# MVC Architecture:

Let's understand the MVC architecture in ASP.NET.

MVC stands for Model, View and Controller. MVC separates application into three components –

M -> Model

V -> View

C -> Controller.

**Model**: Model represents shape of the data and business logic. It maintains the data of the application. Model objects retrieve and store model state in a database.

Model is a data and business logic.

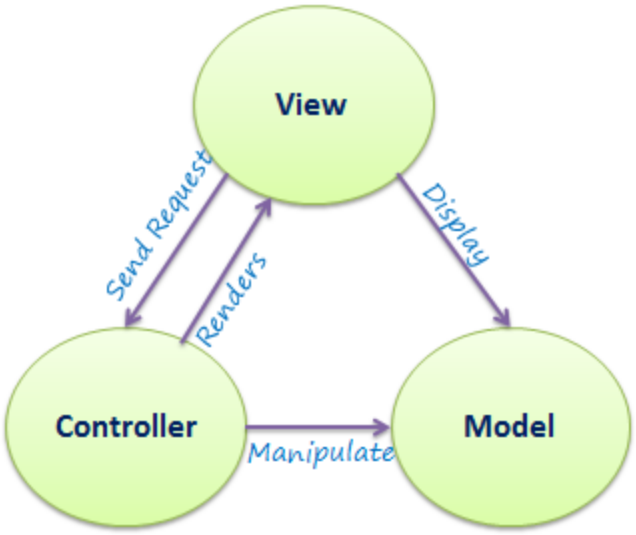
**View**: View is a user interface. View display data using model to the user and also enables them to modify the data.

View is a User Interface.

**Controller**: Controller handles the user request. Typically, user interact with View, which in-turn raises appropriate URL request, this request will be handled by a controller. The controller renders the appropriate view with the model data as a response.

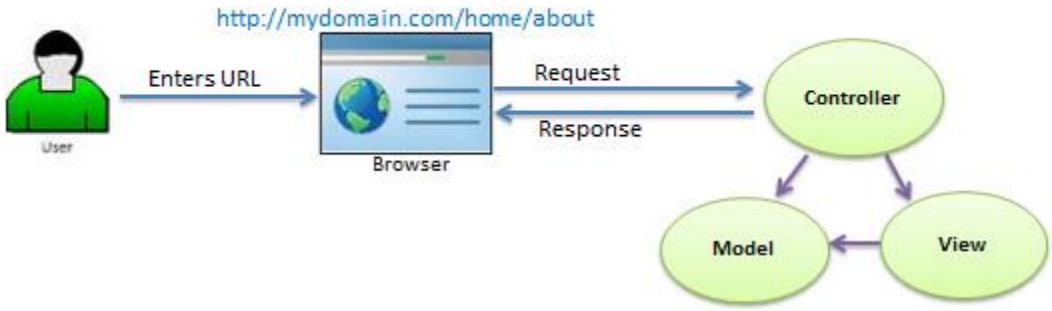
Controller is a request handler.

The following figure illustrates the interaction between Model, View and Controller.




MVC Architecture

The following figure illustrates the flow of the user's request in ASP.NET MVC.

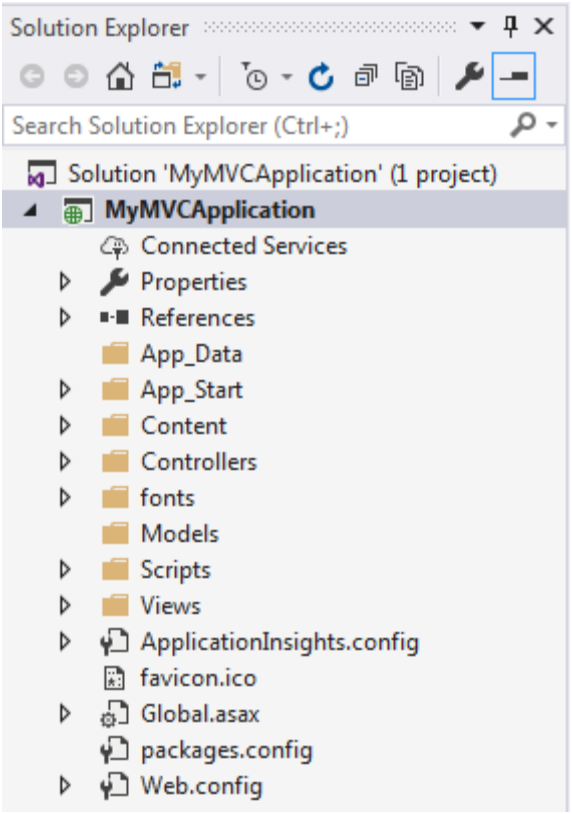



Request/Response in MVC Architecture

As per the above figure, when the user enters a URL in the browser, it goes to the server and calls appropriate controller. Then, the Controller uses the appropriate View and Model and creates the response and sends it back to the user. We will see the details of the interaction in the next few sections.

# ASP.NET MVC Folder Structure

We have created our first MVC 5 application in the previous section. Visual Studio creates the following folder structure for MVC application by default.




MVC Folder Structure

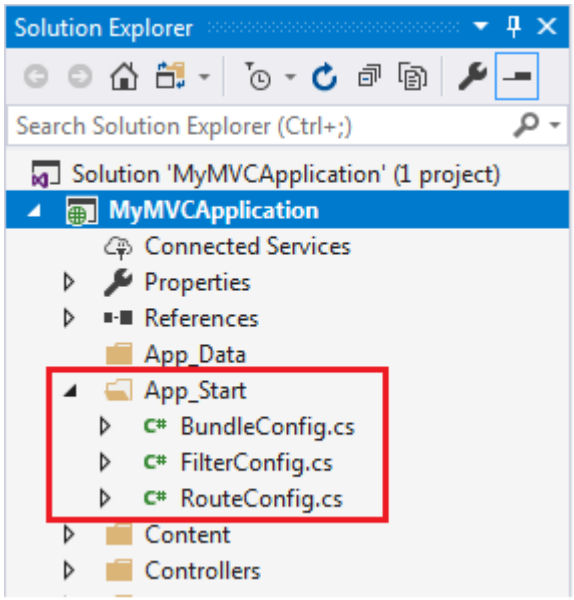
Let's see significance of each folder.

## App_Data

App_Data folder can contain application data files like LocalDB, .mdf files, xml files and other data related files. IIS will never serve files from App_Data folder.

## App_Start

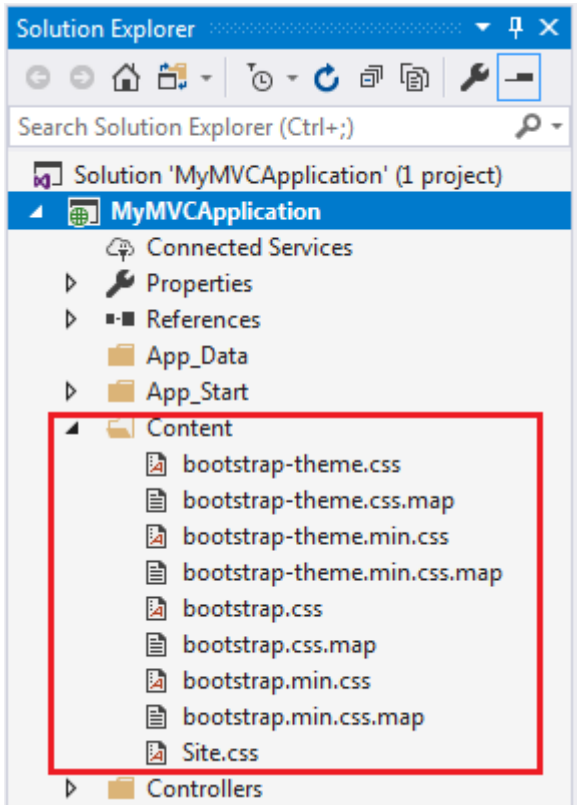
App_Start folder can contain class files which will be executed when the application starts. Typically, these would be config files like AuthConfig.cs, BundleConfig.cs, FilterConfig.cs, RouteConfig.cs etc. MVC 5 includes BundleConfig.cs, FilterConfig.cs and RouteConfig.cs by default. We will see significance of these files later.




App_Start Folder

## Content

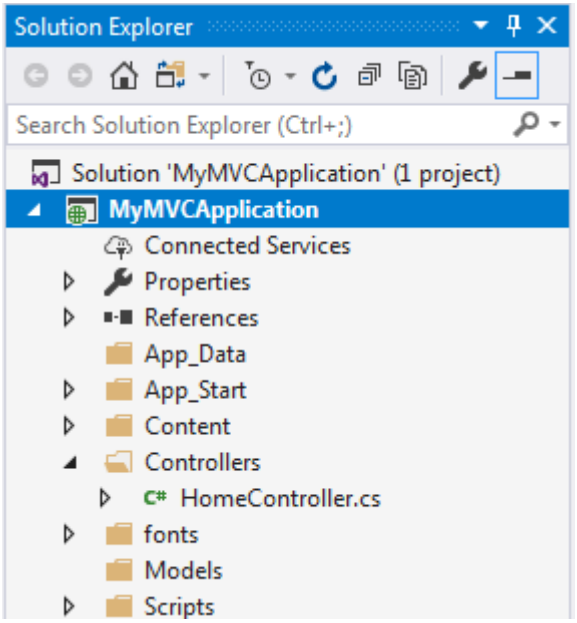
Content folder contains static files like css files, images and icons files. MVC 5 application includes bootstrap.css, bootstrap.min.css and Site.css by default.




Content Folder

## Controllers

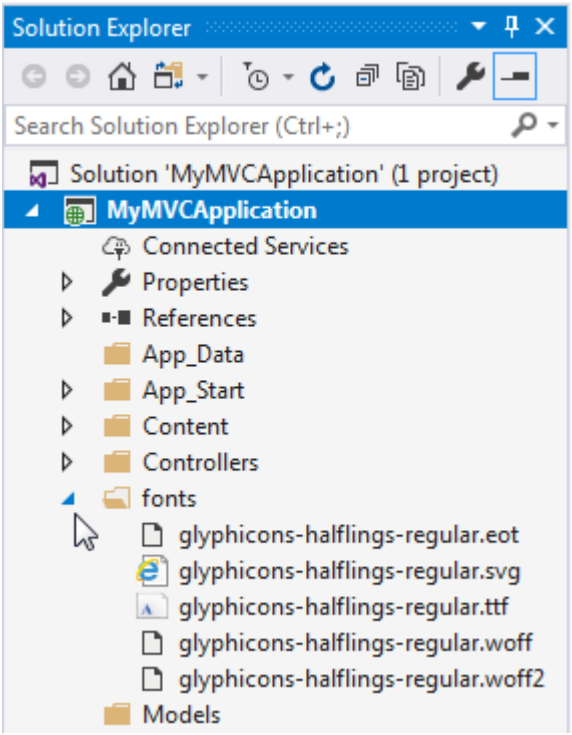
Controllers folder contains class files for the controllers. Controllers handles users' request and returns a response. MVC requires the name of all controller files to end with "Controller". You will learn about the controller in the next section.




Controller Folder

## fonts

Fonts folder contains custom font files for your application.



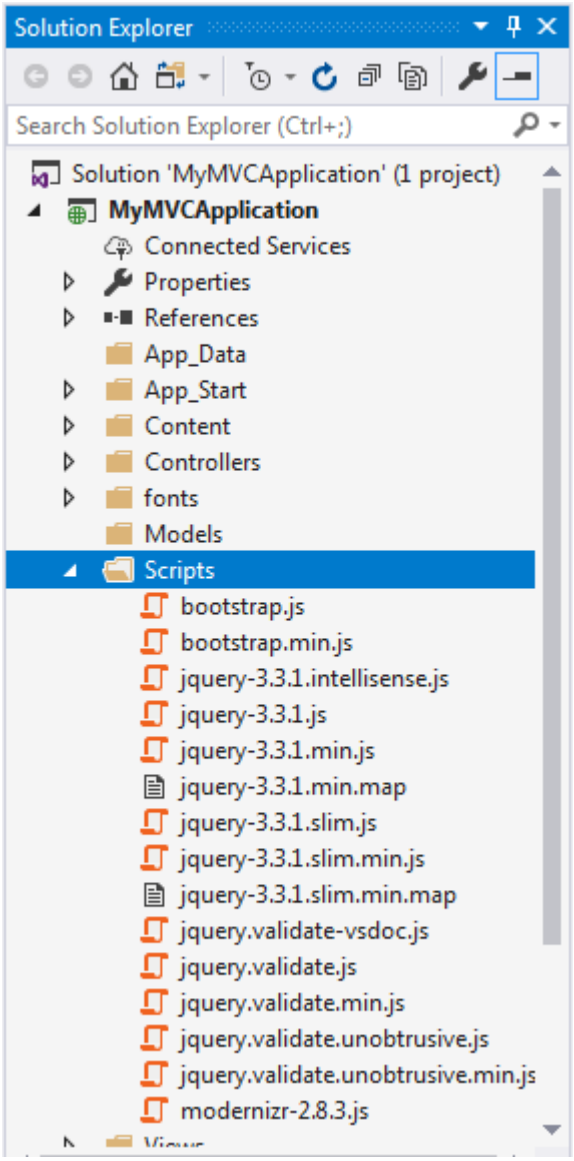

Fonts folder

## Models

Models folder contains model class files. Typically model class includes public properties, which will be used by application to hold and manipulate application data.

## Scripts

Scripts folder contains JavaScript or VBScript files for the application. MVC 5 includes javascript files for bootstrap, jquery 1.10 and modernizer by default.



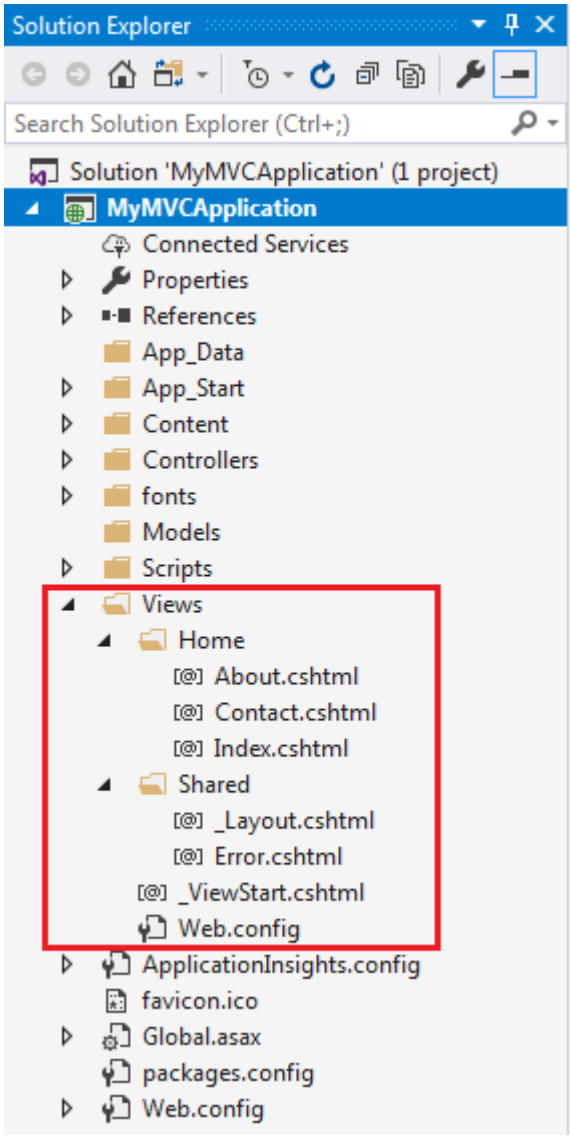
Scripts Folder

## Views

Views folder contains html files for the application. Typically view file is a .cshtml file where you write html and C# or VB.NET code.

