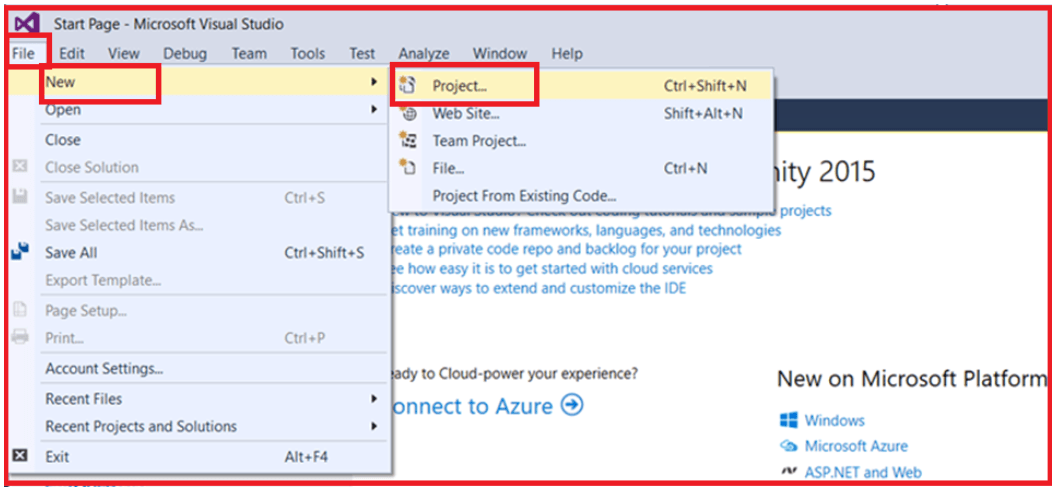
# Creating First ASP.NET MVC Application

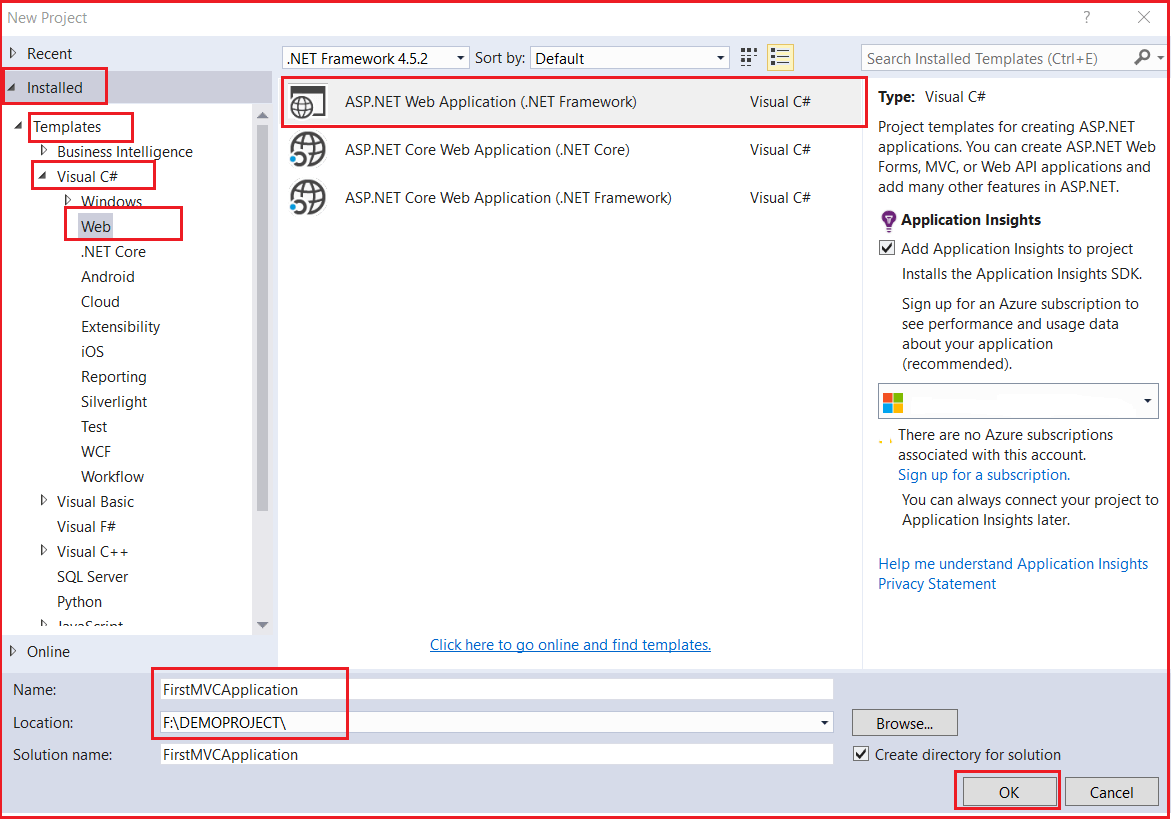
## ****Creating First ASP.NET MVC Application using Visual Studio****

##### ****Creating the first ASP.NET MVC Application:****

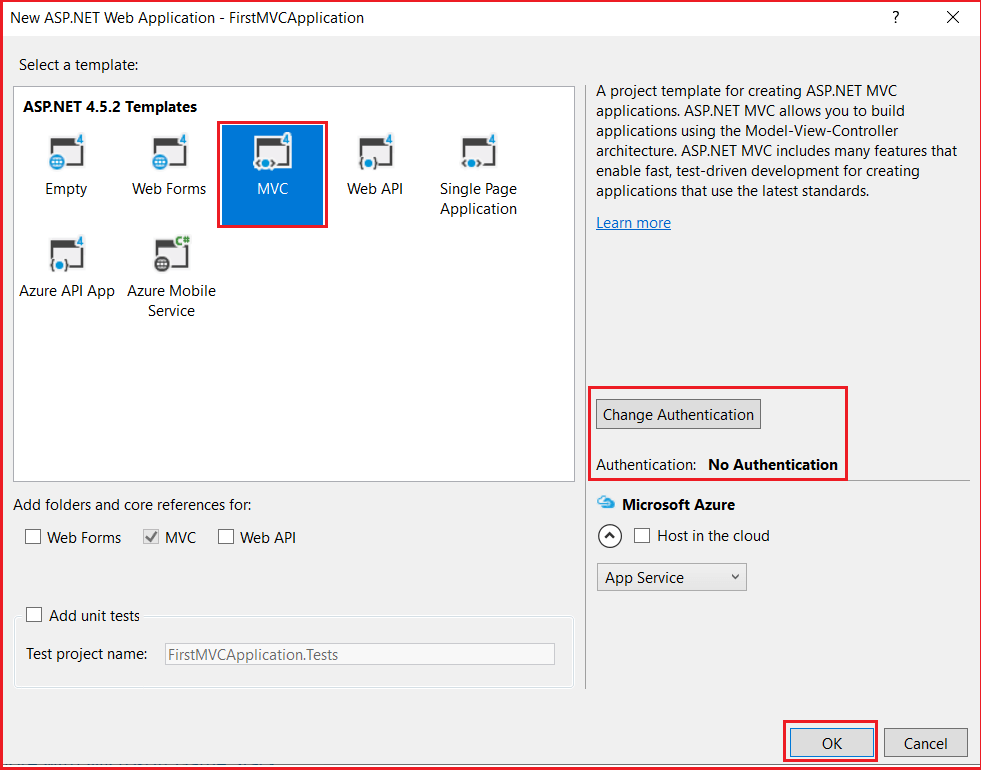
First of all, open the Visual Studio and then click on the **New Project** link which appeared on the **Startup** page. An alternative approach to creating an application is to select the **File =>** **New =>Project**option as shown in the below image.



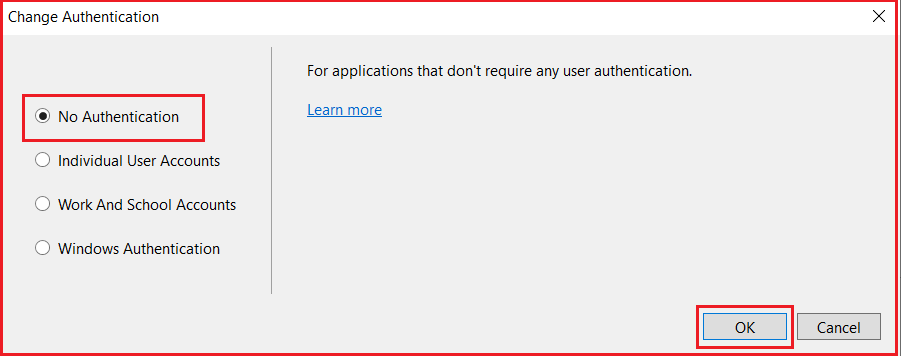
From the **New Project** dialog window, from the left pane select the **Web** option under the **Visual C#** which is under the “**Installed – Templates**” section. From the middle pane select **ASP.NET Web Application**. Name your project as **FirstMVCApplication** (you can give any name to your application). Change the location where you want to create the application by clicking on the **Browse** button. Finally, click on the **OK**button as shown in the image below**.**



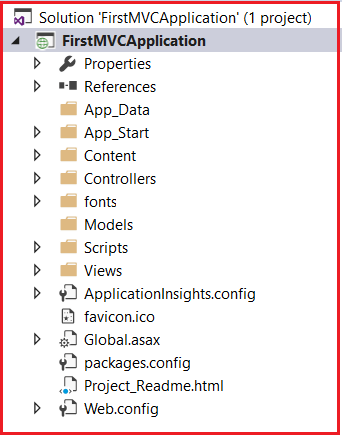
Once you click on the **OK** button, then a new dialog window will open with the name **New ASP.NET Web Application** for selecting the **Project Templates**. From this window select the **MVC** project template. Then Change the Authentication type to **No Authentication.**Finally, click on the **OK** button as shown in the below image.



To change the authentication, you just need to click on the **Change Authentication** button. Then it will open the following popup. From that popup select the appropriate authentication mode. As we don’t want to use any authentication for this demo, so select the **No Authentication** radio button and click on the **OK** button as shown below.

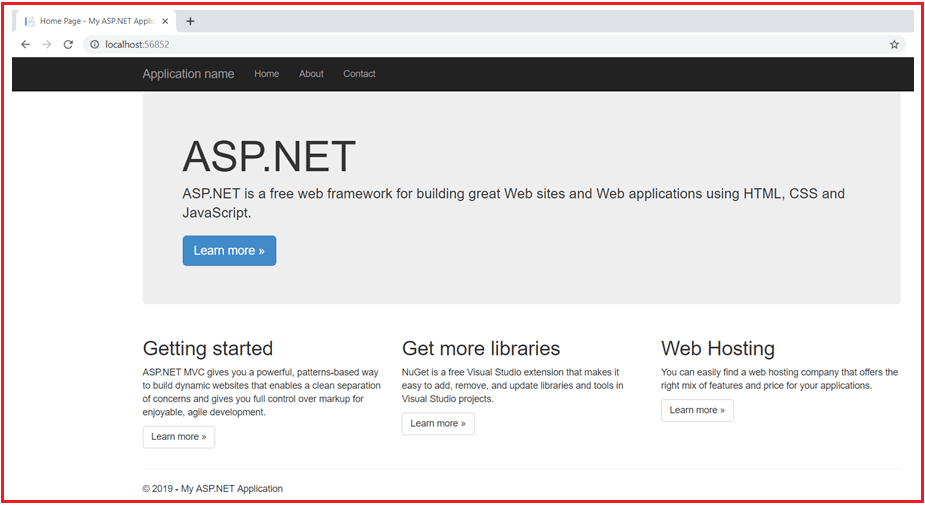


Once you click on the OK button, it will take some time to create the project for us with the following default folder structure.

