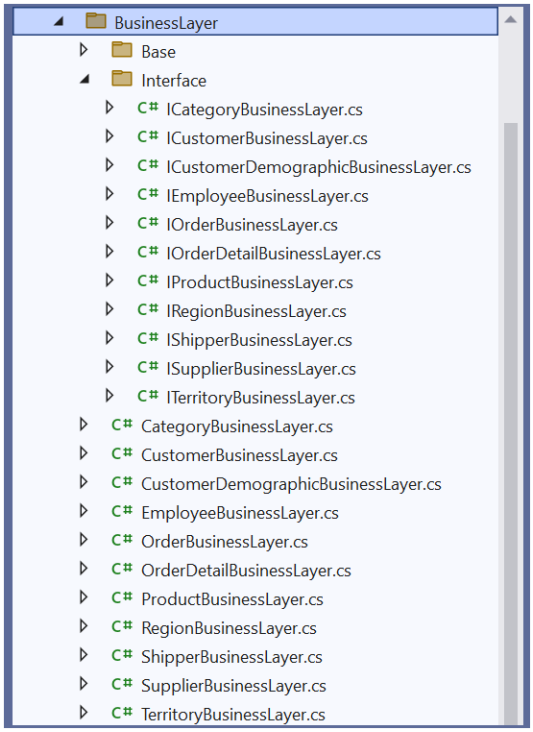
The generated *Middle Layer Project* contains the *Middle-Tier* and *Data-Tier* part of the N-tier layer generated code (and shared libraries as well). This is a *Class Library* project. ***Classes/Interfaces* here can be reused by other projects/clients.**

The *Middle Layer Project* is referenced by the *Web Application Project* and *Web API Project* for use. You can also reference this project from other projects that you add to the generated *Solution* or altogether copy the whole project to your own custom projects, and many more possibilities for reuse.



3.2.1 Business Layer (Middle Tier)

The *Business Layer (Middler Tier) Interface and Class Files* are located in the *BusinessLayer Folder*.



The *Business Layer's* (or *Middle Tier*) main purpose is to serve as a client's (a program's) only access to the Business Layer. The *Business Layer's* purpose is to calculate things. For the purposes of AspCoreGen 6.0 MVC code generation, in most parts, there really is nothing to calculate, instead, the *Business Layer* classes simply returns data handed to it by the *Data Repository* (Data Tier), or carries and passes the CRUD* operations that the *Data Repository* need to handle.

The *Calculations* we are talking about here are not just math problems, instead, these are logic that the *Client* (controller, asp.net web form, web api, wcf program etc.) needs. For most parts, any *Client* should not be doing any kind of *Calculation*, instead, a line code referencing a *Business Layer Class's Method* should readily return that logic.

For example (just an example and not generated by AspCoreGen 6.0 MVC), let's say somewhere in the *Controller* it needs the full name of a person.

*var fullName = **User**.GetFullName();*

In this example, *"User"* is the *Business Layer (Middle-Tier Class)*, "*GetFullName()"* is a *Public Method* in the *"User" Business Layer Class.* Somewhere in the "*GetFullName()" Method,* it's calculating the first name and last name of the user, return a full name, e.g.

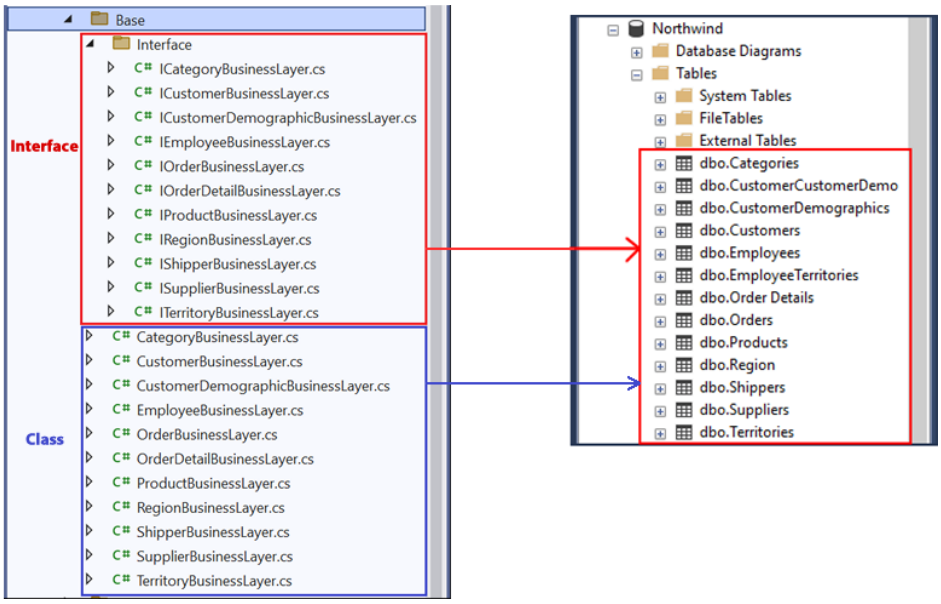
return User.FirstName + " " + User.LastName;

3.2.1.1 Partial Interface/Partial Class – Used Like A Base Interface/Class

These are the interface and class files generated in the *BusinessLayer\Base* folder.

Do not add any code in these *Interface* and *Class* files.

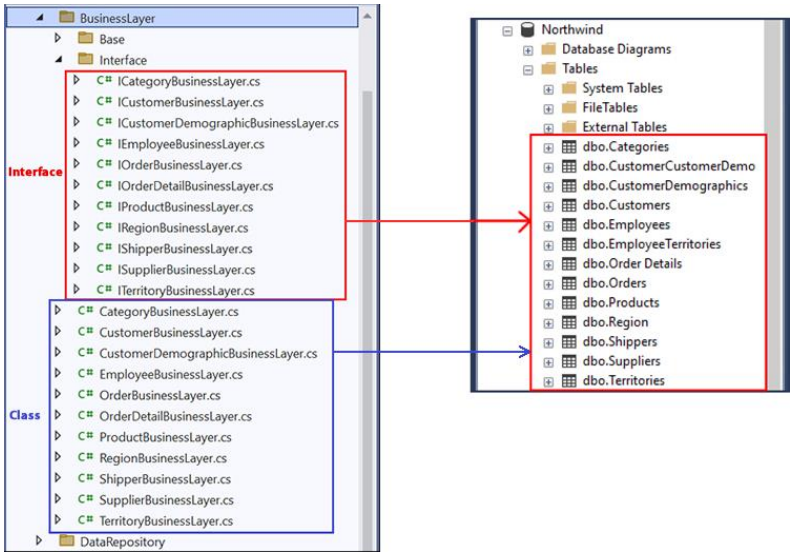
One *Partial Interface and Class* (in the Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Interfaces/Classes in Visual Studio under Base Folder (Left) – Database Tables in MS SQL Server (Right)

3.2.1.2 Business Layer - Empty

These are the interface and class files generated directly under the *BusinessLayer* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName***BusinessLayer.*cs*. One *Business Layer interface* and *class* is generated per *Database Table*.



Interfaces/Classes in Visual Studio directly under BusinessLayer Folder (Left) – Database Tables in MS SQL Server (Right)

You can add code in these *Interface* and *Class* files. You access all the *Business Layer* methods and properties using these *interface*s and *classes*.

These are the *Interfaces/Classes* that **any client** should access. You can also access the *Web API Project's* public methods when you generate the optional *Web API* project.

Note 1: When you generate the optional *Web API Project,* AspCoreGen 6.0 MVC's generated code will always access *Web API Methods* from clients like the *Controller* class. These *Web API Methods* encapsulates calls to the related/respective *Business Layer Methods* as shown in the *N-Tier Layering* in page 4.

Note 2: You don't always have to access the *Web API Methods* (from any client) generated by AspCoreGen 6.0 MVC, you can also access the *Business Layer Classes* directly if you want to. Again, please refer to *Note 1* above.

3.2.2 Data Repository (Data Tier)

The *Data Repository*'s (or *Data Tier*) main purpose is to interact with the database. It does all the CRUD* operations.

Note 1: *Data Repository* is called by the *Business Layer*, and once the CRUD operation is done it returns the control back to the *Business Layer*.

Note 2: Most of the time, **a *Data Respository Class* should only be called by their respective *Business Layer Class*.** *Data Repository Interfaces/Classes* have an "*internal*" access modifier to prevent clients outside of the *Middle Layer Project* access.

Note 3: Since each *Data Repository Interface/Class* have an "*internal*" access modifier, any (*Interface*/*Class, Method*) code you create in the *Business Layer and Data Repository API Project* will be able to access these objects. Again, no *Interface/Class* should access a *Data Repository Interface/Class* other than a *Business Layer Interface/Class*, please see *Note 1*.

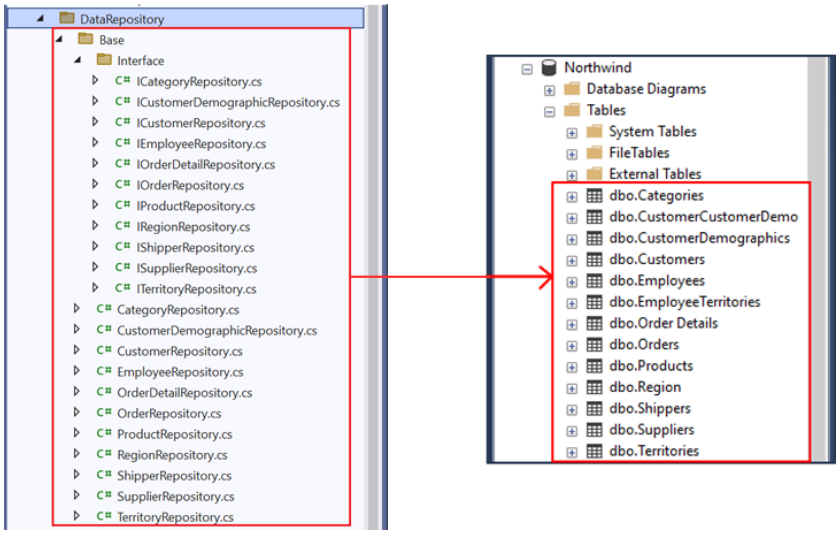
These *Data Repository* (*Data Tier*) *Interface*/*Class Files* are located in the *DataRepository Folder*.

3.2.2.1 Partial Interface/Partial Class – Used Like A Base Interface/Class

These are the interface and class files generated in the *DataRepository**Base* folder.

Do not add any code in these *Interface* and *Class* files.

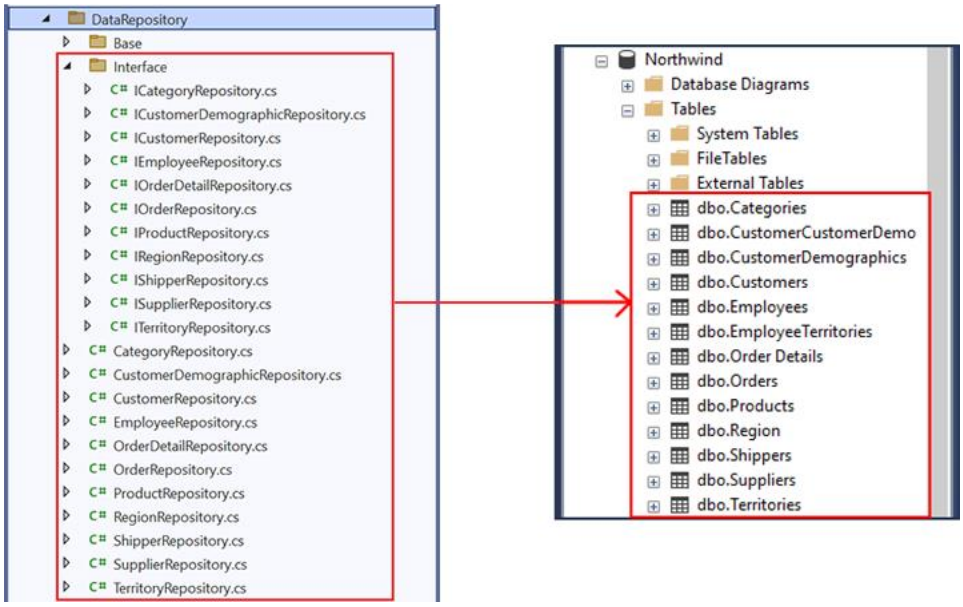
One *Partial Interface and Class* (in the Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Interfaces/Classes in Visual Studio under Base Folder (Left) – Database Tables in MS SQL Server (Right)

3.2.2.2 Data Repository - Empty

These are the interface and class files generated directly under the *DataRepository* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName***DataRepository.*cs*. One *Data Repository interface* and *class* is generated per *Database Table*.



