

Asp.net Mvc

ASP.NET MVC Tutorials

ASP.NET MVC tutorials cover all the features of ASP.NET MVC. You will learn basic to advance level features of ASP.Net MVC. Basic tutorials have used MVC 5, but it is applicable to all the previous versions and upcoming versions of MVC as well.

MVC Architecture:

In this section, you will get an overview of MVC architecture. The MVC architectural pattern has existed for a long time in software engineering. All most all the languages use MVC with slight variation, but conceptually it remains the same.

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

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

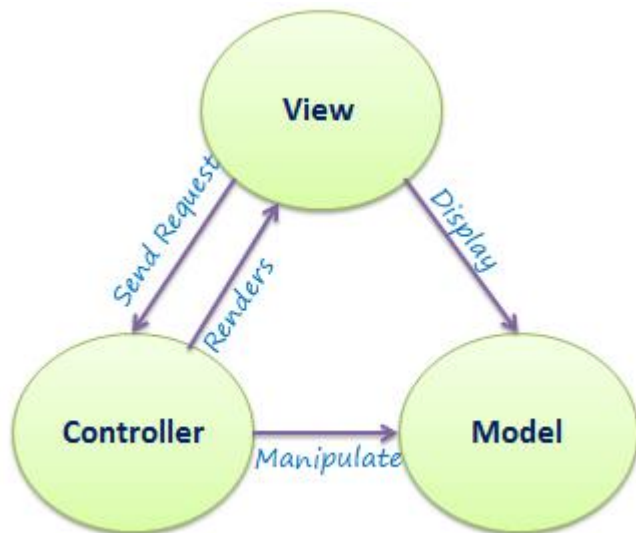
View: View is a user interface. View display data using model to the user and also enables them to modify the data.

View is a User Interface.

Controller: Controller handles the user request. Typically, user interact with View, which in-tern 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.

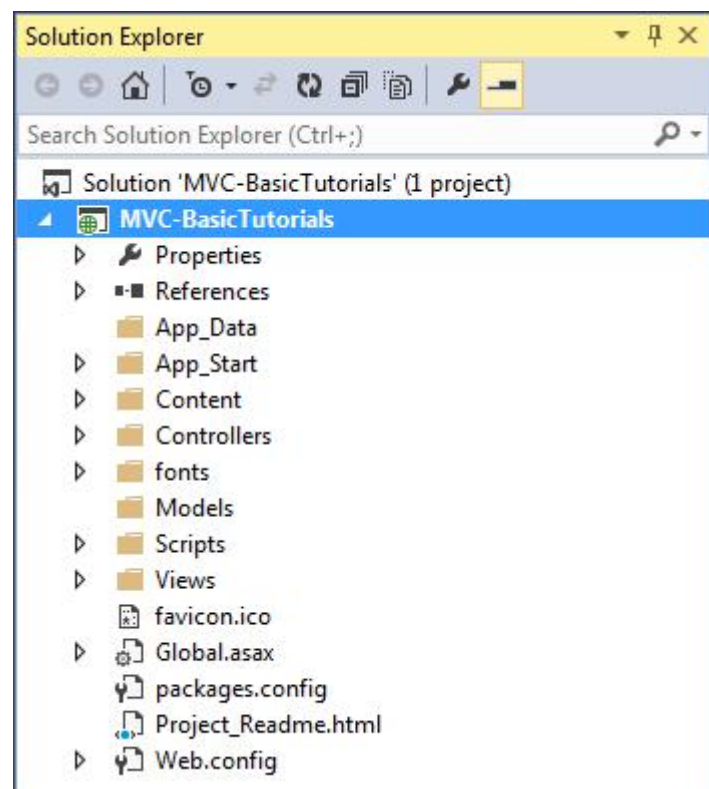
Controller is a request handler.

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



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.

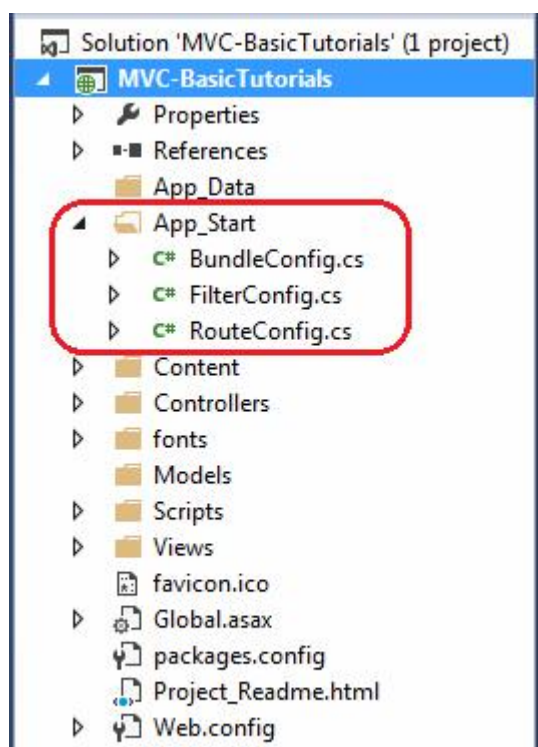


App_Data:

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.

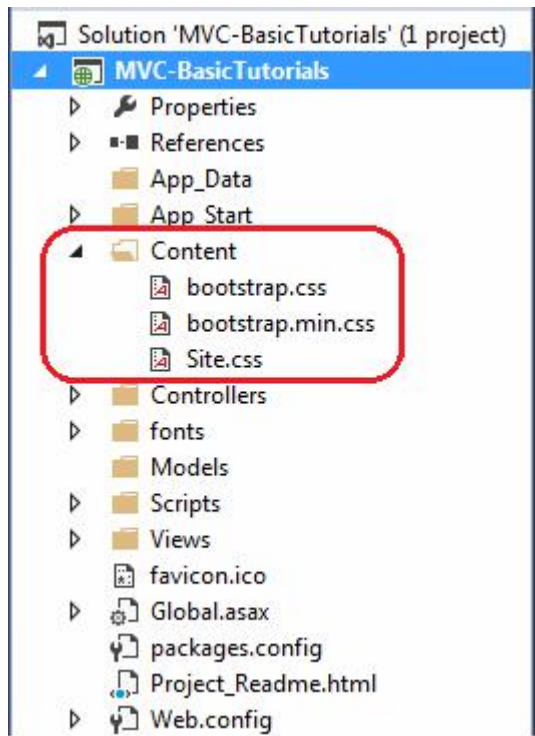
App_Start:

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.



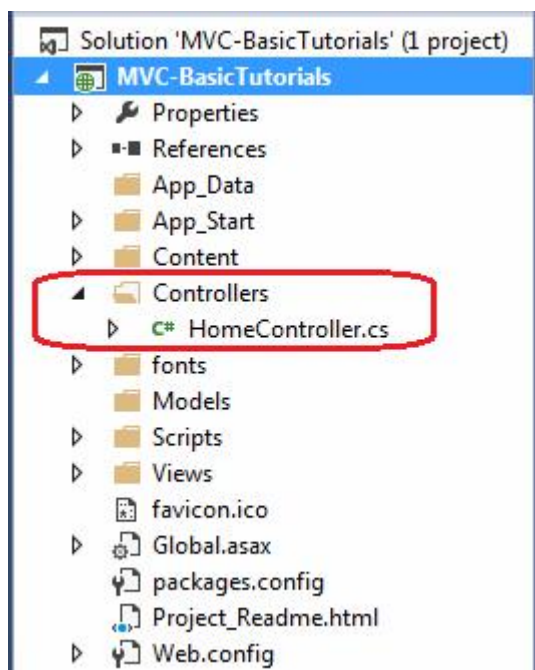
Content:

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.



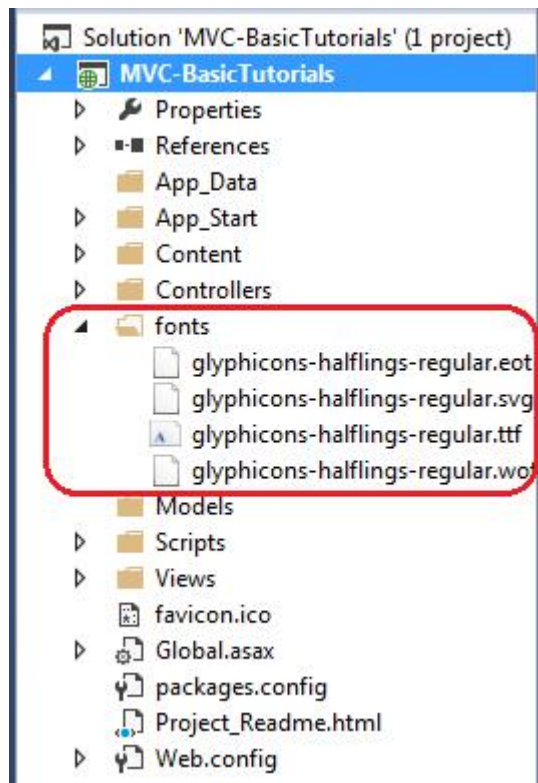
Controllers:

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.



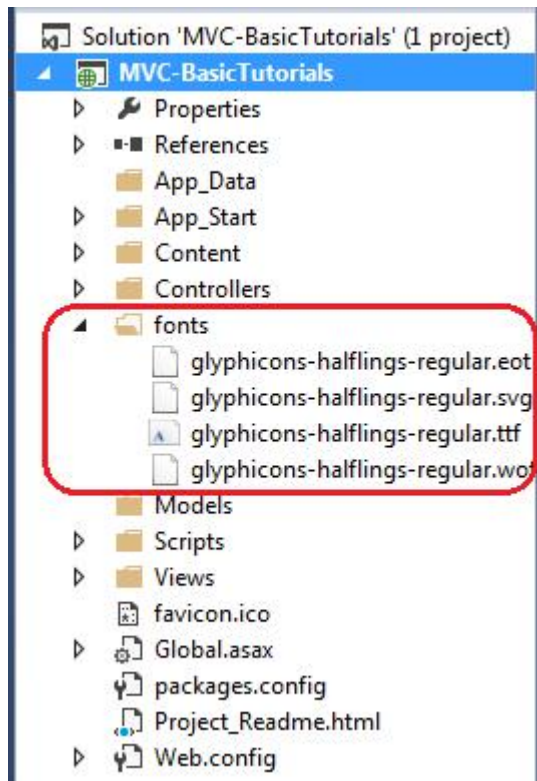
fonts:

Fonts folder contains custom font files for your application.



fonts:

Fonts folder contains custom font files for your application.