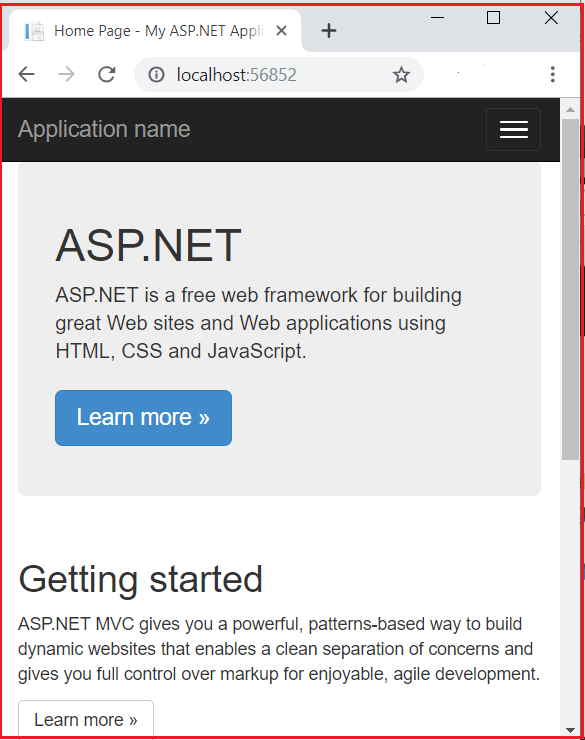


##### ****Running the ASP.NET MVC Application:****

If you want to run the project **with debug** mode then just press **F5**. On the other hand, if you want to run the application **without debugging** then just Press **Ctrl + F5**. Once you run the application, it will open the following page in the browser.



The ASP.NET MVC 5 framework includes the necessary JavaScript and CSS files that are required for bootstrap by default to create a responsive web page. The responsive web page means, the looks and feels of the web page will be changed based on the screen size of the devices on which we are running the application. For example, if you run the application on a mobile device, then the top menu bar will be changed as shown in the below image.



ASP.NET MVC Introduction

**Introduction to ASP.NET MVC Framework**

1. **What is ASP.NET MVC?**
2. **What is MVC?**
3. **How Does the MVC Design pattern Work in ASP.NET MVC Application?**
4. **Advantages of using ASP.NET MVC to develop Web Application**

**What is ASP.NET MVC?**

The ASP.NET MVC is a web application development framework provided by Microsoft which is built on top of the .NET Framework. We can use this ASP.NET MVC Framework to develop web applications that provide a clean separation of code. The ASP.NET MVC framework is the most extensible and customizable framework provided by Microsoft.

The ASP.NET MVC Framework is based on MVC (Model-View-Controller) Design Pattern. So the point that I need to highlight here is ASP.NET MVC is a Framework whereas **MVC is a Design Pattern.**

The ASP.NET MVC Framework is not built from ground zero. You can consider it as an alternative approach to our traditional ASP.NET Web Forms Framework. As it is built on the top of the .NET Framework, developers enjoy almost all the ASP.NET features while working with the MVC application.

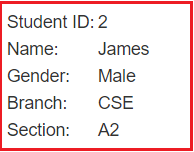
**What is MVC?**

MVC is an architectural software design pattern that is used for developing interactive applications where their user interaction is involved and based on the user interaction some event handling has occurred. It is not only used for web-based applications but it can also be used for Desktop or mobile-based applications where there are user interactions involved.

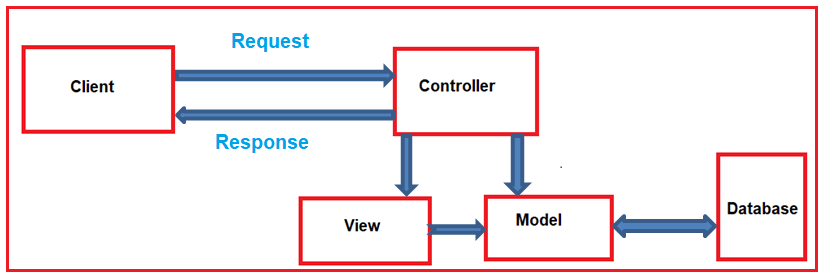
MVC design pattern was introduced in the 1970s that basically divides an application into 3 major components as Model, View, and Controller. The main objective of the MVC design pattern is the separation of concerns (codes), which means the domain model and business logic are separated from the user interface (i.e. view). As a result, maintenance and testing of the application become simpler and easier.

**How does the MVC Design Pattern work in ASP.NET MVC Application?**

Let us understand how does the MVC Design Pattern work in the ASP.NET MVC application with an example. Let say we want to display the student details on a web page as shown in the below image.



So, when the client (user) issues a request something like “**http://xxxxxxxxxx/student/details/2**” from a web browser then the request is handled by the MVC framework as shown below.



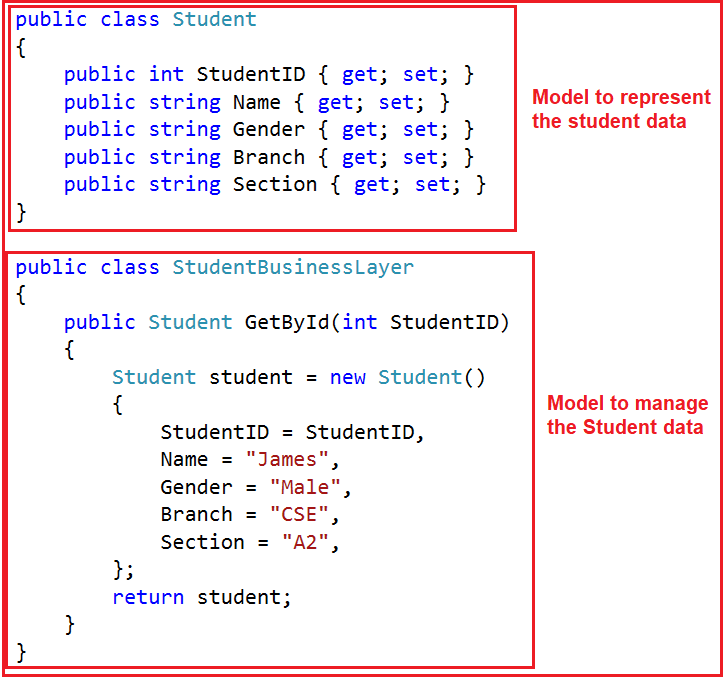
The controller is the component in the ASP.NET MVC application that actually receives the incoming HTTP request and then handles that request. In order to handle the incoming HTTP request, the controller does several things are as follows.

1. The controller creates the model object that is required by a view. The model is the component in the MVC design pattern that contains a set of classes to represent the domain data or business data as well as logic to manage the data.
2. The controller then selects a view to render the domain data or business data. The point that you need to remember is, while selecting a view, it is the responsibility of the controller to pass the model data.
3. In the MVC Design Pattern, the one and only responsibility of a view is to display the model data. So, the responsibility of a view is to generate the necessary HTML which will render the model data or business data. Once the HTML is generated by the view, then that HTML is then sent to the client via the controller who initially made the request.

Now, I hope you understand the basic idea of the MVC Design Pattern. Let us discuss each of the components of the MVC design pattern in detail by comparing it with our example.

**Model**:

The Model is the component in the MVC design pattern that manages that business data or domain data i.e. state of the application in memory. The Model contains a set of classes that represent the data as well as logic to manage the data. So, in our example, the model is consists of Student class to represent the student data as well as StudentBusinessLayer class to retrieve the student data from any persistent medium like a database.



**So in short, a Model:**

1. In ASP.NET MVC is basically a C# or VB.net class to represent the data as well as to manage the data.
2. It is accessible by both controller and view.
3. It can be used to pass data from controller action methods to a view.
4. It can also be used by a view to display data on a page (HTML output).

**View**:

The view is the component in MVC Design Pattern which renders the model data as the user interface with which the end-user can interact. So, the View creates the user interface with data from the model. In our example, we want to display the Student information on a web page. So here the student model carried the student data to the view. This is the student model which should be supplied by the controller to the view. The following code does the same thing.

@model FirstMVCApplication.Models.Student

**<html>**

**<head>**

**<title>**Student Details**</title>**

**</head>**

**<body>**

**<br** **/>**

**<br** **/>**

**<table>**

**<tr>**

**<td>**Student ID: **</td>**

**<td>**@Model.StudentID**</td>**

**</tr>**

**<tr>**

**<td>**Name: **</td>**

**<td>**@Model.Name**</td>**

**</tr>**

**<tr>**

**<td>**Gender: **</td>**

**<td>**@Model.Gender **</td>**

**</tr>**

**<tr>**

**<td>**Branch: **</td>**

**<td>**@Model.Branch**</td>**

**</tr>**

**<tr>**

**<td>**Section: **</td>**

**<td>**@Model.Section **</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

**So in short, a View**

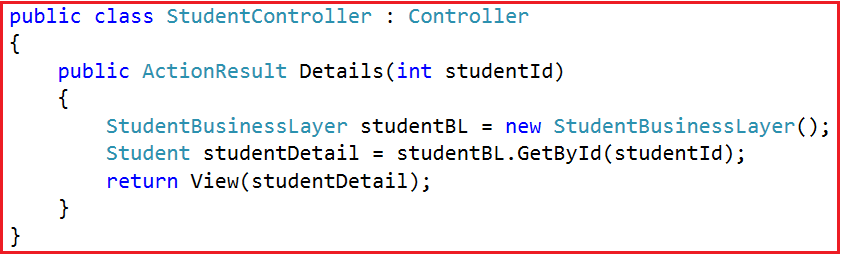
1. In ASP.NET MVC is a cshtml page.
2. It contains all page-specific HTML generation and formatting code.
3. A request to a view can only be made from a controller’s action method.
4. The one and only responsibility of a view is to render the domain data or business data.

**Controller**:

The Controller is the component that contains the control flow logic. It is the one that will interact with both models and views to control the flow of application execution. The controller is the component in MVC Design Pattern that will handle the incoming HTTP Request. Based on the user actions, the respective controller will work with the model and view and then sends the response back to the user who initially made the request. In our example, when the client issued a request to the following URL

**http://xxxxxxxxxxxx/student/details/2**

Then that request is going to be mapped to the Details action method of the Student Controller. Following is the code of our Controller class with the Details action method.



**So, in short, a Controller**:

1. Is basically a C# or VB.NET class that is inherited from the **System.Web.Mvc.Controller**.
2. Is the component which will interact with both Models and views.
3. Contains action methods that are responsible for handling the incoming HTTP Request.
4. Can access and use the model class to pass the data to the views.

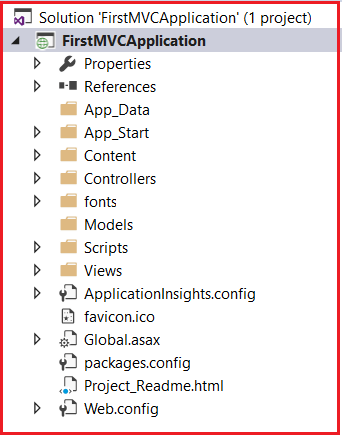
**Advantages of using ASP.NET MVC to develop Web Application**

1. It is lightweight because it does not use view state or server-based forms or server controls.
2. Each developer based on his expertise or experience can work on different parts of the application. For example, one developer may work on the view while the second developer can work on the controller logic and the third developer may work on the business logic.
3. Clean HTML and easy integration with javascript and jQuery.
4. It provides better support for test-driven development (TDD). This is because we can focus on one aspect at a time i.e. we can focus on the view without worrying about business logic.
5. ASP.NET MVC Framework divides the application into three main aspects such as Model, View, and Controller which make it easier to manage the application complexity.
6. Another important advantage of the ASP.NET MVC framework is its components are designed to be extensible and pluggable and therefore they are easily replaced or customized.
7. The MVC framework is built on top of the ASP.NET Framework and hence we can use most of the ASP.NET features such as authentication and authorization scenarios, membership and roles, caching, session, and many more.
8. ASP.NET MVC framework supports a powerful URL routing mechanism (i.e. attribute routing) which helps to build more user-friendly and SEO-friendly URLs for our application.