Views folder includes separate folder for each controllers. For example, all the .cshtml files, which will be rendered by HomeController will be in View > Home folder.

Shared folder under View folder contains all the views which will be shared among different controllers e.g. layout files.



View Folder

Additionally, MVC project also includes following configuration files:

## Global.asax

Global.asax allows you to write code that runs in response to application level events, such as Application_BeginRequest, application_start, application_error, session_start, session_end etc.

## Packages.config

Packages.config file is managed by NuGet to keep track of what packages and versions you have installed in the application.

# Routing in MVC

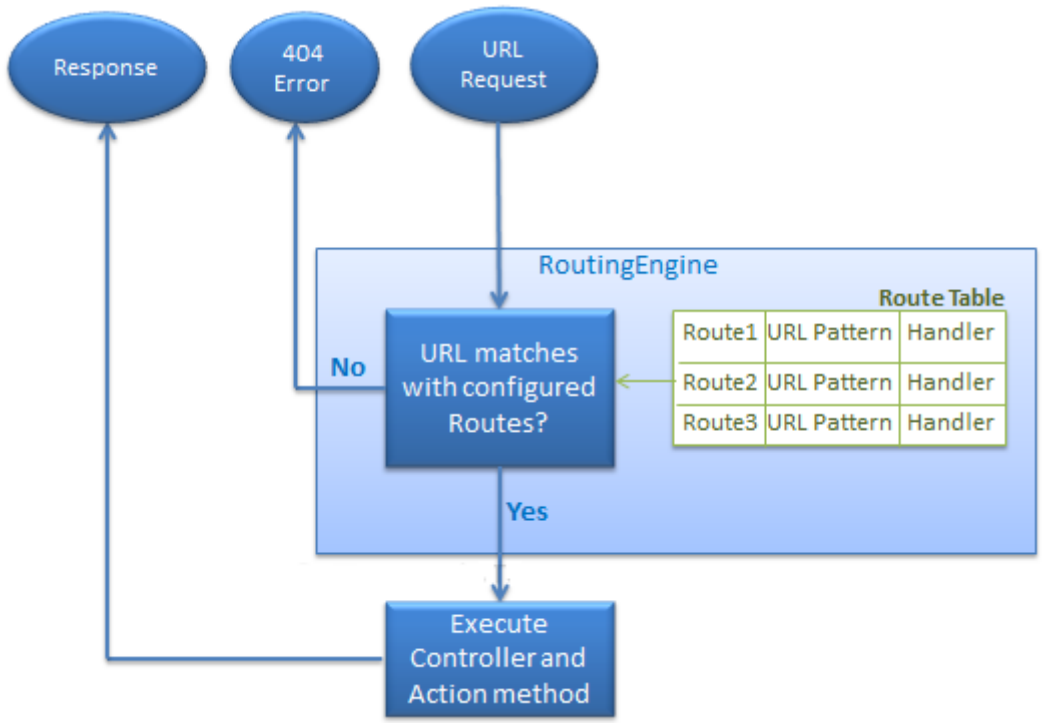
In the ASP.NET Web Forms application, every URL must match with a specific .aspx file. For example, a URL http://domain/studentsinfo.aspx must match with the file studentsinfo.aspx that contains code and markup for rendering a response to the browser.

ASP.NET introduced Routing to eliminate needs of mapping each URL with a physical file. Routing enable us to define URL pattern that maps to the request handler. This request handler can be a file or class. In ASP.NET Webform application, request handler is .aspx file and in MVC, it is Controller class and Action method. For example, http://domain/students can be mapped to http://domain/studentsinfo.aspx in ASP.NET Webforms and the same URL can be mapped to Student Controller and Index action method in MVC.

## Route

Route defines the URL pattern and handler information. All the configured routes of an application stored in RouteTable and will be used by Routing engine to determine appropriate handler class or file for an incoming request.

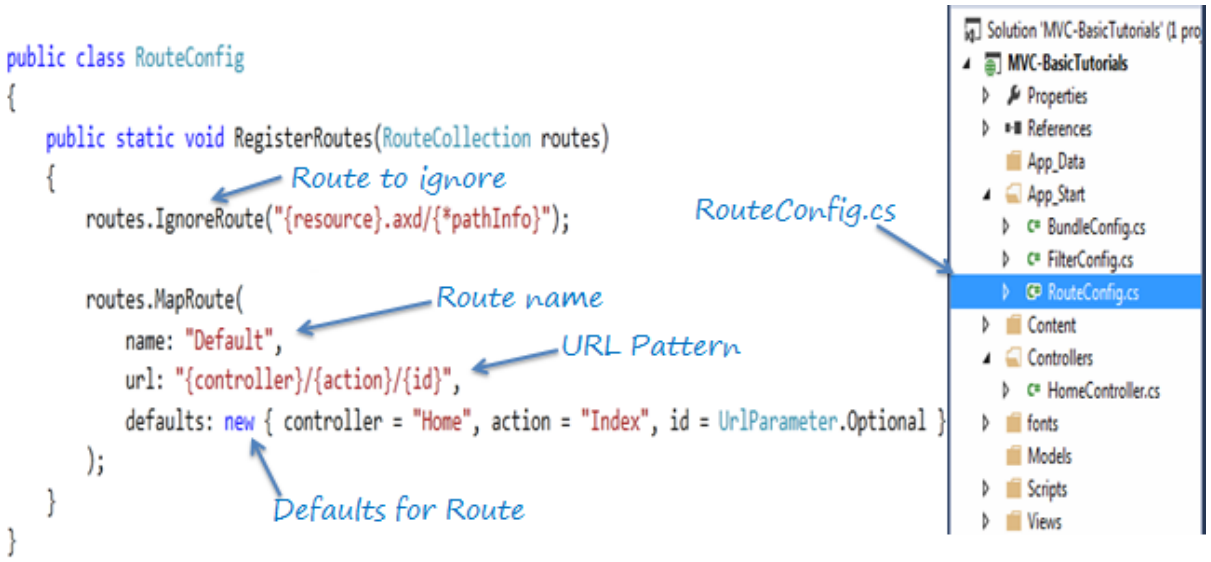
The following figure illustrates the Routing process.




Routing in MVC

# Configure a Route

Every MVC application must configure (register) at least one route, which is configured by MVC framework by default. You can register a route in **RouteConfig** class, which is in RouteConfig.cs under **App_Start** folder. The following figure illustrates how to configure a Route in the RouteConfig class .



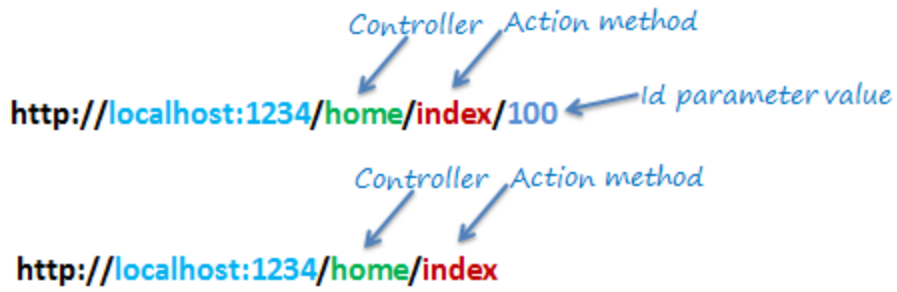

Configure Route in MVC

As you can see in the above figure, the route is configured using the MapRoute() extension method of RouteCollection, where name is "Default", url pattern is *"{controller}/{action}/{id}"* and defaults parameter for controller, action method and id parameter. Defaults specifies which controller, action method or value of id parameter should be used if they do not exist in the incoming request URL.

The same way, you can configure other routes using MapRoute method of RouteCollection. This RouteCollection is actually a property of RouteTable class.

## URL Pattern

The URL pattern is considered only after domain name part in the URL. For example, the URL pattern *"{controller}/{action}/{id}"*would look like localhost:1234/{controller}/{action}/{id}. Anything after "localhost:1234/" would be considered as controller name. The same way, anything after controller name would be considered as action name and then value of id parameter.




Routing in MVC

If the URL doesn't contain anything after domain name then the default controller and action method will handle the request. For example, http://lcoalhost:1234 would be handled by HomeController and Index method as configured in the defaults parameter.

The following table shows which Controller, Action method and Id parameter would handle different URLs considering above default route.

| URL | Controller | Action | Id |
|---|---|---|---|
| http://localhost/home | HomeController | Index | null |
| http://localhost/home/index/123 | HomeController | Index | 123 |
| http://localhost/home/about | HomeController | About | null |
| http://localhost/home/contact | HomeController | Contact | null |
| http://localhost/student | StudentController | Index | null |
| http://localhost/student/edit/123 | StudentController | Edit | 123 |

# Controller

In this section, you will learn about the Controller in ASP.NET MVC.

The Controller in MVC architecture handles any incoming URL request. Controller is a class, derived from the base class `System.Web.Mvc.Controller`. Controller class contains public methods called **Action** methods. Controller and its action method handles incoming browser requests, retrieves necessary model data and returns appropriate responses.

In ASP.NET MVC, every controller class name must end with a word "Controller". For example, controller for home page must be HomeController and controller for student must be StudentController. Also,

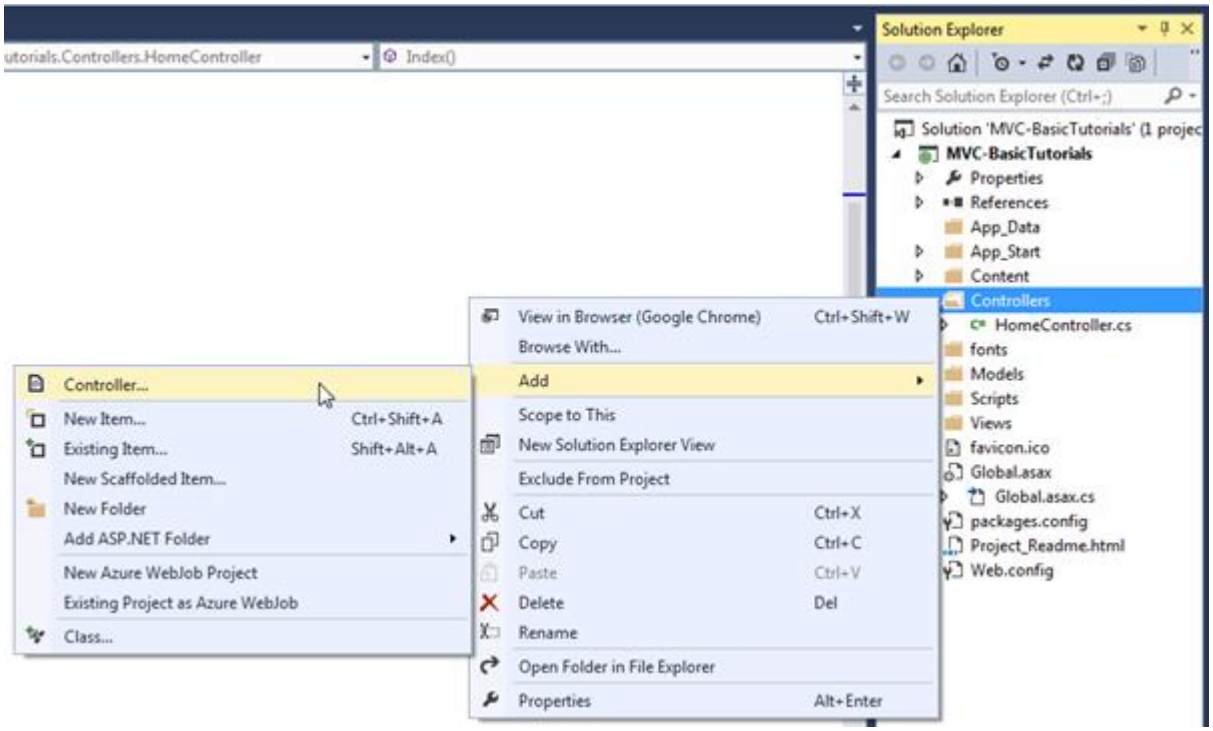
every controller class must be located in Controller folder of MVC folder structure.

## Adding a New Controller

Now, let's add a new empty controller in our MVC application in Visual Studio.

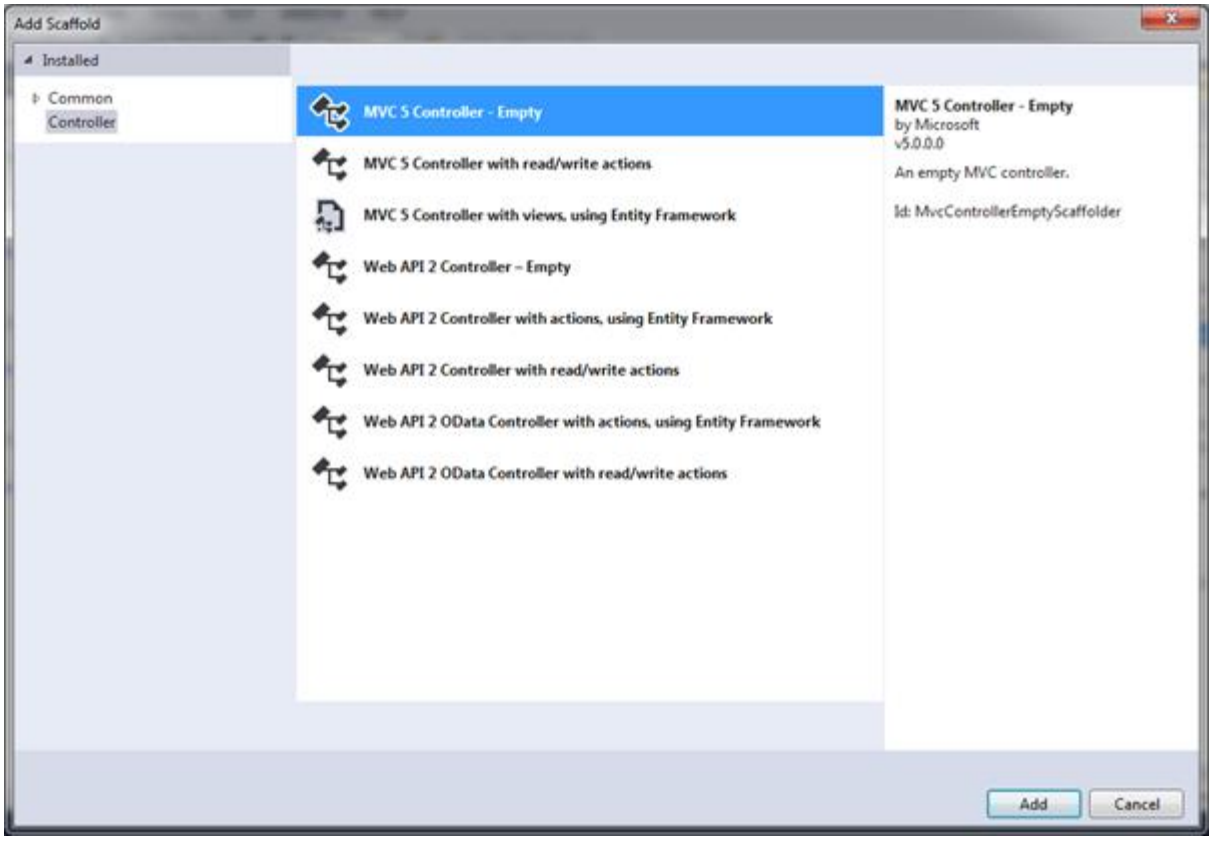
In the previous section we learned how to create our first MVC application, which in turn created a default HomeController. Here, we will create a new StudentController.