Interfaces/Classes in Visual Studio directly under DataRepository Folder (Left) – Database Tables in MS SQL Server (Right)

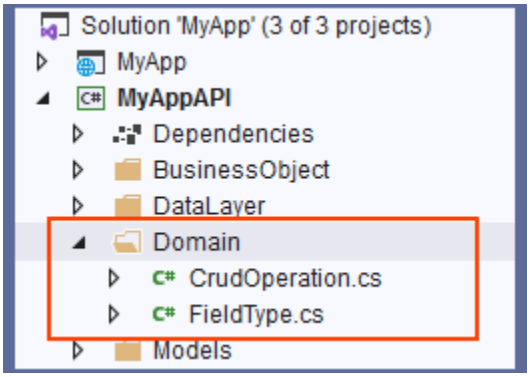
You can add code in these *Interface/Class* files. You access all the *Data Repository* methods and properties using this *Interface/Class.*

These are the *Interfaces/Classes* that *Business Layer Interfaces/Classes* should access.

Note 1: Only a *Business Layer Interface/Class* should access their respective *Data Repository Class*.

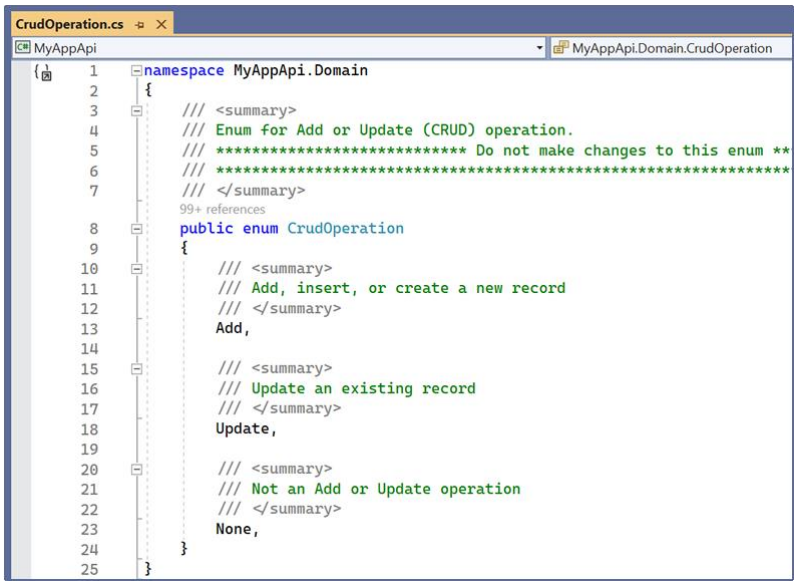
3.2.3 Domain

The *Domain Folder* contains 2 reusable *enum* type objects; the *CrudOperation.cs* and *FieldType.cs*.



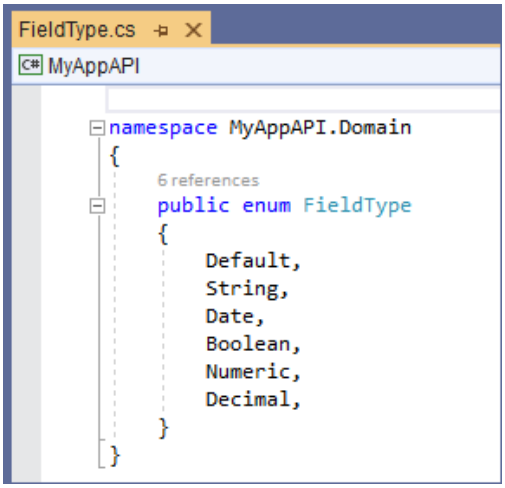
3.2.3.1 CrudOperation.cs

The *CrudOperation enum* is used to determine whether an *Add* or *Update* operation needs to be handled. When not doing an *Add* or *Update* operation, use *None*.



3.2.3.2 FieldType.cs

The *FieldType enum* is used to determine a field's type before executing an operation.

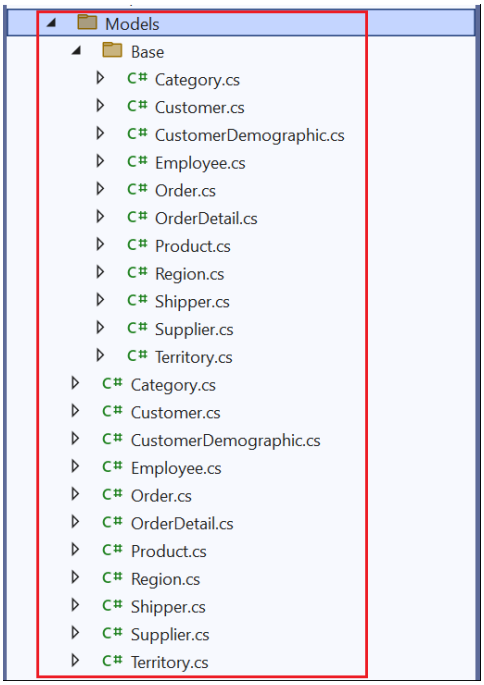


3.2.4 Models

These are *Classes* that contains *Properties* for each of the *Database Table* you generated code for. A *Property* is equivalent to a *Field* or *Column* in their respective *Database Table.* *Models* is the *"M"* in MVC. Sometimes *Models* are misinterpreted as the *Data Tier* part of MVC, in this case, it is not.

So why are *Models* generated in the *Middle Layer Project* instead of the *Web Application Project* where the MVC *Views* and *Controllers* are generated in (after all it's called Models, Views, Controllers, hence MVC)? Simple, ***Models* are reusable**.

Models are located in the *Models Folder*.

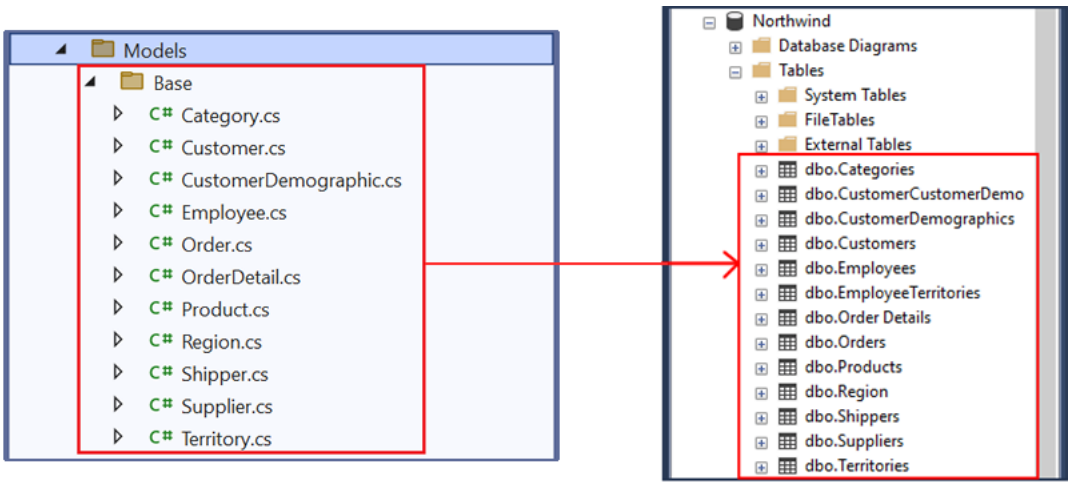


3.2.4.1 Partial Class – Used Like A Base Class

These are the class files generated in the *Models\Base* folder.

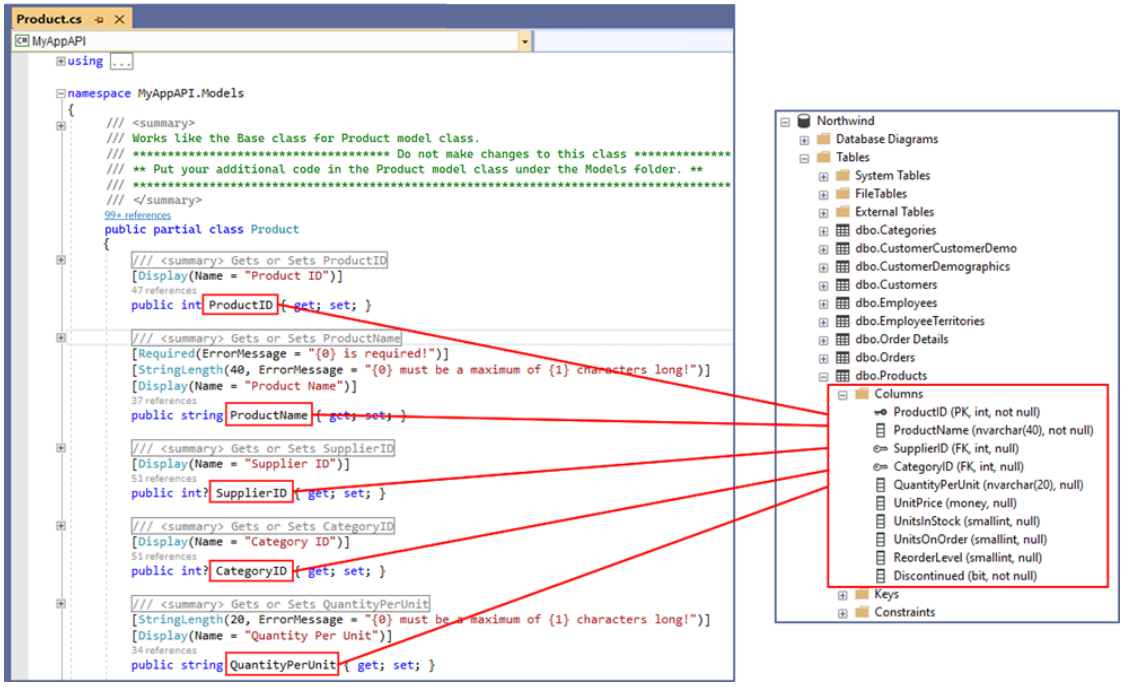
Do not add any code in these *Class* files.

One *Partial Class* (in the Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Classes in Visual Studio under Base Folder (Left) – Database Tables in MS SQL Server (Right)

Here's an example of the *Products Table Columns* (*Fields*) in the *Northwind* database in relation to the generated *Model*.

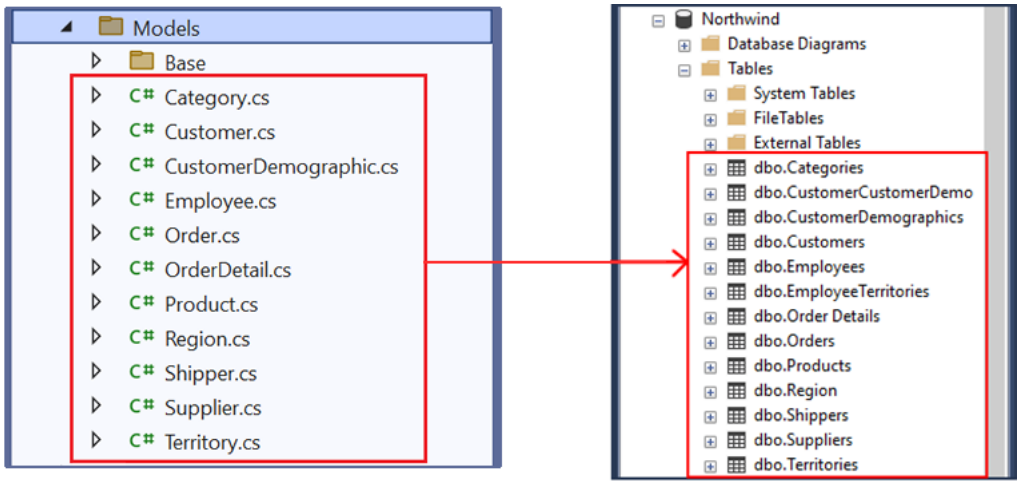


Partial Model Class in Visual Studio under Base Folder (Left) – Product Database Table Columns in MS SQL Server (Right)

3.2.4.2 Models - Empty

These are the class files generated directly under the *Models* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName***Model.*cs*. One *Model class* is generated per *Database Table*.

You can add code in these *Class* files.



