

MVC Framework - Quick Guide

Advertisements



Upgrade
VPN, Ad Blocker, inte



⬅ Previous Page

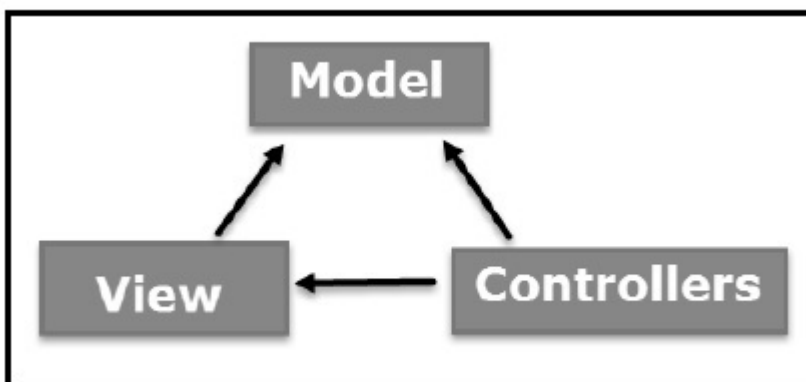
Next Page ➡

MVC Framework - Introduction

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

Following are the components of MVC –



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 will 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 will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will 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). ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with the 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 a template in Visual Studio.

ASP.NET MVC Features

ASP.NET MVC provides the following features –

- Ideal for developing complex but lightweight applications.

- 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.

- 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, 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.

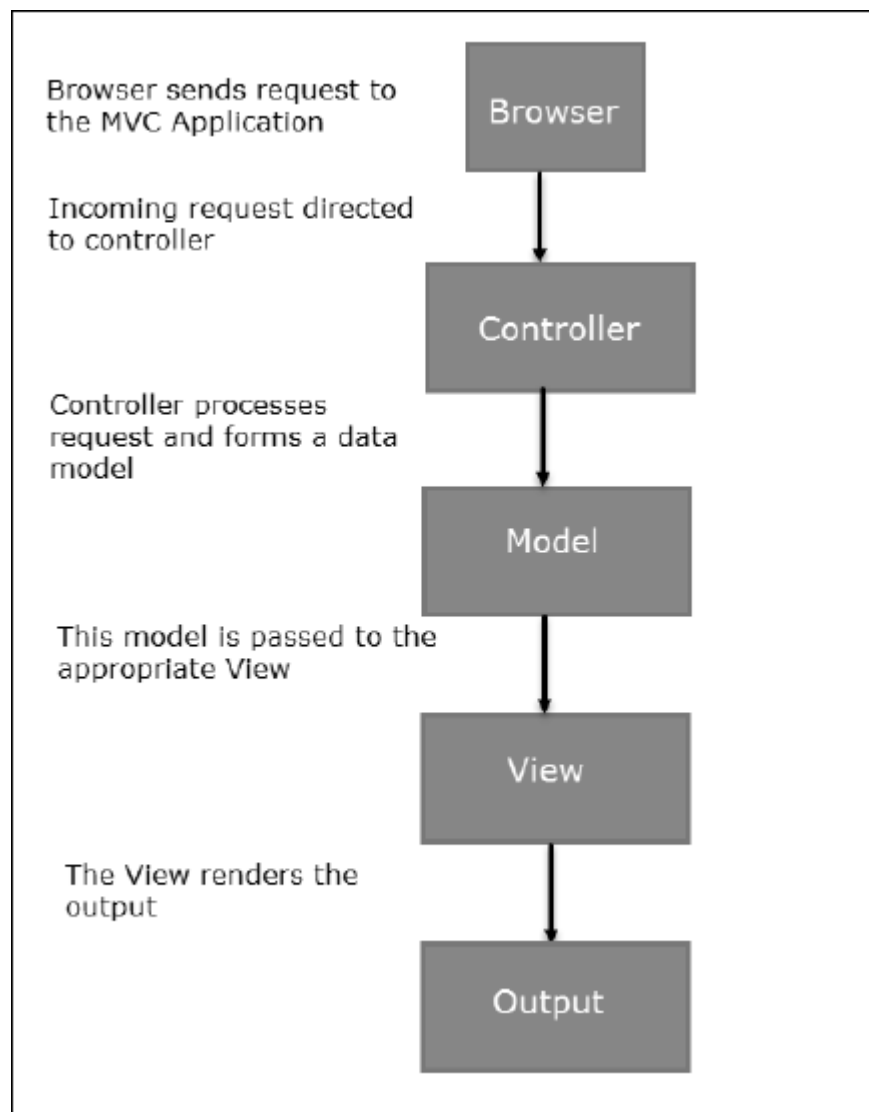
Does not use the concept of View State (which is present in ASP.NET). This helps in building applications, which are lightweight and gives full control to the developers.

Thus, you can consider MVC Framework as a major framework built on top of ASP.NET providing a large set of added functionality focusing 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 take a look at how the execution of an MVC application takes place when there is a certain request from the client. The following diagram illustrates the flow.

MVC Flow Diagram



Flow Steps

Step 1 – The client browser sends request to the MVC Application.

Step 2 – Global.ascx receives this request and performs routing based on the URL of the incoming request using the RouteTable, RouteData, UrlRoutingModule and MvcRouteHandler objects.

Step 3 – This routing operation calls the appropriate controller and executes it using the IControllerFactory object and MvcHandler object's Execute method.

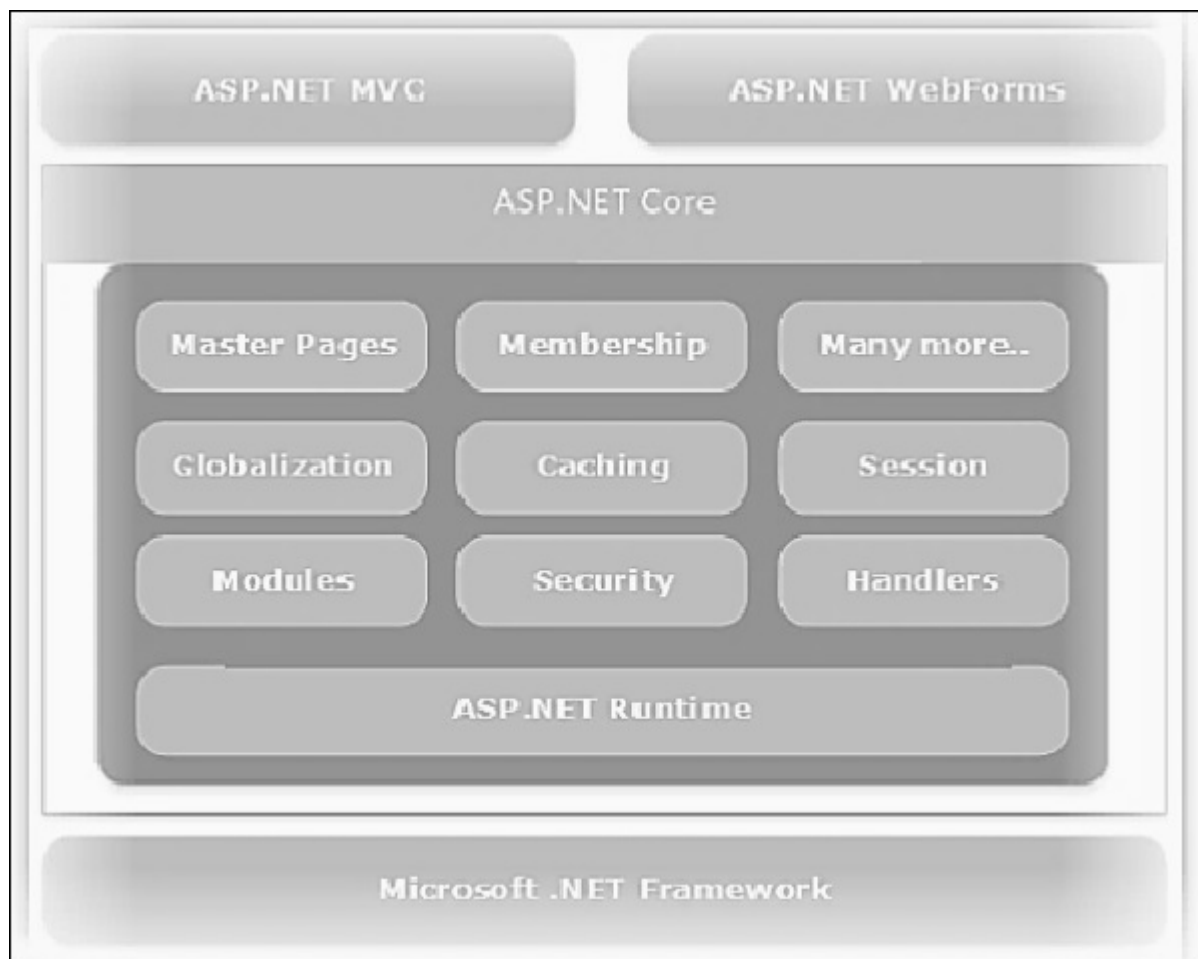
Step 4 – The Controller processes the data using Model and invokes the appropriate method using ControllerActionInvoker object

Step 5 – 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 yet different models of development, depending on the requirement of the application and other factors. At a high level, you can consider that MVC is an 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

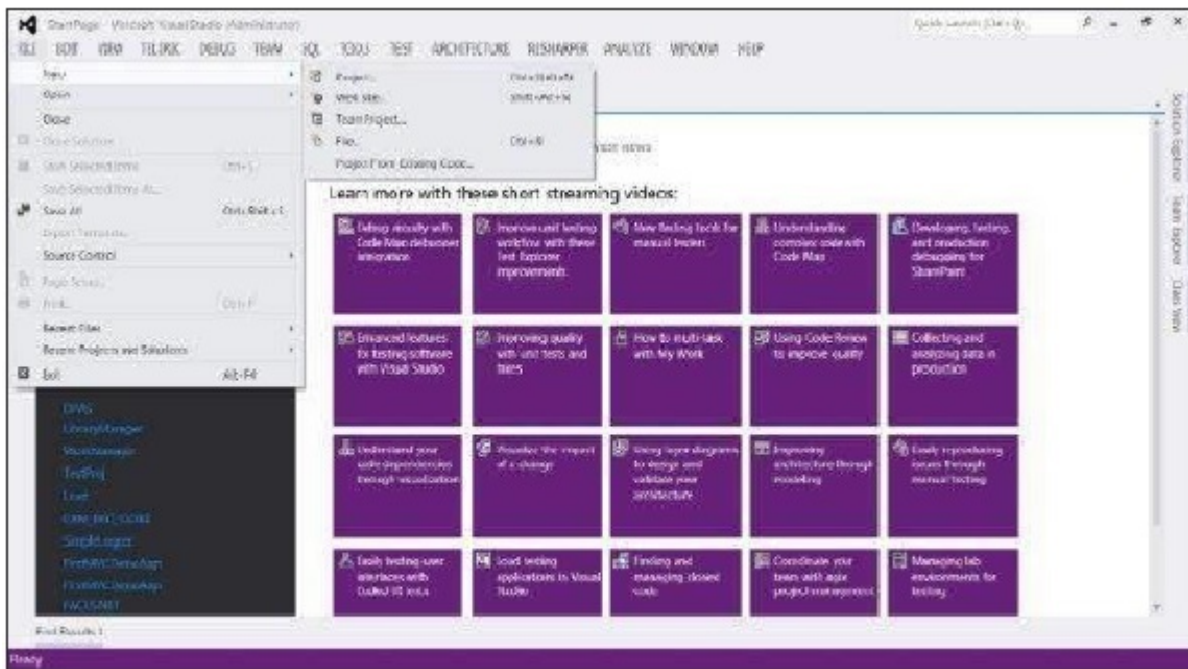
Comparison Factors	ASP.NET Web Forms	ASP.NET MVC
Rendering Approach	Follows Page Control pattern approach for rendering layout.	Front Controller Approach is used.
Separation of Concern	No separation of concerns and all Web Forms are tightly coupled.	Very clean separation of concerns.
Automated Testing	Automated testing is really difficult.	TDD is Quite 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.

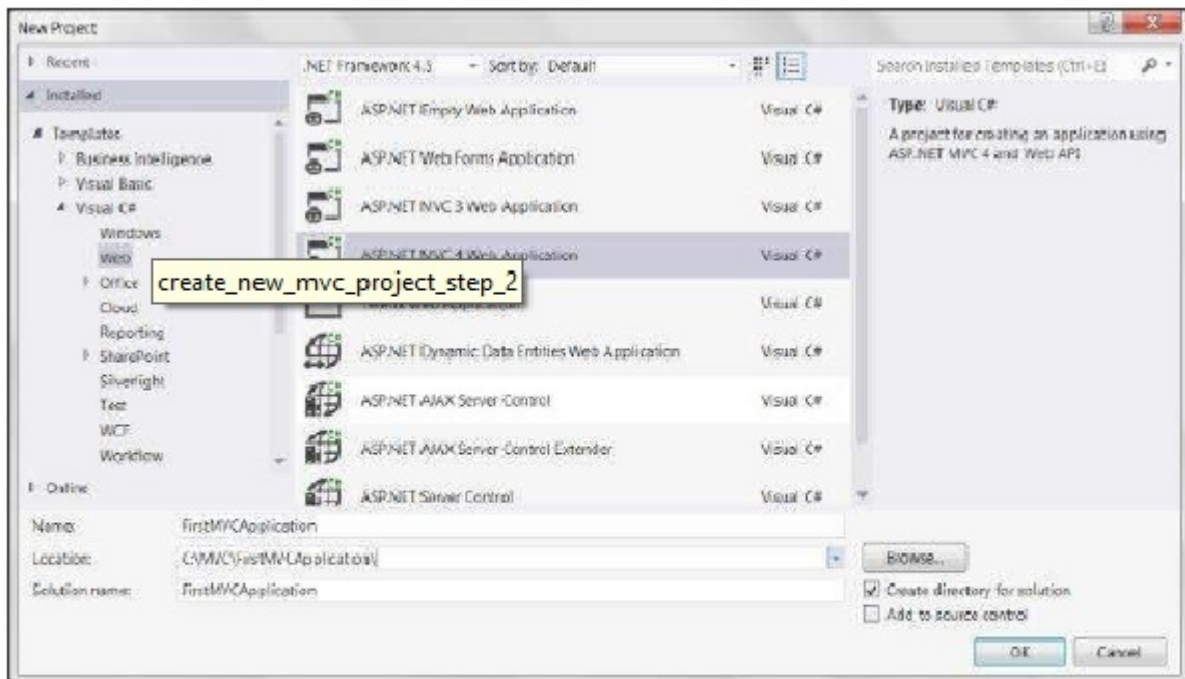
MVC Framework - First Application

Let us jump in and create our first MVC application using Views and Controllers. Once we have a small hands-on experience on how a basic MVC application works, we will learn all the individual components and concepts in the coming chapters.

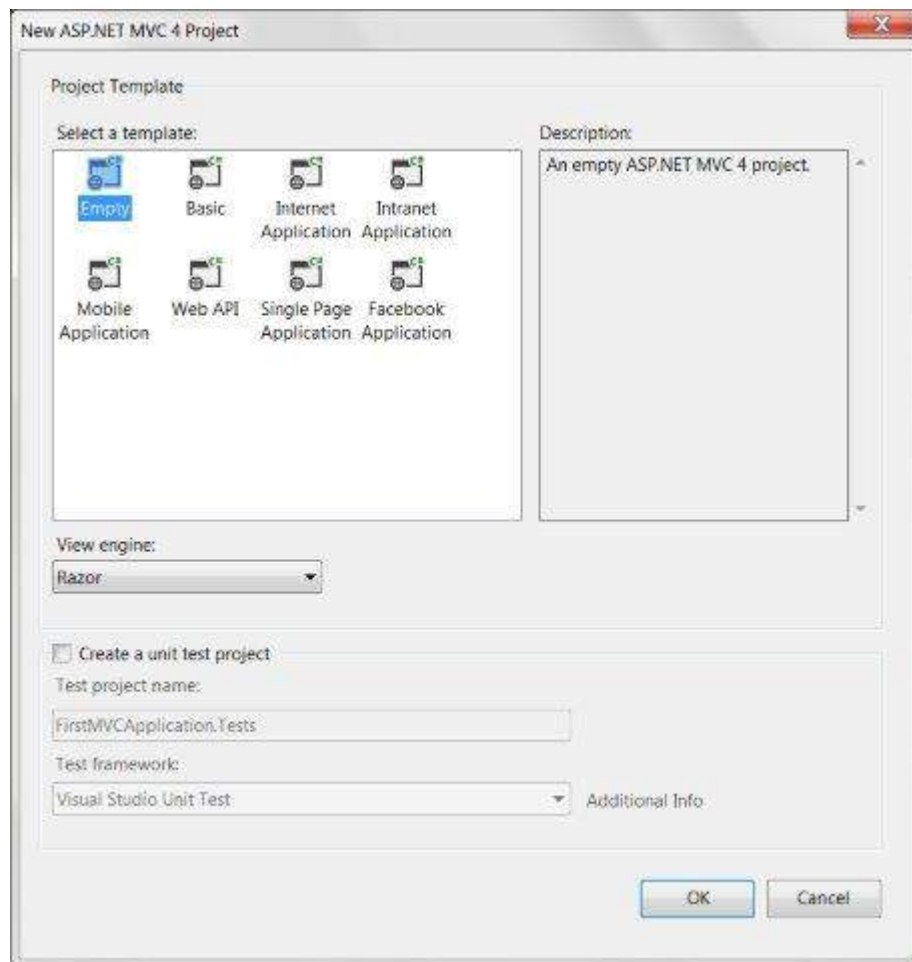
Create First MVC Application

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

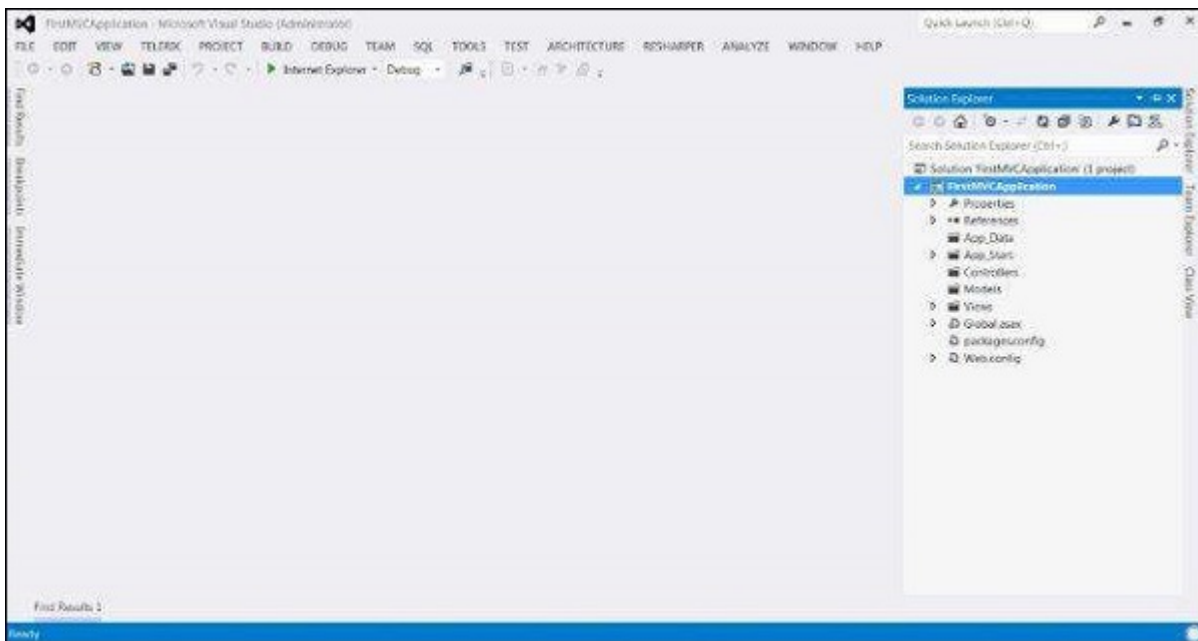




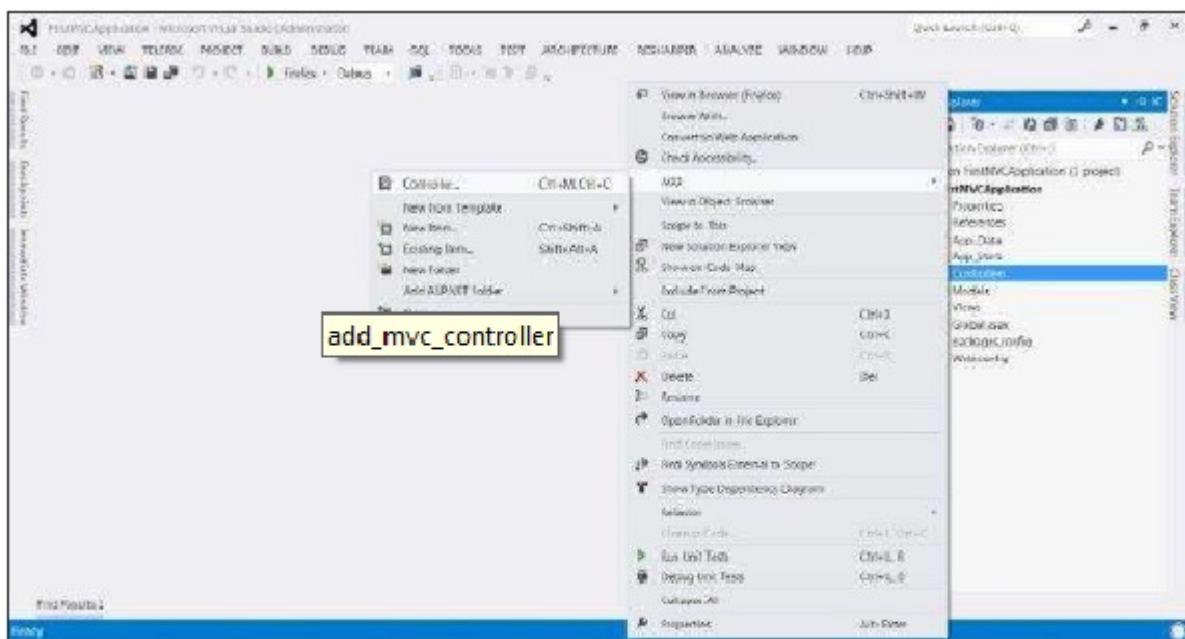
Step 2 – This will open the Project Template option. Select Empty template and View Engine as Razor. Click OK.