In the Visual Studio, right click on the Controller folder -> select **Add** -> click on **Controller..**



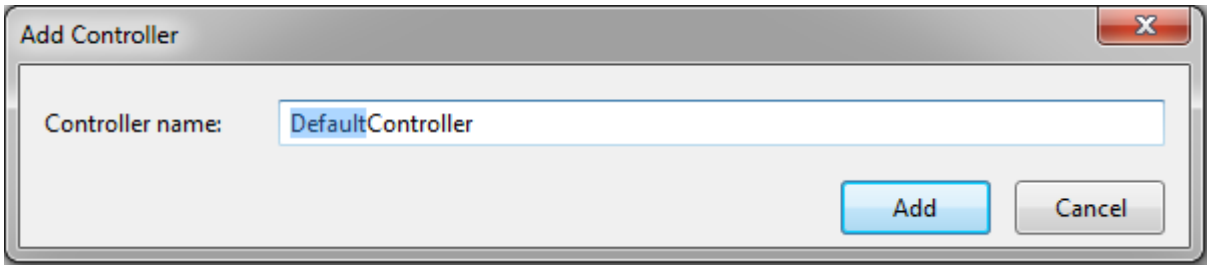

Add New Controller

This opens Add Scaffold dialog as shown below.



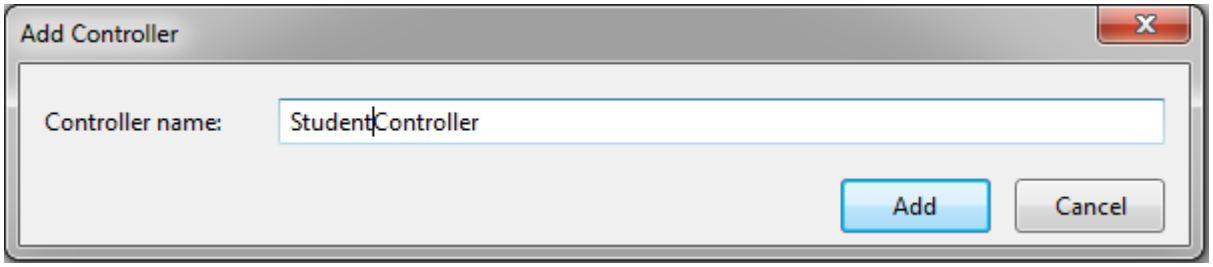
Adding Controller

Add Scaffold dialog contains different templates to create a new **controller**. We will learn about other templates later. For now, select **"MVC 5 Controller - Empty"** and click **Add**. It will open Add Controller dialog as shown below



Adding Controller

In the Add Controller dialog, enter the name of the controller. Remember, controller name must end with Controller. Let's enter StudentController and click **Add**.



Adding Controller

This will create StudentController class with Index method in StudentController.cs file under Controllers folder, as shown below.

## Example: Controller

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

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

As you can see above, the StudentController class is derived from Controller class. Every controller in MVC must derived from this abstract Controller class. This base Controller class contains helper methods that can be used for various purposes.

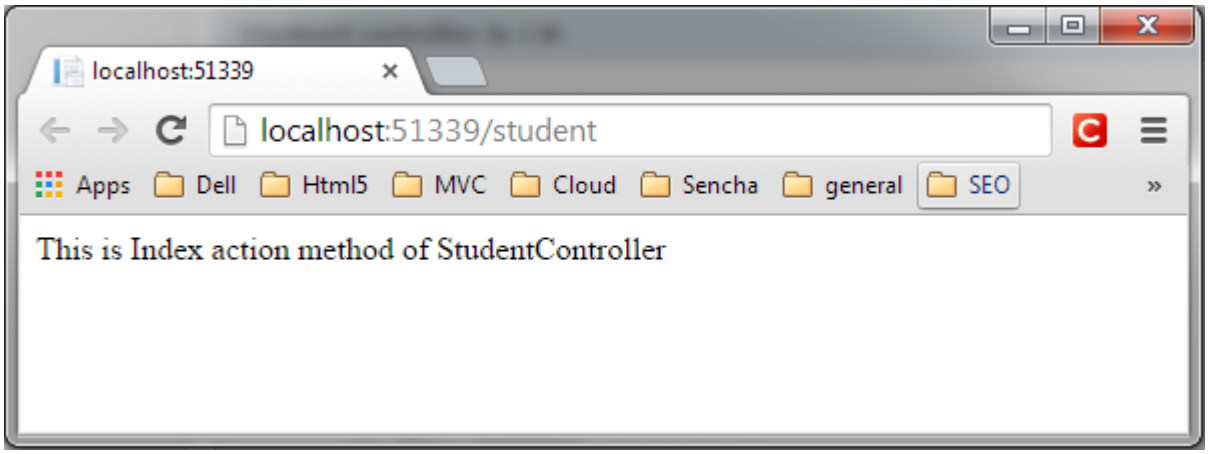
Now, we will return a dummy string from Index action method of above StudentController. Changing the return type of Index method from ActionResult to string and returning dummy string is shown below. You will learn about ActionResult in the next section.

## Example: Controller

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

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
```

```
        public string Index()
        {
                return "This is Index action method of StudentController";
        }
    }
}
```

We have already seen in the routing section that the URL request *http://localhost/student* or *http://localhost/student/index* is handled by the Index() method of StudentController class, shown above. So let's invoke it from the browser and you will see the following page in the browser.




Controller

## Points to Remember :

1. A Controller handles incomming URL requests. MVC routing sends request to appropriate controller and action method based on URL and configured Routes.
2. All the public methods in the Controlle class are called Action methods.
3. A Controller class must be derived from System.Web.Mvc.Controller class.
4. A Controller class name must end with "Controller".
5. New controller can be created using different scaffolding templates. You can create custom scaffolding template also.

# Model in ASP.NET MVC

In this section, you will learn about the Model in ASP.NET MVC framework.

Model represents domain specific data and business logic in MVC architecture. It maintains the data of the application. Model objects retrieve and store model state in the persistance store like a database.

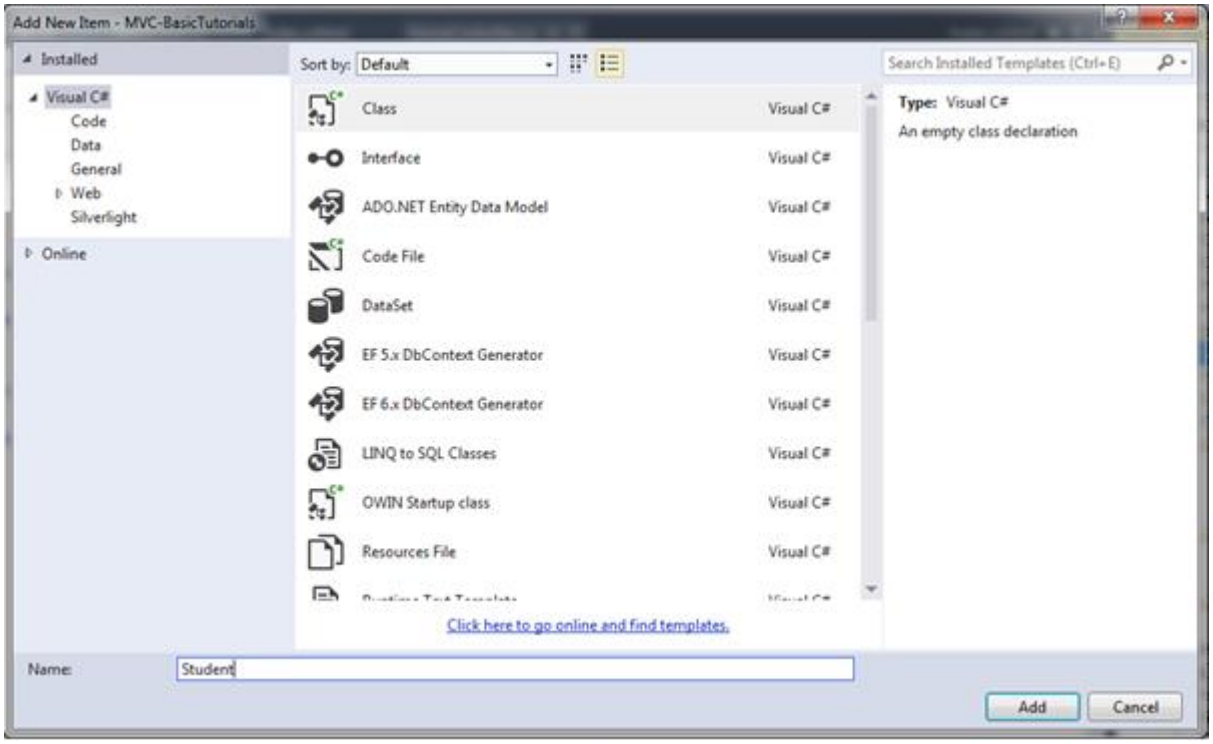
Model class holds data in public properties. All the Model classes reside in the Model folder in MVC folder structure.

Let's see how to add model class in ASP.NET MVC.

## Adding a Model

Open our first MVC project created in previous step in the Visual Studio. Right click on Model folder -> Add -> click on Class..

In the Add New Item dialog box, enter class name 'Student' and click **Add**.




Create Model Class

This will add new Student class in model folder. Now, add Id, Name, Age properties as shown below.

## Example: Model class

