# Razor

**Notes:**

- ASP.NET supports the following types of razor view files:

.cshtml: C# Razor view. Supports C# code with HTML tags.

.vbhtml: Visual Basic Razor view. Supports Visual Basic code with HTML tags.

.aspx: ASP.Net web form

.ascx: ASP.NET web control

- Razor is the new technology. The traditional old way is ASP (with syntax: <% %>).

## Razor Markup

Razor is a view engine for server side markup language. It lets you embed server-based code (C# or Visual Basic) into HTML.

**Syntax**:

* C# code blocks are enclosed in @{ ... }
* Inline expressions (variables or functions) start with @
* Statements start with @

**Example 1**:

<!-- Single statement block -->

@{ var myMessage = "Hello World"; }

<!-- Inline expression or variable -->

<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->

@{

var greeting = "Welcome to our site!";

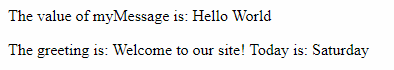
var weekDay = DateTime.Now.DayOfWeek;

var greetingMessage = greeting + " Today is: " + weekDay;

}

<p>The greeting is: @greetingMessage</p>

Output:



**Example 2:**

<!-- Statement if...else...  -->

@{var price=20;}

<html>

<body>

@if (price>30) {

  <p>The price is too high.</p>

}

else {

  <p>The price is OK.</p>

}

</body>

</html>

<!-- Statement switch...case...  -->

@{

var weekday = DateTime.Now.DayOfWeek;

var day = weekday.ToString();

var message = "";

}

<html>

<body>

@switch(day)

{

case "Monday":

    message = "This is the first weekday.";

    break;

case "Thursday":

    message = "Only one day before weekend.";

    break;

case "Friday":

    message = "Tomorrow is weekend!";

    break;

default:

    message = "Today is " + day;

    break;

}

<p>@message</p>

</body>

</html>

<!-- Statement for  -->

<html>

<body>

@for(var i = 10; i < 21; i++) {

<p>Line @i</p>

}

</body>

</html>

<!-- Statement foreach  -->

<html>

<body>

<ul>

@foreach (var x in Request.ServerVariables) {

<li>@x</li>

}

</ul>

</body>

</html>

# Web Pages

## Folder Structure

|  |  |
| --- | --- |
| Below is a typical folder structure for an ASP.NET **Web Forms** project: | Below is a typical folder structure for an ASP.NET **MVC project**:  ASP.NET MVC Folder Structure |

Where:

* App\_Data: contains databases and data files (LocalDB, .mdf files, XML files, and other data related files)
* App\_Start: contains class files which will be executed when the application starts. Typically, these would be config files like BundleConfig.cs, RouteConfig.cs, FilterConfig.cs, AuthConfig.cs, etc.
* Content: contains static files (CSS files, images, and icons files).
* fonts: contains all fonts used in the website
* Scripts: contains browser scripts (JavaScript for bootstrap, jquery 1.10, and modernizeror. Or VBScript)
* Controllers: contains class files for the controllers. MVC requires the name of all controller files to end with "Controller".
* Models: contains model class files, typically including public properties.
* Views: contains HTML files for the views. Typically, view file is a .cshtml file where you write HTML and C# or VB.NET code.

The Views folder includes a separate folder for each controller.

* + Home: contains HTML files which are rendered by HomeController
  + Shared: contains common files (like layout and style files)
  + Etc.
* Images: contains images
* Account: contains logon and security files

Additionally, MVC project includes the following configuration files:

* Packages.config: is managed by NuGet to track what packages and versions you have installed in the application.
* Web.config: contains application-level configurations.
* Global.asax: containts application-level events, such as Application\_BeginRequest, Application\_Start, Application\_Error, Session\_Start, Session\_End, etc.
* About.asax
* Contact.asax

## Page Layout

Standard websites must have a **consistent look** and feel:

* Every page have the same header
* Every page have the same footer
* Every page have the same style and layout

To do that, we have two ways:

1. Have reusable content blocks, like headers and footers, in **separate files** which can be imported to any page in the website.
2. Define a consistent layout for all pages, using a **layout template**.

### Content Blocks

With Web Pages, you can use the @RenderPage() method to import content from separate files.