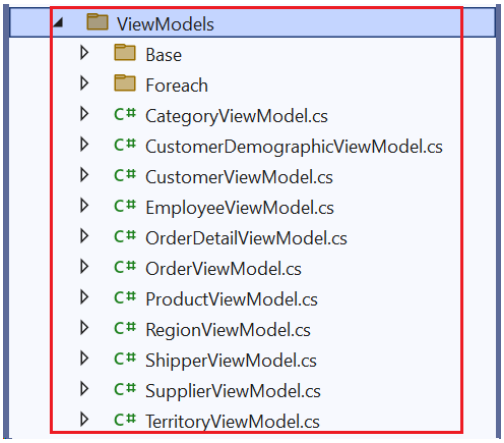
Classes in Visual Studio directly under Models Folder (Left) – Database Tables in MS SQL Server (Right)

3.2.5 View Models

These are *Classes* that contains models (properties) used by MVC *Views*, hence the name *ViewModels*.

So why are *ViewModels* generated in the *Middle Layer Project* instead of the *Web Application Project* where the MVC *Views* and *Controllers* are generated in? Simple, ***ViewModels* are reusable**.

ViewModels are located in the *ViewModels Folder*.

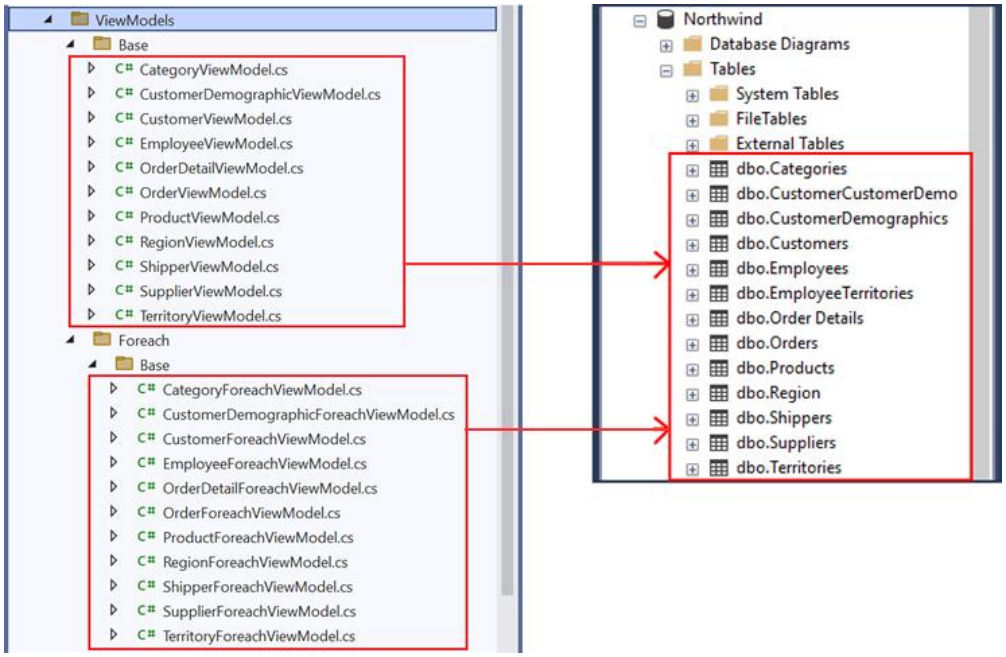


3.2.5.1 Partial Class – Used Like A Base Class

These are the class files generated in the *ViewModels**Base* folder.

Do not add any code in these *Class* files.

One *Partial Class* (in the Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Partial View Model Class in Visual Studio under Base Folder (Left) – Product Database Table Columns in MS SQL Server (Right)

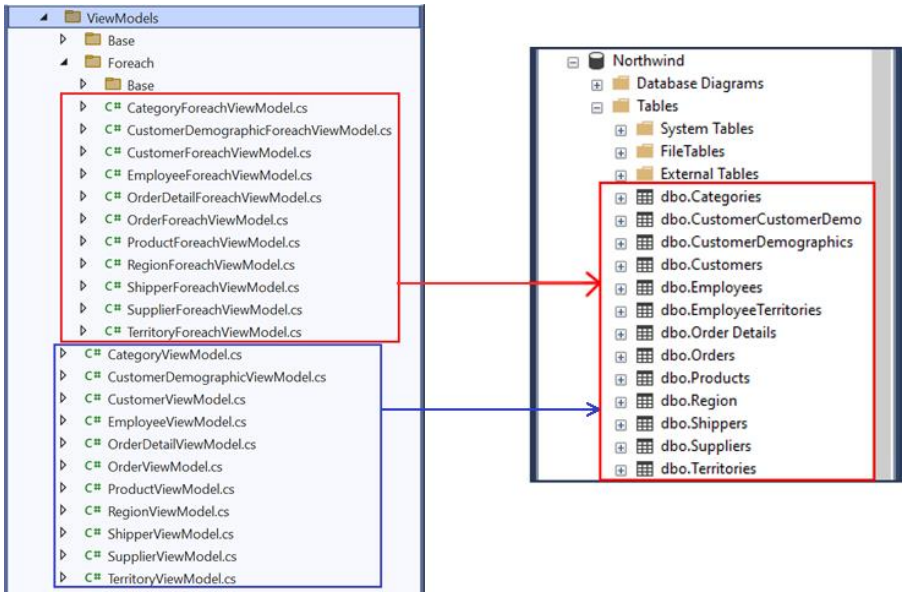
Here's an example of the *ProductViewModel* code.

```
ProductViewModel.cs
MyAppApi                                   MyAppApi.ViewModels.ProductViewModel
 1   using ...
 4
 5   namespace MyAppApi.ViewModels
 6   {
 7       /// <summary>
 8       /// Works like the Base class for ProductViewModel.
 9       /// ********************************** Do not make changes to this class **********
10       /// ** Put your additional code in the ProductViewModel class under the ViewModel folder. **
11       /// ***********************************************************************************
12       /// </summary>
         34 references
13       public partial class ProductViewModel
14       {
15           /// <summary> The model used by the MVC view
             99+ references
18           public MyAppApi.Models.Product Product { get; set; }
19
20           /// <summary> Add new record or Update existing record
             3 references
23           public CrudOperation Operation { get; set; }
24
25           /// <summary> Controller Name used by the MVC view
             3 references
28           public string ViewControllerName { get; set; }
29
30           /// <summary> Action Name used by the MVC view
             3 references
33           public string ViewActionName { get; set; }
34
35           /// <summary> URL where the current MVC view redirects to after the operation.
             8 references
38           public string ViewReturnUrl { get; set; }
39
40           /// <summary> Data used by the Supplier drop down list control
             13 references
43           public List<Models.Supplier> SupplierDropDownListData { get; set; }
44
45           /// <summary> Data used by the Category drop down list control
             13 references
48           public List<Models.Category> CategoryDropDownListData { get; set; }
49       }
50   }
```

3.2.5.2 View Model - Empty

These are the class files generated directly under the *ViewModels* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName***ViewModel.*cs*. One *ViewModel class* is generated per *Database Table*.

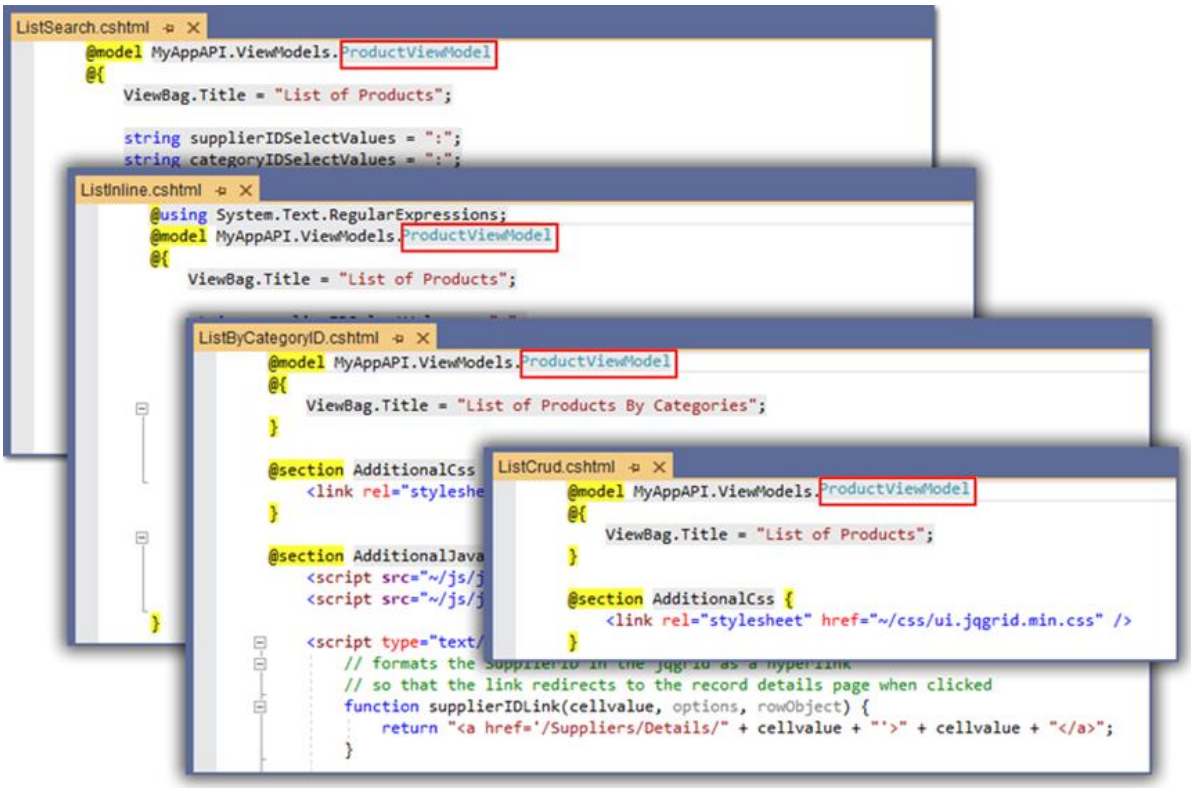
You can add code in these *Class* files.



Classes in Visual Studio directly under ViewModels Folder (Left) – Database Tables in MS SQL Server (Right)

These *ViewModels (ProductViewModel* in the example*)* are referenced and used by the following MVC *Views*:

1. *ListSearch.cshtml*
2. *ListInline.cshtml*
3. *ListCrud.cshtml*
4. *ListBy**ForeignKey**.cshtml*

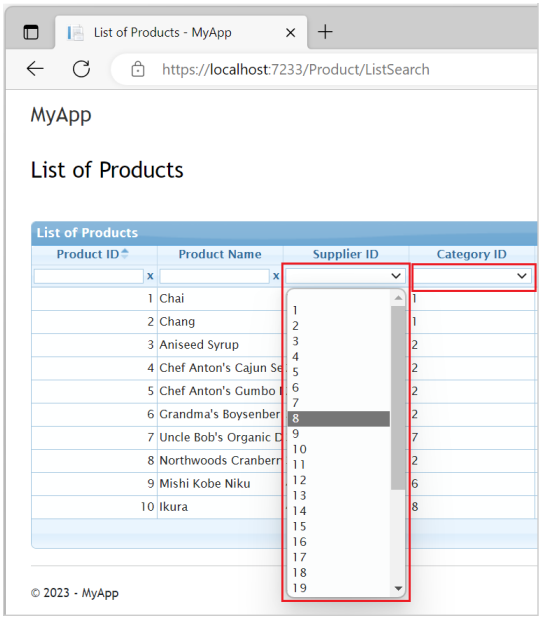


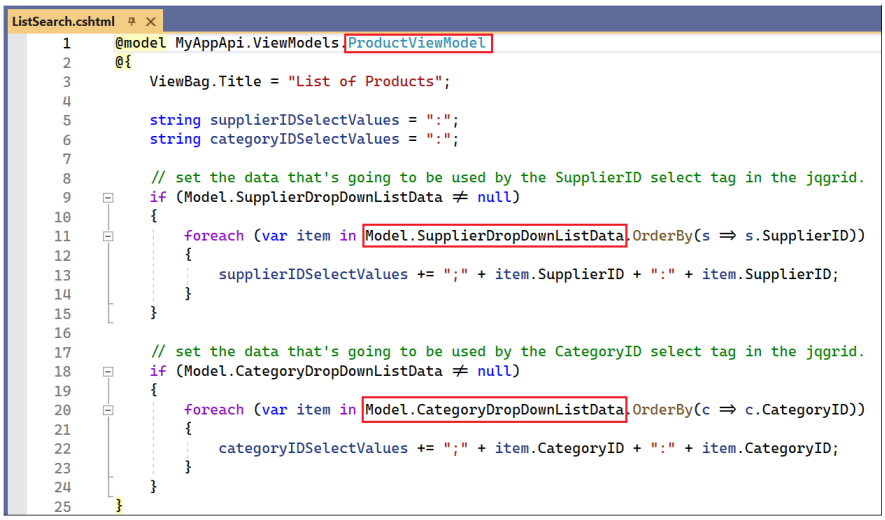
Note: By default ASP.NET MVC *Views* use "***Model***" as a *Keyword*. Also by default, you can set the MVC *View*'s *Model* following the *@model* directive. Here's an example on how to set an MVC *View*'s *Model:*

```
@model MyAppApi.ViewModels.ProductForeachViewModel
```

3.2.5.2.1 ListSearch.cshtml

This MVC *View* uses the *ProductViewModel* as its *Model*. It uses the MVC *View*'s *Model* (*ProductViewModel*) to fill the *Select* tags for the *SupplierID* and *CategoryID* using the MVC *View*'s *Model*, the *SupplierDropDownListData* and *CategoryDropDownListData* respectively, these are *Properties* of the *ProductViewModel* as shown in the code example in page 32.