# ASP.NET MVC File and Folder Structure

## ****ASP.NET MVC File and Folder Structure****

Nowadays, the ASP.NET MVC framework becomes more popular among developers because of the separation of concerns (codes) and folder structure. As a developer, it is very important for you to understand the need and use of each file and folder of an ASP.NET MVC application. When we create an ASP.NET MVC 5 application, Visual Studio by default creates the following files and for our application.



Let us discuss the need and use of each File and Folder in detail one by one.

##### ****App\_Data:****

The App\_Data folder of an ASP.NET MVC application contains application-related data files like .mdf files, LocalDB, and XML files, etc. The most important point that you need to remember is IIS is never going to serve files from this App\_Data folder. In our example, this folder is empty as we don’t have any data files. In our upcoming articles, we will discuss how to store application-related data files in App\_Data Folder.

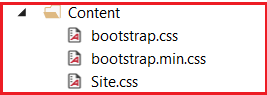
##### ****App\_Start:****

The App\_Start folder of an ASP.NET MVC application contains configuration-related class files which are needed to be executed at the time of application starts. The classes like BundleConfig, FilterConfig, RouteConfig, IdentityConfig, etc are stored within this folder. We will discuss the use of each of these class files in detail in our upcoming articles.



##### ****Content:****

The Content Folder of an ASP.NET MVC application contains static files such as the image files, CSS files, and icons files, etc. When we create a new ASP.NET MVC 5 application, then by default bootstrap.css, Site.css, and bootstrap.min.css files are included by Visual Studio as shown in the image below.



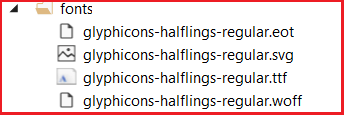
##### ****Controllers:****

The Controllers Folder of an ASP.NET MVC application contains all the controllers of your application. The Controllers are nothing but classes that are inherited from the base Controller class. The name of the Controller should end with the word “Controller”. It is this class that actually handles the user’s request i.e. the incoming HTTP Requests and returns a response. In our upcoming articles, we will discuss the controllers in detail.

Controllers folder in MVC application

##### ****Fonts:****

The Fonts folder of an ASP.NET MVC application contains the custom font files that are required for the application.

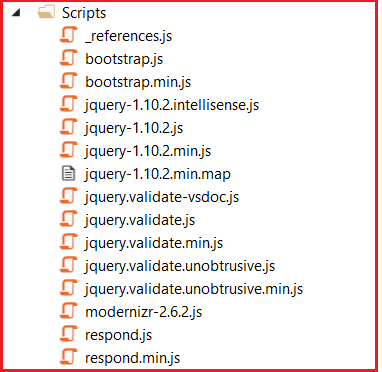


##### ****Models:****

The Models folder of an ASP.NET MVC application contains the class files which are used to store the domain data (you can also say business data) as well as business logic to manage the data. In our example, the Models folder is empty as we have not created any models for our application. In our upcoming articles, we will discuss Models in detail.

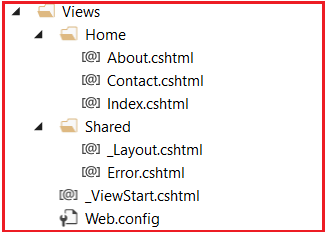
##### ****Scripts:****

The Scripts Folder of an ASP.NET MVC application contains all the JavaScript files that are required for your application. When we create an ASP.NET MVC 5 application, by default the necessary javascript files for jquery and bootstrap are included. If you want to create any custom javascript files then they should be created within this folder or any subfolder of this folder. This is not mandatory but it is a good programming practice as in later time you can easily find out the javascript files.



##### ****Views:****

The Views Folder of an ASP.NET MVC application contains all the “.cshtml” files of your application. In MVC, the .cshtml file is a file where we need to write the HTML code along with the C# code. The Views folder also includes separate folders for each and every controller for your application. For example, all the .cshtml files of the HomeController will be in the View => Home folder. We also have the Shared folder under the Views folder. The Shared Folder contains all the views which are needed to be shared by different controllers e.g. error files,  layout files, etc.



Now, let  us discuss the configuration files which are created by the framework by default:

##### ****Global.asax:****

The Global.asax file in an ASP.NET MVC application allows us to write the code that we want to run at the application level or you can say global level, such as Application\_BeginRequest, Application\_Error, Application\_Start, Session\_Start, Session\_End, etc. In our upcoming articles, we will discuss the use of these application-level events in detail.

##### ****Packages.config:****

The Packages.config file in an ASP.NET MVC application is managed by NuGet Package Manager which will keep track of what packages and versions have been installed in your application.

##### ****Web.config:****

The Web.config file of an ASP.NET MVC application is one of the most useful and important files which contains the application-level configurations such as connection strings, global variables, etc.

Here in this article, we discussed the ASP.NET MVC File and Folder structure. Once you understand the need and use of each folder and file of an MVC application, then it is easy for you to find, store and organize project-related files.

**Controllers in ASP.NET MVC Application**

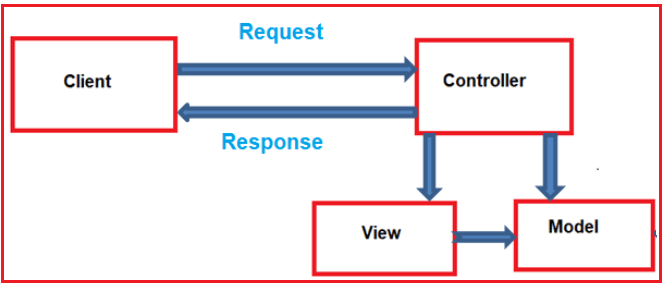
**Controllers in ASP.NET MVC Application**

1. **What is a Controller in the ASP.NET MVC application?**
2. **How to Create a Controller in MVC?**
3. **Understanding Controller with multiple Examples.**

**What is a Controller in an ASP.NET MVC application?**

A controller in an ASP.NET MVC Application is a class having a set of public methods. These public methods of the controller class are called action methods or simple actions. These action methods in the ASP.NET MVC application are going to handle the incoming HTTP Requests.

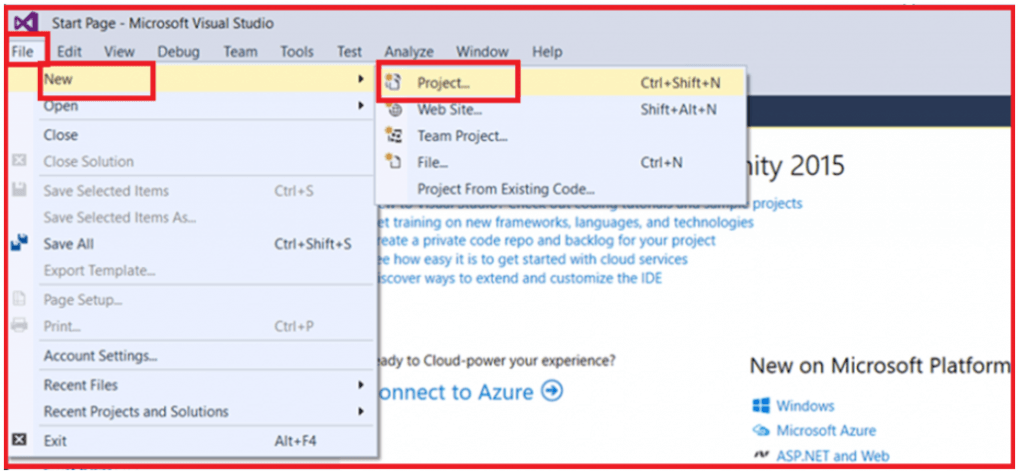
The Controllers in an ASP.NET MVC application logically group similar types of actions together. This grouping of action together allows us to define sets of rules such as routing, caching, and authorization which is going to be applied collectively. Please have a look at the following diagram for a better understanding of the controller.



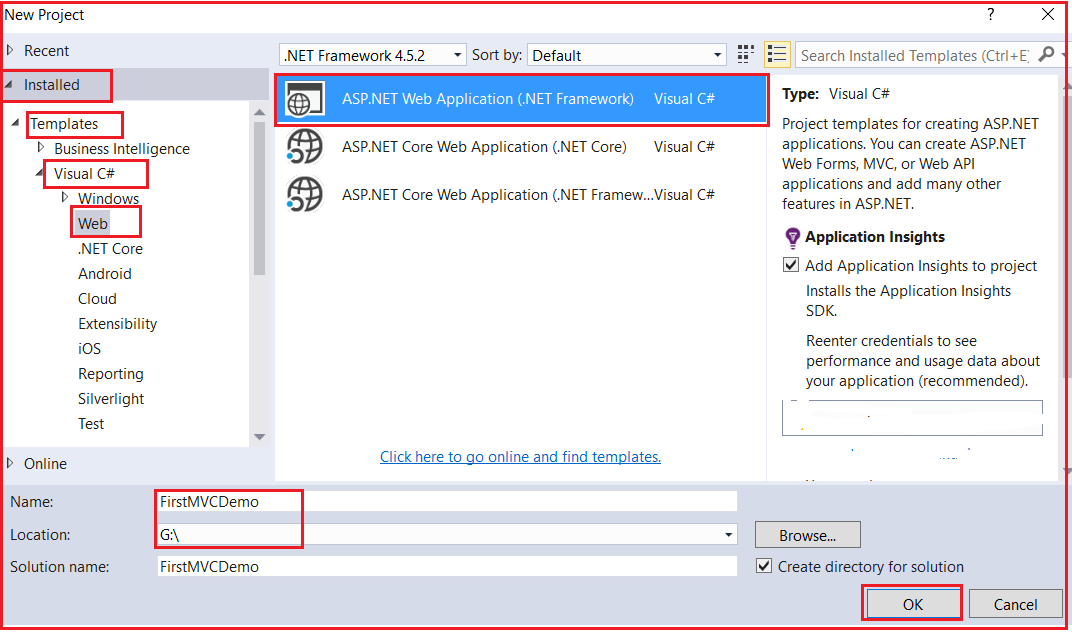
Let’s understand the Controllers in ASP.NET MVC application with an example.

**Creating an Empty ASP.NET MVC application:**

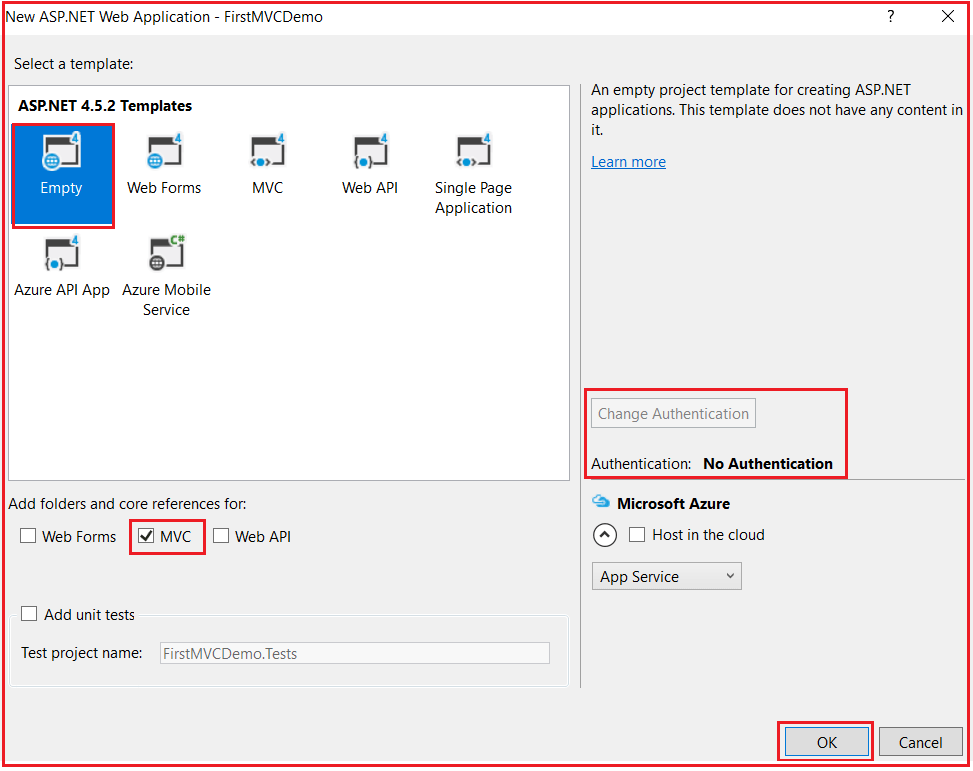
Open Visual Studio and select **File => New => Project** as shown in the image below.



