Using MVVM in MVC Applications – Part 1

This blog post is the first in a series of four posts to discuss how to use a Model-View-View-Model (MVVM) approach in an MVC application. The MVVM approach has long been used in WPF applications, but has not been prevalent in MVC applications. Using a View Model class in MVC makes good sense as this blog post illustrates. You are going to be guided step-by-step building an MVC application using the Entity Framework and a View Model class to create a full CRUD web page.

Model-View-View-Model Approach

The reasons why programmers are adopting MVVM design pattern is the same reasons why programmers adopted Object Oriented Programming (OOP) over 30 years ago: reusability, maintainability and testability. Wrapping the logic of your application into classes allows you to reuse those classes in many different applications. Maintaining logic in classes allows you to fix any bugs in just one place and any other classes using that class automatically get the fix. When you don't use global variables in an application, testing your application becomes much simpler. Wrapping all variables and logic that operates upon those variables into one class allows you to create a set of tests to check each property and method in the class quickly and easily.

The thing to remember with MVVM is all you are doing is moving more of the logic out of the code behind of a user interface, or an MVC controller, and into a class that contains properties and methods you bind to the user interface. To use MVVM you must be using classes and not just writing all your code in a MVC controller. The whole key to MVVM or MVC is the use of classes with properties and methods that mimic the behavior you want in the UI. This means setting properties that are bound to UI controls and calling methods when you want to perform some action that you would normally write in a controller method.

Why Use MVVM in MVC

There are many reasons for using MVVM in an MVC application. Below are some of them.

1. Eliminate code in controller
    a. Only two controller methods are needed
2. Simplify the controller logic
3. Can just unit test the view model and not the controller
4. View model classes can be reused in different types of projects (WPF, Web Forms, etc.)

Our Goal

This series of blog posts have a few goals for you to accomplish.

1. Create a product table
2. Create a set of MVC pages to list, search, add, edit, delete and validate product data
3. Create a view model class to handle all these functions
4. Write just two small methods in an MVC controller

Create a Product Table

For this sample, create a table called Product in a SQL Server database. In the following code snippet, you see the full definition of the Product table.

```
CREATE TABLE Product (
  ProductId int NOT NULL
                IDENTITY(1,1)
                PRIMARY KEY NONCLUSTERED,
  ProductName varchar(150) NOT NULL,
  IntroductionDate datetime NOT NULL,
  Url varchar(255) NOT NULL,
  Price money NOT NULL
)
```

Add Data to the Table

Once you create the table, add some data to each field. Create enough rows that you can see several rows on your final HTML page. Below is the data that I used to create the list of product data for this sample application.

ProductId	ProductName	IntroductionDate	Url	Price
1	Extending Bootstrap with CSS, JavaScript and jQuery	6/11/2015	http://bit.ly/1SNzc0i	$29.00
2	Build your own Bootstrap Business Application Template in MVC	1/29/2015	http://bit.ly/1I8ZqZg	$29.00
3	Building Mobile Web Sites Using Web Forms, Bootstrap, and HTML5	8/28/2014	http://bit.ly/1J2dcrj	$29.00
4	How to Start and Run A Consulting Business	9/12/2013	http://bit.ly/1L8kOwd	$29.00
5	The Many Approaches to XML Processing in .NET Applications	7/22/2013	http://bit.ly/1DBfUqd	$29.00
6	WPF for the Business Programmer	6/12/2009	http://bit.ly/1UF858z	$29.00
7	WPF for the Visual Basic Programmer - Part 1	12/16/2013	http://bit.ly/1uFxS7C	$29.00
8	WPF for the Visual Basic Programmer - Part 2	2/18/2014	http://bit.ly/1MjQ9NG	$29.00

Figure 1: Product data

Here is the script to add the data to the Product table.

```
SET IDENTITY_INSERT Product ON
GO
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (1, 'Extending Bootstrap with CSS, JavaScript and
jQuery', '2015-06-11', 'http://bit.ly/1SNzc0i', 29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (2, 'Build your own Bootstrap Business Application
Template in MVC', '2015-01-29', 'http://bit.ly/1I8ZqZg',
29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (3, 'Building Mobile Web Sites Using Web Forms,
Bootstrap, and HTML5', '2014-08-28', 'http://bit.ly/1J2dcrj',
29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (4, 'How to Start and Run A Consulting Business',
'2013-09-12', 'http://bit.ly/1L8kOwd', 29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (5, 'The Many Approaches to XML Processing in .NET
Applications', '2013-07-22', 'http://bit.ly/1DBfUqd',
29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (6, 'WPF for the Business Programmer', '2009-06-12',
'http://bit.ly/1UF858z', 29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (7, 'WPF for the Visual Basic Programmer - Part 1',
'2013-12-16', 'http://bit.ly/1uFxS7C', 29.0000);
INSERT Product (ProductId, ProductName, IntroductionDate, Url,
Price)
VALUES (8, 'WPF for the Visual Basic Programmer - Part 2',
'2014-02-18', 'http://bit.ly/1MjQ9NG', 29.0000);
SET IDENTITY_INSERT Product OFF
GO
```

Create All Projects

As there are three pieces to a MVVM application, the Model, the View and the View Model, there are three Visual Studio projects you need to build within a single solution.

- Data Layer Class Library (Model)

- MVC Web Project (View)
- View Model Class Library (View Model)

Let's start building those now.

Create the Web Project

Open Visual Studio and click on the File | New Project to display the New Project window. Select Web from the Templates under Visual C#. From the list in the middle of the window select ASP.NET Web Application (.NET Framework). Set the name of this project to PTC as shown in Figure 2.