The *ViewModel* used by the *ListSearch.cshtml View* is assigned from the respective *Get Action* method found in the *Controller (Base)*, it is then injected to the *ListSearch.cshtml View.* See code example below.

```
/// <summary>
/// GET: /Product/ListSearch/
/// Gets the view model used by the ListSearch razor view
/// </summary>
0 references
public async Task<IActionResult> ListSearch()
{
    // return view model
    _productViewModel = await this.GetViewModelAsync("ListSearch");
    return View(_productViewModel);
}

/// <summary>
/// Gets the view model based on the actionName
/// </summary>
8 references
private async Task<ProductViewModel> GetViewModelAsync(string actionName,
string controllerName = "Product", string returnUrl = null, Product objProduct = null,
CrudOperation operation = CrudOperation.None, bool isFillSupplierDdl = true, bool isFillCategoryDdl = true)
{
    // assign values to the view model
    _productViewModel.Product = objProduct;
    _productViewModel.Operation = operation;
    _productViewModel.ViewControllerName = controllerName;
    _productViewModel.ViewActionName = actionName;
    _productViewModel.ViewReturnUrl = returnUrl;

    if (isFillSupplierDdl)
        _productViewModel.SupplierDropDownListData = await this.GetSupplierDropDownListDataAsync();
    else
        _productViewModel.SupplierDropDownListData = null;

    if (isFillCategoryDdl)
        _productViewModel.CategoryDropDownListData = await this.GetCategoryDropDownListDataAsync();
    else
        _productViewModel.CategoryDropDownListData = null;

    // return the view model
    return _productViewModel;
}
```

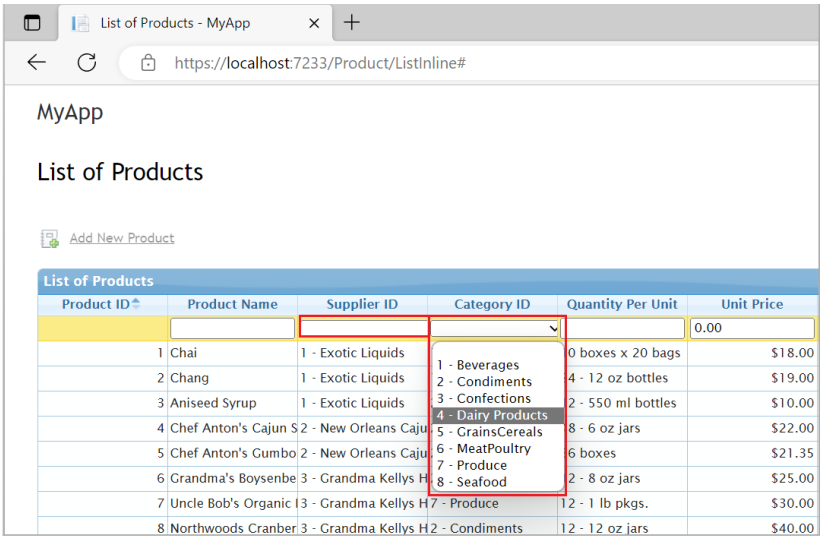
Action

Injecting the ViewModel to the MVC View

3.2.5.2.2 ListInline.cshtml

This MVC *View* uses the *ProductViewModel* as its *Model*.

The *ListInline.cshtml* uses the MVC *View*'s *Model* (*ProductViewModel*) to fill the *Select* tags for the *SupplierID* and *CategoryID* in the dialog shown below using the MVC *View's Model*, the *SupplierDropDownListData* and *CategoryDropDownListData* respectively, these are *Properties* of the *ProductViewModel* as shown in 32.



ListInline.cshtml

```
@using System.Text.RegularExpressions;
@model MyAppApi.ViewModels.ProductViewModel
@{
    ViewBag.Title = "List of Products";

    string supplierIDSelectValues = ":";
    string categoryIDSelectValues = ":";

    // set the data that's going to be used by the SupplierID select tag in the jqgrid.
    if (Model.SupplierDropDownListData != null)
    {
        foreach (var item in Model.SupplierDropDownListData.OrderBy(s => s.SupplierID))
        {
            supplierIDSelectValues += ";" + item.SupplierID + ":" + item.SupplierID + " - " + Regex.Replace
        }
    }

    // set the data that's going to be used by the CategoryID select tag in the jqgrid.
    if (Model.CategoryDropDownListData != null)
    {
        foreach (var item in Model.CategoryDropDownListData.OrderBy(c => c.CategoryID))
        {
            categoryIDSelectValues += ";" + item.CategoryID + ":" + item.CategoryID + " - " + Regex.Replace
        }
    }
}
```

The *ViewModel* used by the *ListInline.cshtml View* is assigned from the respective *Get Action* method found in the *Controller* it is then injected to the *ListInline.cshtml View.* See code example below.

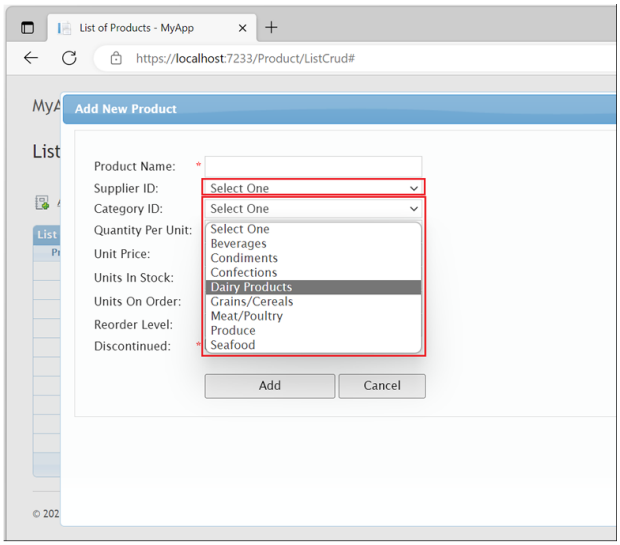
```
public async Task<IActionResult> ListInline()        ——— Action
{
    // return view model
    _productViewModel = await this.GetViewModelAsync("ListInline");
    return View(_productViewModel);        ——— Injecting the ViewModel to the MVC View
}
```

3.2.5.2.3 ListCrud.cshtml

This MVC *View* uses the *ProductViewModel* as its *Model*.

In this MVC *View*, when you *Add a New Record* or *Update an Existing Record*, a dialog pops up.

The *ListCrud.cshtml* uses the MVC *View*'s *Model* (*ProductViewModel*) to fill the *Select* tags for the *SupplierID* and *CategoryID* in the dialog shown below using the MVC *View's Model*, the *SuppliersDropDownListData* and *CategoriesDropDownListData* respectively, these are *Properties* of the *ProductViewModel* as shown in the *ProductViewModelBase* code example in page 30.




```
ListCrud.cshtml
<tr>
    <td class="editor-label"><label asp-for="Product.SupplierID"></label>:</td>
    <td></td>
    <td class="editor-field">
        @if (Model.SupplierDropDownListData != null)
        {
            <select id="supplierID" asp-for="Product.SupplierID" asp-items="@(new SelectList(Model.SupplierDropDownListData, "SupplierID", "CompanyName"))"><option value="">Select One</option></select>
        }
        else
        {
            <select id="supplierID"><option value="">Select One</option></select>
        }
    </td>
    <td class="editor-field"><span id="supplierIDValidation" style="color: red;"></span></td>
</tr>
<tr>
    <td class="editor-label"><label asp-for="Product.CategoryID"></label>:</td>
    <td></td>
    <td class="editor-field">
        @if (Model.CategoryDropDownListData != null)
        {
            <select id="categoryID" asp-for="Product.CategoryID" asp-items="@(new SelectList(Model.CategoryDropDownListData, "CategoryID", "CategoryName"))"><option value="">Select One</option></select>
        }
        else
        {
            <select id="categoryID"><option value="">Select One</option></select>
        }
    </td>
    <td class="editor-field"><span id="categoryIDValidation" style="color: red;"></span></td>
</tr>
```

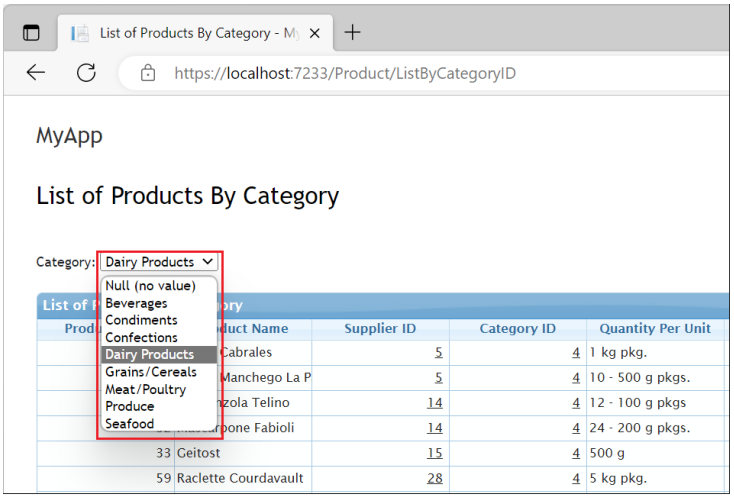
The *ViewModel* used by the *ListInline.cshtml View* is assigned from the respective *Get Action* method found in the *Controller (Base)*, it is then injected to the *ListInline.cshtml View.* See code example below.

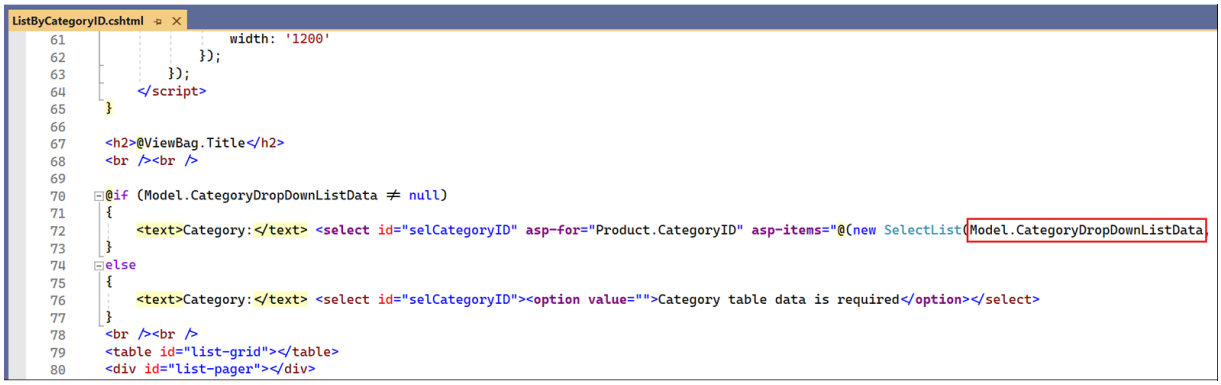
```
public async Task<IActionResult> ListCrud()        ——— Action
{
    // return view model
    _productViewModel = await this.GetViewModelAsync("ListCrud");
    return View(_productViewModel);        ——— Injecting the ViewModel to the MVC View
}
```

3.2.5.2.4 ListBy*ForeignKey*.cshtml

This MVC *View* uses the *ProductViewModel* as its *Model*.

