**ASP.NET MVC Version History**

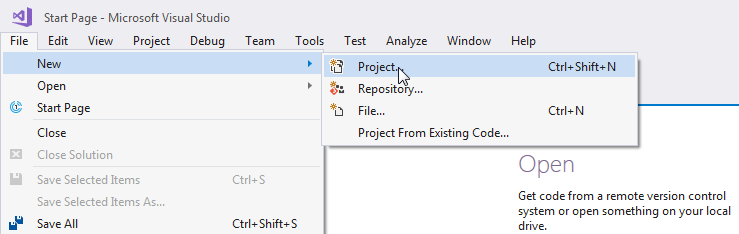
Microsoft had introduced ASP.NET MVC in .Net 3.5, since then lots of new features have been added.

The following table list brief history of ASP.NET MVC.

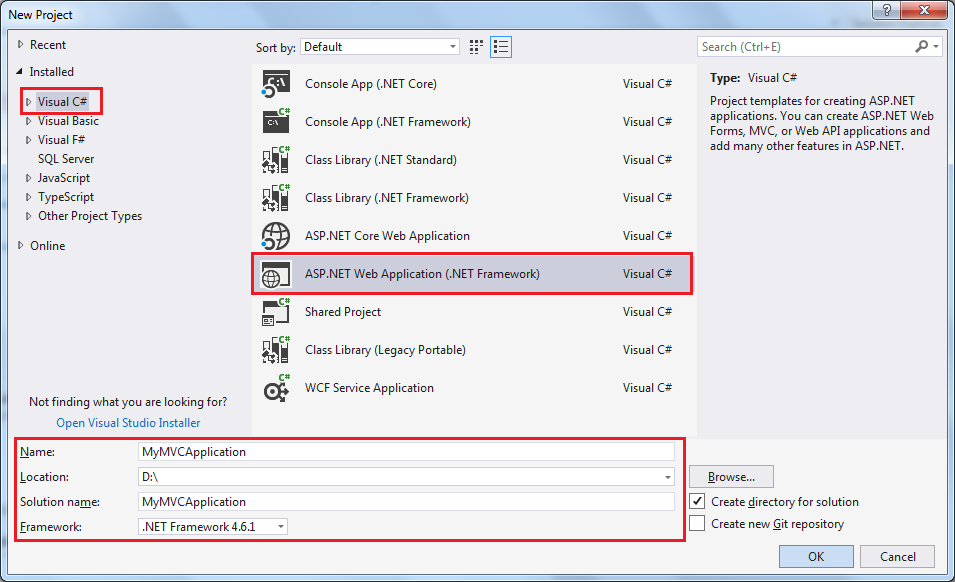
| MVC Version | Visual Studio | .Net Version | Release date | Features |
| --- | --- | --- | --- | --- |
| MVC 1.0 | VS2008 | .Net 3.5 | 13-Mar-2009 | * MVC architecture with webform engine * Routing * HTML Helpers * Ajax Helpers * Auto binding |
| MVC 2.0 | VS 2008, | .Net 3.5/4.0 | 10-Mar-2010 | * Area * Asynchronous controller * Html helper methods with lambda expression * DataAnnotations attributes * Client side validation * Custom template * Scaffolding |
| MVC 3.0 | VS 2010 | .Net 4.0 | 13-Jan-2011 | * Unobtrusive javascript validation * Razor view engine * Global filters * Remote validation * Dependency resolver for IoC * ViewBag |
| MVC 4.0 | VS 2010 SP1, VS 2012 | .NET 4.0/4.5 | 15-Aug-2012 | * Mobile project template * Bundling and minification * Support for Windows Azure SDK |
| MVC 5.0 | VS 2013 | .NET 4.5 | 17-oct-2013 | * Authentication filters * Bootstrap support * New scaffolding items * ASP.Net Identity |
| **MVC 5.2** - Current | VS 2013 | .NET 4.5 | 28-Aug-2014 | * Attribute based routing * bug fixes and minor features upate |

## Create first simple MVC application

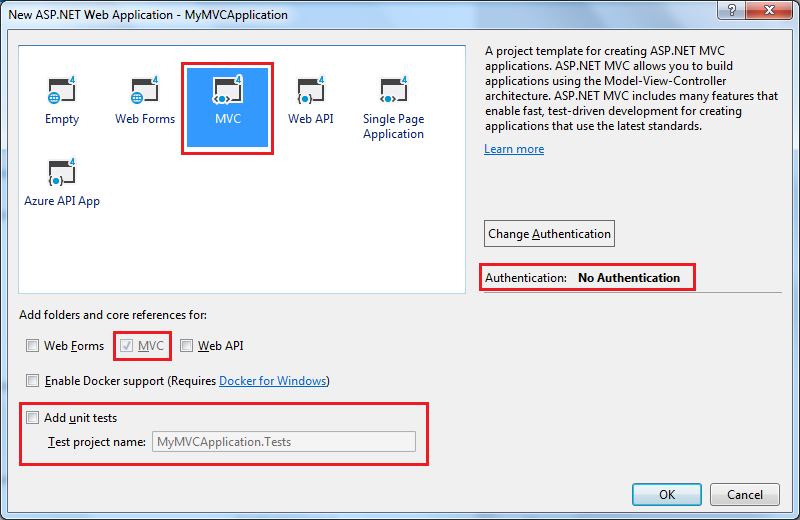
First of all, open a Visual Studio 2017 Community edition and select **File menu** -> **New** -> **Project**as shown below.



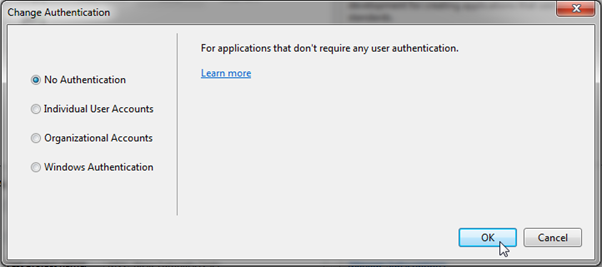
From the **New Project** dialog as shown below, expand Visual C# node and select **Web** in the left pane, and then select **ASP.NET Web Application (.NET Framework)** in the middle pane. Enter the name of your project MyMVCApplication. (You can give any appropriate name for your application). Also, you can change the location of the MVC application by clicking on **Browse..** button. Finally, click **OK.**

Create MVC Project in Visual Studio

From the **New ASP.NET Web Application** dialog, select MVC (if not selected already) as shown below.