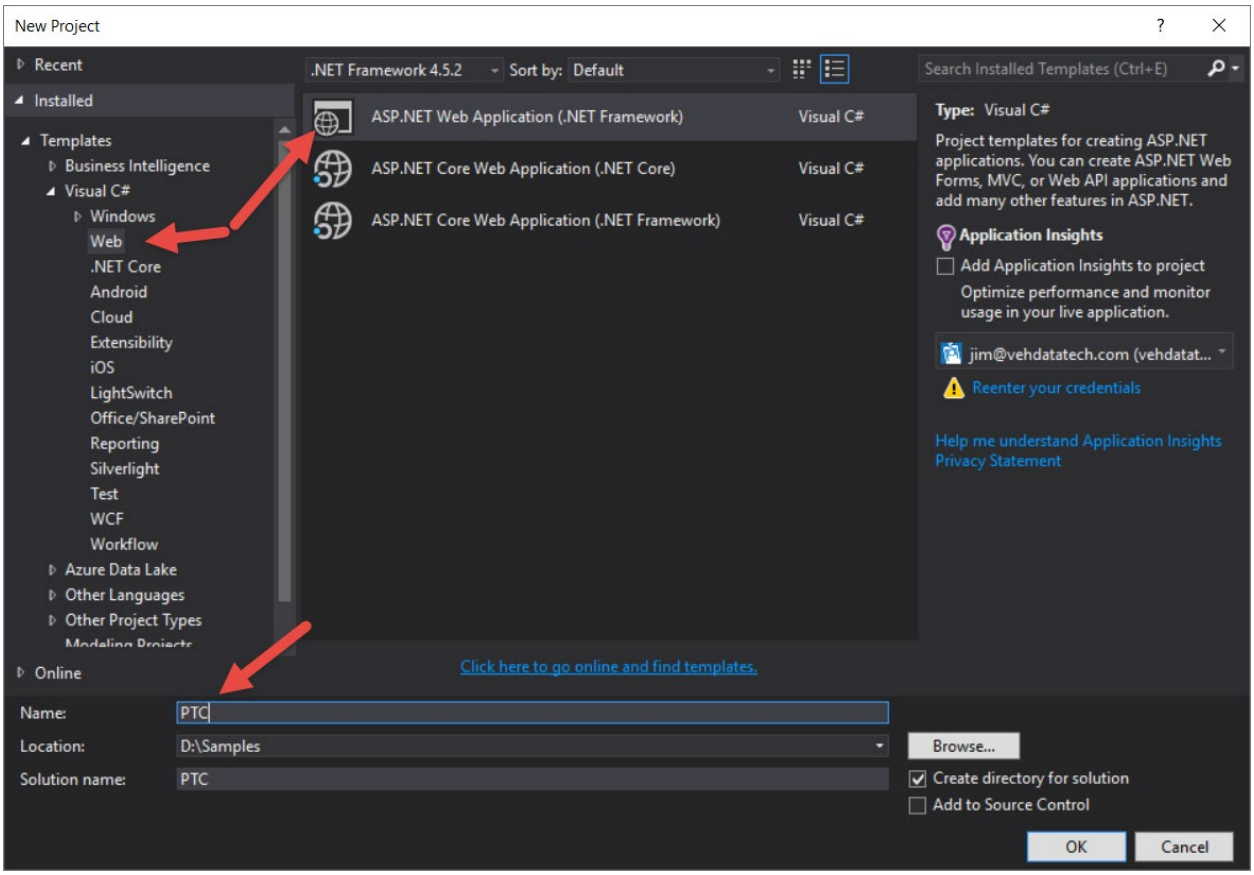

Figure 2: Create a new MVC Application named PTC.

Click the OK button. When prompted, choose the MVC template as shown in Figure 3 and click the OK button.

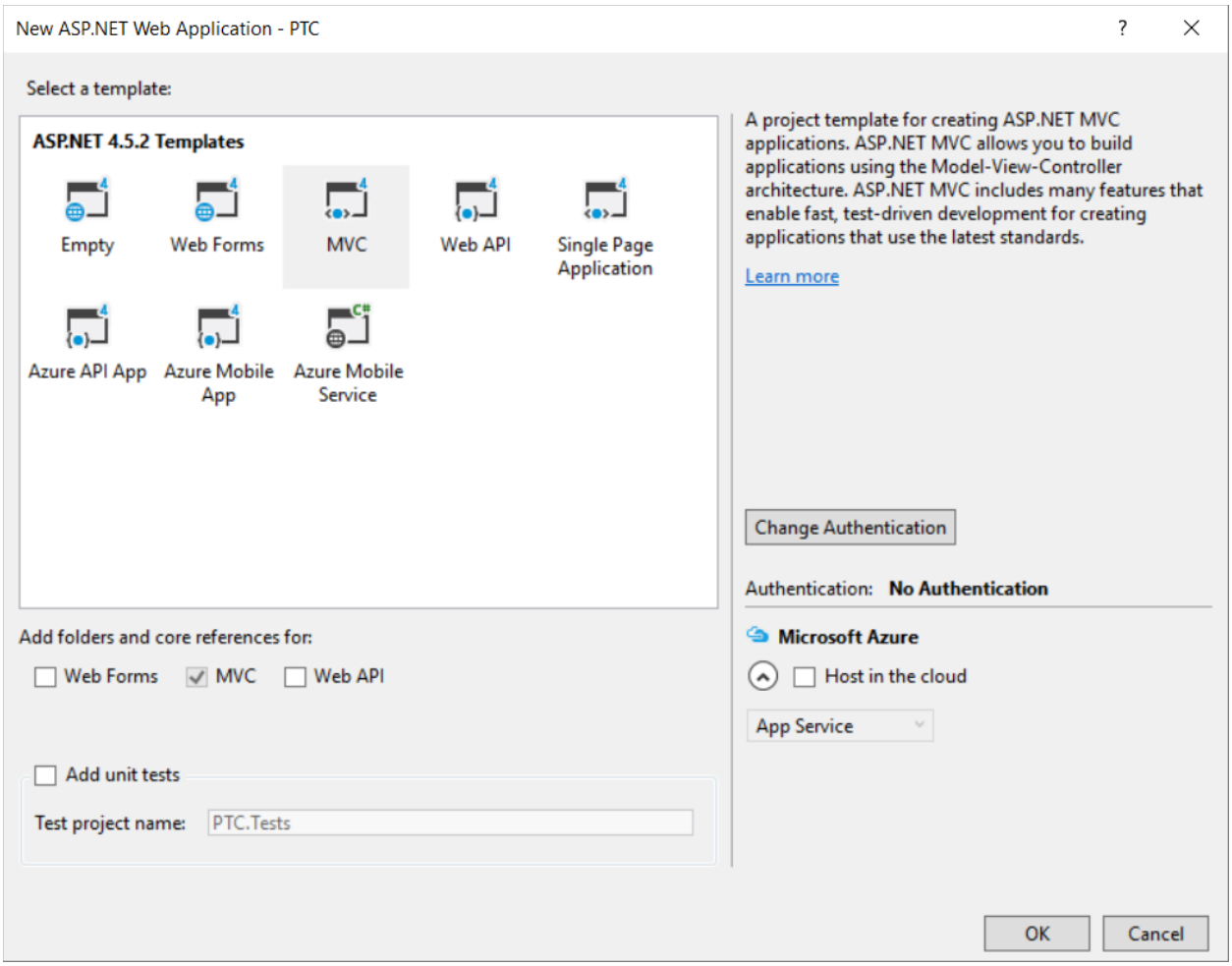

Figure 3: Choose the MVC template

Add Entity Framework

You are going to be using the Entity Framework in all threes projects. While you are still building this MVC project, let's go ahead and add the necessary DLLs and configuration file entries. Right mouse click on the project and select **Manage NuGet Packages…** Click on the Browse tab and search for Entity Framework. Click the Install button as shown in Figure 4.

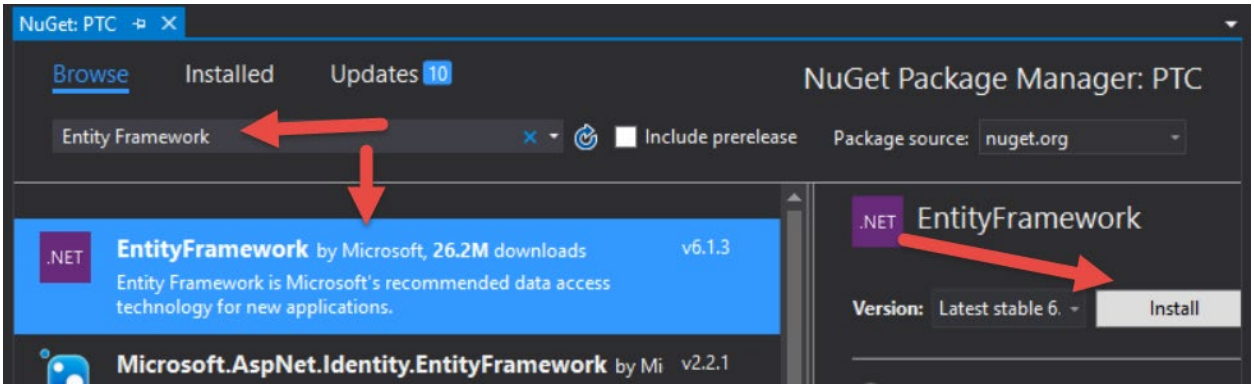

Figure 4: Install the Entity Framework using NuGet

Create the Data Layer Project

Let's now build a project to just hold all data access classes and any other Entity classes we need to create. Right mouse click on the solution and choose **Add** | **New Project**. Select Windows from the Templates on the left tree view. Select Class Library from the middle part of the window. Set the name to PTC.DataLayer as shown in Figure 5.