The *ListBy**ForeignKey**.cshtml* uses the MVC *View*'s *Model* (*ProductViewModel*) to fill the *Select* tag for the *ForeignKey* (*CategoryID)* in the dialog shown below using the MVC *View's Model*, the *CategoriesDropDownListData*, this is one of the *Properties* of the *ProductViewModel* as shown in the example code in page 32.





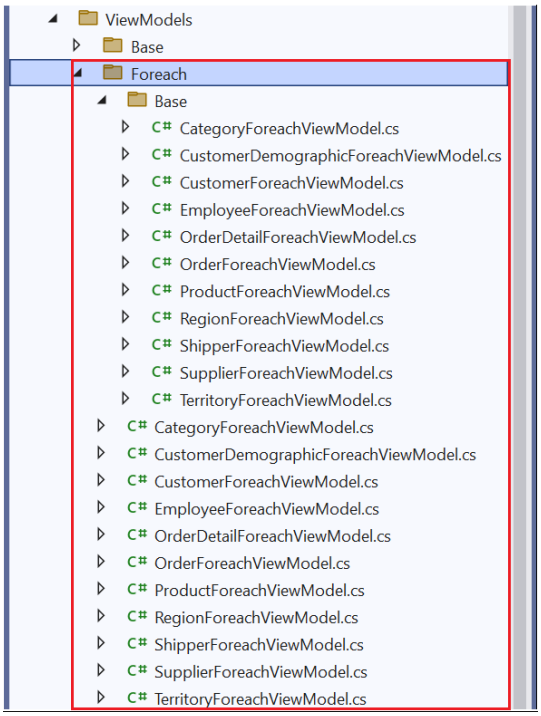
The *ViewModel* used by the *ListBy**ForeignKey**.cshtml View* is assigned from the respective *Get Action* method found in the *Controller (Base)*, it is then injected to the *ListBy**ForeignKey**.cshtml View*. You will notice that code in the *Controller*'s *Action Method* only assigns one *ViewModel Property* compared to the *ListSearch.cshtml*, *ListInline.cshtml*, and *ListCrud.cshtml*, the *CategoriesDropDownListData*. Because it only needs data for one *Select Tag (Categories)* as seen in the image above.

```
public async Task<IActionResult> ListByCategoryID()        ——— Action
{
    // get records
    List<Category> objCategoriesList = null;
    string responseBody = await Functions.HttpClientGetAsync("CategoryApi/SelectCategoryDropDownListData/");

    // make sure responseBody is not empty before deserialization
    if(!String.IsNullOrEmpty(responseBody))
        objCategoriesList = JsonConvert.DeserializeObject<List<Category>>(responseBody);

    // assign values to the view model
    _productViewModel.CategoryDropDownListData = objCategoriesList;

    // return the view model
    return View(_productViewModel);        ——— Injecting the ViewModel to the MVC View
}
```

3.2.5.3 Foreach View Models

These are *Classes* that contains models (properties) used by the *ListForeach.cshtml* MVC *Views*.

The *ForeachViewModels* are located in the *ViewModels/Foreach Folder*.

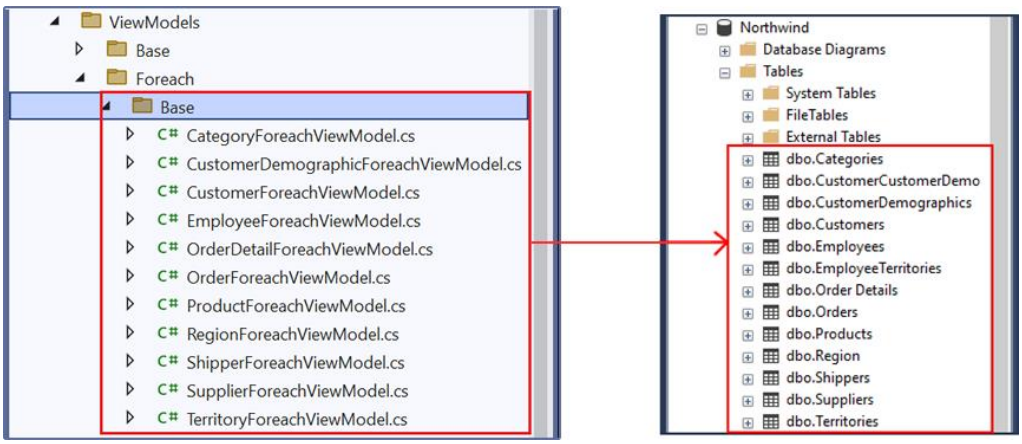


3.2.5.4 Partial Class – Used Like A Base Class

These are the class files generated in the *ViewModels\Foreach\Base* folder.

Do not add any code in these *Class* files.

One *Partial Class* (in the Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Partial ForeacjViewModel in Visual Studio under Base Folder (Left) – Product Database Table Columns in MS SQL Server (Right)

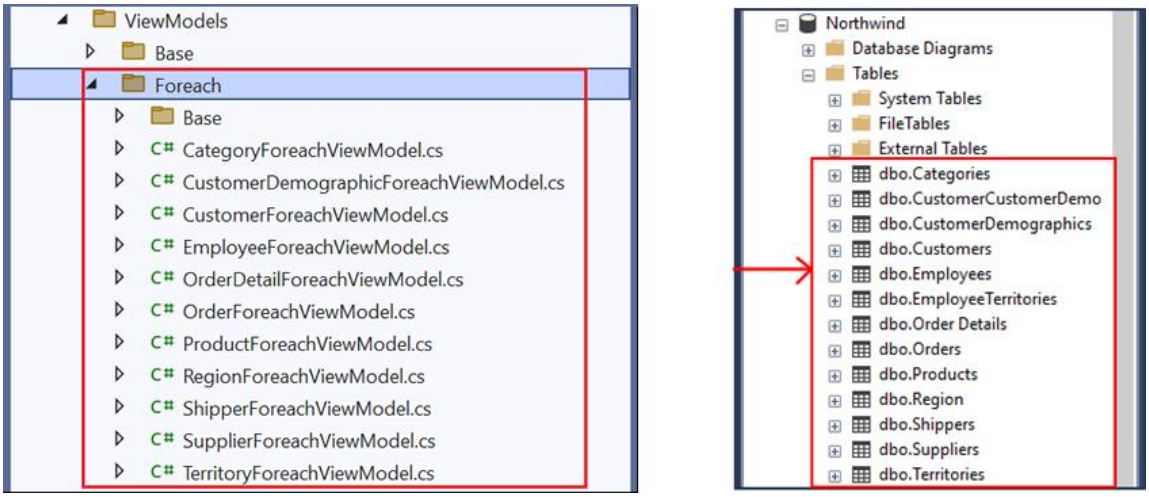
Here's an example of the *ProductForeachViewModel* code.

```
ProductForea...ViewModel.cs
MyAppApi                                   MyAppApi.ViewModels.ProductForeachViewModel
 1  using MyAppApi.Models;
 2  using MyAppApi.BusinessLayer;
 3  using System.Collections.Generic;
 4
 5  namespace MyAppApi.ViewModels
 6  {
 7      /// <summary>
 8      /// Works like the Base class for ProductForeachViewModel
 9      /// ********************************** Do not make changes to this class **********************************
10      /// ** Put your additional code in the ProductForeachViewModel class under the ViewModels/Foreach folder. **
11      /// ******************************************************************************************************
12      /// </summary>
            7 references
13      public partial class ProductForeachViewModel
14      {
                3 references
15          public List<Product> ProductData { get; set; }
                4 references
16          public string[,] ProductFieldNames { get; set; }
                6 references
17          public string FieldToSort { get; set; }
                5 references
18          public string FieldToSortWithOrder { get; set; }
                6 references
19          public string FieldSortOrder { get; set; }
                4 references
20          public int StartPage { get; set; }
                4 references
21          public int EndPage { get; set; }
                3 references
22          public int CurrentPage { get; set; }
                2 references
23          public int NumberOfPagesToShow { get; set; }
                3 references
24          public int TotalPages { get; set; }
                0 references
25          public List<string> UnsortableFields { get; set; }
26      }
27  }
```

3.2.5.5 Foreach View Model – Empty

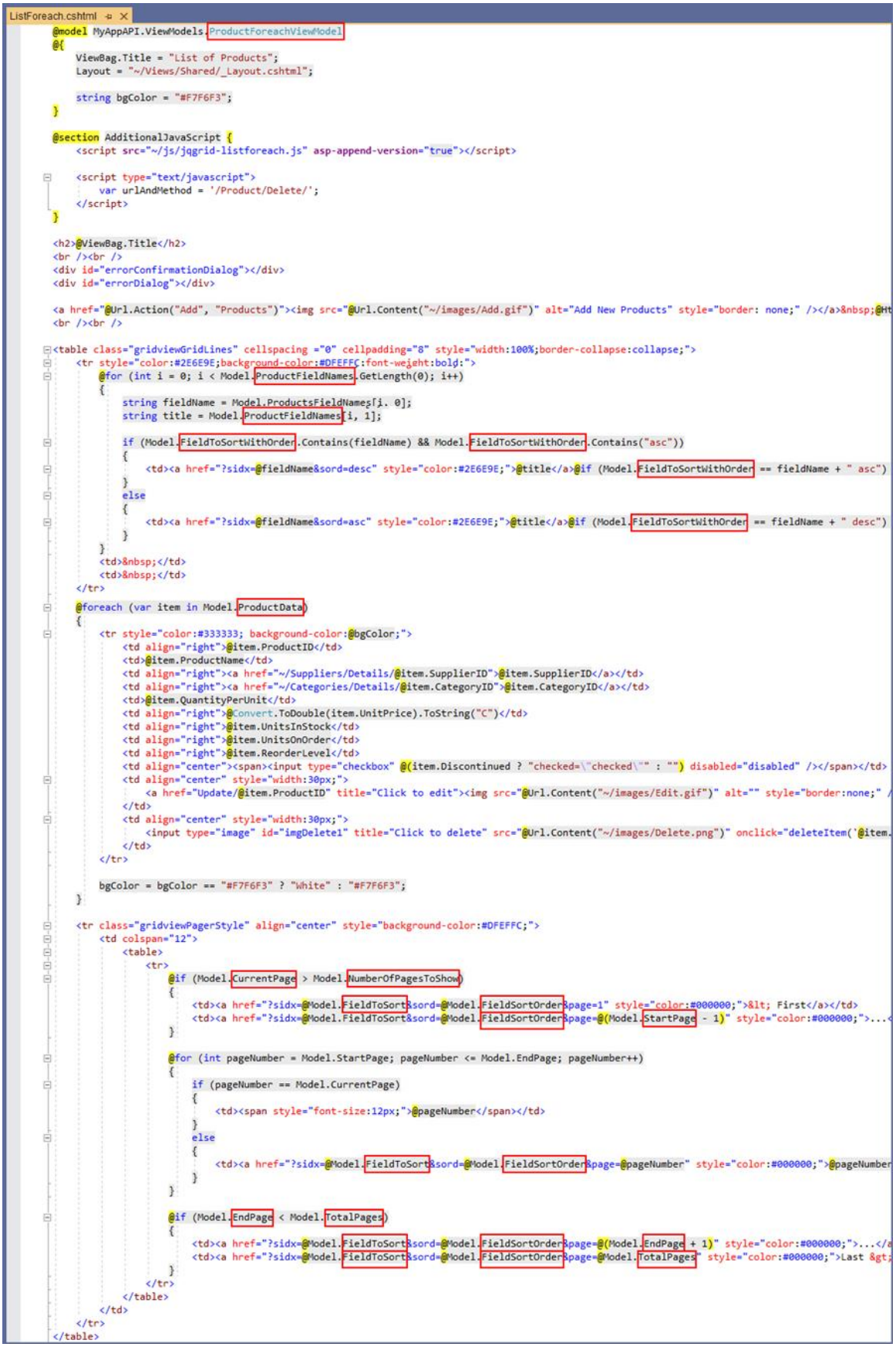
These are the class files generated directly under the *ViewModels\Foreach* folder (not including everything inside the *Base* folder). The naming convention used is: **TableName**ForeachViewModel.*cs*. One *ViewModel class* is generated per *Database Table*.

You can add code in these *Class* files.



