

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.



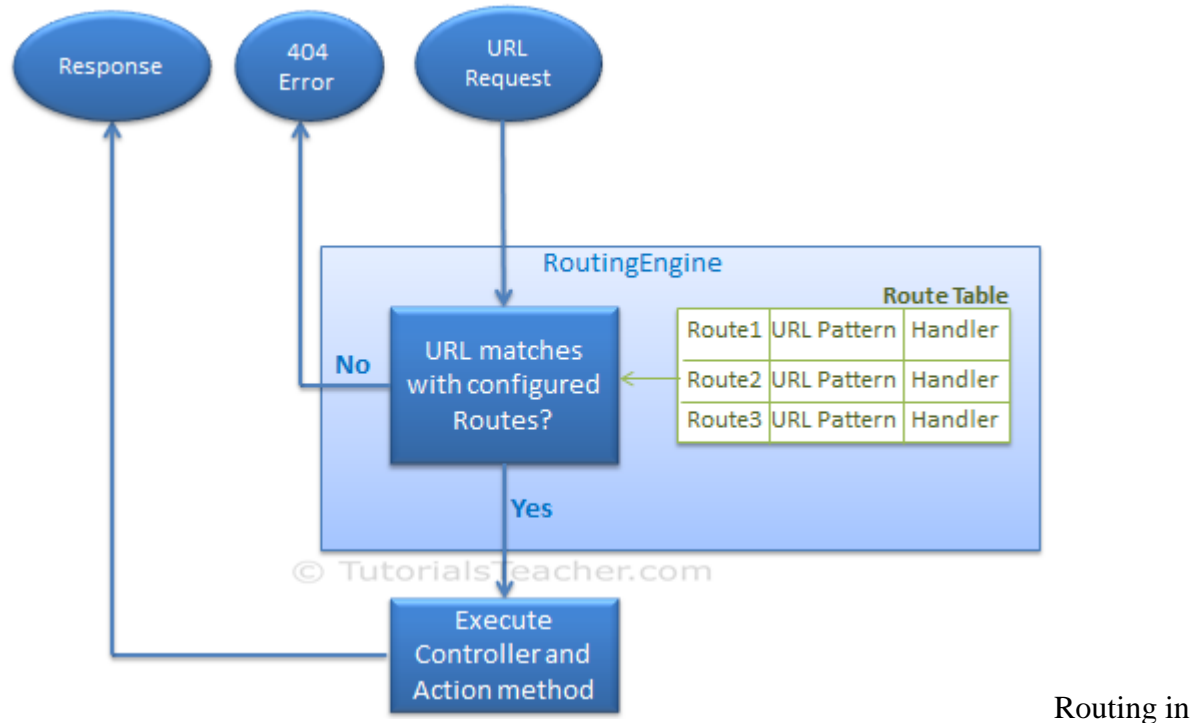
Routing is not specific to MVC framework. It can be used with ASP.NET Webform application or MVC application.

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.

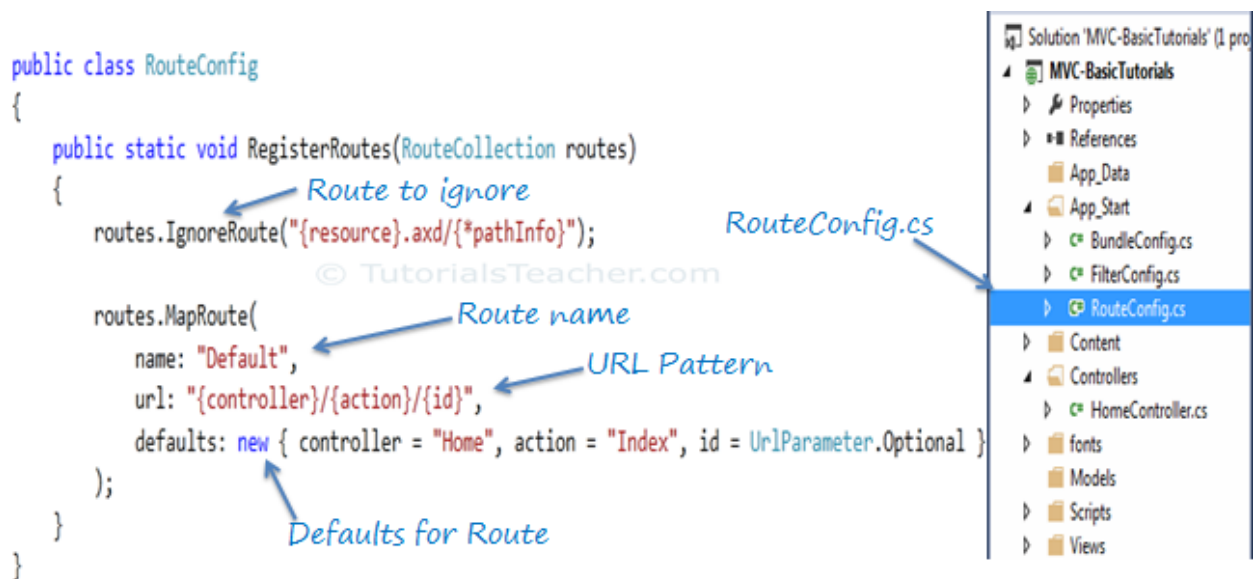
The following figure illustrates the Routing process.



MVC

Configure a Route

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



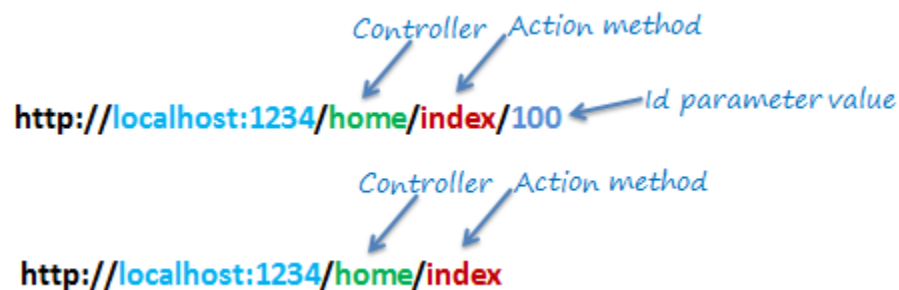
Configure Route in MVC

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

The same way, you can configure other routes using `MapRoute` method of `RouteCollection`. This `RouteCollection` is actually a property of [RouteTable](#) class.

URL Pattern

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



Routing in MVC

If the URL doesn't contain anything after domain name then the default controller and action method will handle the request. For example, `http://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

URL	Controller	Action	Id
http://localhost/home/contact	HomeController	Contact	null
http://localhost/student	StudentController	Index	null
http://localhost/student/edit/123	StudentController	Edit	123

Multiple Routes

You can also configure a custom route using MapRoute extension method. You need to provide at least two parameters in MapRoute, route name and url pattern. The Defaults parameter is optional.

You can register multiple custom routes with different names. Consider the following example where we register "Student" route.

Example: Custom Routes

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

        routes.MapRoute(
            name: "Student",
            url: "students/{id}",
            defaults: new { controller = "Student", action = "Index" }
        );

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

As shown in the above code, URL pattern for the Student route is *students/{id}*, which specifies that any URL that starts with domainName/students, must be handled by StudentController. Notice that we haven't specified {action} in the URL pattern because we want every URL that starts with student should always use Index action of StudentController. We have specified default controller and action to handle any URL request which starts from domainname/students.

MVC framework evaluates each route in sequence. It starts with first configured route and if incoming url doesn't satisfy the URL pattern of the

route then it will evaluate second route and so on. In the above example, routing engine will evaluate Student route first and if incoming url doesn't starts with /students then only it will consider second route which is default route.

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

URL	Controller	Action	Id
http://localhost/student/123	StudentController	Index	123
http://localhost/student/index/123	StudentController	Index	123
http://localhost/student?Id=123	StudentController	Index	123

Route Constraints

You can also apply restrictions on the value of parameter by configuring route constraints. For example, the following route applies a restriction on id parameter that the value of an id must be numeric.