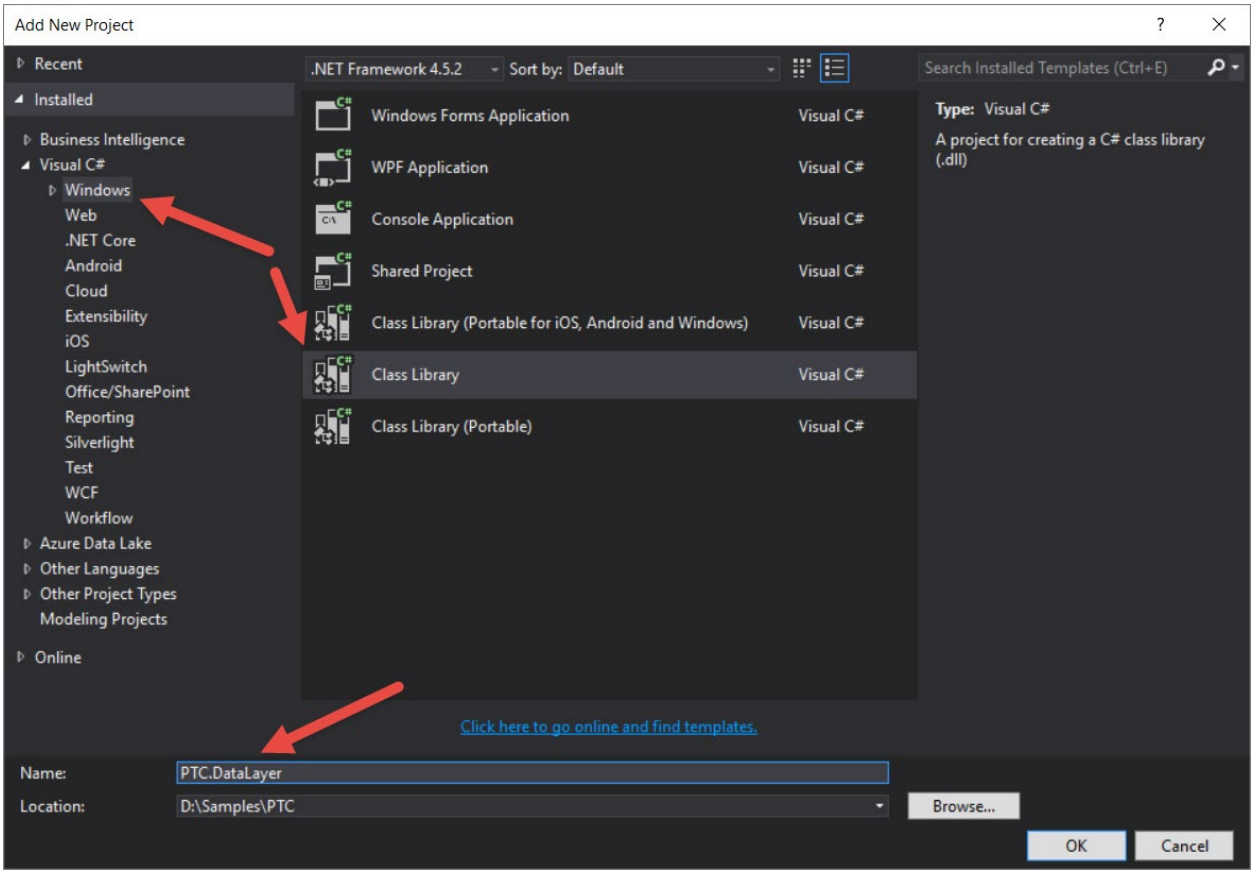

Figure 5: Create a Class Library project for your Data Layer

Click the OK button to create the new project. Delete the file Class1.cs as you won't need this. Right mouse click on this project and select **Add | New Item…** From the New Item window select **Data | ADO.NET Entity Data Model**. Set the name to PTCData as shown in Figure 6. Click the Add button.

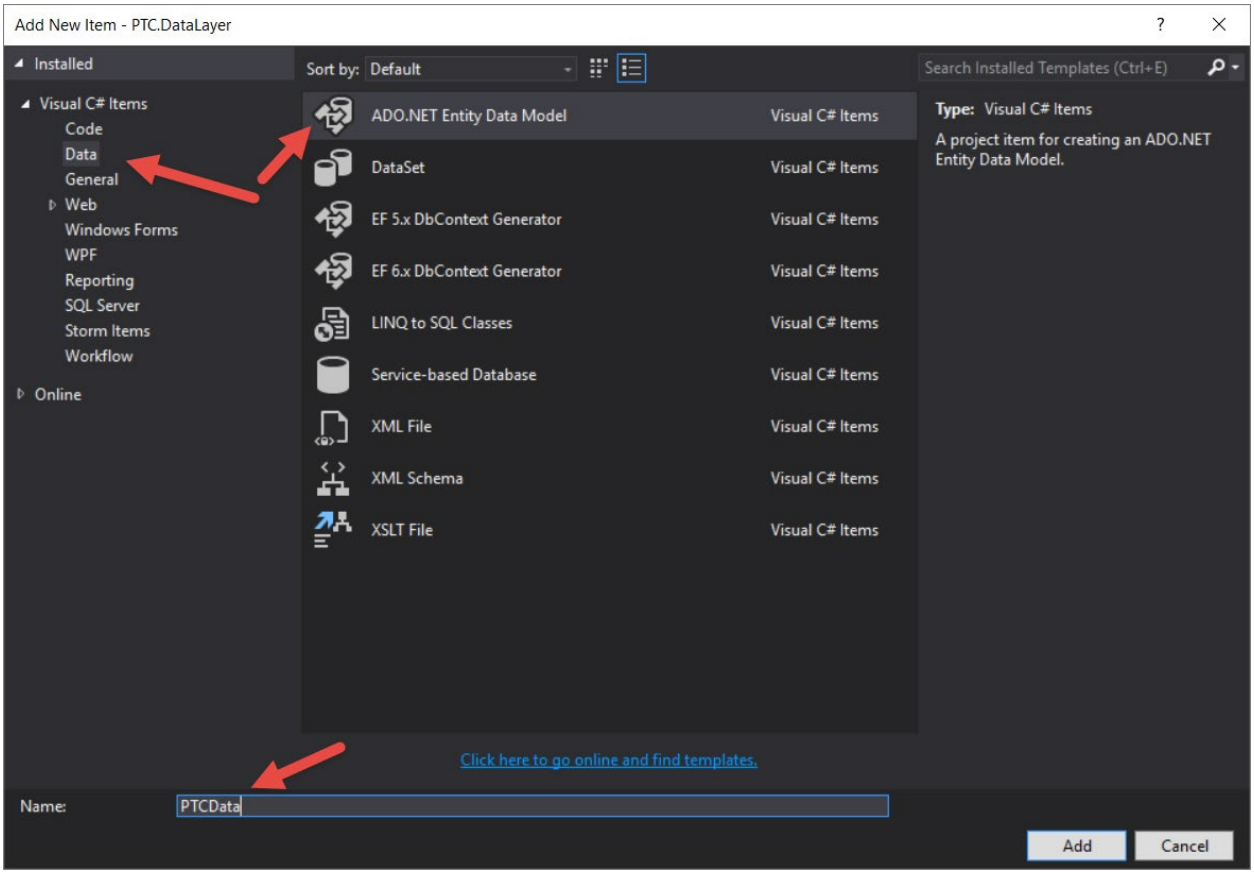

Figure 6: Add an ADO.NET Entity Data Model to the Data Layer project

On the Entity Data Model Wizard screen that is now displayed (Figure 7), select the **Code First from database** option.

