Highlights of this tutorial

- MVC Framework - Introduction
- MVC Framework - Architecture
- MVC Framework - ASP.NET Forms
- MVC Framework - First Application
- MVC Framework - Folders
- MVC Framework - Models
- MVC Framework - Controllers
- MVC Framework - Views

MVC Framework – Introduction

What is MVC?

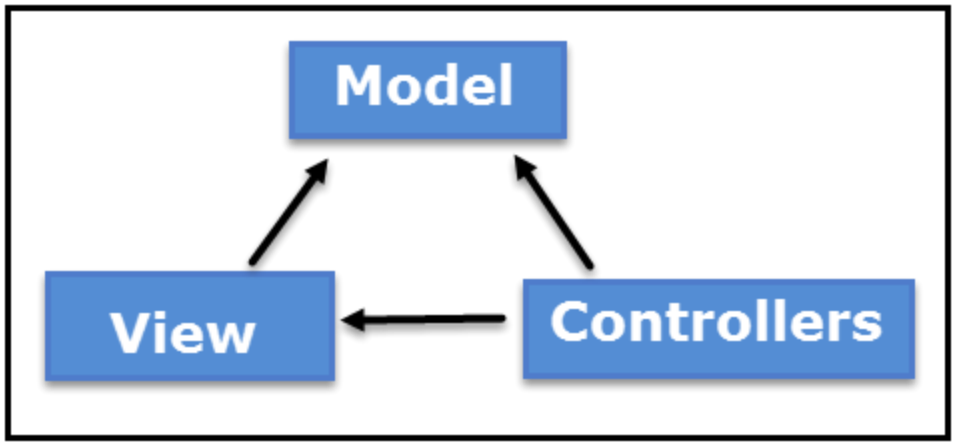
The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the **view**, and the **controller**. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

MVC Components

Model: The Model component corresponds to all the data related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

View: The View component is used for all the UI logic of the application. For example, the Customer view would include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller: Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller would handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller would be used to view the Customer data.



ASP.NET MVC

ASP.NET supports three major development models: Web Pages, Web Forms and MVC (Model View Controller). The ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with existing ASP.NET features, such as master pages, authentication, etc. Within .NET, this framework is defined in the System.Web.Mvc assembly. The latest version of the MVC Framework is 5.0. We use Visual Studio to create ASP.NET MVC applications which can be added as template in Visual Studio.

ASP.NET MVC Features

The ASP.NET MVC provides the following features:

- Ideal for developing complex but light weight applications
- It provides an extensible and pluggable framework which can be easily replaced and customized. For example, if you do not wish to use the in-built Razor or ASPX View Engine, then you can use any other third-party view engines or even customize the existing ones.

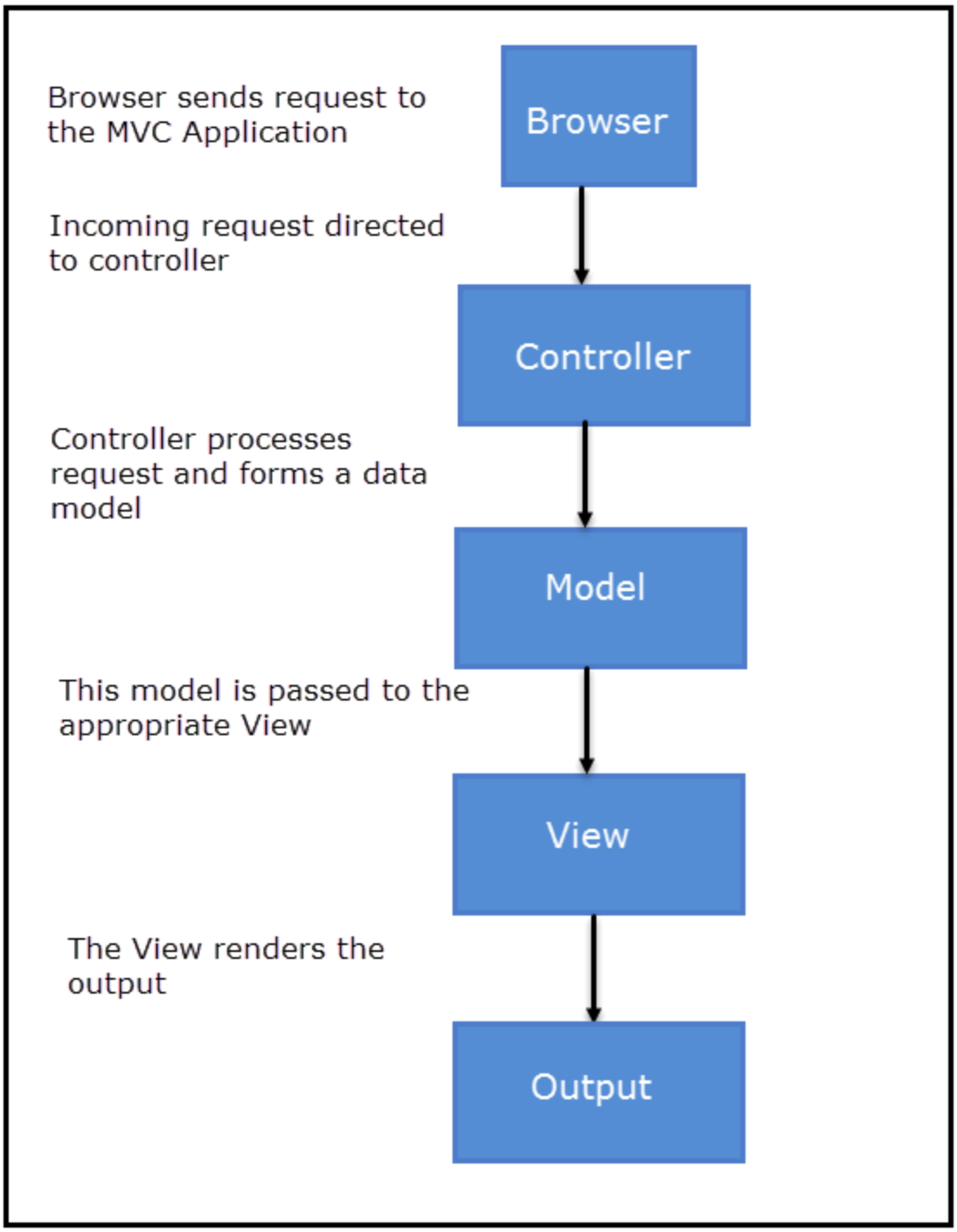
- Utilizes the component-based design of the application by logically dividing it into Model, View and Controller components. This enables the developers to manage the complexity of large-scale projects and work on individual components.
- The MVC structure enhances the test-driven development and testability of the application since all the components can be designed interface-based and tested using mock objects. Hence the ASP.NET MVC Framework is ideal for projects with large team of web developers.
- Supports all the existing vast ASP.NET functionalities such as Authorization and Authentication, Master Pages, Data Binding, User Controls, Memberships, ASP.NET Routing, etc.
- It does not use the concept of View State (which is present in ASP.NET). This helps in building applications which are light-weight and gives full control to the developers.

Thus, you can consider MVC Framework as a major framework built on top of ASP.NET providing a large set of added functionality with focus on component-based development and testing.

MVC Framework – Architecture

In the last chapter, we studied the high-level architecture flow of MVC Framework. Now let us have a look at how the execution of an MVC application takes place when certain request comes from the client. The diagram below shows the flow:

MVC Flow Diagram


Browser sends request to the MVC Application
Browser
Incoming request directed to controller
Controller
Controller processes request and forms a data model
Model
This model is passed to the appropriate View
View
The View renders the output
Output

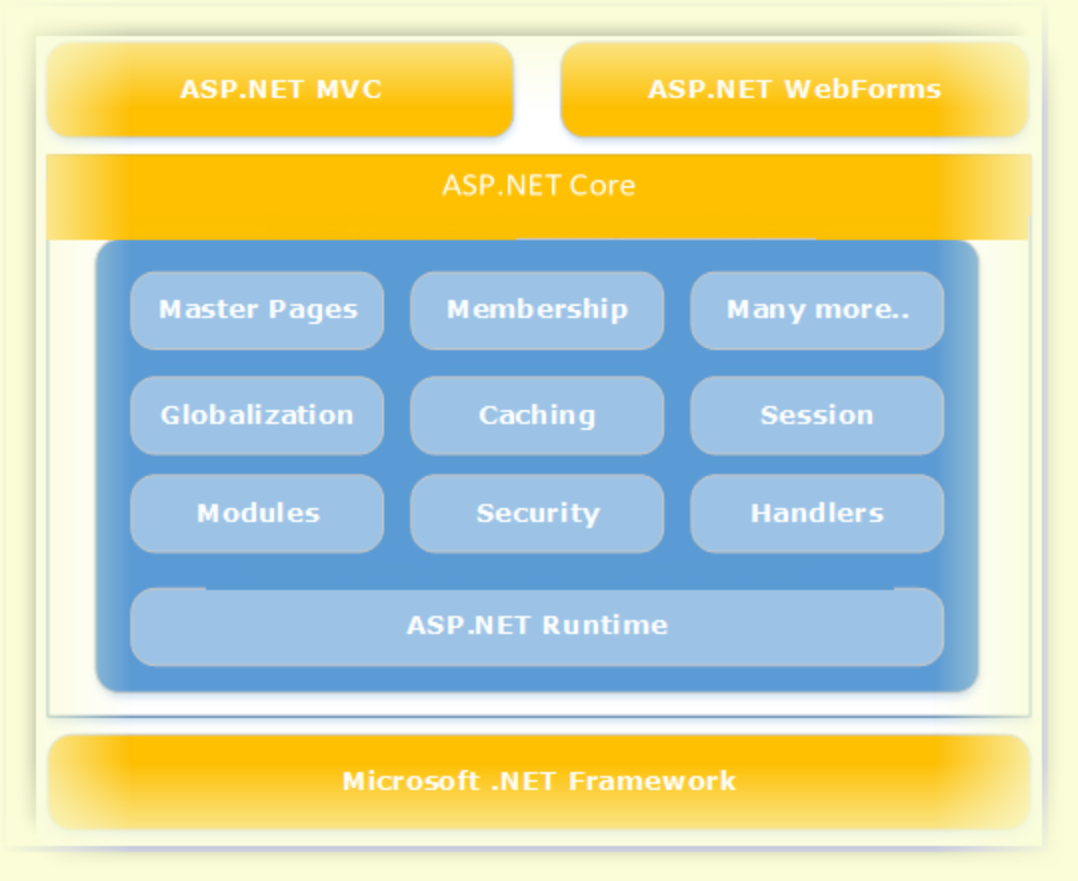
Flow Steps

- The client browser sends request to the MVC Application.
- Global.ascx receives this request and performs routing based on the URL of incoming request using the RouteTable, RouteData, UrlRoutingModule and MvcRouteHandler objects.
- This routing operation calls the appropriate controller and executes it using the IControllerFactory object and MvcHandler object's Execute method.
- The Controller processes the data using Model and invokes the appropriate method using ControllerActionInvoker object
- The processed Model is then passed to the View which in turn renders the final output.