```
namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set;  }
        public int Age { get; set;  }
    }
}
```

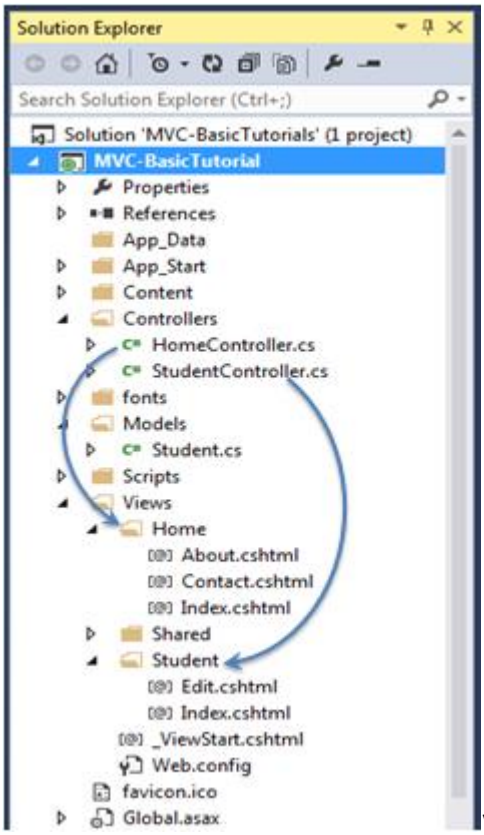
So in this way, you can create a model class which you can use in View. You will learn how to implement validations using model later.

# View in ASP.NET MVC

In this section, you will learn about the View in ASP.NET MVC framework.

View is a user interface. View displays data from the model to the user and also enables them to modify the data.

ASP.NET MVC views are stored in **Views** folder. Different action methods of a single controller class can render different views, so the Views folder contains a separate folder for each controller with the same name as controller, in order to accommodate multiple views. For example, views, which will be rendered from any of the action methods of HomeController, resides in Views > Home folder. In the same way, views which will be rendered from StudentController, will resides in Views > Student folder as shown below.



View folders for Controllers

## Note:

Shared folder contains views, layouts or partial views which will be shared among multiple views.

## Razor View Engine

Microsoft introduced the Razor view engine and packaged with MVC 3. You can write a mix of html tags and server side code in razor view. Razor uses @ character for server side code instead of traditional <% %>. You can use C# or Visual Basic syntax to write server side code inside razor view. Razor view engine maximize the speed of writing code by minimizing the

number of characters and keystrokes required when writing a view. Razor views files have .cshtml or vbhtml extension.

ASP.NET MVC supports following types of view files:

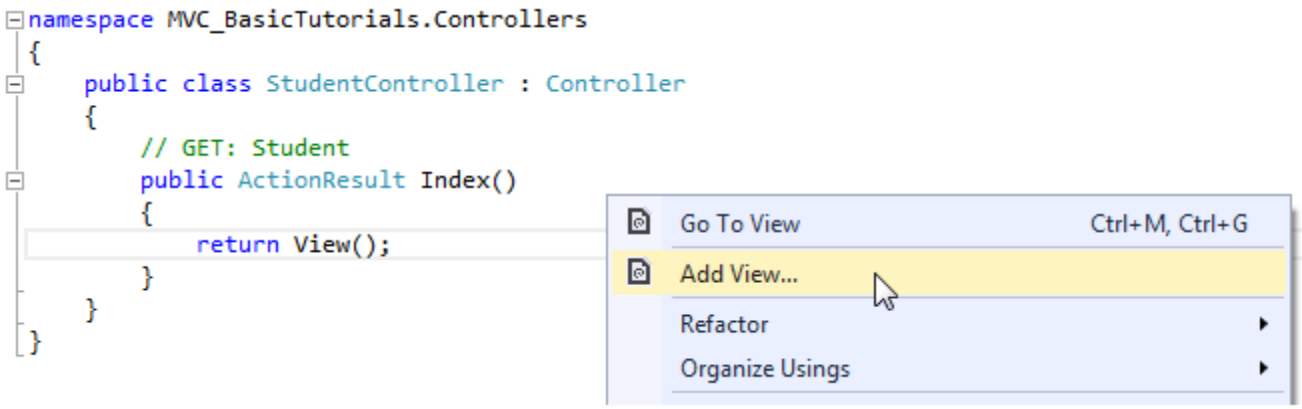
| View file extension | Description |
|---|---|
| .cshtml | C# Razor view. Supports C# with html tags. |
| .vbhtml | Visual Basic Razor view. Supports Visual Basic with html tags. |
| .aspx | ASP.Net web form |
| .ascx | ASP.NET web control |

Learn Razor syntax in the next section. Let's see how to create a new view using Visual Studio 2013 for Web with MVC 5.

## Create New View

We have already created StudentController and Student model in the previous section. Now, let's create a Student view and understand how to use model into view.

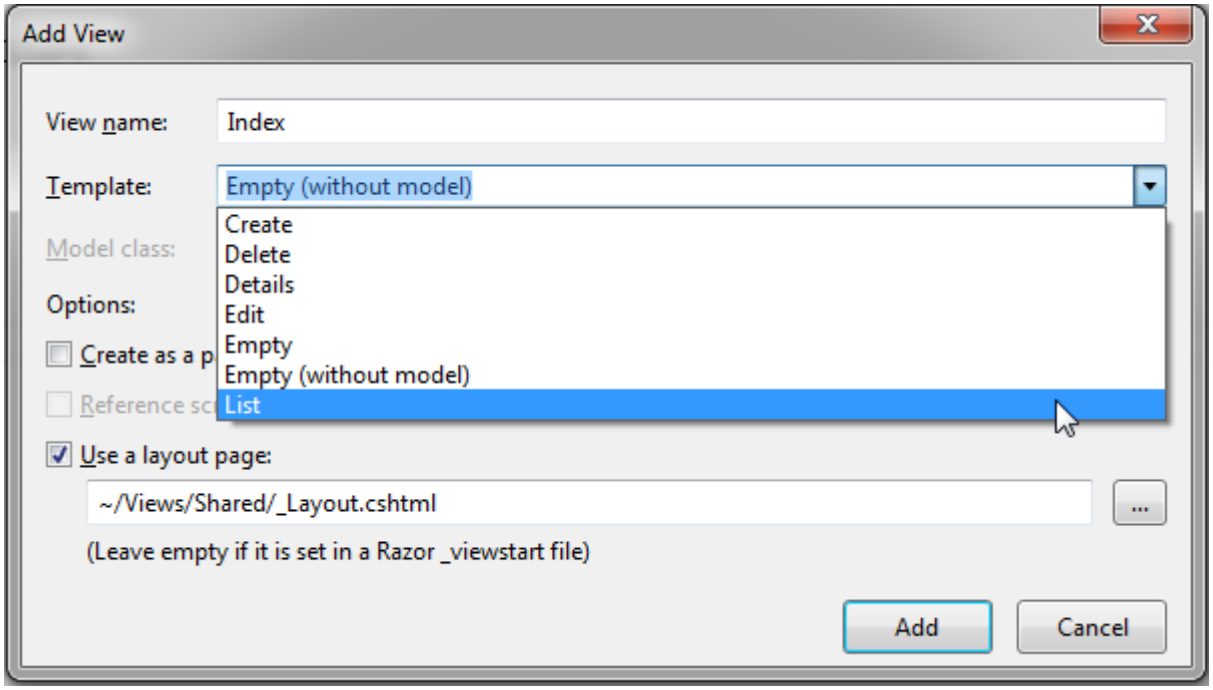
We will create a view, which will be rendered from Index method of StudentContoller. So, open a StudentController class -> right click inside Index method -> click **Add View..**




Create a View

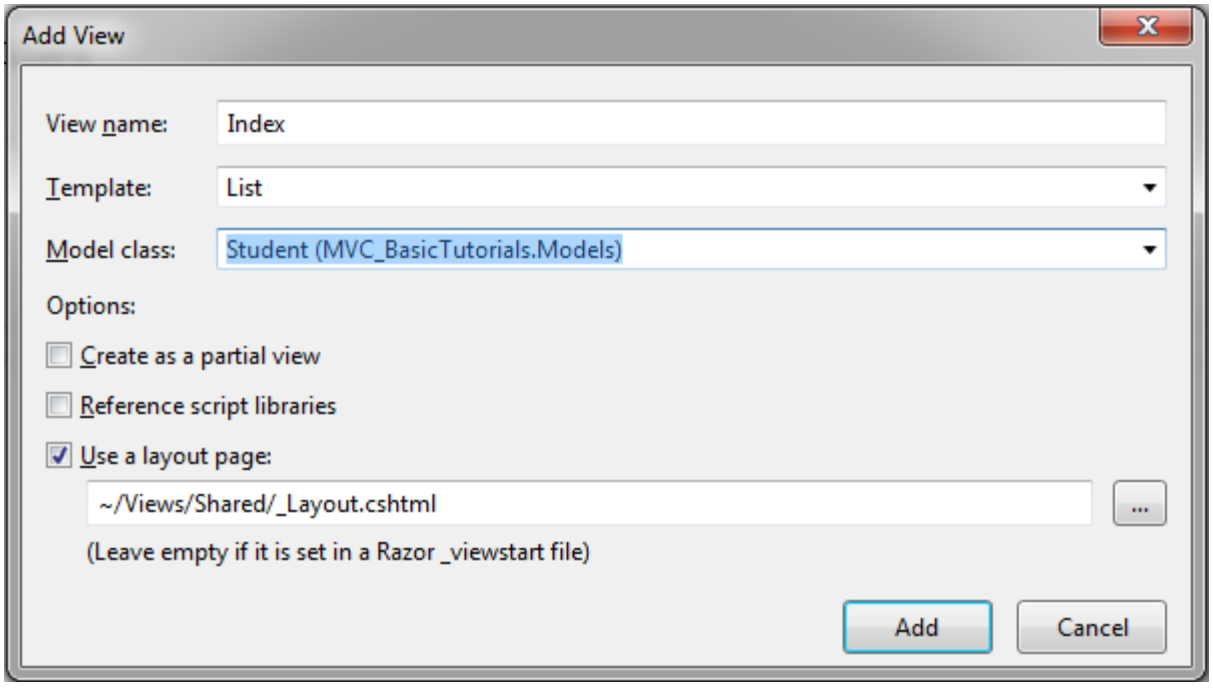
In the Add View dialogue box, keep the view name as Index. It's good practice to keep the view name the same as the action method name so that you don't have to specify view name explicitly in the action method while returning the view.

Select the scaffolding template. Template dropdown will show default templates available for Create, Delete, Details, Edit, List or Empty view. Select "List" template because we want to show list of students in the view.



View

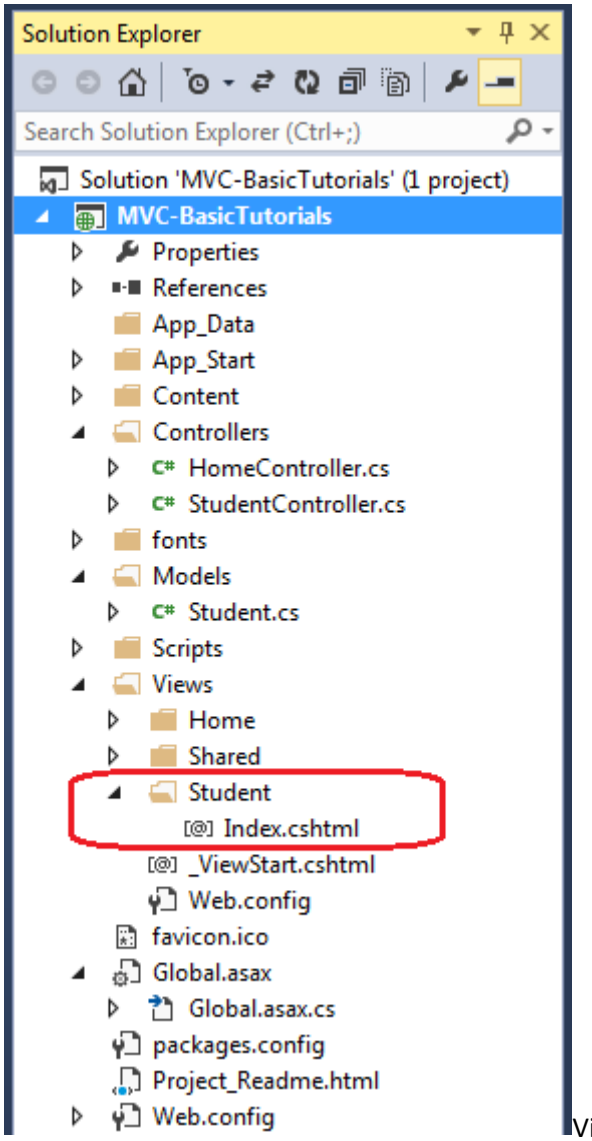
Now, select Student from the Model class dropdrown. Model class dropdown automatically displays the name of all the classes in the Model folder. We have already created Student Model class in the previous section, so it would be included in the dropdown.



View

Check "Use a layout page" checkbox and select _Layout.cshtml page for this view and then click **Add** button. We will see later what is layout page but for now think it like a master page in MVC.

This will create Index view under View -> Student folder as shown below:



View

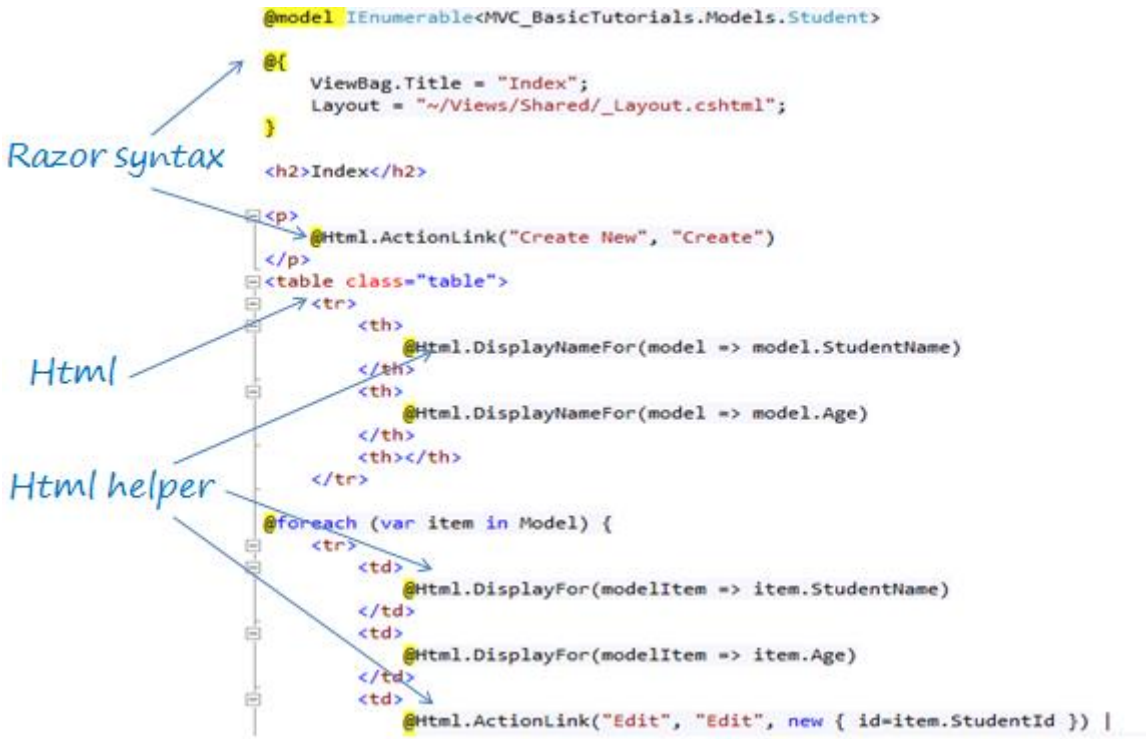
The following code snippet shows an Index.cshtml created above.

## Views\Student\Index.cshtml:

```
@model IEnumerable<MVC_BasicTutorials.Models.Student>

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

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
```

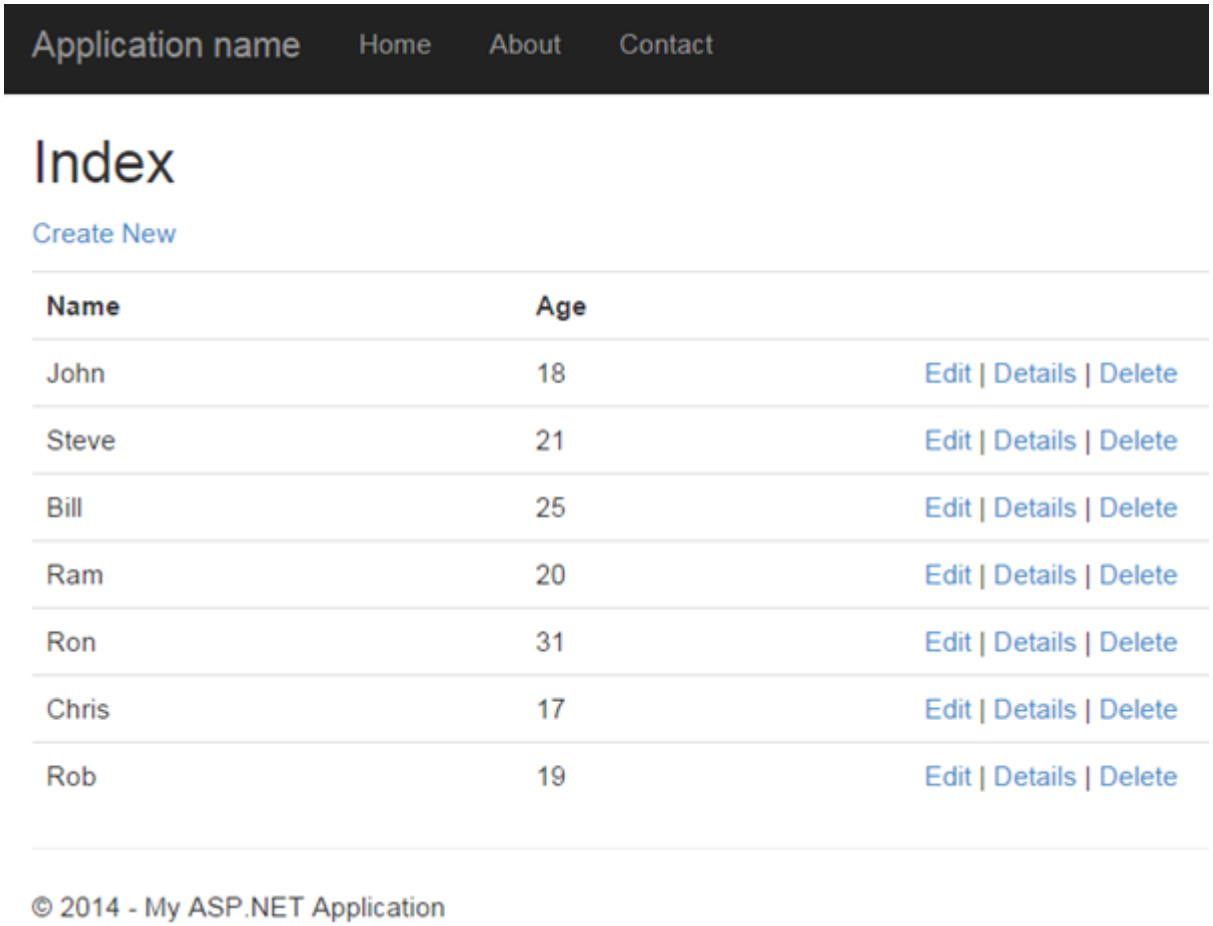
```
            <th>
                @Html.DisplayNameFor(model => model.Age)
            </th>
            <th></th>
        </tr>

@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.StudentName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Age)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |
                @Html.ActionLink("Details", "Details", new { id=item.StudentId  }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.StudentId })
            </td>
        </tr>
}

</table>
```

As you can see in the above Index view, it contains both Html and razor codes. Inline razor expression starts with @ symbol. @Html is a helper class to generate html controls. You will learn razor syntax and html helpers in the coming sections.




Index.cshtml

The above Index view would look like below.




Index View

## Note:

Every view in the ASP.NET MVC is derived from WebViewPage class included in System.Web.Mvc namespace.

## Points to Remember :

1. View is a User Interface which displays data and handles user interaction.
2. Views folder contains separate folder for each controller.
3. ASP.NET MVC supports Razor view engine in addition to traditional .aspx engine.
4. Razor view files has .cshtml or .vbhtml extension.

# Integrate Controller, View and Model

We have already created StudentController, model and view in the previous sections, but we have not integrated all these components in-order to run it.

The following code snippet shows StudentController and Student model class & view created in the previous sections.

## Example: StudentController

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

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

## Example: Student Model class

```
namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set; }
        public int Age { get; set; }
    }
}
```

## Example: Index.cshtml to display student list

```
@model IEnumerable<MVC_BasicTutorials.Models.Student>

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

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
```

```
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Age)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.StudentName)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Age)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |
            @Html.ActionLink("Details", "Details", new { id=item.StudentId  }) |
            @Html.ActionLink("Delete", "Delete", new { id = item.StudentId })
        </td>
    </tr>
}