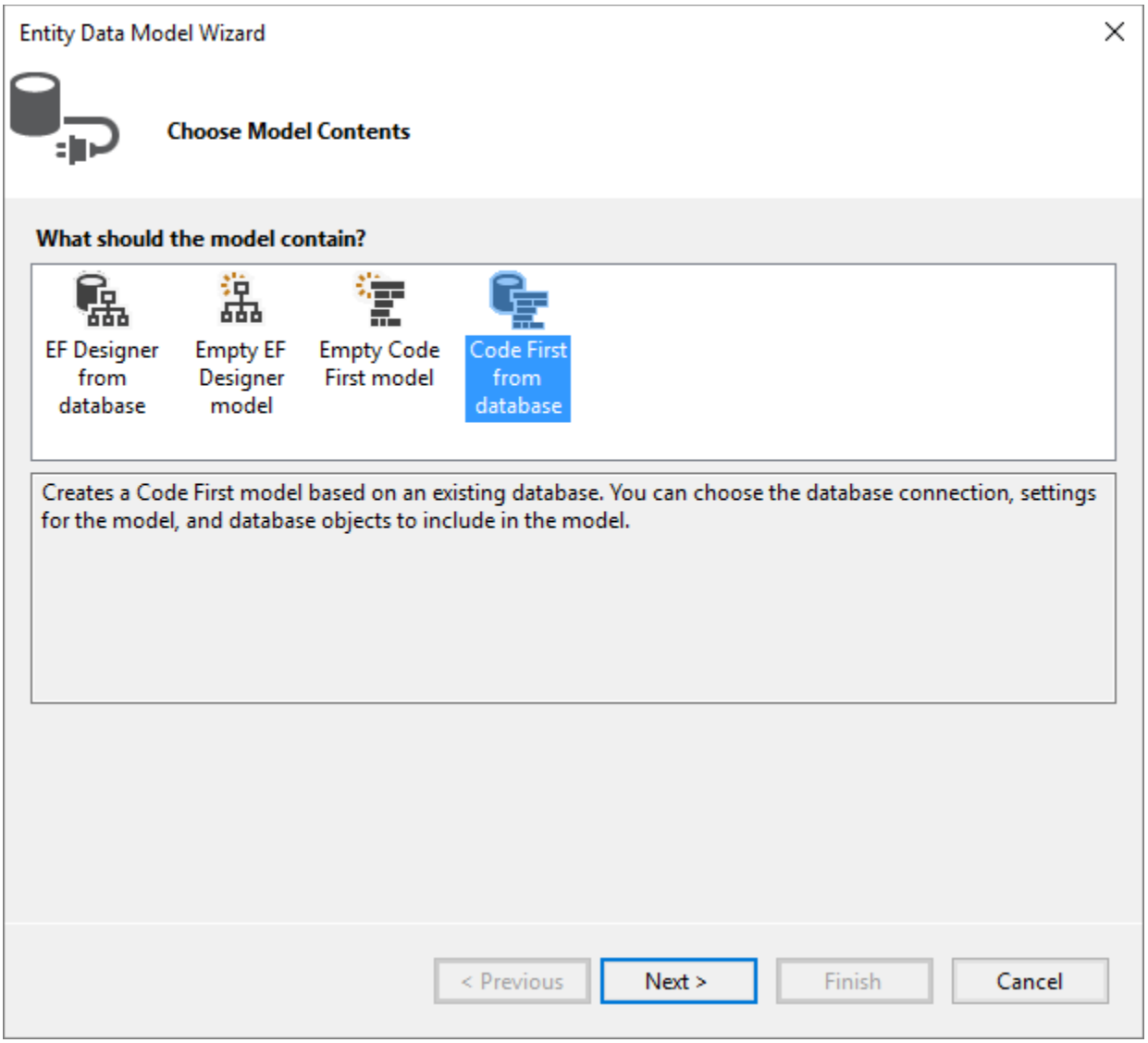

Figure 7: Choose the Code First from database option.

Click the Next button to advance to the next screen. On the next page of this wizard (Figure 8) add a connection string where you created the Product table. Leave everything else as it is on this page and click the Next button.

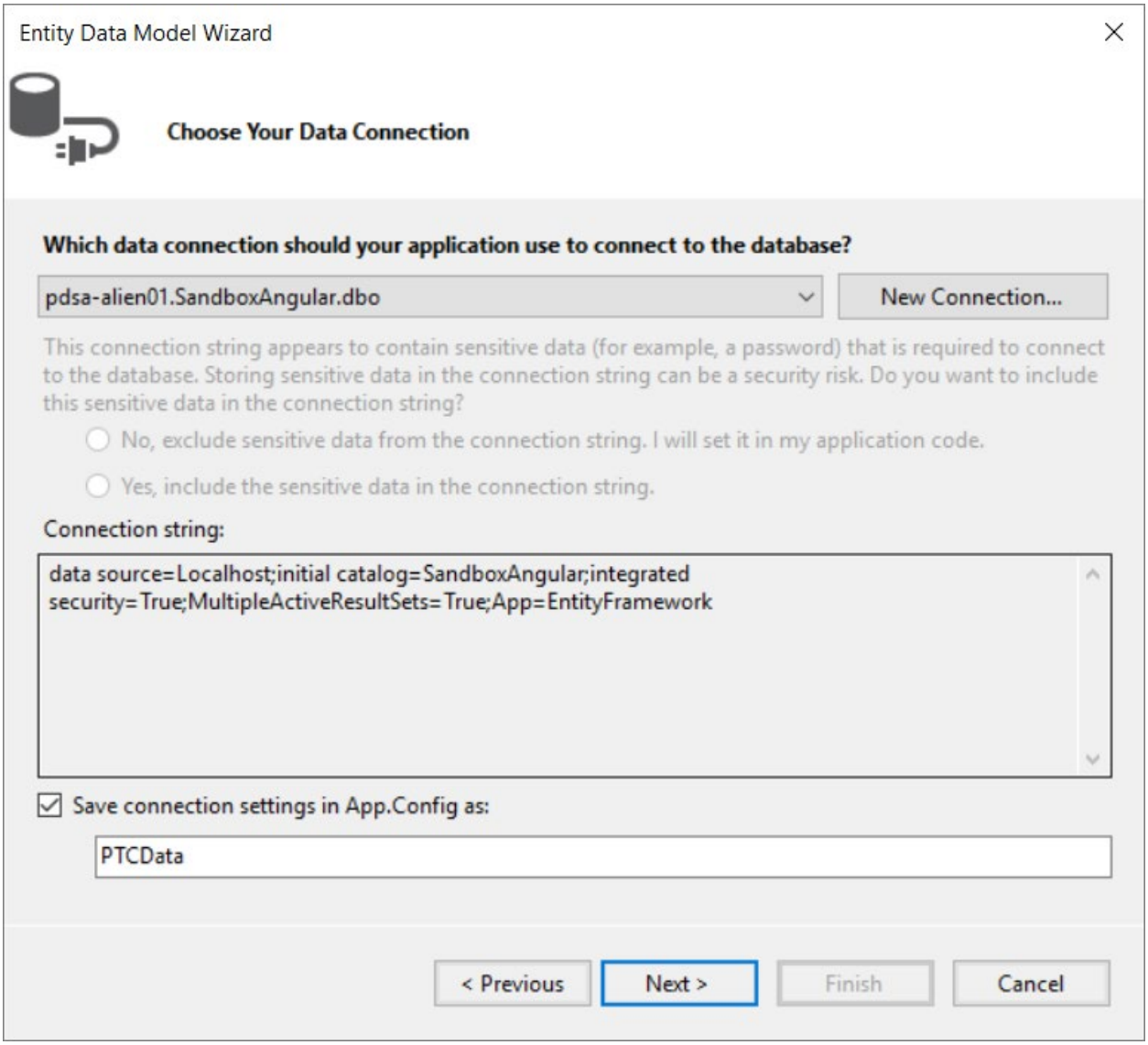

Figure 8: Add a connection string to point to your database.

You are now going to choose the Product table you created earlier in this blog post. Drill down into your collection of tables and check the Product table as shown in Figure 9.