**Example**:

<html>

<body>

@RenderPage("header.cshtml")

<h1>Hello Web Pages</h1>

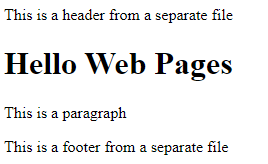
<p>This is a paragraph</p>

@RenderPage("footer.cshtml")

</body>

</html>

Output:



### Layout Template

A *layout page* **contains the structure, but NOT the content, of a web page**. When a web page (*content page*) is linked to a layout page, it will be displayed according to the layout page.

The *layout page* is just like a normal web page, except from a call to the @RenderBody() method where the content page will be included. Each *content page* must start with a Layout directive.

**Example**:

In *layout page*:

<html>

<body>

<p>This is header text</p>

@RenderBody()

<p>&copy; 2014 W3Schools. All rights reserved.</p>

</body>

</html>

In *content page*:

@{Layout = "Layout.cshtml";}

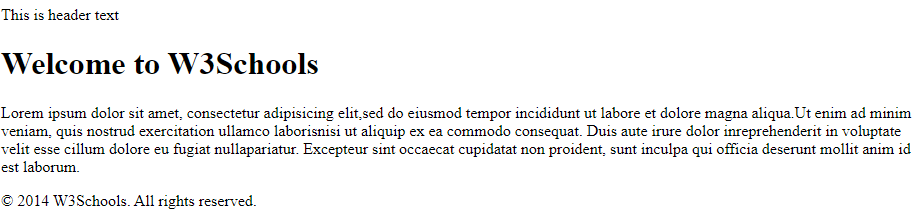
<h1>Welcome to W3Schools</h1>

<p>

Lorem ipsum dolor sit amet, consectetur adipisicing elit,sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laborisnisi ut aliquip ex ea commodo consequat.

</p>

Output:



Notes:

* Layout views are shared with multiple views, so it should be stored in the Shared folder. By default, a layout view \_Layout.cshtml is created when you create MVC application using Visual Studio.

## URLs and Paths

* *URLs* are used to access files from the web
* The URL corresponds to a *physical path* on a server
* A *virtual path* is shorthand to represent physical paths. If you use virtual paths, you can move your pages to a different domain (or server) without having to update the paths.

**For example:**

* **URL**: https://www.w3schools.com/html/html5\_intro.asp
* **Server name**: w3schools
* **Virtual path**: /html/html5\_intro.asp
* **Physical path**: C:\MyWebSites\w3schools\html\html5\_intro.asp

ASP.NET has 3 tools for working with folder paths: the ~ operator, the Server.MapPath method, and the Href method.

### The ~ Operator

It specifies virtual root for the virtual path in programming code, so you can move your website to a different folder or location without changing any code:

var myImagesFolder = "~/images";

var myStyleSheet = "~/styles/StyleSheet.css";

### The Server.MapPath Method

It converts a virtual path (/default.cshtml) to a physical path that the server can understand (C:\Johnny\MyWebSited\Demo\default.cshtml).

You use this method to open data files located on the server (data files can only be accessed with a full physical path):

var pathName = "~/dataFile.txt";

var fileName = Server.MapPath(pathName);

### The Href Method

It converts a virtual path to a path that the browser can understand (the browser cannot understand the ~ operator).

You use this method to create paths to resources like image files and CSS files. It's often used in HTML <a>, <img>, and <link> elements:

@{var myStyleSheet = "~/Shared/Site.css";}

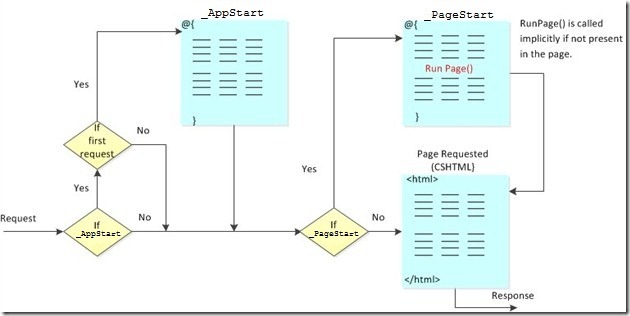
<!-- This creates a link to the CSS file. -->

<link rel="stylesheet" type="text/css" href="@Href(myStyleSheet)" />

<!-- Same as : -->

<link rel="stylesheet" type="text/css" href="/Shared/Site.css" />

## Global Pages



### \_AppStart (Before Web Startup)

If \_AppStart exits, ASP.NET runs it the first time any page in the site is requested. So you can have startup code executed before the site starts.