</table>
```

Now, to run it successfully, we need to pass a model object from controller to Index view. As you can see in the above Index.cshtml, it uses IEnumerable of Student as a model object. So we need to pass IEnumerable of Student model from the Index action method of StudentController class as shown below.

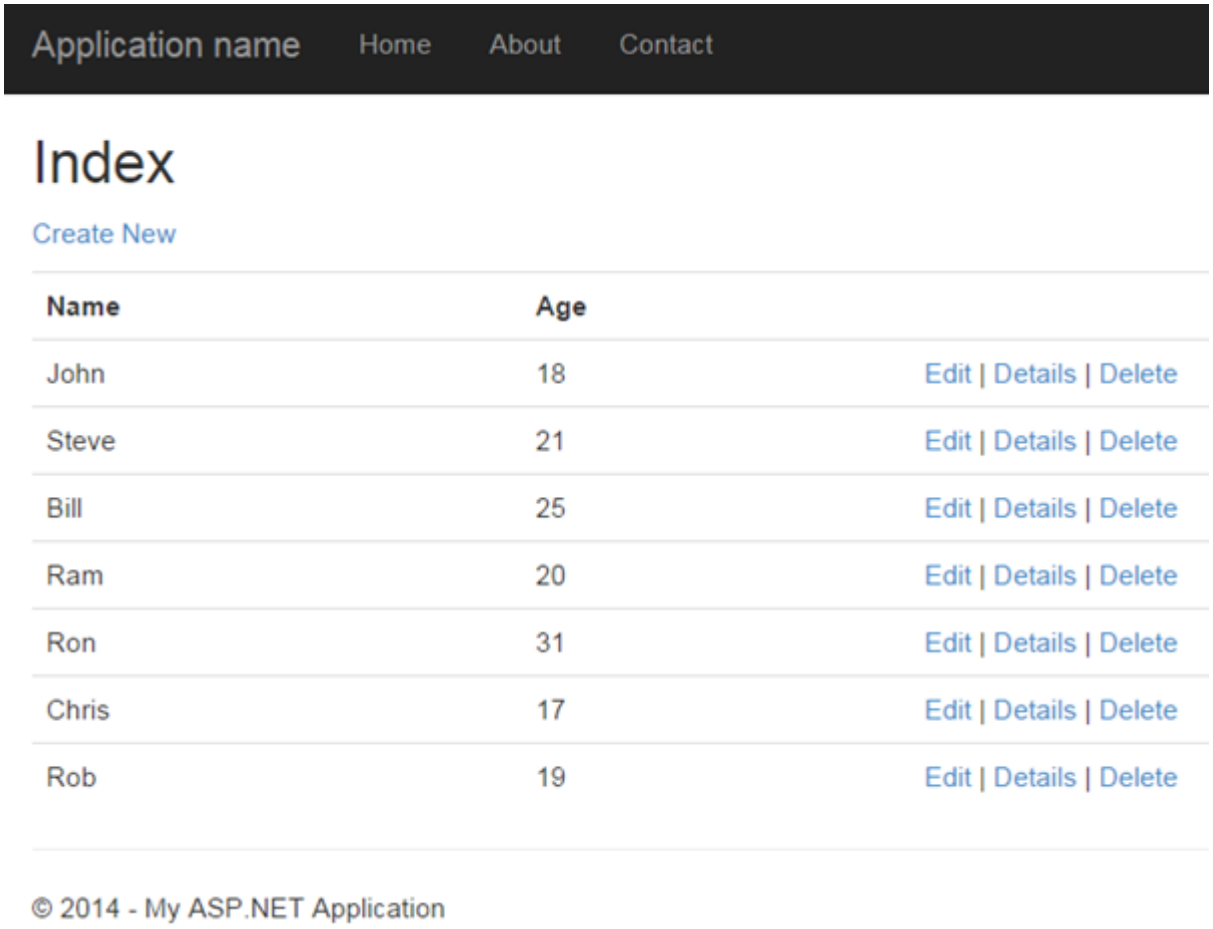
## Example: Passing Model from Controller

```
public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        var studentList = new List<Student>{
                            new Student() { StudentId = 1, StudentName = "John",
Age = 18 } ,
                            new Student() { StudentId = 2, StudentName = "Steve",
Age = 21 } ,
                            new Student() { StudentId = 3, StudentName = "Bill",
Age = 25 } ,
                            new Student() { StudentId = 4, StudentName = "Ram" ,
Age = 20 } ,
                            new Student() { StudentId = 5, StudentName = "Ron" ,
Age = 31 } ,
                            new Student() { StudentId = 4, StudentName = "Chris" ,
Age = 17 } ,
                            new Student() { StudentId = 4, StudentName = "Rob" ,
Age = 19 }
                        };
        // Get the students from the database in the real application

        return View(studentList);
    }
```

}
Try it

As you can see in the above code, we have created a List of student objects for an example purpose (in real life applicatoin, you can fetch it from the database). We then pass this list object as a parameter in the View() method. The View() method is defined in base Controller class, which automatically binds model object to the view.

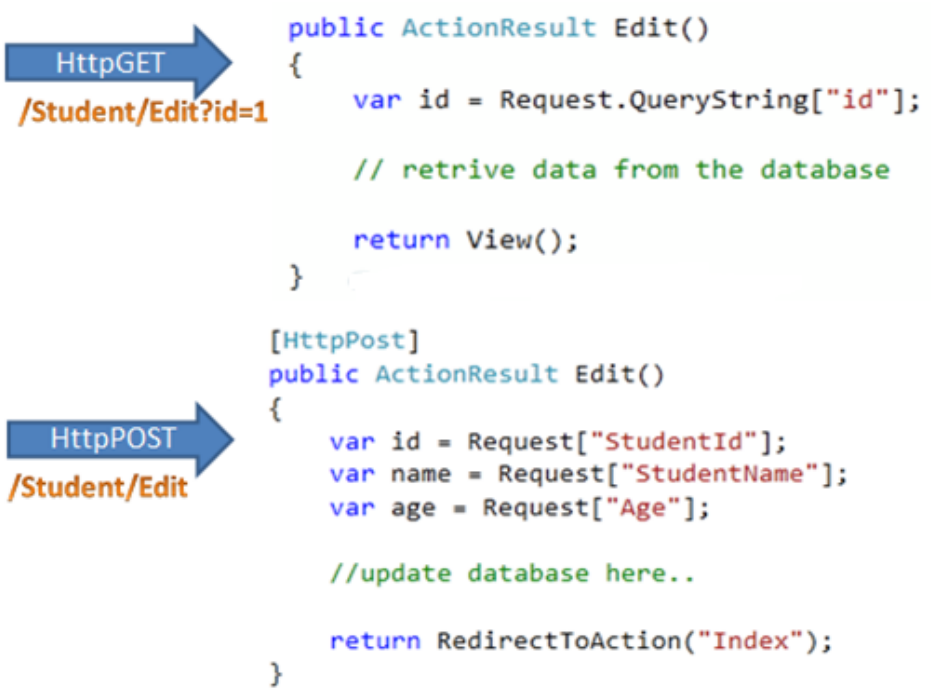
Now, you can run the MVC project by pressing F5 and navigate to *http://localhost/Student*. You will see following view in the browser.




# Model Binding

In this section, you will learn about model binding in MVC framework.

To understand the model binding in MVC, first let's see how you can get the http request values in the action method using traditional ASP.NET style. The following figure shows how you can get the values from HttpGET and HttpPOST request by using the Request object directly in the action method.



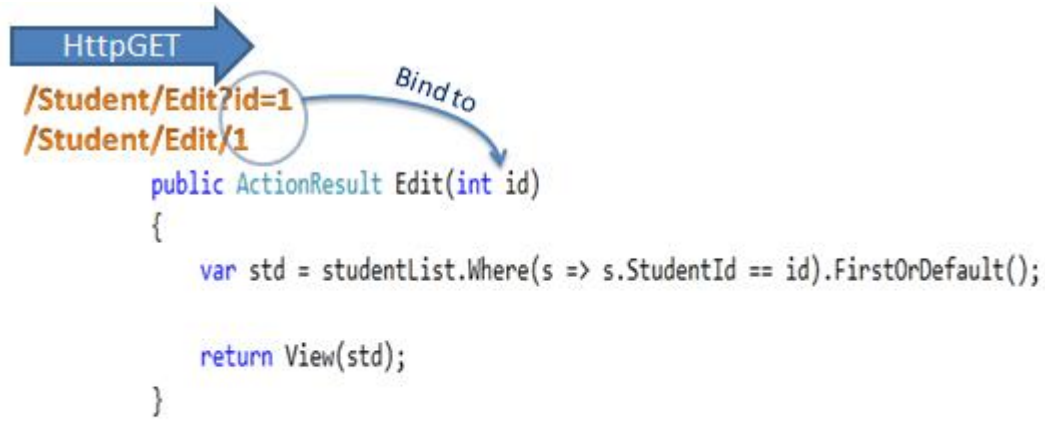

Accessing Request Data

As you can see in the above figure, we use the Request.QueryString and Request (Request.Form) object to get the value from HttpGet and HttpPOST request. Accessing request values using the Request object is a cumbersome and time wasting activity.

With model binding, MVC framework converts the http request values (from query string or form collection) to action method parameters. These parameters can be of primitive type or complex type.

## Binding to Primitive type

HttpGET request embeds data into a query string. MVC framework automatically converts a query string to the action method parameters. For example, the query string "id" in the following GET request would automatically be mapped to the id parameter of the Edit() action method.




Model Binding

You can also have multiple parameters in the action method with different data types. Query string values will be converted into paramters based on matching name.

For example, *http://localhost/Student/Edit?id=1&name=John* would map to id and name parameter of the following Edit action method.

## Example: Convert QueryString to Action Method Parameters

```
public ActionResult Edit(int id, string name)
{
    // do something here

    return View();
}
```

## Binding to Complex type

Model binding also works on complex types. Model binding in MVC framework automatically converts form field data of HttpPOST request to the properties of a complex type parameter of an action method.

Consider the following model classes.

## Example: Model classes in C#

```
public class Student
{
```

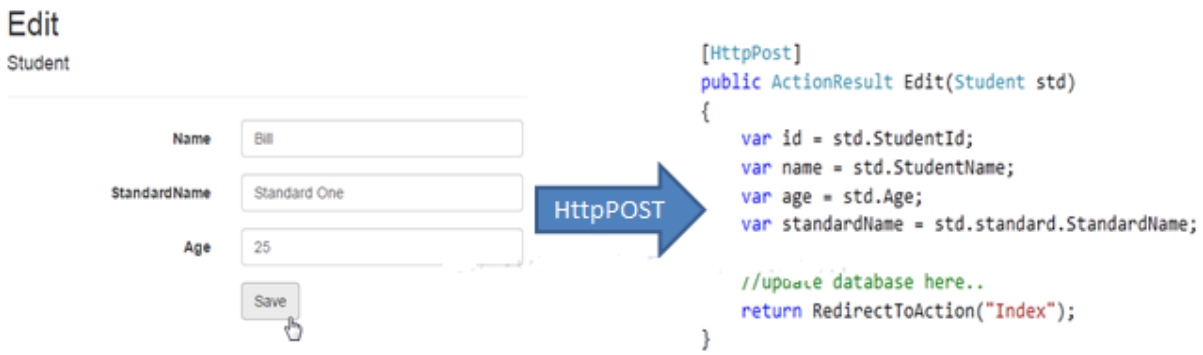
```
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public int Age { get; set; }
    public Standard standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }
}
```

Now, you can create an action method which includes Student type parameter. In the following example, Edit action method (HttpPost) includes Student type parameter.

## Example: Action Method with Class Type Parameter

```
[HttpPost]
public ActionResult Edit(Student std)
{
    var id = std.StudentId;
    var name = std.StudentName;
    var age = std.Age;
    var standardName = std.standard.StandardName;

    //update database here..

    return RedirectToAction("Index");
}
```

So now, MVC framework will automatically maps Form collection values to Student type parameter when the form submits http POST request to Edit action method as shown below.



So thus, it automatically binds form fields to the complex type parameter of action method.

## FormCollection

You can also include FormCollection type parameter in the action method instead of complex type, to retrieve all the values from view form fields as shown below.


Model Binding

# Bind Attribute

ASP.NET MVC framework also enables you to specify which properties of a model class you want to bind. The [Bind] attribute will let you specify the exact properties a model binder should include or exclude in binding.

In the following example, Edit action method will only bind StudentId and StudentName property of a Student model.

## Example: Binding Parameters

```
[HttpPost]
public ActionResult Edit([Bind(Include = "StudentId, StudentName")] Student std)
{
    var name = std.StudentName;

    //write code to update student

    return RedirectToAction("Index");
}
```

You can also use Exclude properties as below.

## Example: Exclude Properties in Binding

```
[HttpPost]
public ActionResult Edit([Bind(Exclude = "Age")] Student std)
{
    var name = std.StudentName;

    //write code to update student

    return RedirectToAction("Index");
}
```

The Bind attribute will improve the performance by only bind properties which you needed.

# Razor Syntax

Razor is one of the view engine supported in ASP.NET MVC. Razor allows you to write mix of HTML and server side code using C# or Visual Basic. Razor view with visual basic syntax has .vbhtml file extension and C# syntax has .cshtml file extension.

Razor syntax has following Characteristics:

- **Compact**: Razor syntax is compact which enables you to minimize number of characters and keystrokes required to write a code.
- **Easy to Learn**: Razor syntax is easy to learn where you can use your familiar language C# or Visual Basic.
- **Intellisense**: Razor syntax supports statement completion within Visual Studio.

Now, let's learn how to write razor code.

## Inline expression

Start with @ symbol to write server side C# or VB code with Html code. For example, write @Variable_Name to display a value of a server side variable. For example, DateTime.Now returns a current date and time. So, write @DateTime.Now to display current datetime as shown below. A single line expression does not require a semicolon at the end of the expression.

# C# Razor Syntax

```
<h1>Razor syntax demo</h1>

<h2>@DateTime.Now.ToShortDateString()</h2>
```

# Output:

**Razor syntax demo**
08-09-2014

## Multi-statement Code block

You can write multiple line of server side code enclosed in braces `@{ ... }`. Each line must ends with semicolon same as C#.

## Example: Server side Code in Razor Syntax

```
@{
    var date = DateTime.Now.ToShortDateString();
    var message = "Hello World";
}

<h2>Today's date is: @date </h2>
<h3>@message</h3>
```

## Output:

```
Today's date is: 08-09-2014
Hello World!
```

## Display Text from Code Block

Use `@:` or `<text>/<text>` to display texts within code block.

## Example: Display Text in Razor Syntax

```
@{
    var date = DateTime.Now.ToShortDateString();
    string message = "Hello World!";
    @:Today's date is: @date <br />
    @message
}
```

## Output:

```
Today's date is: 08-09-2014
Hello World!
```

Display text using <text> within a code block as shown below.

## Example: Text in Razor Syntax

```
@{
    var date = DateTime.Now.ToShortDateString();
    string message = "Hello World!";
    <text>Today's date is:</text> @date <br />
    @message
}
```

## Output:

```
Today's date is: 08-09-2014
Hello World!
```

## if-else condition

Write if-else condition starting with @ symbol. The if-else code block must be enclosed in braces { }, even for single statement.

## Example: if else in Razor

```
@if(DateTime.IsLeapYear(DateTime.Now.Year) )
{
    @DateTime.Now.Year @:is a leap year.
}
else {
    @DateTime.Now.Year @:is not a leap year.
}
```

## Output:

```
2014 is not a leap year.
```

# for loop

## Example: for loop in Razor

```
@for (int i = 0; i < 5; i++) {
    @i.ToString() <br />
}
```

## Output:

```
0
1
2
3
4
```

# Model

Use @model to use model object anywhere in the view.

## Example: Use Model in Razor

```
@model Student

<h2>Student Detail:</h2>
<ul>
    <li>Student Id: @Model.StudentId</li>
    <li>Student Name: @Model.StudentName</li>
    <li>Age: @Model.Age</li>
</ul>
```

## Output:

```
Student Detail:

- Student Id: 1
- Student Name: John
- Age: 18
```

# Declare Variables

Declare a variable in a code block enclosed in brackets and then use those variables inside html with @ symbol.

## Example: Variable in Razor

```
@{
    string str = "";

    if(1 > 0)
    {
        str = "Hello World!";
    }
}

<p>@str</p>
```

## Output:

```
Hello World!
```

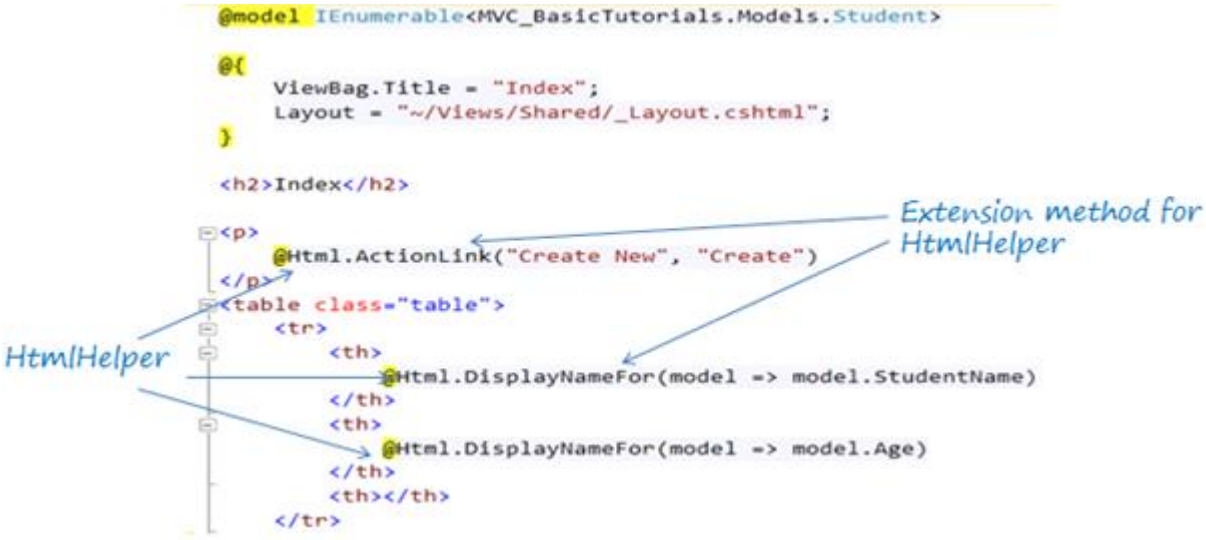
So this was some of the important razor syntaxes. Visit asp.net to learn razor syntax in detail.

# HTML Helpers

In this section, you will learn what are Html helpers and how to use them in the razor view.

HtmlHelper class generates html elements using the model class object in razor view. It binds the model object to html elements to display value of model properties into html elements and also assigns the value of the html elements to the model properties while submitting web form. So always use HtmlHelper class in razor view instead of writing html tags manually.

The following figure shows the use of HtmlHelper class in the razor view.




HTML Helpers

As you can see in the above figure, **@Html** is an object of HtmlHelper class . (@ symbol is used to access server side object in razor syntax). Html is a property of type HtmlHelper included in base class of razor view WebViewPage. ActionLink() and DisplayNameFor() is extension methods included in HtmlHelper class.

HtmlHelper class generates html elements. For example, `@Html.ActionLink("Create New", "Create")` would generate anchor tag `<a href="/Student/Create">Create New</a>`.

There are many extension methods for HtmlHelper class, which creates different html controls.

The following table lists HtmlHelper methods and html control each method generates.

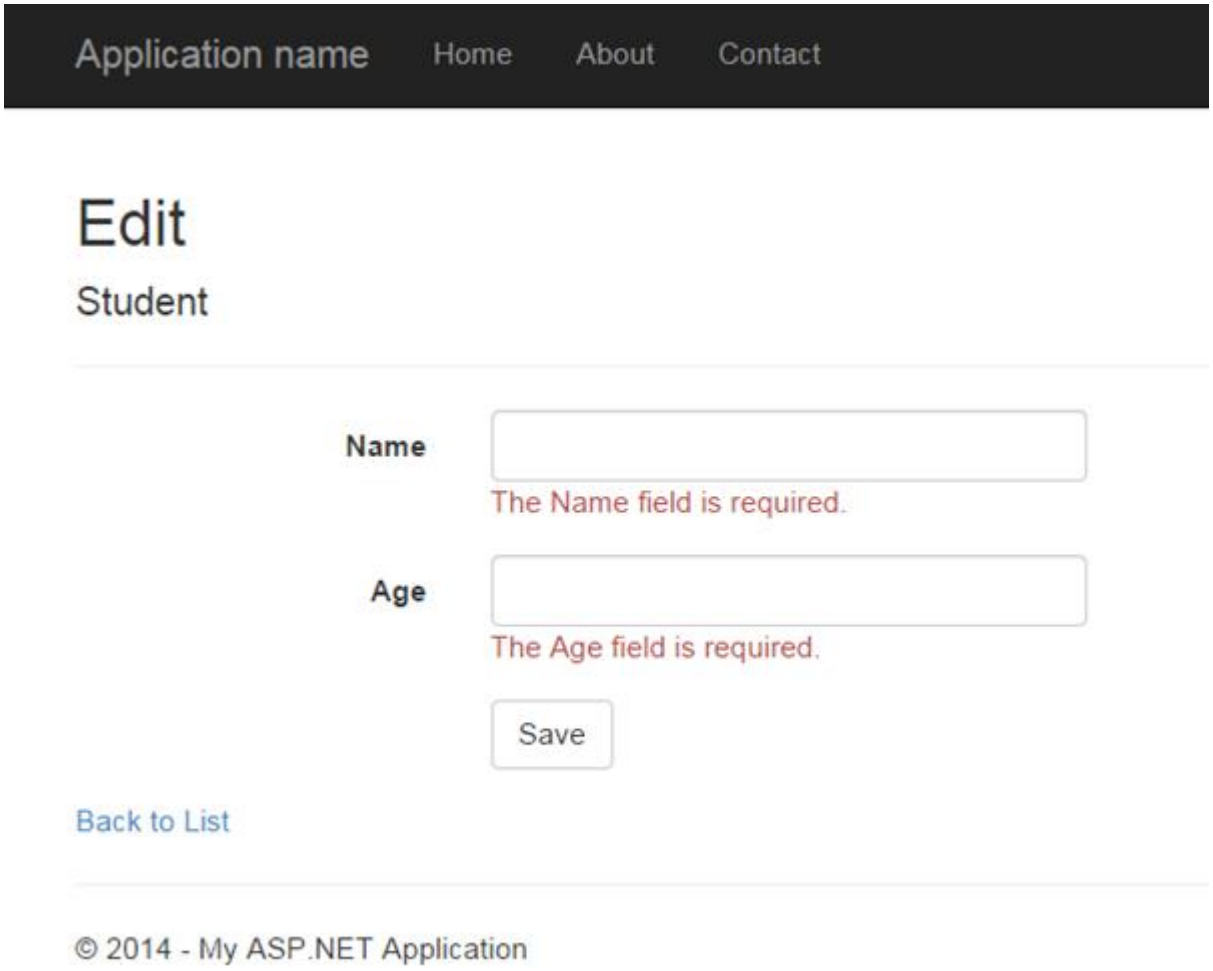
| HtmlHelper | Strongly Typed HtmlHelpers | Html Control |
|---|---|---|
| Html.ActionLink | | Anchor link |
| Html.TextBox | Html.TextBoxFor | Textbox |
| Html.TextArea | Html.TextAreaFor | TextArea |
| Html.CheckBox | Html.CheckBoxFor | Checkbox |
| Html.RadioButton | Html.RadioButtonFor | Radio button |
| Html.DropDownList | Html.DropDownListFor | Dropdown, combobox |
| Html.ListBox | Html.ListBoxFor | multi-select list box |
| Html.Hidden | Html.HiddenFor | Hidden field |
| Password | Html.PasswordFor | Password textbox |
| Html.Display | Html.DisplayFor | Html text |
| Html.Label | Html.LabelFor | Label |
| Html.Editor | Html.EditorFor | Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type. |

The difference between calling the HtmlHelper methods and using an html tags is that the HtmlHelper method is designed to make it easy to bind to view data or model data.

# Implement Data Validation in MVC

In this section, you will learn how to implement data validations in the ASP.NET MVC application.

We have created an Edit view for Student in the previous section. Now, we will implement data validation in the Edit view, which will display validation messages on the click of Save button, as shown below if Student Name or Age is blank.




Validation

## DataAnnotations

ASP.NET MVC uses DataAnnotations attributes to implement validations. DataAnnotations includes built-in validation attributes for different validation rules, which can be applied to the properties of model class. ASP.NET MVC framework will automatically enforce these validation rules and display validation messages in the view.

The DataAnnotations attributes included in *System.ComponentModel.DataAnnotations*namespace. The following table lists DataAnnotations validation attributes.

| Attribute | Description |
|---|---|
| Required | Indicates that the property is a required field |
| StringLength | Defines a maximum length for string field |
| Range | Defines a maximum and minimum value for a numeric field |
| RegularExpression | Specifies that the field value must match with specified Regular Expression |
| CreditCard | Specifies that the specified field is a credit card number |
| CustomValidation | Specified custom validation method to validate the field |
| EmailAddress | Validates with email address format |
| FileExtension | Validates with file extension |
| MaxLength | Specifies maximum length for a string field |
| MinLength | Specifies minimum length for a string field |
| Phone | Specifies that the field is a phone number using regular expression for phone numbers |

Let's start to implement validation in Edit view for student.

**Step 1:** First of all, apply DataAnnotation attribute on the properties of Student model class. We want to validate that StudentName and Age is not blank. Also, Age should be between 5 and 50. Visit Model section if you don't know how to create a model class.

## Example: Apply DataAnnotation Attributes

```
public class Student
{
    public int StudentId { get; set; }

    [Required]
    public string StudentName { get; set; }

    [Range(5,50)]
    public int Age { get; set; }
}
```

You can also apply multiple DataAnnotations validation attributes to a single property if required.

In the above example, we have applied a ***Required*** attribute to the StudentName property. So now, the MVC framework will automatically display the default error message, if the user tries to save the Edit form

without entering the Student Name. In the same way, the ***Range*** attribute is applied with a min and max value to the Age property. This will validate and display an error message if the user has either not entered Age or entered an age less than 5 or more than 50.

**Step 2:** Create the GET and POST Edit Action method in the same as previous section. The GET action method will render Edit view to edit the selected student and the POST Edit method will save edited student as shown below.

## Example: Edit Action methods:

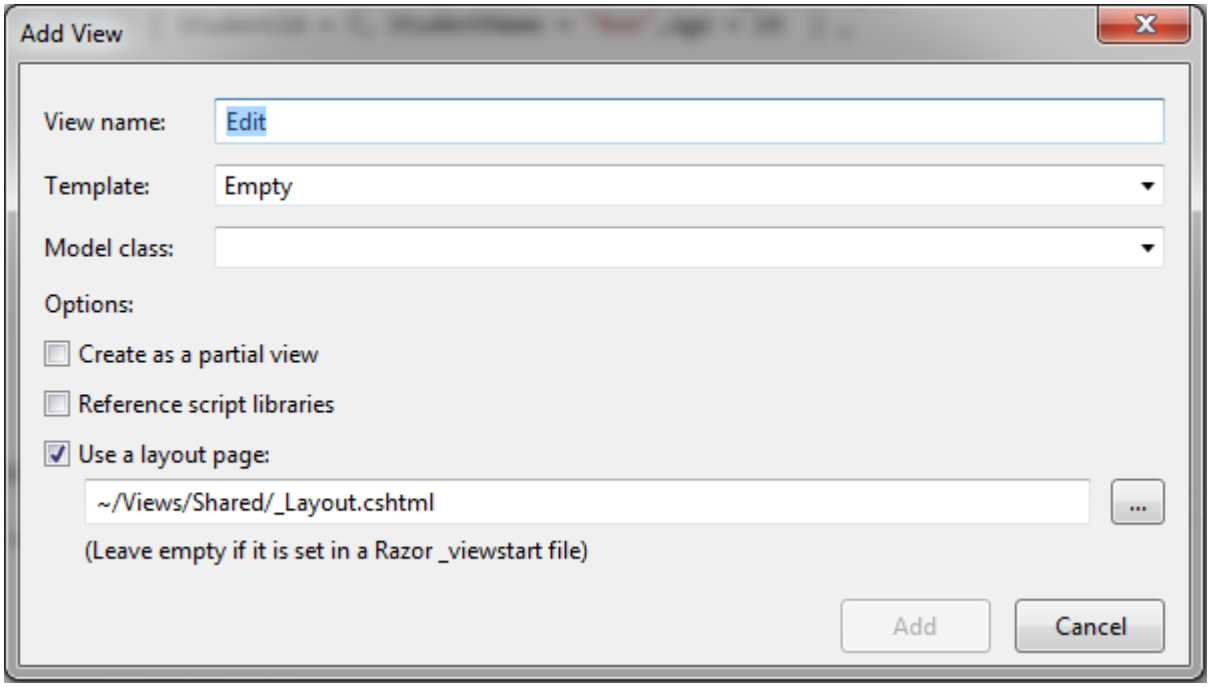
```
using MVC_BasicTutorials.Models;

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        public ActionResult Edit(int id)
        {
            var std = studentList.Where(s => s.StudentId == StudentId)
                                 .FirstOrDefault();

            return View(std);
        }



        [HttpPost]
        public ActionResult Edit(Student std)
        {
            if (ModelState.IsValid) {

                //write code to update student

                return RedirectToAction("Index");
            }

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

As you can see in the POST Edit method, we first check if the ModelState is valid or not. If ModelState is valid then update the student into database, if not then return Edit view again with the same student data.

ModelState.IsValid determines that whether submitted values satisfy all the DataAnnotation validation attributes applied to model properties.

**Step 3:** Now, create an Edit view for Student.

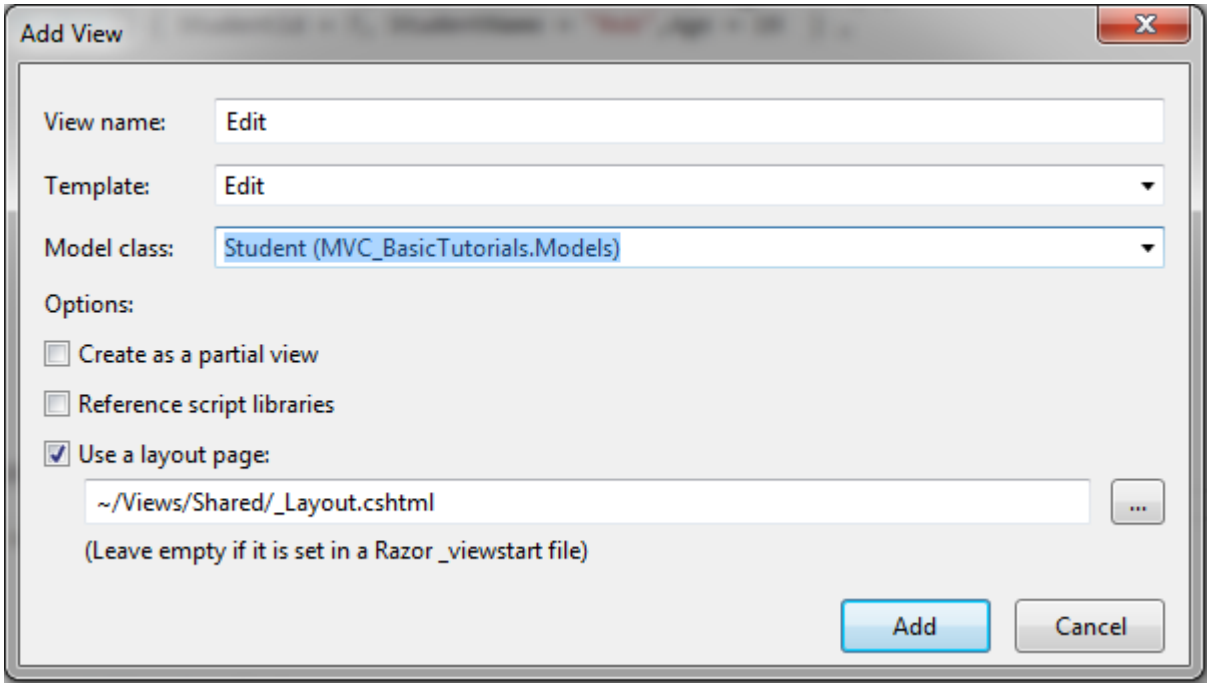
To create an Edit view, right click inside Edit action method -> click **Add View..**




Create Edit View

In the Add View dialogue, keep the view name as Edit. (You can change as per your requirement.)