Example: Route Constraints

```
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.

Register Routes

Now, after configuring all the routes in RouteConfig class, you need to register it in the Application_Start() event in the Global.asax. So that it includes all your routes into RouteTable.

Example: Route Registration

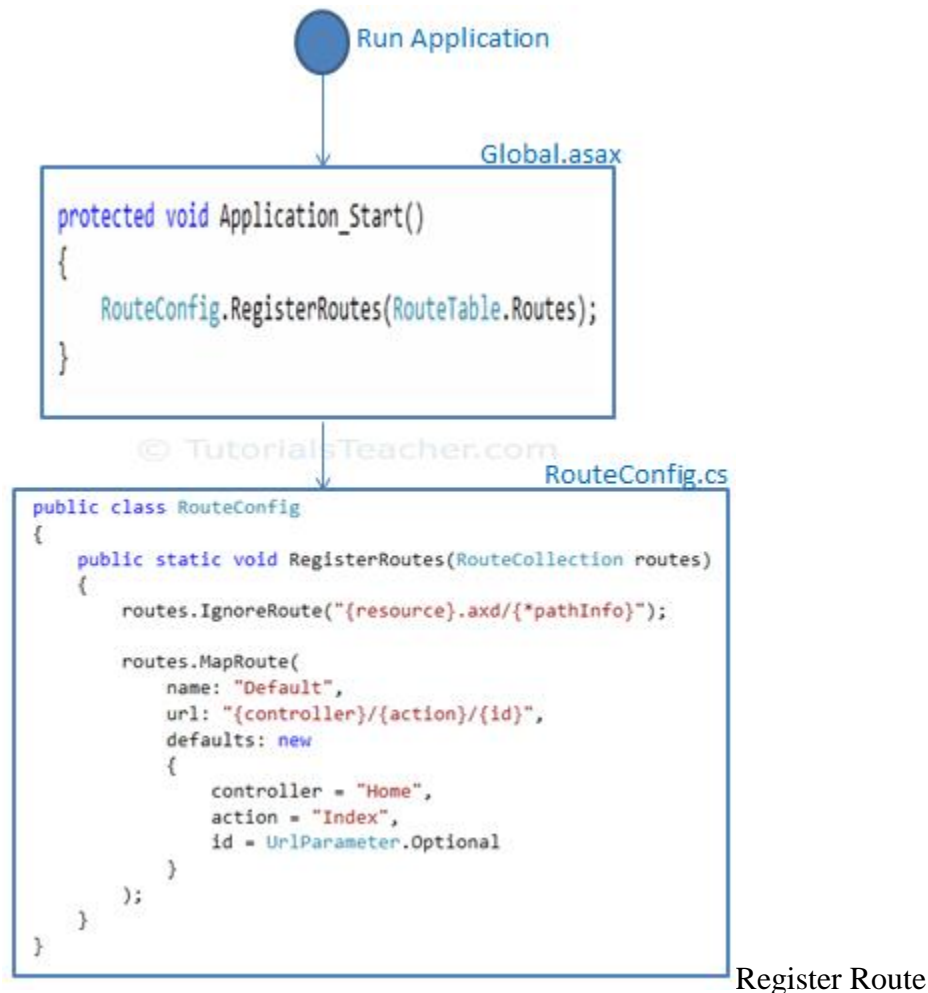
```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
```

```

        RouteConfig.RegisterRoutes(RouteTable.Routes);
    }
}

```

The following figure illustrate Route registration process.



Thus, routing plays important role in MVC framework.



Points to Remember :

1. Routing plays important role in MVC framework. Routing maps URL to physical file or class (controller class in MVC).
2. Route contains URL pattern and handler information. URL pattern starts after domain name.
3. Routes can be configured in RouteConfig class. Multiple custom routes can also be configured.
4. Route constraints applies restrictions on the value of parameters.

5. Route must be registered in Application_Start event in Global.ascx.cs file.

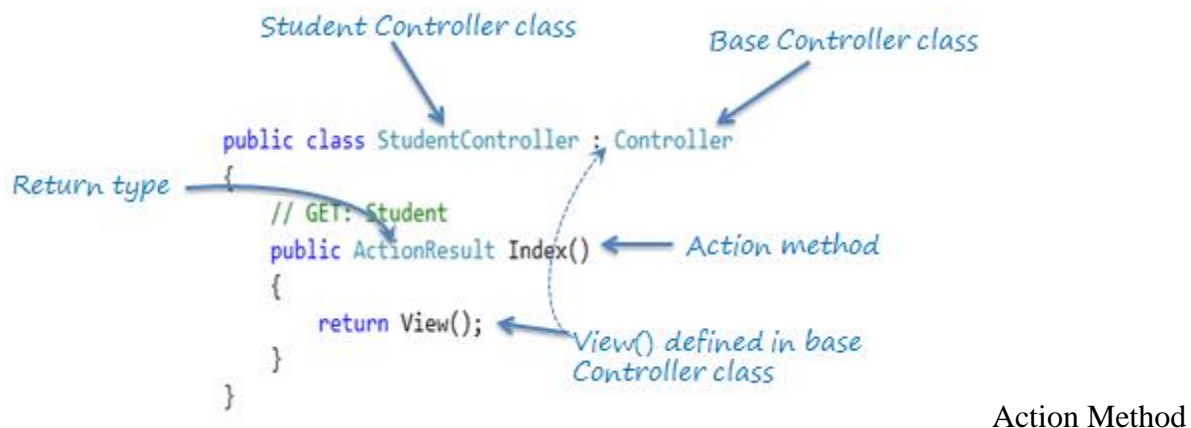
Action method

In this section, you will learn about the action method of controller class.

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



As you can see in the above figure, Index method is a public method and it returns `ActionResult` using the `View()` method. The `View()` method is defined in the Controller base class, which returns the appropriate `ActionResult`.

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.

Default Route:

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

```
        id = UrlParameter.Optional  
    });
```

However, you can change the default action name as per your requirement in RouteConfig class.

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 ActionResult class is a base class of all the above result classes, so it can be return type of action methods which returns any type of result listed above. However, you can specify appropriate result class as a return type of action method.

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.	
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.

Example: Action method parameters

```
[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");  
}
```

Please note that action method parameter can be [Nullable Type](#).

By default, the values for action method parameters are retrieved from the request's data collection. The data collection includes name/values pairs for form data or query string values or cookie values. Model binding in ASP.NET MVC automatically maps the URL query string or form data collection to the action method parameters if both names are matching. Visit [model binding](#) section for more information on it.



Points to Remember :

1. All the public methods in the Controller class are called Action methods.
2. Action method has 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.
3. ActionResult is a base class of all the result type which returns from Action method.
4. Base Controller class contains methods that returns appropriate result type e.g. View(), Content(), File(), JavaScript() etc.
5. Action method can include [Nullable](#) type parameters.

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.

Example: ActionName

```

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.

Example: NonAction

```

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

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

```



Points to Remember :

1. MVC framework routing engine uses Action Selectors attributes to determine which action method to invoke.

2. Three action selectors attributes are available in MVC 5
 - ActionName
 - NonAction
 - ActionVerbs
3. ActionName attribute is used to specify different name of action than method name.
4. NonAction attribute marks the public method of controller class as non-action method. It cannot be invoked.

ActionVerbs

In this section, you will learn about the ActionVerbs selectors attribute.

The ActionVerbs 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 ActionVerbs, 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 diagram illustrates how different HTTP verbs map to specific action methods in a controller. It shows two examples of the `Edit` action method.