MVC Framework - ASP.NET Forms

MVC and ASP.NET Web Forms are inter-related but different models of development depending on the requirement of the application and other factors. At a high level, you can consider that MVC is a more advanced and sophisticated web application framework designed with separation of concerns and testability in mind. Both the frameworks have their advantages and disadvantages depending on specific requirements. This concept can be visualized using the following diagram:

MVC and ASP.NET Diagram



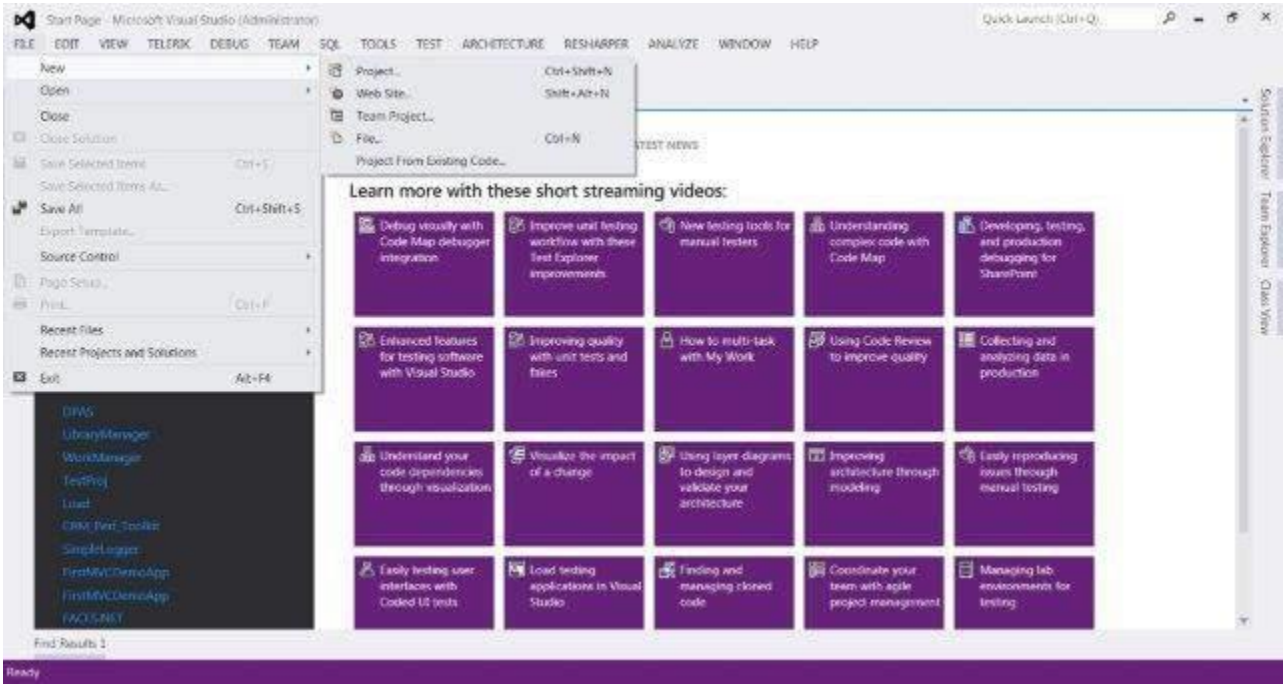
Comparison Table

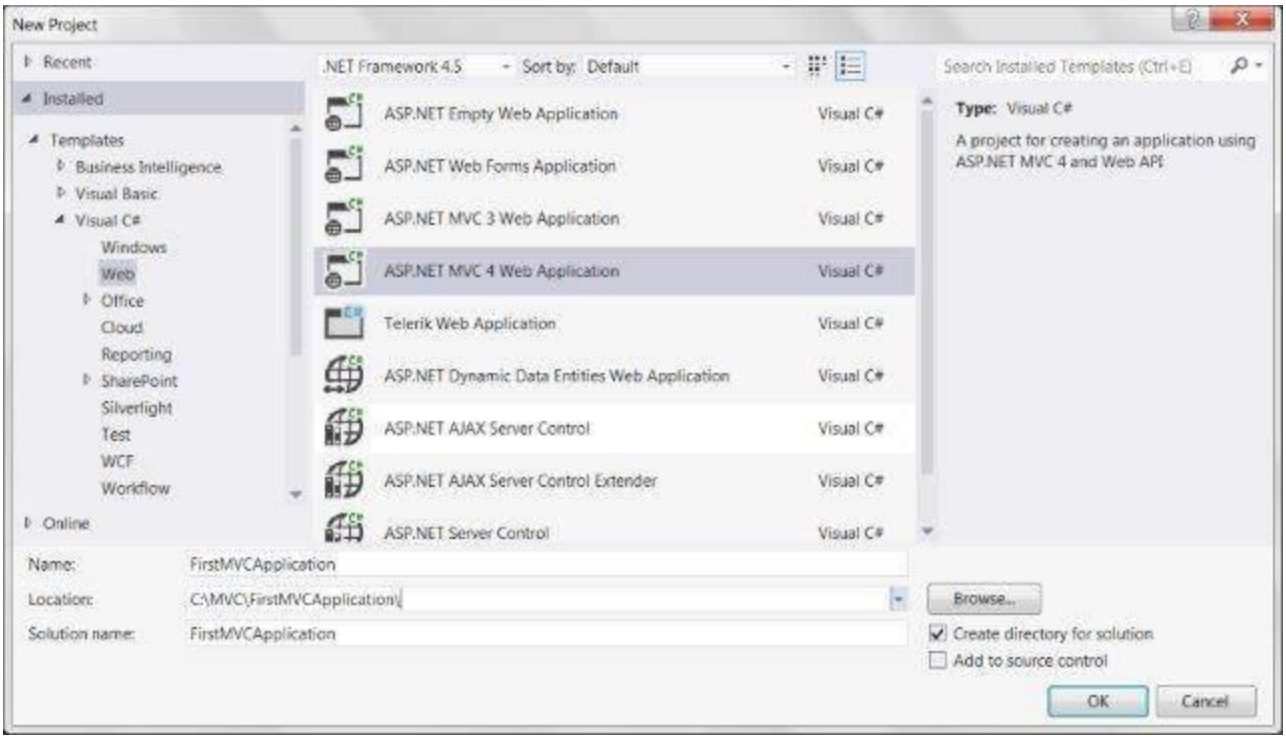
Comparison Factors	ASP.NET Web Forms	ASP.NET MVC
Rendering Approach	Follows Page Control pattern approach for rendering layout.	Front Controller Approach is used.
Separation of Concern	No separation of concerns and all Web Forms are tightly coupled.	Very clean separation of concerns.
Automated Testing	Automated testing is really difficult.	TDD is Quiet simple in MVC.
State	Yes, ViewState is used.	Stateless
Performance	Slow due to Large ViewState.	Fast due to clean approach and no ViewState.
Life Cycle	ASP.NET WebForms model follows a Page Life cycle.	No Page Life cycle like WebForms.
Controls	Lots of Server Side controls.	No out of box controls.3rd Party controls can be used.
Control Over Layout	The above abstraction was good but provides limited control over HTML, JavaScript and CSS which is necessary in many cases.	Full control over HTML, JavaScript and CSS.
RAD support	Yes	No
Scalability	It's good for small scale applications with limited team size.	It's better as well as recommended approach for large-scale applications.

MVC Framework - First Application

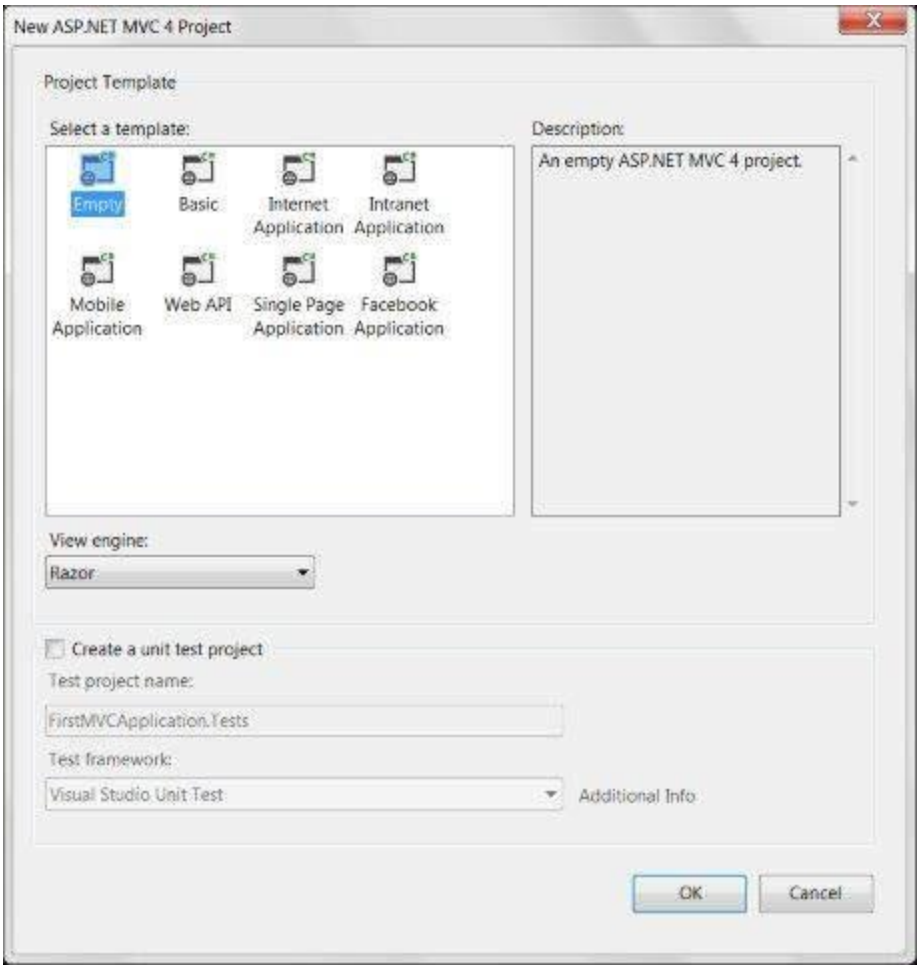
Steps to create first MVC Application

Step 1: Start your Visual Studio and select File->New->Project. Select Web->ASP.NET MVC Web Application and name this project as FirstMVCApplication. Select the Location as C:\MVC. Click OK.

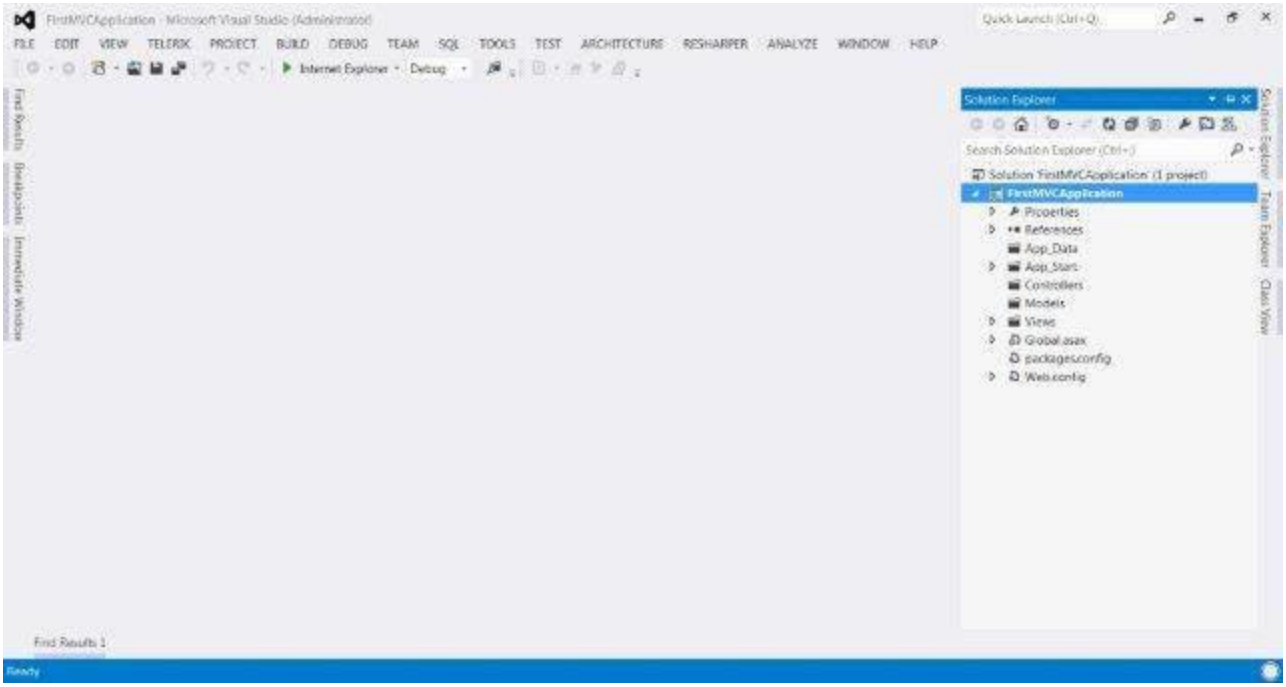




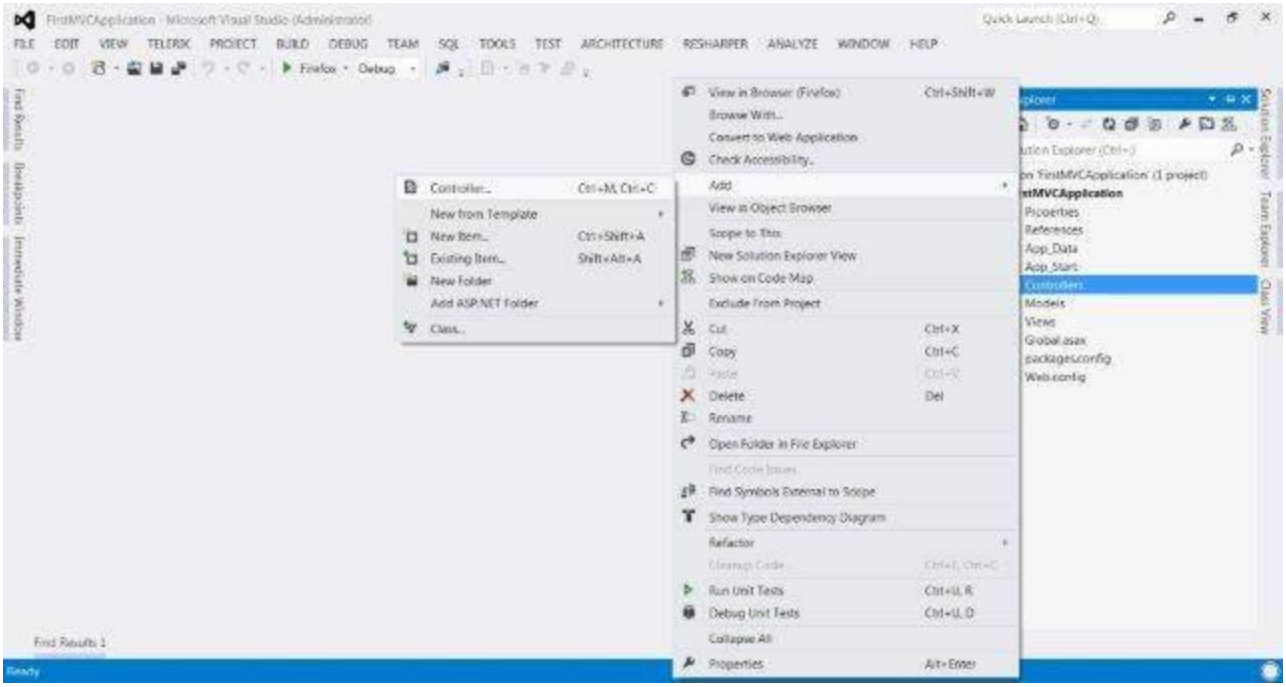
Step 2: This will open the Project Template option. Select Empty template and View Engine as Razor. Click OK.