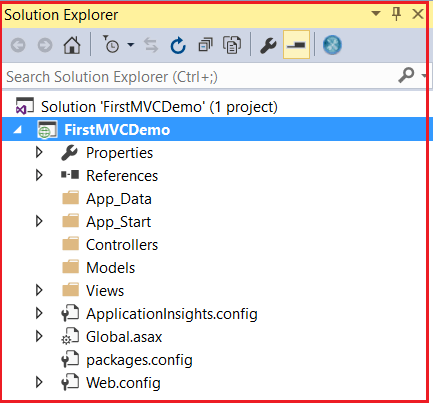
Once you click on the “Project” link a new dialog will pop up. From that New Project window, from the left pane select Web template under the Visual C# which is under the “Installed – Templates” section. From the middle pane select ASP.NET Web Application and give the name of your project as FirstMVCDemo. Finally, click on the OK button as shown in the image below.



Once you click on the OK button, then a new dialog window will open with the name New ASP.NET Web Application for selecting the Project Templates. From this window, we are going to select the Empty project template as we are going to do everything from scratch. Finally, click on the OK button as shown in the below image.



Once we click on the OK button, it will take some time to create the project with the empty template with the following default folder structure.

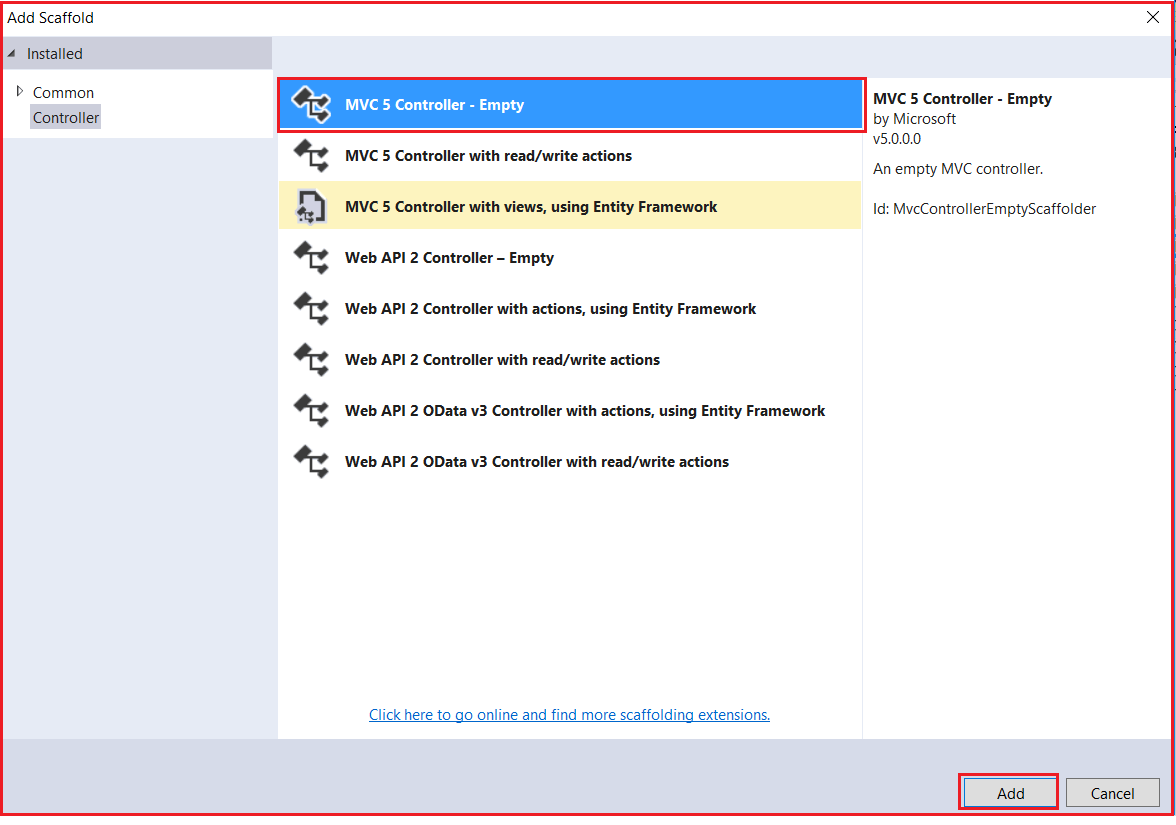


As you can see in the above image, the project is created with several folders such as **Models, Views, Controllers,** etc. As the names suggest these folders are going to contain Models, Views, and Controllers respectively. We will discuss Models and Views in our upcoming article and in this article, we are going to focus on the Controllers.

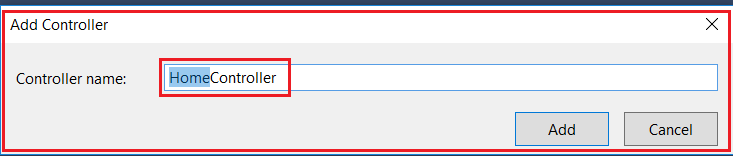
**Adding Controller to the Project:**

To add a controller to your project follow the below steps

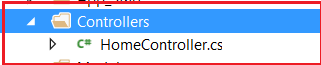
1. Right Click on the **“Controllers”** folder
2. Select **Add** > **Controller**
3. Select **MVC 5 Controller – Empty**
4. Click on the **ADD**button as shown in the image below



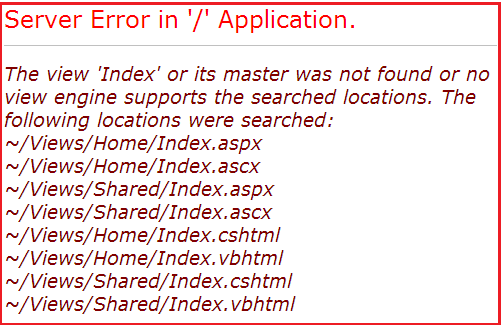
Once you click on the Add button, then a new pop-up window will open where you need to provide a name for your controller. Set the Controller Name as HomeController and click on the Add button as shown in the image below.



Once you click on the Add button, then you should have HomeController.cs within the “Controllers” folder as shown in the below image.



At this point, if you run your application, then you will get the following error.



**Fixing the Error:**

In order to fix the above error, we need to add a view with the name **“Index”**. We will discuss views in detail in our next article. So here we will fix the above issue in another way. The following is the function that is auto-generated by the HomeController class.

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**}**

**}**

To fix the issue, modify the return type of the **Index()** action method from **“ActionResult”** to **“string”.**Then return a string such as **“Hello MVC 5 Application”** as shown below.

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** string Index**()**

**{**

**return** "Hello MVC 5 Application";

**}**

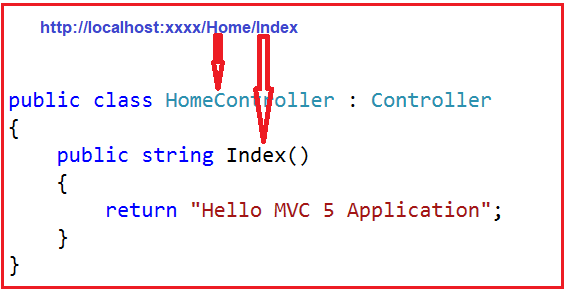
**}**

**}**

With the above changes, run the application and you will see the string as expected in the browser window.

Now change the URL to **http://localhost:xxxx/Home/Index**

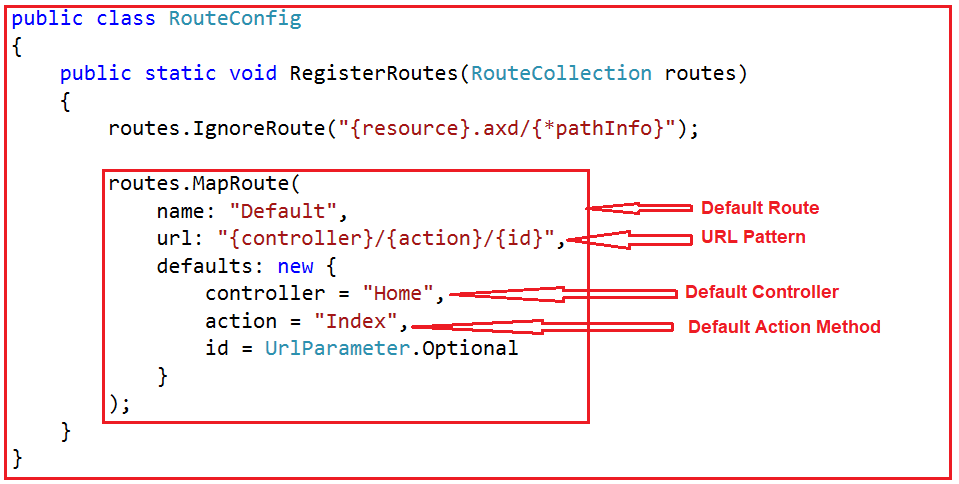
In the URL **“Home”** is the name of the controller and **“Index”** is the name of the action method within the HomeController class. So the important point that you need to keep in mind is the incoming URL in an ASP.NET MVC application is mapped to a controller action method as shown in the below image.



**So the next obvious question that should come to your mind is where this mapping is defined?**

Well, the mapping is defined within the **RegisterRoutes()** method of the **RouteConfig** class. As we already discussed in our [**ASP.NET MVC Folder and File Structure**](https://dotnettutorials.net/lesson/asp-dot-net-mvc-folder-structure/) article,  all the configuration-related files are stored within the **App\_Start** folder. So you can find this **RouteConfig** class within the **App\_Start** Folder.

Now open the **RouteConfig.cs** class file and you will see that the **RegisterRoutes()** method of the **RouteConfig** class has got a default route as shown in the below image.

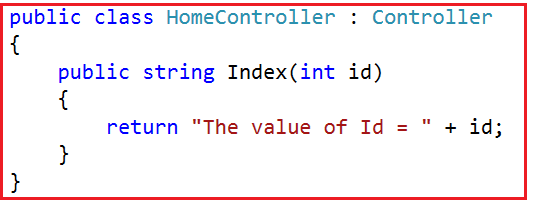


In the URL, we have not specified the id value. But still, it working and we got the output as expected. This is because as you can see in the defaults, the id parameter is declared as optional.

Let pass the id value in the URL and see what happens.

**http://localhost:xxxx/Home/Index/10**

When we navigate to the above URL nothing has happened. Now, change the implementation of the Index() action method of the HomeController class as shown below.



Now when you navigate to “**/Home/Index/10**“, then the value 10 is assigned to the id parameter of the Index action method and hence it will display the following message in the browser.