Example 1: HttpGET

Request: `http://localhost/Student/Edit/1`

Method: `HttpGet` (indicated by a blue arrow)

Code:

```
public ActionResult Edit(int Id)
{
    var std = students.Where(s => s.StudentId == Id).FirstOrDefault();

    return View(std);
}
```

© TutorialsTeacher.com

Example 2: HttpPOST

Request: `http://localhost/Student/Edit`

Method: `HttpPost` (indicated by a blue arrow)

Code:

```
[HttpPost]
public ActionResult Edit(Student std)
{
    //update database here..

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

ActionVerbs

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.
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.

Visit W3.org for more information on Http Methods.

The following example shows different action methods supports different ActionVerbs:

Example: ActionVerbs

```
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:

Example: AcceptVerbs

```

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

```



Points to Remember :

1. ActionVerbs are another Action Selectors which selects an action method based on request methods e.g POST, GET, PUT etc.
2. Multiple action methods can have same name with different action verbs. Method overloading rules are applicable.
3. Multiple action verbs can be applied to a single action method using AcceptVerbs attribute.

- [Share](#)
- [Tweet](#)
- [Share](#)
- [Whatsapp](#)

ASP.NET MVC - Model

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

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

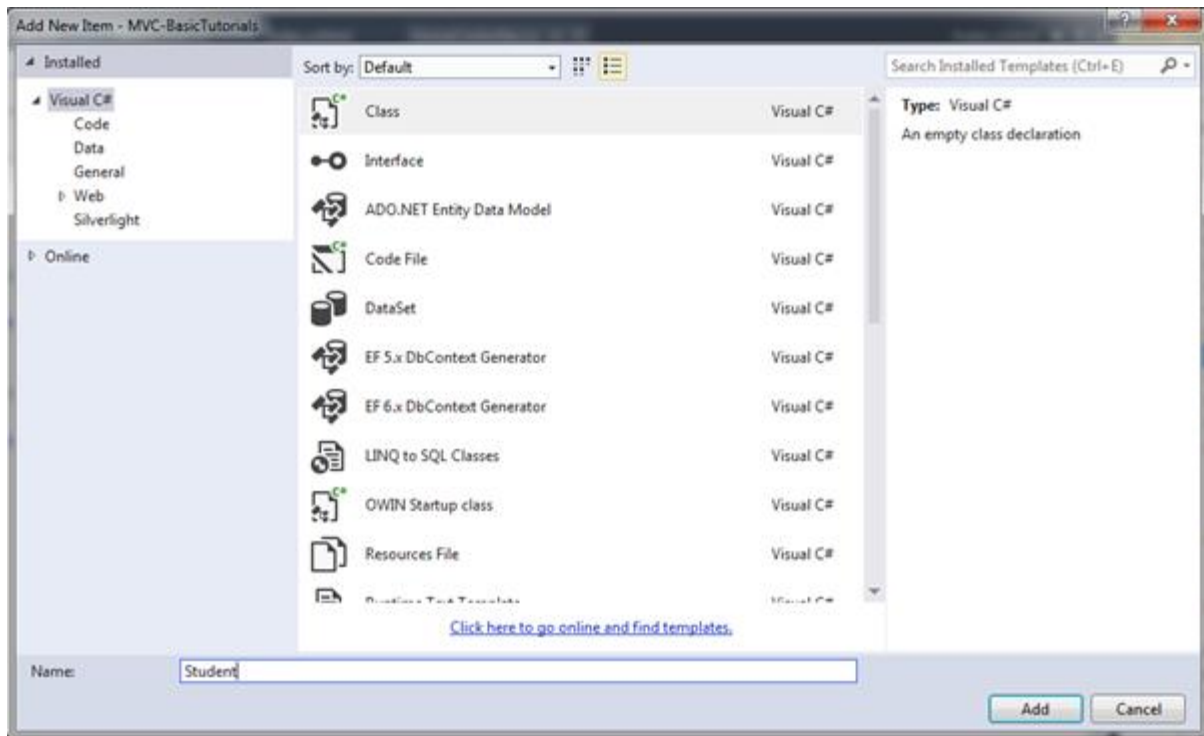
Model class holds data in public properties. All the Model classes reside in the Model folder in MVC folder structure.

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

Adding a Model

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

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



Create Model Class

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

Example: Model class

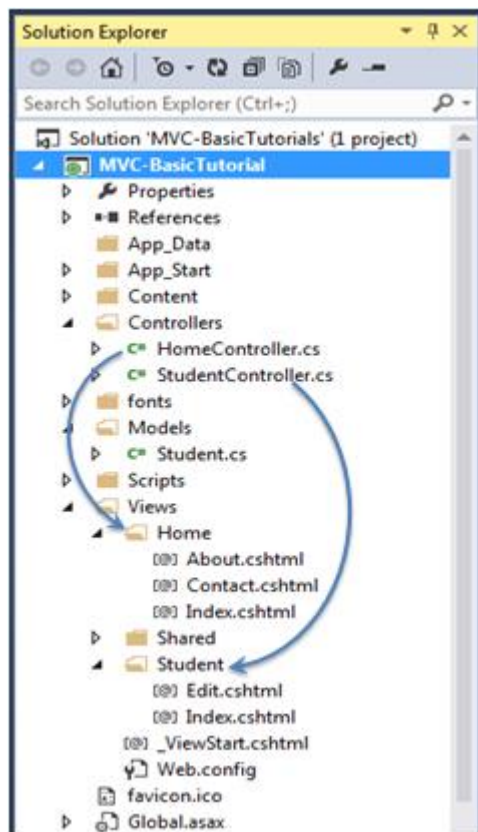
```
namespace MVC_BasicTutorials.Models
{
    public class Student
    {
        public int StudentId { get; set; }
        public string StudentName { get; set; }
        public int Age { get; set; }
    }
}
```

View in ASP.NET MVC

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

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

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



View folders for Controllers

Note:

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

Razor View Engine

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

engine maximize the speed of writing code by minimizing the number of characters and keystrokes required when writing a view. Razor views files have .cshtml or vbhtml extension.

ASP.NET MVC supports following types of view files:

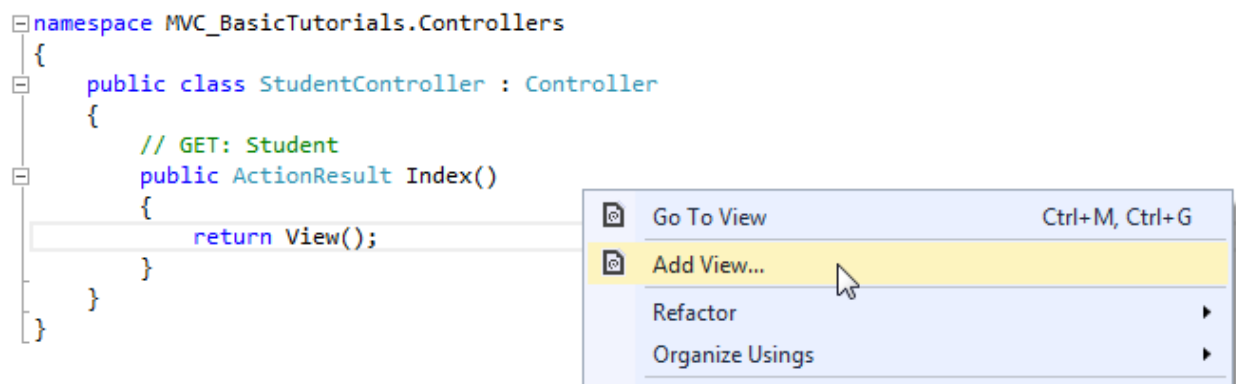
View file extension	Description
.cshtml	C# Razor view. Supports C# with html tags.
.vbhtml	Visual Basic Razor view. Supports Visual Basic with html tags.
.aspx	ASP.Net web form
.ascx	ASP.NET web control