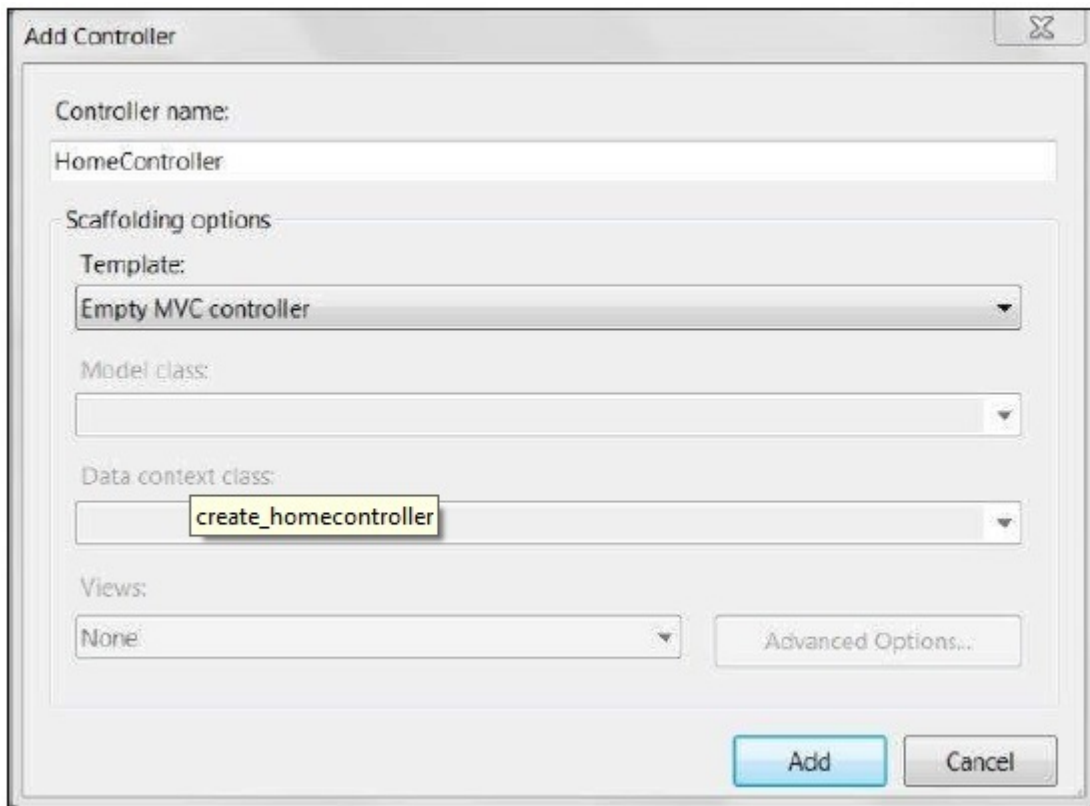


Now, Visual Studio will create our first MVC project as shown in the following screenshot.



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 HomeController 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;

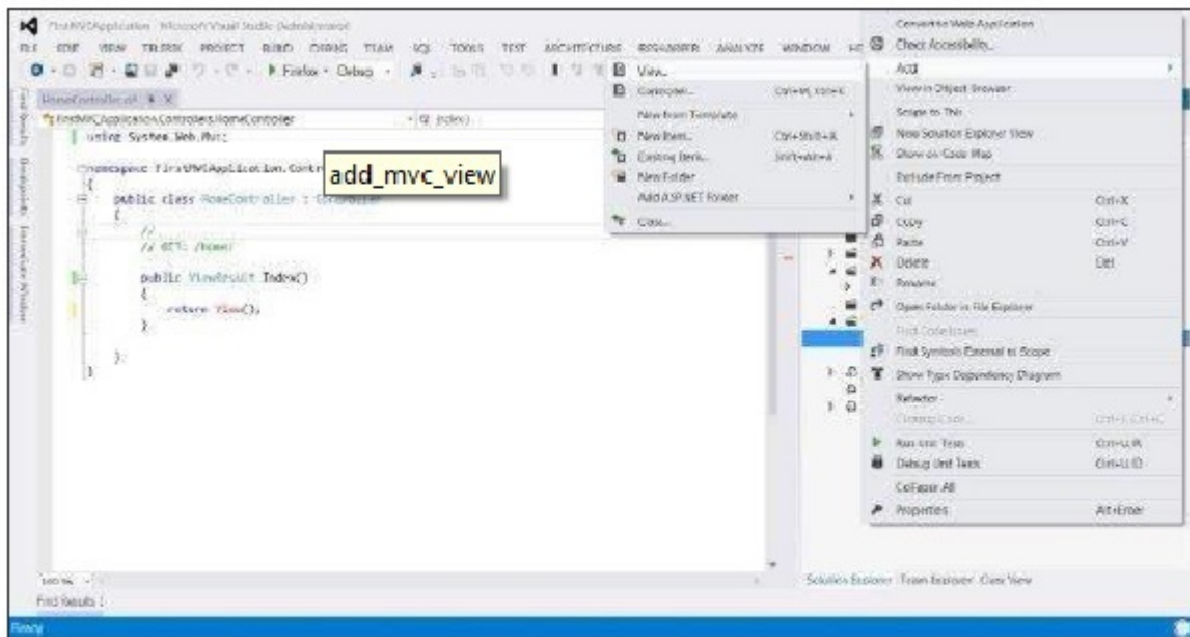
namespace FirstMVCApplication.Controllers {

    public class HomeController : Controller {

        public ActionResult 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, rightclick view folder and click Add → View.



Step 5 – Name the new View as Index and View Engine as Razor (SHTML). Click Add.

Add View

View name:
Index

View engine:
Razor (C#HTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:
Empty

☒ Reference script libraries

☐ Create as a partial view

☐ Use a layout or master page:
...
(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:
MainContent

Add Cancel

This will add a new **cshhtml** file inside Views/Home folder with the following code –

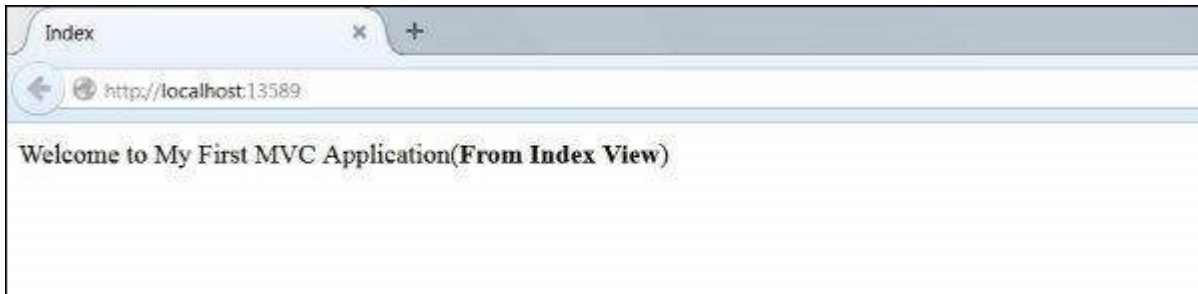
```
@{  
    Layout = null;  
}  
  
<html>  
    <head>  
        <meta name = "viewport" content = "width = device-width" />  
        <title>Index</title>  
    </head>  
  
    <body>  
        <div>  
  
        </div>
```

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

Step 6 – 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 7 – 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 produces the output.



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

Step 8 – 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 ActionResult 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 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.

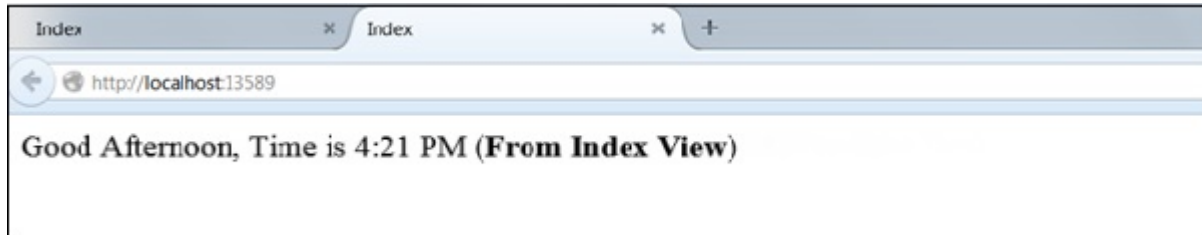
Step 9 – Open the Index.cshtml and copy the following code in the body section.

```
<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).

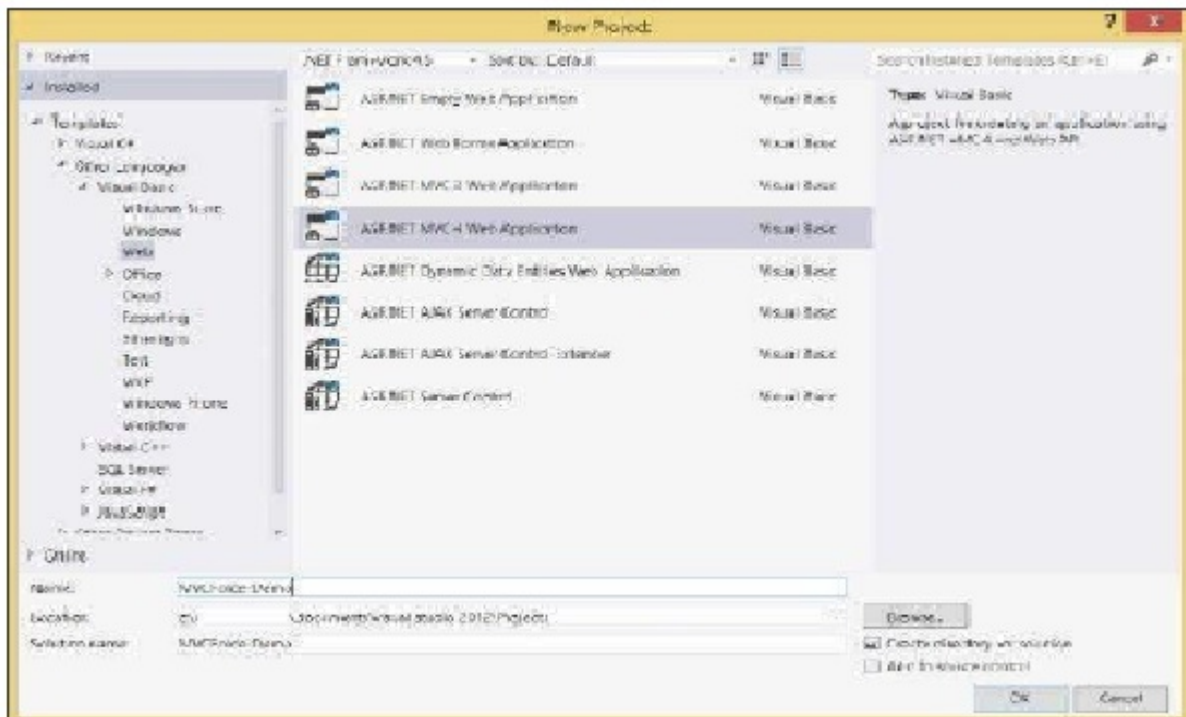
Step 10 – Now run the application again. This time our code will run the Controller first, set the ViewBag and then render it using the View code. Following will be the output.



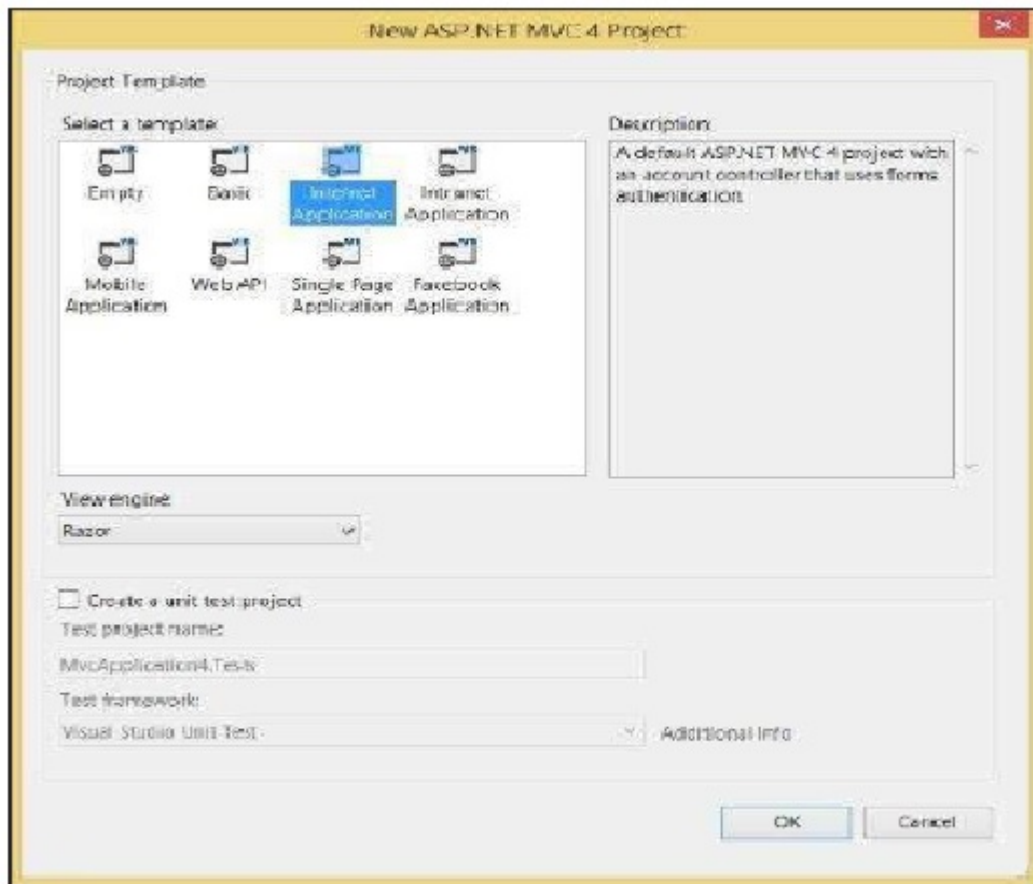
MVC Framework - Folders

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

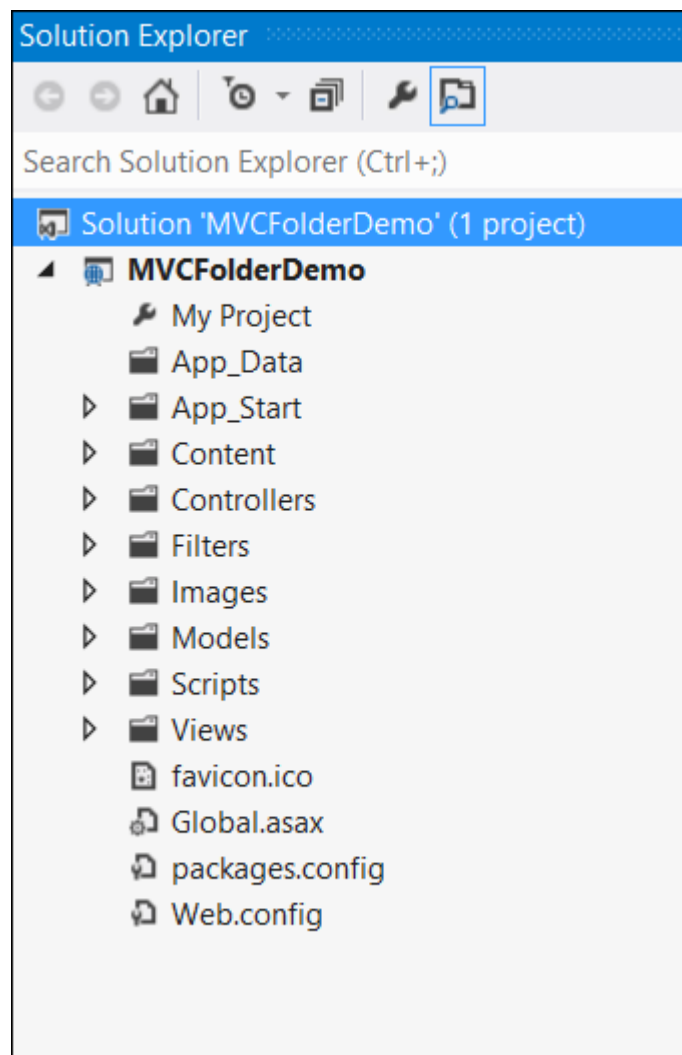
In your Visual Studio, open File → New → Project and select ASP.NET MVC Application. Name it as **MVCFolderDemo**.



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 in the following screenshot.

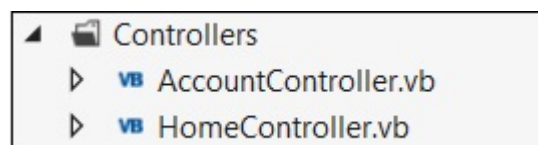


Note – Files present in this project are coming out of the default template that we have selected. These may change slightly as per different versions.

Controllers Folder

This folder will contain all the Controller classes. MVC requires the name of all the controller files to end with Controller.

In our example, the Controllers folder contains two class files: `AccountController` and `HomeController`.

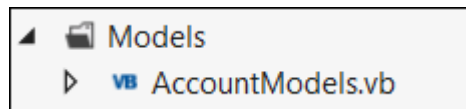


Models Folder

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

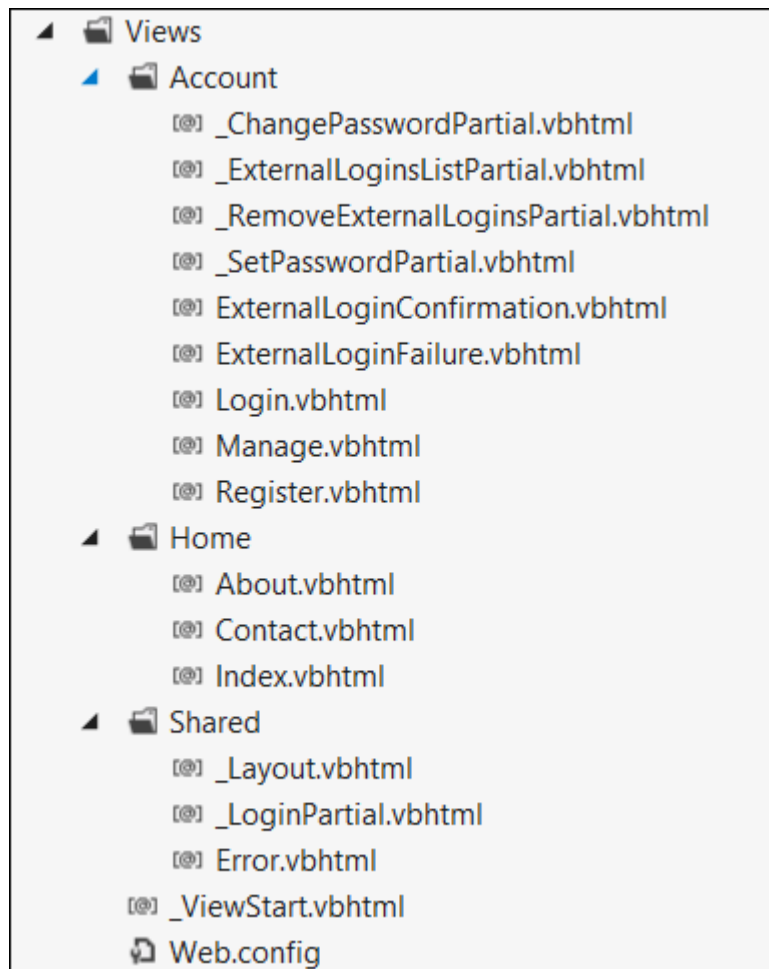
example.



Views Folder

This folder stores the HTML files related to application display and user interface. It contains one folder for each controller.

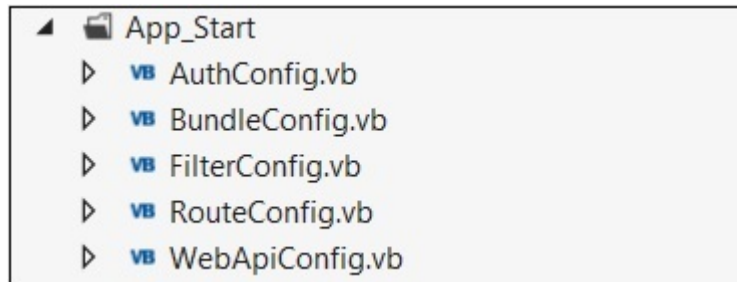
In our example, you will see three sub-folders under Views, namely Account, Home and Shared which contains html files specific to that view area.



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

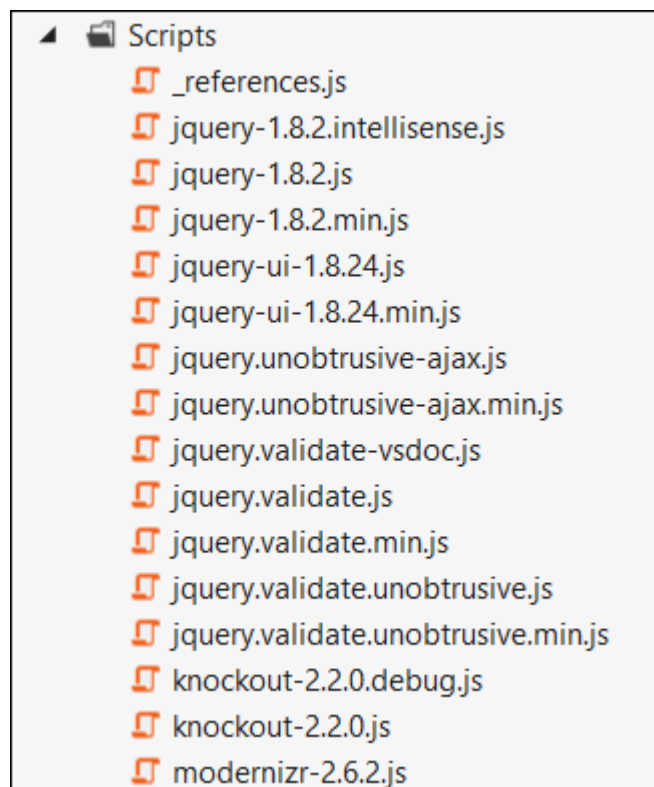
This folder contains all the static files, such as css, images, icons, etc.

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.



MVC Framework - Models

The component '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 the coming chapters. Thus in this chapter, we will try the other option, i.e. generating from the database so that you have hands-on experience on both the methods.

Create 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,
```