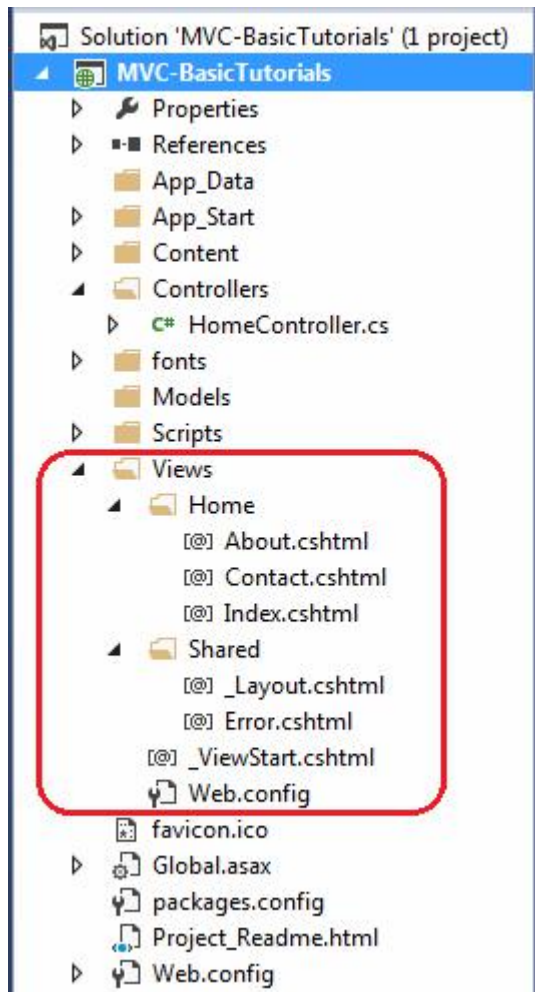


Views:

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.



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.

Web.config:

Web.config file contains application level configurations.

Learn how MVC framework handles request using routing in the next section.

Routing in MVC:

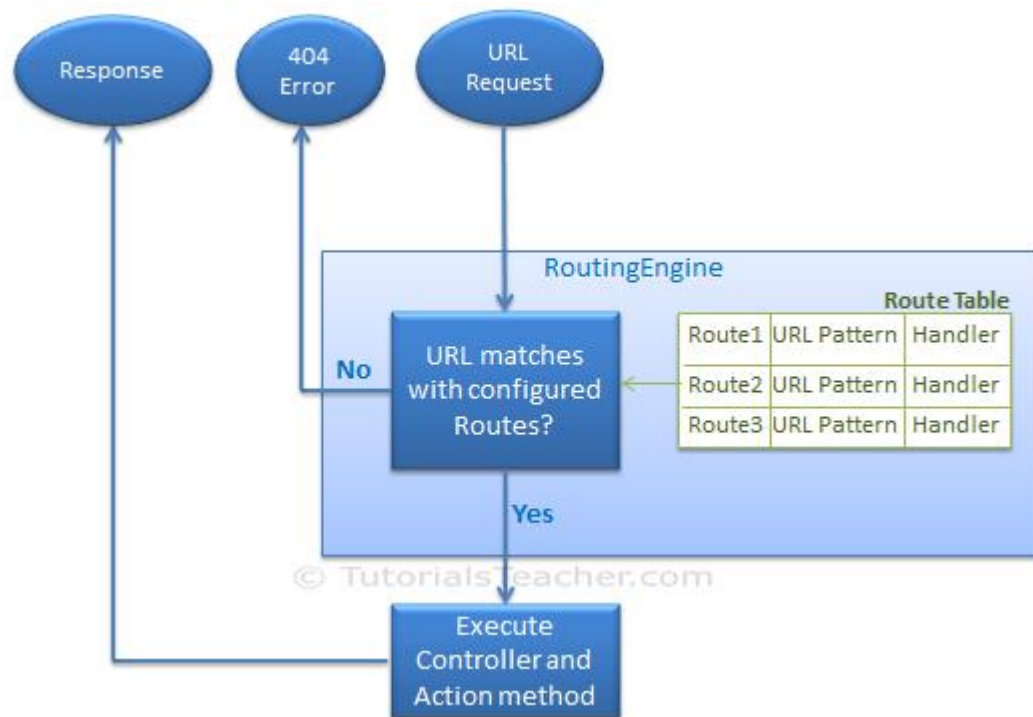
In the ASP.NET Web Forms application, every URL must match with a specific .aspx file. For example, a URL `http://domain/studentsinfo.aspx` must match with the file `studentsinfo.aspx` that contains code and markup for rendering a response to the browser.

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.



Configure 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 .

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
        );
    }
}

```

Route to ignore (points to IgnoreRoute)

Route name (points to name: "Default")

URL Pattern (points to url: "{controller}/{action}/{id}")

Defaults for Route (points to defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional })

RouteConfig.cs

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.



If the URL doesn't contain anything after domain name then the default controller and action method will handle the request. For example, `http://localhost: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
<code>http://localhost/home</code>	HomeController	Index	null
<code>http://localhost/home/index/123</code>	HomeController	Index	123
<code>http://localhost/home/about</code>	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:

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.

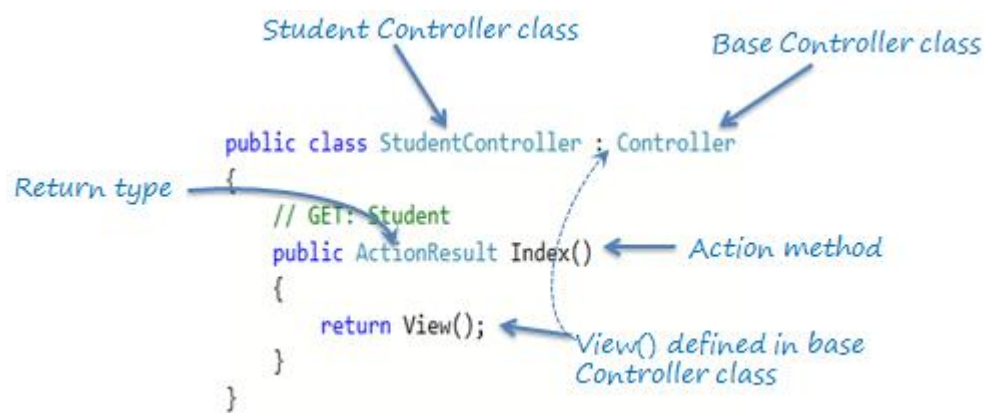
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Action method:

All the public methods of a Controller class are called Action methods. They are like any other normal methods with the following restrictions:

1. Action method must be public. It cannot be private or protected
2. Action method cannot be overloaded
3. Action method cannot be a static method.

The following is an example of Index action method of StudentController



Default Action method:

Every controller can have default action method as per configured route in RouteConfig class. By default, Index is a default action method for any controller, as per configured default root as shown below.

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}/{name}",
    defaults: new { controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
    });
```

ActionResult:

MVC framework includes various result classes, which can be return from an action methods. There result classes represent different types of responses

such as html, file, string, json, javascript etc. The following table lists all the result classes available in ASP.NET MVC.

Result Class	Description
ViewResult	Represents HTML and markup.
EmptyResult	Represents No response.
ContentResult	Represents string literal.
FileContentResult/ FilePathResult/ FileStreamResult	Represents the content of a file
JavaScriptResult	Represent a JavaScript script.
JsonResult	Represent JSON that can be used in AJAX
RedirectResult	Represents a redirection to a new URL
RedirectToRouteResult	Represent another action of same or other controller
PartialViewResult	Returns HTML from Partial view
HttpUnauthorizedResult	Returns HTTP 403 status

The Index() method of StudentController in the above figure uses View() method to return ViewResult (which is derived from ActionResult). The View() method is defined in base Controller class. It also contains different methods, which automatically returns particular type of result as shown in the below table.

Result Class	Description	Base Controller method
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	

Result Class	Description	Base Controller method
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file	File()
JavaScriptResult	Represent a JavaScript script.	JavaScript()
JsonResult	Represent JSON that can be used in AJAX	Json()
RedirectResult	Represents a redirection to a new URL	Redirect()
RedirectToRouteResult	Represent another action of same or other controller	RedirectToRoute()
PartialViewResult	Returns HTML	PartialView()
HttpUnauthorizedResult	Returns HTTP 403 status	

As you can see in the above table, View method returns ViewResult, Content method returns string, File method returns content of a file and so on. Use different methods mentioned in the above table, to return different types of results from an action method.

Action method Parameters:

Every action methods can have input parameters as normal methods. It can be primitive data type or complex type parameters as shown in the below example.

```
[HttpPost]public ActionResult Edit(Student std)
{
    // update student to the database

    return RedirectToAction("Index");
}

[HttpDelete]public ActionResult Delete(int id)
{
    // delete student from the database whose id matches with specified id
}
```

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


Action Selectors:

Action selector is the attribute that can be applied to the action methods. It helps routing engine to select the correct action method to handle a particular request. MVC 5 includes the following action selector attributes:

1. ActionName
2. NonAction
3. ActionVerbs

ActionName:

ActionName attribute allows us to specify a different action name than the method name. Consider the following example.

```
public class StudentController : Controller
{
    public StudentController()
    {
    }

    [ActionName("find")]
    public ActionResult GetById(int id)
    {
        // get student from the database
        return View();
    }
}
```

In the above example, we have applied `ActionName("find")` attribute to GetById action method. So now, action name is "find" instead of "GetById". This action method will be invoked on `http://localhost/student/find/1` request instead of `http://localhost/student/getbyid/1` request.

NonAction:

NonAction selector attribute indicates that a public method of a Controller is not an action method. Use NonAction attribute when you want public method in a controller but do not want to treat it as an action method.

For example, the GetStudent() public method cannot be invoked in the same way as action method in the following example.

```
public class StudentController : Controller
{
    public StudentController()
    {
    }

    [NonAction]
    public Student GetStudnet(int id)
    {
        return studentList.Where(s => s.StudentId == id)
            .FirstOrDefault();
    }
}
```

Action Verbs:

The Action Verbs selector is used when you want to control the selection of an action method based on a Http request method. For example, you can define two different action methods with the same name but one action method responds to an HTTP Get request and another action method responds to an HTTP Post request.

MVC framework supports different Action Verbs, such as HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions & HttpPatch. You can apply these attributes to action method to indicate the kind of Http request the action method supports. If you do not apply any attribute then it considers it a GET request by default.

The following figure illustrates the HttpGET and HttpPOST action verbs.



The following table lists the usage of http methods:

Http method	Usage
GET	To retrieve the information from the server. Parameters will be appended in the query string.
POST	To create a new resource.
PUT	To update an existing resource.

Http method	Usage
HEAD	Identical to GET except that server do not return message body.
OPTIONS	OPTIONS method represents a request for information about the communication options supported by web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

```

public class StudentController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    [HttpPost]
    public ActionResult PostAction()
    {
        return View("Index");
    }

    [HttpPut]
    public ActionResult PutAction()
    {
        return View("Index");
    }

    [HttpDelete]
    public ActionResult DeleteAction()
    {
        return View("Index");
    }

    [HttpHead]
    public ActionResult HeadAction()
    {
        return View("Index");
    }

    [HttpOptions]
    public ActionResult OptionsAction()

```

```
{  
    return View("Index");  
}  
  
[HttpPatch]  
public ActionResult PatchAction()  
{  
    return View("Index");  
}  
}
```

You can also apply multiple http verbs using `AcceptVerbs` attribute. `GetAndPostAction` method supports both, GET and POST ActionVerbs in the following example:

```
[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Get)]  
public ActionResult GetAndPostAction()  
{  
    return RedirectToAction("Index");  
}
```

Model in ASP.NET MVC

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 persistence 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.