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

Create New View

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

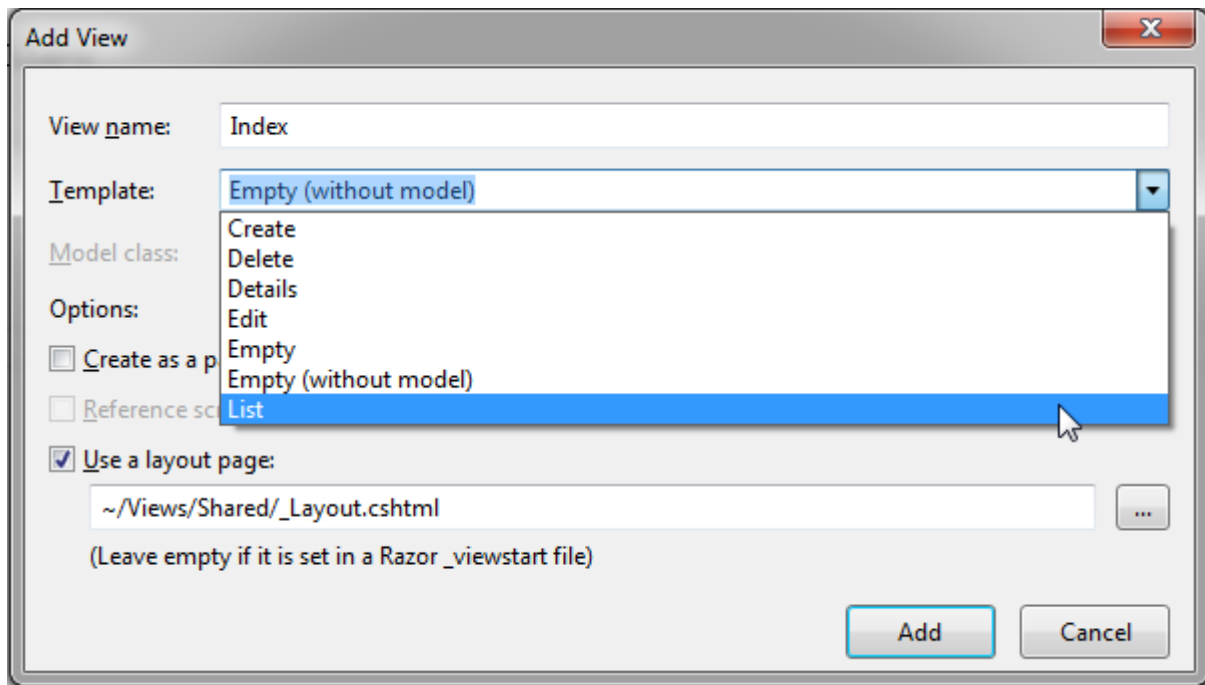
We will create a view, which will be rendered from Index method of StudentController. So, open a StudentController class -> right click inside Index method -> click **Add View..**



Create a View

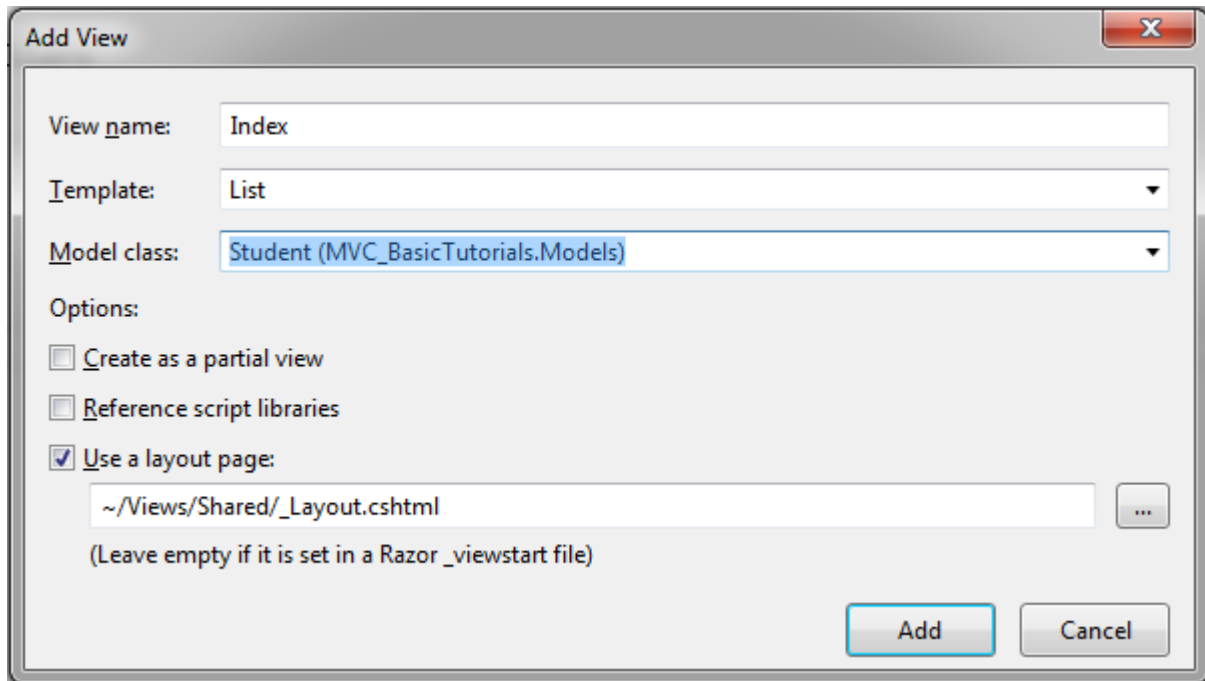
In the Add View dialogue box, keep the view name as Index. It's good practice to keep the view name the same as the action method name so that you don't have to specify view name explicitly in the action method while returning the view.

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



View

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

The image shows a standard Windows-style dialog box titled "Add View". It has a close button (X) in the top right corner. The dialog contains several input fields and checkboxes. The "View name:" field is a text box containing the word "Index". The "Template:" field is a dropdown menu with "List" selected. The "Model class:" field is a dropdown menu with "Student (MVC_BasicTutorials.Models)" selected. Below these is a section labeled "Options:" containing three checkboxes: "Create as a partial view" (unchecked), "Reference script libraries" (unchecked), and "Use a layout page:" (checked). Below the "Use a layout page:" checkbox is a text box containing the path "~/Views/Shared/_Layout.cshtml" and a browse button (three dots). At the bottom of the dialog are two buttons: "Add" and "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

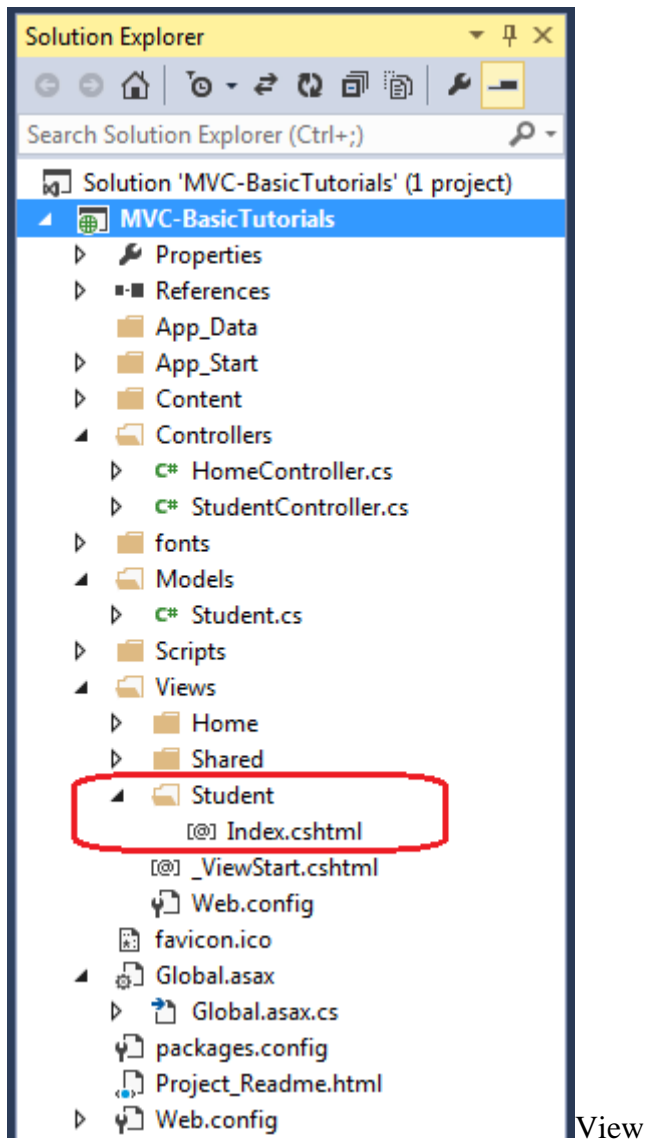
(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