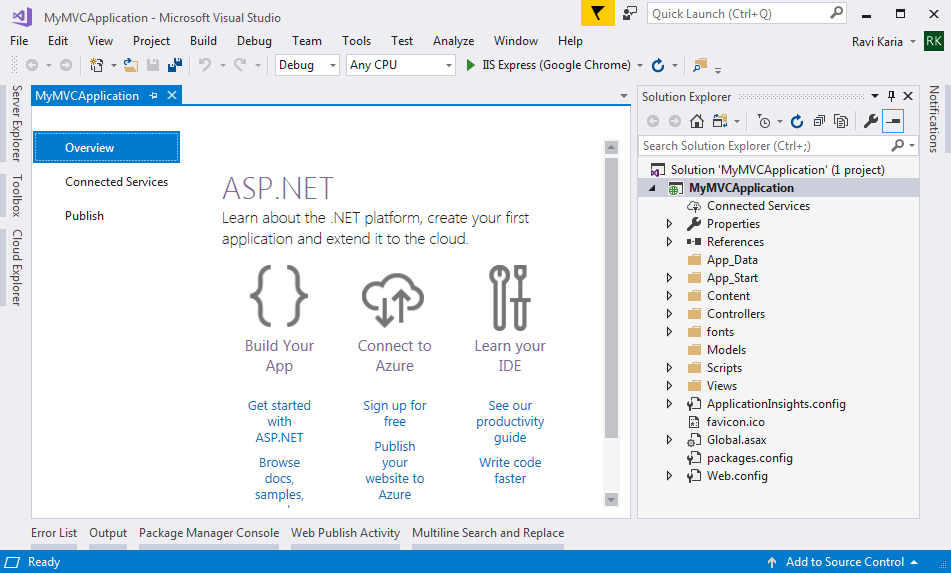


You can also change the authentication by clicking on **Change Authentication** button. You can select appropriate authentication mode for your application as shown below.

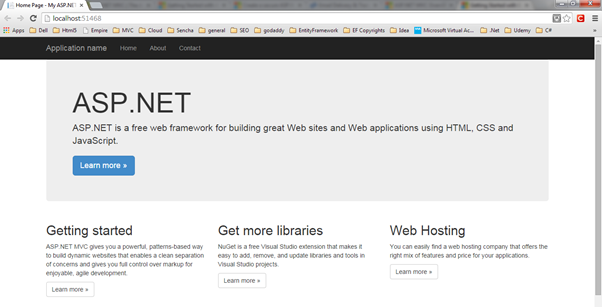
Select Authenctication Type

Here, we are keeping the default authentication for our application which is No Authentication. Click **OK** to continue.

Wait for some time till Visual Studio creates a simple MVC project using default template as shown below.

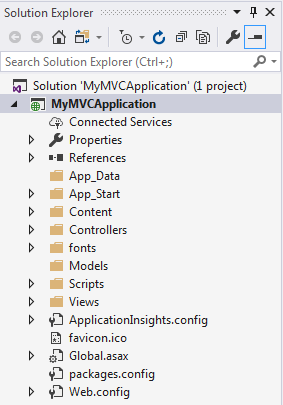
First MVC Application

Now, press F5 to run the project in debug mode or Ctrl + F5 to run the project without debugging. It will open home page in the browser as shown below.

Run MVC Application

# ASP.NET MVC Folder Structure

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



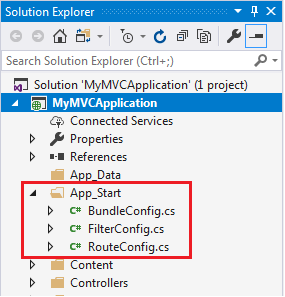
MVC Folder Structure

### 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.

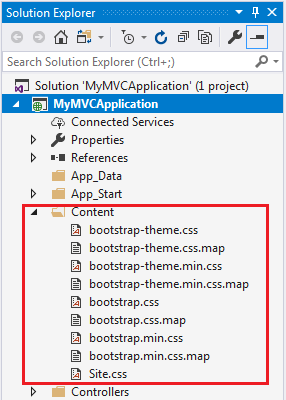
### App\_Start

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.



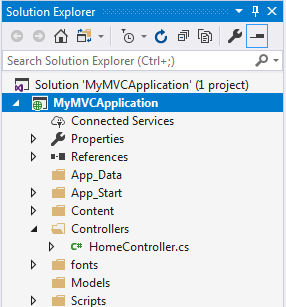
### Content

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.

Content Folder

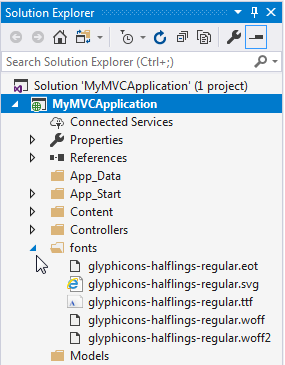
### Controllers

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

Controller Folder

### fonts

Fonts folder contains custom font files for your application.

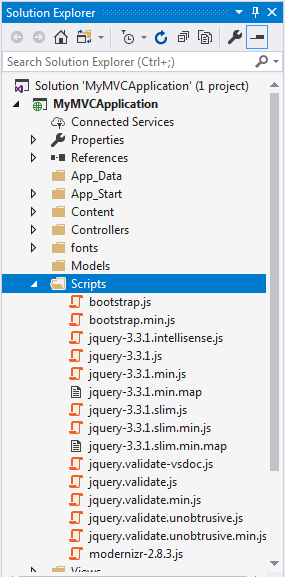


### Models

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

### Scripts

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

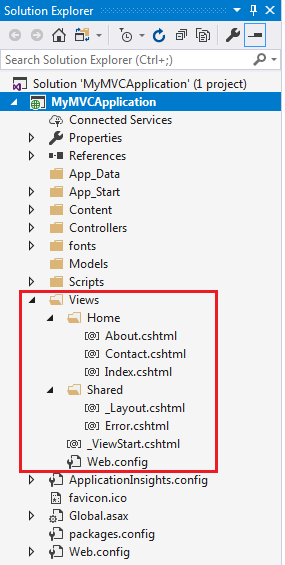
Scripts Folder

### Views

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

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

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

View Folder

Additionally, MVC project also includes following configuration files:

### Global.asax

Global.asax allows you to write code that runs in response to application level events, such as Application\_BeginRequest, application\_start, application\_error, session\_start, session\_end etc.

### Packages.config

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

### Web.config

Web.config file contains application level configurations.

# Routing in MVC

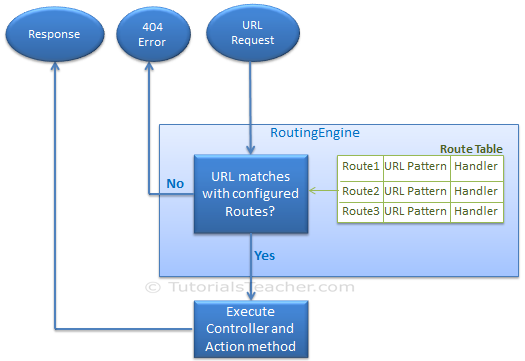
In the ASP.NET Web Forms application, every URL must match with a specific .aspx file. For example, a URL http://domain/studentsinfo.aspx must match with the file studentsinfo.aspx that contains code and markup for rendering a response to the browser.

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

## Route

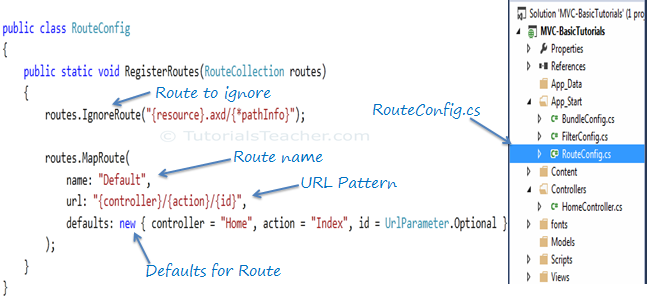
Route defines the URL pattern and handler information. All the configured routes of an application stored in RouteTable and will be used by Routing engine to determine appropriate handler class or file for an incoming request.