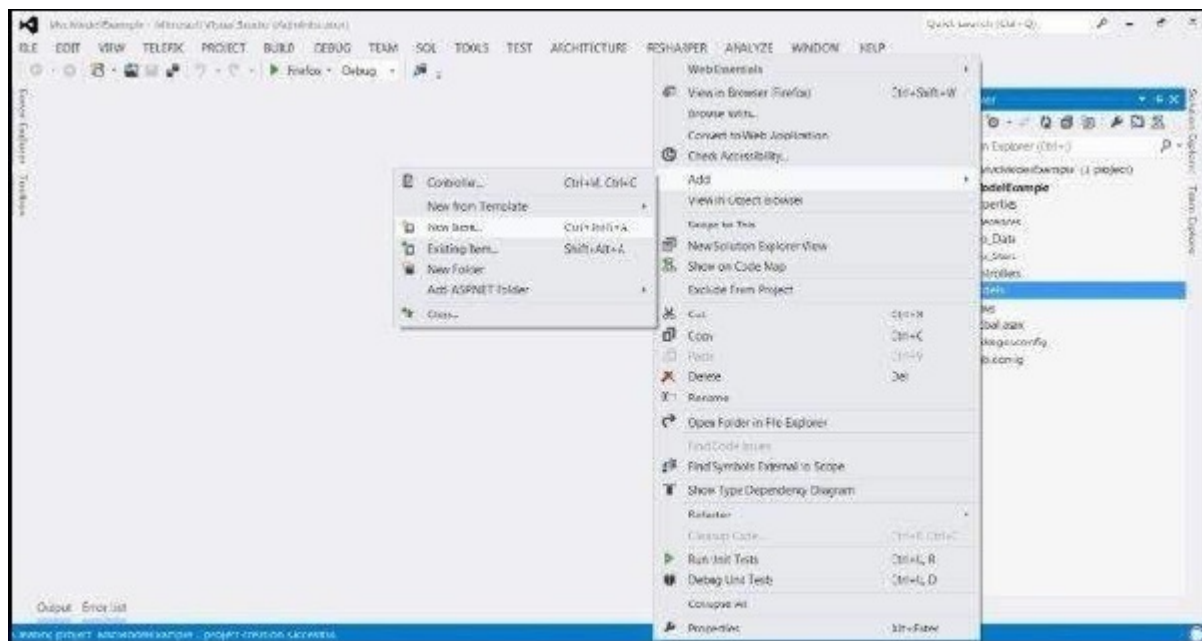
```

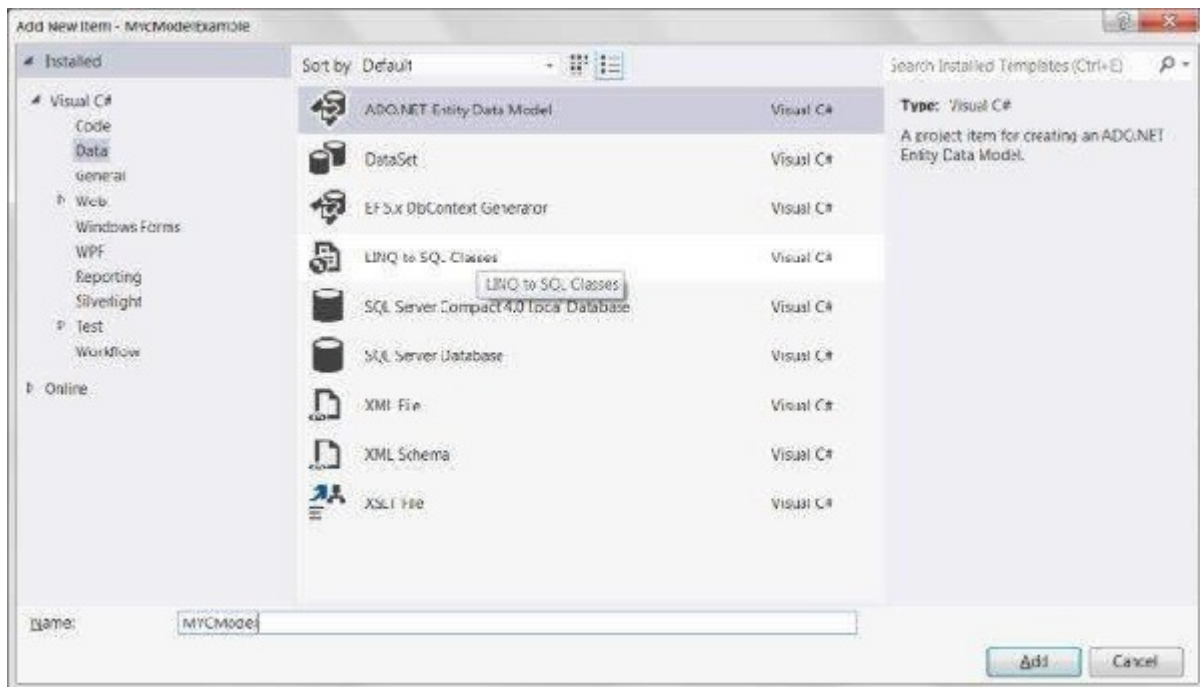
[Grade]          DECIMAL(3,2) NULL,
[CourseID]       INT NOT NULL,
[StudentID]      INT NOT NULL,
PRIMARY KEY CLUSTERED ([EnrollmentID] ASC),
    CONSTRAINT [FK_dbo.Enrollment_dbo.Course_CourseID] FOREIGN KEY ([CourseID])
REFERENCES [dbo].[Course]([CourseID]) ON DELETE CASCADE,
    CONSTRAINT [FK_dbo.Enrollment_dbo.Student_StudentID]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

```

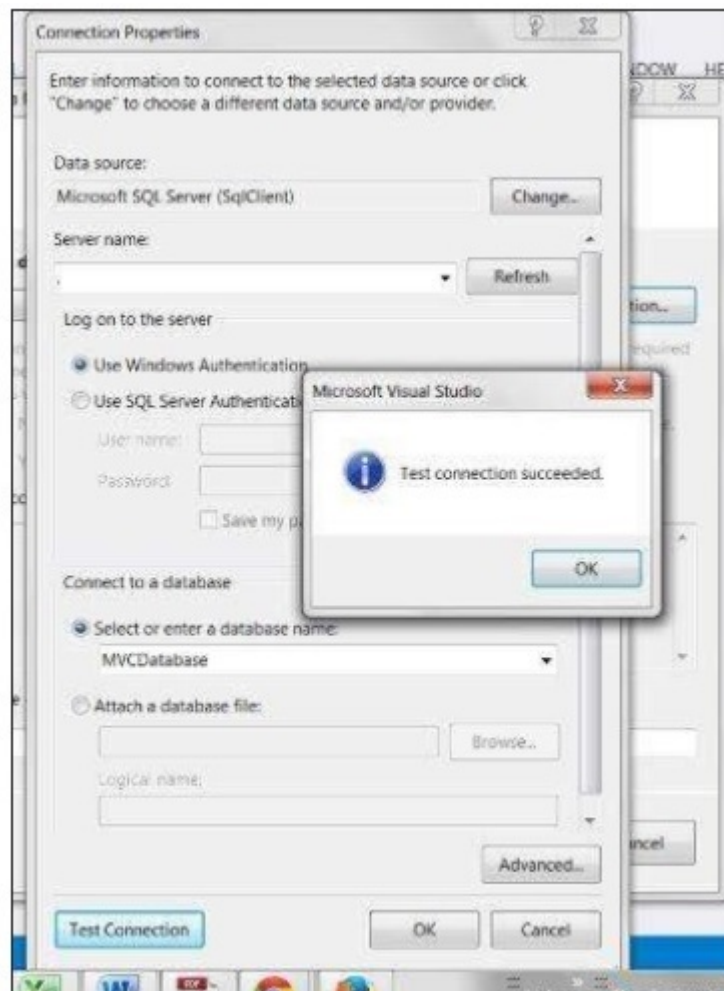
Generate Models Using Database Entities

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

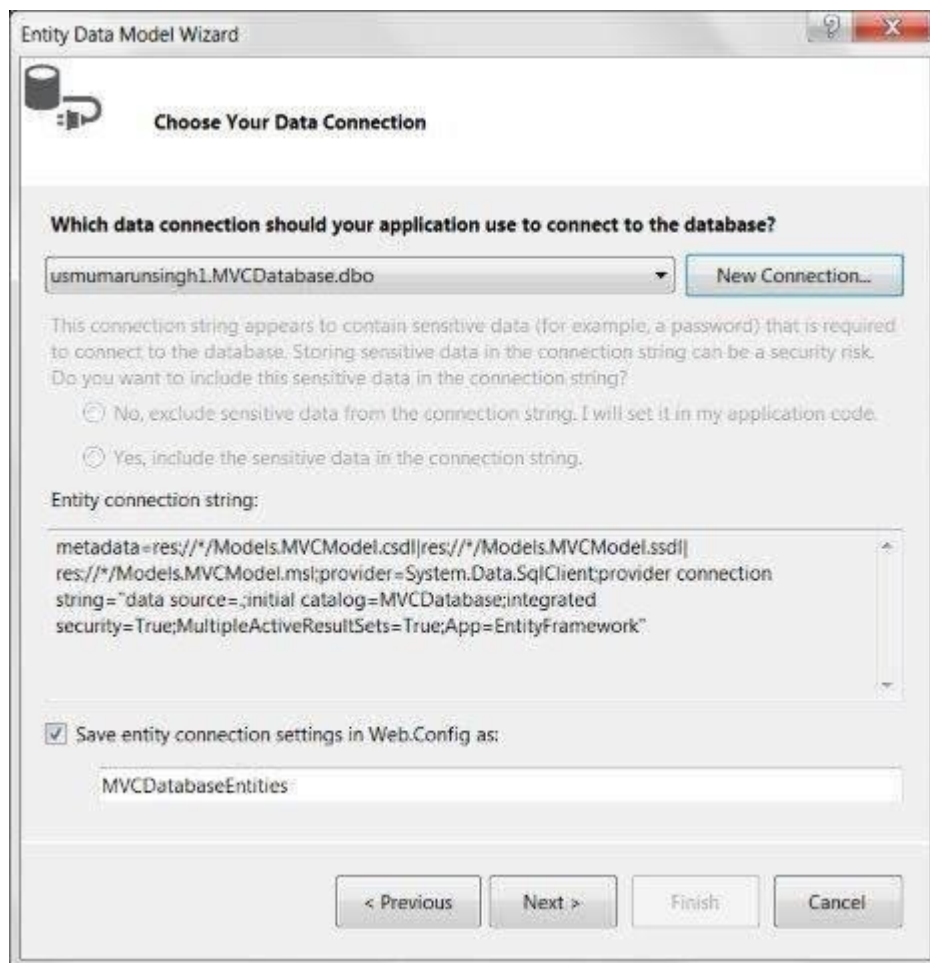




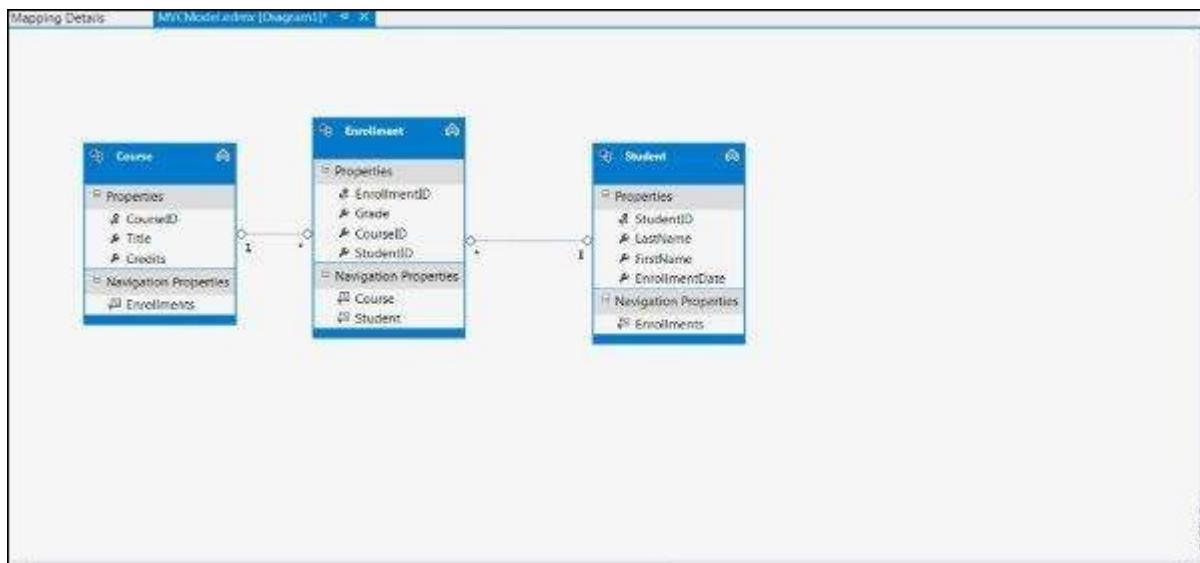
In the next wizard, choose Generate From Database and click Next. Set the Connection to your SQL database.



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



Select Tables, Views, and Stored Procedures and Functions. Click Finish. You will see the Model View created as shown in the following screenshot.



The above operations would automatically create a Model file for all the database entities. For example, 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 application, a controller is responsible for returning the response to that request. The controller can perform one or more actions. The controller action can return different types of action results to a particular request.

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

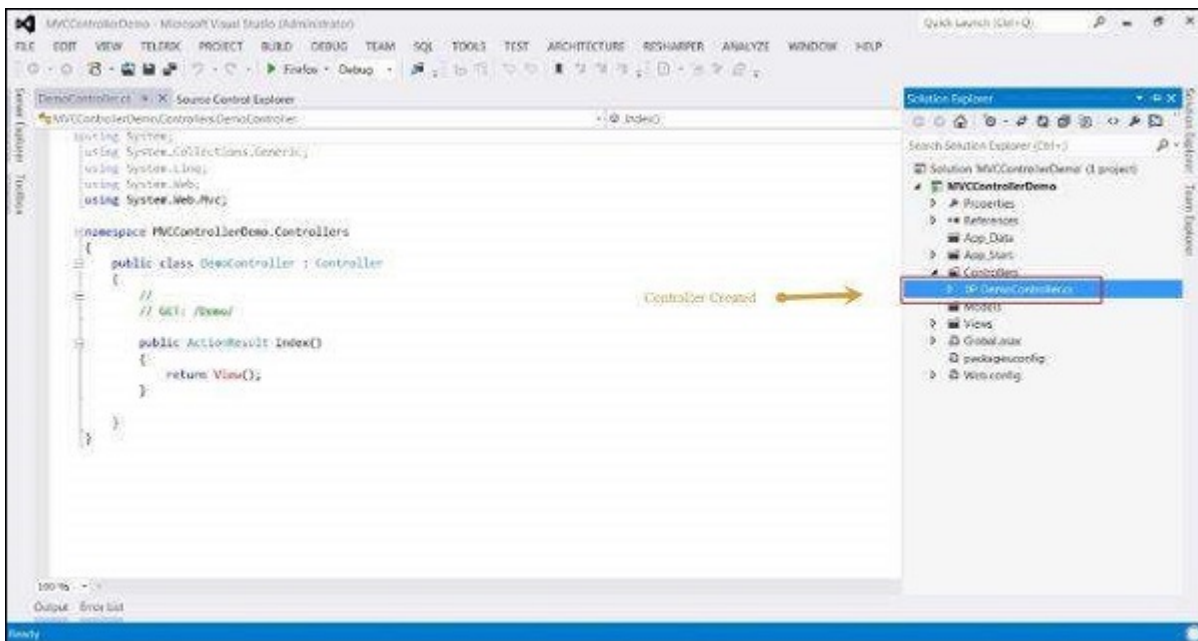
Create a Controller

To create a Controller –

Step 1 – Create an MVC Empty Application and then right-click on the Controller folder in your MVC application.

Step 2 – Select the menu option Add → Controller. After selection, the Add Controller dialog is displayed. Name the Controller as **DemoController**.

A Controller class file will be created as shown in the following screenshot.