View

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

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



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

Views\Student\Index.cshtml:

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

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

```
<h2>Index</h2>
```

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

```

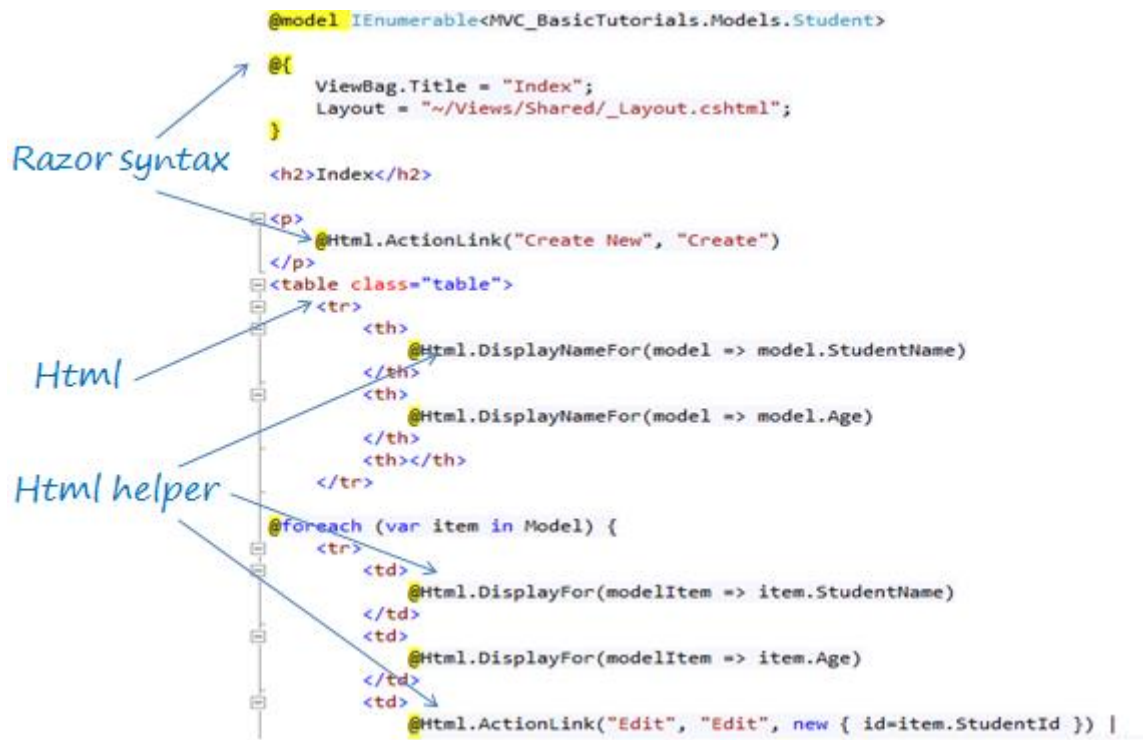
<table class="table">
  <tr>
    <th>
      @Html.DisplayNameFor(model => model.StudentName)
    </th>
    <th>
      @Html.DisplayNameFor(model => model.Age)
    </th>
  </tr>

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

```

</table>

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



Index.cshtml

The above Index view would look like below.

Index

[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

Index View

Note:

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



Points to Remember :

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

Integrate Controller, View and Model

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

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

Example: StudentController

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

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

Example: Student Model class

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

Example: Index.cshtml to display student list

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

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

<h2>Index</h2>

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



```

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Age)
        </th>
        <th></th>
    </tr>

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

```

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

Example: Passing Model from Controller

```

public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        var studentList = new List<Student>{
            new Student() { StudentId = 1, StudentName = "John", Age
= 18 } ,
            new Student() { StudentId = 2, StudentName = "Steve", Age
= 21 } ,
            new Student() { StudentId = 3, StudentName = "Bill", Age
= 25 } ,
            new Student() { StudentId = 4, StudentName = "Ram" , Age
= 20 } ,
            new Student() { StudentId = 5, StudentName = "Ron" , Age
= 31 } ,
            new Student() { StudentId = 4, StudentName = "Chris" , Age
= 17 } ,

```

```

        new Student() { StudentId = 4, StudentName = "Rob" , Age
= 19 }
        };
        // Get the students from the database in the real application

        return View(studentList);
    }
}

```

[Try it](#)

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

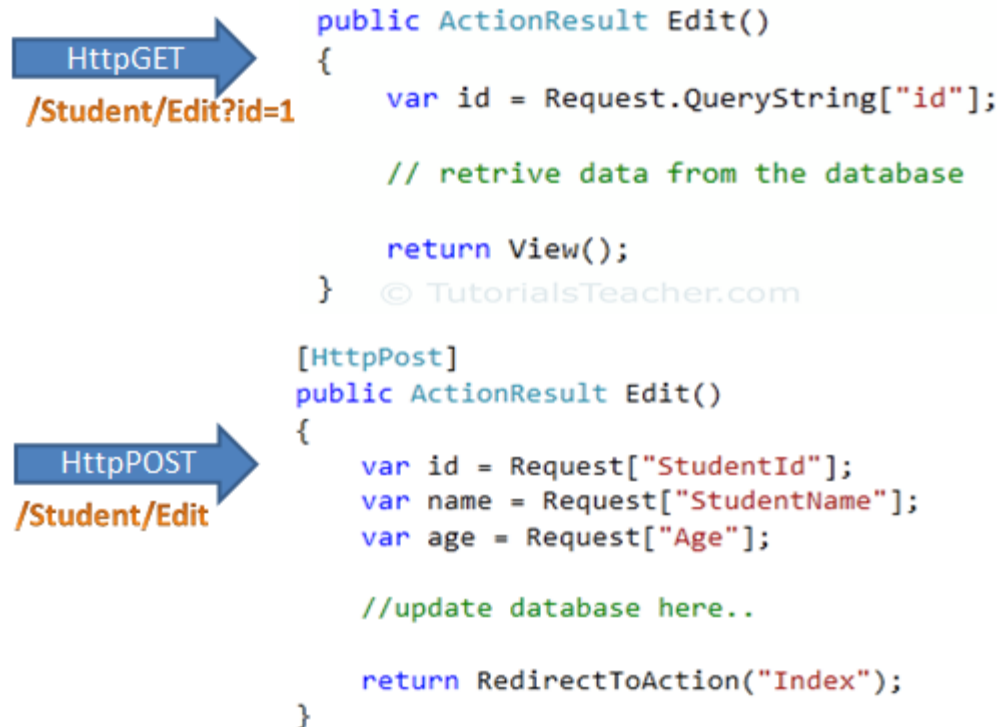
Now, you can run the MVC project by pressing F5 and navigate to <http://localhost/Student>. You will see following view in the browser.