**The value of Id = 10**

Let change the implementation of the Index action method as shown below to accept the query string parameter.

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** string Index**(**string id, string name**)**

**{**

**return** "The value of Id = " + id + " and Name = " + name;

**}**

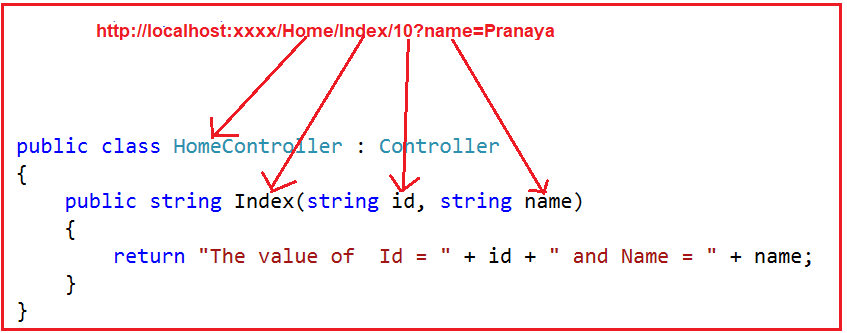
**}**

**}**

Let us issue a request as shown below using the query string.

**http://localhost:xxxx/Home/Index/10?name=James**

In the above URL, the value 10 is assigned to the id parameter and the query string name is assigned to the name parameter of the Index action method. So the mapping is now done as shown in the image below.



In MVC, you can also use the “**Request.QueryString**” to retrieve the query string parameter as shown below.

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** string Index**(**string id, string name**)**

**{**

**return** "Id = " + id + " ,Name = " + Request.QueryString**[**"name"**]**;

**}**

**}**

**}**

**So in short,**

1. A controller in ASP.NET MVC Application is a class that is inherited from **System.Web.Mvc.Controller**.
2. The MVC controller is the one that is going to interact with both Models and views.
3. The controller class contains a set of public methods which are also called the action methods. It is these action methods which is going to handle the incoming HTTP Requests.
4. In ASP.NET MVC, every controller class name must end with the word “**Controller**”. For example, the controller for the home page must be HomeController and the controller for a student must be StudentController.
5. Every controller class must be located in the Controllers folder.

**Views in ASP.NET MVC**

**Views in ASP.NET MVC Application**

1. **What are the Views in ASP.NET MVC?**
2. **Where ASP.NET MVC View Files are Stored?**
3. **How to create Views in ASP.NET MVC Application?**
4. **Understanding Views in MVC with Multiple Examples.**
5. **Advantages of Using Views in MVC**

**What are the Views in ASP.NET MVC?**

In the MVC pattern, the view component contains the logic to represent the model data as a user interface with which the end-user can interact. Typically, it creates the user interface with the data from the model provided to it by the controller. So you can consider the Views in ASP.NET MVC as HTML templates embedded with Razor syntax which generates HTML content that sends to the client.

**Where ASP.NET MVC View Files are Stored?**

In ASP.NET MVC, the views are having a “**.cshtml**” extension when we use C# as the programming language with Razor syntax. Usually, each controller will have its own folder in which the controller-specific .cshtml files (views) are going to be saved. The controller-specific folders are going to be created within the Views folder. The most important point that you need to keep in mind is the view file name and the controller action name are going to be the same.

**Example:**

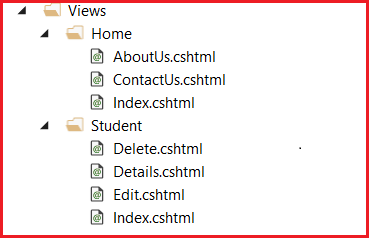
Let’s say, we created an ASP.NET MVC application with two controllers i.e. StudentController and HomeController. The HomeController that we created is having the following three action methods.

1. **AboutUs()**
2. **ContactUs()**
3. **Index()**

Similarly, the StudentController is created with the following four action methods.

1. **Index()**
2. **Details()**
3. **Edit()**
4. **Delete()**

The views are going to be created and saved in the following order.



As we have two controllers in our application, so there are two folders created with the Views folder one per Controller. The Home  Folder is going to contain all the view files (i.e. cshtml files) which are specific to HomeController. Similarly, the Student Folder is going to contain all the .cshtml files which are specific to Student Controller. This is the reason why, the Home folder contains the Index, AboutUs, and ContactUs cshtml files. Similarly, the Student folder contains the Index, Details, Edit, and Delete view files.

**Understanding Views in MVC with Examples:**

To understand the views in the ASP.NET MVC application, let us first modify the HomeController as shown below.

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

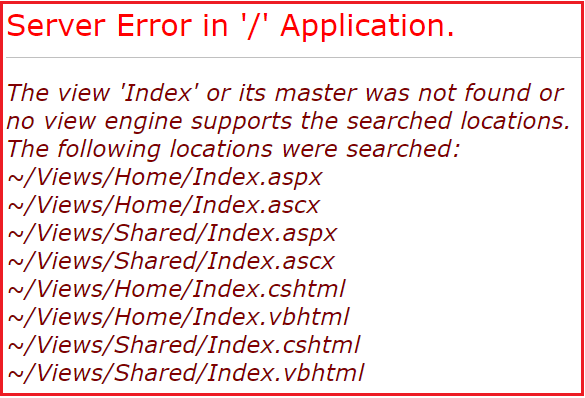
**return** View**()**;

**}**

**}**

**}**

In the above HomeController, we created an Action method that is going to return a view. In order to return a view from an action method in ASP.NET MVC Application, we need to use the **View()** extension method which is provided by **System.Web.Mvc.Controller** Base class. Now run the application and navigate to the **“/Home/Index”** URL and you will get the following error.



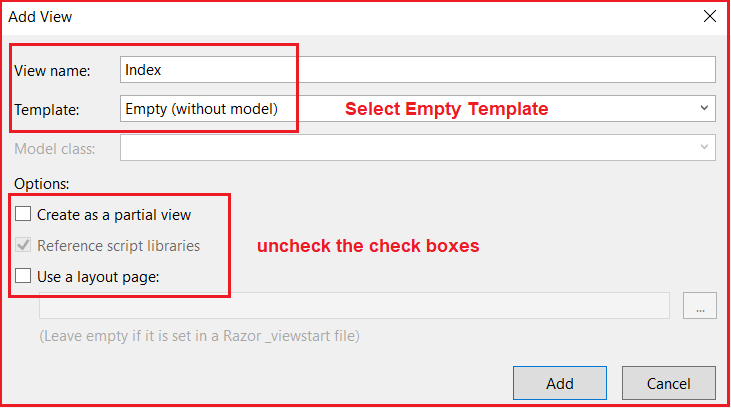
**Let us understand why we got the above error.**

As we are returning a view from the Index action method of Home Controller, by default the MVC Framework will look for a file with the name **Index.aspx or Index.ascx** within the Home and Shared folder of the application if the view engine is APSX. If it is not found there then it will search for a view file with the name **Index.cshtml or Index.vbhtml**within the Home and Shared folder of your application.

If the requested view file is found in any of the above locations, then the View generates the necessary HTML and sends the generated HTML back to the client who initially made the request. On the other hand, if the requested view file is not found in any of the above locations, then we will get the above error.

**Adding Views in ASP.NET MVC**

In order to add the Index view, Right-click anywhere with the Index() function and then click on the “Add View” option which will open the following Add View dialog window. From the Add View window, provide the name for the view as Index, select Template as Empty, uncheck the checkboxes for “create as a partial view” and “use a layout page” option. Finally, click on the Add button as shown below.



Once the Index view is created, then copy and paste the following in it.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Index**</title>**

**</head>**

**<body>**

**<div>**

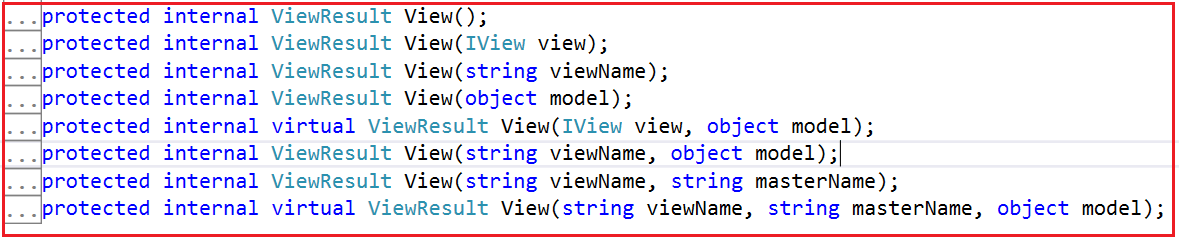
**<h1>**Index View Coming From Views/Home Folder**</h1>**

**</div>**

**</body>**

**</html>**

That’s it. Now run the application and navigates to the “**/Home/Index**” URL and you will see the output as expected. If you go to the definition of **Controller** base class, then you will find there eight overload versions of the View method which return a view as shown below.



Each of the above-overloaded versions we will discuss as we progress through this course.

**Advantages of Using Views in MVC Application:**

The Views in ASP.NET MVC application provides the separation of concerns (codes). It means, it separates the user interface from the rest of the application such as the business layer and data access layer. The ASP.NET MVC views use the advanced Razor syntax which makes it easy to switch between the HTML and C# code. The common or repetitive sections of the web pages can be easily reused by using layout and partial views which we will discuss in our upcoming articles.

**Models in ASP.NET MVC**

**Models in ASP.NET MVC Application**

1. **What are the Models in ASP.NET MVC?**
2. **How to create and use Models in MVC.**

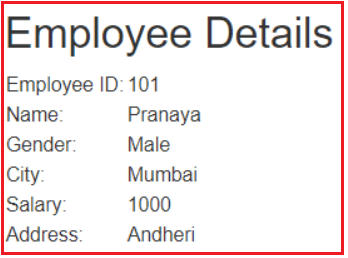
**What are the Models in ASP.NET MVC?**

The Models in ASP.NET MVC application are the component which contains a set of classes that are used to represent the business data (or domain data) as well as logic to manage the business data. So in simple words, we can say that the model in ASP.NET MVC is used to manage the domain data i.e. the state of the application in memory.

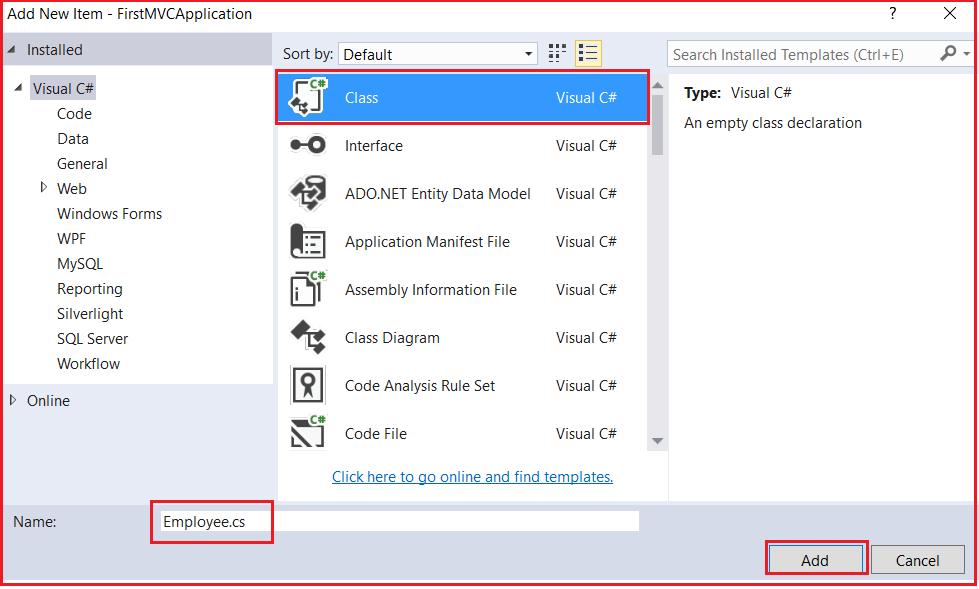
**Note:** It is not mandatory, but it is a good programming practice to store all model classes within the Models folder of an ASP.NET MVC application.