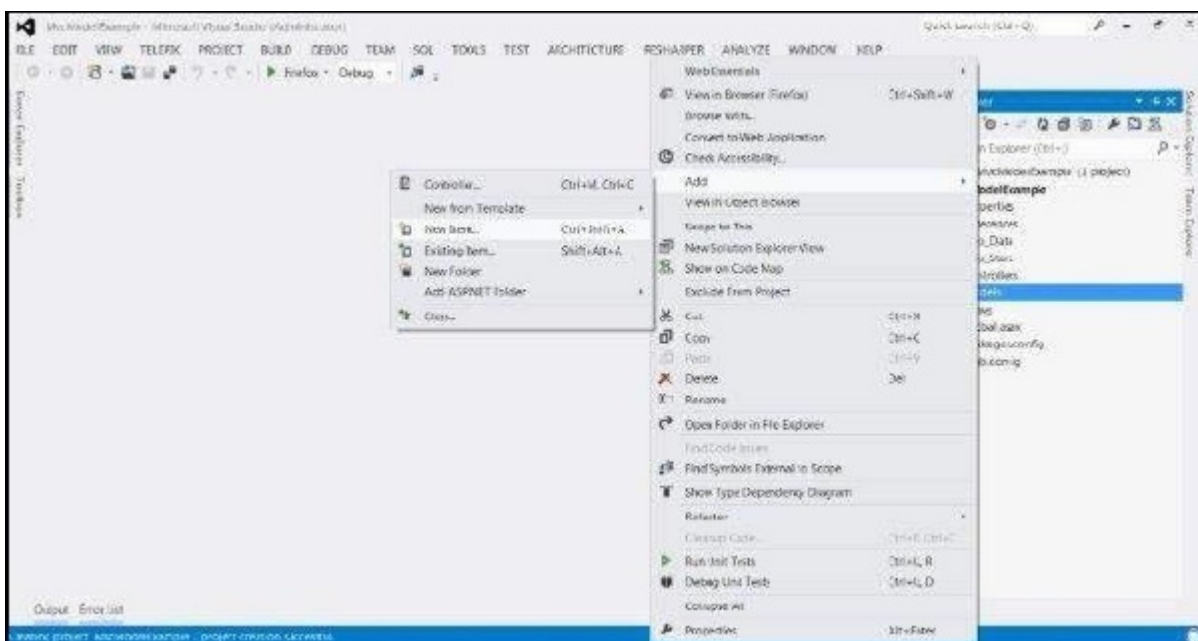


Create a Controller with IController

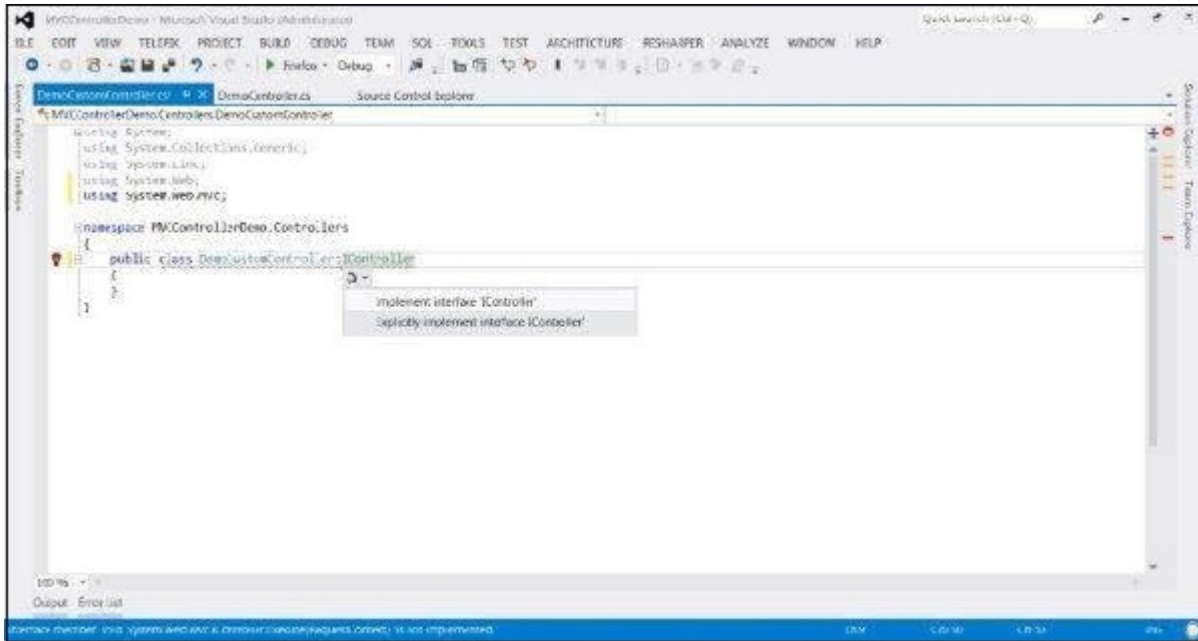
In the MVC Framework, controller classes must implement the `IController` interface from the `System.Web.Mvc` namespace.

```
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.



Step 1 – Add a new class file and name it as DemoCustomController. Now modify this class to inherit IController interface.



Step 2 – 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));  
    }  
}
```

Step 3 – Run the application and you will receive the following output.



MVC Framework - Views

As seen in the initial introductory chapters, View is the component involved with the application's User Interface. These Views are generally bind from the model data and have extensions such as html, aspx, cshtml, vbhtml, etc. In our First MVC Application, we had used Views with Controller to display data to the final user. For rendering these static and dynamic content to the browser, MVC Framework utilizes View Engines. View Engines are

basically markup syntax implementation, which are responsible for rendering the final HTML to the browser.

MVC Framework comes with two built-in view engines –

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.

ASPX Engine – ASPX or the Web Forms engine is the default view engine that is included in the MVC Framework since the beginning. Writing a code with this engine is similar to writing a 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 an 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 –

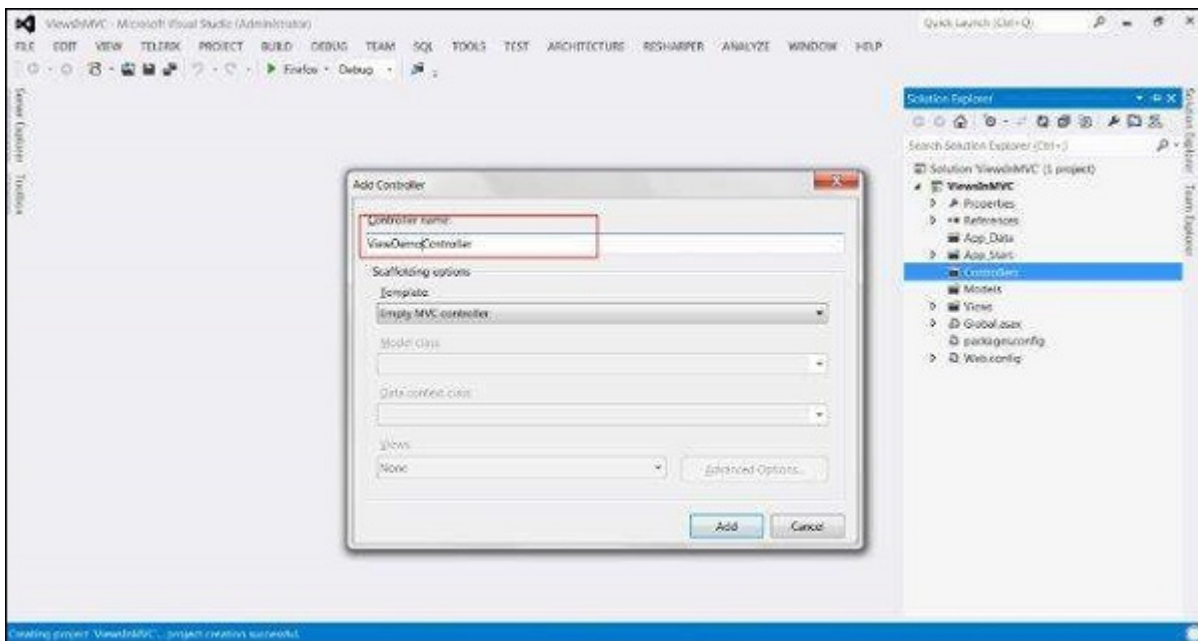
Strongly typed

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;
        }

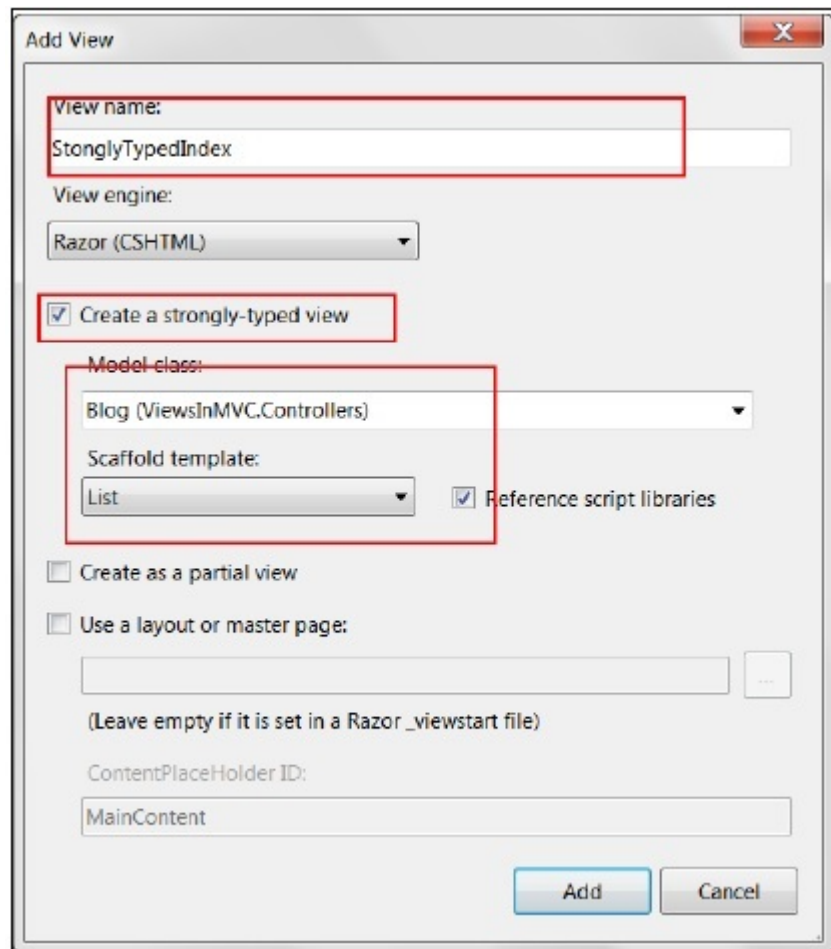
        private readonly List topBlogs = new List {
            new Blog { Name = "Joe Delage", URL = "http://tutorialspoint/joe/"},
            new Blog {Name = "Mark Dsouza", URL = "http://tutorialspoint/mark"},
            new Blog {Name = "Michael Shawn", URL = "http://tutorialspoint/michael"}
        };

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

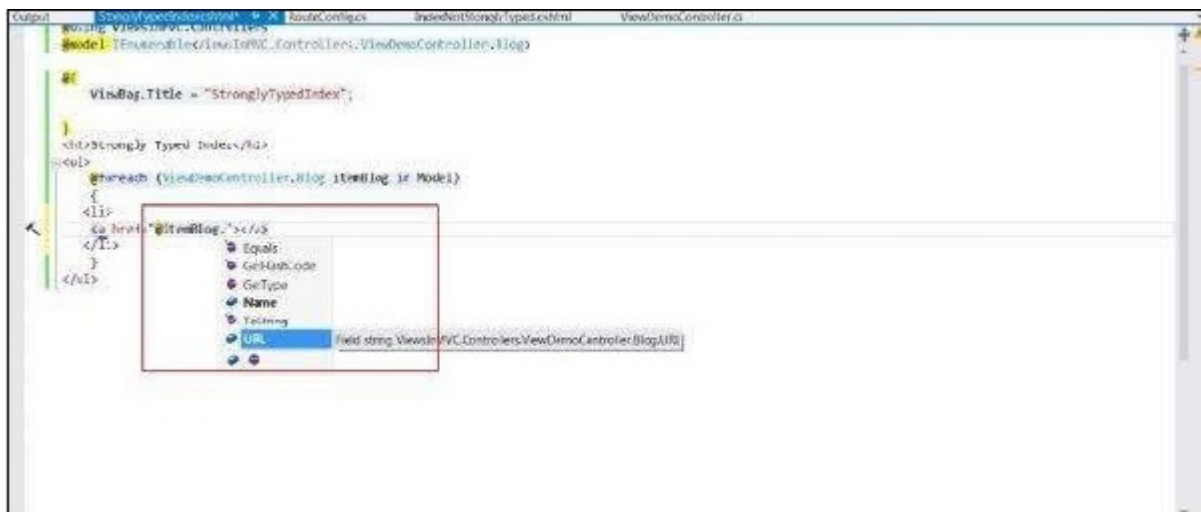
        public ActionResult IndexNotStonglyTyped() {
            return View(topBlogs);
        }
    }
}
```

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 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.

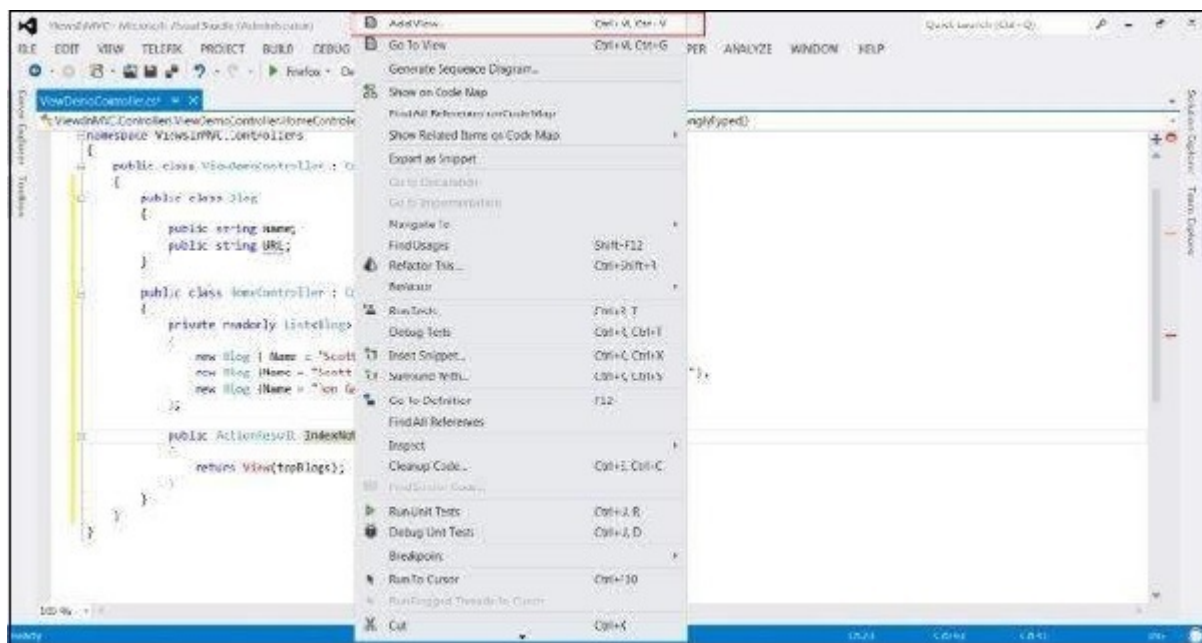


A View file similar to the following screenshot will be created. As you can note, it has included the ViewDemoController's Blog model class at the top. You will also be able to use IntelliSense in your code with this approach.



Dynamic Typed Views

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



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

Add View

View name:

View engine:
Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:
Empty ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

Add Cancel

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>
```



```
</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.

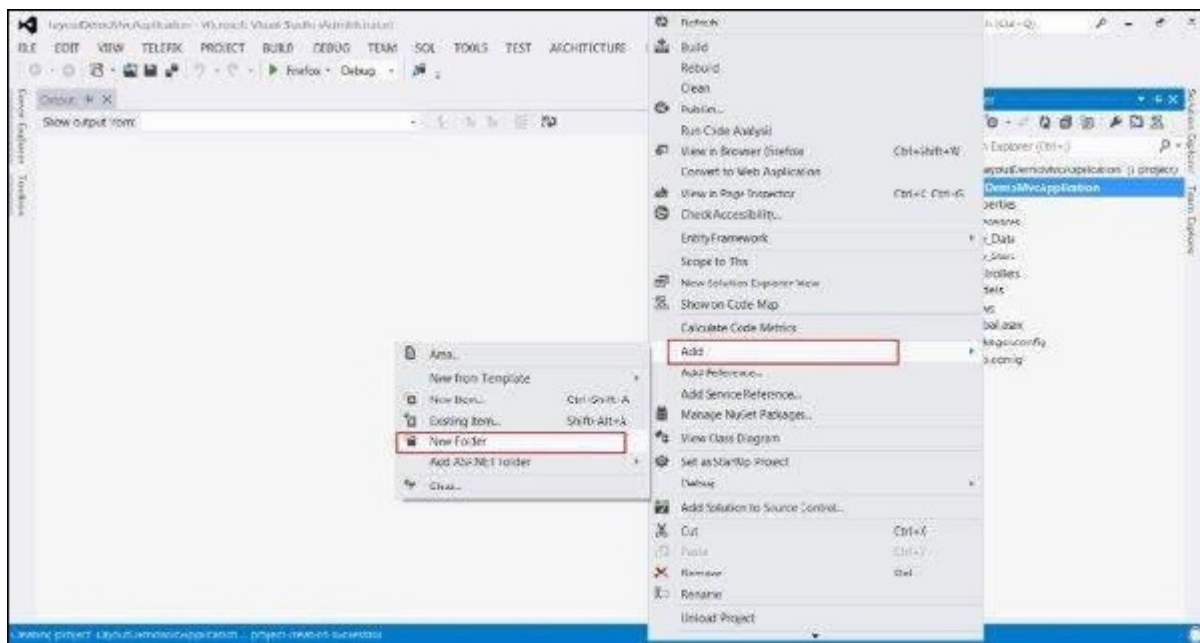
Strongly typed Views is considered as a better approach since we already know what data is being passed as the Model unlike dynamic typed Views in which the data gets bind at runtime and may lead to runtime errors, if something changes in the linked model.

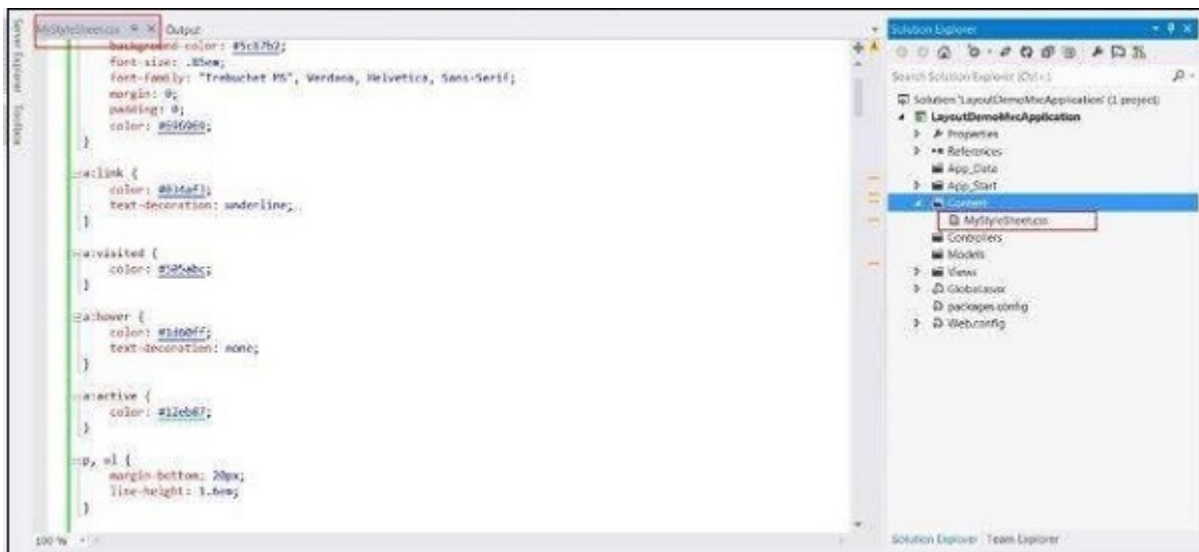
MVC Framework - Layouts

Layouts are used in MVC to provide a consistent look and feel on all the pages of our application. It is the same as defining the Master Pages but MVC provides some more functionalities.

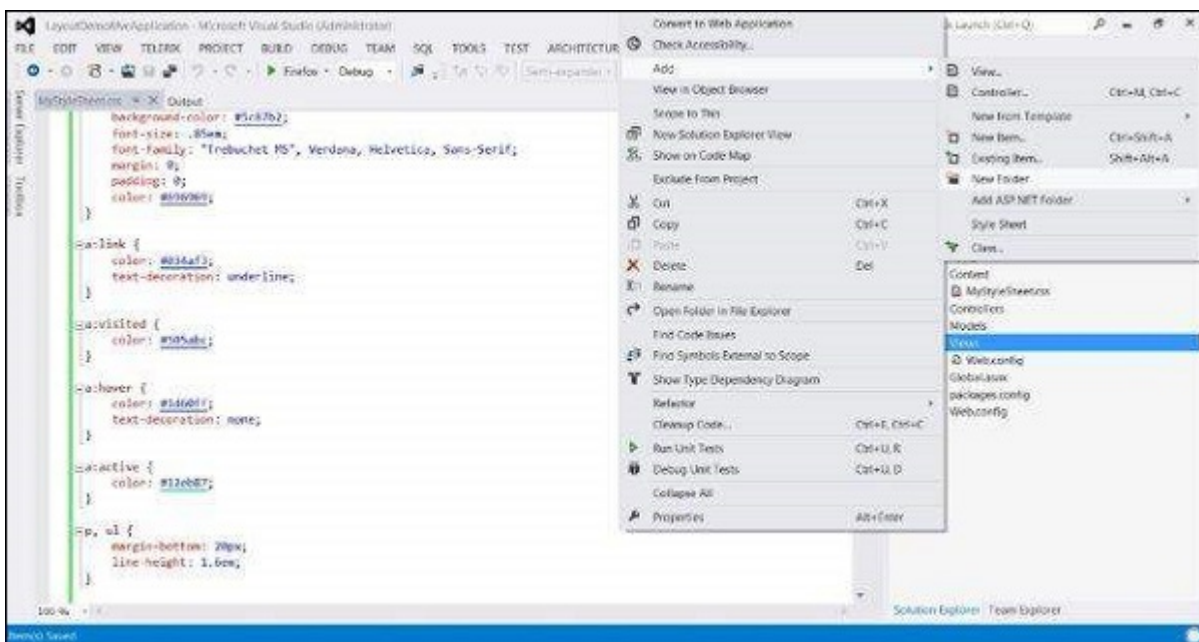
Create MVC Layouts

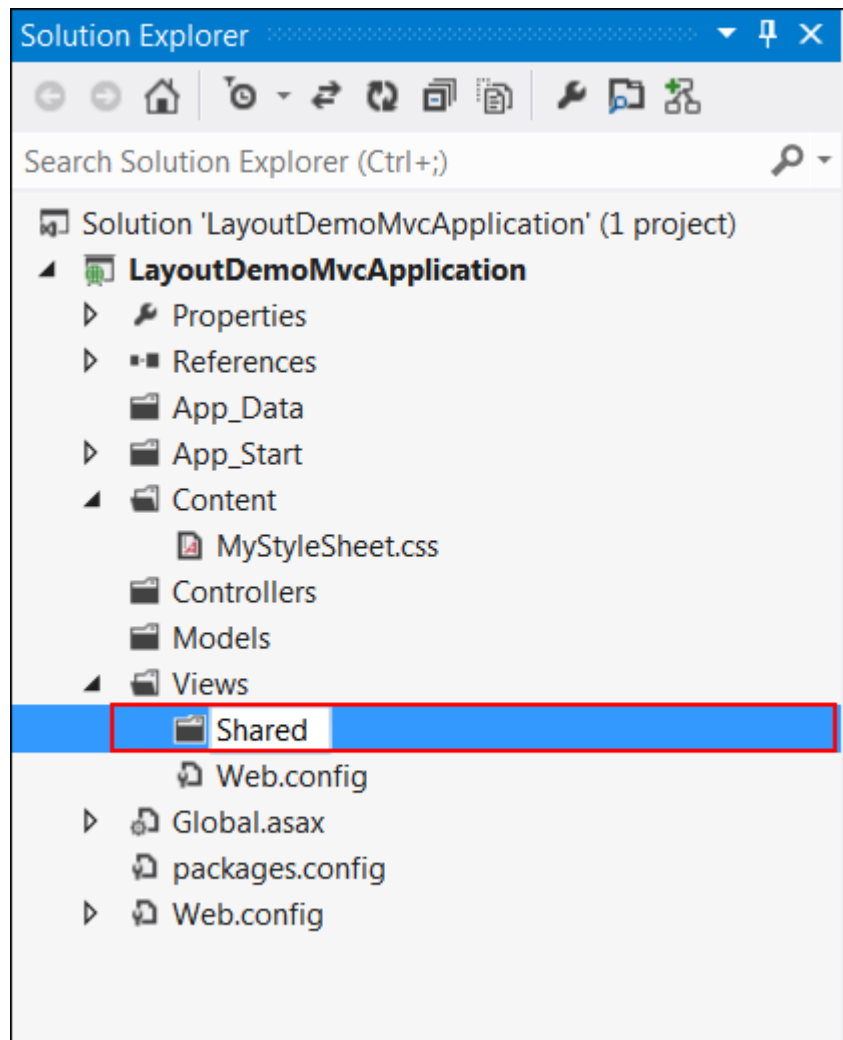
Step 1 – Create a sample MVC application with Internet application as Template and create a Content folder in the root directory of the web application.



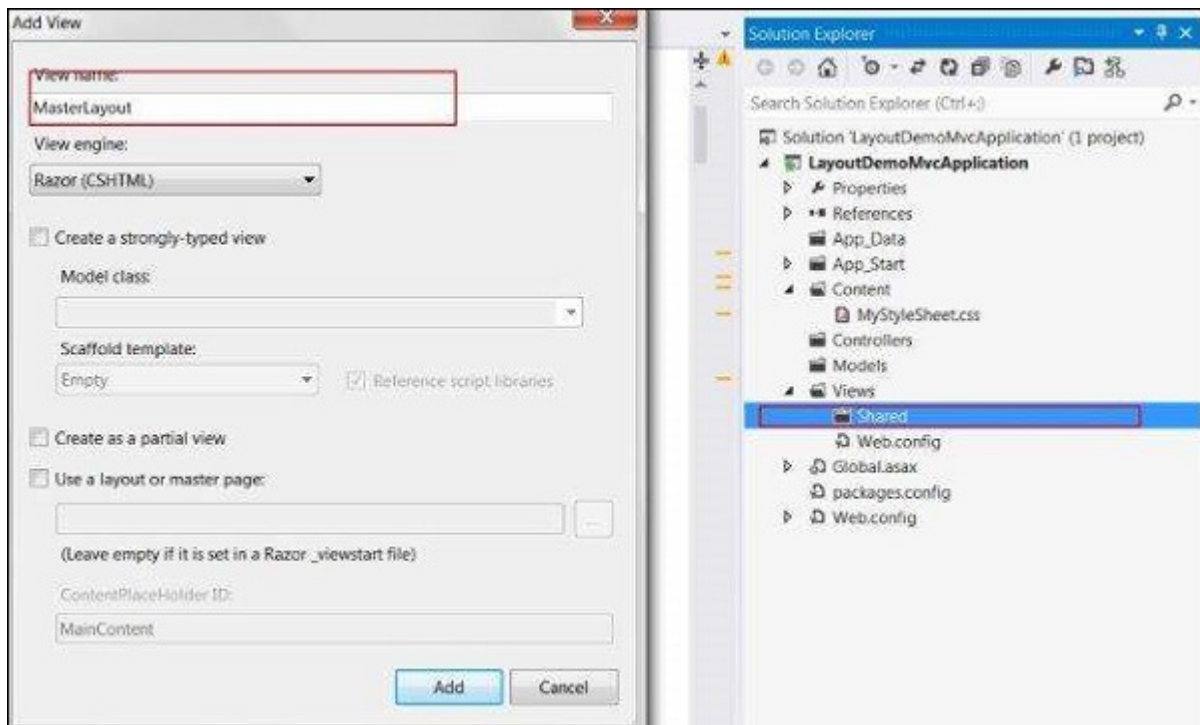


Step 3 – Create a Shared folder under the View folder.





Step 4 – Create a MasterLayout.cshtml file under the Shared folder. The file MasterLayout.cshtml represents the layout of each page in the application. Right-click on the Shared folder in the Solution Explorer, then go to Add item and click View. Copy the following layout code.



Layout Code