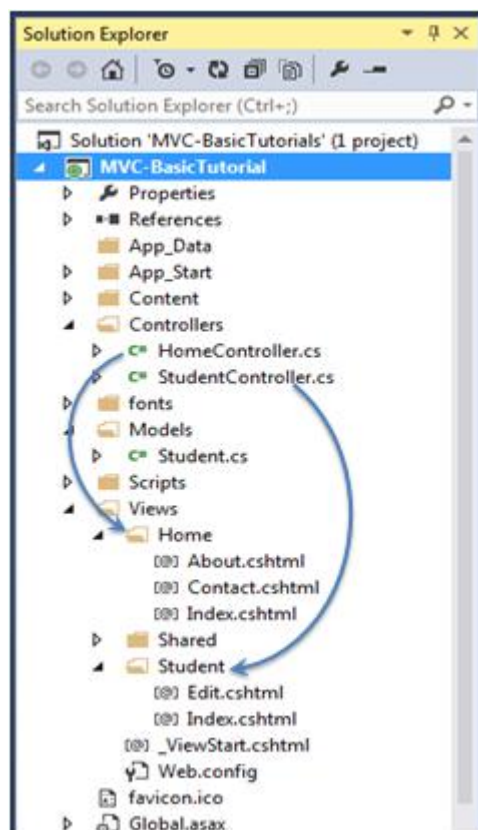
Adding Model:

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

View in ASP.NET MVC:

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 reside in Views > Student folder as shown below.



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

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 StudentController. So, open a StudentController class -> right click inside Index method -> click **Add View..**

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



Add View

View name: Index

Template: Empty (without model)

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

~/Views/Shared/_Layout.cshtml

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

Add Cancel

Add View

View name: Index

Template: List

Model class: Student (MVC_BasicTutorials.Models)

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

~/Views/Shared/_Layout.cshtml

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

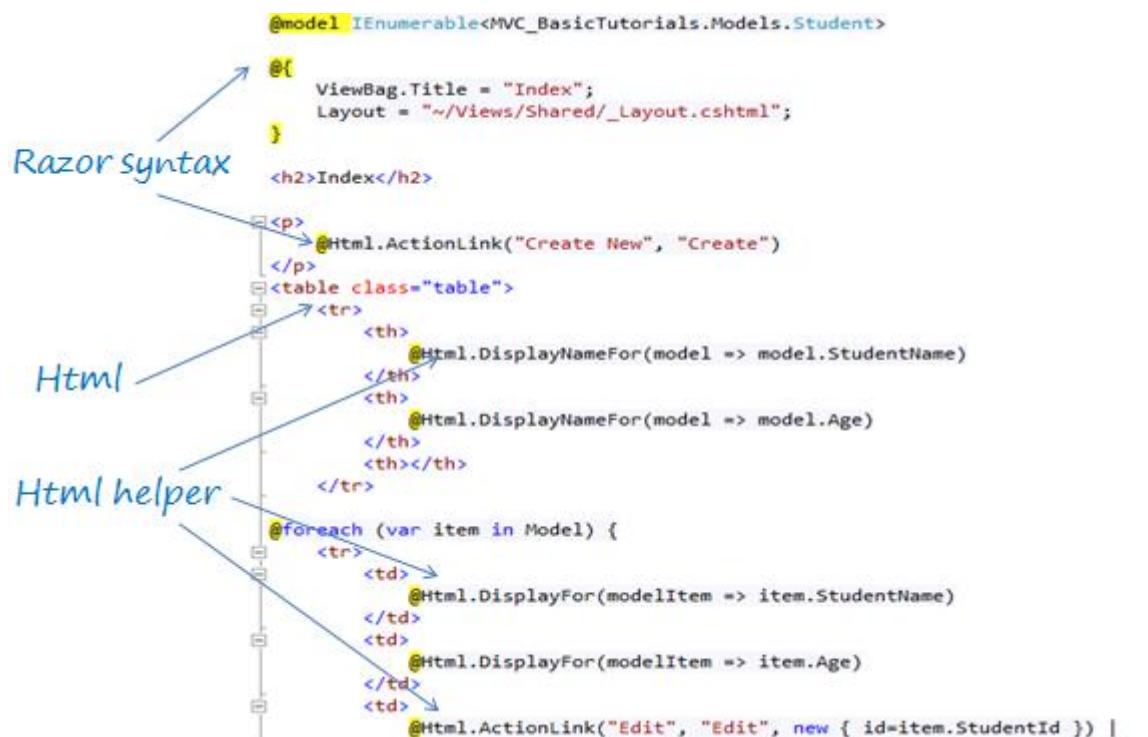
Add Cancel

```
@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>
</tr>
</table>
```

```

        <th>
            @Html.DisplayNameFor(model => model.Age)
        </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.Stude
ntId }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.Stude
ntId })
            </td>
        </tr>
    }
</table>

```



Index

[Create New](#)

Name	Age	
John	18	Edit Details Delete
Steve	21	Edit Details Delete
Bill	25	Edit Details Delete
Ram	20	Edit Details Delete
Ron	31	Edit Details Delete
Chris	17	Edit Details Delete
Rob	19	Edit Details Delete

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.

```
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();
        }
    }
}
```

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

```
@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>
    </tr>
</table>
```

```

        <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.

```

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);
}
}

```

As you can see in the above code, we have created a List of student objects for an example purpose (in real life application, 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.

Index

[Create New](#)

Name	Age	
John	18	Edit Details Delete
Steve	21	Edit Details Delete
Bill	25	Edit Details Delete
Ram	20	Edit Details Delete
Ron	31	Edit Details Delete
Chris	17	Edit Details Delete
Rob	19	Edit Details Delete

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.

```
<h1>Razor syntax demo</h1>
<h2>@DateTime.Now.ToShortDateString()</h2>
```

RESULT:

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#.

```
@{
    var date = DateTime.Now.ToShortDateString();
    var message = "Hello World";
}
<h2>Today's date is: @date </h2><h3>@message</h3>
```

RESULT:

Razor syntax demo

Today's date is: 08-09-2014

Hello World!

Display text from code block:

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

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

Razor syntax demo

Today's date is: 08-09-2014

Hello World!

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

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

Razor syntax demo

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.

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

2014 is not a leap year.

for loop:

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

RESULT:

0
1
2
3
4

Model:

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

```
@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>
```

RESULT:

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.

```
@{
    string str = "";

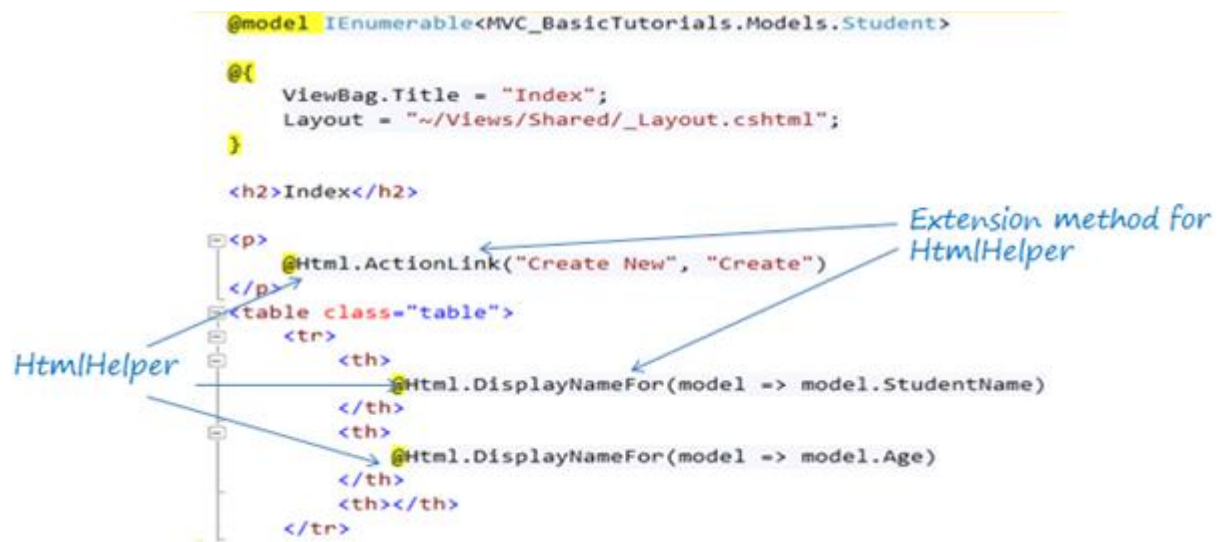
    if(1 > 0)
    {
        str = "Hello World!";
    }
}
<p>@str</p>
```

Hello World!

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.



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 `Create New`.

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.

<i>HtmlHelper</i>	<i>Strogly Typed HtmlHelpers</i>	<i>Html Control</i>
<i>Html.ActionLink</i>		<i>Anchor link</i>
<i>Html.TextBox</i>	<i>Html.TextBoxFor</i>	<i>Textbox</i>
<i>Html.TextArea</i>	<i>Html.TextAreaFor</i>	<i>TextArea</i>
<i>Html.CheckBox</i>	<i>Html.CheckBoxFor</i>	<i>Checkbox</i>
<i>Html.RadioButton</i>	<i>Html.RadioButtonFor</i>	<i>Radio button</i>
<i>Html.DropDownList</i>	<i>Html.DropDownListFor</i>	<i>Dropdown, combobox</i>
<i>Html.ListBox</i>	<i>Html.ListBoxFor</i>	<i>multi-select list box</i>
<i>Html.Hidden</i>	<i>Html.HiddenFor</i>	<i>Hidden field</i>
<i>Password</i>	<i>Html.PasswordFor</i>	<i>Password textbox</i>
<i>Html.Display</i>	<i>Html.DisplayFor</i>	<i>Html text</i>
<i>Html.Label</i>	<i>Html.LabelFor</i>	<i>Label</i>
<i>Html.Editor</i>	<i>Html.EditorFor</i>	<i>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.</i>

Create TextBox using HtmlHelper:

Learn how to generate textbox control using HtmlHelper in razor view in this section.

HtmlHelper class includes two extension methods which creates a textbox (<input type="text">) element in razor view: TextBox() and TextBoxFor(). The TextBox() method is loosely typed method whereas TextBoxFor() is a strongly typed method.

We will use following Student model with TextBox() and TextBoxFor() method.