Typical use for \_AppStart is startup code and initialization of global values like counters and global names.

### \_PageStart (Before Every Page)

Just like \_AppStart runs before your site starts, \_PageStart runs before any page in each folder. For each folder in your web, you can add a file named \_PageStart.

Typical use for \_PageStart is setting the layout page for all pages in a folder, or checking that a user is logged in before running a page.

## Forms

A form is a section of an HTML document where you put input controls (text boxes, check boxes, radio buttons, and pull-down lists).

**Example 1:**

<html>

IsPost is true when Submit button is clicked

<body>

@{

if (IsPost) {

string companyname = Request["CompanyName"];

string contactname = Request["ContactName"];

<p>You entered: <br />

Company Name: @companyname <br />

Contact Name: @contactname </p>

}

else

{

<form method="post" action="">

Company Name:<br />

<input type="text" name="CompanyName" value="" /><br />

Contact Name:<br />

<input type="text" name="ContactName" value="" /><br /><br />

<input type="submit" value="Submit" class="submit" />

</form>

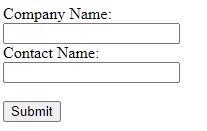
}

}

</body>

</html>

Output:



Assume you enter Company Name as "Personal", and Contact Name as "triho". Once click Submit button, you get following output:

You entered:

Company Name: Personal

Contact Name: triho

**Example 2:**

Request["Choice"] is true when Submit button is clicked

@{

var imagePath="";

if( Request["Choice"] != null)

   {imagePath="images/" + Request["Choice"];}

}

<!DOCTYPE html>

<html>

<body>

<h1>Display Images</h1>

<form method="post" action="">

I want to see:

<select name="Choice">

   <option value="Pic1.jpg">Photo 1</option>

   <option value="Pic2.jpg">Photo 2</option>

   <option value="Pic3.jpg">Photo 3</option>

</select>

<input type="submit" value="Submit">

@if(imagePath != "")

{

<p>

<img src="@imagePath" alt="Sample">

</p>

}

</form>

</body>

</html>

Output:



## Page Objects

### Page Object Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| href() | Builds a URL using the specified parameters |
| RenderPage(page) | Renders the content of one page within another page |
| RenderSection(section) | Renders the content of a named section (In layout pages) |
| RenderBody() | Renders the portion of a content page that is not within a named section (In layout pages) |
| Write(object) | Writes the object as an HTML-encoded string |
| WriteLiteral() | Writes an object without HTML-encoding it first. |

**Examples:**

* href()
* RenderPage(page): [this section](#_Content_Blocks)
* RenderSection(section)
* RenderBody(): [this section](#_Layout_Template)
* Write(object)
* WriteLiteral()

Page Object Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| IsPost | Returns true if the HTTP data transfer method used by the client is a POST request |
| Layout | Gets or sets the path of a layout page |
| Page | Provides property-like access to data shared between pages and layout pages |
| Request | Gets the HttpRequest object for the current HTTP request |
| Server | Gets the HttpServerUtility object that provides web-page processing methods |

**Examples:**

* IsPost: [this section](#_Forms)
* Layout
* Page
* Request: [this section](#_Forms)
* Server

# ASP.NET Web Forms Architecture

## What Is Web Forms?

## Routing in Web Forms

ASP.NET introduced Routing to **eliminate the needs of mapping each URL with a physical file** by allowing us to define a *URL pattern that maps to the request handler*. This request handler can be a file or class.

In the ASP.NET Web Forms application, request handler is .aspx file. So, 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.

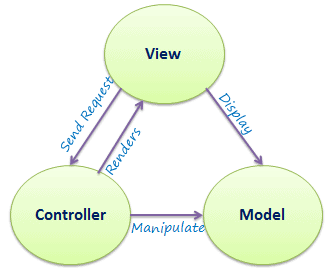
# ASP.NET MVC Architecture

## What Is MVC?

MVC stands for Model, View and Controller:

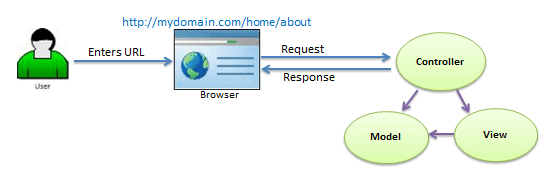
* **Model** represents the data and business logic.
* **View** is the User Interface. View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor) that makes it easy to communicate with the model and the controller.
* **Controller** is the request handler. Typically, the user uses the view and raises an HTTP request. The controller processes the request and returns the appropriate view as a response.