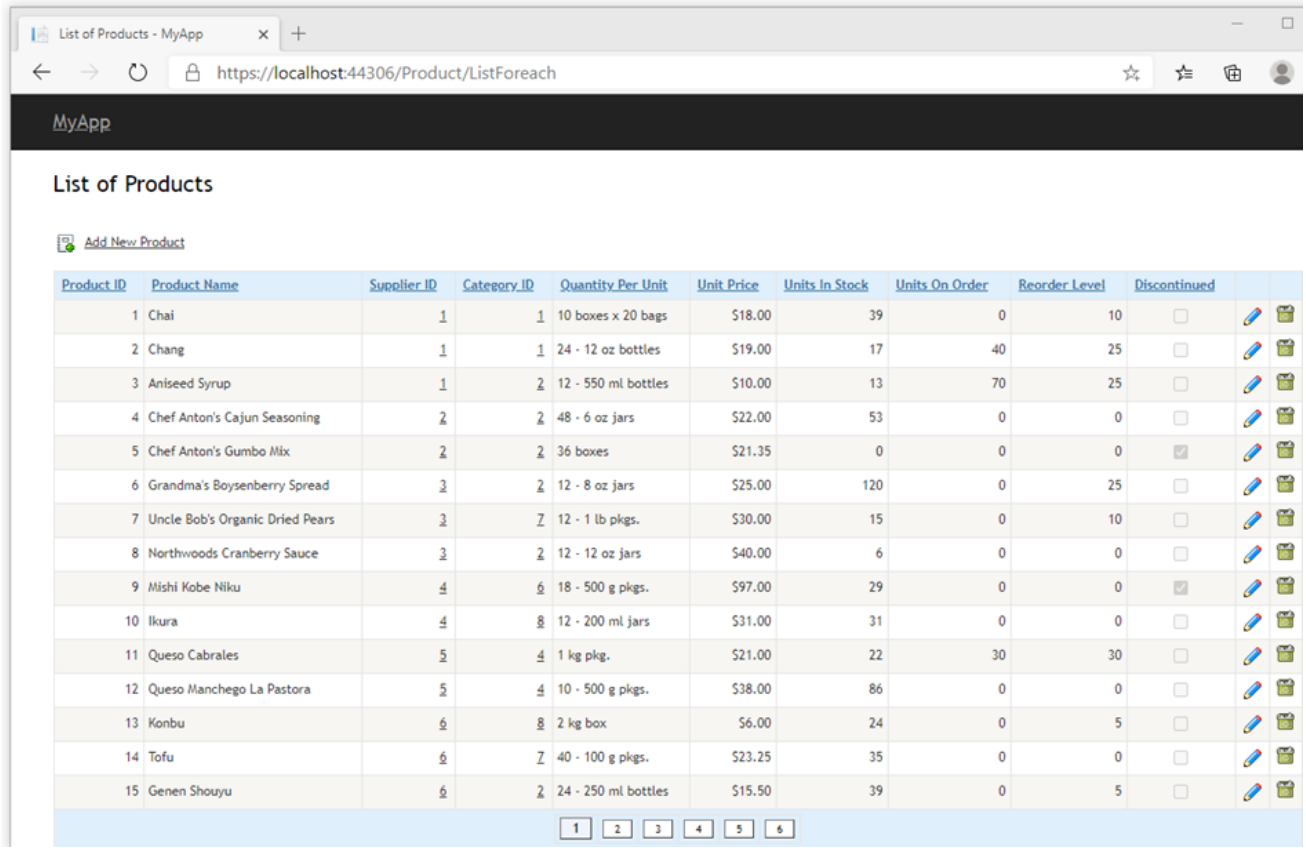
Classes in Visual Studio directly under ViewModels\Foreach Folder (Left) – Database Tables in MS SQL Server (Right)

Here's an example on how the *ListForeach.cshtml* MVC *View* uses the MVC *View*'s *Model* (*ProductForeachViewModel*) to manually build the data grid. The snapshot below shows the *ProductForeachViewModel*'s *Properties* referenced in the *ListForeach.cshtml* MVC *View.*



```
ListForeach.cshtml
@model MyAppAPI.ViewModels.ProductForeachViewModel
@{
    ViewBag.Title = "List of Products";
    Layout = "~/Views/Shared/_Layout.cshtml";

    string bgColor = "#F7F6F3";
}

@section AdditionalJavaScript {
    <script src="~/js/jqgrid-listforeach.js" asp-append-version="true"></script>

    <script type="text/javascript">
        var urlAndMethod = '/Product/Delete/';
    </script>
}

<h2>@ViewBag.Title</h2>
<br /><br />
<div id="errorConfirmationDialog"></div>
<div id="errorDialog"></div>

<a href="@Url.Action("Add", "Products")"><img src="@Url.Content("~/images/Add.gif")" alt="Add New Products" style="border: none;" /></a> @Ht
<br /><br />

<table class="gridviewGridLines" cellspacing ="0" cellpadding="8" style="width:100%;border-collapse:collapse;">
    <tr style="color:#2E6E9E;background-color:#DFEFFC;font-weight:bold;">
        @for (int i = 0; i < Model.ProductFieldNames.GetLength(0); i++)
        {
            string fieldName = Model.ProductsFieldNames[i. 0];
            string title = Model.ProductFieldNames[i, 1];

            if (Model.FieldToSortWithOrder.Contains(fieldName) && Model.FieldToSortWithOrder.Contains("asc"))
            {
                <td><a href="?sidx=@fieldName&sord=desc" style="color:#2E6E9E;">@title</a>@if (Model.FieldToSortWithOrder == fieldName + " asc")
            }
            else
            {
                <td><a href="?sidx=@fieldName&sord=asc" style="color:#2E6E9E;">@title</a>@if (Model.FieldToSortWithOrder == fieldName + " desc")
            }
        }
        <td> </td>
        <td> </td>
    </tr>
    @foreach (var item in Model.ProductData)
    {
        <tr style="color:#333333; background-color:@bgColor;">
            <td align="right">@item.ProductID</td>
            <td>@item.ProductName</td>
            <td align="right"><a href="~/Suppliers/Details/@item.SupplierID">@item.SupplierID</a></td>
            <td align="right"><a href="~/Categories/Details/@item.CategoryID">@item.CategoryID</a></td>
            <td>@item.QuantityPerUnit</td>
            <td align="right">@Convert.ToDouble(item.UnitPrice).ToString("C")</td>
            <td align="right">@item.UnitsInStock</td>
            <td align="right">@item.UnitsOnOrder</td>
            <td align="right">@item.ReorderLevel</td>
            <td align="center"><span><input type="checkbox" @(item.Discontinued ? "checked=\"checked\"" : "") disabled="disabled" /></span></td>
            <td align="center" style="width:30px;">
                <a href="Update/@item.ProductID" title="Click to edit"><img src="@Url.Content("~/images/Edit.gif")" alt="" style="border:none;" /
            </td>
            <td align="center" style="width:30px;">
                <input type="image" id="imgDelete1" title="Click to delete" src="@Url.Content("~/images/Delete.png")" onclick="deleteItem('@item.
            </td>
        </tr>

        bgColor = bgColor == "#F7F6F3" ? "White" : "#F7F6F3";
    }

    <tr class="gridviewPagerStyle" align="center" style="background-color:#DFEFFC;">
        <td colspan="12">
            <table>
                <tr>
                    @if (Model.CurrentPage > Model.NumberOfPagesToShow)
                    {
                        <td><a href="?sidx=@Model.FieldToSort&sord=@Model.FieldSortOrder&page=1" style="color:#000000;">&lt; First</a></td>
                        <td><a href="?sidx=@Model.FieldToSort&sord=@Model.FieldSortOrder&page=@(Model.StartPage - 1)" style="color:#000000;">...<
                    }

                    @for (int pageNumber = Model.StartPage; pageNumber <= Model.EndPage; pageNumber++)
                    {
                        if (pageNumber == Model.CurrentPage)
                        {
                            <td><span style="font-size:12px;">@pageNumber</span></td>
                        }
                        else
                        {
                            <td><a href="?sidx=@Model.FieldToSort&sord=@Model.FieldSortOrder&page=@pageNumber" style="color:#000000;">@pageNumber
                        }
                    }

                    @if (Model.EndPage < Model.TotalPages)
                    {
                        <td><a href="?sidx=@Model.FieldToSort&sord=@Model.FieldSortOrder&page=@(Model.EndPage + 1)" style="color:#000000;">...</a
                        <td><a href="?sidx=@Model.FieldToSort&sord=@Model.FieldSortOrder&page=@Model.TotalPages" style="color:#000000;">Last &gt;
                    }
                </tr>
            </table>
        </td>
    </tr>
</table>
```

Here's the *ListForeach.cshtml* MVC *View* in action.



Product ID	Product Name	Supplier ID	Category ID	Quantity Per Unit	Unit Price	Units In Stock	Units On Order	Reorder Level	Discontinued
1	Chai	1	1	10 boxes x 20 bags	$18.00	39	0	10	☐
2	Chang	1	1	24 - 12 oz bottles	$19.00	17	40	25	☐
3	Aniseed Syrup	1	2	12 - 550 ml bottles	$10.00	13	70	25	☐
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	$22.00	53	0	0	☐
5	Chef Anton's Gumbo Mix	2	2	36 boxes	$21.35	0	0	0	☑
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	$25.00	120	0	25	☐
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	$30.00	15	0	10	☐
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	$40.00	6	0	0	☐
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	$97.00	29	0	0	☑
10	Ikura	4	8	12 - 200 ml jars	$31.00	31	0	0	☐
11	Queso Cabrales	5	4	1 kg pkg.	$21.00	22	30	30	☐
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	$38.00	86	0	0	☐
13	Konbu	6	8	2 kg box	$6.00	24	0	5	☐
14	Tofu	6	7	40 - 100 g pkgs.	$23.25	35	0	0	☐
15	Genen Shouyu	6	2	24 - 250 ml bottles	$15.50	39	0	5	☐

1 2 3 4 5 6

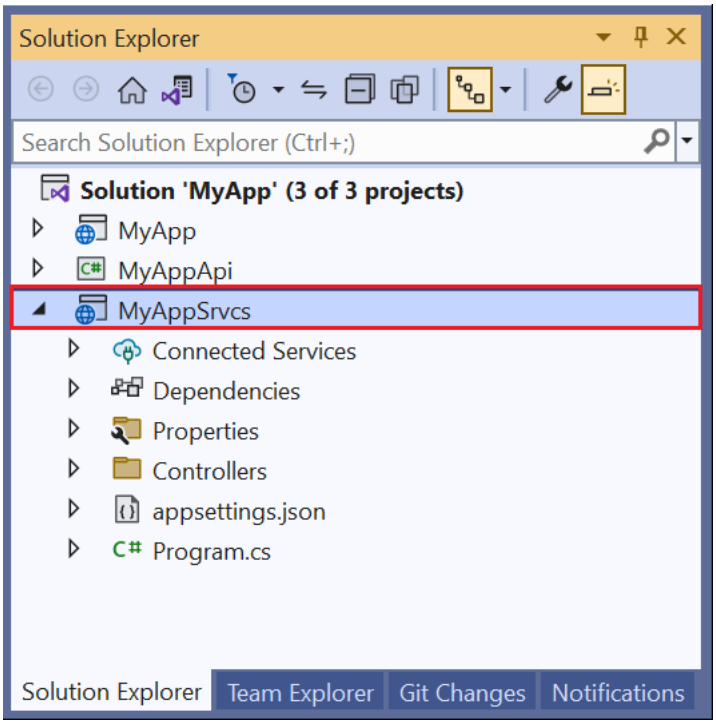
3.3 Web API Project (Web Services)

The generated *Web API Project* is an optional project. This is an ASP.NET MVC API core project. The application's main purpose is to serve as *Web API*s to clients such as the *Web Application Project*. In the *Ntier-Layering* illustrations #2 and #3 in page 5, the *Web Application Project (ASP.NET MVC Core)* and other clients are seen accessing the *Web API*s instead of directly accessing the *Business Layer* (Middle Tier Objects).

These *Web API*s encapsulates the *Middle Layer (Business Layer).* As mentioned in this document, clients can either access the generated *Web API*s or the *Middle Layer (Business Layers)* directly. But, when you generate the optional *Web API Project*, the generated code will directly reference the *Web API*s instead of the the *Middle Layer (Business Layers).*

The main difference between the generated *Web Application Project* and the *Web API Project* is that the *Web API Project* only contains *Controllers (Web APIs)* as the main objects of the project, and it does not have a user interface* (see note).