The following figure illustrates the Routing process.

Routing in MVC

## Configure a Route

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

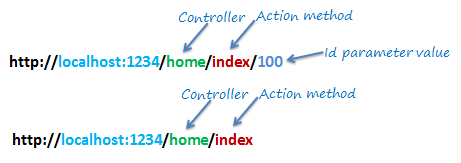
Configure Route in MVC

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

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

### URL Pattern

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

Routing in MVC

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

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

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

## 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()

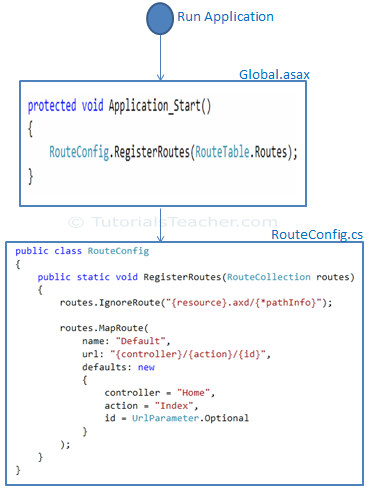
{

RouteConfig.RegisterRoutes(RouteTable.Routes);

}

}

The following figure illustrate Route registration process.

Register Route

Thus, routing plays important role in MVC framework.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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.

# Controller

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

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

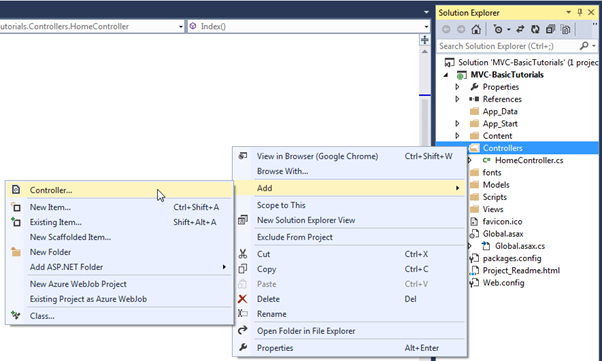
In ASP.NET MVC, every controller class name must end with a word "Controller". For example, controller for home page must be HomeController and controller for student must be StudentController. Also, every controller class must be located in Controller folder of MVC folder structure.

## Adding a New Controller

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

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

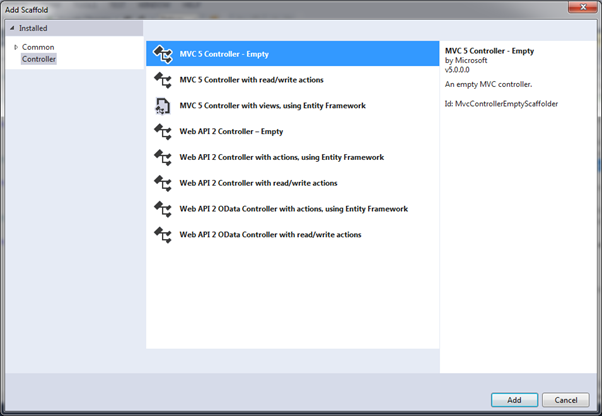
In the Visual Studio, right click on the Controller folder -> select **Add** -> click on **Controller..**

Add New Controller

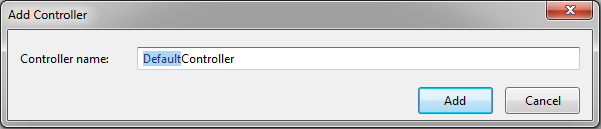
This opens Add Scaffold dialog as shown below.

 Note:

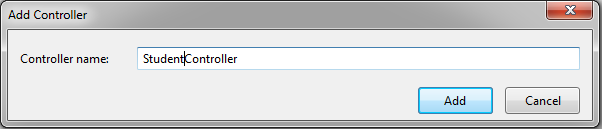
Scaffolding is an automatic code generation framework for ASP.NET web applications. Scaffolding reduces the time taken to develop a controller, view etc. in MVC framework. You can develop a customized scaffolding template using T4 templates as per your architecture and coding standard.

Adding Controller

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

Adding Controller

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

Adding Controller

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

Example: Controller

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MVC\_BasicTutorials.Controllers

{

public class StudentController : Controller

{

// GET: Student

public ActionResult Index()

{

return View();

}

}

}

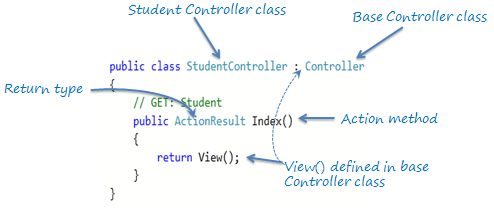
# Action method

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

Action Method

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 paramter 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.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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 ActioName("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 GetStudnet(int id)

{

return studentList.Where(s => s.StudentId == id).FirstOrDefault();

}

}

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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.

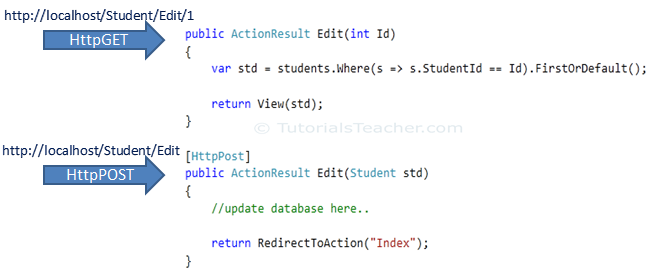
# ActionVerbs

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.

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. |

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");

}

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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.

# Model in ASP.NET MVC

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

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

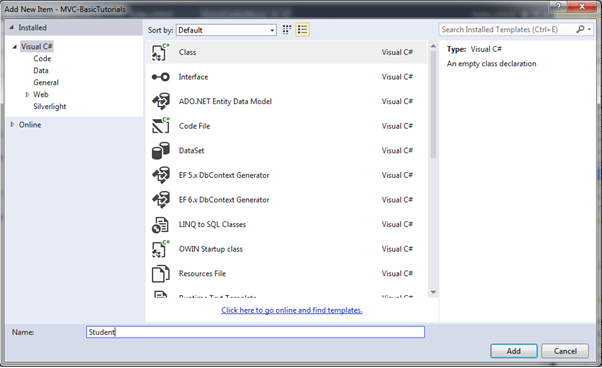
Model class holds data in public properties. All the Model classes reside in the Model folder in MVC folder structure.