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

[](https://www.tutorialsteacher.com/Content/images/mvc/mvc-architecture.png)

*MVC Architecture*

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

[](https://www.tutorialsteacher.com/Content/images/mvc/request-handling-in-mvc.png)

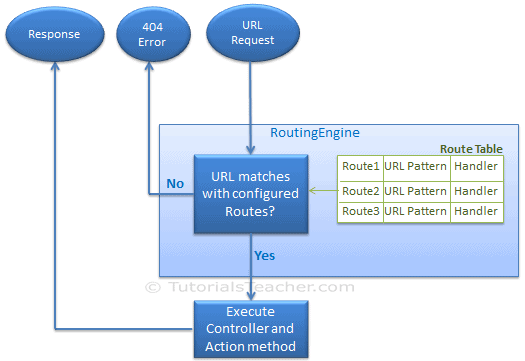
*Request Flow in MVC Architecture*

## Routing in MVC

ASP.NET introduced Routing to **eliminate the needs of mapping each URL with a physical file** by allowing us to define a *URL pattern that maps to the request handler*. This request handler can be a file or class.

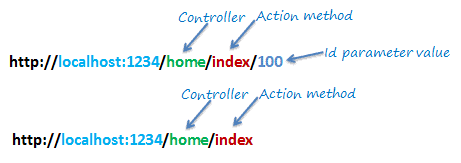
In MVC, request handler is the Controller class and Action method. For example, http://domain/students must mapped to Student Controller and Index action method in MVC.

The following figure illustrates the Routing process in MVC.

[](https://www.tutorialsteacher.com/Content/images/mvc/routing-process.png)

### URL Pattern

The URL pattern is considered only after the domain name in the URL. For example, the URL pattern "{controller}/{action}/{id}" would look like http://domain/{controller}/{action}/{id}



### Configure Routes

Routes are registered in RegisterRoutes() method of RouteConfig class, which is in RouteConfig.cs under App\_Start folder.

Then call the RegisterRoutes() method in the Application\_Start() event in the Global.asax.

For example:

public class MvcApplication : System.Web.HttpApplication

{

  protected void Application\_Start()

  {

    RouteConfig.RegisterRoutes(RouteTable.Routes);

  }

}

public class RouteConfig

{

  public static void RegisterRoutes(RouteCollection routes)

Routes to ignore

  {

    routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

Any URL starting with *domain/Home* must be handled by HomeController class

    routes.MapRoute(

      name: "Default",

      url: "{controller}/{action}/{id}",

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

    );

  }

}

The following table shows how different URLs will be mapped to the Default route:

|  |  |  |  |
| --- | --- | --- | --- |
| **URL** | **Controller** | **Action** | **ID** |
| http://domain/home | HomeController | Index | null |
| http://domain/student | StudentController | Index | null |
| http://domain/home/about | HomeController | About | null |
| http://domain/home/index/123 | HomeController | Index | 123 |
| http://domain/student/edit/123 | StudentController | Edit | 123 |

### Multiple Routes

You can register multiple custom routes with different names.

For example:

public class RouteConfig

{

  public static void RegisterRoutes(RouteCollection routes)

  {

    routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

Any URL starting with *domain/students* must be handled by StudentController class

    routes.MapRoute(

      name: "Student",

      url: "students/{id}",

      defaults: new { controller = "Student", action = "Index"}

    );

Default param is optional

    routes.MapRoute(

      name: "Default",

      url: "{controller}/{action}/{id}",

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

    );

  }

}

The following table shows how different URLs will be mapped to the Student route:

|  |  |  |  |
| --- | --- | --- | --- |
| **URL** | **Controller** | **Action** | **ID** |
| http://domain/student/123 | StudentController | Index | 123 |
| http://domain/student/index/123 | StudentController | Index | 123 |
| http://domain/student?Id=123 | StudentController | Index | 123 |

Note:

* We haven't specified {action} in the URL pattern because we want every URL that starts with students should always use the Index() action of the StudentController class.
* MVC framework **evaluates each route in sequence**. It starts with the first configured route, and if incoming URL doesn't satisfy the URL pattern of the route, then it will evaluate the second route and so on.