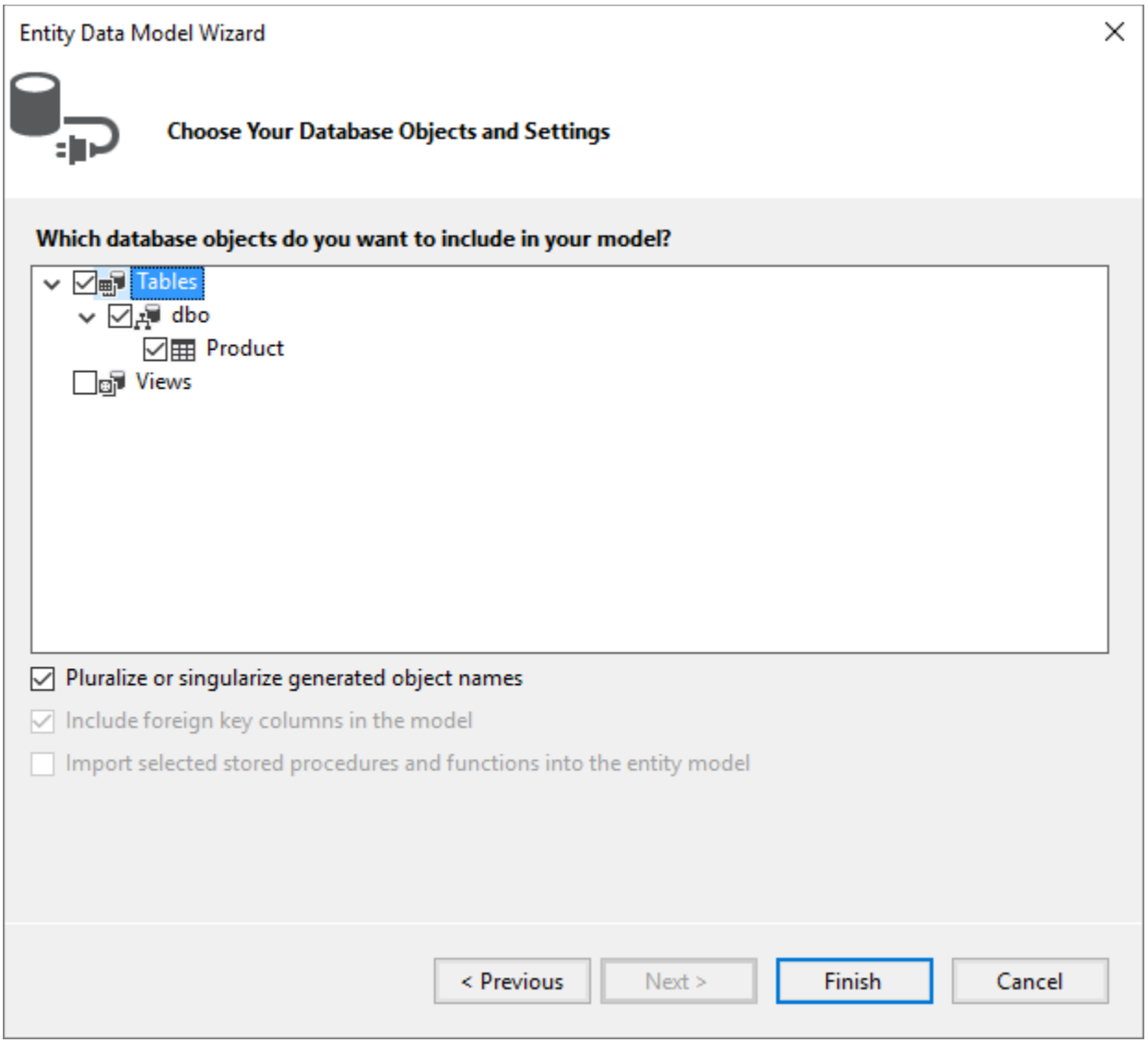

Figure 9: Select the Product table

Click the Finish button and Visual Studio will generate new classes that allow you to create, read, update and delete data in the Product table.

Create View Model Project

The last project you need to add is one for your View Model classes. Right mouse click on the solution and choose **Add** | **New Project**. Select Windows from the Templates on the left tree view. Select Class Library from the middle part of the window. Set the name to **PTC.ViewModel** as shown in Figure 10.

Click the OK button. Rename **Class1.cs** to **ProductViewModel.cs**.

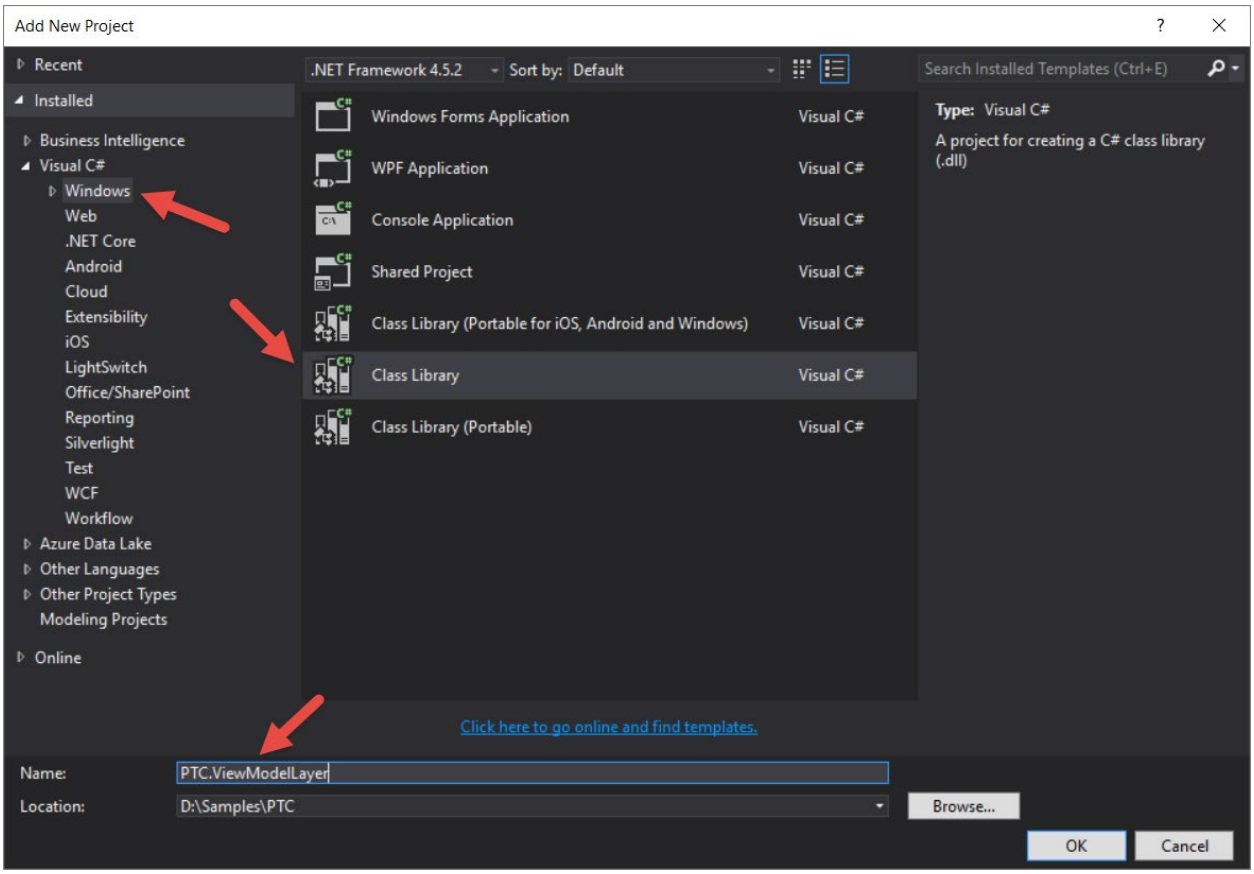

Figure 10: Add a Class Library project for your View Models

Add Entity Framework

Just like you added the Entity Framework to the MVC project, you need to add it to this project as well. Right mouse click on the project and select **Manage NuGet Packages…** Click on the Browse tab and search for Entity Framework. Click the Install button as shown in Figure 11.