Select the Edit template in the Template dropdown and also select Student Model class as shown below.




Create Edit View

Now, click **Add** to generate Edit view under View/Student folder. Edit.cshtml will be generated as shown below.

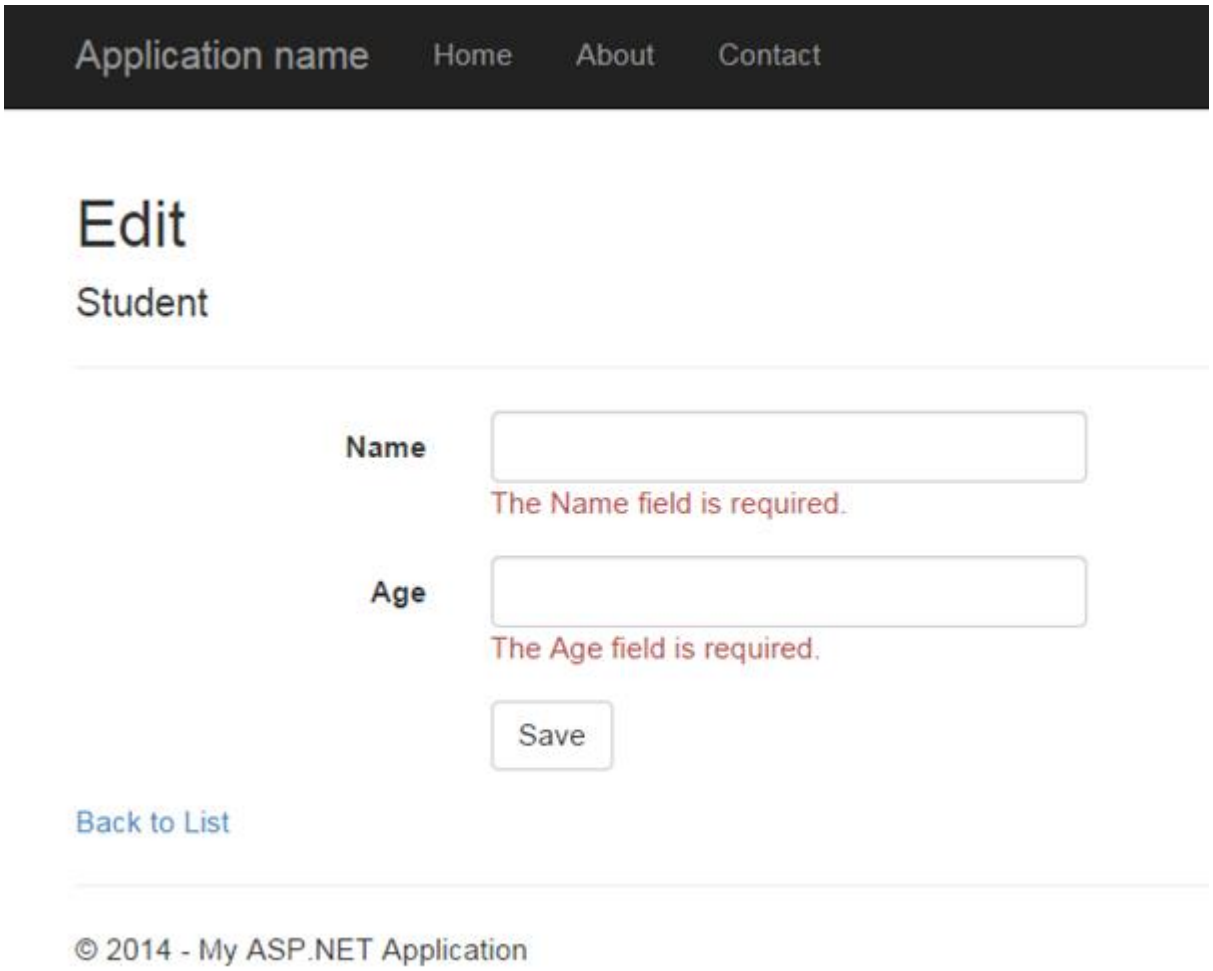
# Edit.cshtml:

```
@model MVC_BasicTutorials.Models.Student

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

<h2>Edit</h2>

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

    <div class="form-horizontal">
        <h4>Student</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class = "text-danger" })
        @Html.HiddenFor(model => model.StudentId)

        <div class="form-group">
            @Html.LabelFor(model => model.StudentName, htmlAttributes: new { @class
= "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.StudentName, new { htmlAttributes =
new { @class = "form-control" } })
                @Html.ValidationMessageFor(model => model.StudentName, "", new {
@class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Age, htmlAttributes: new { @class =
"control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Age, new { htmlAttributes = new {
@class = "form-control" } })
                @Html.ValidationMessageFor(model => model.Age, "", new { @class =
"text-danger" })
            </div>
        </div>

        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

As you can see in the above Edit.cshtml, it calls Html Helper method **ValidationMessageFor**for every field and **ValidationSummary** method at the top. ValidationMessageFor is

responsible to display error message for the specified field. ValidationSummary displays a list of all the error messages at once.

So now, it will display default validation message when you submit an Edit form without entering a Name or Age.




Validation

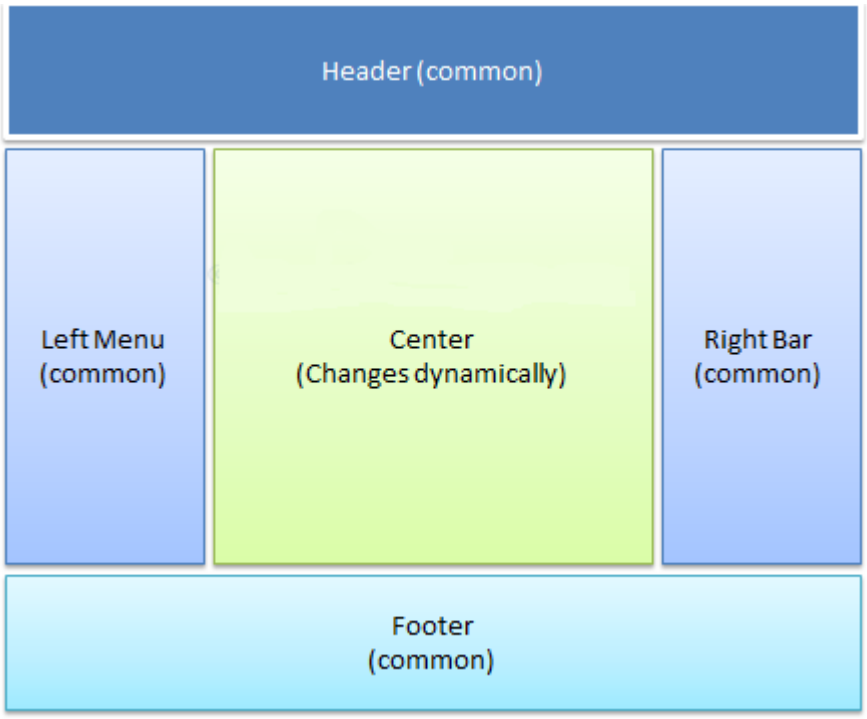
Thus, you can implement validations by applying various DataAnnotation attributes to the model class and using ValidationMessage() or ValidationMessageFor() method in the view.

# Layout View

In this section, you will learn about the layout view in ASP.NET MVC.

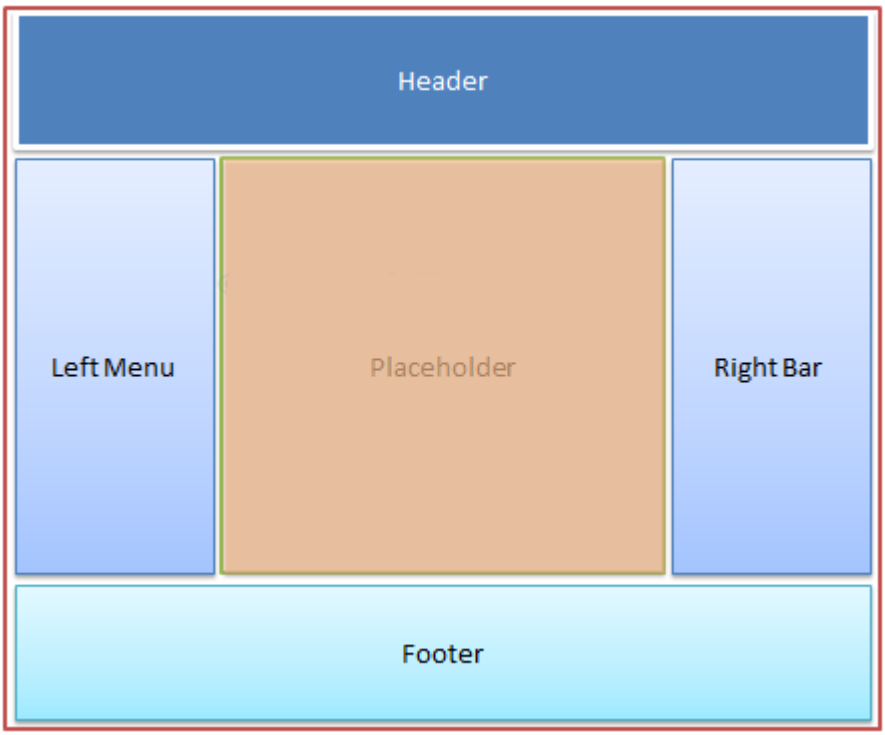
An application may contain common parts in the UI which remains the same throughout the application such as the logo, header, left navigation bar, right bar or footer section. ASP.NET MVC introduced a Layout view which contains these common UI parts, so that we don't have to write the same code in every page. The layout view is same as the master page of the ASP.NET webform application.

For example, an application UI may contain Header, Left menu bar, right bar and footer section that remains same in every page and only the centre section changes dynamically as shown below.



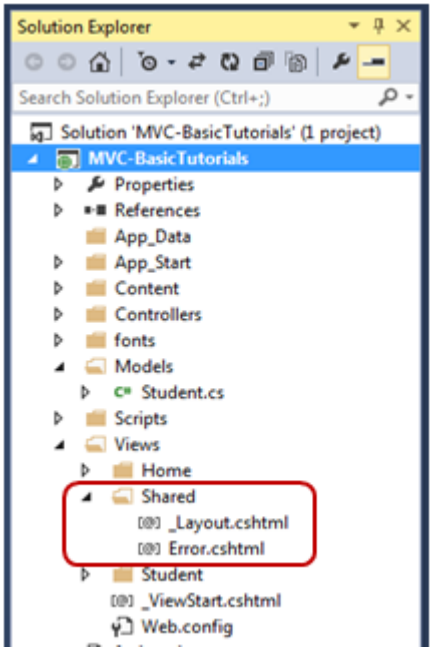

Sample Application UI Parts

The layout view allows you to define a common site template, which can be inherited in multiple views to provide a consistent look and feel in multiple pages of an application. The layout view eliminates duplicate coding and enhances development speed and easy maintenance. The layout view for the above sample UI would contain a Header, Left Menu, Right bar and Footer sections. It contains a placeholder for the center section that changes dynamically as shown below.




Layout View

The razor layout view has same extension as other views, .cshtml or .vbhtml. Layout views are shared with multiple views, so it must be stored in the Shared folder. For example, when we created our first MVC application in the previous section, it also created _Layout.cshtml in the Shared folder as shown below.




Layout Views in Shared Folder

The following is an auto-generated _Layout.cshtml.

## _Layout.cshtml:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Application name", "Index", "Home", new { area =
"" }, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

As you can see, the layout view contains html Doctype, head and body as normal html, the only difference is call to RenderBody() and RenderSection() methods. RenderBody acts like a placeholder for other views. For example, Index.cshtml in the home folder will be injected and rendered in the layout view, where the RenderBody() method is being called. You will learn about these rendering methods later in this section.

## Use Layout View

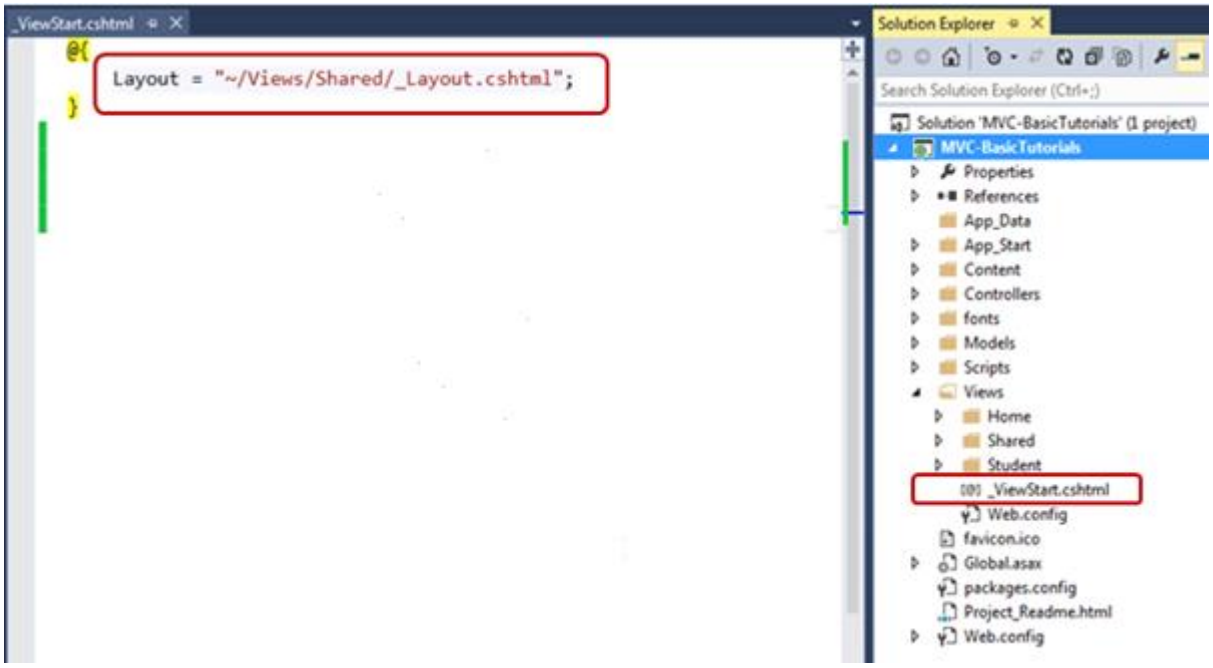
You must be wondering that how would the View know which layout view to use?

You can set the layout view in multiple ways, by using _ViewStart.cshtml or setting up path of the layout page using Layout property in the individual view or specifying layout view name in the action method.

## _ViewStart.cshtml

_ViewStart.cshtml is included in the Views folder by default. It sets up the default layout page for all the views in the folder and its subfolders using the Layout property. You can assign a valid path of any Layout page to the Layout property.

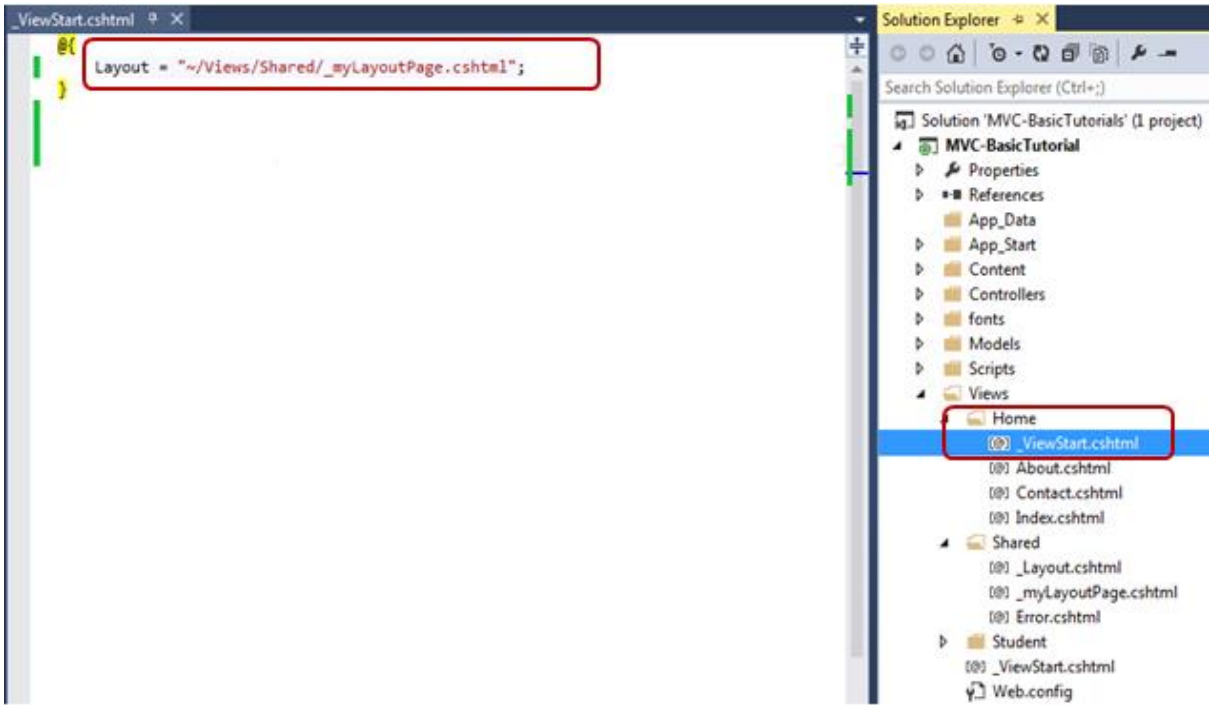
For example, the following _ViewStart.cshtml in the **Views** folder, sets the Layout property to "~/Views/Shared/_Layout.cshtml". So now, _layout.cshtml would be layout view of all the views included in Views and its subfolders. So by default, all the views derived default layout page from _ViewStart.cshtml of Views folder.




_ViewStart.cshtml

_ViewStart.cshtml can also be included in sub folder of View folder to set the default layout page for all the views included in that particular subfolder only.

For example, the following _ViewStart.cshtml in Home folder, sets Layout property to _myLayoutPage.cshtml. So this _ViewStart.cshtml will influence all the views included in the Home folder only. So now, Index, About and Contact views will be rendered in _myLayoutPage.cshtml instead of default _Layout.cshml.



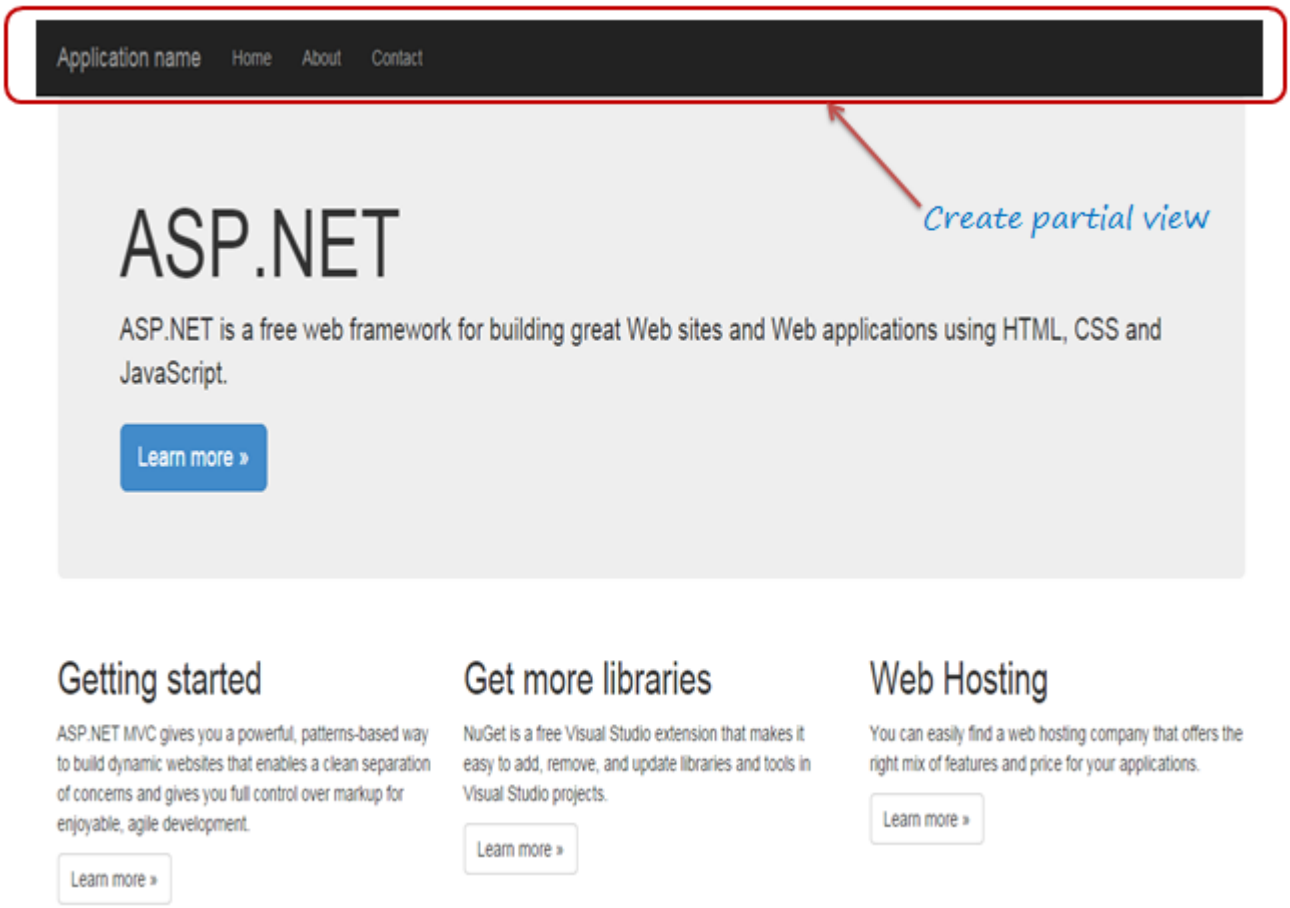

Layout View

# Partial View

In this section you will learn about partial views in ASP.NET MVC.

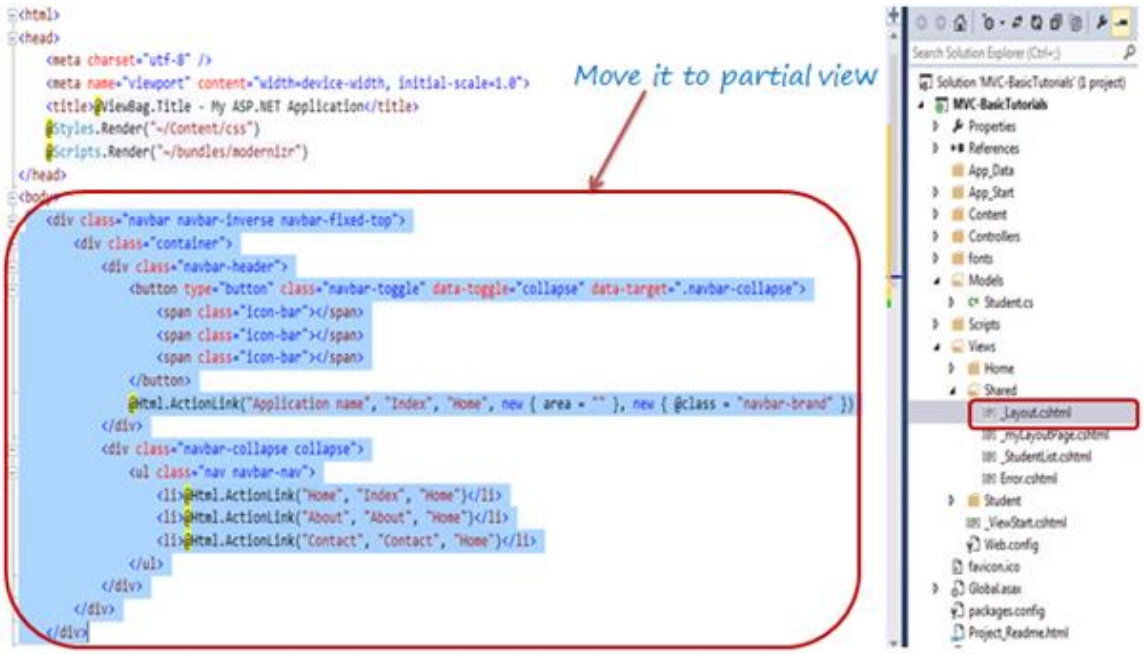
Partial view is a reusable view, which can be used as a child view in multiple other views. It eliminates duplicate coding by reusing same partial view in multiple places. You can use the partial view in the layout view, as well as other content views.

To start with, let's create a simple partial view for the following navigation bar for demo purposes. We will create a partial view for it, so that we can use the same navigation bar in multiple layout views without rewriting the same code everywhere.




Partial View

The following figure shows the html code for the above navigation bar. We will cut and paste this code in a seperate partial view for demo purposes.



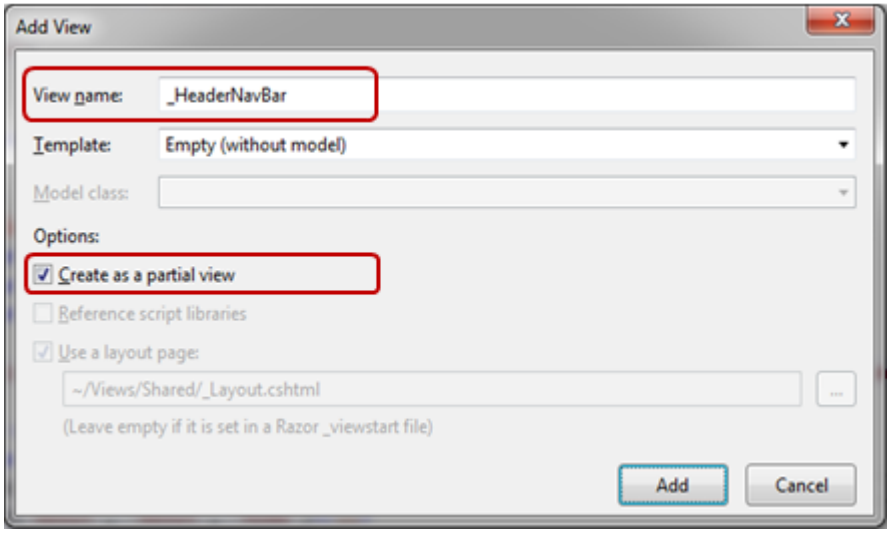
# Create a New Partial View

To create a partial view, right click on Shared folder -> select **Add** -> click on **View..**

## Note:

If a partial view will be shared with multiple views of different controller folder then create it in the Shared folder, otherwise you can create the partial view in the same folder where it is going to be used.

In the Add View dialogue, enter View name and select "Create as a partial view" checkbox and click Add.



Partial Views

We are not going to use any model for this partial view, so keep the Template dropdown as Empty (without model) and click **Add**. This will create an empty partial view in Shared folder.

Now, you can cut the above code for navigation bar and paste it in _HeaderNavBar.cshtml as shown below:

## Example: Partial View _HeaderNavBar.cshtml