### Route Constraints

You can **apply** **restrictions on the value of the parameter** by configuring route constraints.

For example, the following route applies a limitation on the id parameter that the id's value must be numeric:

routes.MapRoute(

  name: "Student",

  url: "student/{id}/{name}/{standardId}",

  defaults: new { controller = "Student", action = "Index",

                  id = UrlParameter.Optional,

                  name = UrlParameter.Optional,

                  standardId = UrlParameter.Optional },

  constraints: new { id = @"\d+" }

);

So if you give non-numeric value for id parameter, then that request will be handled by another route or, if there are no matching routes, then "The resource could not be found" error will be thrown.

## Controller

### Controller Class

* In MVC, all controllers must be stored in Controllers folder.
* MVC requires the name of all controller files to end with "Controller".
* Your Controller class must be derived from System.Web.Mvc.Controller class.
* All the public methods in the Controller class are called Action methods.

For example, create a Student 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

It's called Action method

    public string Index()

    {

       return "This is Index action method of StudentController";

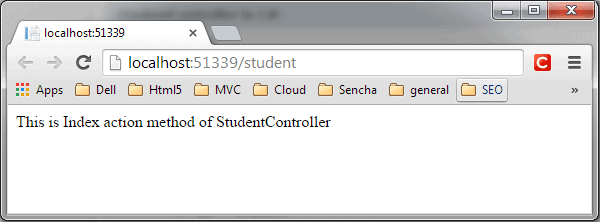
    }

  }

}

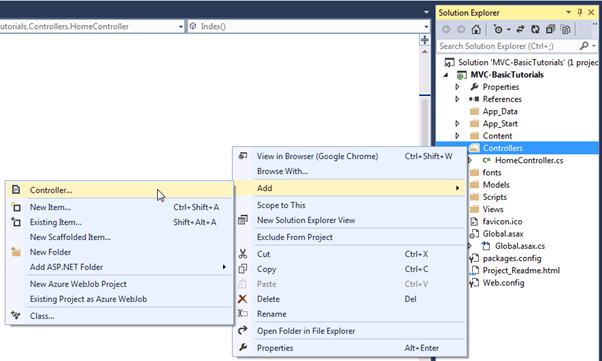
Output:

As learned in [Routing](#_Routing_in_MVC) seccion, the URL http://localhost/student or http://localhost/student/index is handled by the Index() method of the StudentController class.



**Tips:**

* Fast way to create a controller:



### Action Methods

Action methods are like any other normal methods with the following restrictions:

* Action method must be public. It cannot be private or protected
* Action method cannot be overloaded
* Action method cannot be a static method.
* Action method can return something (ActionResult) or nothing (void).
* Action methods can have parameters. It can be primitive data type or complex type.

**Example:**

public ActionResult Index()

{

  return View();

}

[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");

}

Details: <https://www.c-sharpcorner.com/article/action-result-in-asp-net-mvc/>

### Action Selectors

#### ActionName

This selector allows us to specify a **different action name than the method name**. For example:

public class StudentController : Controller

{

Now this action method will be invoked on http://localhost/student/find/1 request, NOT http://localhost/student/getbyid/1 request

  public StudentController()

  {

  }

  [ActionName("Find")]

  public ActionResult GetById(int id)

  {

    // get student from the database

    return View();

  }

}

#### NonAction

Use this selector when you want **public method** in a controller but **do not want to treat it as an action method**.

#### ActionVerbs

This selector is to handle different type of HTTP requests. The MVC framework includes:

* HttpGet (default)
* HttpPost
* HttpPut
* HttpDelete
* HttpOptions
* HttpPatch

**Example 1:**

public class StudentController : Controller

{

  public ActionResult Index() // handles GET requests by default

  {

    return View();

  }

  [HttpPost]

  public ActionResult PostAction() // handles POST requests by default

  {

    return View("Index");

  }

  [HttpPut]

  public ActionResult PutAction() // handles PUT requests by default

  {

    return View("Index");

  }

  [HttpDelete]

  public ActionResult DeleteAction() // handles DELETE requests by default

  {

    return View("Index");

  }

  [HttpHead]

  public ActionResult HeadAction() // handles HEAD requests by default

  {

    return View("Index");

  }

  [HttpOptions]

  public ActionResult OptionsAction() // handles OPTION requests by default

  {

    return View("Index");

  }

  [HttpPatch]

  public ActionResult PatchAction() // handles PATCH requests by default

  {

    return View("Index");

  }

}

**Example 2:** You can apply multiple action verbs using the AcceptVerbs attribute:

[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Get)]

public ActionResult GetAndPostAction()

{

  return RedirectToAction("Index");

}

## Model

### Model Class

* In MVC all models must be stored in Models folder.