```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public int Age { get; set; }
    public bool isNewlyEnrolled { get; set; }
    public string Password { get; set; }
}
```

TextBox():

The Html.TextBox() method creates <input type="text" > element with specified name, value and html attributes.

TextBox() method signature:

```
MvcHtmlString Html.TextBox(string name, string value, object htmlAttributes)
```

The TextBox() method is a loosely typed method because name parameter is a string. The name parameter can be a property name of model object. It binds specified property with textbox. So it automatically displays a value of the model property in a textbox and visa-versa.

```
@model Student
@Html.TextBox("StudentName", null, new { @class = "form-control" })
```

```
@Html.TextBox("myTextBox", "This is value", new { @class = "form-control" })
```



TextBoxFor:

TextBoxFor helper method is a strongly typed extension method. It generates a text input element for the model property specified using a lambda expression. TextBoxFor method binds a specified model object property to input text. So it automatically displays a value of the model property in a textbox and visa-versa.

TextBoxFor() method Signature:

```
MvcHtmlString TextBoxFor(Expression<Func<TModel,TValue>> expression, object  
htmlAttributes)
```

```
@model Student  
@Html.TextBoxFor(m => m.StudentName, new { @class = "form-control" })
```

Difference between TextBox and TextBoxFor:

- @Html.TextBox() is loosely typed method whereas @Html.TextBoxFor() is a strongly typed (generic) extension method.
- TextBox() requires property name as string parameter where as TextBoxFor() requires lambda expression as a parameter.
- TextBox doesn't give you compile time error if you have specified wrong property name. It will throw run time exception.
- TextBoxFor is generic method so it will give you compile time error if you have specified wrong property name or property name changes. (Provided view is not compile at run time.)

Create TextArea using HtmlHelper:

Learn how to generate TextArea control using HtmlHelper in razor view in this section.

HtmlHelper class includes two extension methods to generate a multi line <textarea> element in a razor view: TextArea() and TextAreaFor(). By default, it creates textarea with rows=2 and cols=20.

We will use the following Student model with the TextArea() and TextAreaFor() method.

```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public string Description { get; set; }
}
```

TextArea():

The Html.TextArea() method creates <textarea rows="2" cols="20" > element with specified name, value and html attributes.

```
@model Student
@Html.TextArea("Description", null, new { @class = "form-control" })
```

```
@Html.TextArea("myTextArea", "This is value", new { @class = "form-control" })
```

TextAreaFor:

TextAreaFor helper method is a strongly typed extension method. It generates a multi line <textarea> element for the property in the model object specified using a lambda expression. TextAreaFor method binds a specified model object property to textarea element. So it automatically displays a value of the model property in a textarea and visa-versa.

TextAreaFor() method Signature:

```
MvcHtmlString TextAreaFor(<Expression<Func<TModel,TValue>> expression, object htmlAttributes)
```

```
@model Student  
@Html.TextAreaFor(m => m.Description, new { @class = "form-control" })
```

Create CheckBox using HtmlHelper:

HtmlHelper class includes two extension methods to generate a <input type="checkbox"> element in razor view: CheckBox() and CheckBoxFor().

We will use following Student model with CheckBox() and CheckBoxFor() method.

```
public class Student  
{  
    public int StudentId { get; set; }  
    [Display(Name="Name")]  
    public string StudentName { get; set; }  
    public int Age { get; set; }  
    public bool isNewlyEnrolled { get; set; }  
    public string Password { get; set; }  
}
```

1-CheckBox():

The Html.CheckBox() is a loosely typed method which generates a <input type="checkbox" > with the specified name, isChecked boolean and html attributes.

CheckBox() method Signature:

```
MvcHtmlString CheckBox(string name, bool isChecked, object htmlAttributes)
```

```
@Html.CheckBox("isNewlyEnrolled", true)
```

HTML RESULT :

```
<input checked="checked"
      id="isNewlyEnrolled"
      name="isNewlyEnrolled"
      type="checkbox"
      value="true" />
```

In the above example, first parameter is "isNewlyEnrolled" property of Student model class which will be set as a name & id of textbox. The second parameter is a boolean value, which checks or unchecks the checkbox.

2-CheckBoxFor:

CheckBoxFor helper method is a strongly typed extension method. It generates <input type="checkbox"> element for the model property specified using a lambda expression. CheckBoxFor method binds a specified model object property to checkbox element. So it automatically checked or unchecked a checkbox based on the property value.

CheckBoxFor() method Signature:

```
MvcHtmlString CheckBoxFor(<Expression<Func<TModel,TValue>> expression, object
htmlAttributes)
```

```
@model Student
@Html.CheckBoxFor(m => m.isNewlyEnrolled)
```

Create RadioButton using HtmlHelper:

HtmlHelper class include two extension methods to generate a <input type="radio"> element in a razor view: RadioButton() and RadioButtonFor().

We will use the following Student model with the RadioButton() and RadioButtonFor() method.

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

RadioButton():

RadioButton() method Signature:

```
MvcHtmlString RadioButton(string name, object value, bool isChecked, object htmlAttributes)
```

Male: @Html.RadioButton("Gender", "Male")
Female: @Html.RadioButton("Gender", "Female")
Male: <input checked="checked"

```
id="Gender"
name="Gender"
type="radio"
value="Male" />
```

Female: <input id="Gender"
name="Gender"
type="radio"
value="Female" />

RadioButtonFor:

RadioButtonFor helper method is a strongly typed extension method. It generates `<input type="radio">` element for the property specified using a lambda expression. RadioButtonFor method binds a specified model object property to RadioButton control. So it automatically checked or unchecked a RadioButton based on the property value.

RadioButtonFor() method Signature:

```
MvcHtmlString RadioButtonFor(<Expression<Func<TModel,TValue>> expression,  
object value, object htmlAttributes)
```

```
@model Student
```

```
@Html.RadioButtonFor(m => m.Gender, "Male")
```

```
@Html.RadioButtonFor(m => m.Gender, "Female")
```

Create DropDownList using HtmlHelper:

HtmlHelper class includes two extension methods to generate a <select> element in a razor view: DropDownList() and DropDownListFor().

We will use the following Student model with DropDownList() and DropDownListFor() method.

```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public Gender StudentGender { get; set; }
}
public enum Gender
{
    Male,
    Female
}
```

DropDownList():

The Html.DropDownList() method generates a select element with specified name, list items and html attributes.

DropDownList() method signature:

```
MvcHtmlString Html.DropDownList(string name, IEnumerable<SelectListItem>
selectList, string optionLabel, object htmlAttributes)
```

```
@using MyMVCApp.Models
@model Student
@Html.DropDownList("StudentGender",
    new SelectList(Enum.GetValues(typeof(Gender))),
    "Select Gender",
    new { @class = "form-control" })
```

In the above example, the first parameter is a property name for which we want to display list items. The second parameter is list of values to be included in the dropdownlist. We have used Enum methods to get the Gender enum values. The third parameter is a label which will be the first list

item and the fourth parameter is for html attributes like css to be applied on the dropdownlist.

Please note that you can add `MyMVCAApp.Models` namespace into `<namespaces>` section in web.config in the Views folder instead of using `@using` to include namespaces in all the views.

DropDownListFor:

DropDownListFor helper method is a strongly typed extension method. It generates `<select>` element for the property specified using a lambda expression. DropDownListFor method binds a specified model object property to dropdownlist control. So it automatically list items in DropDownList based on the property value.

DropDownListFor() method signature:

```
MvcHtmlString Html.DropDownListFor(Expression<Func<dynamic,TProperty>> expression, IEnumerable<SelectListItem> selectList, string optionLabel, object htmlAttributes)
```

```
@using MyMVCAApp.Models
@model Student
@Html.DropDownListFor(m => m.StudentGender,
    new SelectList(Enum.GetValues(typeof(Gender))),
    "Select Gender")
```

In the above example, the first parameter in DropDownListFor() method is a lambda expression that specifies the model property to be bind with the select element. We have specified StudentGender property of enum type. The second parameter specifies the items to show into dropdown list using SelectList. The third parameter is optionLabel which will be the first item of dropdownlist. So now, it generates `<select>` element with id & name set to property name - StudentGener and two list items - Male & Female as shown below.

Gender:

Male ▼
Select Gender
Male
Female

Create Hidden field using HtmlHelper:

HtmlHelper class includes two extension methods to generate a hidden field (<input type="hidden">) element in a razor view: Hidden() and HiddenFor().

We will use the following Student model with Hidden() and HiddenFor() method.

```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public int Age { get; set; }
    public bool isNewlyEnrolled { get; set; }
    public string Password { get; set; }
}
```

Hidden():

The Html.Hidden() method generates a input hidden field element with specified name, value and html attributes.

Hidden() method signature:

```
MvcHtmlString Html.Hidden(string name, object value, object htmlAttributes)
```

The following example creates a hidden field for StudentId property of Student model. It binds StudentId with the hidden field, so that it can assign value of StudentId to the hidden field and visa-versa.

```
@model Student
@Html.Hidden("StudentId")
```

HiddenFor:

HiddenFor helper method is a strongly typed extension method. It generates a hidden input element for the model property specified using a lambda expression. HiddenFor method binds a specified model object property to

<input type="hidden">. So it automatically sets a value of the model property to hidden field and visa-versa.

HiddenFor() method signature:

```
MvcHtmlString      Html.HiddenFor(Expression<Func<dynamic,TProperty>>  
expression)
```

```
@model Student  
@Html.HiddenFor(m => m.StudentId)
```

Create Password field using HtmlHelper:

HtmlHelper class includes two extension methods to generate a password field (<input type="password">) element in a razor view: Password() and PasswordFor().

We will use following Student model with Password() and PasswordFor() method.

```
public class Student  
{  
    public int StudentId { get; set; }  
    [Display(Name="Name")]  
    public string StudentName { get; set; }  
    public int Age { get; set; }  
    public bool isNewlyEnrolled { get; set; }  
    public string OnlinePassword { get; set; }  
}
```

Password():

The Html.Password() method generates an input password element with specified name, value and html attributes.