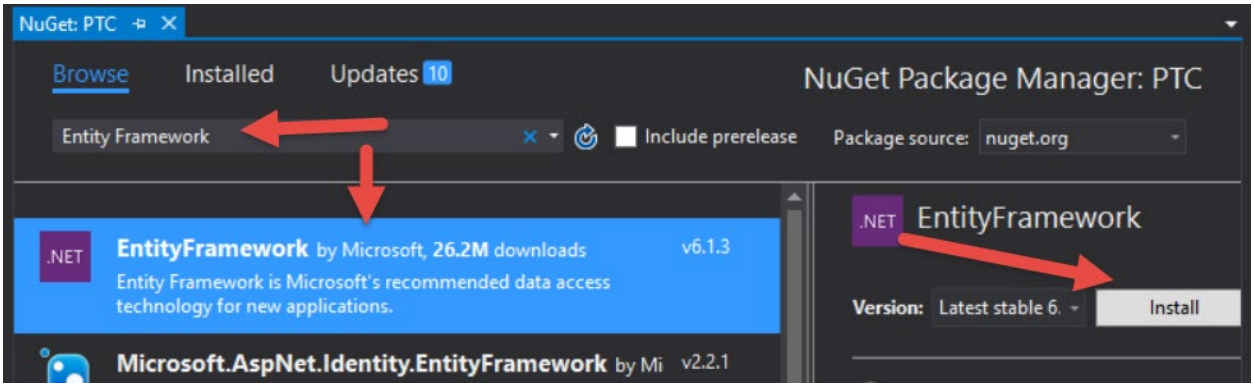

Figure 11: Add the Entity Framework to your View Model layer

Add References

Now that you have all the projects created, you need to add the appropriate references in each. Add a reference to the **PTC.DataLayer** and **PTC.ViewModelLayer** project from the **PTC** project. Add a reference to the **PTC.DataLayer** project from the **PTC.ViewModelLayer** project.

Add Connection String to PTC Project

Copy the <connectionString> element from the PTC.DataLayer app.config file to the PTC web.config file.

```
<connectionStrings>
  <add name="PTCData"
       connectionString="YOUR CONNECT STRING HERE"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

At this point, your solution should look like Figure 12.

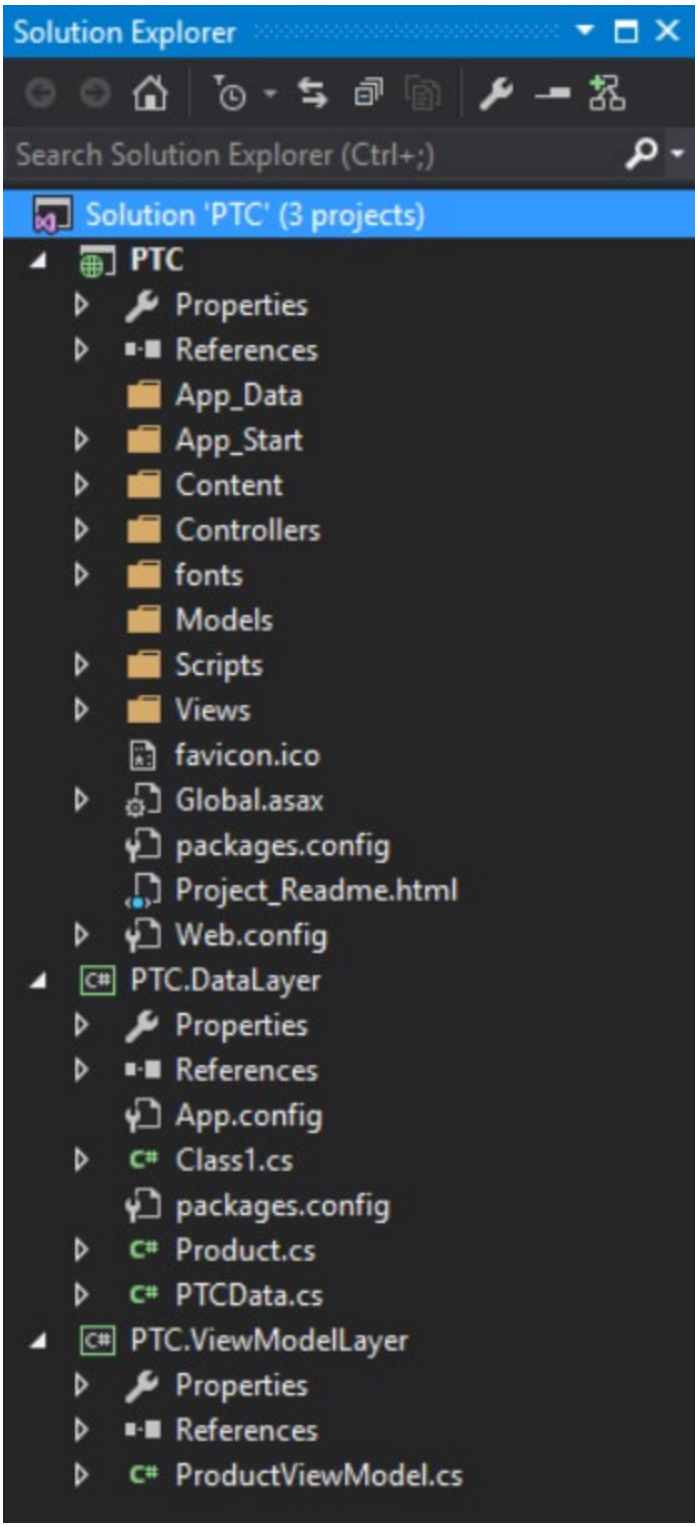

Figure 12: Your solution is now ready to start working within.

Retrieve Product Data

Let's start to write code in the ProductViewModel to retrieve a collection of Products using the Entity Framework generated code. Open the ProductViewModel class and add a couple of **using** statements:

```
using PTC.DataLayer;
using System.Collections.Specialized;
```

Add a property to the view model class to hold the collection of products.

```
public List<Product> DataCollection { get; set; }
```

Add another property to hold any messages to display to the user.

```
public string Message { get; set; }
```

Add an Init() method to initialize the DataCollection property and message property.

```
public void Init() {
  // Initialize properties in this class
  DataCollection = new List<Product>();
  Message = string.Empty;
}
```

Add a constructor to call the Init() method

```
public ProductViewModel()
  : base() {
  Init();
}
```

Add a couple of methods to handle exceptions. You are not going to do any exception publishing in this blog post, but you want to have the methods there so you can add it easily later.

```
public void Publish(Exception ex, string message) {
  Publish(ex, message, null);
}

public void Publish(Exception ex, string message,
                    NameValueCollection nvc) {
  // Update view model properties
  Message = message;

  // TODO: Publish exception here

}
```

Add a method named BuildCollection() that calls the PTCData class to retrieve the list of products from the database table.

```
protected void BuildCollection() {
  PTCData db = null;

  try {
    db = new PTCData();

    // Get the collection
    DataCollection = db.Products.ToList();
  }
  catch (Exception ex) {
    Publish(ex, "Error while loading products.");
  }
}
```

HandleRequest Method

You are going to use the View Model class to handle many requests from your UI. Instead of exposing many different methods from the ProductViewModel class, let's create a single public method named HandleRequest(). For now, you are just going to call the BuildCollection() method from this method. However, later, you are going to add a switch…case statement to handle many different requests.