For example, create a Student model:

public class Student

{

  public int StudentId { get; set; }

  public string StudentName { get; set;  }

  public int Age { get; set;  }

}

The model class can be used in the view to populate the data, as well as sending data to the controller.

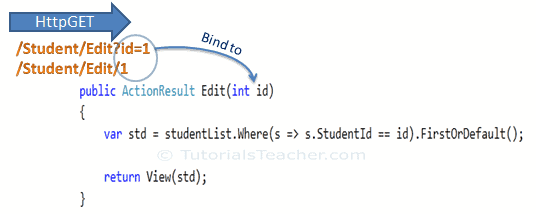
### Model Binding

The model binding refers to converting the HTTP request data (from the query string or form collection) to an action method parameters

#### Binding to Primitive Type

The HTTP GET request embeds data into a query string. MVC framework automatically converts a query string to the action method parameters provided their names are matching.

For example, the query string id in the following GET request would automatically be mapped to the Edit() action method's id parameter.

[](https://www.tutorialsteacher.com/Content/images/mvc/model-binding-1.png)

Notes:

* This binding is **case-insensitive**. So "id" parameter can be "ID" or "Id".
* You can have multiple parameters in the action method with different data types. Query string values will be converted into parameters based on the matching names.

For example, the query string parameters of the GET request http://localhost/Student/Edit?id=1&name=John would map to id and name parameters of the following Edit() action method:

public ActionResult Edit(int id, string name)

{

  // do something here

  return View();

}

#### Binding to Complex Type

With complex types, model bidning also automatically converts the input fields data on the view to the properties of a complex type parameter of an action method in HttpPost request if the properties' names match with the fields on the view.

**Example 1**: Map Form collection values to the Student type parameter when the form submits an HTTP POST request to the Edit() action method:

// In model class

public class Student

{

  public int StudentId { get; set; }

  public string StudentName { get; set; }

  public int Age { get; set; }

  public Standard standard { get; set; }

}

public class Standard

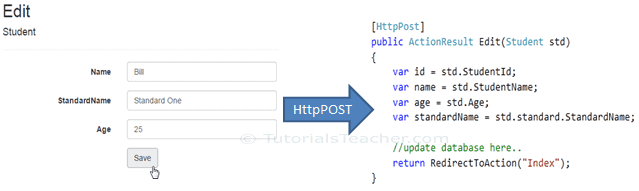
{

  public int StandardId { get; set; }

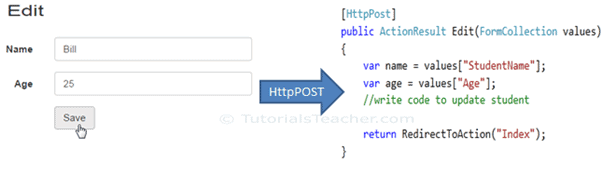
  public string StandardName { get; set; }

}

Output:

[](https://www.tutorialsteacher.com/Content/images/mvc/model-class-binding.png)

**Example 2**: You can also include the FormCollection type parameter in the action method instead of a complex type to retrieve all the values from view form fields, as shown below.

[](https://www.tutorialsteacher.com/Content/images/mvc/formcollection.png)

#### Bind Attribute

ASP.NET MVC framework also enables you to **specify which properties of a model class you want to bind** with the [Bind] attribute. This helps improve the performance by only bind properties that you needed.

In the following example, the Edit() action method will only bind StudentId and StudentName properties of the Student model class.

[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 exclude the properties:

[HttpPost]

public ActionResult Edit([Bind(Exclude = "Age")] Student std)

{

  var name = std.StudentName;

  //write code to update student

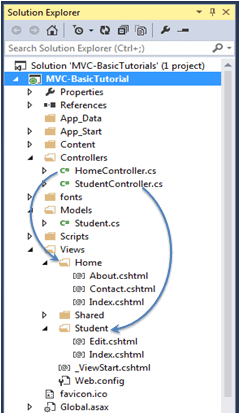
  return RedirectToAction("Index");

}

## View

### View Class

* In MVC, all views must be stored in Views folder.
* Because a controller can render one or more views, for easy maintenance, the MVC requires a separate **sub-folder for each controller** with the same name as a controller, under the Views folder.