```
<!DOCTYPE html>

<html lang = "en">
  <head>
    <meta charset = "utf-8" />
    <title>@ViewBag.Title - Tutorial Point</title>
    <link href = "~/favicon.ico" rel = "shortcut icon" type = "image/x-icon" />
    <link rel = "stylesheet" href = "@Url.Content("~/Content/MyStyleSheet.css")" />
  </head>

  <body>
    <header>

      <div class = "content-wrapper">
        <div class = "float-left">
          <p class = "site-title">
            @Html.ActionLink("Tutorial Point", "Index", "Home")
          </p>
        </div>

        <div class = "float-right">
          <nav>
            <ul id = "menu">
              <li>@Html.ActionLink("Home", "Index", "Home")</li>
              <li>@Html.ActionLink("About", "About", "Home")</li>
            </ul>
          </nav>
        </div>
      </div>

    </header>
    <div id = "body">
      @RenderSection("featured", required: false)
      <section class = "content-wrapper main-content clear-fix">
        @RenderBody()
      </section>
    </div>
  </body>
</html>
```

```

        </section>
    </div>

    <footer>
        <div class = "content-wrapper">
            <div class = "float-left">
                <p>© @DateTime.Now.Year - Tutorial Point</p>
            </div>
        </div>
    </footer>

</body>
</html>

```

In this layout, we are using an HTML helper method and some other system-defined methods, hence let's look at these methods one by one.

Url.Content() – This method specifies the path of any file that we are using in our View code. It takes the virtual path as input and returns the absolute path.

Html.ActionLink() – This method renders HTML links that link to action of some controller. The first parameter specifies the display name, the second parameter specifies the Action name, and the third parameter specifies the Controller name.

RenderSection() – Specifies the name of the section that we want to display at that location in the template.

RenderBody() – Renders the actual body of the associated View.

Step 5 – Finally, open the `_ViewStart.cshtml` file inside Views folder and add the following code –

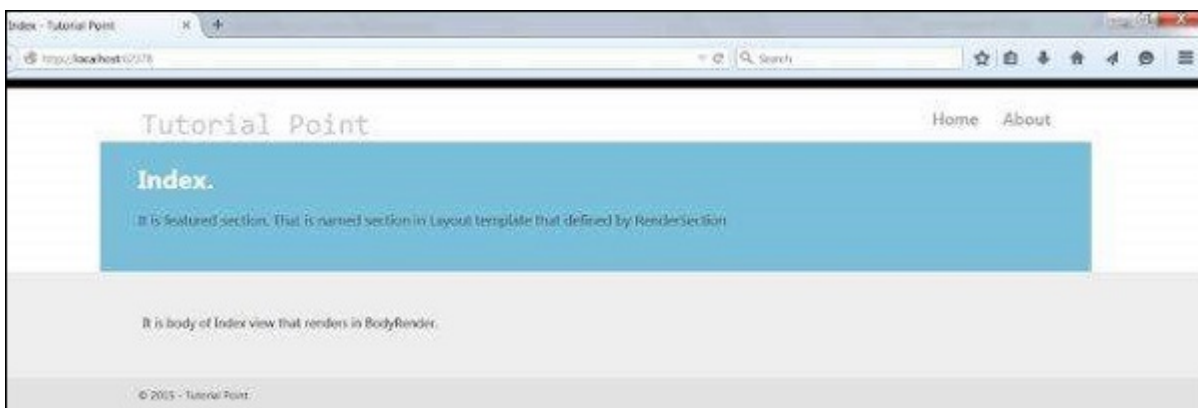
```

@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

```

If the file is not present, you can create the file with this name.

Step 6 – Run the application now to see the modified home page.



MVC Framework - Routing Engine

ASP.NET MVC Routing enables the use of URLs that are descriptive of the user actions and are more easily understood by the users. At the same time, Routing can be used to hide data which is not intended to be shown to the final user.

For example, in an application that does not use routing, the user would be shown the URL as `http://myapplication/Users.aspx?id=1` which would correspond to the file `Users.aspx` inside `myapplication` path and sending ID as 1, Generally, we would not like to show such file names to our final user.

To handle MVC URLs, ASP.NET platform uses the routing system, which lets you create any pattern of URLs you desire, and express them in a clear and concise manner. Each route in MVC contains a specific URL pattern. This URL pattern is compared to the incoming request URL and if the URL matches this pattern, it is used by the routing engine to further process the request.

MVC Routing URL Format

To understand the MVC routing, consider the following URL –

```
http://servername/Products/Phones
```

In the above URL, `Products` is the first segment and `Phone` is the second segment which can be expressed in the following format –

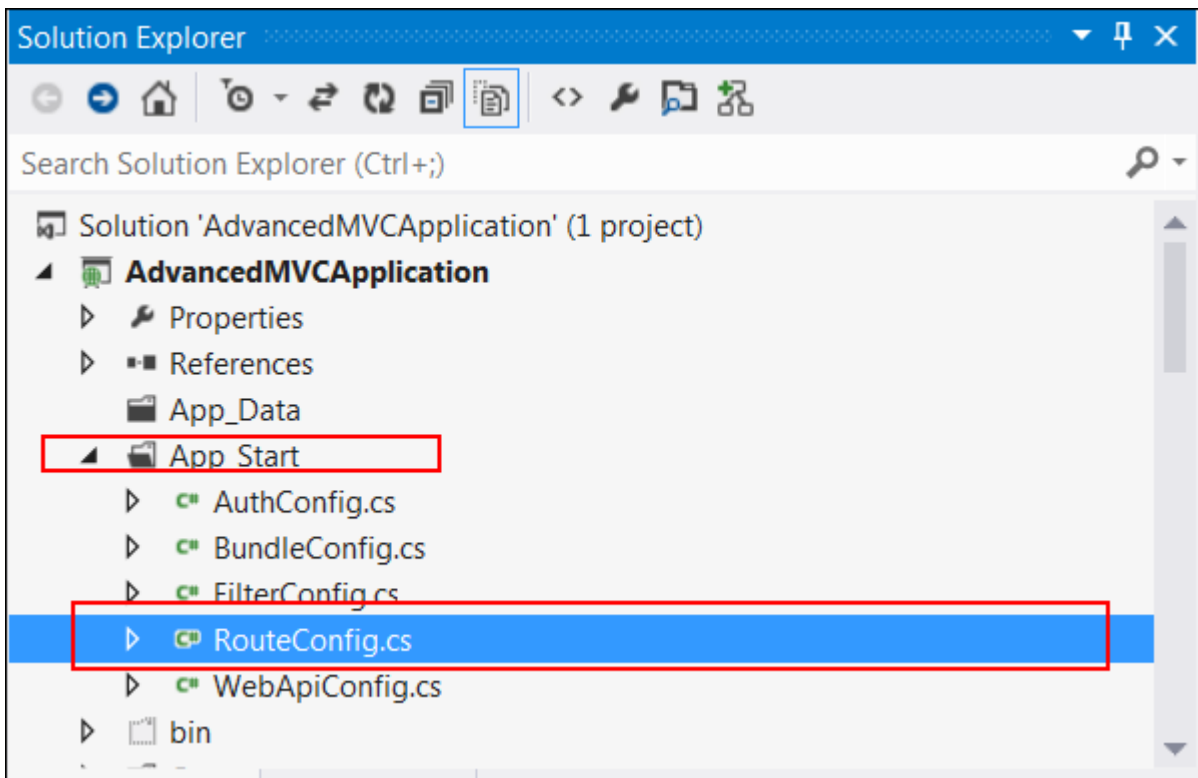
```
{controller}/{action}
```

The MVC framework automatically considers the first segment as the Controller name and the second segment as one of the actions inside that Controller.

Note – If the name of your Controller is `ProductsController`, you would only mention `Products` in the routing URL. The MVC framework automatically understands the Controller suffix.

Create a Simple Route

Routes are defined in the `RouteConfig.cs` file which is present under the `App_Start` project folder.



You will see the following code inside this file –

```
public class RouteConfig {  
  
    public static void RegisterRoutes(RouteCollection routes) {  
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
  
        routes.MapRoute(  
            name: "Default",  
            url: "{controller}/{action}/{id}",  
            defaults: new { controller = "Home", action = "Index",  
                           id = UrlParameter.Optional }  
        );  
    }  
}
```

This RegisterRoutes method is called by the Global.ascx when the application is started. The Application_Start method under Global.ascx calls this MapRoute function which sets the default Controller and its action (method inside the Controller class).

To modify the above default mapping as per our example, change the following line of code –

```
defaults: new { controller = "Products", action = "Phones", id = UrlParameter.Optional }
```

This setting will pick the ProductsController and call the Phone method inside that. Similarly, if you have another method such as Electronics inside ProductsController, the URL for it would be –

http://servername/Products/Electronics

MVC Framework - Action Filters

In ASP.NET MVC, controllers define action methods and these action methods generally have a one-to-one relationship with UI controls, such as clicking a button or a link, etc. For example, in one of our previous examples, the UserController class contained methods UserAdd, UserDelete, etc.

However, many times we would like to perform some action before or after a particular operation. For achieving this functionality, ASP.NET MVC provides a feature to add pre- and post-action behaviors on the controller's action methods.

Types of Filters

ASP.NET MVC framework supports the following action filters –

Action Filters – Action filters are used to implement logic that gets executed before and after a controller action executes. We will look at Action Filters in detail in this chapter.

Authorization Filters – Authorization filters are used to implement authentication and authorization for controller actions.

Result Filters – Result filters contain logic that is executed before and after a view result is executed. For example, you might want to modify a view result right before the view is rendered to the browser.

Exception Filters – Exception filters are the last type of filter to run. You can use an exception filter to handle errors raised by either your controller actions or controller action results. You also can use exception filters to log errors.

Action filters are one of the most commonly used filters to perform additional data processing, or manipulating the return values or cancelling the execution of action or modifying the view structure at run time.

Action Filters

Action Filters are additional attributes that can be applied to either a controller section or the entire controller to modify the way in which an action is executed. These attributes are special .NET classes derived from System.Attribute which can be attached to classes, methods, properties, and fields.

ASP.NET MVC provides the following action filters –

Output Cache – This action filter caches the output of a controller action for a specified amount of time.

Handle Error – This action filter handles errors raised when a controller action executes.

Authorize – This action filter enables you to restrict access to a particular user or role.

Now, we will see the code example to apply these filters on an example controller ActionFilterDemoController. (ActionFilterDemoController is just used as an example. You can use these filters on any of your controllers.)

Output Cache

Example – Specifies the return value to be cached for 10 seconds.

```
public class ActionFilterDemoController : Controller {  
    [HttpGet]  
    [OutputCache(Duration = 10)]  
  
    public string Index() {  
        return DateTime.Now.ToString("T");  
    }  
}
```

Handle Error

Example – Redirects application to a custom error page when an error is triggered by the controller.

```
[HandleError]  
public class ActionFilterDemoController : Controller {  
  
    public ActionResult Index() {  
        throw new NullReferenceException();  
    }  
  
    public ActionResult About() {  
        return View();  
    }  
}
```

With the above code, if any error happens during the action execution, it will find a view named Error in the Views folder and render that page to the user.

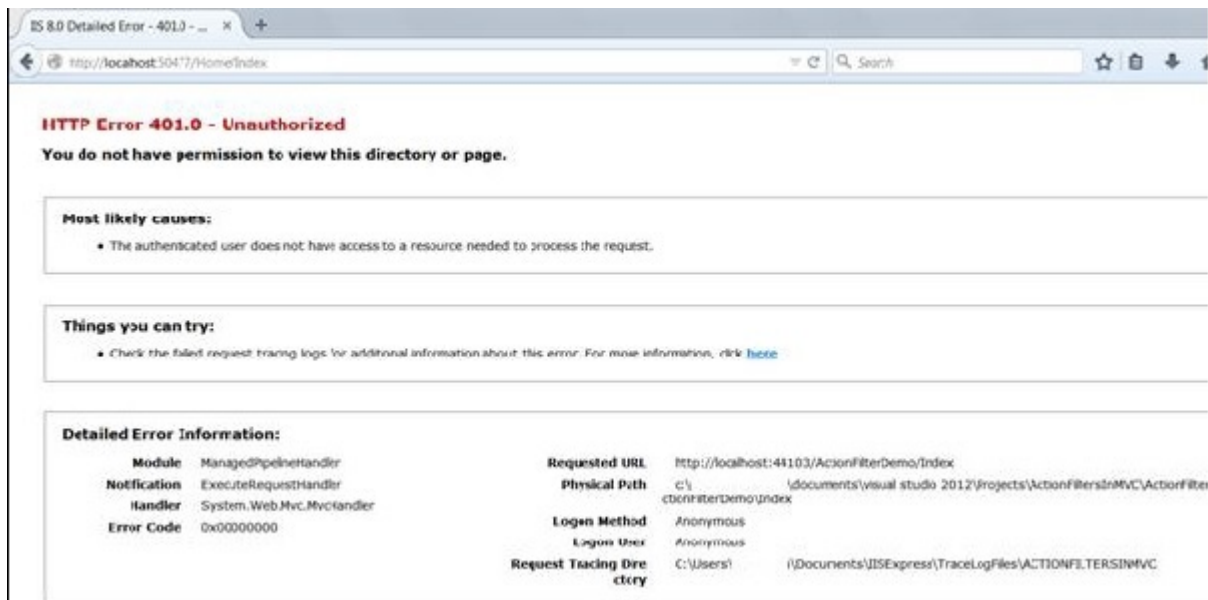
Authorize

Example – Allowing only authorized users to log in the application.