**Let us see an example to understand the Models in ASP.NET MVC.**

We need to display the employee information on a webpage as shown below.



In order to store the employee data, we are going to use the Employee model class. To do so,  right-click on the “Models” folder and then select Add => Class option. Provide the name as Employee.cs and finally click on the Add button as shown in the image below.



**Now open the Employee.cs class file and then copy and paste the following code.**

**namespace** *FirstMVCDemo.Models*

**{**

**public** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Address **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **decimal** Salary **{** **get**; **set**; **}**

**}**

**}**

This is our Employee model which is going to hold the employee data in memory. As we already discussed, the Models in ASP.NET MVC Framework also contain the business logic to manage the business data. So in our example, in order to manage the employee data i.e. to perform the CRUD operation on the employee data, we are going to use the following EmployeeBusinessLayer model.

**Creating EmployeeBusinessLayer Model:**

Right-click on the Models folder and then add a new class file with the name EmployeeBusinessLayer.cs. Once you create the EmployeeBusinessLayer class file, then copy and paste the following code into it.

**namespace** *FirstMVCDemo.Models*

**{**

**public** **class** EmployeeBusinessLayer

**{**

**public** Employee GetEmployeeDetails**(int** EmployeeId**)**

**{**

//Here we hardcoded the data

//later we will discuss how to retrieve

//the data from a database

Employee employee = new Employee**()**

**{**

EmployeeId = EmployeeId,

Name = "Pranaya",

Gender = "Male",

City = "Mumbai",

Salary = 1000,

Address = "Andheri"

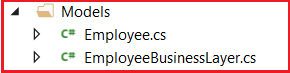
**}**;

**return** employee;

**}**

**}**

**}**

Once you created the required models for your application, then the model folder structure should look like below.

**Modifying the HomeController**

Now let us modify the HomeController class as shown below to use the Employee and EmployeeBusinessLayer model to retrieve the employee data.

**using** *FirstMVCDemo.Models;*

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**(int** id**)**

**{**

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer**()**;

Employee employee = employeeBL.GetEmployeeDetails**(**id**)**;

**return** View**()**;

**}**

**}**

**}**

That’s it. In the next article, we will discuss how to pass the employee model data to a view, so that the view generates the necessary HTML.

# ViewData in ASP.NET MVC

## ****ViewData in ASP.NET MVC Application****

1. **What is ViewData in ASP.NET MVC?**
2. **How to Pass and Retrieve data From ViewData in ASP.NET MVC?**
3. **Example of ViewData in ASP.NET MVC.**

In the ASP.NET MVC application, we can pass the model data from a controller to a view in many ways such as by using ViewBag, ViewData, TempData, Session, and Application as well as you can also use strongly typed views. You can also use the Session and Application State variable as we use in our traditional Web Forms to manage the data during a user session or throughout the application.

Now the most important question that comes to your mind is when to use ViewData, ViewBag, TempData, Session, and Application as each one having its own advantages and disadvantages. As we progress through this course you will come to know when to use one over another. Here in this article, I will show you how to use ViewData to pass the data from a controller action method to a view.

##### ****What is ViewData in ASP.NET MVC?****

The ViewData in ASP.NET MVC Framework is a mechanism to pass the data from a controller action method to a view. If you go to the definition of ViewData by right-clicking on it and select go to definition, then you will see that ViewData is defined as a property in the ConstrollerBase class and its type is ViewDataDictionary as shown in the below image.

signature of the ViewData in MVC

As you can see in the above image, the return type of ViewData is ViewDataDictionary. Let’s have a look at the definition of the ViewDataDictionary class.

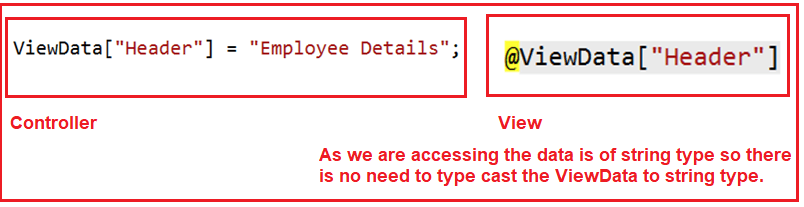
Definition of ViewDataDictionary class in ASP.NET MVC

As you can see, the ViewDataDictionary class implements the IDictionary interface. So we can say that the **ViewData in ASP.NET MVC** Framework is a dictionary object. As it is a dictionary object, so it is going to store the data in the form of **key-value pairs** where each **key must be a string** and the value that we are passing to the dictionary is going to be stored in the form of an **object type**.

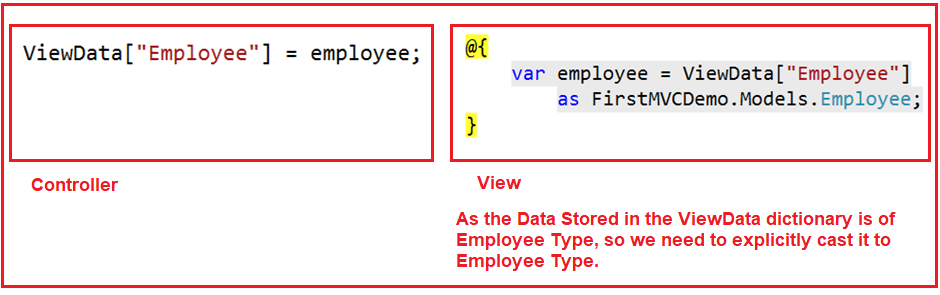
##### ****How to Pass and Retrieve data From ViewData in ASP.NET MVC?****

The most important point that you need to remember is, as it stores the data in the form of an object, so while retrieving the data from ViewData type casting is required. If you are accessing string data from the ViewData, then it is not required to typecast the ViewData to string type. But it is mandatory to typecast explicitly to the actual type if you are accessing data other than the string type.

###### **ViewData in ASP.NET MVC with String Type:**

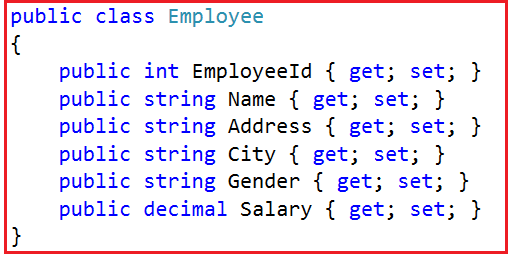


###### **ViewData in ASP.NET MVC with Complex Type:**

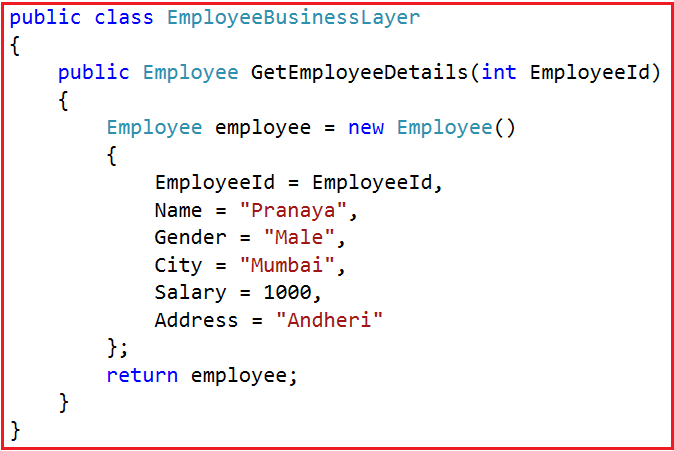


##### ****Example of ViewData in ASP.NET MVC Application:****

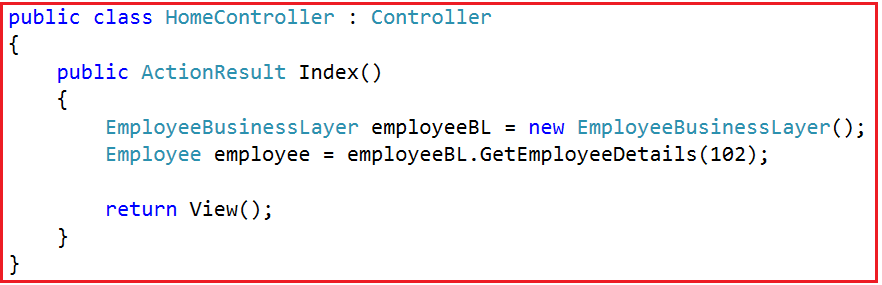
Let us see an example to understand how to use the ViewData to pass data from a controller action method to a view. Please read our previous article as we are going to work with the same example. Let us first recap of what we did in our previous article. First, we create the following Employee Model to hold the employee data in memory.



Then we created the following EmployeeBusinessLayer model to manage the employee data. Here we created one method which will take the employee id as input parameter and returns that employee information. As of now, we have hardcoded the employee data and in our upcoming article, we will discuss retrieving the employee data from a database like SQL Server, MySQL, Oracle, etc.



Then we modify the Index action method of Home Controller as shown below to retrieve the employee data from EmployeeBusinesslayer and store it in the Employee model.



##### ****Passing ViewData From a Controller Action Method to a View:****

Now we will see, how to use the ViewData to pass the employee object to the Index view. Along with we are also going to pass the page Header using ViewData. So, modify the Index action method of the Home Controller class as shown below.

**using** *FirstMVCDemo.Models;*

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer**()**;

Employee employee = employeeBL.GetEmployeeDetails**(**102**)**;

ViewData**[**"Employee"**]** = employee;

ViewData**[**"Header"**]** = "Employee Details";

**return** View**()**;

**}**

**}**

**}**

##### ****Accessing ViewData in a View:****

Now we will see how to access the ViewData within an ASP.NET MVC view. So, modify the Index Action method which is there within the Home folder in your application as shown below.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Page Title**</title>**

**</head>**

**<body>**

@{

var employee = ViewData["Employee"]

as FirstMVCDemo.Models.Employee;

}

**<h2>**@ViewData["Header"]**</h2>**

**<table** style="font-family:Arial"**>**

**<tr>**

**<td>**Employee ID:**</td>**

**<td>**@employee.EmployeeId **</td>**

**</tr>**

**<tr>**

**<td>**Name:**</td>**

**<td>**@employee.Name**</td>**

**</tr>**

**<tr>**

**<td>**Gender:**</td>**

**<td>**@employee.Gender**</td>**

**</tr>**

**<tr>**

**<td>**City:**</td>**

**<td>**@employee.City**</td>**

**</tr>**

**<tr>**

**<td>**Salary:**</td>**

**<td>**@employee.Salary**</td>**

**</tr>**

**<tr>**

**<td>**Address:**</td>**

**<td>**@employee.Address**</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

That’s it. Now run the application and you will see the employee details on the webpage as expected.

The ViewData in MVC is resolved dynamically at runtime. As a result, it does not provide compile-time error checking as well as we will not get the intelligence support. For example, if we miss-spell the key names then we wouldn’t get any compile-time error rather we came to know the error at runtime.

The ViewData in ASP.NET MVC can only transfer the data from a controller action method to a view. That means it is valid only during the current request.

# ViewBag in ASP.NET MVC

## ****ViewBag in ASP.NET MVC Application****

1. **What is ViewBag in ASP.NET MVC?**
2. **How to Pass and Retrieve data From ViewBag in ASP.NET MVC?**
3. **Example of ViewBag in MVC.**
4. **What are the Difference and Similarities between ViewData and ViewBag in ASP.NET MVC?**

As we already discussed we can pass the data from a controller action method to a view using ViewData, ViewBag, TempData, and strongly typed model. Here in this article, I am going to show you how to use ViewBag to pass the data from a controller action method to a view.