```
@model Student  
@Html.Password("OnlinePassword")
```

HTML Result:

```
<input  
    id="OnlinePassword"
```



```
name="OnlinePassword"
type="password"
value="" />
```

Password:

PasswordFor():

PasswordFor helper method is a strongly typed extension method. It generates a `<input type="password">` element for the model object property specified using a lambda expression. PasswordFor method binds a specified model object property to `<input type="password">`. So it automatically sets a value of the model property to password field and visa-versa.

PasswordFor() method signature:

```
MvcHtmlString PasswordFor(Expression<Func<dynamic, TProperty>>
expression, object htmlAttributes)
```

```
@model Student
@Html.PasswordFor(m => m.Password)
```

```
<input id="Password" name="Password" type="password" value="mypasswor
d" />
```

Create Html String using HtmlHelper:

Learn how to create html string literal using HtmlHelper in razor view in this section.

HtmlHelper class includes two extension methods to generate html string : Display() and DisplayFor().

We will use the following Student model with the Display() and DisplayFor() method.

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

Display():

The Html.Display() is a loosely typed method which generates a string in razor view for the specified property of model.

Display() method Signature: `MvcHtmlString Display(string expression)`

Display() method has many overloads. Please visit MSDN to know all the [overloads of Display\(\) method](#)

```
@Html.Display("StudentName")
```

DisplayFor:

```
@model Student
@Html.DisplayFor(m => m.StudentName)
```

Create Label using HtmlHelper:

HtmlHelper class includes two extension methods to generate html label : Label() and LabelFor().

We will use following Student model with to demo Label() and LabelFor() method.

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

Label():

The Html.Label() method generates a <label> element for a specified property of model object.

Label() method Signature: *MvcHtmlString Label(string expression, string labelText, object htmlAttributes)*

```
@Html.Label("StudentName")
```

```
@Html.Label("StudentName", "Student-Name")
```

LabelFor:

LabelFor helper method is a strongly typed extension method. It generates a html label element for the model object property specified using a lambda expression.

LabelFor() method Signature: *MvcHtmlString*
LabelFor(<Expression<Func<TModel,TValue>> expression)

```
@model Student
@Html.LabelFor(m => m.StudentName)
```

HtmlHelper.Editor:

We have seen different HtmlHelper methods used to generate different html elements in the previous sections. ASP.NET MVC also includes a method that generates html input elements based on the datatype. Editor() or EditorFor() extension method generates html elements based on the data type of the model object's property.

Property DataType	Html Element
string	<input type="text" >
int	<input type="number" >
decimal, float	<input type="text" >
boolean	<input type="checkbox" >
Enum	<input type="text" >
DateTime	<input type="datetime" >

```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public int Age { get; set; }
    public bool isNewlyEnrolled { get; set; }
    public string Password { get; set; }
    public DateTime DoB { get; set; }
}
```

Editor():

Editor() method requires a string expression parameter to specify the property name. It creates a html element based on the datatype of the specified property.

```
StudentId: @Html.Editor("StudentId")
Student Name: @Html.Editor("StudentName")
Age: @Html.Editor("Age")
Password: @Html.Editor("Password")
isNewlyEnrolled: @Html.Editor("isNewlyEnrolled")
Gender: @Html.Editor("Gender")
DoB: @Html.Editor("DoB")
```

StudentId:	<input type="text" value="1"/>
Student Name:	<input type="text" value="Jogn"/>
Age:	<input type="text" value="19"/>
Password:	<input type="text" value="sdf"/>
isNewlyEnrolled:	<input checked="" type="checkbox"/>
Gender:	<input type="text" value="Boy"/>
DoB:	<input type="text" value="02-06-2015 11:39:15"/>

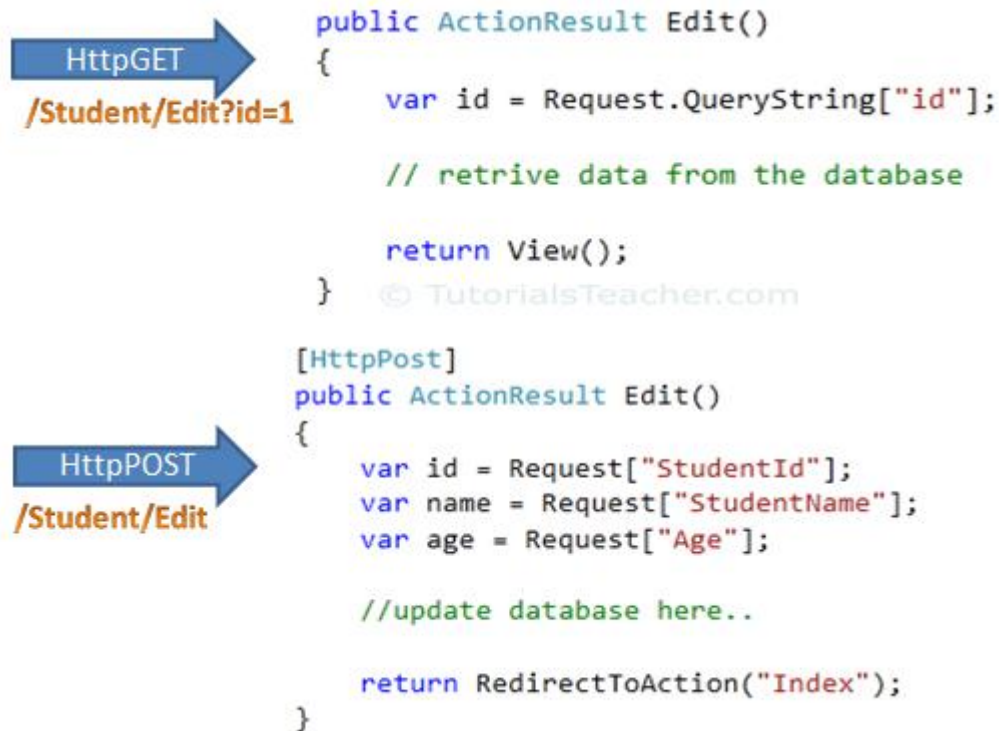
EditorFor:

```
StudentId: @Html.EditorFor(m => m.StudentId)
Student Name: @Html.EditorFor(m => m.StudentName)
Age: @Html.EditorFor(m => m.Age)
Password: @Html.EditorFor(m => m.Password)
isNewlyEnrolled: @Html.EditorFor(m => m.isNewlyEnrolled)
Gender: @Html.EditorFor(m => m.Gender)
DoB: @Html.EditorFor(m => m.DoB)
```

StudentId:	<input type="text" value="1"/>
Student Name:	<input type="text" value="Jogn"/>
Age:	<input type="text" value="19"/>
Password:	<input type="text" value="sdf"/>
isNewlyEnrolled:	<input checked="" type="checkbox"/>
Gender:	<input type="text" value="Boy"/>
DoB:	<input type="text" value="02-06-2015 11:39:15"/>

Model Binding:

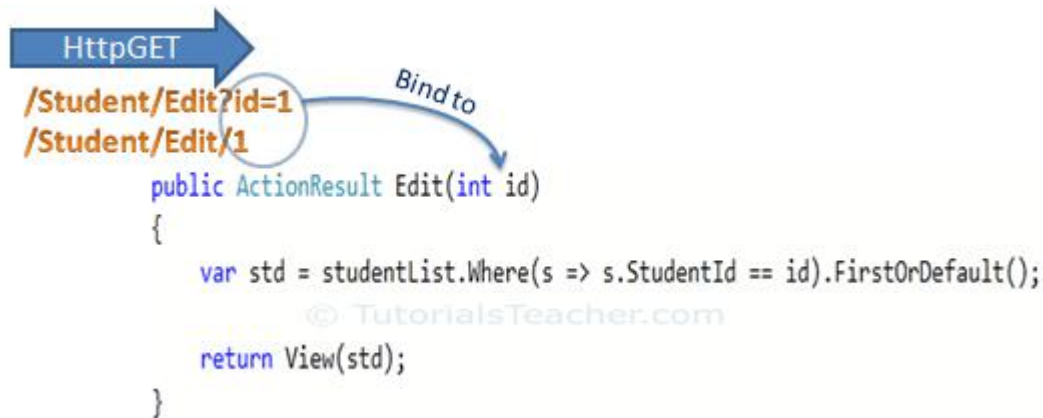
To understand the model binding in MVC, first let's see how you can get the http request values in the action method using traditional ASP.NET style. The following figure shows how you can get the values from HttpGET and HttpPOST request by using the Request object directly in the action method.



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.

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.



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

```
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 HTTP POST request to the properties of a complex type parameter of an action method.

```
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.

```
[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");
}
```



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.



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.

Example1:

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

    //write code to update student

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

Example 2:

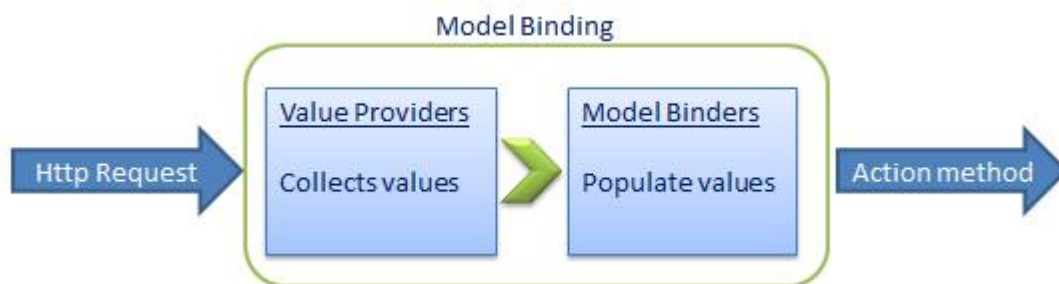
```
[HttpPost]public ActionResult Edit([Bind(Exclude = "Age")] Student std)
{
    var name = std.StudentName;

    //write code to update student

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

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

Value providers are responsible for collecting values from request and Model Binders are responsible for populating values.



Default value provider collection evaluates values from the following sources:

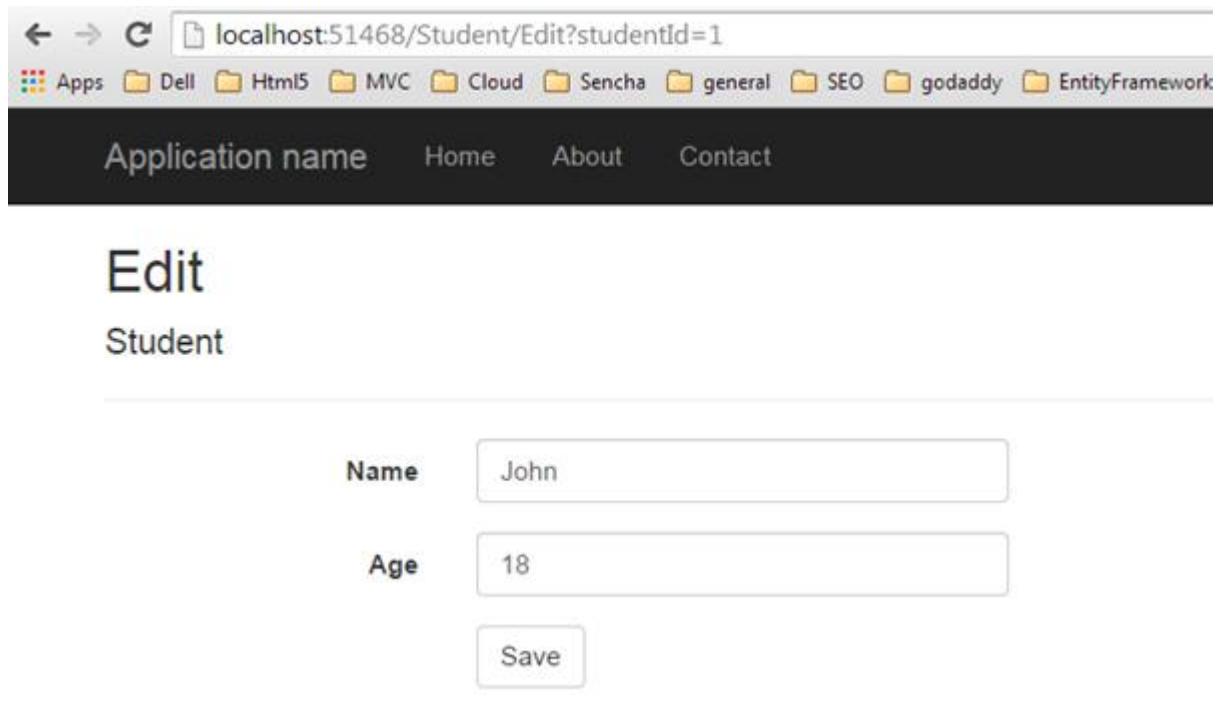
1. Previously bound action parameters, when the action is a child action
2. Form fields (Request.Form)

3. The property values in the JSON Request body (Request.InputStream), but only when the request is an AJAX request
4. Route data (RouteData.Values)
5. Querystring parameters (Request.QueryString)
6. Posted files (Request.Files)

MVC includes [DefaultModelBinder](#) class which effectively binds most of the model types.