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

## Adding a Model

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

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

Create Model Class

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

Example: Model class

namespace MVC\_BasicTutorials.Models

{

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

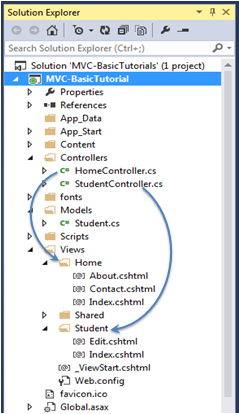
}

# View in ASP.NET MVC

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

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

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

View folders for Controllers

## Razor View Engine

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

ASP.NET MVC supports following types of view files:

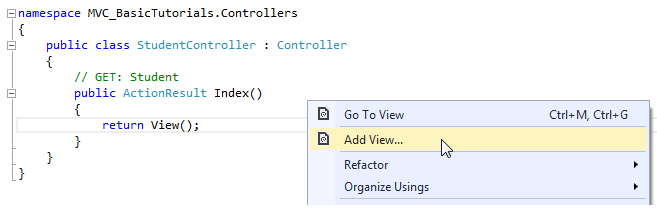
| **View file extension** | **Description** |
| --- | --- |
| .cshtml | C# Razor view. Supports C# with html tags. |
| .vbhtml | Visual Basic Razor view. Supports Visual Basic with html tags. |
| .aspx | ASP.Net web form |
| .ascx | ASP.NET web control |

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

## Create New View

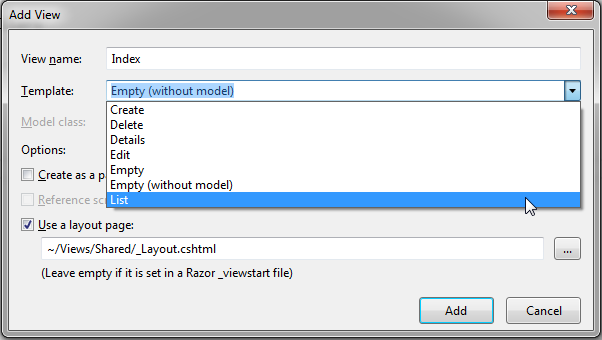
We have already created StudentController and Student model in the previous section. Now, let's create a Student view and understand how to use model into view.

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

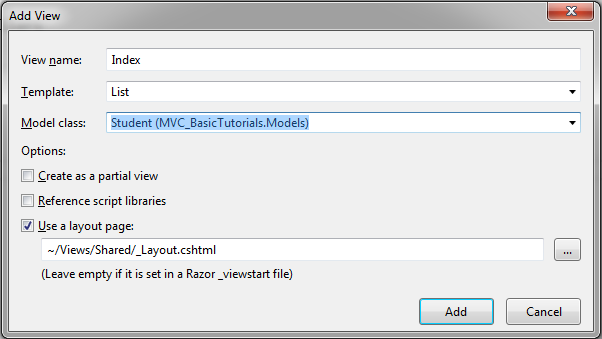
Create a View

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

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

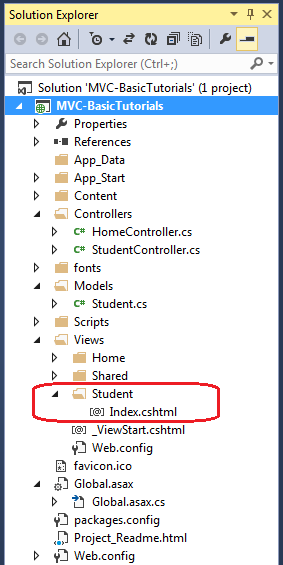
View

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

View

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

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

View

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

Views\Student\Index.cshtml:

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

@{

ViewBag.Title = "Index";

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

}

<h2>Index</h2>

<p>

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

</p>

<table class="table">

<tr>

<th>

@Html.DisplayNameFor(model => model.StudentName)

</th>

<th>

@Html.DisplayNameFor(model => model.Age)

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.StudentName)

</td>

<td>

@Html.DisplayFor(modelItem => item.Age)

</td>

<td>

@Html.ActionLink("Edit", "Edit", new { id=item.StudentId }) |

@Html.ActionLink("Details", "Details", new { id=item.StudentId }) |

@Html.ActionLink("Delete", "Delete", new { id = item.StudentId })

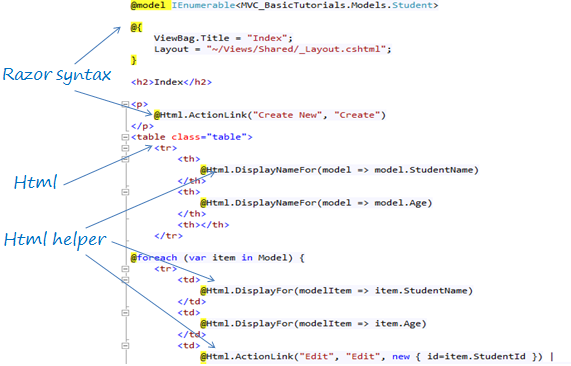
</td>

</tr>

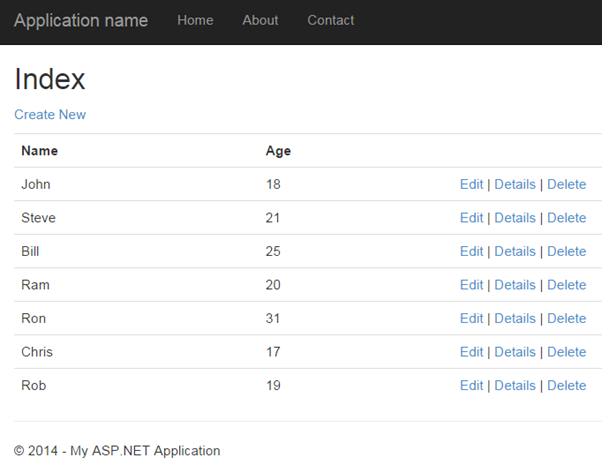
}

</table>

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

Index.cshtml

The above Index view would look like below.



# 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);

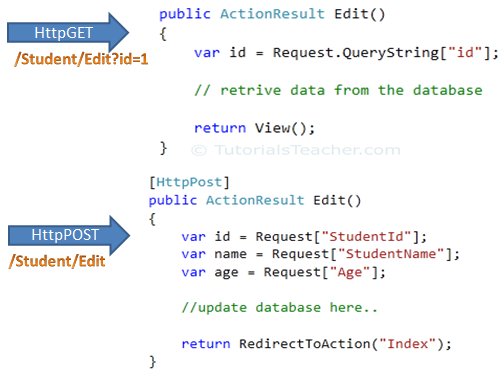
}

}

# Model Binding

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

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

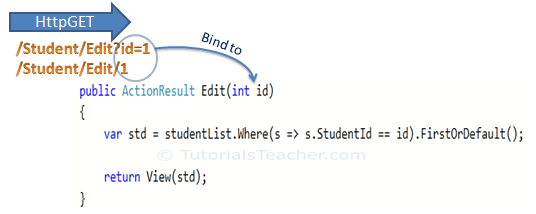
Accessing Request Data

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

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

## Binding to Primitive type

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



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;

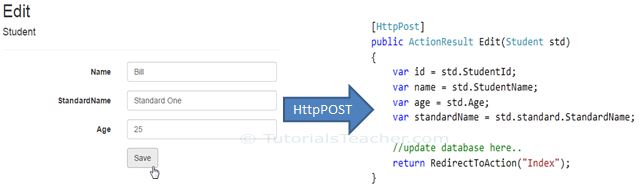
var standardName = std.standard.StandardName;

//update database here..

return RedirectToAction("Index");

}

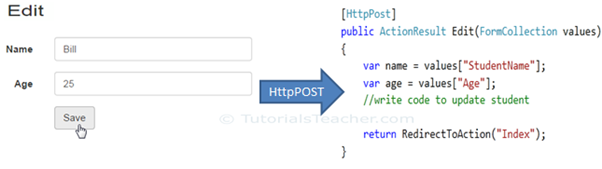
So now, MVC framework will automatically maps Form collection values to Student type parameter when the form submits http POST request to Edit action method as shown below.