```
public class ActionFilterDemoController: Controller {  
    [Authorize]  
  
    public ActionResult Index() {  
        ViewBag.Message = "This can be viewed only by authenticated users only";  
        return View();  
    }  
  
    [Authorize(Roles="admin")]  
    public ActionResult AdminIndex() {  
        ViewBag.Message = "This can be viewed only by users in Admin role only";  
    }  
}
```

```
    return View();  
}  
}
```

With the above code, if you would try to access the application without logging in, it will throw an error similar to the one shown in the following screenshot.

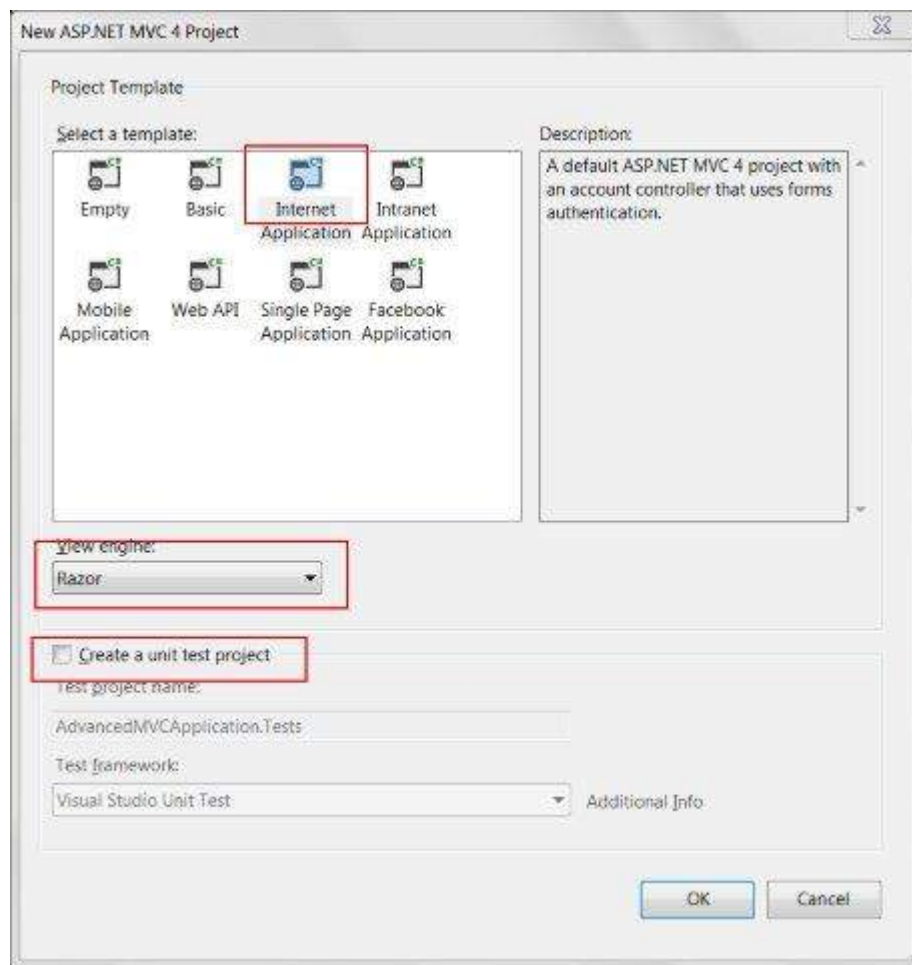


MVC Framework - Advanced Example

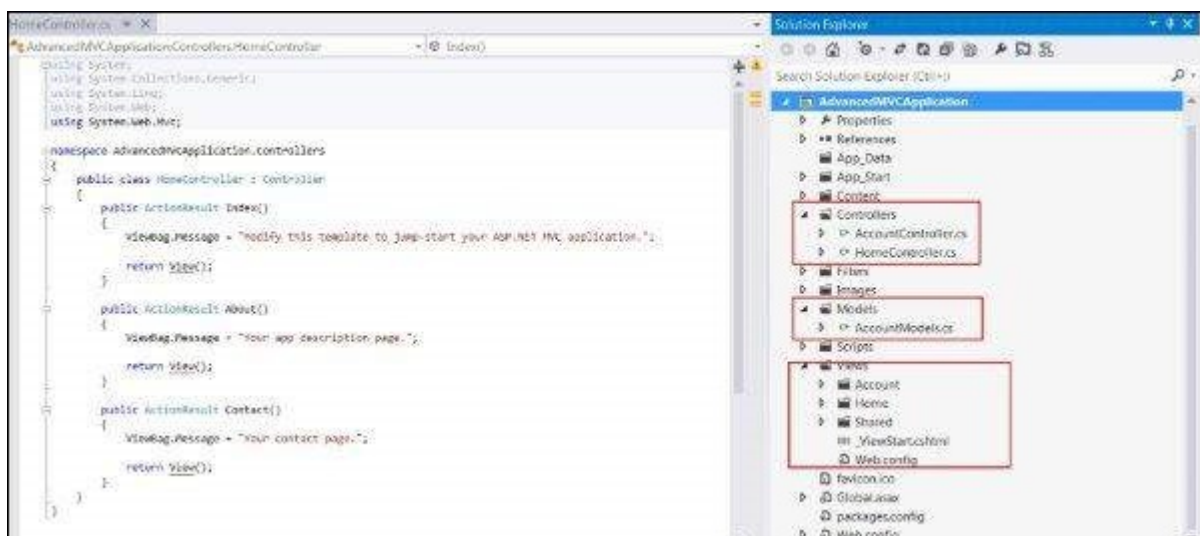
In the first chapter, we learnt how Controllers and Views interact in MVC. In this tutorial, we are going to take a step forward and learn how to use Models and create an advanced application to create, edit, delete, and view the list of users in our application.

Create an Advanced MVC Application

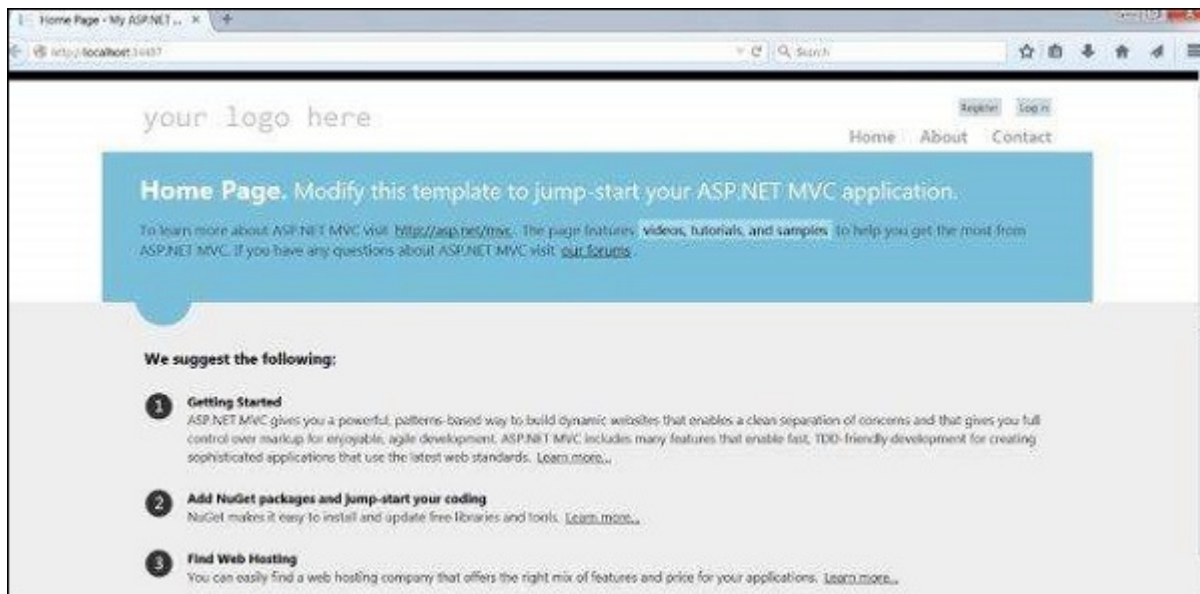
Step 1 – Select File → New → Project → ASP.NET MVC Web Application. Name it as AdvancedMVCApplcation. Click Ok. In the next window, select Template as Internet Application and View Engine as Razor. Observe that we are using a template this time instead of an Empty application.



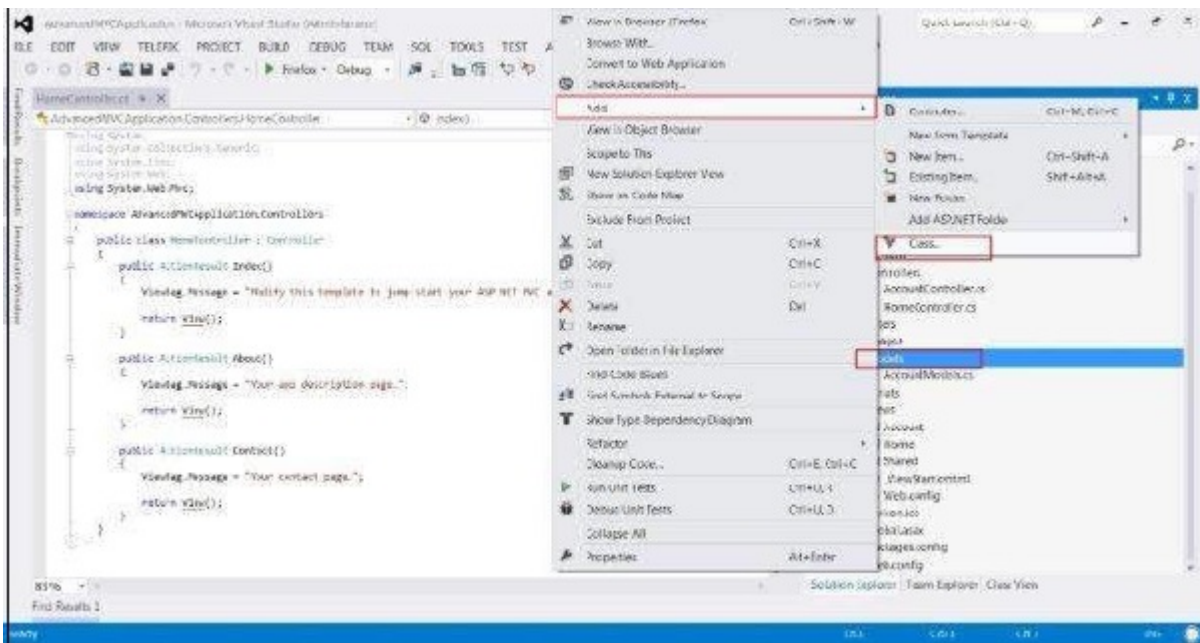
This will create a new solution project as shown in the following screenshot. Since we are using the default ASP.NET theme, it comes with sample Views, Controllers, Models and other files.

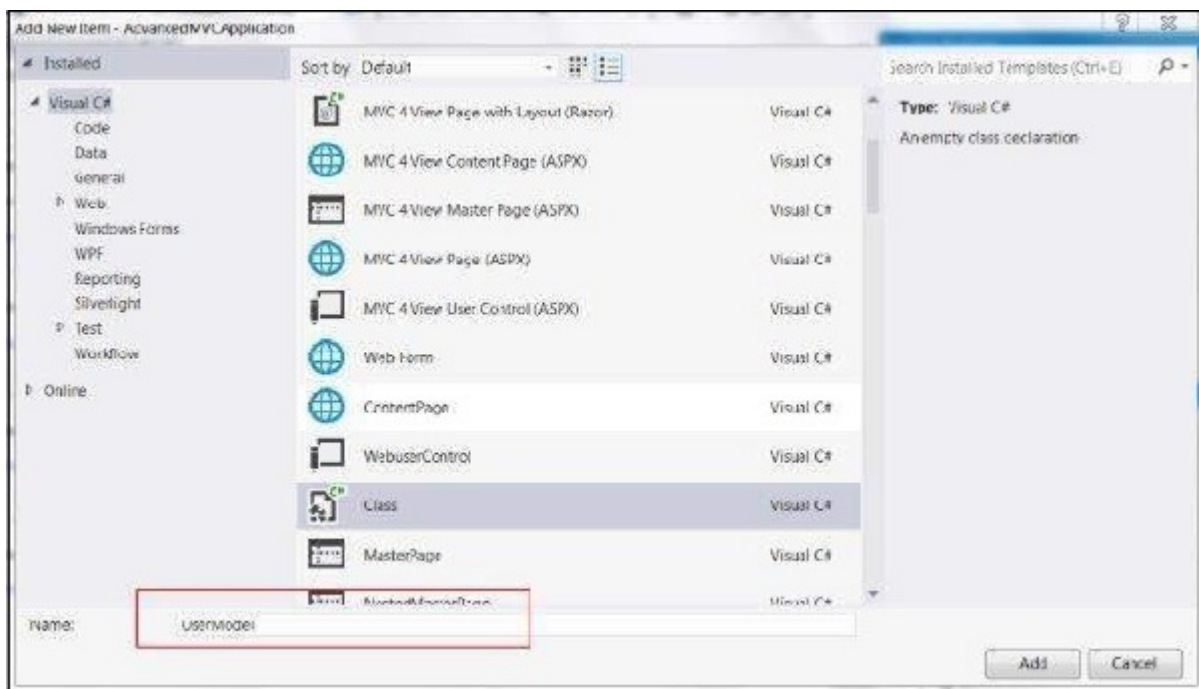


Step 2 – Build the solution and run the application to see its default output as shown in the following screenshot.



Step 3 – Add a new model which will define the structure of users data. Right-click on Models folder and click Add → Class. Name this as UserModel and click Add.





Step 4 – Copy the following code in the newly created UserModel.cs.

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc.Html;

namespace AdvancedMVCAApplication.Models {
    public class UserModels {

        [Required]
        public int Id { get; set; }
        [DisplayName("First Name")]
        [Required(ErrorMessage = "First name is required")]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }

        public string Address { get; set; }

        [Required]
        [StringLength(50)]
        public string Email { get; set; }

        [DataType(DataType.Date)]
        public DateTime DOB { get; set; }

        [Range(100,1000000)]
        public decimal Salary { get; set; }
    }
}
```

In the above code, we have specified all the parameters that the User model has, their data types and validations such as required fields and length.

Now that we have our User Model ready to hold the data, we will create a class file Users.cs, which will contain methods for viewing users, adding, editing, and deleting users.

Step 5 – Right-click on Models and click Add → Class. Name it as Users. This will create users.cs class inside Models. Copy the following code in the users.cs class.

```
using System;
using System.Collections.Generic;
using System.EnterpriseServices;

namespace AdvancedMVCAApplication.Models {

    public class Users {
        public List UserModel = new List();

        //action to get user details
        public UserModel GetUser(int id) {
            UserModel usrMdl = null;

            foreach (UserModel um in UserModel)

                if (um.Id == id)
                    usrMdl = um;
                return usrMdl;
        }

        //action to create new user
        public void CreateUser(UserModel userModel) {
            UserModel.Add(userModel);
        }

        //action to update existing user
        public void UpdateUser(UserModel userModel) {

            foreach (UserModel usr1st in UserModel) {

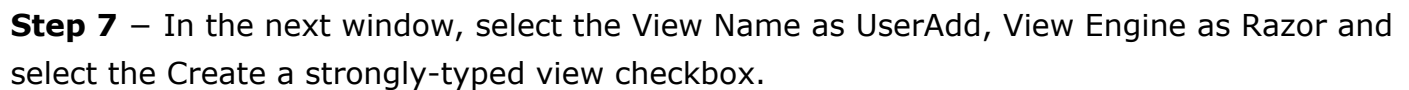
                if (usr1st.Id == userModel.Id) {
                    usr1st.Address = userModel.Address;
                    usr1st.DOB = userModel.DOB;
                    usr1st.Email = userModel.Email;
                    usr1st.FirstName = userModel.FirstName;
                    usr1st.LastName = userModel.LastName;
                    usr1st.Salary = userModel.Salary;
                    break;
                }
            }
        }

        //action to delete existing user
        public void DeleteUser(UserModel userModel) {

            foreach (UserModel usr1st in UserModel) {

                if (usr1st.Id == userModel.Id) {
                    UserModel.Remove(usr1st);
                    break;
                }
            }
        }
    }
}
```

Step 6 – Right-click on the Views folder and click Add → View.



Add View

View name: UserAdd

View engine: Razor (CSHTML)

☒ Create a strongly-typed view

Model class: UserModels (AdvancedMVCApplication.Models)

Scaffold template: Empty

☒ Reference script libraries

☐ Create as a partial view

☐ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID: MainContent

Add Cancel

Step 8 – Click Add. This will create the following CSHTML code by default as shown below

```
@model AdvancedMVCApplication.Models.UserModels

@{
    ViewBag.Title = "UserAdd";
}

<h2>UserAdd</h2>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>UserModels</legend>
        <div class = "editor-label">
            @Html.LabelFor(model => model.FirstName)
```

```

</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.FirstName)
    @Html.ValidationMessageFor(model => model.FirstName)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.LastName)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.LastName)
    @Html.ValidationMessageFor(model => model.LastName)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Address)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Address)
    @Html.ValidationMessageFor(model => model.Address)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Email)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Email)
    @Html.ValidationMessageFor(model => model.Email)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.DOB)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.DOB)
    @Html.ValidationMessageFor(model => model.DOB)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Salary)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Salary)
    @Html.ValidationMessageFor(model => model.Salary)
</div>

<p>
    <input type = "submit" value = "Create" />
</p>
</fieldset>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

```

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

As you can see, this view contains view details of all the attributes of the fields including their validation messages, labels, etc. This View will look like the following in our final application.

The screenshot shows a web application titled 'Tutorial Point' with navigation links for Home, About, and Contact. The main content area is titled 'UserAdd' and contains a form with the following fields: First Name (Mark), LastName (Gubbins), Address (XYZ), Email (mark@something), DOB (05/09/2015), and Salary. A 'Create' button is located below the form, and a 'Back to List' link is at the bottom left. The footer indicates '© 2015 - My ASP.NET MVC Application'.

Similar to UserAdd, now we will add four more Views given below with the given code –

Index.cshtml

This View will display all the users present in our system on the Index page.

```
@model IEnumerable<AdvancedMVCAApplication.Models.UserModels>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "UserAdd")
</p>

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.FirstName)
        </th>
```

```

<th>
    @Html.DisplayNameFor(model => model.LastName)
</th>

<th>
    @Html.DisplayNameFor(model => model.Address)
</th>

<th>
    @Html.DisplayNameFor(model => model.Email)
</th>

<th>
    @Html.DisplayNameFor(model => model.DOB)
</th>

<th>
    @Html.DisplayNameFor(model => model.Salary)
</th>

<th></th>
</tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.FirstName)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.LastName)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Address)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Email)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.DOB)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.Salary)
        </td>

        <td>
            @Html.ActionLink("Edit", "Edit", new { id = item.Id }) |
            @Html.ActionLink("Details", "Details", new { id = item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id = item.Id })
        </td>
    </tr>
}
</table>

```

This View will look like the following in our final application.