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		

Model Binding

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

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



```
public ActionResult Edit()
{
    var id = Request.QueryString["id"];

    // retrieve data from the database

    return View();
} © TutorialsTeacher.com

[HttpPost]
public ActionResult Edit()
{
    var id = Request["StudentId"];
    var name = Request["StudentName"];
    var age = Request["Age"];

    //update database here..

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

Accessing

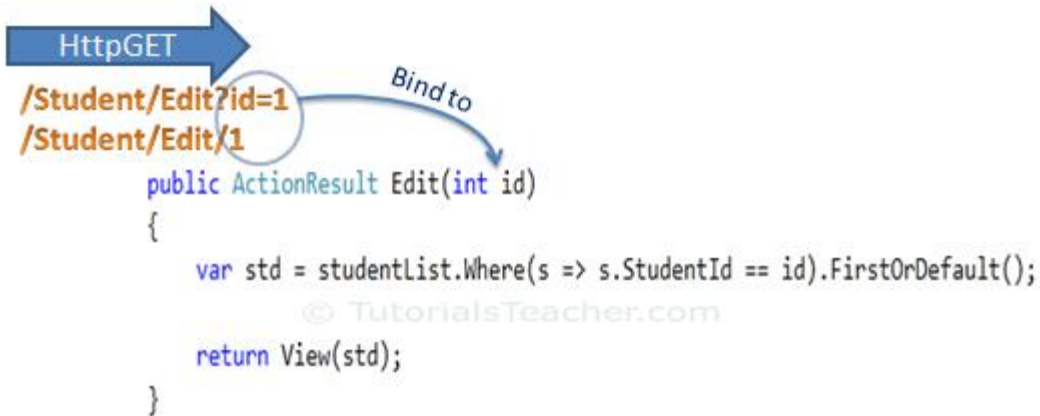
Request Data

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

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

Binding to Primitive type

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



Model

Binding



This binding is case insensitive. So "id" parameter can be "ID" or "Id".

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.

Example: Convert QueryString to Action Method Parameters

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

    return View();
}
```

Binding to Complex type

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

Consider the following model classes.

Example: Model classes in C#

```
public class Student
{
```

```

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

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

```

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

Example: Action Method with Class Type Parameter

```

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

    //update database here..

    return RedirectToAction("Index");
}

```

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



Model Binding

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

FormCollection

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



Model Binding

Bind Attribute

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

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

Example: Binding Parameters

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

    //write code to update student

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

You can also use Exclude properties as below.

Example: Exclude Properties in Binding

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

    //write code to update student

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

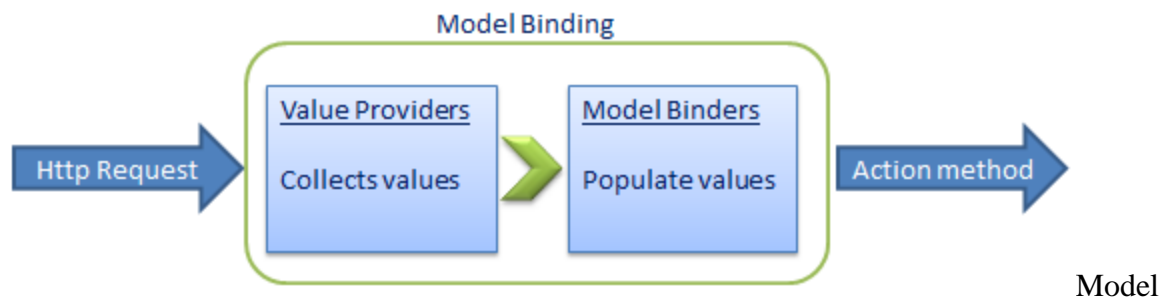
}

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

Inside Model Binding

As you have seen that Model binding automatically converts request values into a primitive or complex type object. Model binding is a two step process. First, it collects values from the incoming http request and second, populates primitive type or complex type with these values.

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



Binding in MVC

Default value provider collection evaluates values from the following sources:

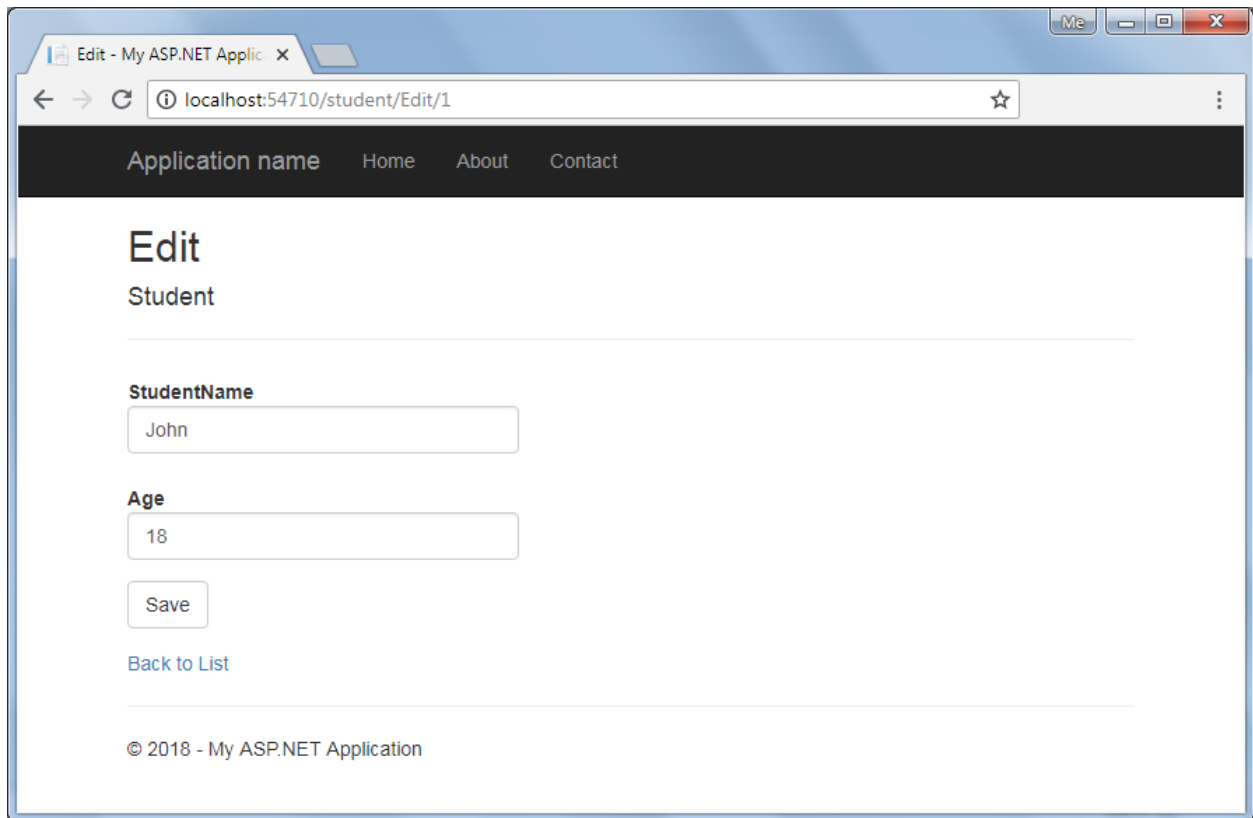
1. Previously bound action parameters, when the action is a child action
2. Form fields (Request.Form)
3. The property values in the JSON Request body (Request.InputStream), but only when the request is an AJAX request
4. Route data (RouteData.Values)
5. Querystring parameters (Request.QueryString)
6. Posted files (Request.Files)

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

Visit MSDN for detailed information on [Model binding](#).

Create Edit View in ASP.NET MVC

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.



Edit View

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.