##### ****What is ViewBag in ASP.NET MVC?****

The ViewBag in ASP.NET MVC Framework is one of the mechanisms to pass the data from a controller action method to a view. If you go to the definition, of ViewBag, then you will see that it is defined as a property in the ControllerBase class with the following signature.

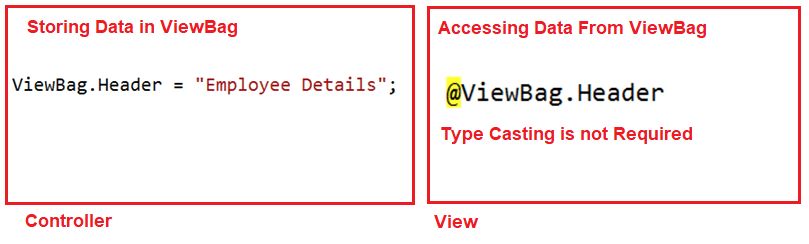
ViewBag in ASP.NET MVC Application

As you can see in the above image, the ViewBag is a dynamic property (a new feature introduced in C# 4.0). Dynamic data type means at runtime based on the value, it will decide the data type. The ViewBag is also like ViewData which also transfers the data from a controller action method to a view.

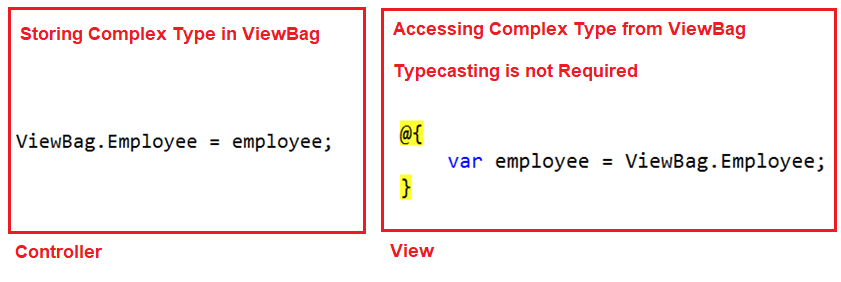
##### ****How to Pass and Retrieve data From ViewBag in ASP.NET MVC?****

As the ViewBag is operating on the new dynamic data type. The advantage is that we do not require typecasting while accessing the data from a ViewBag irrespective of the data that we are accessing.

###### **ViewBag in ASP.NET MVC with String Type:**



###### **ViewBag in ASP.NET MVC with Complex Type:**



##### ****Example of ViewBag in MVC Application:****

Let us see an example to understand how to use the new dynamic type ViewBag in ASP.NET MVC Application to pass data from a controller action method to a view. We are going to work with the same example that we worked on in our previous article with ViewData. So, modify the Index action method of HomeController class as shown below.

**using** *FirstMVCDemo.Models;*

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer**()**;

Employee employee = employeeBL.GetEmployeeDetails**(**101**)**;

ViewBag.Employee = employee;

ViewBag.Header = "Employee Details";

**return** View**()**;

**}**

**}**

**}**

As you can see in the above example, here we are using ViewBag to pass the data.

##### ****Accessing the ViewBag in a View in ASP.NET MVC****

Now we will see how to access the ViewBag data within an ASP.NET MVC view. So, modify the **Index.cshtml** view file as shown below.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Page Title**</title>**

**</head>**

**<body>**

@{

var employee = ViewBag.Employee;

}

**<h2>**@ViewBag.Header**</h2>**

**<table** style="font-family:Arial"**>**

**<tr>**

**<td>**Employee ID:**</td>**

**<td>**@employee.EmployeeId **</td>**

**</tr>**

**<tr>**

**<td>**Name:**</td>**

**<td>**@employee.Name**</td>**

**</tr>**

**<tr>**

**<td>**Gender:**</td>**

**<td>**@employee.Gender**</td>**

**</tr>**

**<tr>**

**<td>**City:**</td>**

**<td>**@employee.City**</td>**

**</tr>**

**<tr>**

**<td>**Salary:**</td>**

**<td>**@employee.Salary**</td>**

**</tr>**

**<tr>**

**<td>**Address:**</td>**

**<td>**@employee.Address**</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

As you can see, here we are accessing the data from the ViewBag using the same dynamic properties Header and Employee. Now run the application and navigate to the “**/Home/Index**” URL and you will see the data as expected on the webpage.

**Note:**The ViewBag is a dynamic property that is also resolved at runtime like ViewData; as a result, here also it will not provide compile-time error checking as well as intelligence support. For example, if we miss-spell the property names of the ViewBag, then we wouldn’t get any compile-time error rather we came to know about the error at runtime.

Internally, ViewBag is a wrapper around ViewData. So, it will throw a runtime exception if the ViewBag property name matches the key of ViewData.

##### ****Difference and Similarities between ViewData and ViewBag in ASP.NET MVC****

1. In ASP.NET MVC, we can use both ViewData and ViewBag to pass the data from a Controller action method to a View.
2. The ViewData is a dictionary object whereas the ViewBag is a dynamic property. Both ViewData and ViewBag are used to create loosely typed views in ASP.NET MVC.
3. In ViewData, we use the string as the key to store and retrieve the data whereas in ViewBag we use the dynamic properties to store and retrieve data.
4. The ViewData requires typecasting for complex data types and also checks for null values to avoid any exceptions whereas ViewBag doesn’t require any typecasting for the complex data type.
5. Both ViewData keys and ViewBag dynamic properties are resolved only at runtime. As a result, both do not provide compile-time error checking and because of this, we will not get any intelligence support.
6. So if we misspell the key names or dynamic property names then we will not get any compile-time error rather we came to know about the error only at run time. This is the reason why we rarely used ViewBag and ViewData in our application.

The best and preferred approach in ASP.NET MVC Framework to pass data from a controller action method to a view is by using a strongly typed model. The strongly typed models provide compile-time error checking which in turn provides us the intelligence support.

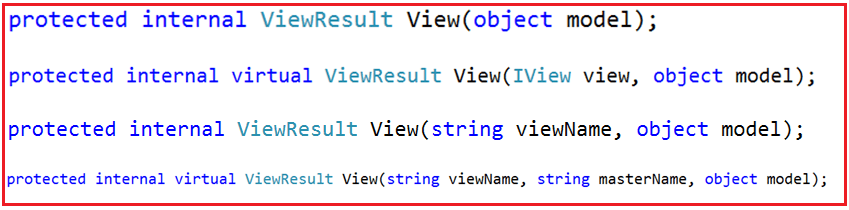
# Strongly Typed Views in ASP.NET MVC

## ****Strongly Typed Views in ASP.NET MVC Application****

In ASP.NET MVC, we can pass the data from the controller action method to a view in many different ways like ViewBag, ViewData, TempData and strongly typed model object. If we pass the data to a View using ViewBag, TempData, or ViewData, then that view becomes a loosely typed view. Here we will discuss how to create a strongly typed view in the ASP.NET MVC application.

##### ****Creating Strongly Typed View in MVC****

In order to create a strongly typed view in ASP.NET MVC application, we need to pass the model object as a parameter to the View() extension method. The Controller base class provide us the following four overloaded versions of View() extension method which we can use to pass the model data from the controller action method to a view.



We are going to use the overloaded version which takes only the model object as the input parameter. As the input parameter is of object type, so we can pass any data. Modify the Index action method of the Home Controller as shown below to pass the Employee object as a parameter to the View extension method.

**using** *FirstMVCDemo.Models;*

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer**()**;

Employee employee = employeeBL.GetEmployeeDetails**(**101**)**;

ViewBag.Header = "Employee Details";

**return** View**(**employee**)**;

**}**

**}**

**}**

##### ****Changes in Index.cshtml View:****

In order to create a strongly typed view in ASP.NET MVC Application, we need to specify the model type within the view by using the **@model** directive. As here, the Employee class is going to be the model so we need to specify the model directive as shown below.

**@model FirstMVCDemo.Models.Employee**

The above statement will tell that we are going to use **FirstMVCDemo.Models.Employee** as the model for this view. Here in the directive (**@model**), the letter **m**is in lowercase and the statement should not be terminated with the semicolon.

Then we can access the model properties simply by using **@Model**, here the letter **M** is in uppercase. So, in our example, we can access the Employee object properties such as Name, Gender, City, Salary, etc. by using **@Model.Name, @Model.Gender, @Model.City, and @Model.Salary** respectively.

###### **So Modify the Index.cshtml view file as shown below to make the view as strongly typed.**

@model FirstMVCDemo.Models.Employee

<!DOCTYPE html>

**<html>**

**<head>**

**<meta** name="viewport" content="width=device-width" **/>**

**<title>**Page Title**</title>**

**</head>**

**<body>**

**<h2>**@ViewBag.Header**</h2>**

**<table** style="font-family:Arial"**>**

**<tr>**

**<td>**Employee ID:**</td>**

**<td>**@Model.EmployeeId **</td>**

**</tr>**

**<tr>**

**<td>**Name:**</td>**

**<td>**@Model.Name**</td>**

**</tr>**

**<tr>**

**<td>**Gender:**</td>**

**<td>**@Model.Gender**</td>**

**</tr>**

**<tr>**

**<td>**City:**</td>**

**<td>**@Model.City**</td>**

**</tr>**

**<tr>**

**<td>**Salary:**</td>**

**<td>**@Model.Salary**</td>**

**</tr>**

**<tr>**

**<td>**Address:**</td>**

**<td>**@Model.Address**</td>**

**</tr>**

**</table>**

**</body>**

**</html>**

That’s it. Now run the application and navigate to the “**/Home/Index**” URL and you will see the employee data as expected in the webpage.

##### ****Advantages of using Strongly Typed View in ASP.NET MVC Application:****

We will get the following advantages when we use strongly typed views in the ASP.NET MVC application.

1. Strongly Typed View in ASP.NET MVC provides compile-time error checkingas well as intelligence support.
2. If we misspell the property name, then it comes to know at compile time rather than at runtime.

In our example, we are still using ViewBag to pass the Header from the Controller to the View. Then the question that should come to your mind is how we will pass the Header to a strongly typed view without using ViewBag. Well, we can do this by using the View Model in the ASP.NET MVC application.

**ViewModel in ASP.NET MVC**

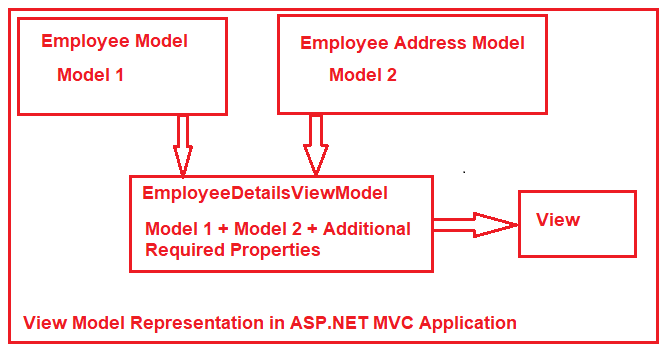
**ViewModel in ASP.NET MVC Application**

**What is a ViewModel in ASP.NET MVC?**

In an ASP.NET MVC application, a single model object may not contain all the necessary data required for a view. For example, a view may require different model data. Then in such situations like this, we need to use the concept ViewModel. A ViewModel in ASP.NET MVC application is a model which contains more than one model data required for a particular view. As this model is specific for a particular view, we call this ViewModel in ASP.NET MVC.

**Understanding ViewModel in ASP.NET MVC**:

Let us have a look at the following diagram which shows the visual representation of a ViewModel in the MVC application.



Let say we want to display the employee details on a webpage. And in our application, we have two different models to represent the employee data. The Employee Model is used to represent the basic details of an employee whereas the Employee Address model is used to represent the employee address.