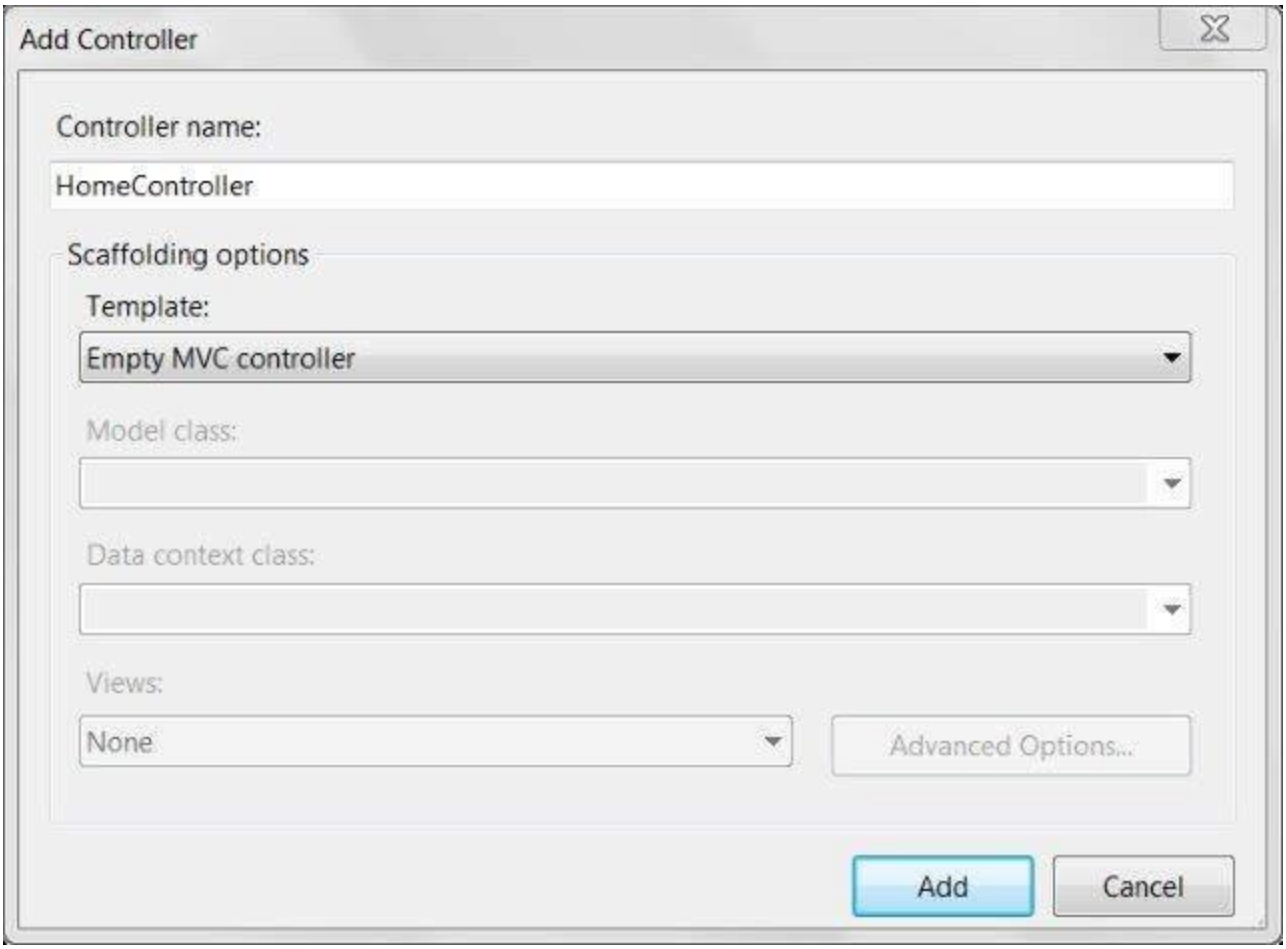


By this, Visual Studio will create our first MVC project like this (shown in screenshot):



Step 3: Now we will create the first Controller in our application. Controllers are just simple C# classes which contains multiple public methods, known as action methods. To add a new Controller, right click the Controllers folder in our project and select Add->Controller. Name the Controller as HomeContoller and click Add.





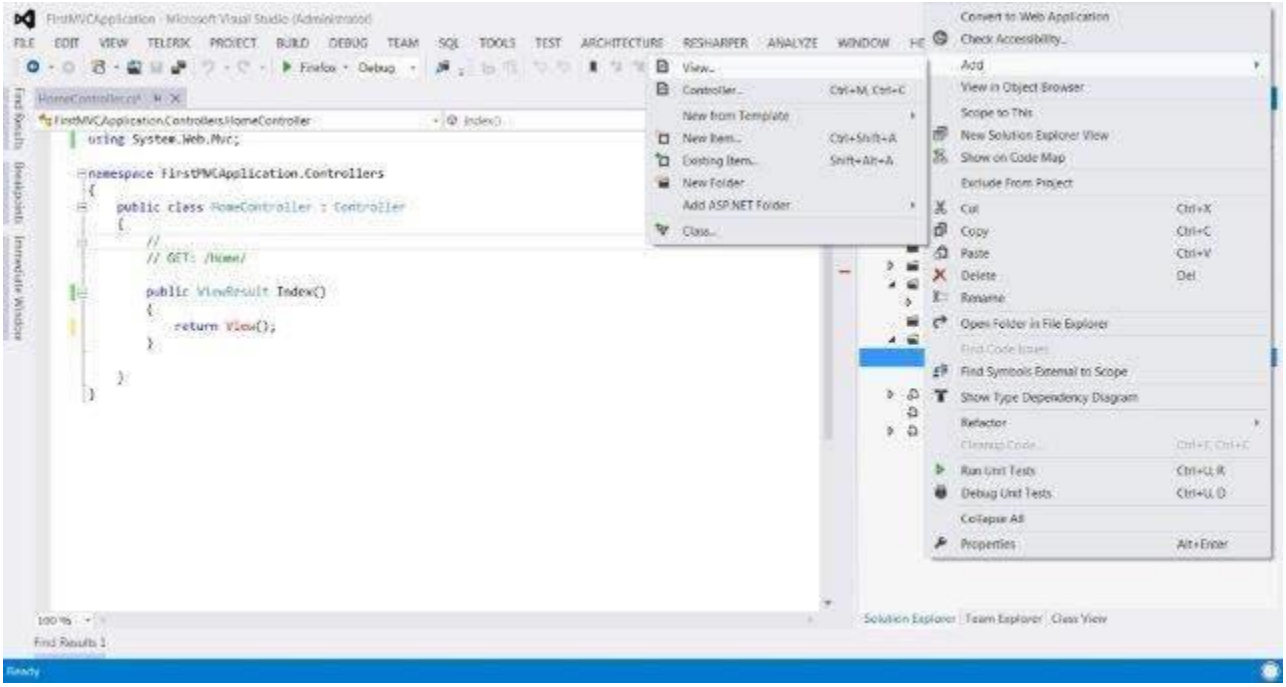
This will create a class file HomeController.cs under the Controllers folder with the following default code.

```
using System;
using System.Web.Mvc;


namespace FirstMVCApplication.Controllers
{
    public class HomeController : Controller
    {
     public ViewResult Index()
        {
            return View();
        }
```

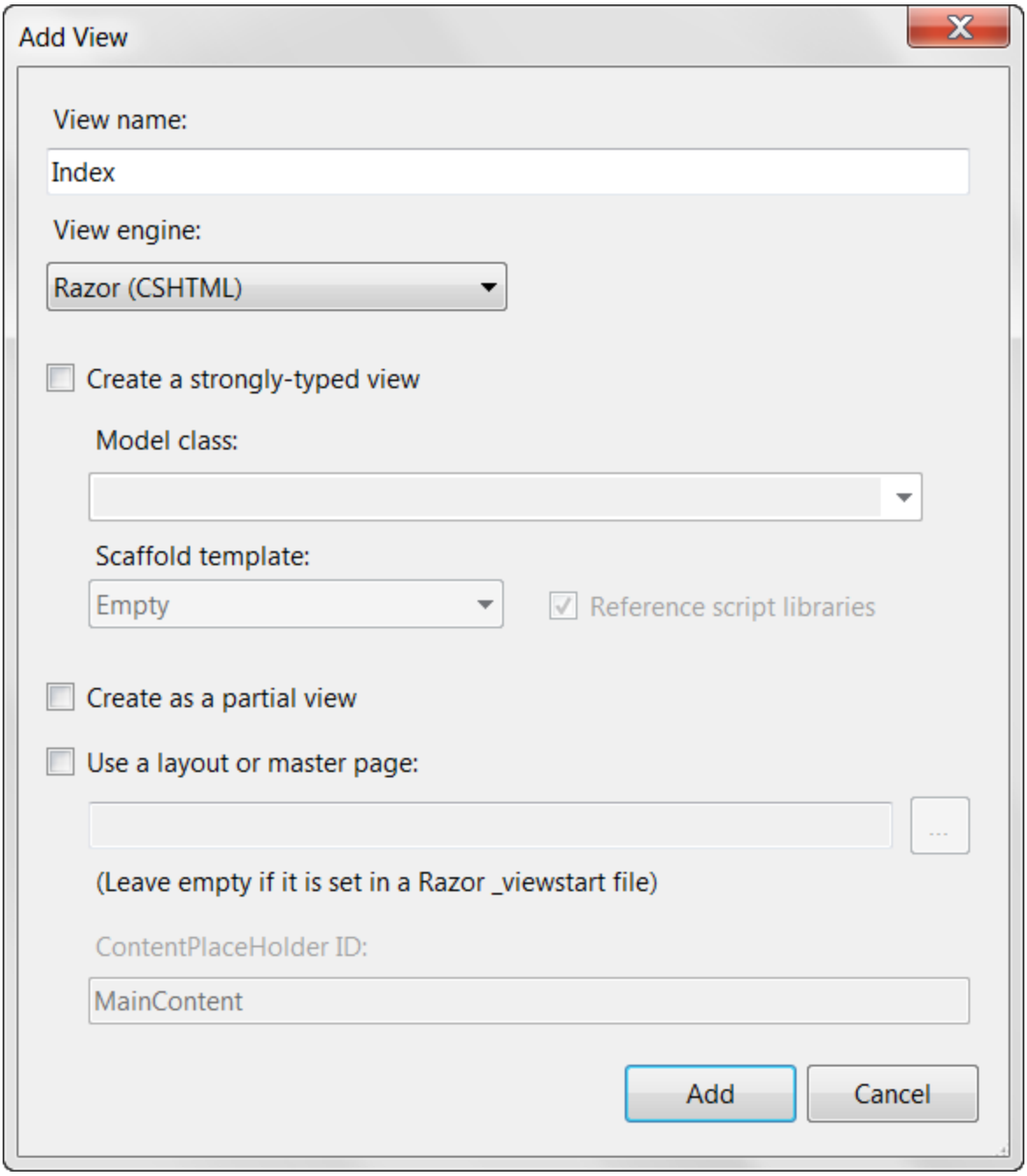
```
    }
}
```

The above code basically defines a public method Index inside our HomeController and returns a ViewResult object. In the next steps, we will learn how to return a View using the ViewResult object.

Step 4: Now we will add a new View to our Home Controller. To add a new View, right click view folder and click Add->View.



Name the new View as Index and View Engine as Razor (SCHTML). Click Add.