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");

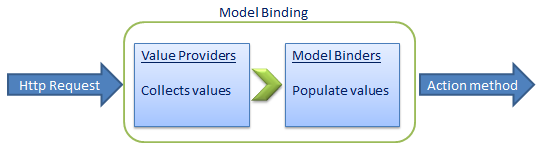
}

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.

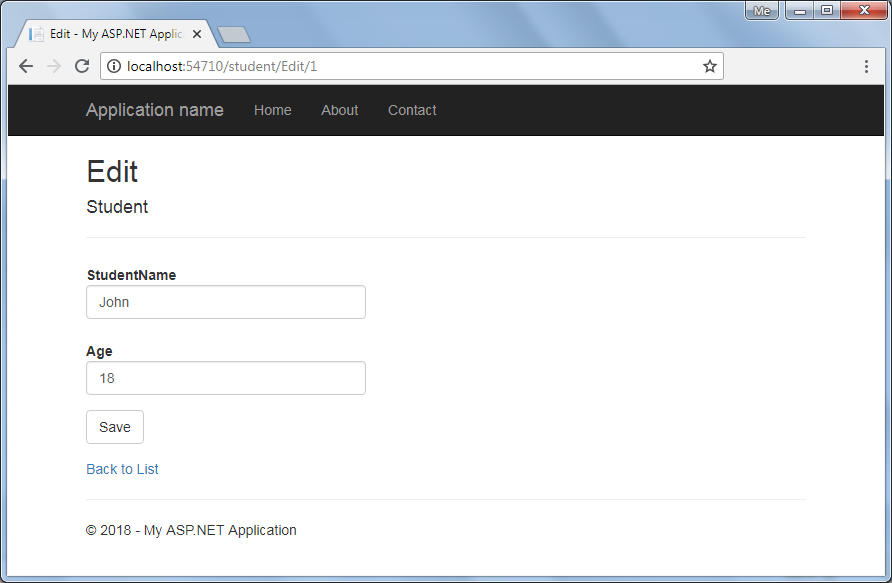
Model Binding in MVC

Default value provider collection evaluates values from the following sources:

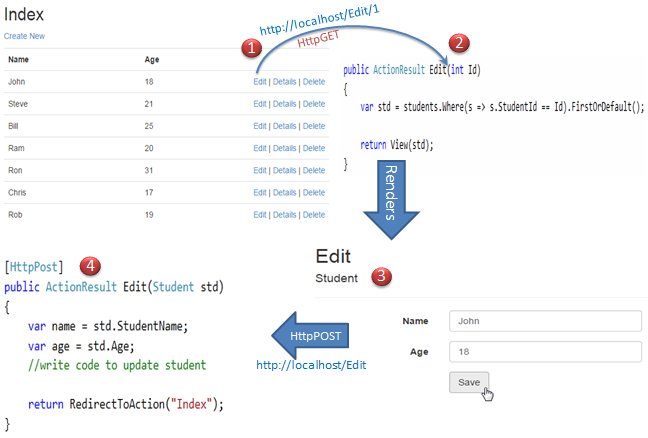
1. Previously bound action parameters, when the action is a child action
2. Form fields (Request.Form)
3. The property values in the JSON Request body (Request.InputStream), but only when the request is an AJAX request
4. Route data (RouteData.Values)
5. Querystring parameters (Request.QueryString)
6.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

# 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.

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; }

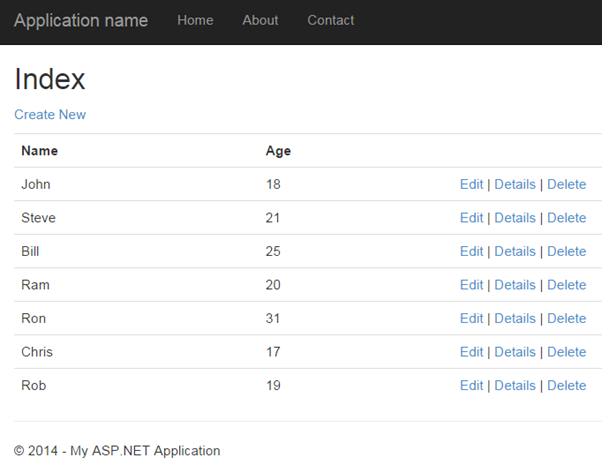
[Display( Name="Name")]

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 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 underlaying 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);

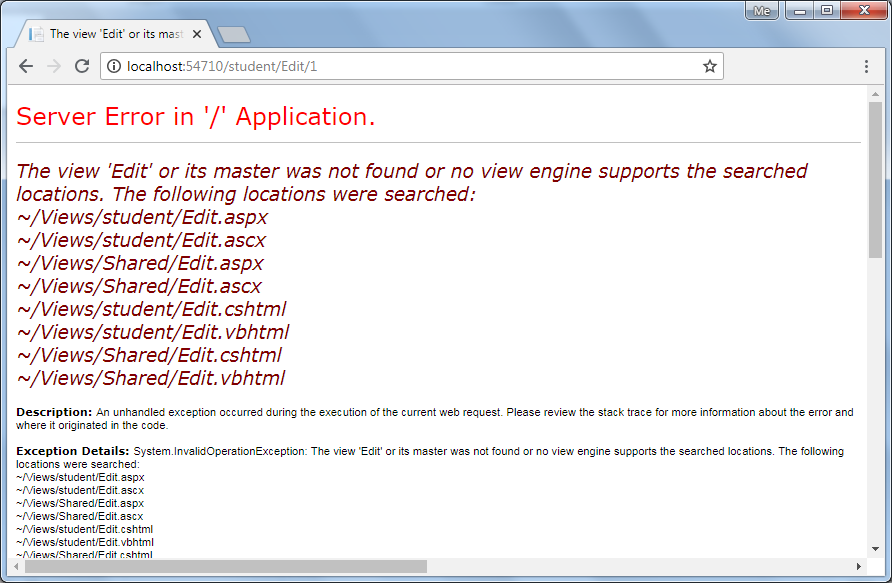
}

}

}

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.

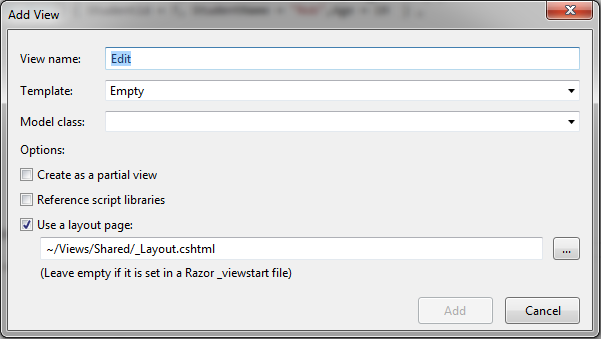
Now, if you click on the Edit link from Index view then you will get following error.