The Shared folder contains views, layout views, and partial views, which will be shared among multiple controllers.

* In these sub-folders, have different views for different actions.

**For example**: Create Index view for Index action of Student controller:

In Views\Student\Index.cshtml:

@model IEnumerable<MVC\_BasicTutorials.Models.Student>

@{

  ViewBag.Title = "Index";

  Layout = "~/Views/Shared/\_Layout.cshtml";

}

<h2>Index</h2>

HTML helper

<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, back to the Student model:

public class Student

{

  public int StudentId { get; set; }

  public string StudentName { get; set;  }

  public int Age { get; set;  }

}

And the Index action of Student controller:

public class StudentController : Controller

{

  static 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 }

  };

The View() has param of type IList<Student>

So,

@model IEnumerable<MVC\_BasicTutorials.Models.Student>

  // GET: Student

  public ActionResult Index()

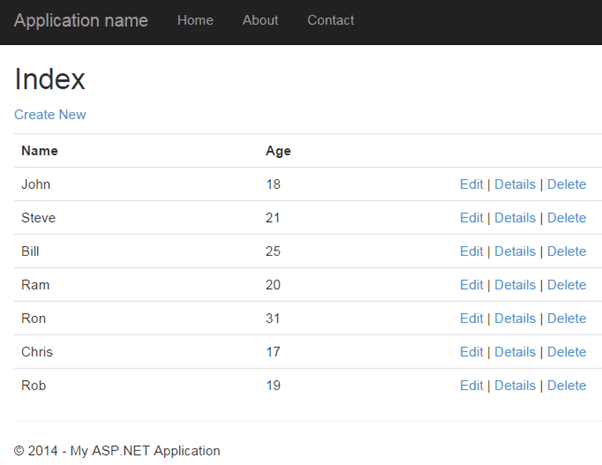
  {

    return View(studentList);

  }

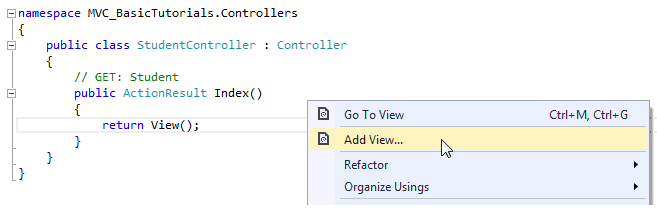
}

Output: Navigate to http://localhost/student:



**Tips:**

* Fast way to create a new view

[](https://www.tutorialsteacher.com/Content/images/mvc/add-view-1.png)

# HTML Helpers

**What:**

The HtmlHelper class generates HTML elements.

For example:

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

would generate anchor tag:

<a href="/Student/Create">Create New</a>.

**Why:**

But the main difference between calling the HtmlHelper methods and using an HTML tags is that the it's designed to **make it easy to bind to view data or model data**.

**How:**

Html is a property of the HtmlHelper class included in base class of razor view WebViewPage

There are many [extension methods for HtmlHelper](https://docs.microsoft.com/en-us/previous-versions/aspnet/dd493095(v=vs.118)) class, which creates different HTML controls.

The following table lists the HtmlHelper methods and HTML control each method renders:

|  |  |  |
| --- | --- | --- |
| **Extension Method** | **Strongly Typed Method** | **Html Control** |
| Html.ActionLink() | NA | <a></a> |
| Html.TextBox() | Html.TextBoxFor() | <input type="textbox"> |
| Html.TextArea() | Html.TextAreaFor() | <input type="textarea"> |
| Html.CheckBox() | Html.CheckBoxFor() | <input type="checkbox"> |
| Html.RadioButton() | Html.RadioButtonFor() | <input type="radio"> |
| Html.DropDownList() | Html.DropDownListFor() | <select> <option> </select> |
| Html.ListBox() | Html.ListBoxFor() | multi-select list box: <select> |
| Html.Hidden() | Html.HiddenFor() | <input type="hidden"> |
| Html.Password() | Html.PasswordFor() | <input type="password"> |
| 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. |

# Exception Handling

<https://www.tutorialsteacher.com/mvc/exception-handling-in-mvc>

# Validation

https://www.tutorialsteacher.com/mvc/implement-validation-in-asp.net-mvc