This will add a new cshtml file inside Views/Home folder with the following code:

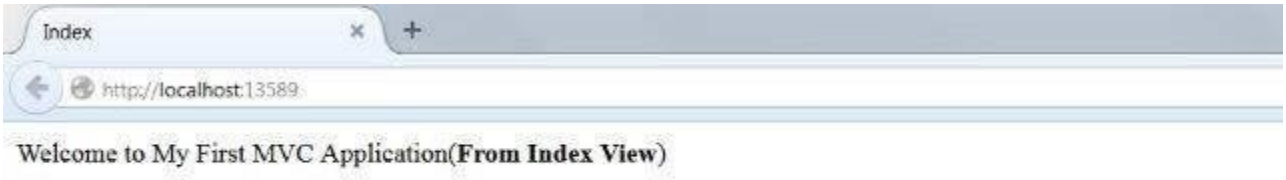
```
@{
    Layout = null;
}

<html>
<head>
```

```
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>


    </div>
</body>
</html>
```

Step 5: Modify the above View's body content with the following code:

```
<body>
    <div>
     Welcome to My First MVC Application (<b>From Index View</b>)
    </div>
</body>
```

Step 6: Now run the application. This will give you the following output in the browser. This output is rendered based on the content in our View file. The application first calls the Controller which in turn calls this View and gives the output.



Step 7: In Step 6, the output we received was based on the content of our View file and had no interaction with the Controller. Moving a step forward, now we will now create a small example to display a Welcome message with current time using interaction of View and Controller.

MVC uses the ViewBag object to pass data between Controller and View. Open the HomeController.cs and edit the Index function to the following code.

```
public ViewResult Index()
```

```
{
    int hour = DateTime.Now.Hour;

    ViewBag.Greeting =
    hour < 12
    ? "Good Morning. Time is" +  DateTime.Now.ToShortTimeString()
    : "Good Afternoon. Time is " + DateTime.Now.ToShortTimeString();

return View();
}
```

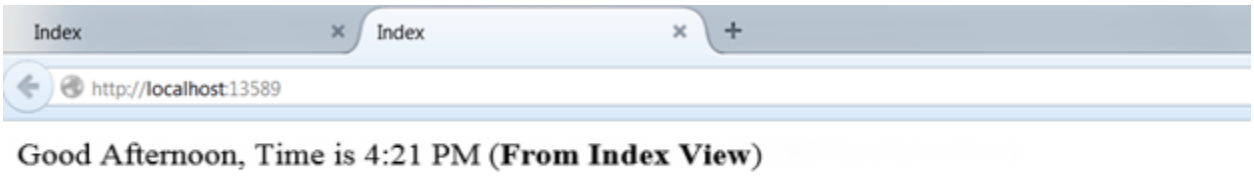
In the above code, we set set the value of the Greeting attribute of the ViewBag object. The code checks the current hour and returns the Good Morning/Afternoon message accordingly using return View() statement. Note that here Greeting is just an example attribute that we have used with ViewBag object. You can use any other attribute name in place of Greeting.

Step 8: Now open the Index.cshtml and copy the following code in the body section:

```
<body>
    <div>
     @ViewBag.Greeting (<b>From Index View</b>)
    </div>
</body>
```

In the above code, we are accessing the value of Greeting attribute of the ViewBag object using @ (which would be set from the Controller).

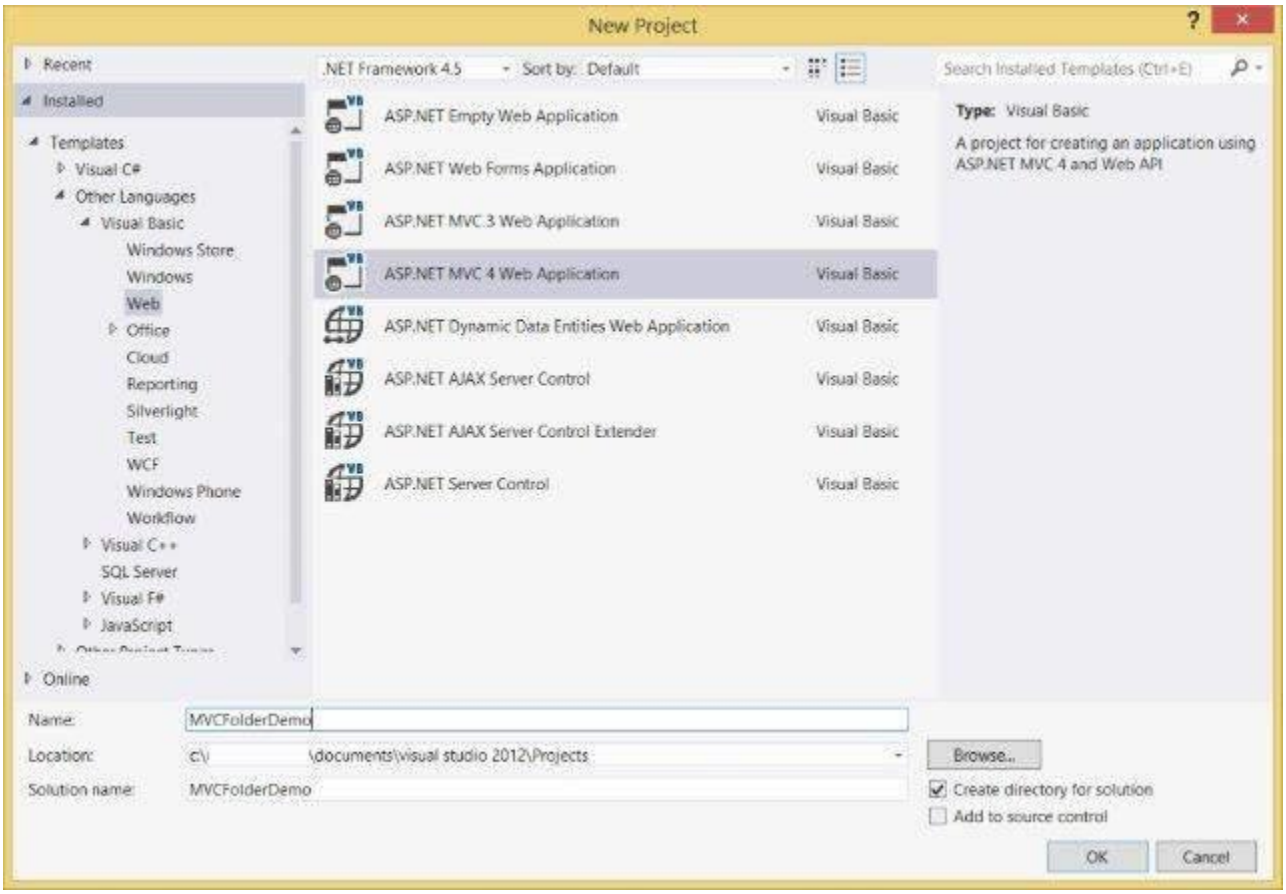
Step 9: Now run the application again. This time our code will run the Controller first, set the ViewBag and then render it using the View code. The output would look like below:



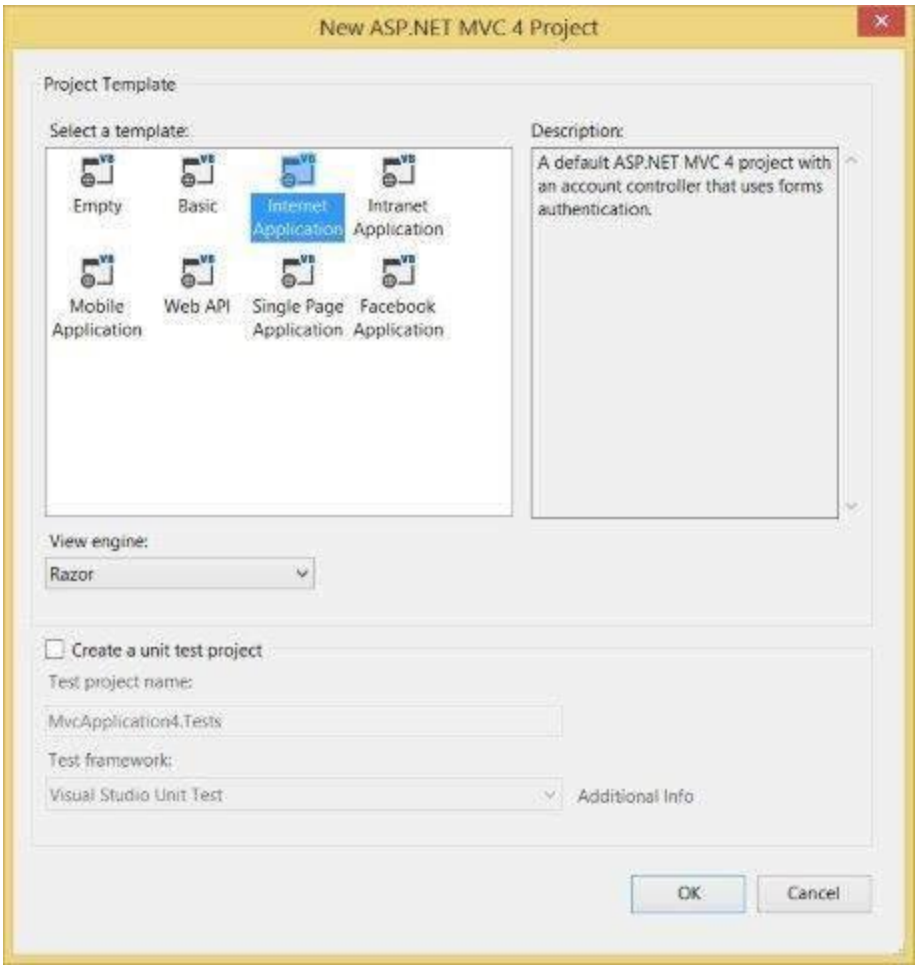
MVC Framework – Folders

Now that we have already created a sample MVC application, let us understand the folder structure of an MVC project. We will create new MVC project to learn this.

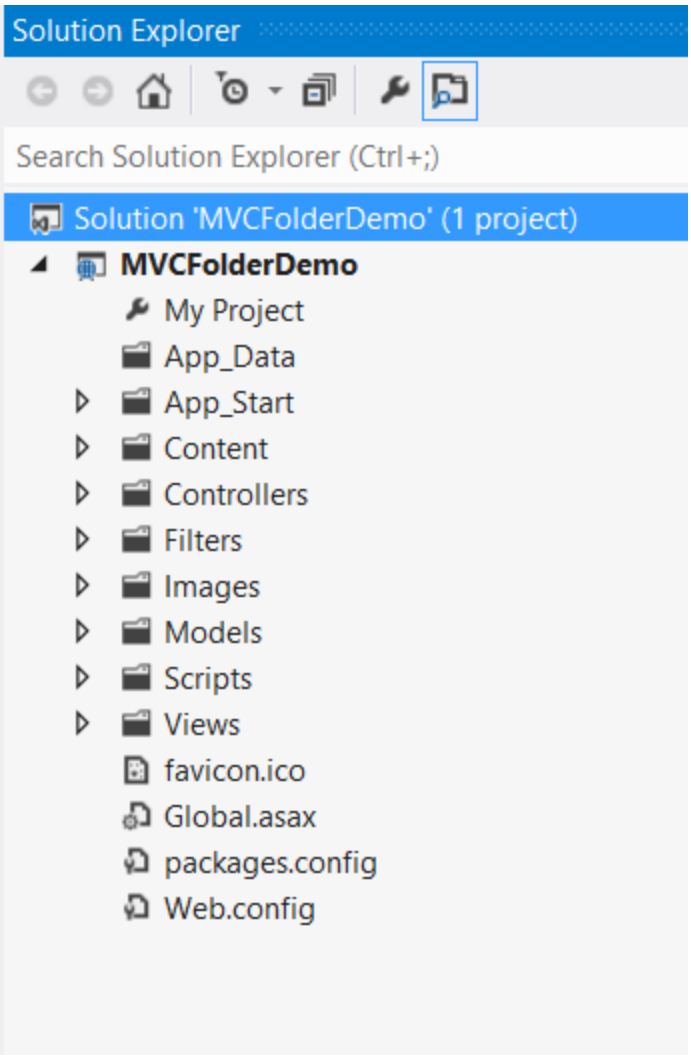
In your visual studio, open File->New->Project and select ASP.NET MVC Application. Name it as MVCFolderDemo.



Click OK. In the next window, select Internet Application as the Project Template and click OK.



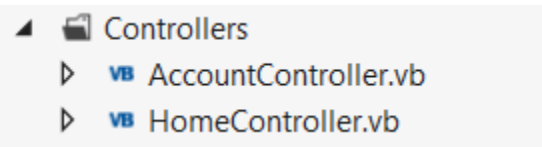
This will create a sample MVC application as shown below:



Note that the filers present in this project are coming out of the default template that we have selected. These may change slightly as per different versions.

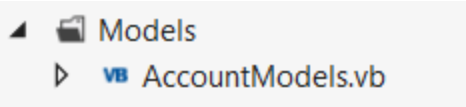
Controllers Folder

- This folder will contain all the Controller classes. MVC requires name of all the controller files to end with Controller.
- In our example, the Controllers folder contains two class files: AccountController and HomeController.