Edit View Error

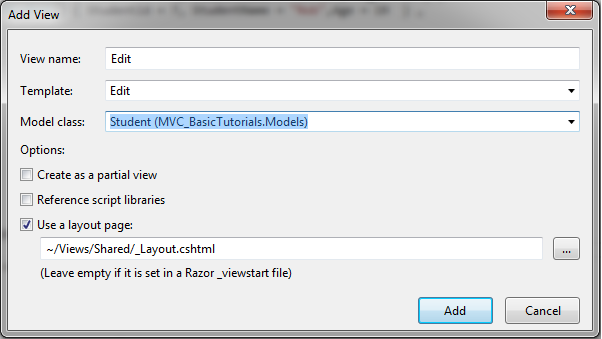
The above error occurred because we have not created an Edit view yet. By default, MVC framework will look for Edit.cshtml or Edit.vbhtml or Edit.aspx or Edit.ascx file in View -> Student or Shared folder.

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>

}

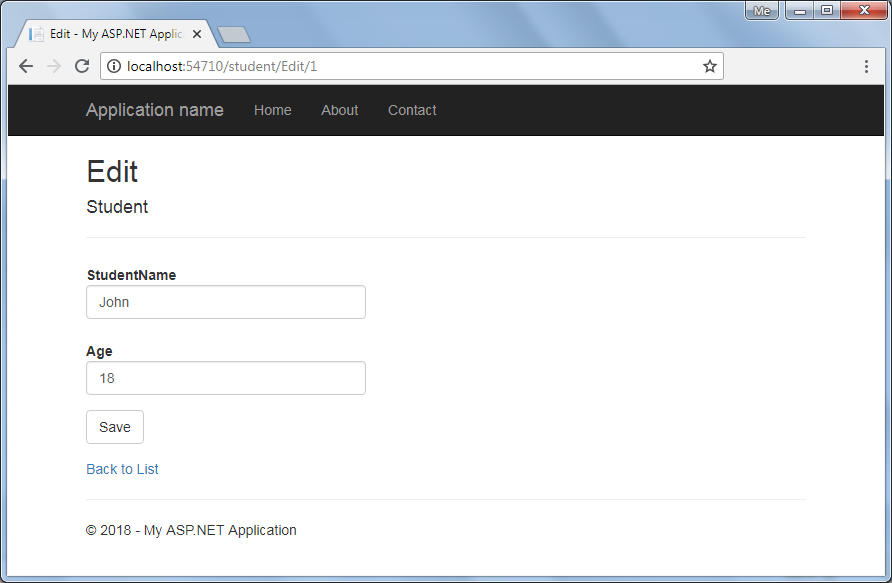
<div>

@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.

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:

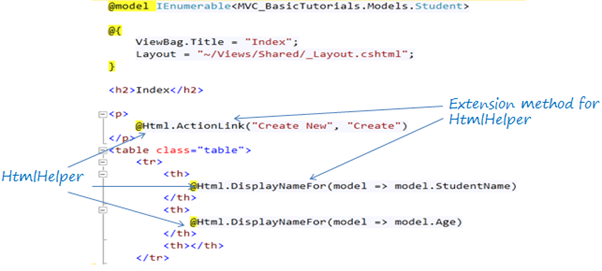
Hello World!

# HTML Helpers

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

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

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

HTML Helpers

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

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

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

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

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

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

# HtmlHelper - TextBox

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.

Example: Student Model

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)

# HtmlHelper - TextArea

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.

Example: Student Model

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.

### TextArea() method Signature

MvcHtmlString Html.TextArea(string name, string value, object htmlAttributes)

TextArea method has many overloads. Please visit MSDN to know all the overloads of TextArea method.

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

Example: Html.TextArea() in Razor View

@model Student

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

# HtmlHelper - CheckBox

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

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.

Example: Student Model

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; }

}

## 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)

Please visit MSDN to know all the overloads of CheckBox() method.

Example: Html.CheckBox() in Razor View

@Html.CheckBox("isNewlyEnrolled", true)

# HtmlHelper - RadioButton

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

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.

Example: Student Model

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()

The Html.RadioButton() method creates an radio button element with a specified name, isChecked boolean and html attributes.

### RadioButton() method Signature

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

Please visit MSDN to know all the overloads of RadioButton() method.

Example: Html.RadioButton() in Razor View

Male: @Html.RadioButton("Gender","Male")

Female: @Html.RadioButton("Gender","Female")

# HtmlHelper - DropDownList

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

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.

Example: Student Model

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<SelectLestItem> selectList, string optionLabel, object htmlAttributes)*

Please visit MSDN to know all the overloads of DropDownList() method.

Example: Html.DropDownList() in Razor View

@using MyMVCApp.Models

@model Student

@Html.DropDownList("StudentGender",

new SelectList(Enum.GetValues(typeof(Gender))),

"Select Gender",

new { @class = "form-control" })

## 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<SelectLestItem> selectList, string optionLabel, object htmlAttributes)*

Visit MSDN to know all the overloads of DropDownListFor().

The following example creates dropdown list for the above Gender enum.

Example: Html.DropDownListFor() in Razor View

@using MyMVCApp.Models

@model Student

@Html.DropDownListFor(m => m.StudentGender,

new SelectList(Enum.GetValues(typeof(Gender))),

"Select Gender")

# HtmlHelper - Hidden field

Learn how to generate hidden field using HtmlHelper in razor view in this section.

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.

Example: Student Model

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)*

Hidden() method has many overloads. Please visit MSDN to know all the overloads of Hidden() method.

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.

Example: Html.Hidden() in Razor View

@model Student

@Html.Hidden("StudentId")

# HtmlHelper - Password

Learn how to generate Password field using HtmlHelper in razor view in this section.

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.

Example: Student Model

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 a input password element with specified name, value and html attributes.

### Password() method signature

MvcHtmlString Html.Password(string name, object value, object htmlAttributes)

## 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 Html.PasswordFor(Expression<Func<dynamic,TProperty>> expression, object htmlAttributes)*

Visit MSDN to know all the overloads of PasswordFor() method.

Example: PasswordFor() in Razor View

@model Student

@Html.PasswordFor(m => m.Password)

# HtmlHelper - Display HTML String

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.

Example: Student Model

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

Example: Html.Display() in Razor View

@Html.Display("StudentName")

Html Result:

"Steve"

## DisplayFor

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

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

Visit MSDN to know all the overloads of DisplayFor() method.

Example: DisplayFor() in Razor View

@model Student

@Html.DisplayFor(m => m.StudentName)

# HtmlHelper - Label

Learn how to create <label> element using HtmlHelper in razor view in this section.

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.

Example: Student Model

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)

Label() method has many overloads. Please visit MSDN to know all the overloads of Label() method

Example: Html.Label() in Razor View

@Html.Label("StudentName")

Html Result:

<label for="StudentName">Name</label>

In the above example, we have specified a StudentName property as a string. So, it will create <label> element that display *Name*.

You can specify another label text instead of property name as shown below.

Example: Html.Label() in Razor View

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

Html Result:

<label for="StudentName">Student-Name</label>

## 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)

Visit MSDN to know all the overloads of LabelFor() method.

Example: LabelFor() in Razor View

@model Student

@Html.LabelFor(m => m.StudentName)

# HtmlHelper - Editor

We have seen different HtmlHelper methods used to generated 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.

The following table list the html element created for each data type by Editor() or EditorFor() method.

| 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" > |

We will use the following model class with Editor and EditorFor method.

Example: Student Model

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 creats a html element based on the datatype of the specified property.

Editor() signature: MvcHtmlString Editor(string propertyname)

Visit MSDN to know all the overloads of Editor() method

Consider the following example to understand the Editor() method.