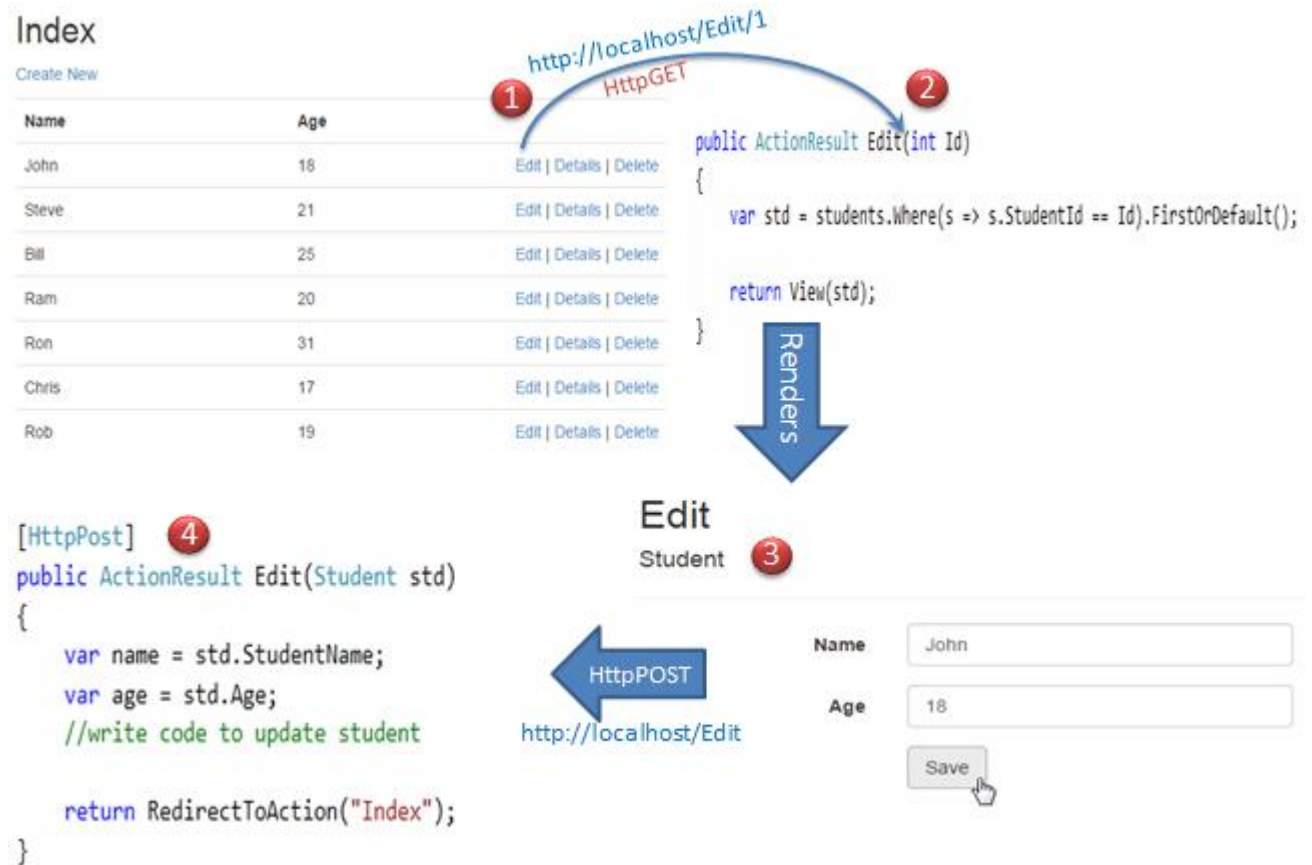
Create Edit View:

We have already created the Index view in the previous section. In this section, we will create the Edit view using a default scaffolding template as shown below. The user can update existing student data using the Edit view.



The screenshot shows a web browser window with the address bar displaying `localhost:51468/Student/Edit?studentId=1`. The browser's file explorer shows a directory structure including `Apps`, `Dell`, `Html5`, `MVC`, `Cloud`, `Sencha`, `general`, `SEO`, `godaddy`, and `EntityFramework`. The web application has a dark navigation bar with the text "Application name" and links for "Home", "About", and "Contact". The main content area is titled "Edit Student" and contains a form with two input fields: "Name" with the value "John" and "Age" with the value "18". Below these fields is a "Save" button.

The Edit view will be rendered on the click of the Edit button in Index view. The following figure describes the complete set of editing steps.



The above figure illustrates the following steps.

1. The user clicks on the Edit link in Index view which will send HttpGET request `http://localhost/student/edit/{Id}` with corresponding Id parameter in the query string. This request will be handled by HttpGET Edit action method.(by default action method handles HttpGET request if no attribute specified)
2. HttpGet Edit action method will fetch student data from the database, based on the supplied Id parameter and render the Edit view with that particular Student data.
3. The user can edit the data and click on the Save button in the Edit view. The Save button will send a HttpPOST request `http://localhost/Student/Edit` with the Form data collection.
4. The HttpPOST Edit action method in StudentController will finally update the data into the database and render an Index page with the refreshed data using the RedirectToAction method as a fourth step.

So this will be the complete process in order to edit the data using Edit view in ASP.NET MVC.

So let's start to implement above steps.

We will be using following Student model class for our Edit view.

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

    [Display( Name="Name")]
    public string StudentName { get; set; }

    public int Age { get; set; }
}
```

Step1:

We have already created an Index view in the [previous section](#) using a List scaffolding template which includes an Edit action link as shown below.

Application name Home About Contact		
Index		
Create New		
Name	Age	
John	18	Edit Details Delete
Steve	21	Edit Details Delete
Bill	25	Edit Details Delete
Ram	20	Edit Details Delete
Ron	31	Edit Details Delete
Chris	17	Edit Details Delete
Rob	19	Edit Details Delete
© 2014 - My ASP.NET Application		

An Edit link sends HttpGet request to the Edit action method of StudentController with corresponding StudentId in the query string. For example, an Edit link with student John will append a StudentId=1 query string to the request url because John's StudentId is 1. Likewise all the Edit link will include a respective StudentId in the query string.

Step 2:

Now, create a HttpGET Edit action method in StudentController. The Index view shown above will send the StudentId parameter to the HttpGet Edit action method on the click of the Edit link.

The HttpGet Edit() action method must perform two tasks, first it should fetch the student information from the underlying data source, whose StudentId matches with the StudentId in the query string. Second, it should render Edit view with the student information so that the user can update it.

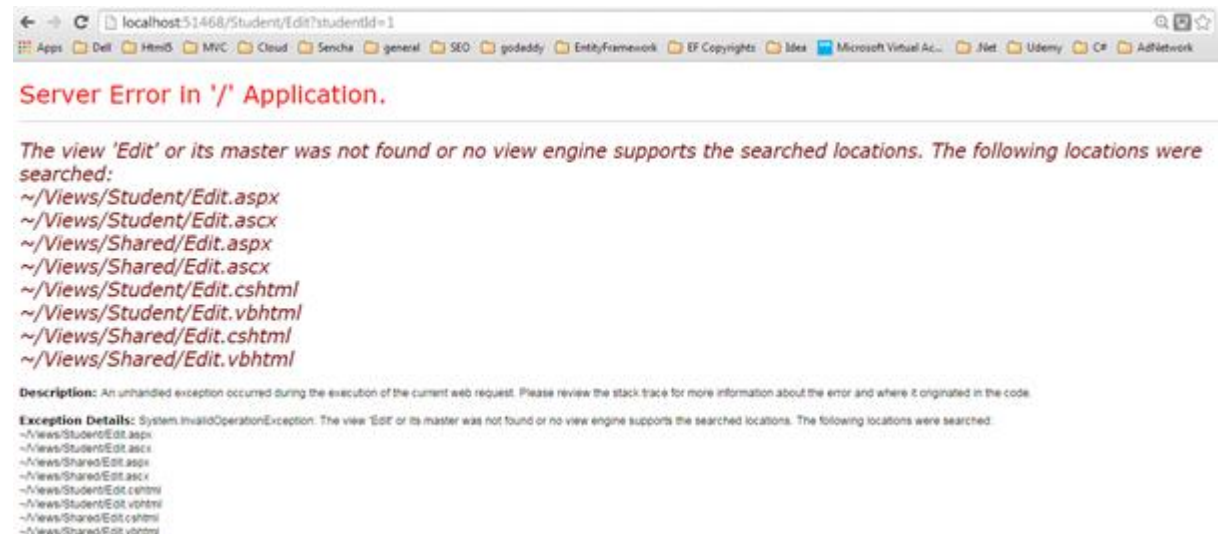
So, the Edit() action method should have a StudentId parameter. MVC framework will automatically bind a query string to the parameters of an action method if the name matches. Please make sure that parameter name matches with the query string.

```
using MVC_BasicTutorials.Models;
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        IList<Student> 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=6, StudentName="Chris", Age = 17 },
            new Student()
            { StudentId=7, StudentName="Rob", Age = 19 }
        };

        public ActionResult Edit(int StudentId)
        {
            //Get the student from studentList sample collection for demo purpose.
            //You can get the student from the database in the real application
            var std = studentList.Where(s => s.StudentId == StudentId).
                FirstOrDefault();

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

Now, if you click on the Edit link from Index view then you will get following error.