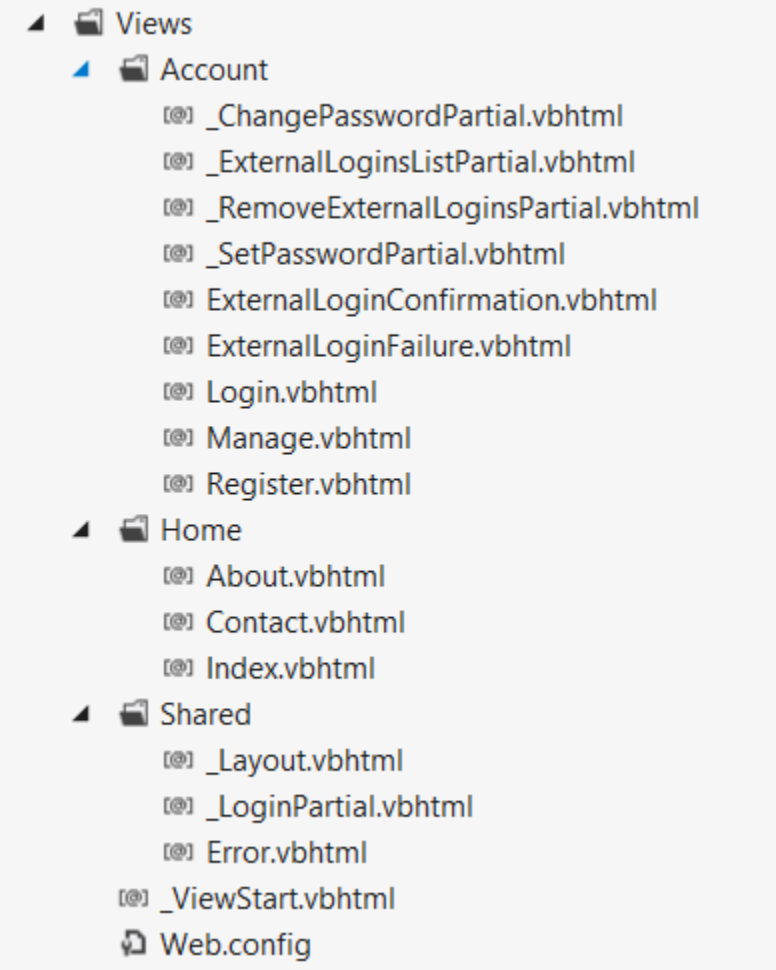
Models Folder

- This folder will contain all the Model classes which are used to work on application data.
- In our example, the Models folder contains AccountModels. You can open and look at the code in this file to see how the data model is created for managing accounts in our example.



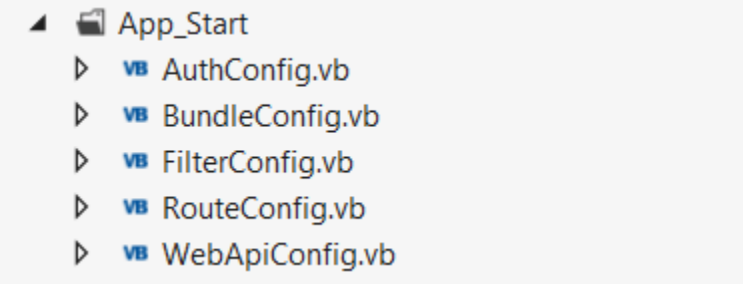
Views Folder

- This folder stores the HTML files related to application display and user interface.
- It contains one folder for each controller.
- In our example, you will see three sub-folders under Views namely Account, Home and Shared which contains html files specific to that view area.



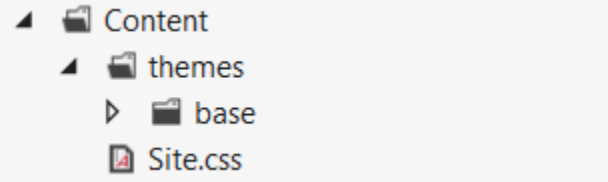
App_Start Folder

- This folder contains all the files which are needed during the application load.
- For e.g., the RouteConfig file is used to route the incoming URL to the correct Controller and Action



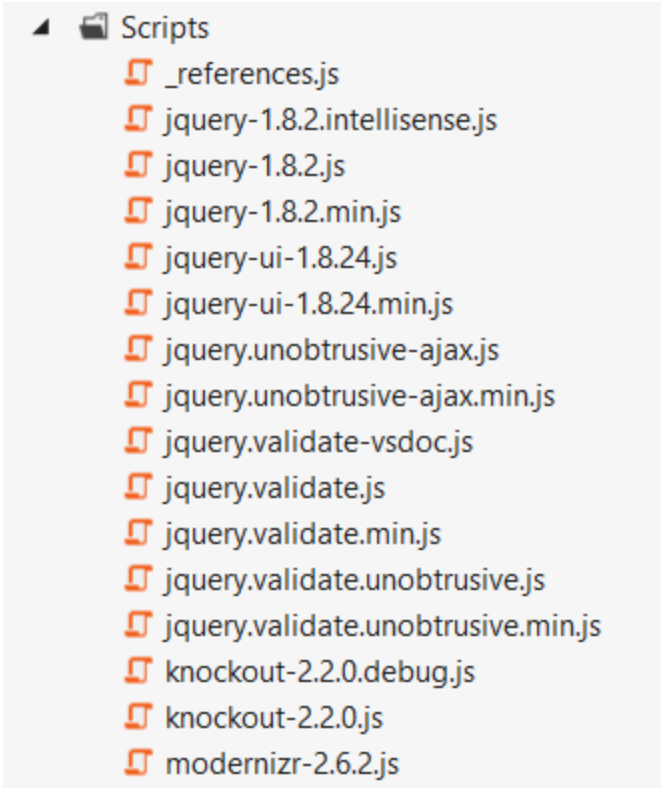
Content Folder

- This folder contains all the static files such as css, images, icons, etc.
- The Site.css file inside this folder is the default styling that the application applies.



Scripts Folder

- This folder stores all the JS files in the project. By default Visual Studio adds MVC, jQuery and other standard JS libraries.


Scripts
_references.js
jquery-1.8.2.intellisense.js
jquery-1.8.2.js
jquery-1.8.2.min.js
jquery-ui-1.8.24.js
jquery-ui-1.8.24.min.js
jquery.unobtrusive-ajax.js
jquery.unobtrusive-ajax.min.js
jquery.validate-vsdoc.js
jquery.validate.js
jquery.validate.min.js
jquery.validate.unobtrusive.js
jquery.validate.unobtrusive.min.js
knockout-2.2.0.debug.js
knockout-2.2.0.js
modernizr-2.6.2.js

MVC Framework – Models

The model is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

Model classes can either be created manually or generated from database entities. We are going to see a lot of examples for manually creating Models in coming chapters. So in this chapter we will try the other option, i.e. generating from database so that you have hands-on on both of the methods.

Creating Database Entities

Connect to SQL Server and create a new database.

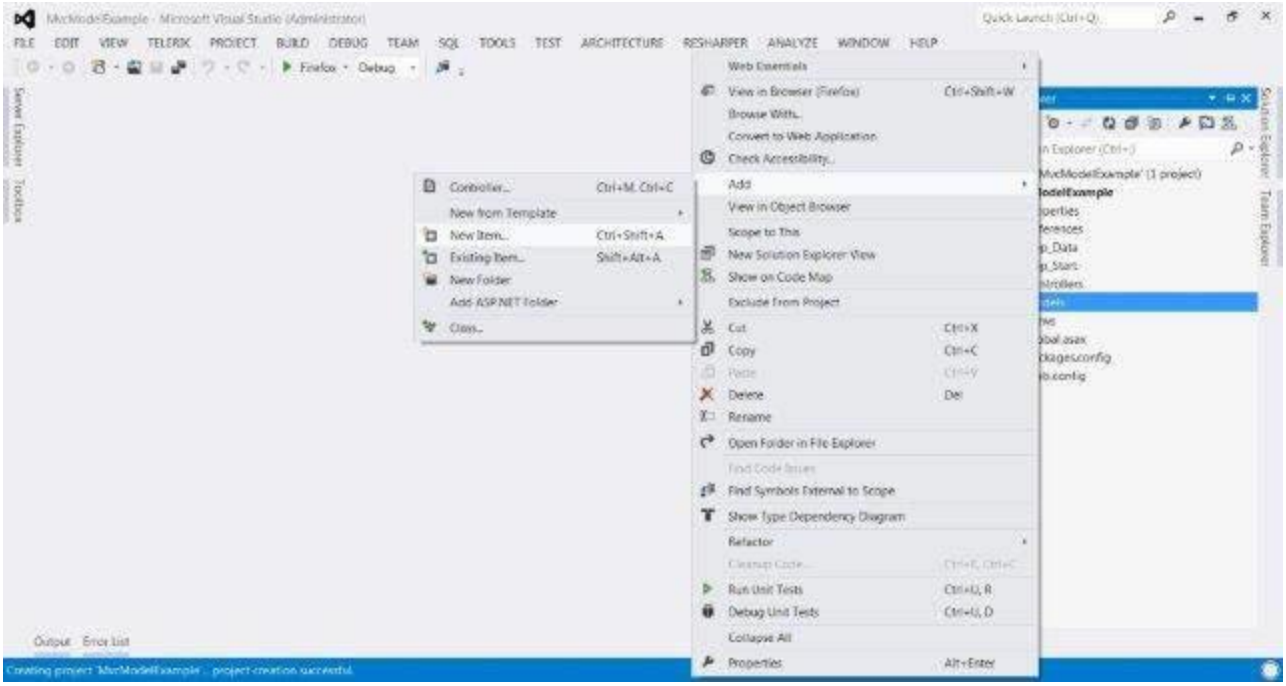
Now run the following queries to create new tables:

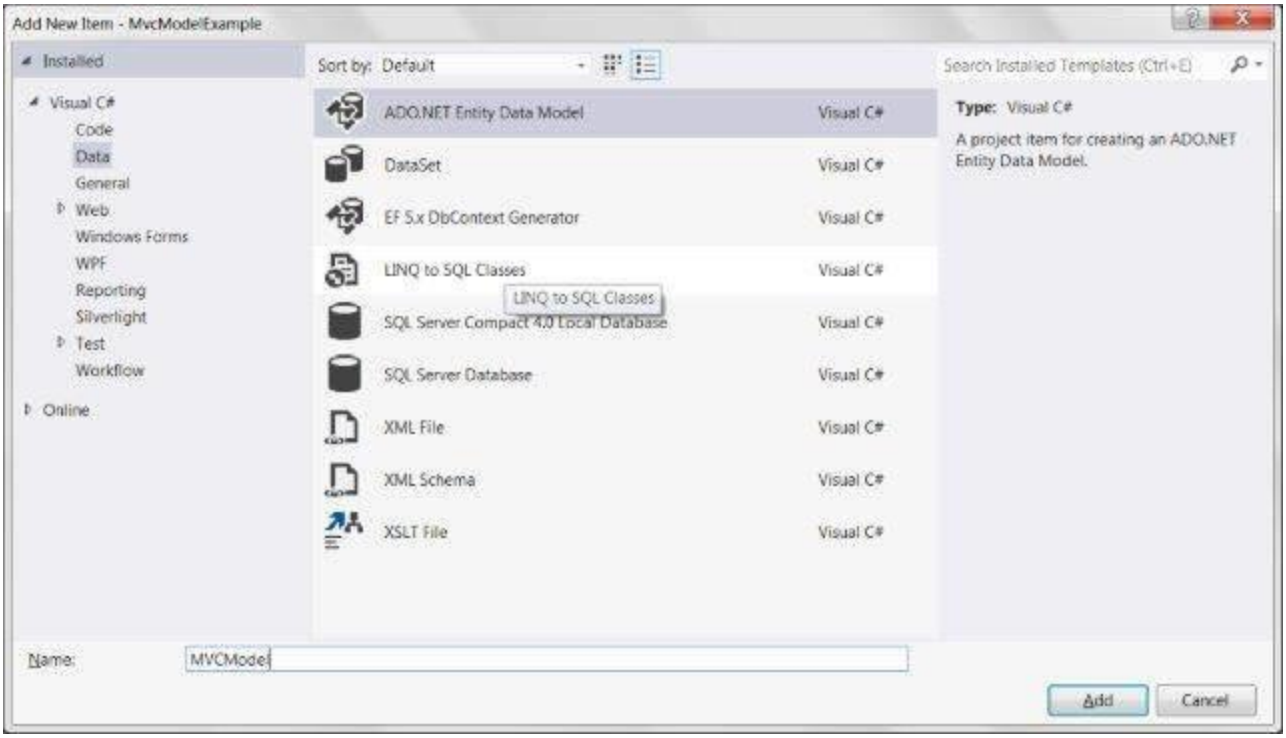
```
CREATE TABLE [dbo].[Student](
[StudentID]      INT           IDENTITY (1,1) NOT NULL,
[LastName]       NVARCHAR (50) NULL,
[FirstName]      NVARCHAR (50) NULL,
[EnrollmentDate] DATETIME      NULL,
    PRIMARY KEY CLUSTERED ([StudentID] ASC)
)

CREATE TABLE [dbo].[Course](
[CourseID] INT           IDENTITY (1,1) NOT NULL,
[Title]    NVARCHAR (50) NULL,
[Credits]  INT           NULL,
    PRIMARY KEY CLUSTERED ([CourseID] ASC)
)

CREATE TABLE [dbo].[Enrollment](
[EnrollmentID] INT IDENTITY (1,1) NOT NULL,
```

```
[Grade]        DECIMAL(3,2) NULL,
[CourseID]     INT NOT NULL,
[StudentID]    INT NOT NULL,
    PRIMARY KEY CLUSTERED ([EnrollmentID] ASC),
    CONSTRAINT [FK_dbo.Enrollment_dbo.Course_CourseID] FOREIGN KEY ([CourseID])
        REFERENCES [dbo].[Course]([CourseID]) ON DELETE CASCADE,
    CONSTRAINT [FK_dbo.Enrollment_dbo.Student_StudentID] FOREIGN KEY ([StudentID])
        REFERENCES [dbo].[Student]([StudentID]) ON DELETE CASCADE
)
```