```
public void HandleRequest() {
  BuildCollection();
}
```

Create Product Controller

Now that you have the data layer and the view model classes created and ready to return data, you need a controller that can call our HandleRequest() method. Go back to the PTC project and right mouse click on the \Controllers folder. Select **Add | Controller…** from the menu. Choose the **MVC 5 Controller – Empty** template and click the Add method as shown in Figure 13.

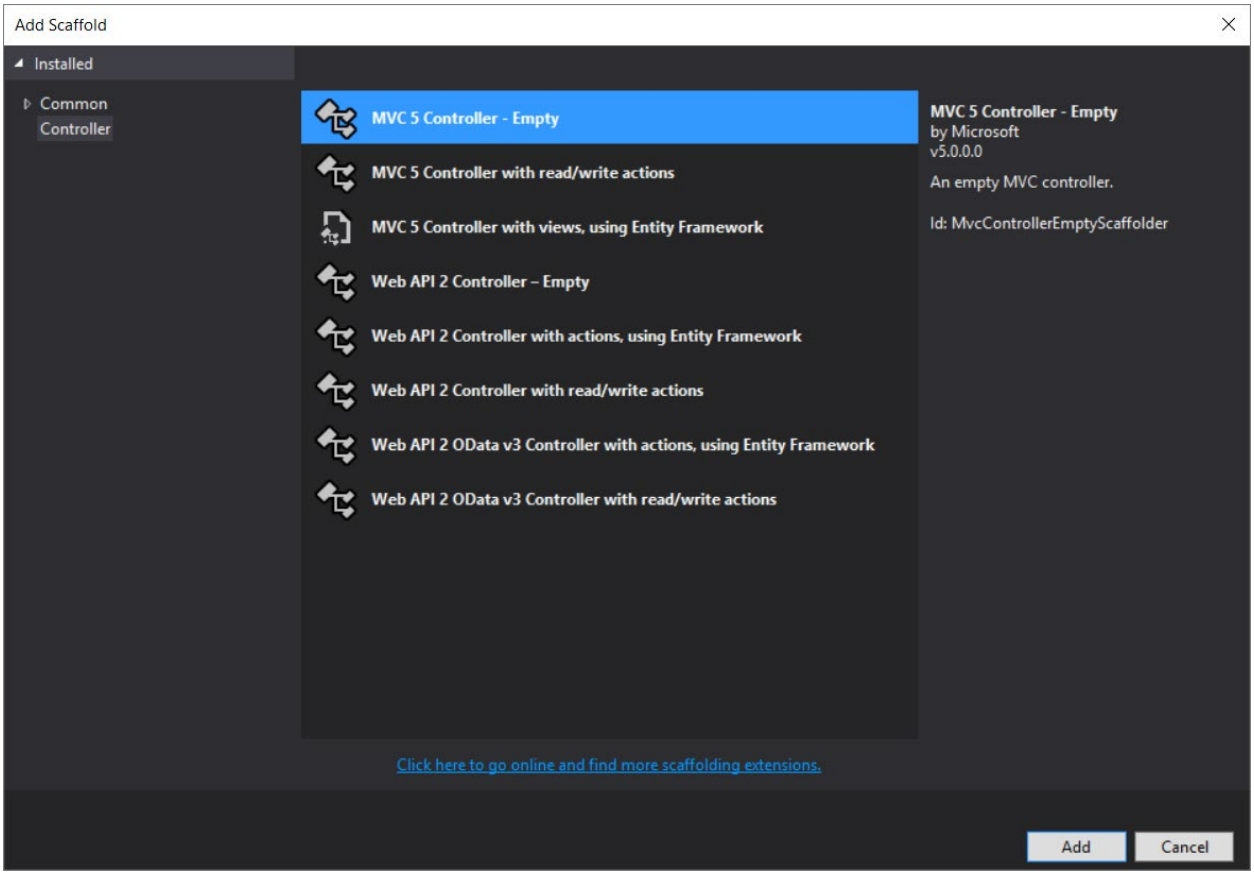

Figure 13: Add an empty MVC contoller

When prompted to set the Controller name, type in ProductController and click the Add button as shown in Figure 14.

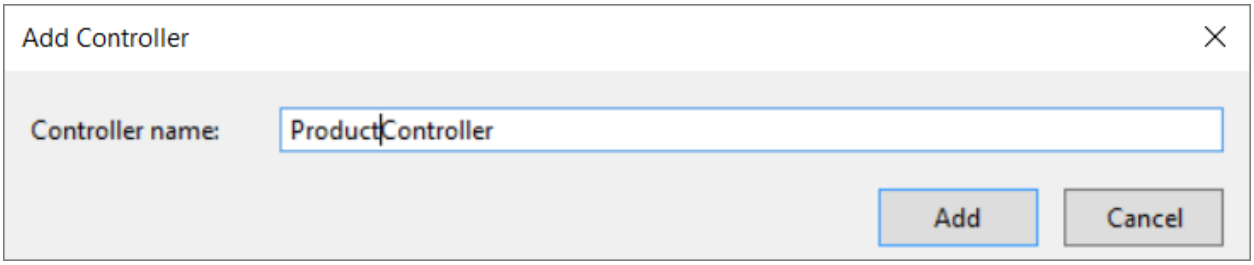

Figure 14: Create your Product controller

You will now see a controller class in your \Controllers folder that looks like the following.

```
public class ProductController : Controller
{
  // GET: Product
  public ActionResult Index() {
    return View();
  }
}
```

Add a using statement at the top of this class so you can use the ProductViewModel class in this controller.

```
using PTC.ViewModelLayer;
```

Modify the GET method to look like the following code.

```
public ActionResult Product() {
  ProductViewModel vm = new ProductViewModel();

  vm.HandleRequest();