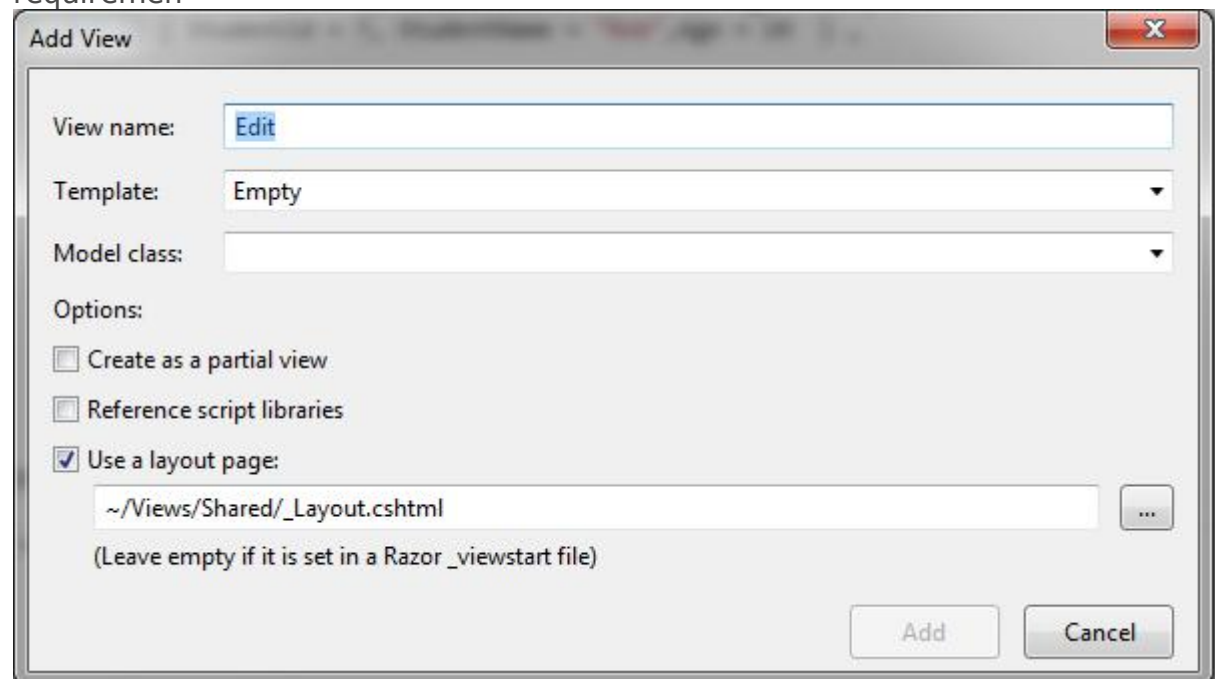


The above error occurred because we have not created an Edit view yet. By default, MVC framework will look for Edit.cshtml or Edit.vbhtml or Edit.aspx or Edit.ascx file in View -> Student or Shared folder.

Step 3:

To create Edit view, right click inside Edit action method and click on **Add View..** It will open Add View dialogue.

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



t.)

Add View

View name:

Template:

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

...

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

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

```
@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 { html
Attributes = 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 { htmlAttribut
es = 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-defau
lt" />
            </div>
        </div>
    </div>
}
<div>
    @Html.ActionLink("Back to List", "Index")</div>

```

Please notice that Edit.cshtml includes HtmlHelper method `@using (Html.BeginForm())` to create a html form element. `Html.BeginForm` sends a `HttpPost` request by default.

Now, click on the Edit link of any student in the Index view. Edit view will be display student information whose Edit link clicked, as shown below.

The screenshot shows a web browser window with the address bar displaying 'localhost:51468/Student/Edit?studentId=1'. The browser's tab bar shows several tabs: 'Apps', 'Dell', 'Html5', 'MVC', 'Cloud', 'Sencha', 'general', 'SEO', 'godaddy', and 'EntityFramework'. The page content includes a dark navigation bar with links for 'Application name', 'Home', 'About', and 'Contact'. Below this, the page is titled 'Edit Student'. The form contains two input fields: 'Name' with the value 'John' and 'Age' with the value '18'. A 'Save' button is located below the 'Age' field.

You can edit the Name or Age of Student and click on Save. Save method should send a HttpPOST request because the POST request sends form data as a part of the request, not in the querystring. So write a POST method as fourth step.

Step 4:

Now, write POST Edit action method to save the edited student as shown below.

```
[HttpPost]public ActionResult Edit(Student std)
{
    //write code to update student

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

```
using MVC_BasicTutorials.Models;
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        IList<Student> 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=6, StudentName="Chris", Age = 17 },
            new Student()
            { StudentId=7, StudentName="Rob", Age = 19 }
        };

        // GET: Student
        public ActionResult Index()
        {
            return View(studentList);
        }

        public ActionResult Edit(int StudentId)
        {

```

```
        //Get the student from studentList sample collection for d
emo purpose.
        //You can get the student from the database in the real ap
plication
        var std = studentList.Where(s => s.StudentId == StudentId).
FirstOrDefault();

        return View(std);
    }

    [HttpPost]
    public ActionResult Edit(Student std)
    {
        //write code to update student

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

Implement Data Validation in MVC:

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.

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

Edit

Student

Name

The Name field is required.

Age

The Age field is required.

Save

[Back to List](#)

© 2014 - My ASP.NET Application

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

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.

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

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

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

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.

```
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);
        }
    }
}
```

-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..**

Add View

View name:

Template:

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

...

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

Add View

View name:

Template:

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☒ Use a layout page:

...

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

```

@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 { html
Attributes = 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 { htmlAttribut
es = 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-defau
lt" />
        </div>
    </div>
</div>
}
<div>
    @Html.ActionLink("Back to List", "Index")</div>

```

Points to Remember :

1. ASP.NET MVC uses DataAnnotations attributes for validation.
2. DataAnnotations attributes can be applied to the properties of the model class to indicate the kind of value the property will hold.
3. The following validation attributes available by default
 1. Required
 2. StringLength
 3. Range
 4. RegularExpression
 5. CreditCard
 6. CustomValidation
 7. EmailAddress
 8. FileExtension

- 9. MaxLength
- 10. MinLength
- 11. Phone
- 4. Use **ValidationSummary** to display all the error messages in the view.
- 5. Use **ValidationMessageFor** or **ValidationMessage** helper method to display field level error messages in the view.
- 6. Check whether the model is valid before updating in the action method using ModelState.IsValid.
- 7. Enable client side validation to display error messages without postback effect in the browser.

ValidationMessage:

You have learned how to implement validation in a view in the previous section. Here, we will see the HtmlHelper extension method ValidationMessage in detail.

The Html.ValidationMessage() is an extension method, that is a loosely typed method. It displays a validation message if an error exists for the specified field in the ModelStateDictionary object.

ValidationMessage() Signature:

```
MvcHtmlString ValidateMessage(string modelName, string validationMessage, object htmlAttributes)
```

```
@model Student
@Html.Editor("StudentName") <br />
@Html.ValidationMessage("StudentName",
    "",
    new { @class = "text-danger" })
```

-the first parameter in the ValidationMessage method is a property name for which we want to show the error message e.g. StudentName.

- The second parameter is for custom error message

-the third parameter is for html attributes like css, style etc.

-The ValidationMessage() method will only display an error, if you have configured the DataAnnotations attribute to the specified property in the model class.

```
public class Student
{
    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

The above code will generate following html.

```
<input id="StudentName"
      name="StudentName"
      type="text"
      value="" />

<span class="field-validation-valid text-danger"
      data-valmsg-for="StudentName"
      data-valmsg-replace="true">
</span>
```

Custom Error Message:

You can display your own error message instead of the default error message as shown above. You can provide a custom error message either in the DataAnnotations attribute or ValidationMessage() method.

```
public class Student
{
    public int StudentId { get; set; }
    [Required(ErrorMessage="Please enter student name.")]
    public string StudentName { get; set; }
    public int Age { get; set; }
}

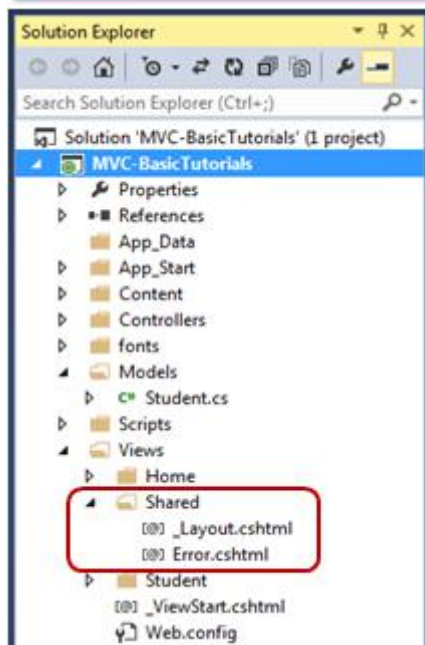
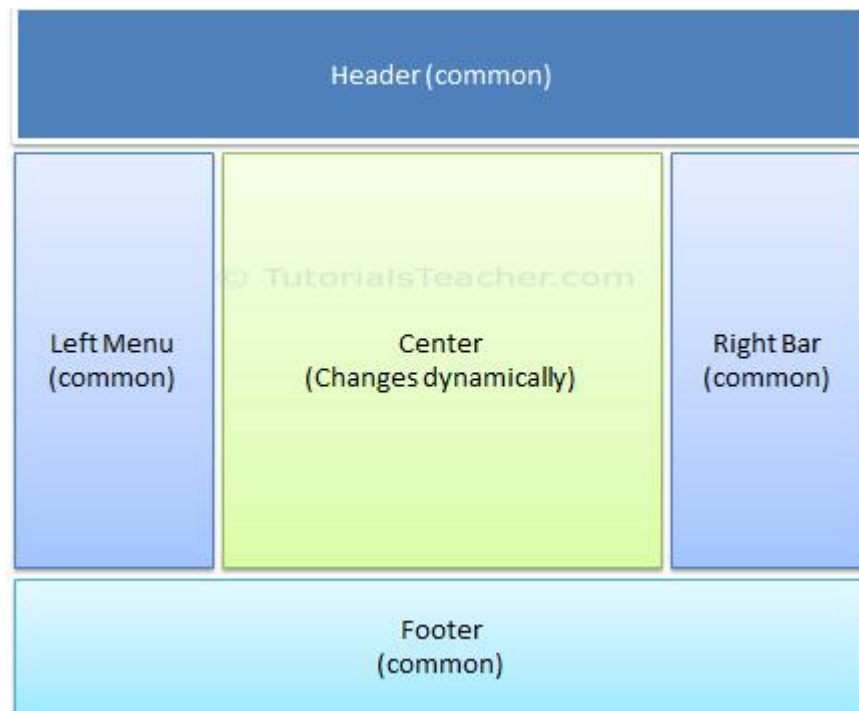
@model Student
@Html.Editor("StudentName") <br />
@Html.ValidationMessage("StudentName",
    "Please enter student name.",
    new { @class = "text-danger" })
```

ValidationMessageFor:

```
@model Student
@Html.Editor("StudentName") <br />@Html.ValidationMessageFor(m =>
    m.StudentName, "Please enter student name.", new { @class = "text-da
nger" })
```

Layout View:

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.