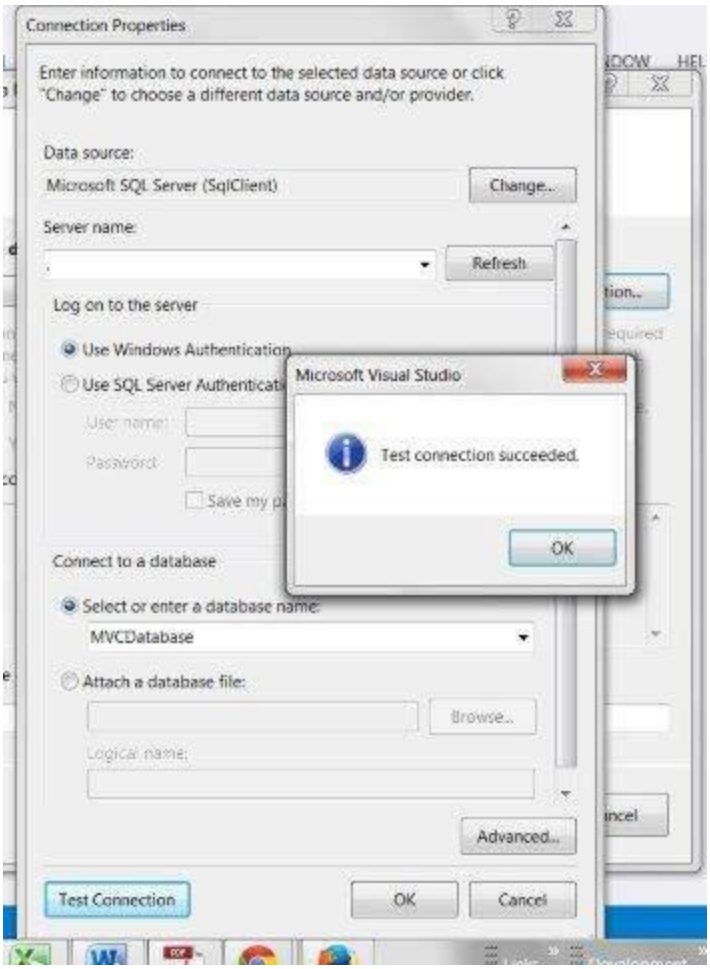
Generating Models using Database Entities

After creating the database and setting up the tables, you can go ahead and create a new MVC Empty Application. Now right click on the Models folder in your project and select Add->New Item and select ADO.NET Entity Data Model.

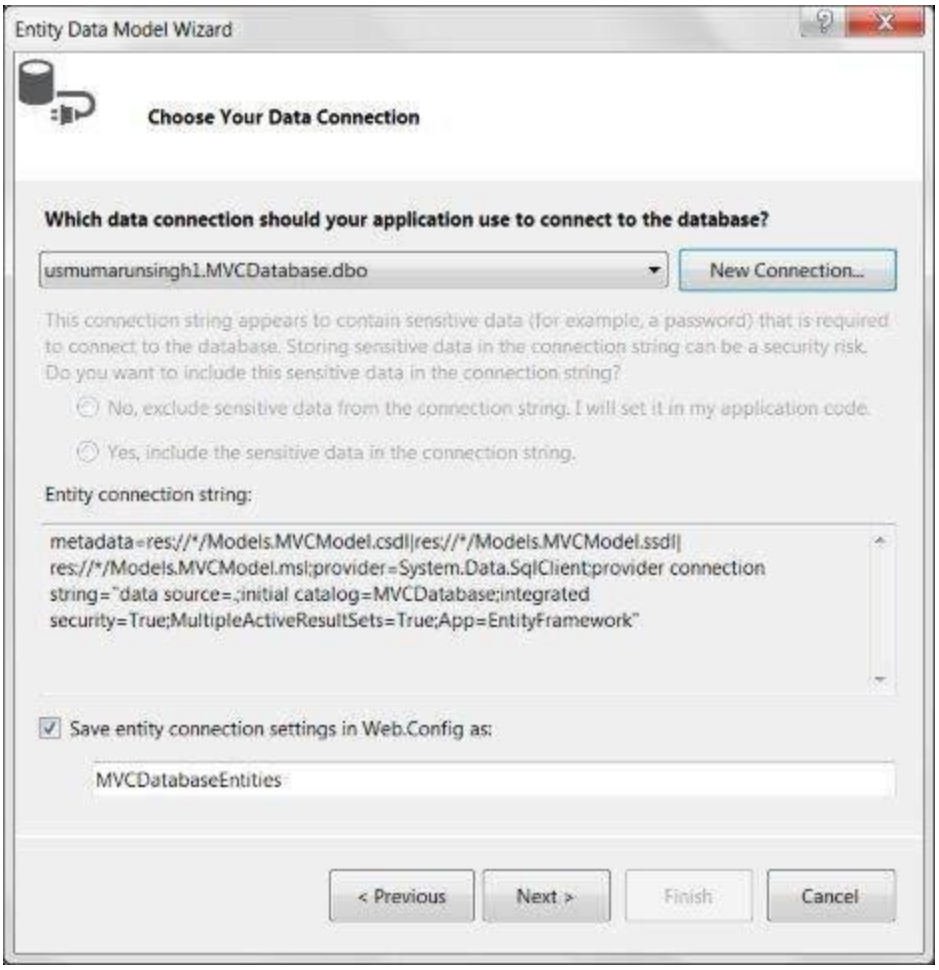




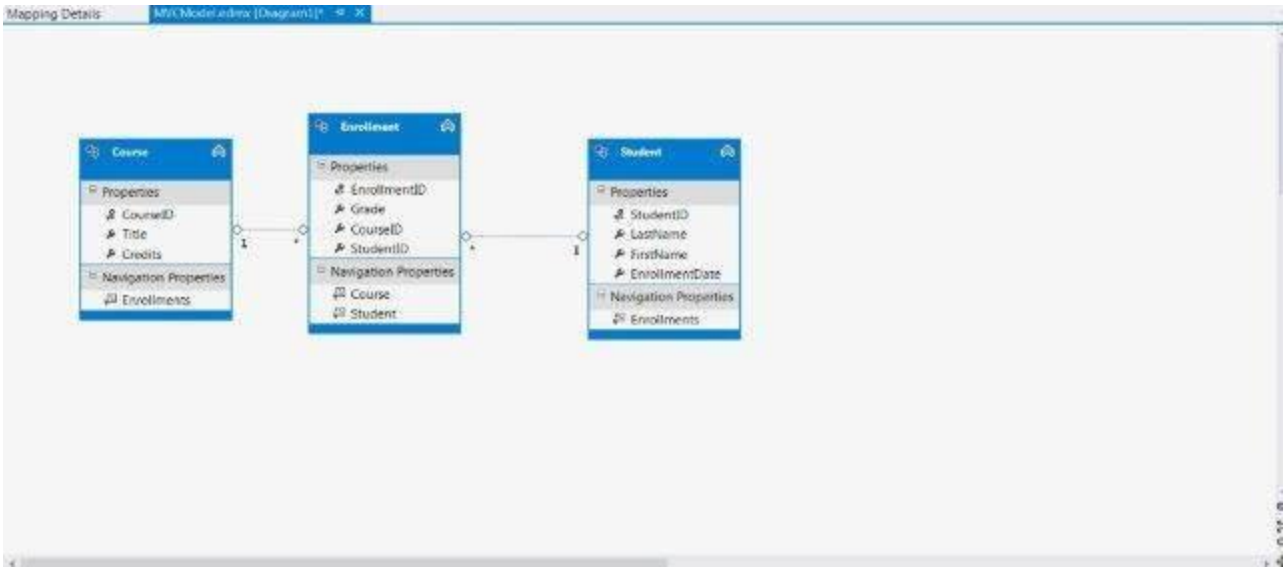
In the next wizard, choose Generate From Database and click Next. Set the Connection to your SQL database.



Select your database and click on Test Connection. A screen similar to this will follow. Click Next.



Select Tables, Views and Stored Procedures and Functions and click Finish. You will see the Model View created something like this:



The above operations would automatically create a Model file for all the database entities. For e.g., the Student table that we created will result in a Model file Student.cs with the following code:

```
namespace MvcModelExample.Models
{
    using System;
    using System.Collections.Generic;

    public partial class Student
    {
        public Student()
        {
            this.Enrollments = new HashSet();
        }

        public int StudentID { get; set; }
        public string LastName { get; set; }
        public string FirstName { get; set; }
        public Nullable EnrollmentDate { get; set; }

        public virtual ICollection Enrollments { get; set; }
    }
}
```

MVC Framework – Controllers

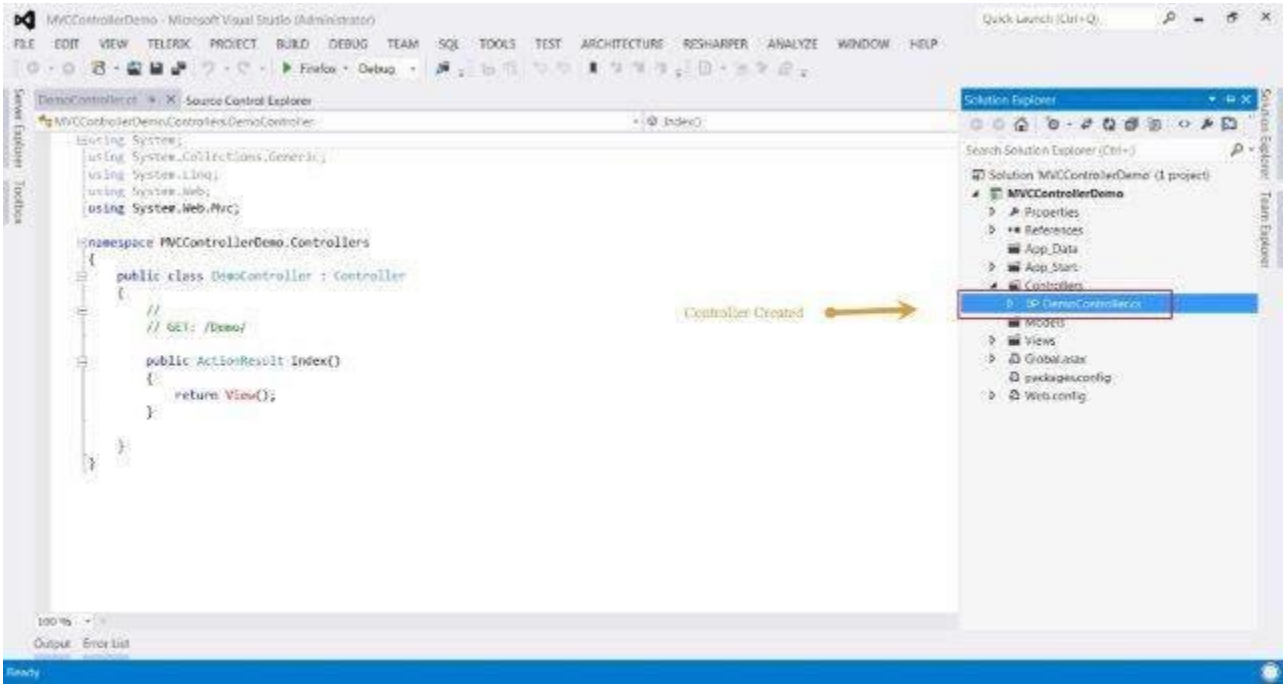
Asp.net MVC Controllers are responsible for controlling the flow of the application execution. When you make a request (means request a page) to MVC applications, a controller is responsible for returning the response to that request. A controller can have one or more actions. A controller action can return different types of action results to a particular request.