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

1 <http://localhost/Edit/1>
HttpGET

```
public ActionResult Edit(int Id)
{
    var std = students.Where(s => s.StudentId == Id).FirstOrDefault();
    return View(std);
}
```



Edit

Student

```
[HttpPost]
public ActionResult Edit(Student std)
{
    var name = std.StudentName;
    var age = std.Age;
    //write code to update student

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

HttpPOST
<http://localhost/Edit>

Name	<input type="text" value="John"/>
Age	<input type="text" value="18"/>
	<input type="button" value="Save"/>

Editing Steps in MVC

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.

Student Model - C#:

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

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

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

Step:

1

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.

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

Index View

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 is matches. Please make sure that parameter name matches with the query string.

Example: HttpGet Edit() Action method - C#

```
using MVC_BasicTutorials.Models;

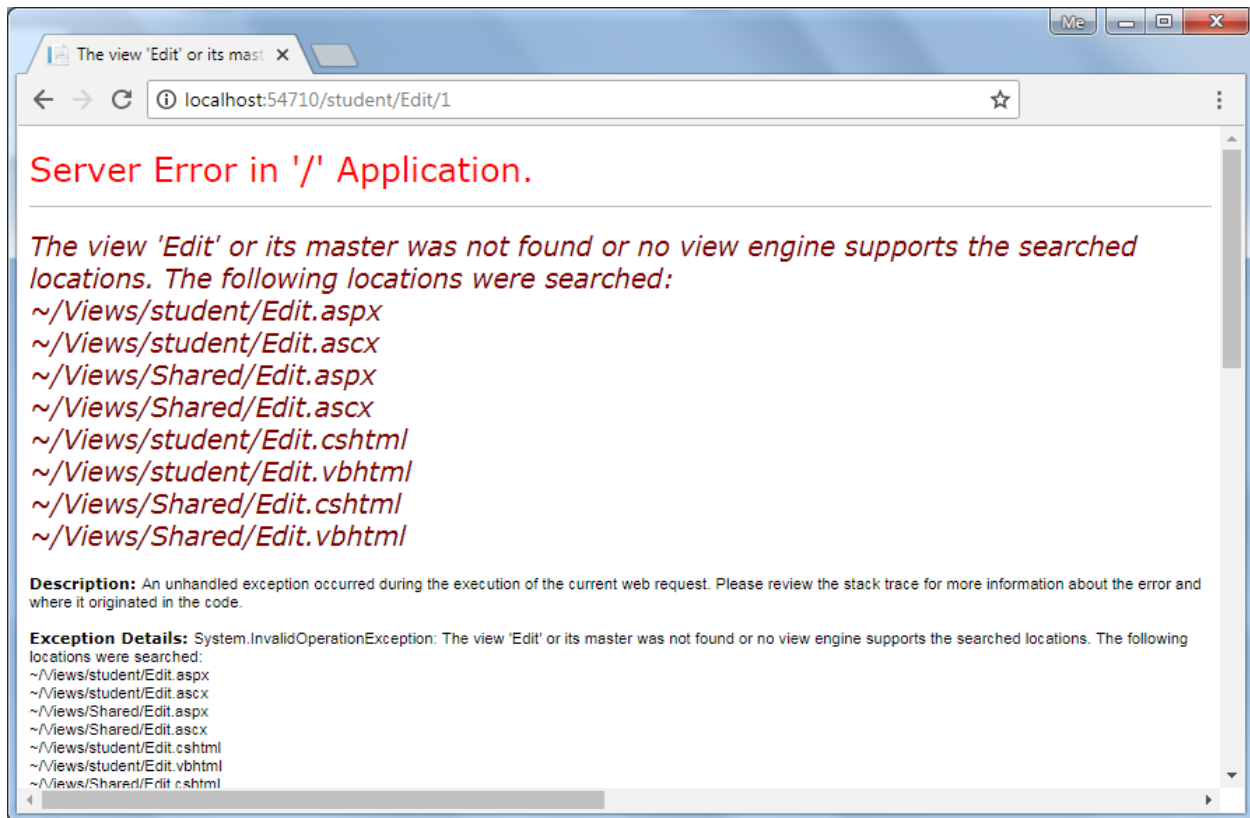
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        IList<Student> studentList = new List<Student>() {
            new Student(){ StudentId=1, StudentName="John", Age = 18 },
            new Student(){ StudentId=2, StudentName="Steve", Age = 21 },
            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 Id)
        {
            //Get the student from studentList sample collection for demo purpose.
            //Get the student from the database in the real application
            var std = studentList.Where(s => s.StudentId == Id).FirstOrDefault();

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

As you can see in the above Edit method, we have used a LINQ query to get the Student from the sample studentList collection whose StudentId matches with supplied StudentId, and then we inject that Student object into View. In a real life application, you can get the student from the database instead of sample collection.

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

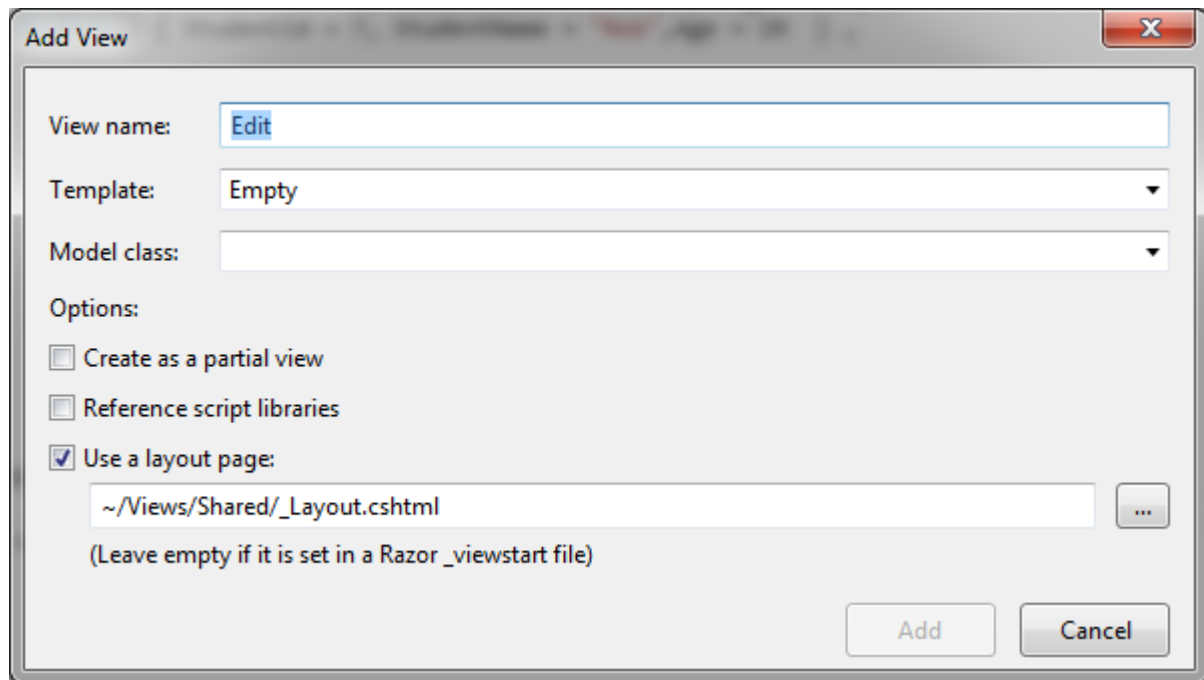


Edit View Error

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.