Tutorials Point

[Register](#)[Log in](#)

[Home](#)[About](#)[Contact](#)

Index

[Create New](#)

First Name	LastName	Address	Email	DOB	Salary	
Mark	Gubbins	Chicago, USA	mark@something	3/1/1991	15000.00	Edit Details Delete
Rob	Dsouza	Baltimore	rob@something	4/2/1990	20000.00	Edit Details Delete
Ashish	Trivedi	India	ashish@something	3/13/2015	129900.00	Edit Details Delete

Details.cshtml

This View will display the details of a specific user when we click on the user record.

```
@model AdvancedMVCApplication.Models.UserModels

@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>
<fieldset>
    <legend>UserModels</legend>
    <div class = "display-label">
        @Html.DisplayNameFor(model => model.FirstName)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.FirstName)
    </div>

    <div class = "display-label">
        @Html.DisplayNameFor(model => model.LastName)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.LastName)
    </div>

    <div class = "display-label">
        @Html.DisplayNameFor(model => model.Address)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.Address)
    </div>

    <div class = "display-label">
        @Html.DisplayNameFor(model => model.Email)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.Email)
    </div>

    <div class = "display-label">
        @Html.DisplayNameFor(model => model.DOB)
    </div>
```



```

</div>

<div class = "display-field">
    @Html.DisplayFor(model => model.DOB)
</div>

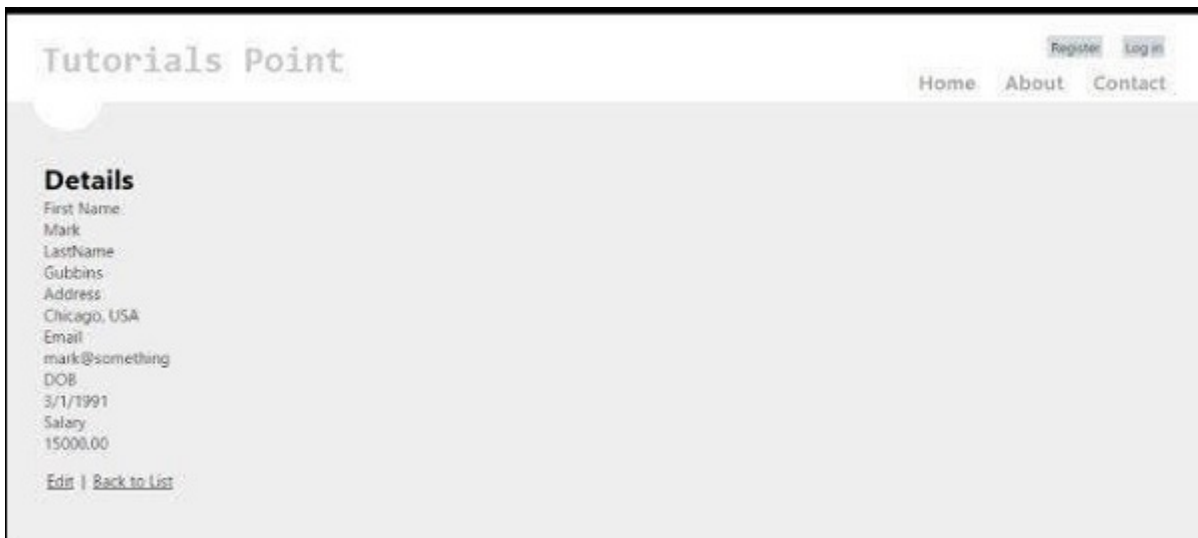
<div class = "display-label">
    @Html.DisplayNameFor(model => model.Salary)
</div>

<div class = "display-field">
    @Html.DisplayFor(model => model.Salary)
</div>

</fieldset>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>

```

This View will look like the following in our final application.



Edit.cshtml

This View will display the edit form to edit the details of an existing user.

```

@model AdvancedMVCAApplication.Models.UserModels

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>
@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()
    @Html.ValidationSummary(true)

    <fieldset>
        <legend>UserModels</legend>
        @Html.HiddenFor(model => model.Id)
        <div class = "editor-label">
            @Html.LabelFor(model => model.FirstName)
        </div>

```

```

<div class = "editor-field">
    @Html.EditorFor(model => model.FirstName)
    @Html.ValidationMessageFor(model => model.FirstName)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.LastName)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.LastName)
    @Html.ValidationMessageFor(model => model.LastName)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Address)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Address)
    @Html.ValidationMessageFor(model => model.Address)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Email)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Email)
    @Html.ValidationMessageFor(model => model.Email)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.DOB)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.DOB)
    @Html.ValidationMessageFor(model => model.DOB)
</div>

<div class = "editor-label">
    @Html.LabelFor(model => model.Salary)
</div>

<div class = "editor-field">
    @Html.EditorFor(model => model.Salary)
    @Html.ValidationMessageFor(model => model.Salary)
</div>

<p>
    <input type = "submit" value = "Save" />
</p>
</fieldset>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {

```

```

@Scripts.Render("~/bundles/jqueryval")
}

```

This View will look like the following in our application.

The screenshot shows a web application titled 'Tutorials Point' with navigation links for Home, About, and Contact. The main content area is titled 'Edit' and contains a form for editing a user. The form fields are: First Name (Mark), LastName (Gubbirs), Address (Chicago, USA), Email (mark@something), DOB (mm/dd/yyyy), and Salary (15000.00). A 'Save' button is located below the fields, and a 'Back to List' link is at the bottom left of the form area.

Delete.cshtml

This View will display the form to delete the existing user.

```

@model AdvancedMVCAApplication.Models.UserModels

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>
<h3>Are you sure you want to delete this?</h3>
<fieldset>
    <legend>UserModels</legend>
    <div class = "display-label">
        @Html.DisplayNameFor(model => model.FirstName)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.FirstName)
    </div>

    <div class = "display-label">
        @Html.DisplayNameFor(model => model.LastName)
    </div>

    <div class = "display-field">
        @Html.DisplayFor(model => model.LastName)
    </div>

```

```

<div class = "display-label">
    @Html.DisplayNameFor(model => model.Address)
</div>

<div class = "display-field">
    @Html.DisplayFor(model => model.Address)
</div>

<div class = "display-label">
    @Html.DisplayNameFor(model => model.Email)
</div>

<div class = "display-field">
    @Html.DisplayFor(model => model.Email)
</div>

<div class = "display-label">
    @Html.DisplayNameFor(model => model.DOB)
</div>

<div class = "display-field">
    @Html.DisplayFor(model => model.DOB)
</div>

<div class = "display-label">
    @Html.DisplayNameFor(model => model.Salary)
</div>

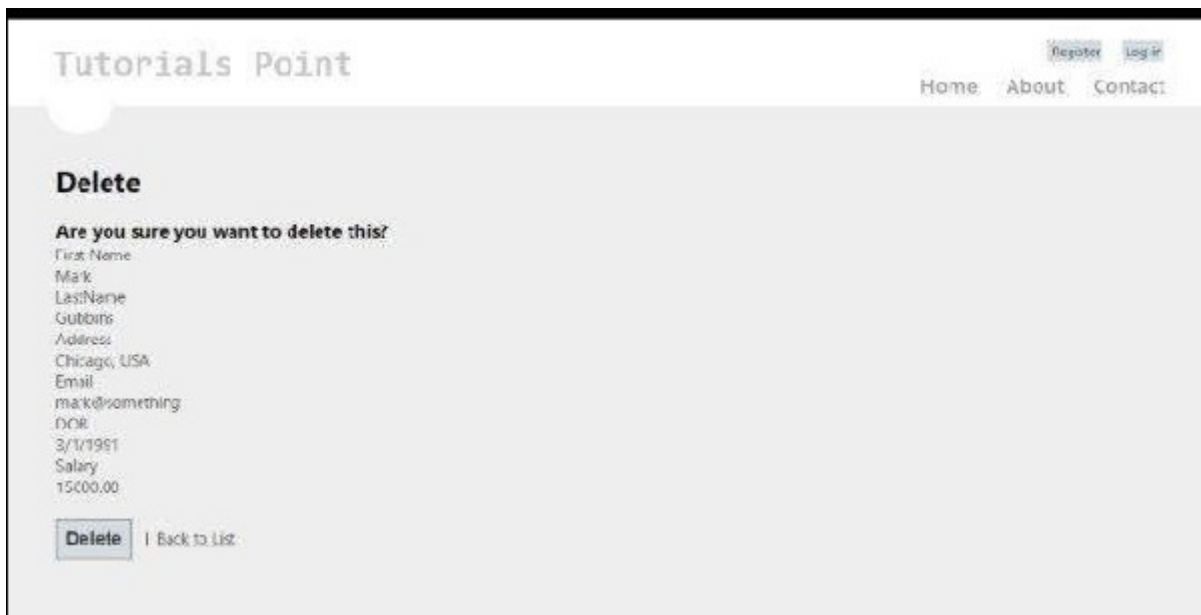
<div class = "display-field">
    @Html.DisplayFor(model => model.Salary)
</div>
</fieldset>

@using (Html.BeginForm()) {
    @Html.AntiForgeryToken()

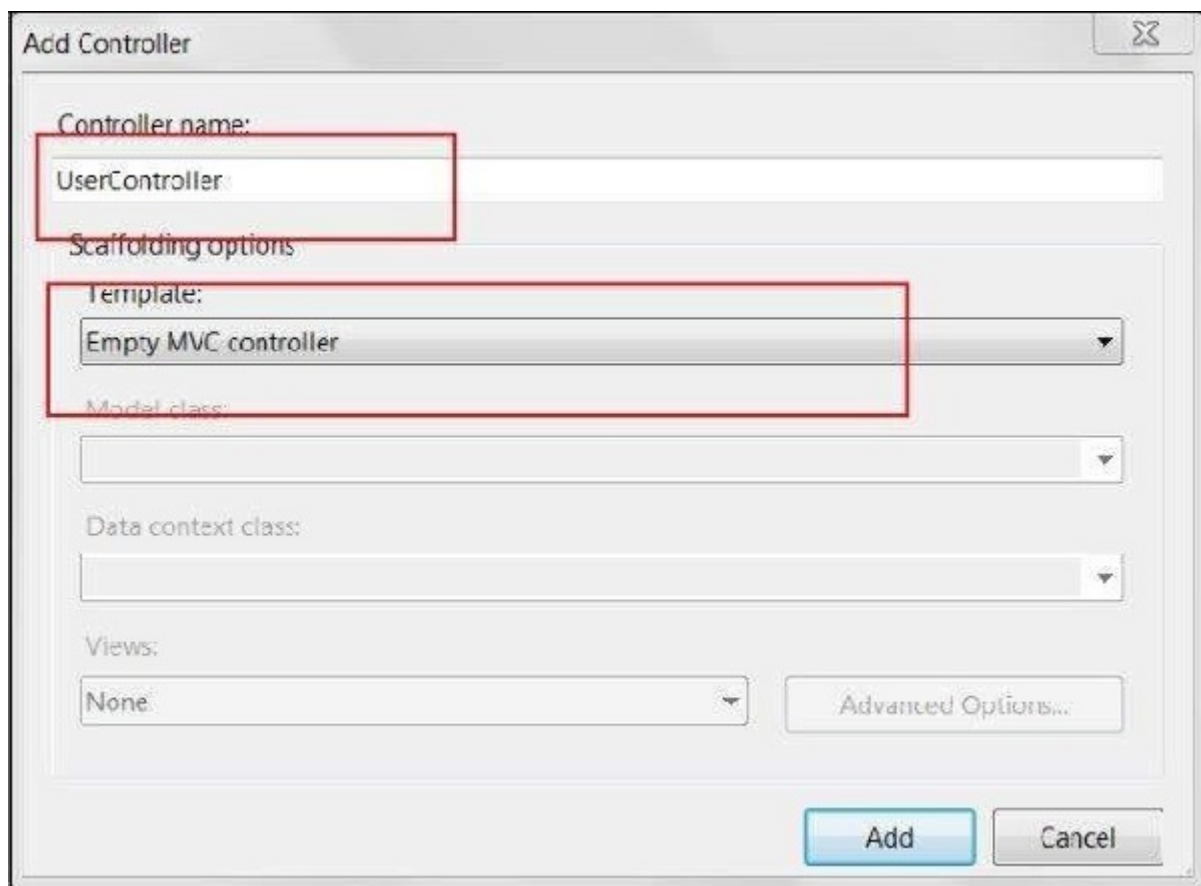
    <p>
        <input type = "submit" value = "Delete" /> |
        @Html.ActionLink("Back to List", "Index")
    </p>
}

```

This View will look like the following in our final application.



Step 9 – We have already added the Models and Views in our application. Now finally we will add a controller for our view. Right-click on the Controllers folder and click Add → Controller. Name it as UserController.



By default, your Controller class will be created with the following code –

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

```
using AdvancedMVCAApplication.Models;

namespace AdvancedMVCAApplication.Controllers {

    public class UserController : Controller {
        private static Users _users = new Users();

        public ActionResult Index() {
            return View(_users.UserList);
        }
    }
}
```

In the above code, the Index method will be used while rendering the list of users on the Index page.

Step 10 – Right-click on the Index method and select Create View to create a View for our Index page (which will list down all the users and provide options to create new users).

Add View

View name: Index

View engine: Razor (CSHTML)

☒ Create a strongly-typed view

Model class: UserModels (AdvancedMVCApplication.Models)

Scaffold template: List ☒ Reference script libraries

☐ Create as a partial view

☒ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID: MainContent

Add Cancel

Step 11 – Now add the following code in the UserController.cs. In this code, we are creating action methods for different user actions and returning corresponding views that we created earlier.

We will add two methods for each operation: GET and POST. HttpGet will be used while fetching the data and rendering it. HttpPost will be used for creating/updating data. For example, when we are adding a new user, we will need a form to add a user, which is a GET operation. Once we fill the form and submit those values, we will need the POST method.

```
//Action for Index View
public ActionResult Index() {
    return View(_users.UserList);
}
```



```

//Action for UserAdd View
[HttpGet]
public ActionResult UserAdd() {
    return View();
}

[HttpPost]
public ActionResult UserAdd(UserModels userModel) {
    _users.CreateUser(userModel);
    return View("Index", _users.UserList);
}

//Action for Details View
[HttpGet]
public ActionResult Details(int id) {
    return View(_users.UserList.FirstOrDefault(x => x.Id == id));
}

[HttpPost]
public ActionResult Details() {
    return View("Index", _users.UserList);
}

//Action for Edit View
[HttpGet]
public ActionResult Edit(int id) {
    return View(_users.UserList.FirstOrDefault(x=>x.Id==id));
}

[HttpPost]
public ActionResult Edit(UserModels userModel) {
    _users.UpdateUser(userModel);
    return View("Index", _users.UserList);
}

//Action for Delete View
[HttpGet]
public ActionResult Delete(int id) {
    return View(_users.UserList.FirstOrDefault(x => x.Id == id));
}

[HttpPost]
public ActionResult Delete(UserModels userModel) {
    _users.DeleteUser(userModel);
    return View("Index", _users.UserList);
} sers.UserList);

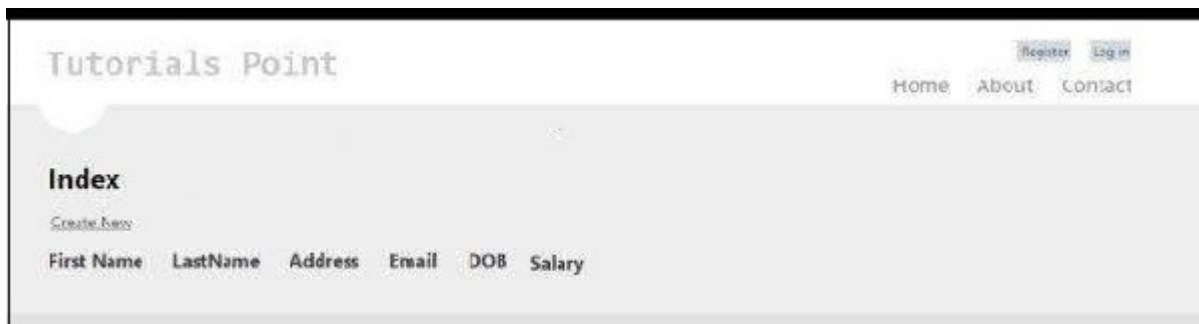
```

Step 12 – Last thing to do is go to RouteConfig.cs file in App_Start folder and change the default Controller to User.

```
defaults: new { controller = "User", action = "Index", id = UrlParameter.Optional }
```

That's all we need to get our advanced application up and running.

Step 13 – Now run the application. You will be able to see an application as shown in the following screenshot. You can perform all the functionalities of adding, viewing, editing, and deleting users as we saw in the earlier screenshots.

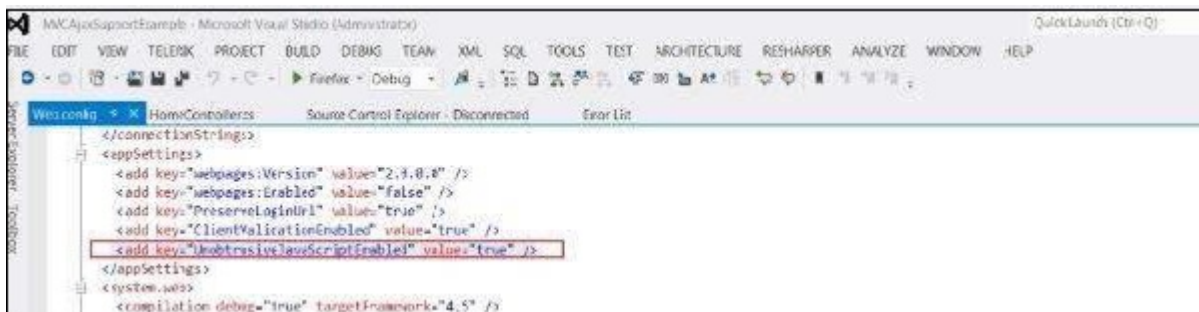


MVC Framework - Ajax Support

As you might be knowing, Ajax is a shorthand for Asynchronous JavaScript and XML. The MVC Framework contains built-in support for unobtrusive Ajax. You can use the helper methods to define your Ajax features without adding a code throughout all the views. This feature in MVC is based on the jQuery features.

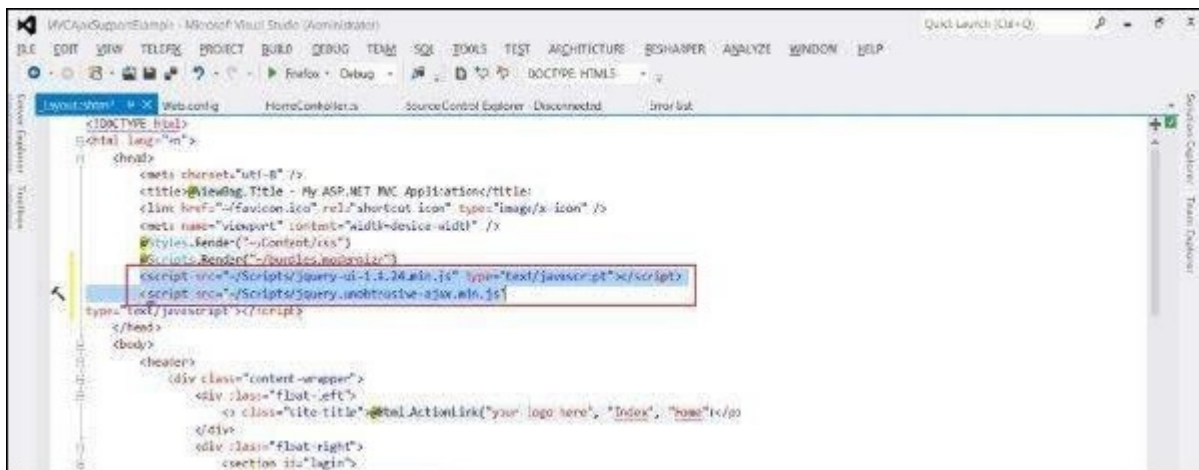
To enable the unobtrusive AJAX support in the MVC application, open the Web.Config file and set the UnobtrusiveJavaScriptEnabled property inside the appSettings section using the following code. If the key is already present in your application, you can ignore this step.

```
<add key = "UnobtrusiveJavaScriptEnabled" value = "true" />
```



After this, open the common layout file **_Layout.cshtml** file located under Views/Shared folder. We will add references to the jQuery libraries here using the following code –

```
<script src = "~/Scripts/jquery-ui-1.8.24.min.js" type = "text/javascript">  
</script>  
  
<script src = "~/Scripts/jquery.unobtrusive-ajax.min.js" type = "text/javascript">  
</script>
```



Create an Unobtrusive Ajax Application

In the example that follows, we will create a form which will display the list of users in the system. We will place a dropdown having three options: Admin, Normal, and Guest. When you will select one of these values, it will display the list of users belonging to this category using unobtrusive AJAX setup.