Along with the above two models to represent the employee data, we also required some static information like page header and title in the view. In order to achieve this, here we need to create a view model such as EmployeeDetailsViewModel. It is this view model which is going to contain both Employee and Employee Address models as well as properties to store the title and header of the web page.

**Creating the Required Models:**

First, create a class file with the name Employee.cs within the Models folder. The Employee model is going to represent the basic information such as name, gender, department, etc. Once you create the Employee.cs class file, then copy and paste the below code in it.

**namespace** *FirstMVCDemo.Models*

**{**

**public** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **decimal** Salary **{** **get**; **set**; **}**

**public** **int** AddressId **{** **get**; **set**; **}**

**}**

**}**

Now, we need to create the Address model to represent the employee Address such as City, State, Country, etc. So, create a class file with the name Address.cs within the Models folder and then copy and paste the following code in it.

**namespace** *FirstMVCDemo.Models*

**{**

**public** **class** Address

**{**

**public** **int** AddressId **{** **get**; **set**; **}**

**public** string Country **{** **get**; **set**; **}**

**public** string State **{** **get**; **set**; **}**

**public** string City **{** **get**; **set**; **}**

**public** string Pin **{** **get**; **set**; **}**

**}**

**}**

**Creating the ViewModel in ASP.NET MVC:**

Now it’s time to create the required View Model to store the required data which is required for a particular view. The View Model that we are going to create will represent the Employee Model + Employee Address Model + Some additional properties like title and page header.

In the ASP.NET MVC application, you can create the View Models anywhere within your project, but it is always a good programming practice to create all the View Models within a special folder called ViewModels.

So first create a folder with the name ViewModels and then create a class file with the name EmployeeDetailsViewModel.cs within the ViewModels folder. Then copy and paste the following code into it.

**using** *FirstMVCDemo.Models;*

**namespace** *FirstMVCDemo.ViewModels*

**{**

**public** **class** EmployeeDetailsViewModel

**{**

**public** Employee Employee **{** **get**; **set**; **}**

**public** Address Address **{** **get**; **set**; **}**

**public** string PageTitle **{** **get**; **set**; **}**

**public** string PageHeader **{** **get**; **set**; **}**

**}**

**}**

Here we created the view model class with the name as EmployeeDetailsViewModel. Here the word Employee represents the Controller name, the word Details represent the action method name. As it is a view model so we prefixed the word ViewModel. Although it is not mandatory to follow this naming convention, I personally feel it is good to follow this naming convention.

**Creating Employee Controller:**

Right-click on the Controllers folder and then add a new MVC 5 Empty controller with the name EmployeeController.cs and then copy and paste the following code in it.

**using** *FirstMVCDemo.ViewModels;*

**using** *FirstMVCDemo.Models;*

**using** *System.Web.Mvc;*

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ViewResult Details**()**

**{**

//Employee Basic Details

Employee employee = new Employee**()**

**{**

EmployeeId = 101,

Name = "Dillip",

Gender = "Male",

Department = "IT",

Salary = 10000,

AddressId = 1001

**}**;

//Employee Address

Address address = new Address**()**

**{**

AddressId = 1001,

City = "Bhubaneswar",

State = "Odisha",

Country = "India",

Pin = "755019"

**}**;

//Creating the View model

EmployeeDetailsViewModel employeeDetailsViewModel = new EmployeeDetailsViewModel**()**

**{**

Employee = employee,

Address = address,

PageTitle = "Employee Details Page",

PageHeader = "Employee Details",

**}**;

//Pass the employeeDetailsViewModel to the view

**return** View**(**employeeDetailsViewModel**)**;

**}**

**}**

**}**

As you can see in the above code, here we are passing the employee details view model as a parameter to the view. And one more thing you need to notice is that now we are not using any ViewData or ViewBag within our Details action method.

**Creating the Details View:**

First, add a folder with the name Employee within the Views folder of your application. Once you add the Employee Folder, then you need to add a view file with the name Details.cshtml within the Employee folder and then copy and paste the following code in it.

@model FirstMVCDemo.ViewModels.EmployeeDetailsViewModel

@{

Layout = null;

}

**<html** xmlns="http://www.w3.org/1999/xhtml"**>**

**<head>**

**<title>**@Model.PageTitle**</title>**

**</head>**

**<body>**

**<h1>**@Model.PageHeader**</h1>**

**<div>**

EmployeeID : @Model.Employee.EmployeeId

**</div>**

**<div>**

Name : @Model.Employee.Name

**</div>**

**<div>**

Gender : @Model.Employee.Gender

**</div>**

**<div>**

Department : @Model.Employee.Department

**</div>**

**<div>**

Salary : @Model.Employee.Salary

**</div>**

**<h1>**Employee Address**</h1>**

**<div>**

City : @Model.Address.City

**</div>**

**<div>**

State : @Model.Address.State

**</div>**

**<div>**

Country : @Model.Address.Country

**</div>**

**<div>**

Pin : @Model.Address.Pin

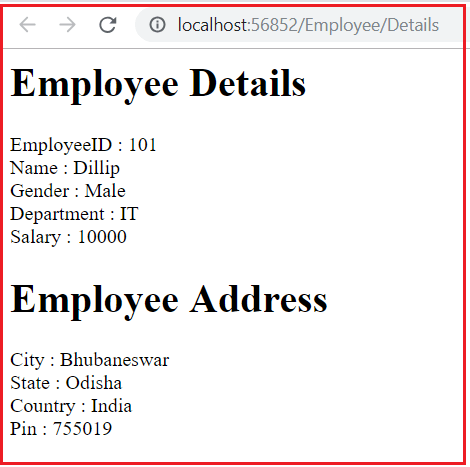
**</div>**

**</body>**

**</html>**

Now, the Details view has access to the EmployeeDetailsViewModel object. By using the **@model** directive, we set EmployeeDetailsViewModel as the Model for our Details view. Then we access Employee, Address, PageTitle, and PageHeader properties using the **@Model** property.

Now run the application, and navigate to the “/Employee/Details” URL and you will see the output as expected as shown in the below image.



**TempData in ASP.NET MVC**

**TempData in ASP.NET MVC Application**

1. **Why do we need TempData in the ASP.NET MVC Application?**
2. **What exactly is TempData in MVC?**
3. **How to use TempData in MVC?**
4. **How to pass and retrieve data from TempData in MVC Application?**
5. **How to retain TempData values in the consecutive request?**

**Why do we need TempData in the ASP.NET MVC Application?**

As we already discussed in our previous articles, we can use ViewData, ViewBag, and strongly typed models to pass the data from a controller action method to a view. Now, we will see another approach to send the data from the controller action method to a view using the TempData.

The limitation of both ViewData and ViewBag is they are limited to one HTTP request only. So, if redirection occurs then their values become null means they will lose the data they hold. In many real-time scenarios, we may need to pass the data from one HTTP Request to the next subsequent HTTP Request. For example, we may need to pass the data from one controller to another controller or one action method to another action method within the same controller. Then in such situations like this, we need to use TempData.

**What is TempData in ASP.NET MVC?**

The TempData in ASP.NET MVC Framework is one of the mechanisms to pass a small amount of temporary data from a controller action method to a view as well as from a controller action method to another action method either within the same controller or to a different controller. TempData value will become null once the subsequent request is completed by default. But if you want then you can also change this default behavior. If you have a look at the definition Controller class, then you will find the following signature of the TempData property.

TempData in ASP.NET MVC

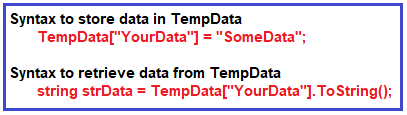
As you can see in the above image, the return type of the TempData is TempDataDictionary. Let us see the definition of the TempDataDictionary class.

TempDataDictionary in MVC

As you can see the TempDataDictionary class implements the IDictionary interface. So we can say that the TempData in ASP.NET MVC is a dictionary object. As it is a dictionary object, so it is going to store the data in the form of key-value pairs where each key must be a string and the value that we are passing to the dictionary is going to be stored in the form of an object type.

**How to Pass and Retrieve data From TempData in ASP.NET MVC:**

The most important point that you need to remember is, as it stores the data in the form of an object so while retrieving the data from TempData type casting is required. If you are accessing string value from the TempData, then it is not required to typecast. But it is mandatory to typecast explicitly to the actual type if you are accessing data other than the string type from the TempData.



**Let us understand TempData in ASP.NET MVC with one example.**

Modify the Employee Controller as shown below

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Method1**()**

**{**

TempData**[**"Name"**]** = "Pranaya";

TempData**[**"Age"**]** = 30;

**return** View**()**;

**}**

**public** ActionResult Method2**()**

**{**

string Name;

**int** Age;

**if** **(**TempData.ContainsKey**(**"Name"**))**

Name = TempData**[**"Name"**]**.ToString**()**;

**if** **(**TempData.ContainsKey**(**"Age"**))**

Age = **int**.Parse**(**TempData**[**"Age"**]**.ToString**())**;

// do something with user Name or Age here

**return** View**()**;

**}**

**public** ActionResult Method3**()**

**{**

string Name;

**int** Age;

**if** **(**TempData.ContainsKey**(**"Name"**))**

Name = TempData**[**"Name"**]**.ToString**()**;

**if** **(**TempData.ContainsKey**(**"Age"**))**

Age = **int**.Parse**(**TempData**[**"Age"**]**.ToString**())**;

// do something with userName or userAge here

**return** View**()**;

**}**

**}**

**}**

In the above example, we have added data into TempData and accessed the same data using a key inside another action method. Please notice that we have converted values into the appropriate type.

**Let’s understand TempData**

**1st Request: http://localhost:xxxxx/Employee/Method1  
2nd Request: http://localhost:xxxxx/Employee/Method2  
3rd Request:**[**http://localhost:xxxxx/Employee/Method3**](http://localhost:xxxxx/Employee/Method3)

As you can see in the above example, we add **Name and Age** in TempData in the first request, and in the second subsequent request, we access the data from the TempData which we stored in the first request. However, we can’t get the same data in the third request because TempData will be cleared out after the second request.

**How to retain TempData values in the consecutive request?**

In order to retain the TempData value in the third consecutive request, we need to call **TempData.Keep()** method. Let’s see the use of **TempData.Keep()** method  with an example

**namespace** *FirstMVCDemo.Controllers*

**{**

**public** **class** EmployeeController : Controller

**{**

**public** ActionResult Method1**()**

**{**

TempData**[**"Name"**]** = "Pranaya";

TempData**[**"Age"**]** = 30;

**return** View**()**;

**}**

**public** ActionResult Method2**()**

**{**

string Name;

**int** Age;

**if** **(**TempData.ContainsKey**(**"Name"**))**

Name = TempData**[**"Name"**]**.ToString**()**;

**if** **(**TempData.ContainsKey**(**"Age"**))**

Age = **int**.Parse**(**TempData**[**"Age"**]**.ToString**())**;

TempData.Keep**()**;

// do something with userName or userAge here

**return** View**()**;

**}**

**public** ActionResult Method3**()**

**{**

string Name;

**int** Age;

**if** **(**TempData.ContainsKey**(**"Name"**))**

Name = TempData**[**"Name"**]**.ToString**()**;

**if** **(**TempData.ContainsKey**(**"Age"**))**

Age = **int**.Parse**(**TempData**[**"Age"**]**.ToString**())**;

// do something with userName or userAge here

**return** View**()**;

**}**

**}**

**}**

There are two overloaded versions of the Keep Method. They are as follows:

1. **public void Keep()**: It will mark all keys in the temp data dictionary for retention.
2. **public void Keep(string key)**: It will mark the specified key in the temp data dictionary for retention. Here, the key parameter is the key to retain in the dictionary