```
<div class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            @Html.ActionLink("Application name", "Index", "Home", new { area = ""
}, new { @class = "navbar-brand" })
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
```

```
                <li>@Html.ActionLink("About", "About", "Home")</li>
                <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
            </ul>
        </div>
    </div>
</div>
```

Thus, you can create a new partial view. Let's see how to render partial view.

## Render Partial View

You can render the partial view in the parent view using html helper methods: Partial() or RenderPartial()

### Html.Partial()

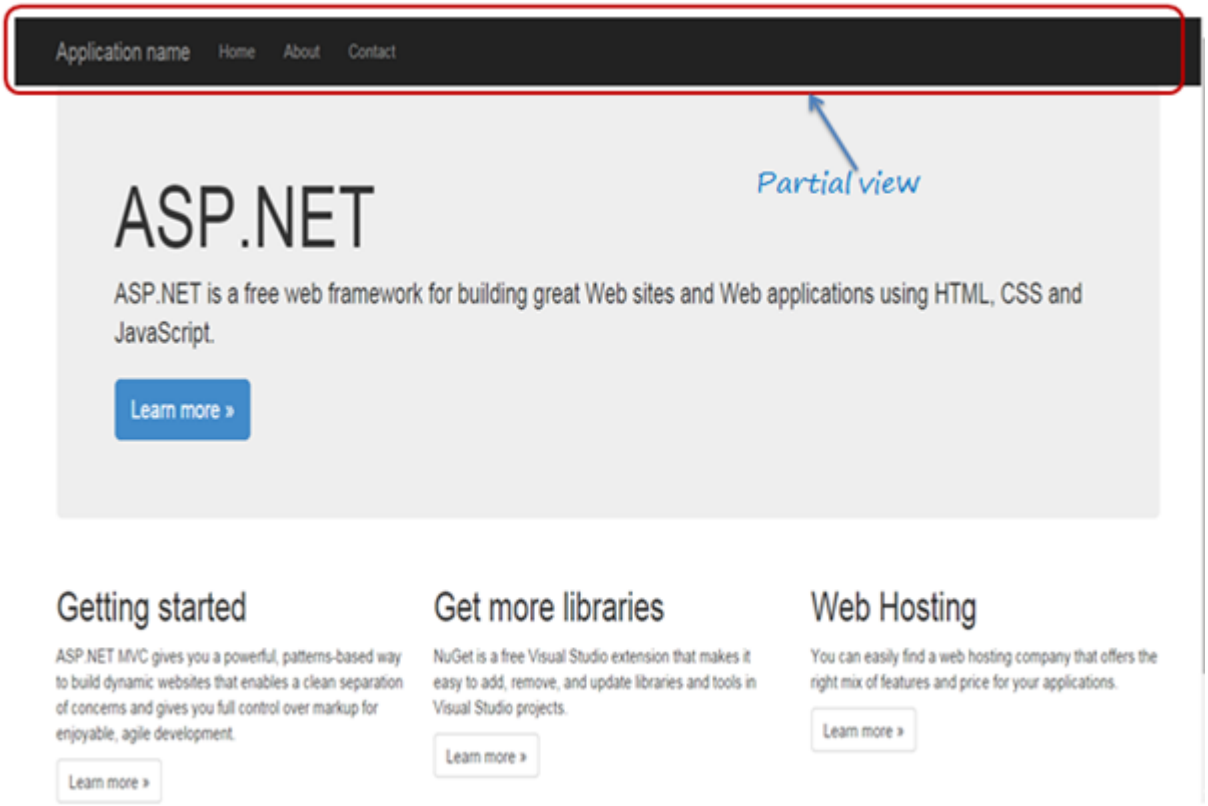
### Example: Html.Partial()

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    @Html.Partial("_HeaderNavBar")
    <div class="container body-content">
        @RenderBody()

        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>
    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

### Note:

@Html.Partial() method doesn't need to be in code block because it returns a html string.

You will see following UI in browser when you run the application.




Index.cshtml

So in this way, you can use partial view without any differences in the UI.

## Points to Remember :

1. Partial view is a reusable view, which can be used as a child view in multiple other views.