Note: Although the *Web API Project* does not have a user interface, the generated Web API methods can be tested in the Swagger Index page. The Swagger Index page can be used to test the generated Web API methods, you may also supply it to (software) clients so they can have an idea on how your Web API methods work (can be accessed). See page 44.



3.3.1 LaunchSettings.json, appsettings.json, Program.cs

These are similar objects as the ones seen in the *Web Application Project*. Please see the *Web Application Project* for more information about these objects.

3.3.2 Controllers

The *Controllers* are the *Web APIs*.

This folder is generally needed by ASP.NET Core MVC by default. It houses *Controller's Methods* that can be used as *Web APIs*.

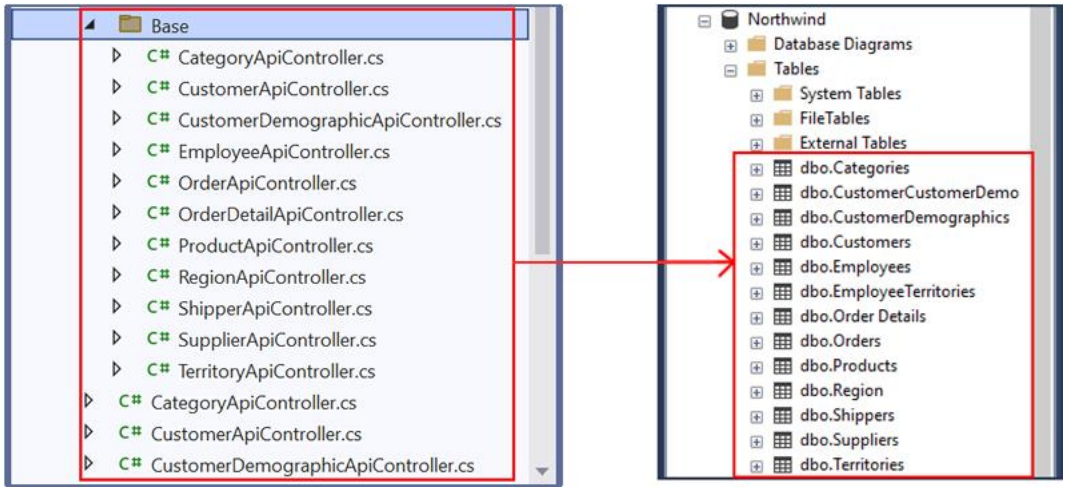
Note: The *Controllers* in the *Web API Project* and the *Web Application Project* are similar in nature. Please read about the *Controllers* under the *Web Application Project* in page 10 for more information.

3.3.2.1 The Controller - Used Like A Base Class

Note: Not a base class. The code needed by the Controller are generated in these partial classes. These are the partial class files generated in the *Controller\Base* folder. The naming convention used is: ***TableName****APIController.cs*.

Do not add any code in these *Partial Class* files.

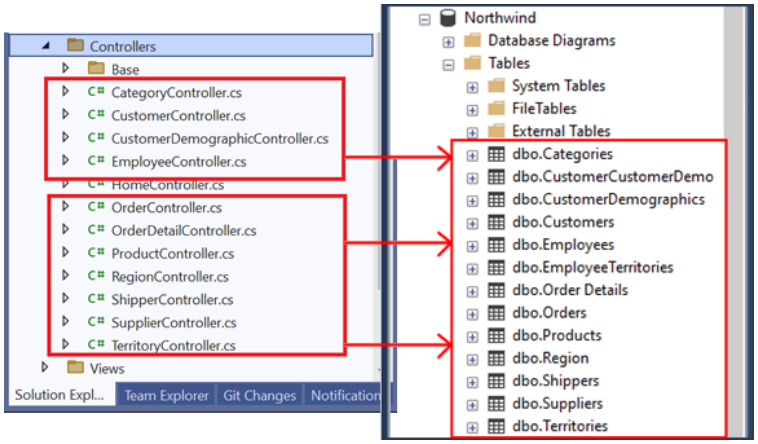
One *Partial Class* (in the Controllers\Base folder) is generated per *Database Table*. The example below shows that you generated code for *All Tables* for the *Northwind* database.



Web API Controllers (Partial Classes) in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

3.3.2.2 The Controller - Empty

These are the *Partial Classes* generated directly under the *Controllers* folder (not including everything inside the *Base* folder). The naming convention used is: ***TableName****APIController.cs*. ASP.NET Core MVC recognizes this as a *Controller* by default because of the suffix "*Controller*" in the name. One *Controller* is generated per *Database Table*. **You can add code in these *Partial Class* files.**



Web API Controllers in Visual Studio (Left) – Database Tables in MS SQL Server (Right)

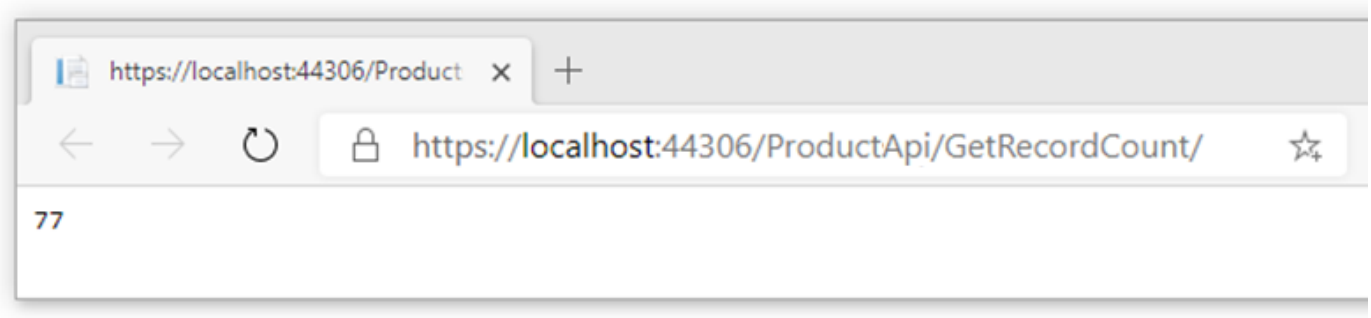
3.3.2.3 Accessing Web API Controllers (Methods)

Just like mentioned above, the *Web API Project* does not have a user interface just like the *Web Application Project*'s MVC *Views*. We need to access *Web API Controllers* via code using *HttpClient* calls.

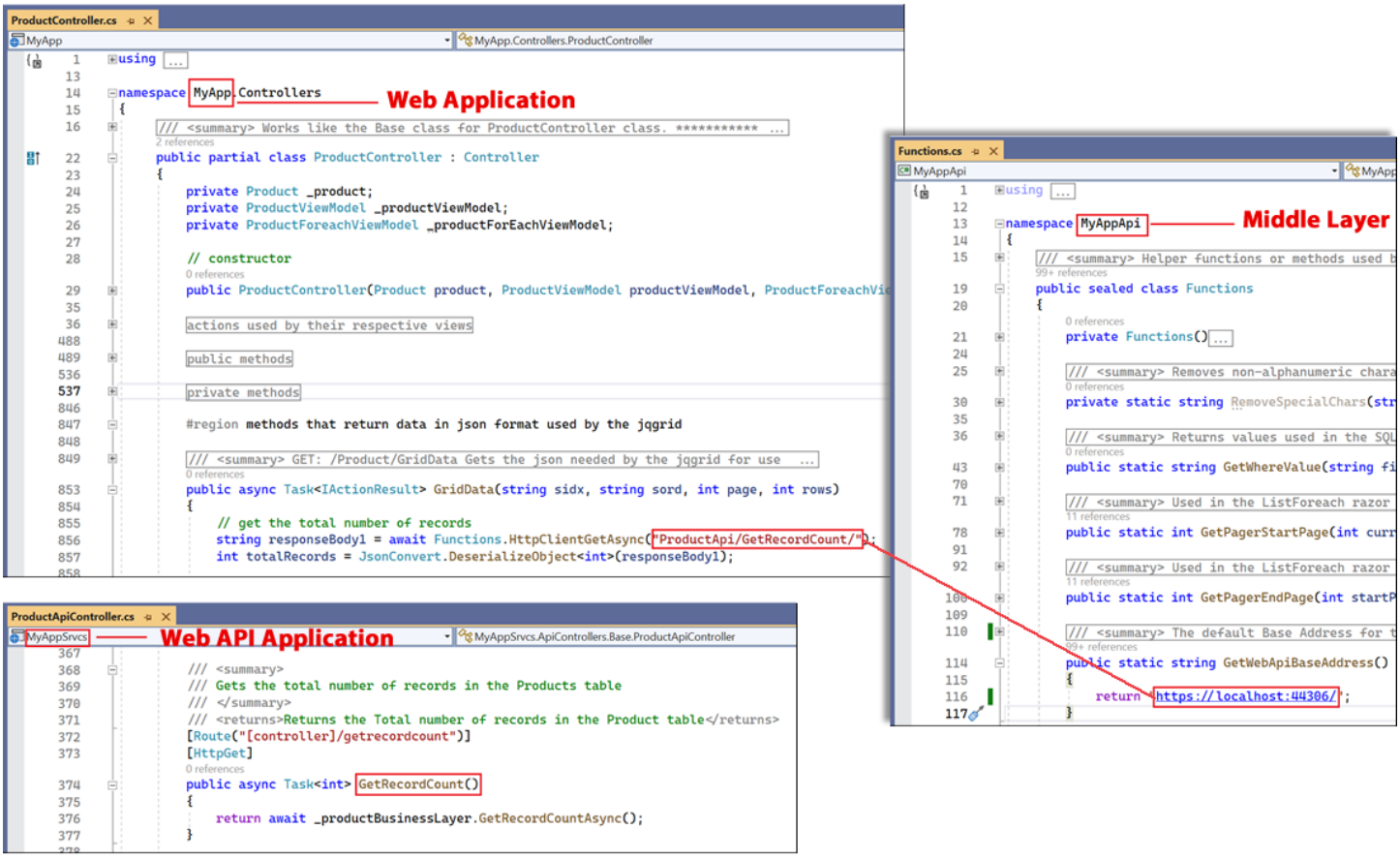
For example, we need to make an *HttpClient Get Request* call from the *Web Application Project*'s *Controller (ProductControllerBase)* to access the *GetRecordCount() Method* in the *Web API Controller (Product****API****ControllerBase)*.

To make an *HttpClient Get Request* call, we use the:

1. *Web API Project*'s *Web Address (URL,* ***https://localhost:44306/****)*
2. *Controller*'s name (***ProductAPI***, minus the word *"Controller"*),
3. And the *Method* name (***GetRecordCount()***)