```

<!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-toggl
e="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")</l
i>
                    <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>

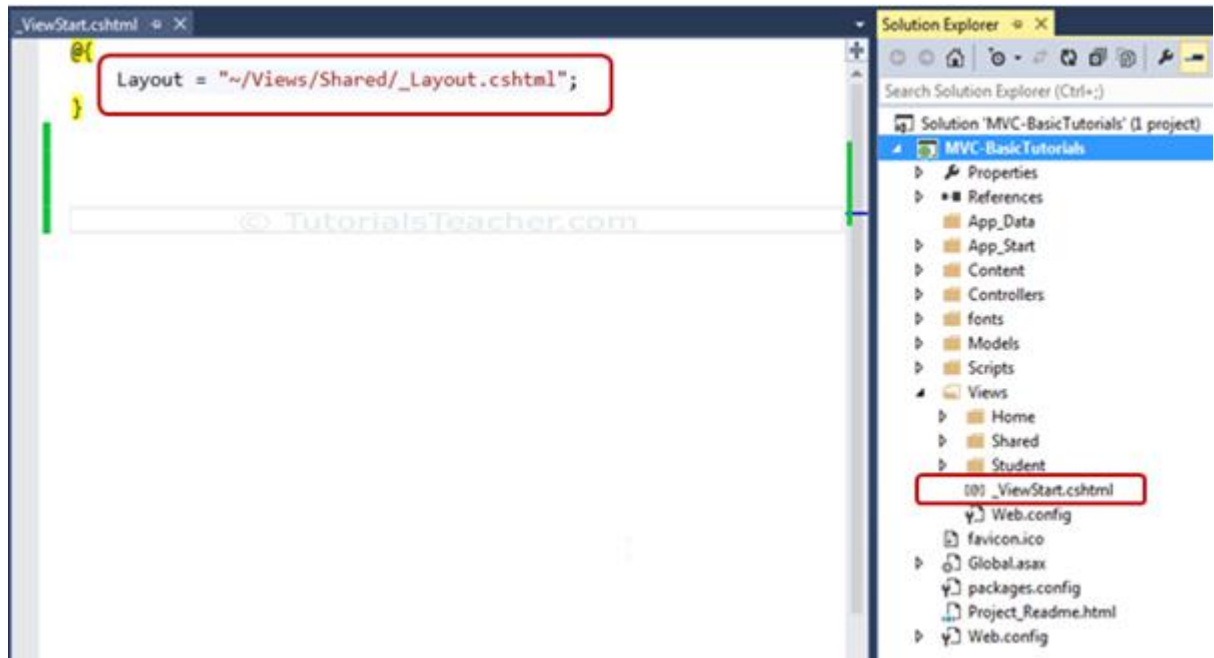
```

Use Layout View:

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.



Setting Layout property in individual view:

You can also override default layout page set by `_ViewStart.cshtml` by setting Layout property in each individual `.cshtml` view. For example, the following Index view use `_myLayoutPage.cshtml` even if `_ViewStart.cshtml` set `_Layout.cshtml`.

```
@{  
    ViewBag.Title = "Home Page";  
    Layout = "~/Views/Shared/_myLayoutPage.cshtml";  
}
```

Specify Layout page in ActionResult method:

You can also specify which layout page to use in while rendering view from action method using `View()` method.

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View("Index", "_myLayoutPage");
    }

    public ActionResult About()
    {
        return View();
    }

    public ActionResult Contact()
    {
        return View();
    }
}

```

Points to Remember :

1. The Layout view contains common parts of a UI. It is same like masterpage of ASP.NET webforms.
2. _ViewStart.cshtml file can be used to specify path of layout page, which in turn will be applicable to all the views of the folder and its subfolder.
3. You can set the Layout property in the individual view also, to override default layout page setting of _ViewStart.cshtml
4. Layout view uses two rendering methods: RenderBody() and RenderSection().
5. RenderBody can be used only once in the layout view, whereas the RenderSection method can be called multiple time with different name.
6. RenderBody method renders all the content of view which is not wrapped in named section.
7. RenderSection method renders the content of a view which is wrapped in named section.
8. RenderSection can be configured as required or optional. If required, then all the child views must included that named section.

ViewBag in ASP.NET MVC:

We have learned in the previous section that the model object is used to send data in a razor view. However, there may be some scenario where you want to send a small amount of temporary data to the view. So for this reason, MVC framework includes ViewBag.

ViewBag can be useful when you want to transfer temporary data (which is not included in model) from the controller to the view. The ViewBag is a **dynamic** type property of ControllerBase class which is the base class of all the controllers.



In the above figure, it attaches Name property to ViewBag with the dot notation and assigns a string value "Bill" to it in the controller. This can be accessed in the view like @ViewBag.Name. (@ is razor syntax to access the server side variable.)

```
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        IList<Student> studentList = new List<Student>() {
            new Student(){ StudentID=1, StudentName="Steve", Age = 21 },
            new Student(){ StudentID=2, StudentName="Bill", Age = 25 },
            new Student(){ StudentID=3, StudentName="Ram", Age = 20 },
            new Student(){ StudentID=4, StudentName="Ron", Age = 31 },
            new Student(){ StudentID=5, StudentName="Rob", Age = 19 }
        };
        // GET: Student
        public ActionResult Index()
        {
            ViewBag.TotalStudents = studentList.Count();

            return View();
        }
    }
}
```



```
}  
    }
```

ViewData:

ViewData is similar to ViewBag. It is useful in transferring data from Controller to View.

ViewData is a dictionary which can contain key-value pairs where each key must be string.

The following figure illustrates the ViewData.



```
public ActionResult Index()
{
    IList<Student> studentList = new List<Student>();
    studentList.Add(new Student(){ StudentName = "Bill" });
    studentList.Add(new Student(){ StudentName = "Steve" });
    studentList.Add(new Student(){ StudentName = "Ram" });

    ViewData["students"] = studentList;

    return View();
}
```

```
<ul>@foreach (var std in ViewData["students"] as IList<Student>)
{
    <li>
        @std.StudentName
    </li>
}</ul>
```

Please notice that we must cast ViewData values to the appropriate data type.

You can also add a KeyValuePair into ViewData as shown below.

```
public ActionResult Index()
{
    ViewData.Add("Id", 1);
    ViewData.Add(new KeyValuePair<string, object>("Name", "Bill"));
    ViewData.Add(new KeyValuePair<string, object>("Age", 20));

    return View();
}
```

TempData:

TempData in ASP.NET MVC can be used to store temporary data which can be used in the subsequent request. TempData will be cleared out after the completion of a subsequent request.

TempData is useful when you want to transfer non-sensitive data from one action method to another action method of the same or a different controller as well as redirects. It is dictionary type which is derived from [TempDataDictionary](#).

```
public class HomeController : Controller
{
    // GET: Student
    public HomeController()
    {
    }

    public ActionResult Index()
    {
        TempData["name"] = "Test data";
        TempData["age"] = 30;

        return View();
    }

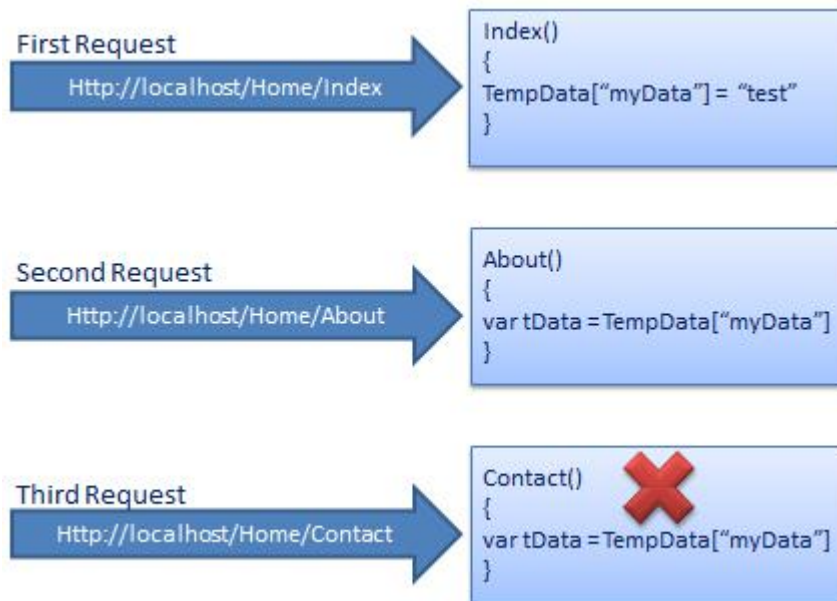
    public ActionResult About()
    {
        string userName;
        int userAge;

        if(TempData.ContainsKey("name"))
            userName = TempData["name"].ToString();

        if(TempData.ContainsKey("age"))
            userAge = int.Parse(TempData["age"].ToString());

        // do something with userName or userAge here

        return View();
    }
}
```



You can't get the same data in the third request because TempData will be cleared out after second request.

Call `TempData.Keep()` to retain TempData values in a third consecutive request.

```
public class HomeController : Controller
```

```
{
```

```
    // GET: Student
```

```
    public HomeController()
```

```
    {
```

```
    }
```

```
    public ActionResult Index()
```

```
    {
```

```
        TempData["myData"] = "Test data";
```

```
        return View();
```

```
    }
```

```
    public ActionResult About()
```

```
{  
  
    string data;  
  
    if(TempData["myData"] != null)  
        data = TempData["myData"] as string;  
  
    TempData.Keep();  
  
    return View();  
}  
  
public ActionResult Contact()  
{  
    string data;  
  
    if(TempData["myData"] != null)  
        data = TempData["myData"] as string;  
  
    return View();  
}  
}
```

Filters in MVC:

In ASP.NET MVC, a user request is routed to the appropriate controller and action method. However, there may be circumstances where you want to execute some logic before or after an action method executes. ASP.NET MVC provides filters for this purpose.

MVC provides different types of filters. The following table lists filter types, built-in filters for the type and interface which must be implemented to create a custom filter class.

<i>Filter Type</i>	<i>Description</i>	<i>Built-in Filter</i>	<i>Interface</i>
<i>Authorization filters</i>	<i>Performs authentication and authorizes before executing action method.</i>	<i>[Authorize], [RequireHttps]</i>	<i>IAuthorizationFilter</i>
<i>Action filters</i>	<i>Performs some operation before and after an action method executes.</i>		<i>IActionFilter</i>
<i>Result filters</i>	<i>Performs some operation before or after the execution of view result.</i>	<i>[OutputCache]</i>	<i>IResultFilter</i>
<i>Exception filters</i>	<i>Performs some operation if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline.</i>	<i>[HandleError]</i>	<i>IExceptionFilter</i>

An exception filter executes when there is an unhandled exception occurs in your application. `HandleErrorAttribute` (`[HandlerError]`) class is a built-in exception filter class in MVC framework. This built-in `HandleErrorAttribute`

class renders Error.cshtml included in the Shared folder by default, when an unhandled exception occurs.