Example: Editor() in Razor view

StudentId: @Html.Editor("StudentId")

Student Name: @Html.Editor("StudentName")

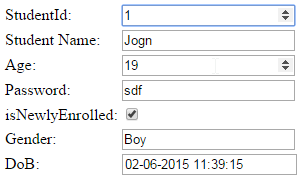
Age: @Html.Editor("Age")

Password: @Html.Editor("Password")

isNewlyEnrolled: @Html.Editor("isNewlyEnrolled")

Gender: @Html.Editor("Gender")

DoB: @Html.Editor("DoB")

Output of Editor() and EditorFor() method

In the above example, we have specified property names of Student model as a string. So, Editor() method created the appropriate input elements based on the datatype as shown in the above figure.

## EditorFor

EditorFor() method is a strongly typed method. It requires the lambda expression to specify a property of the model object.

EditorFor() signature: MvcHtmlString EditorFor(<Expression<Func<TModel,TValue>> expression)

Visit MSDN to know all the overloads of EditorFor() method

Example: EditorFor() in Razor view

StudentId: @Html.EditorFor(m => m.StudentId)

Student Name: @Html.EditorFor(m => m.StudentName)

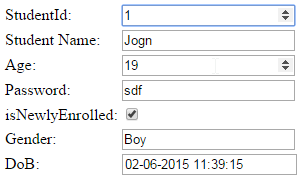
Age: @Html.EditorFor(m => m.Age)

Password: @Html.EditorFor(m => m.Password)

isNewlyEnrolled: @Html.EditorFor(m => m.isNewlyEnrolled)

Gender: @Html.EditorFor(m => m.Gender)

DoB: @Html.EditorFor(m => m.DoB)

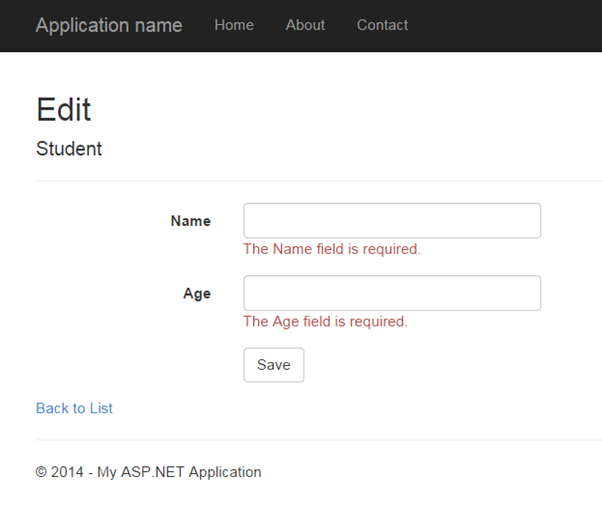
Output of Editor() and EditorFor() method

In the above example of EditorFor() method, we have specified the property name using the lambda expression. The result would be the same as the Editor() method as shown in the above figure.

# Implement Data Validation in MVC

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

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

Validation

## DataAnnotations

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

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

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

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

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

Example: Apply DataAnnotation Attributes

public class Student

{

public int StudentId { get; set; }

[Required]

public string StudentName { get; set; }

[Range(5,50)]

public int Age { get; set; }

}

In the above example, we have applied a ***Required*** attribute to the StudentName property. So now, the MVC framework will automatically display the default error message, if the user tries to save the Edit form without entering the Student Name. In the same way, the ***Range*** attribute is applied with a min and max value to the Age property. This will validate and display an error message if the user has either not entered Age or entered an age less than 5 or more than 50.

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

Example: Edit Action methods:

using MVC\_BasicTutorials.Models;

namespace MVC\_BasicTutorials.Controllers

{

public class StudentController : Controller

{

public ActionResult Edit(int id)

{

var std = studentList.Where(s => s.StudentId == StudentId)

.FirstOrDefault();

return View(std);

}

[HttpPost]

public ActionResult Edit(Student std)

{

if (ModelState.IsValid) {

//write code to update student

return RedirectToAction("Index");

}

return View(std);

}

}

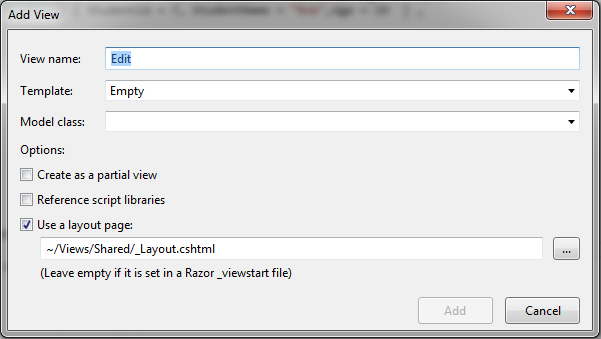
}

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

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

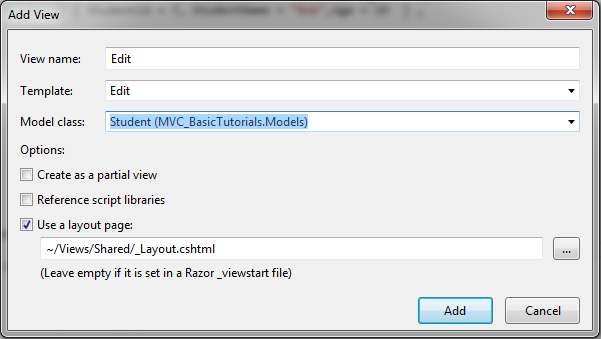
**Step 3:** Now, create an Edit view for Student.

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

Create Edit View

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

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

Create Edit View

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

Edit.cshtml:

@model MVC\_BasicTutorials.Models.Student

@{

ViewBag.Title = "Edit";

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

}

<h2>Edit</h2>

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<h4>Student</h4>

<hr />

**@Html.ValidationSummary(true, "", new { @class = "text-danger" })**

@Html.HiddenFor(model => model.StudentId)

<div class="form-group">

@Html.LabelFor(model => model.StudentName, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.StudentName, new { htmlAttributes = new { @class = "form-control" } })

**@Html.ValidationMessageFor(model => model.StudentName, "", new { @class = "text-danger" })**

</div>

</div>

<div class="form-group">

@Html.LabelFor(model => model.Age, htmlAttributes: new { @class = "control-label col-md-2" })

<div class="col-md-10">

@Html.EditorFor(model => model.Age, new { htmlAttributes = new { @class = "form-control" } })

**@Html.ValidationMessageFor(model => model.Age, "", new { @class = "text-danger" })**

</div>

</div>

<div class="form-group">

<div class="col-md-offset-2 col-md-10">

<input type="submit" value="Save" class="btn btn-default" />

</div>

</div>

</div>

}

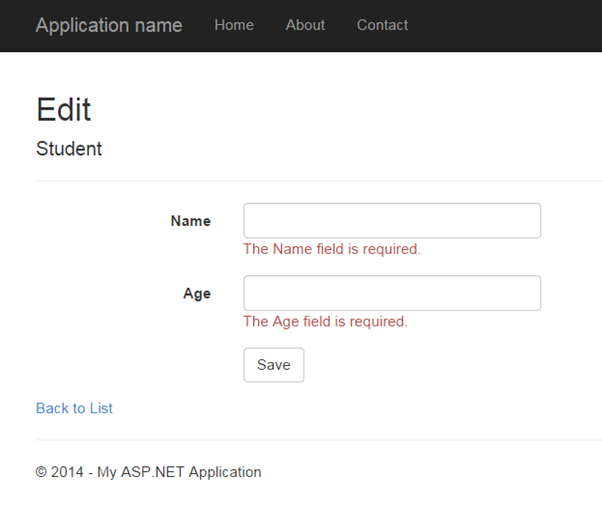
<div>

@Html.ActionLink("Back to List", "Index")

</div>

As you can see in the above Edit.cshtml, it calls Html Helper method **ValidationMessageFor** for every field and **ValidationSummary** method at the top. ValidationMessageFor is responsible to display error message for the specified field. ValidationSummary displays a list of all the error messages at once.