Step 1 – Create a Model file Model.cs and copy the following code.

```
using System;

namespace MVCAjaxSupportExample.Models {

    public class User {
        public int UserId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Role Role { get; set; }
    }

    public enum Role {
        Admin,
        Normal,
        Guest
    }
}
```

Step 2 – Create a Controller file named UserController.cs and create two action methods inside that using the following code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using MVCAjaxSupportExample.Models;

namespace MVCAjaxSupportExample.Controllers {

    public class UserController : Controller {
```

```

private readonly User[] userData =
{
    new User {FirstName = "Edy", LastName = "Clooney", Role = Role.Admin},
    new User {FirstName = "David", LastName = "Sanderson", Role = Role.Admin},
    new User {FirstName = "Pandy", LastName = "Griffyth", Role = Role.Normal},
    new User {FirstName = "Joe", LastName = "Gubbins", Role = Role.Normal},
    new User {FirstName = "Mike", LastName = "Smith", Role = Role.Guest}
};

public ActionResult Index() {
    return View(userData);
}

public PartialViewResult GetUserData(string selectedRole = "All") {
    IEnumerable data = userData;

    if (selectedRole != "All") {
        var selected = (Role) Enum.Parse(typeof (Role), selectedRole);
        data = userData.Where(p => p.Role == selected);
    }

    return PartialView(data);
}

public ActionResult GetUser(string selectedRole = "All") {
    return View((object) selectedRole);
}
}

```

Step 3 – Now create a partial View named GetUserData with the following code. This view will be used to render list of users based on the selected role from the dropdown.

```

@model IEnumerable<MVCAjaxSupportExample.Models.User>

<table>
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.FirstName)
        </th>

        <th>
            @Html.DisplayNameFor(model => model.LastName)
        </th>

        <th>
            @Html.DisplayNameFor(model => model.BirthDate)
        </th>
    </tr>

    @foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.FirstName)
        </td>

        <td>
            @Html.DisplayFor(modelItem => item.LastName)

```

```

        </td>

        <td>
            @Html.DisplayFor(modelItem => item.BirthDate)
        </td>

        <td>

        </td>
    </tr>
}
</table>

```

Step 4 – Now create a View GetUser with the following code. This view will asynchronously get the data from the previously created controller's GetUserData Action.

```

@using MVCAjaxSupportExample.Models
@model string

@{
    ViewBag.Title = "GetUser";

    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody"
    };
}

<h2>Get User</h2>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>

    <tbody id="tableBody">
        @Html.Action("GetUserData", new {selectedRole = Model })
    </tbody>
</table>

@using (Ajax.BeginForm("GetUser", ajaxOpts)) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] { "All" }.Concat(Enum.GetNames(typeof(Role))))
        <button type="submit">Submit</button>
    </div>
}

```

Step 5 – Finally, change the Route.config entries to launch the User Controller.

```

defaults: new { controller = "User", action = "GetUser", id = UrlParameter.Optional }

```

Step 6 – Run the application which will look like the following screenshot.

Tutorial Point Home

Get User

FirstLastRole

FirstName	LastName	BirthDate
Edy	Clooney	1/1/0001 12:00:00 AM
David	Sanderson	1/1/0001 12:00:00 AM
Pandy	Grifyth	1/1/0001 12:00:00 AM
Joe	Gubbins	1/1/0001 12:00:00 AM
Mike	Smith	1/1/0001 12:00:00 AM

All

If you select Admin from the dropdown, it will go and fetch all the users with Admin type. This is happening via AJAX and does not reload the entire page.

Tutorial Point Home

Get User

FirstLastRole

FirstName	LastName	BirthDate
Edy	Clooney	1/1/0001 12:00:00 AM
David	Sanderson	1/1/0001 12:00:00 AM

Admin

MVC Framework - Bundling

Bundling and **Minification** are two performance improvement techniques that improves the request load time of the application. Most of the current major browsers limit the number of simultaneous connections per hostname to six. It means that at a time, all the additional requests will be queued by the browser.

Enable Bundling and Minification

To enable bundling and minification in your MVC application, open the Web.config file inside your solution. In this file, search for compilation settings under system.web –

```
<system.web>
  <compilation debug = "true" />
</system.web>
```

By default, you will see the debug parameter set to true, which means that bundling and minification is disabled. Set this parameter to false.

Bundling

To improve the performance of the application, ASP.NET MVC provides inbuilt feature to bundle multiple files into a single, file which in turn improves the page load performance because of fewer HTTP requests.

Bundling is a simple logical group of files that could be referenced by unique name and loaded with a single HTTP request.

By default, the MVC application's BundleConfig (located inside App_Start folder) comes with the following code –

```
public static void RegisterBundles(BundleCollection bundles) {  
  
    // Following is the sample code to bundle all the css files in the project  
  
    // The code to bundle other javascript files will also be similar to this  
  
    bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(  
        "~/Content/themes/base/jquery.ui.core.css",  
        "~/Content/themes/base/jquery.ui.tabs.css",  
        "~/Content/themes/base/jquery.ui.datepicker.css",  
        "~/Content/themes/base/jquery.ui.progressbar.css",  
        "~/Content/themes/base/jquery.ui.theme.css"));  
}
```

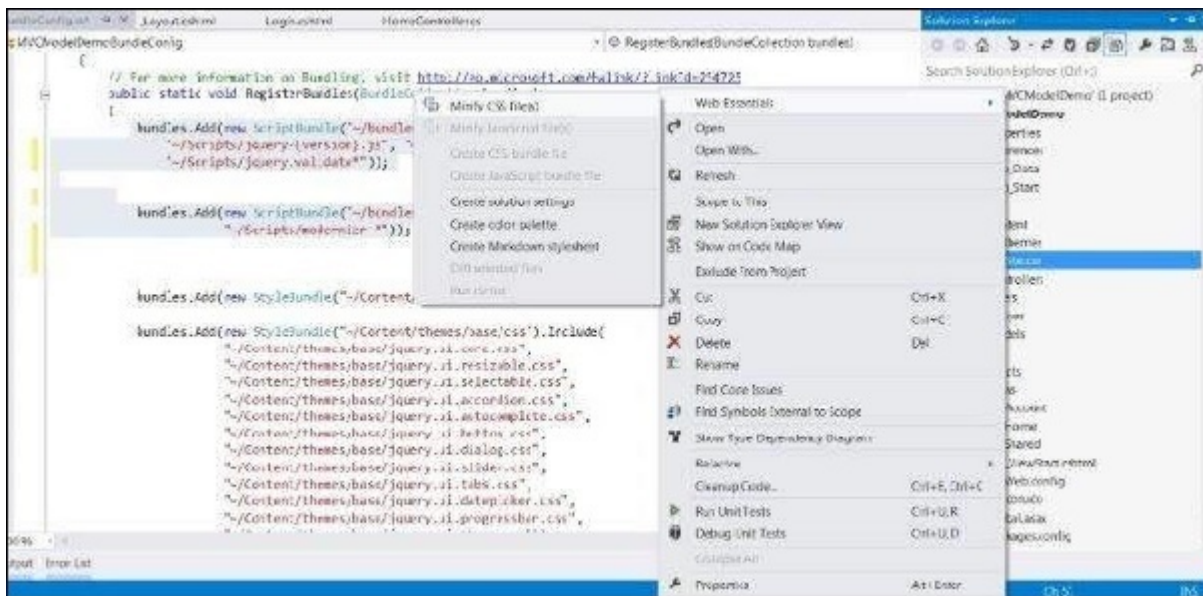
The above code basically bundles all the CSS files present in Content/themes/base folder into a single file.

Minification

Minification is another such performance improvement technique in which it optimizes the javascript, css code by shortening the variable names, removing unnecessary white spaces, line breaks, comments, etc. This in turn reduces the file size and helps the application to load faster.

Minification with Visual Studio and Web Essentials Extension

For using this option, you will have to first install the Web Essentials Extension in your Visual Studio. After that, when you will right-click on any css or javascript file, it will show you the option to create a minified version of that file.



Thus, if you have a css file named Site.css, it will create its minified version as Site.min.css.

Now when the next time your application will run in the browser, it will bundle and minify all the css and js files, hence improving the application performance.

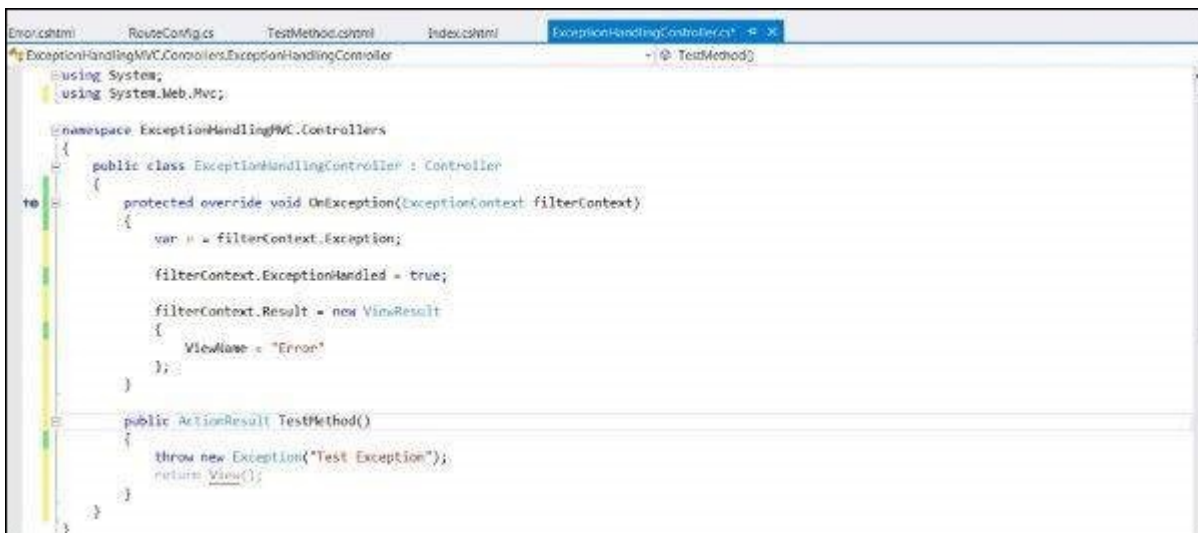
MVC Framework - Exception Handling

In ASP.NET, error handling is done using the standard try catch approach or using application events. ASP.NET MVC comes with built-in support for exception handling using a feature known as exception filters. We are going to learn two approaches here: one with overriding the onException method and another by defining the HandleError filters.

Override OnException Method

This approach is used when we want to handle all the exceptions across the Action methods at the controller level.

To understand this approach, create an MVC application (follow the steps covered in previous chapters). Now add a new Controller class and add the following code which overrides the onException method and explicitly throws an error in our Action method –



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

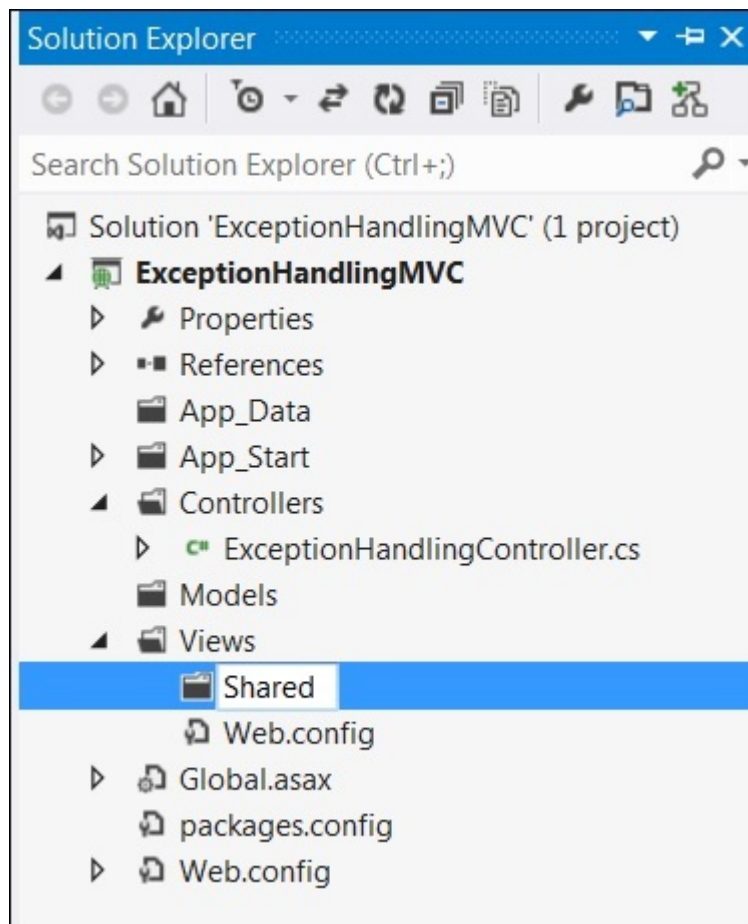
namespace ExceptionHandlingMVC.Controllers
{
    public class ExceptionHandlingController : Controller
    {
        protected override void OnException(ExceptionContext filterContext)
        {
            var ex = filterContext.Exception;

            filterContext.ExceptionHandled = true;

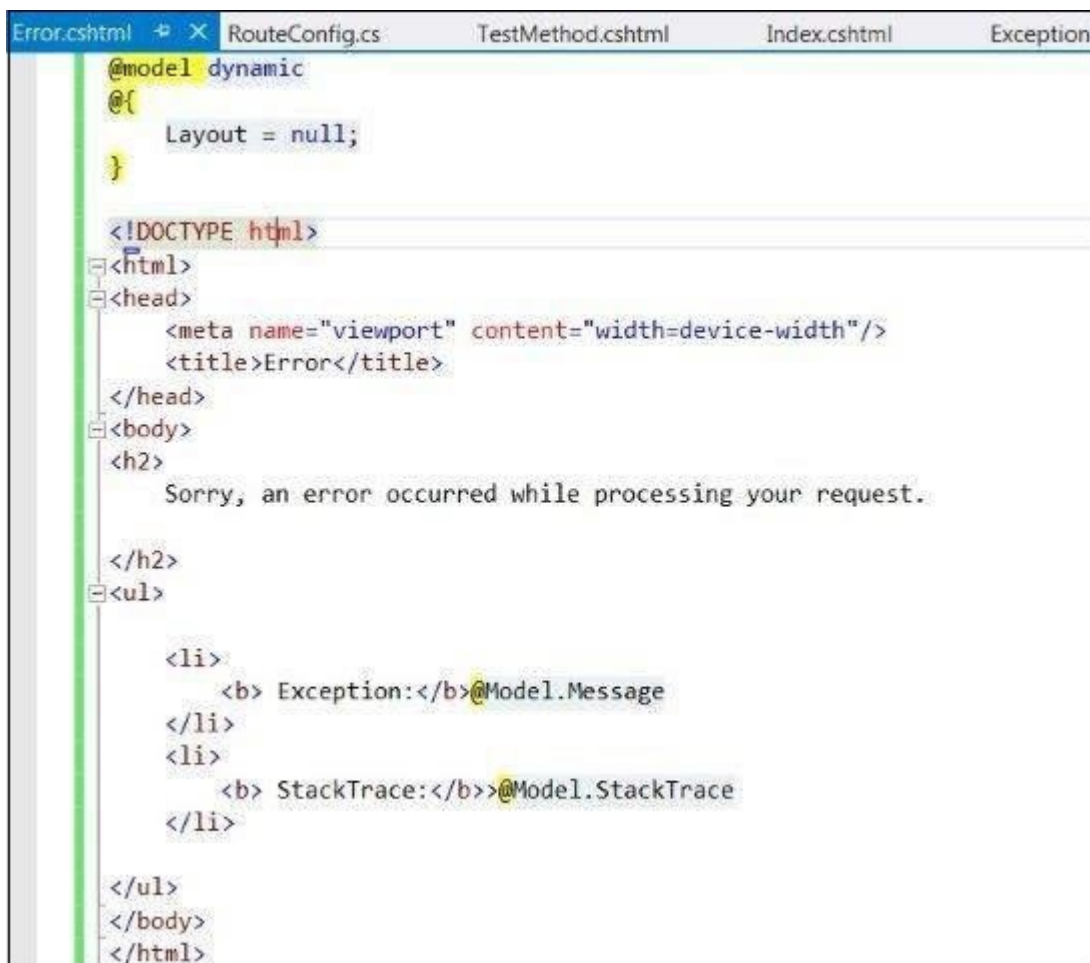
            filterContext.Result = new ViewResult
            {
                ViewName = "Error"
            };
        }

        public ActionResult TestMethod()
        {
            throw new Exception("Test Exception");
            return View();
        }
    }
}
```

Now let us create a common View named **Error** which will be shown to the user when any exception happens in the application. Inside the Views folder, create a new folder called Shared and add a new View named Error.



Copy the following code inside the newly created Error.cshtml –



```
@model dynamic
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"/>
    <title>Error</title>
</head>
<body>
    <h2>
        Sorry, an error occurred while processing your request.
    </h2>
    <ul>
        <li>
            <b> Exception:</b>@Model.Message
        </li>
        <li>
            <b> StackTrace:</b>@Model.StackTrace
        </li>
    </ul>
</body>
</html>
```

If you try to run the application now, it will give the following result. The above code renders the Error View when any exception occurs in any of the action methods within this controller.



The advantage of this approach is that multiple actions within the same controller can share this error handling logic. However, the disadvantage is that we cannot use the same error handling logic across multiple controllers.

HandleError Attribute

The HandleError Attribute is one of the action filters that we studied in Filters and Action Filters chapter. The HandleErrorAttribute is the default implementation of IExceptionHandler. This filter handles all the exceptions raised by controller actions, filters, and views.

To use this feature, first of all turn on the customErrors section in web.config. Open the web.config and place the following code inside system.web and set its value as On.

```
<customErrors mode = "On"/>
```

We already have the Error View created inside the Shared folder under Views. This time change the code of this View file to the following, to strongly-type it with the HandleErrorInfo model (which is present under System.Web.Mvc).

```
@model System.Web.Mvc.HandleErrorInfo

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
    <head>
        <meta name = "viewport" content = "width = device-width" />
        <title>Error</title>
    </head>

    <body>
        <h2>
            Sorry, an error occurred while processing your request.
        </h2>
        <h2>Exception details</h2>

        <p>
            Controller: @Model.ControllerName <br>
            Action: @Model.ActionName
            Exception: @Model.Exception
        </p>

    </body>
</html>
```

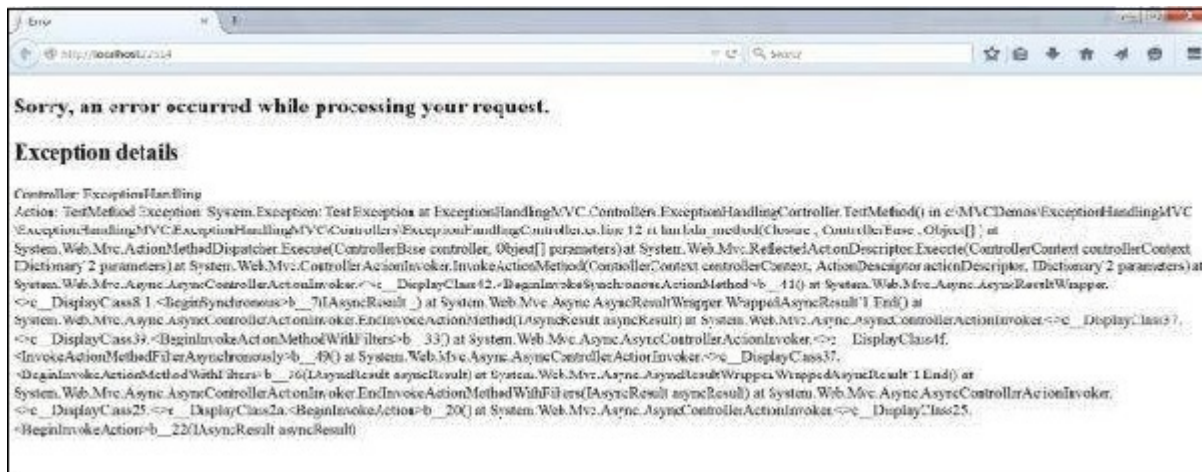
Now place the following code in your controller file which specifies [HandleError] attribute at the Controller file.

```
using System;
using System.Data.Common;
using System.Web.Mvc;

namespace ExceptionHandlingMVC.Controllers {
    [HandleError]
    public class ExceptionHandlingController : Controller {

        public ActionResult TestMethod() {
            throw new Exception("Test Exception");
            return View();
        }
    }
}
```

If you try to run the application now, you will get an error similar to shown in the following screenshot.



As you can see, this time the error contains more information about the Controller and Action related details. In this manner, the `HandleError` can be used at any level and across controllers to handle such errors.

[⬅ Previous Page](#)

Next Page ➞

Advertisements



[FAQ's](#) [Cookies Policy](#) [Contact](#)

© Copyright 2018. All Rights Reserved.