Controller is responsible for controlling the application logic and acts as the coordinator between the View and the Model. The Controller receives input from users via the View, then process the user's data with the help of Model and passing the results back to the View.

Creating a Controller

For creating a Controller, create an MVC Empty Application and then right-click on the Controller folder in your MVC application and select the menu option Add->Controller. After selection the Add Controller dialog is being displayed. Name the Controller as DemoController.

A Controller class file will be created like this:

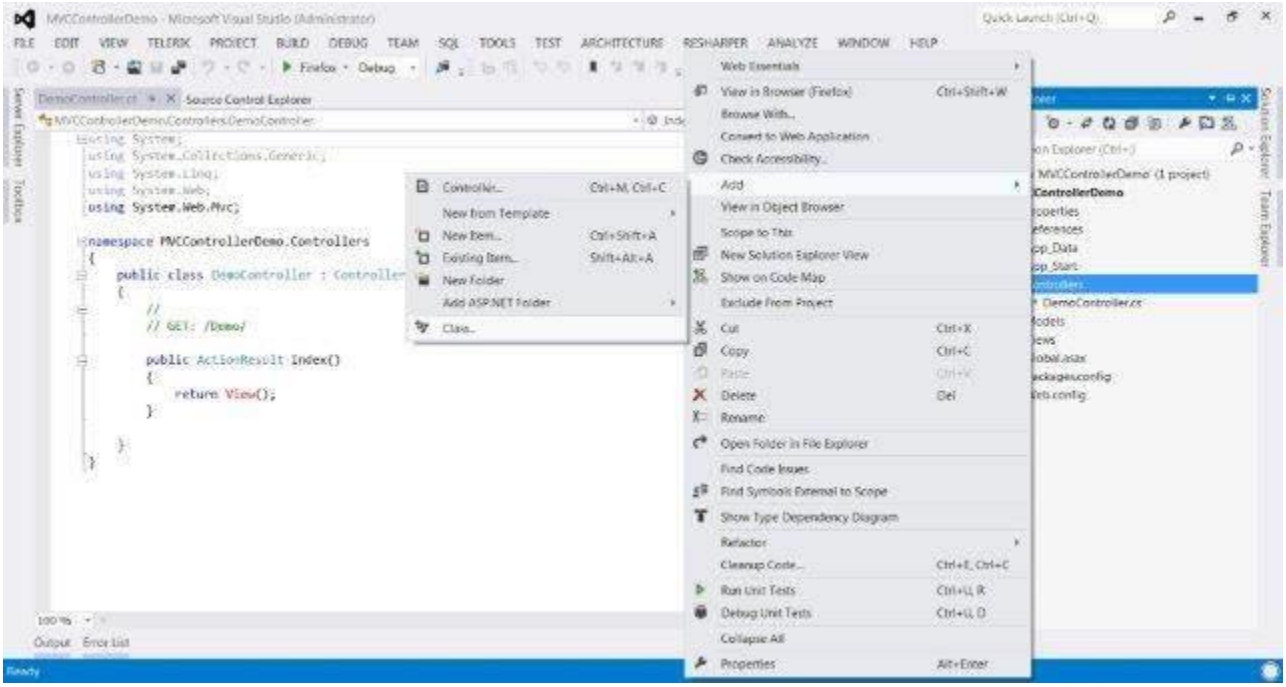


Creating a Controller with Icontroller

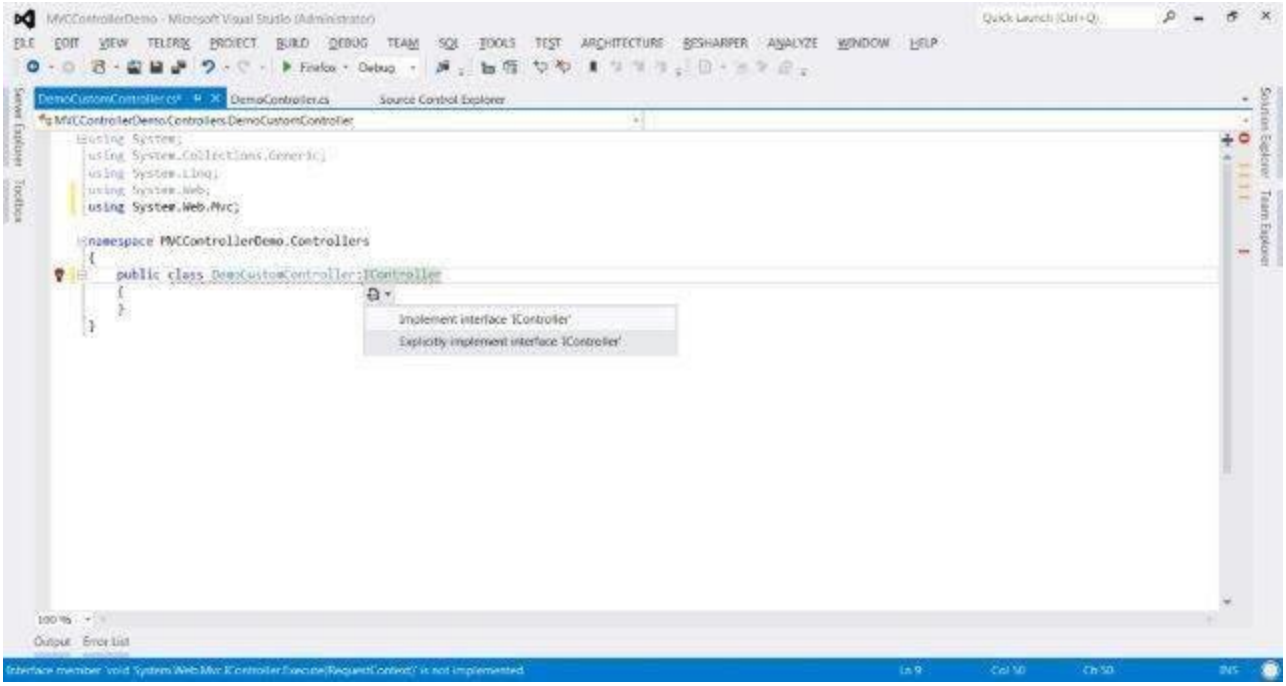
In the MVC Framework, controller classes must implement the IController interface from the System.Web.Mvc namespace.

```
public interface IController
{
void Execute(RequestContext requestContext);
}
```

This is a very simple interface. The sole method, Execute, is invoked when a request is targeted at the controller class. The MVC Framework knows which controller class has been targeted in a request by reading the value of the controller property generated by the routing data.



Add a new class file and name it as DemoCustomController. Now modify this class to inherit IController interface.



Copy the following code inside this class:

```
public class DemoCustomController:IController
    {
        public void Execute(System.Web.Routing.RequestContext requestContext)
        {
            var controller = (string)requestContext.RouteData.Values["controller"];
            var action = (string)requestContext.RouteData.Values["action"];
            requestContext.HttpContext.Response.Write(
            string.Format("Controller: {0}, Action: {1}", controller, action));
        }
    }
```

Now when you will run the application, you will see something like this:

MVC Framework – Views

The MVC Framework comes with two built-in view engines:

1. **Razor Engine:** Razor is a markup syntax that enables the server side C# or VB code into web pages. This server side code can be used to create dynamic content when the web page is being loaded. Razor is an advanced engine as compared to ASPX engine and was launched in the later versions of MVC.

2. **ASPX Engine:** ASPX or the Web Forms engine is the default view engine that is included in the MVC Framework since the beginning. Writing code with this engine is very similar to writing code in ASP.NET Web Forms.

Following are small code snippets comparing both Razor and ASPX engine.

Razor:

```
@Html.ActionLink("Create New", "UserAdd")
```

ASPX:

```
<% Html.ActionLink("SignUp", "SignUp") %>
```

Out of these two, Razor is more advanced View Engine as it comes with compact syntax, test driven development approaches, and better security features. We will use Razor engine in all our examples since it is the most dominantly used View engine.

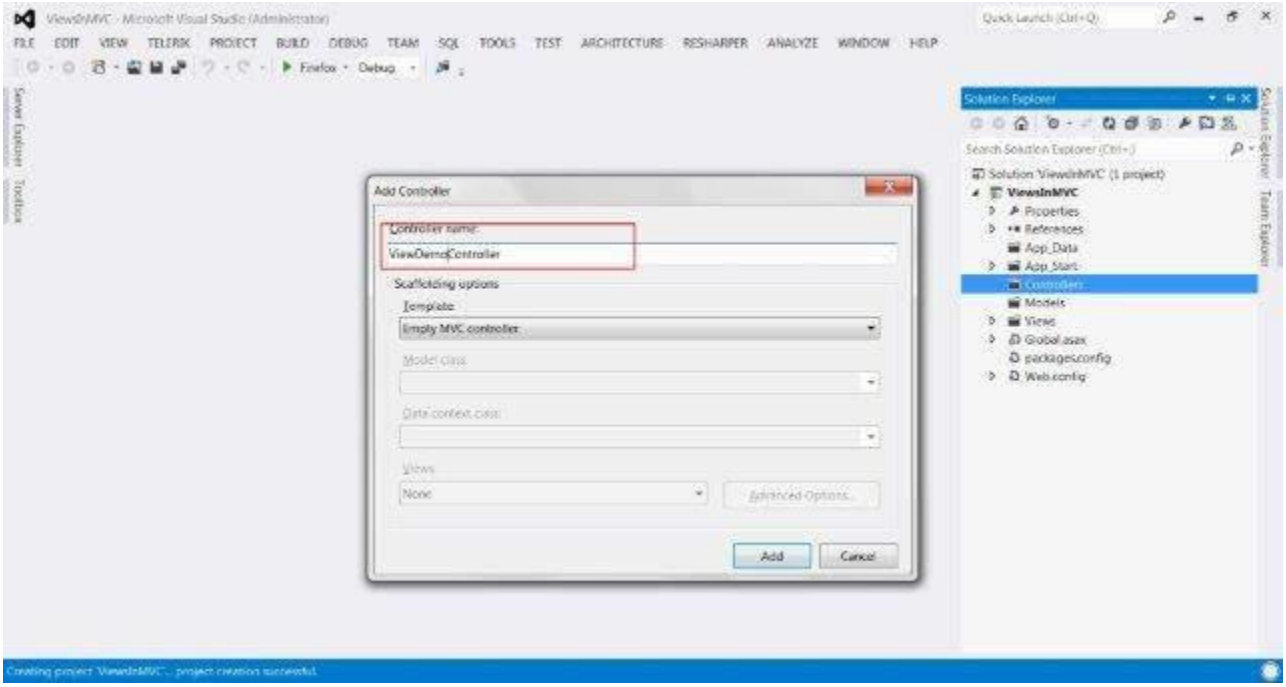
These View Engines can be coded and implemented in following two types:

- Strongly typed
- Dynamic typed

These approaches are similar to early-binding and late-binding respectively in which the models will be bind to the View strongly or dynamically.

Strongly Typed Views

To understand this concept, let us create a sample MVC application (follow the steps in the previous chapters) and add a Controller class file named ViewDemoController:



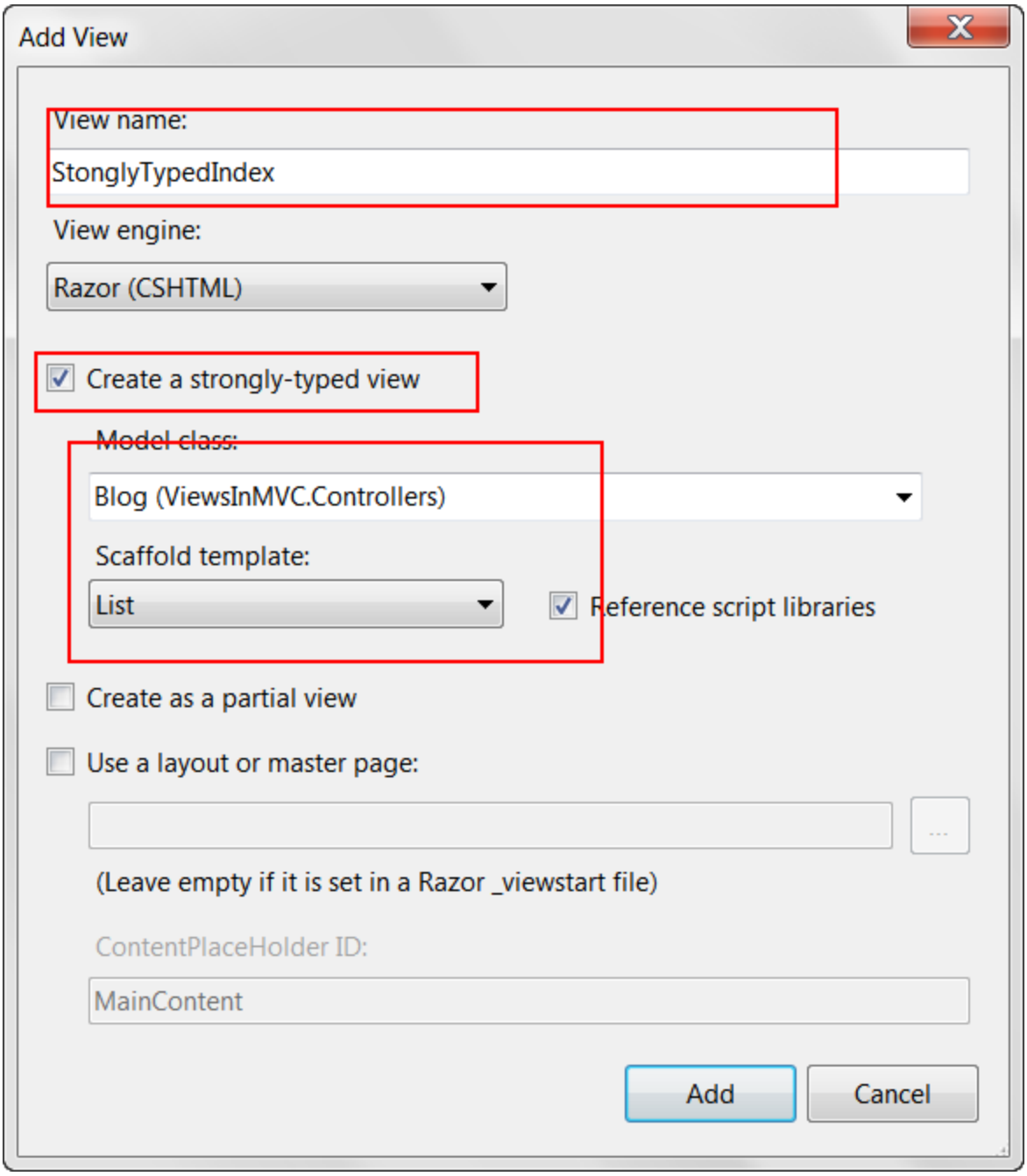
Now copy the following code in the controller file:

```
using System.Collections.Generic;
using System.Web.Mvc;

namespace ViewsInMVC.Controllers
{
    public class ViewDemoController : Controller
    {
        public class Blog
        {
            public string Name;
            public string URL;
        }

        private readonly List topBlogs = new List
        {
```

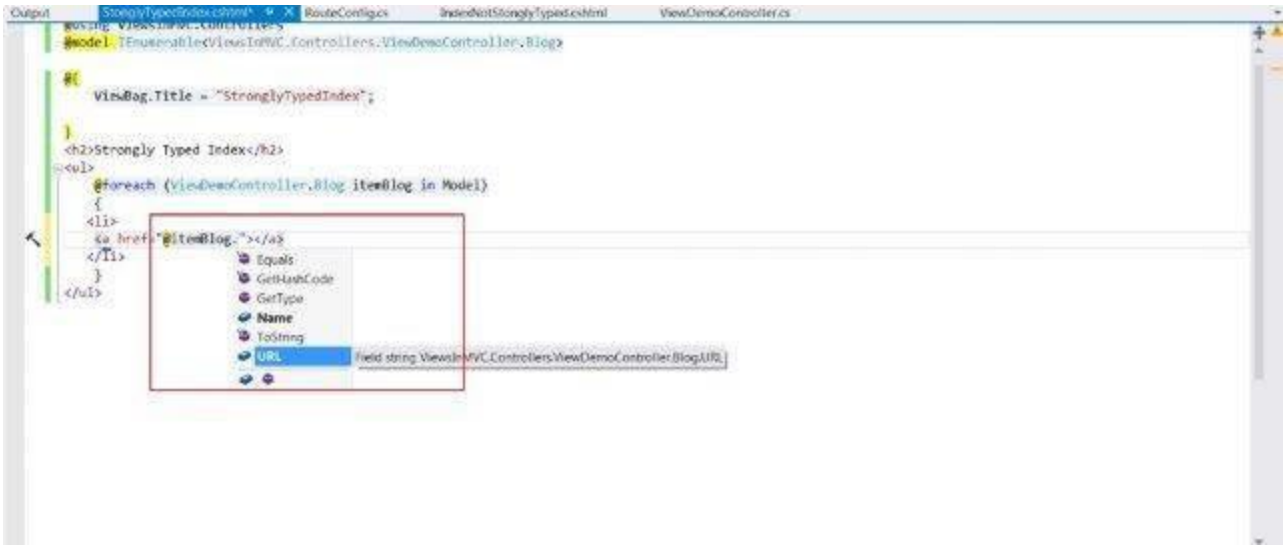
```
            new Blog { Name = "Joe Delage", URL = "http://tutorialspoint/joe/"},
            new Blog {Name = "Mark Dsouza", URL = "http://tutorialspoint/mark"},
            new Blog {Name = "Michael Shawn", URL = "http://tutorialspoint/michael"}
        };

        public ActionResult StonglyTypedIndex()
        {
            return View(topBlogs);
        }

        public ActionResult IndexNotStonglyTyped()
        {
            return View(topBlogs);
        }
    }
}
```

In the above code, we have two action methods defined: StronglyTypedIndex and IndexNotStonglyTyped. We will now add Views for these action methods.

Right click on the StonglyTypedIndex action method and click Add View. In the next window, check the 'Create a strongly-typed view' checkbox. This will also enable the Model Class and Scaffold template options. Select List from Scaffold Template option. Click Add.

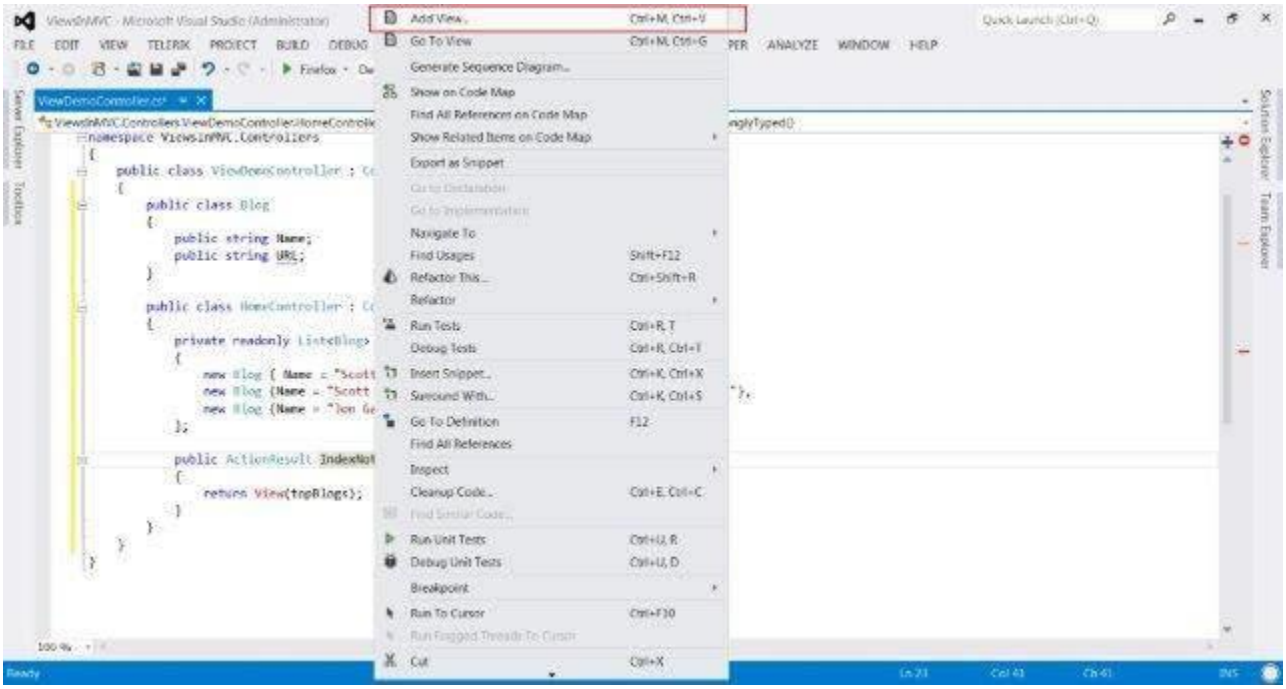


A View file similar to below screenshot will be created. As you can note, it has included the ViewDemoController's Blog model class at the top. You will also be able to use intellisense in your code with this approach.

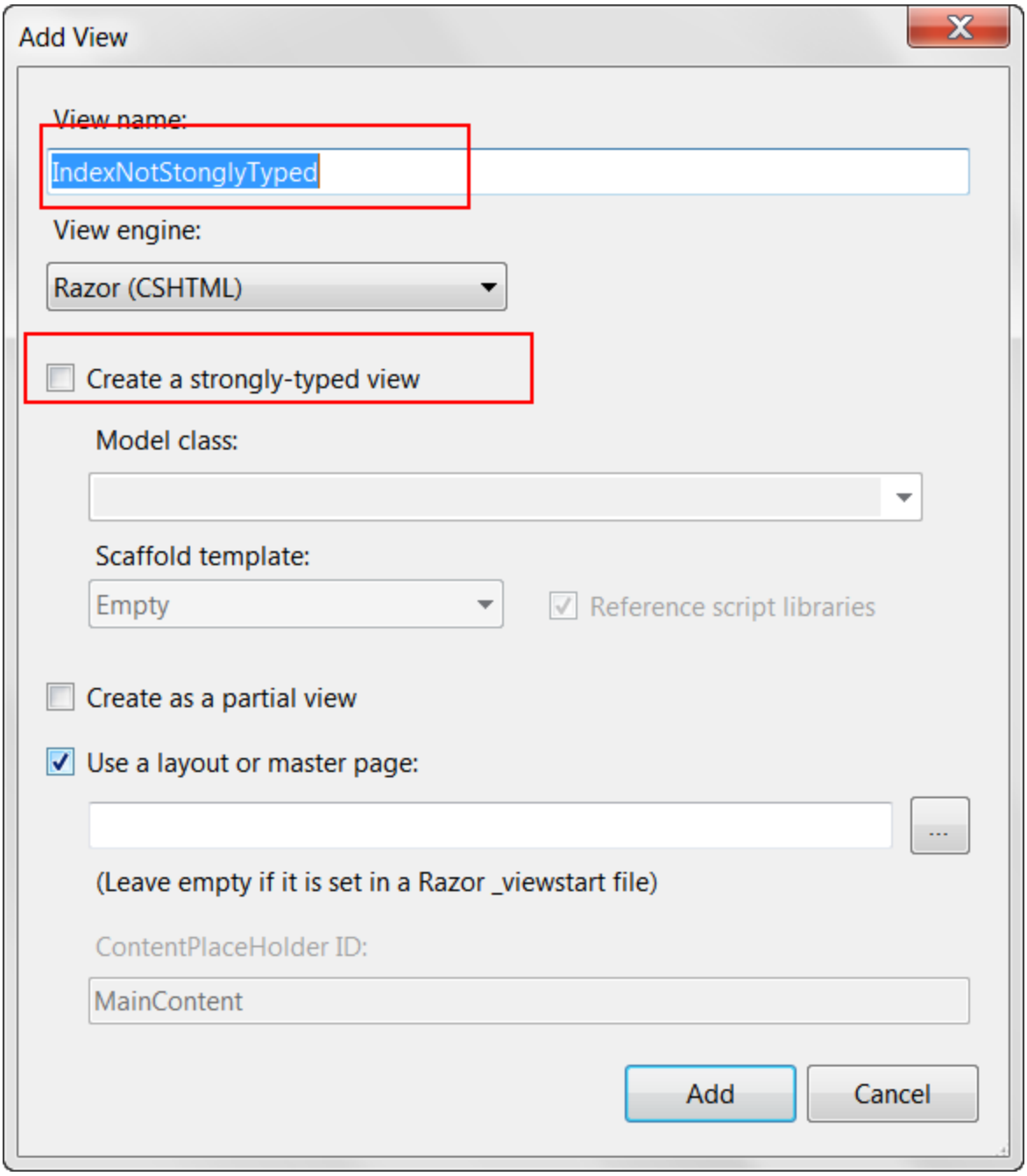


Dynamic Typed Views:

To create dynamic typed views, right click on the IndexNotStonglyTyped action and click Add View.



This time do not select the 'Create a strongly-typed view' checkbox.



The resulting view will have the following code:

```
@model dynamic


@{
    ViewBag.Title = "IndexNotStonglyTyped";
}
```

```
<h2>Index Not Stongly Typed</h2>


<p>
 <ul>
@foreach (var blog in Model) {
   <li>
    <a href="@blog.URL">@blog.Name</a>
   </li>
}
 </ul>
</p>
```

As you can see in the above code, this time it did not add the Blog model to the View as in the previous case. Also, you would not be able to use intellisense this time because this time the binding will be done at run-time.

Strongly typed Views is considered as a better approach since we already know what data is being passed as the Model unlike dynamic typed Views in which the data gets bind at runtime and may lead to runtime errors if something changes in the linked model.