  return View(vm);
}
```

List Products

Under the \Views folder see if you have a \Product folder already. If you don't, then add one. This folder is generally created when you add a controller called ProductController. Right mouse click on the \Views\Product folder and

select the **Add | MVC 5 View Page with Layout (Razor)** from the menu as shown in Figure 15.

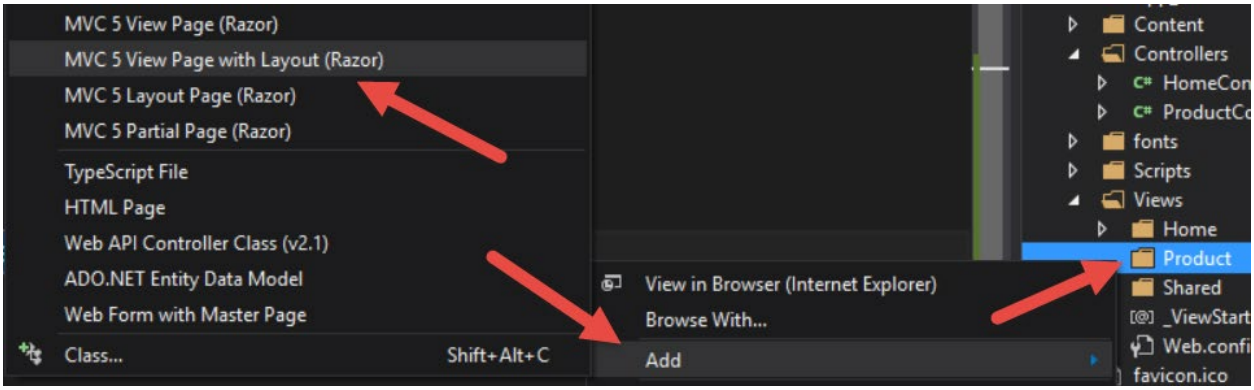

Figure 15: Add an MVC View with a shared layout page

When prompted to set the name for the item, type in Product as shown in Figure 16.

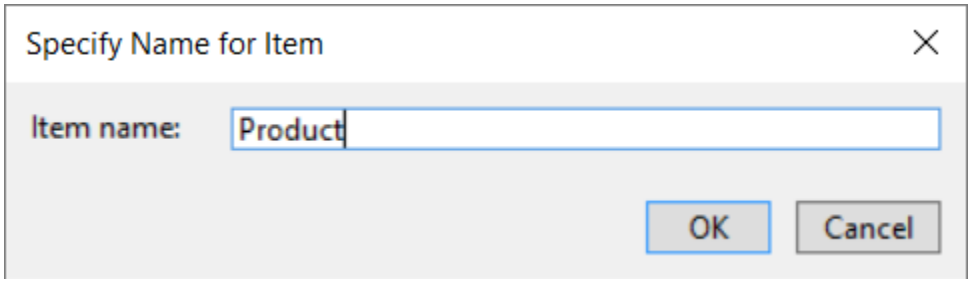

Figure 16: Create your Product MVC page

After setting the name you will be prompted to select the shared layout page. Select the \Views\Shared_Layout.cshtml page as shown in Figure 17.

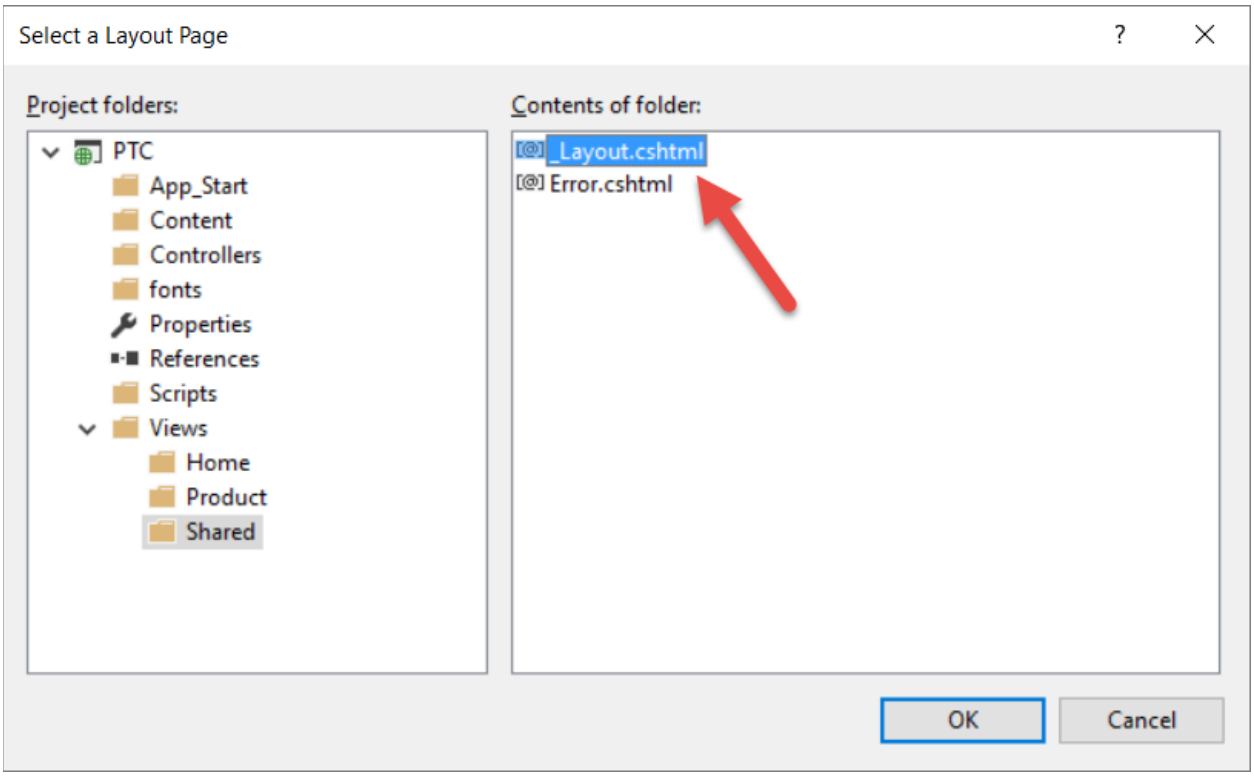

Figure 17: Select the _Layout.cshtml shared layout page

After the MVC page is added, add two statements as shown below.

```
@using PTC.ViewModelLayer
@model ProductViewModel
```

Write the code to display a table of product data. Add the following HTML after the code shown above.

```
<div class="table-responsive">
  <table class="table table-condensed table-bordered
               table-striped table-hover">
    <thead>
      <tr>
        <th>Product Name</th>
        <th>Introduction Date</th>
        <th>Url</th>
        <th>Price</th>
      </tr>
    </thead>

    <tbody>
      @foreach (var item in Model.DataCollection) {
        <tr>
          <td>@item.ProductName</td>
          <td>
            @Convert.ToDateTime(item.IntroductionDate)
                .ToShortDateString()
          </td>
          <td>@item.Url</td>
          <td>
            @Convert.ToDecimal(item.Price).ToString("c")
          </td>
        </tr>
      }
    </tbody>
  </table>
</div>
```

Run the page and you should see a page that looks like the following.

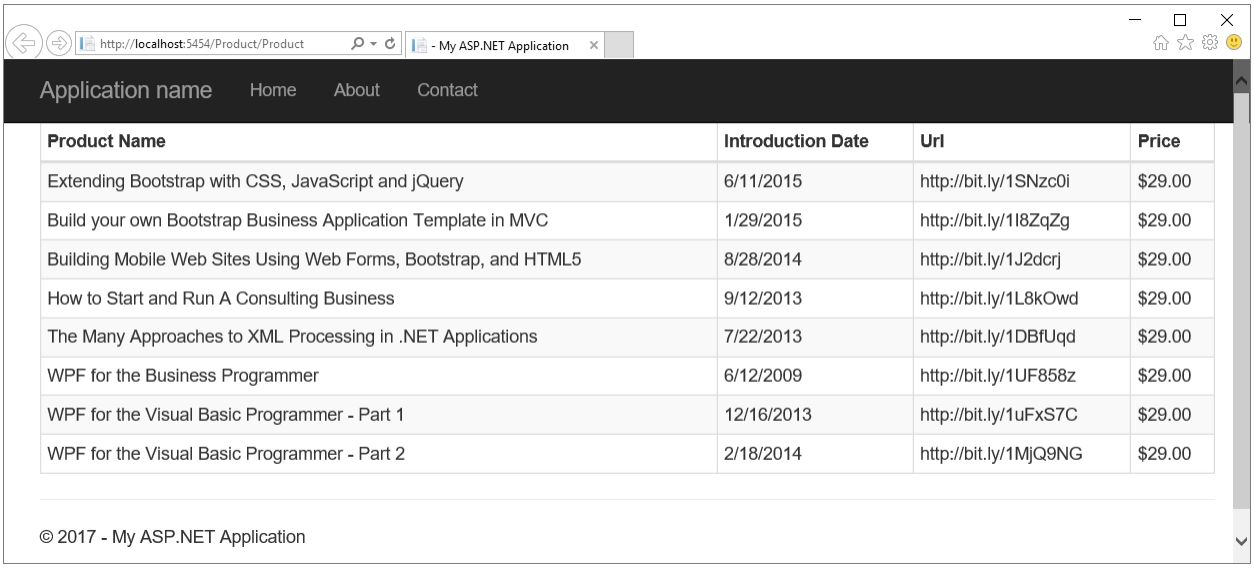

Figure 18: You should now see some product data on your page

Summary

In this blog post, you created the start of an MVC application that is going to use a MVVM design pattern. You learned a few reasons why using an MVVM approach is a solid design decision in almost any kind of application. In the next blog post you learn to search for products and break up your single MVC page into a couple of different partial pages.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category “PDSA Blogs”, then locate the sample **Using MVVM in MVC Applications**.