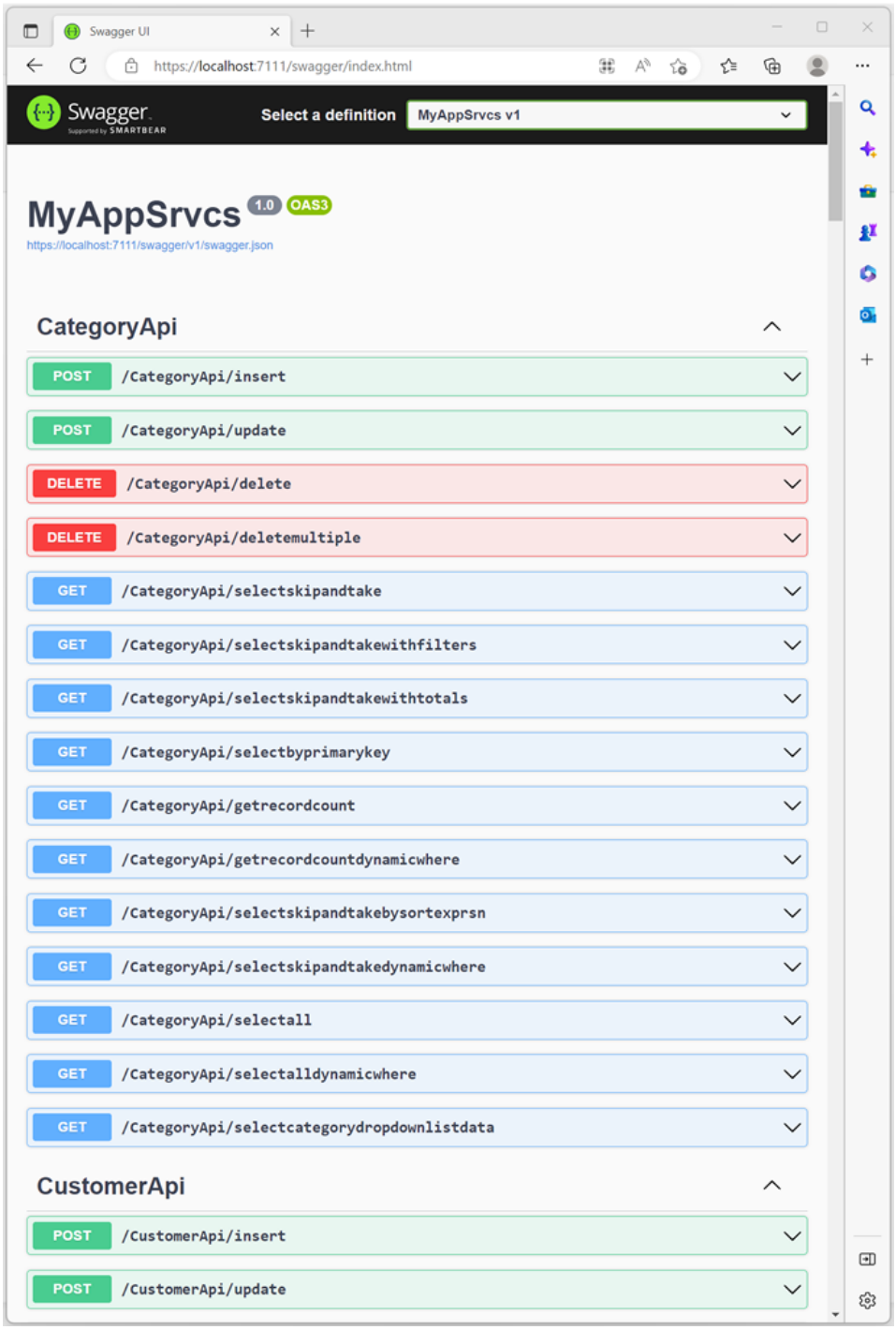


The example below shows that *GetRecordCount() (Web API Project)* was called from the *GridData Method (Web Application Project)* using the *Web API*'s base URL "*https://localhost:44306/" (Functions Class in the Middle Layer Project)* plus the *"ProductAPI/GetRecordCount".*



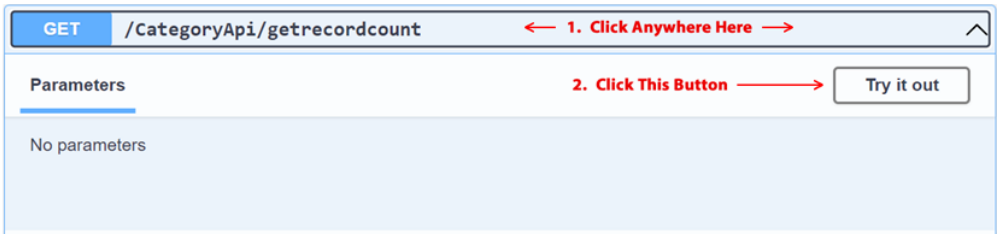
3.3.3 Swagger

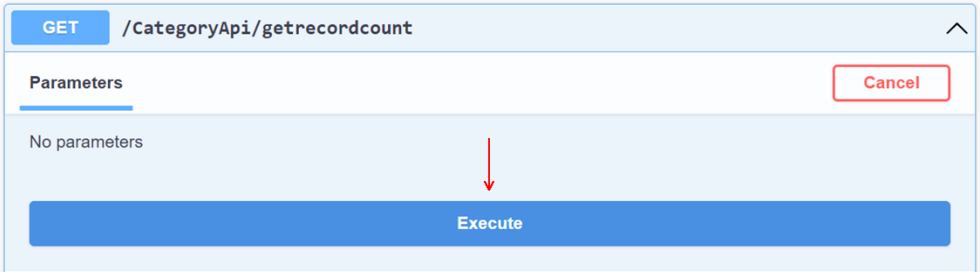
When you run both the *Web Application Project* and the *Web API Project* at the same time in Visual Studio, 2 browser instances launches on the screen, one for each project. The Web App's home page shows the list of objects that was generated while the Web API's home page launches a *Swagger User Interface* showing a list of web API endpoints.



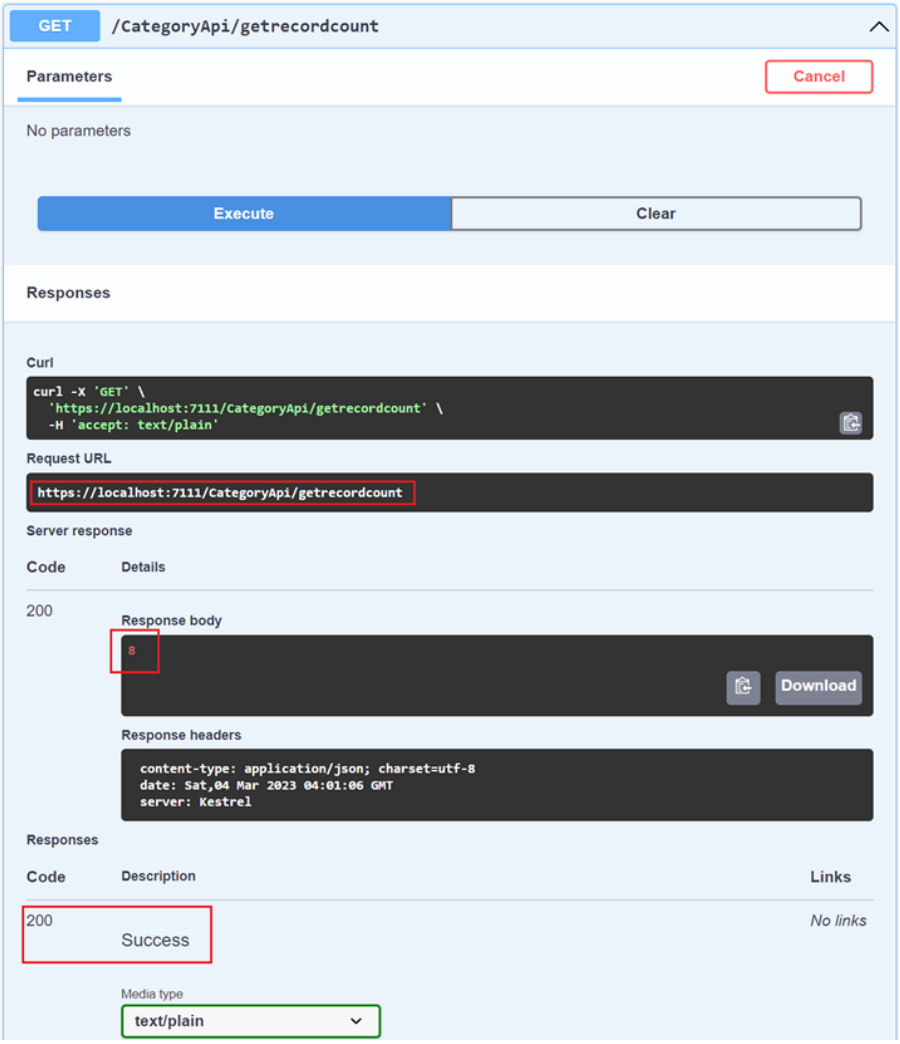
3.3.3.1 Testing An Endpoint (Web API Method)

Click an Endpoint in the list, for example the */CategoryApi/getrecordcount*. And then click the *Try it out* button. And then click the *Execute* button.



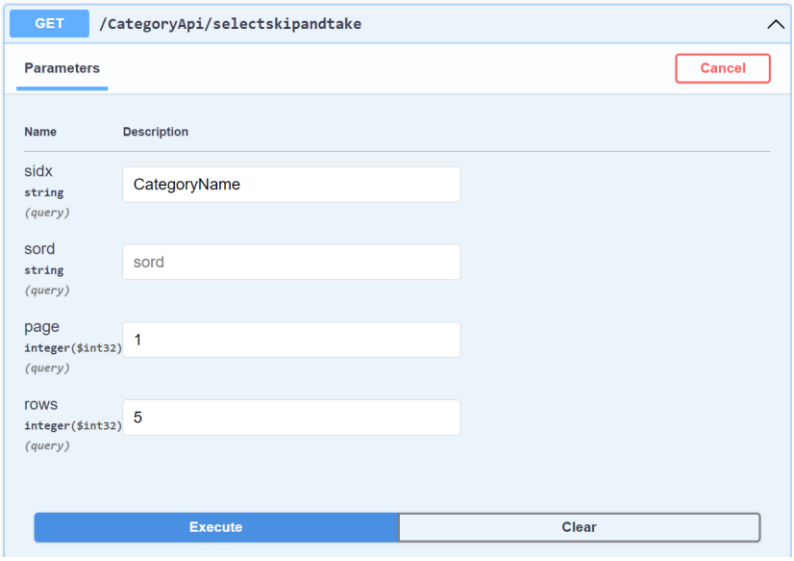


The result will show the *Request URL*, you can use this in your code to call this endpoint. Also shows the *Response Body* **8** (*getrecordcount* endpoint simply returns a number), which is the *total number of records* in the *Categories* database table. And the *Response Code*, *200* means *Success* (the web API call was successful).

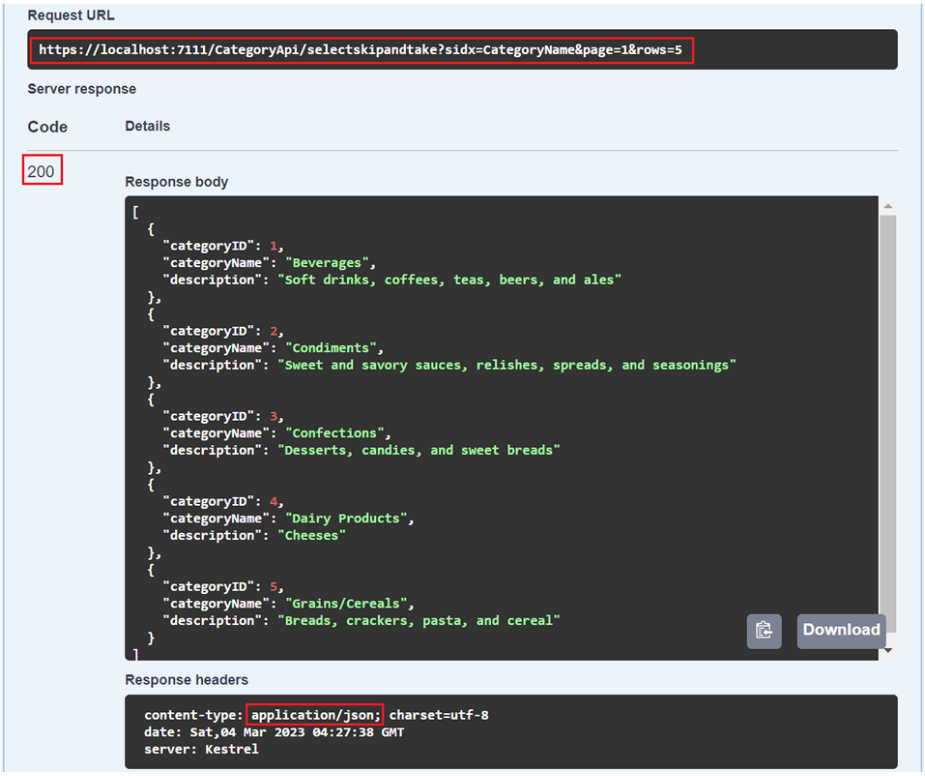


Now try the */CategoryApi/selectskipandtake* endpoint. Click the *Try it out* button. This endpoint requires 4 parameters:

1. **sidx** – Field to sort.
2. **sord** – Sort order. ***asc*** or ***blank*** (nothing) for ascending order, and ***desc*** for descending order.
3. **rows** – Number of rows you want returned.
4. **page** – based on the number of rows you're requesting, there may be several pages to return, this is the *page number* you want returned. For example, if there are 37 records in the *Categories* database table, and the *rows* you requested is 5, then there will be 7 pages, you can request any number from 1 to 7.



The *Response* shows a *Code 200* (success), *5 records* returned (in descending order by *CategoryName*) in *json format*.



* CRUD means Create, Retrieve, Update, and Delete. These are database operations.

You can read end-to-end tutorials on more subjects on using AspCoreGen 6.0 MVC Professional Plus that came with your purchase. These tutorials are available to customers and are included in a link on your invoice when you purchase AspCoreGen 6.0 MVC Professional.

Note: Some features shown here are not available in the Express Edition.

End of tutorial.