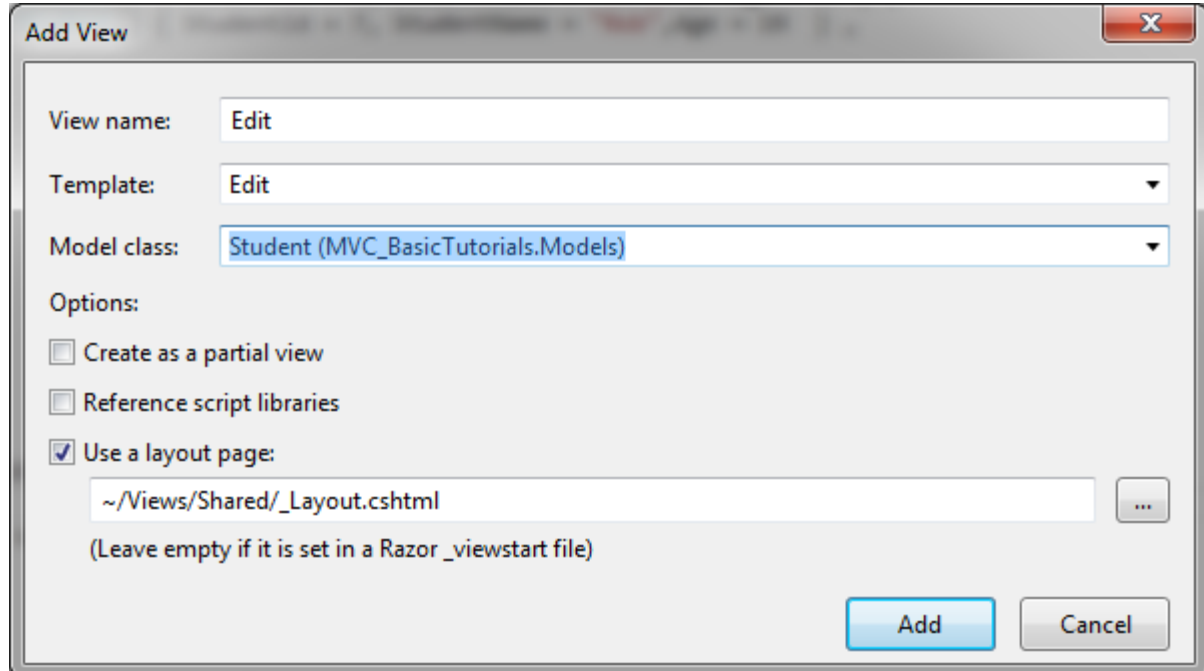
Step 3:
To create Edit view, right click inside Edit action method and click on **Add View..** It will open Add View dialogue.

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



Edit View

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



Edit View

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

Edit.cshtml:

```
@model MVC_BasicTutorials.Models.Student
@{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Edit</h2>

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

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

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

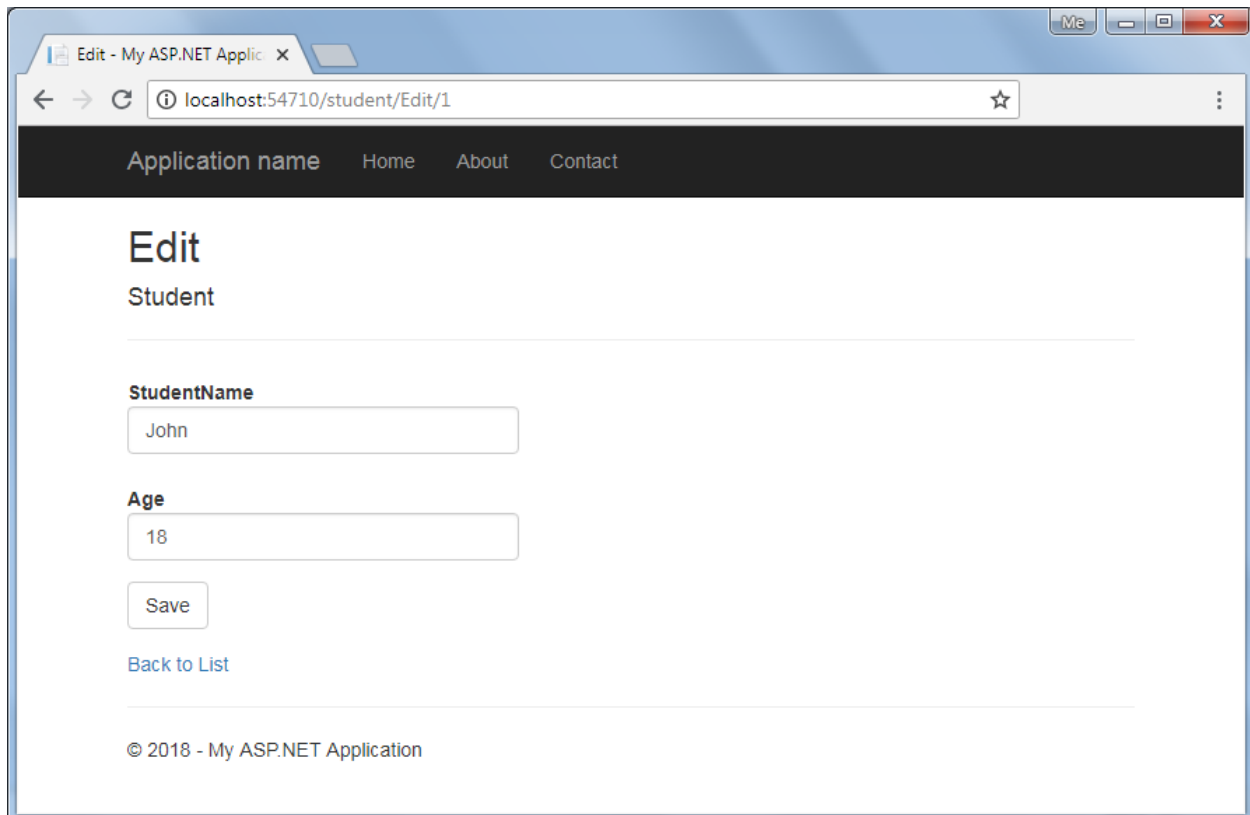
        <div class="form-group">
            @Html.LabelFor(model => model.Age, htmlAttributes: new { @class = "control-
label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Age, new { htmlAttributes = new { @class
= "form-control" } })
                @Html.ValidationMessageFor(model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

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

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

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.

A screenshot of a web browser window showing the 'Edit - My ASP.NET Application' page. The address bar shows 'localhost:54710/student/Edit/1'. The page has a dark navigation bar with links: 'Application name', 'Home', 'About', and 'Contact'. The main content area is titled 'Edit Student'. It contains two text input fields: 'StudentName' with the value 'John' and 'Age' with the value '18'. Below these fields is a 'Save' button and a blue link labeled 'Back to List'. At the bottom, there is a copyright notice: '© 2018 - My ASP.NET Application'.

Edit View

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.

Example: POST Method in MVC

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



```

        return RedirectToAction("Index");
    }

```

As you can see in the above code, the Edit() method requires a Student object as an input parameter. The Edit() view will automatically binds form's data collection to the student model parameter. Please visit [Model Binding](#) section for more information. Here, you can update the information to the database and redirect it to Index action. (we have not written code to update database here for demo purpose)

Now, clicking on the Save button in the Edit view will save the updated information and redirect it to the Index() action method.

In this way, you can provide edit functionality using a default scaffolding Edit template. However, you can also create an Edit view without using an Edit scaffolding template.

The following example demonstrates the StudentController class with all the action methods.

Example: Controller in C#

```

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 Id)
        {
            //Get the student from studentList sample collection for demo purpose.
            //Get the student from the database in the real application
            var std = studentList.Where(s => s.StudentId == Id).FirstOrDefault();

            return View(std);
        }
    }
}

```

```

    }

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

        return RedirectToAction("Index");
    }
}

```

Razor Syntax

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

Razor syntax has following Characteristics:

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

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

Inline expression

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

C# Razor Syntax

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

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

Output:

```
Razor syntax demo
08-09-2014
```

Multi-statement Code block

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

Example: Server side Code in Razor Syntax

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

Output:

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

Display Text from Code Block

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

Example: Display Text in Razor Syntax

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

Output:

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

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

Example: Text in Razor Syntax

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

Output:

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

if-else condition

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

Example: if else in Razor

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

Output:

2014 is not a leap year.

for loop

Example: for loop in Razor

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

Output:

0
1
2
3
4

Model

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

Example: Use Model in Razor

```
@model Student

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

Output:

Student Detail:

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

Declare Variables

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

Example: Variable in Razor

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

Output:

Hello World!

So this was some of the important razor syntaxes. Visit asp.net to learn [razor syntax](#) in detail.



Points to Remember :

1. Use @ to write server side code.
2. Server side code block starts with @{* code * }
3. Use @: or <text></> to display text from code block.
4. if condition starts with @if{ }
5. for loop starts with @for
6. @model allows you to use model object anywhere in the view.