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

Validation

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

Learn how to implement client side validation in ASP.NET MVC.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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.

# ASP.NET MVC: ValidationMessage

You have learned how to implement validation in a view in the presious section. Here, we will see the HtmlHelper extension method ValidtionMessage 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)

Visit MSDN to know all the overloads of ValidationMessage() method.

Example: ValidationMessage

@model Student

@Html.Editor("StudentName") <br />

@Html.ValidationMessage("StudentName", "", new { @class = "text-danger" })

In the above example, 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 and 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 specifed property in the model class. The following is a Student model class where the DataAnnotations attribute "Required" is applied to the StudentName property.

Example: Student Model

public class Student

{

public int StudentId { get; set; }

[Required]

public string StudentName { get; set; }

public int Age { get; set; }

}

Example: Custom error message in the Model

public class Student

{

public int StudentId { get; set; }

[Required(ErrorMessage="Please enter student name.")]

public string StudentName { get; set; }

public int Age { get; set; }

}

Also, you can specify a message as a second parameter in the ValidationMessage() method as shown below.

Example: Custom error message

@model Student

@Html.Editor("StudentName") <br />

@Html.ValidationMessage("StudentName", "Please enter student name.", new { @class = "text-danger" })

# ASP.NET MVC: ValidationMessageFor

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

## ValidationMessageFor() Signature

*MvcHtmlString ValidateMessage(Expression<Func<dynamic,TProperty>> expression, string validationMessage, object htmlAttributes)*

Visit MSDN to know all the overloads of ValidationMessageFor() method.

Consider the following ValidationMessageFor() example.

Example: ValidationMessageFor

@model Student

@Html.EditorFor(m => m.StudentName) <br />

@Html.ValidationMessageFor(m => m.StudentName, "", new { @class = "text-danger" })

In the above example, the first parameter in ValidationMessageFor method is a lambda expression to specify a property for which we want to show the error message. The second parameter is for custom error message and the third parameter is for html attributes like css, style etc.

The ValidationMessageFor() method will only display an error if you have configured DataAnnotations attribute to the specifed property in the model class. The following example is a Student model class where the DataAnnotations attribute "Required" is applied to the StudentName property.

Example: Student Model

public class Student

{

public int StudentId { get; set; }

[Required]

public string StudentName { get; set; }

public int Age { get; set; }

}

The above code will generate the following html.

Html Result:

<input id="StudentName"

name="StudentName"

type="text"

value="" />

<span class="field-validation-valid text-danger"

data-valmsg-for="StudentName"

data-valmsg-replace="true">

</span>

Now, when the user submits a form without entering the StudentName then ASP.NET MVC uses the data- attribute of Html5 for the validation and the default validation message will be injected when validation error occurs, as shown below.

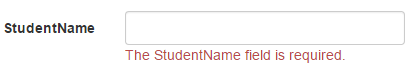
Html with Validation message:

<span class="field-validation-error text-danger"

data-valmsg-for="StudentName"

data-valmsg-replace="true">The StudentName field is required.</span>

The error message will appear as the image shown below.

Output of ValidationMessageFor() method

## Custom Error Message

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

Use the ErrorMessage parameter of the DataAnnotation attributes to provide your own custom error message as shown below.

Example: Custom error message in the Model

public class Student

{

public int StudentId { get; set; }

[Required(ErrorMessage="Please enter student name.")]

public string StudentName { get; set; }

public int Age { get; set; }

}

Also, you can specify a message as a second parameter in the ValidationMessage() method as shown below.

Example: Custom error message

@model Student

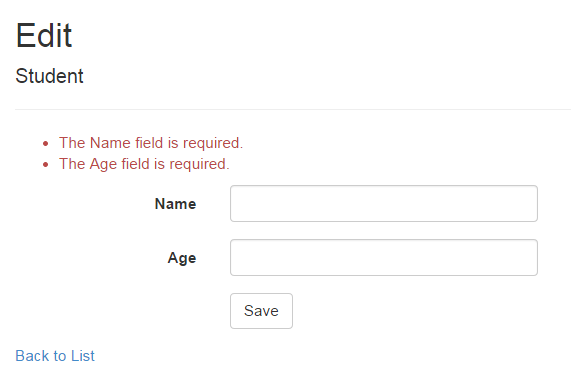
@Html.Editor("StudentName") <br />

@Html.ValidationMessageFor(m => m.StudentName, "Please enter student name.", new { @class = "text-danger" })

# ASP.NET MVC: ValidationSummary

The ValidationSummary helper method generates an unordered list (ul element) of validation messages that are in the ModelStateDictionary object.

The ValidationSummary can be used to display all the error messages for all the fields. It can also be used to display custom error messages. The following figure shows how ValidationSummary displays the error messages.

ValidationSummary

## ValidationSummary() Signature

*MvcHtmlString ValidateMessage(bool excludePropertyErrors, string message, object htmlAttributes)*

Visit MSDN to know all the overloads of ValidationMessage() method.

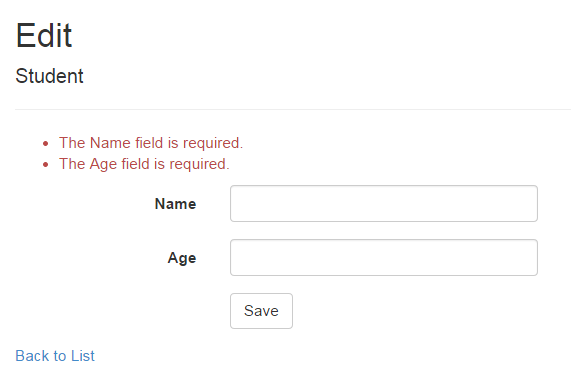
## Display Field Level Error Messages using ValidationSummary

By default, ValidationSummary filters out field level error messages. If you want to display field level error messages as a summary then specify excludePropertyErrors = false.

Example: ValidationSummary to display field errors

@Html.ValidationSummary(false, "", new { @class = "text-danger" })

So now, the following Edit view will display error messages as a summary at the top. Please make sure that you don't have a ValidationMessageFor method for each of the fields.

Error Message using ValidationSummary

## Display Custom Error Messages

You can also display a custom error message using ValidationSummary. For example, we want to display a message if Student Name already exists in the database.

To display a custom error message, first of all, you need to add custom errors into the ModelState in the appropriate action method.

Example: Add error in ModelState

if (ModelState.IsValid) {

//check whether name is already exists in the database or not

bool nameAlreadyExists = \* check database \*

if(nameAlreadyExists)

{

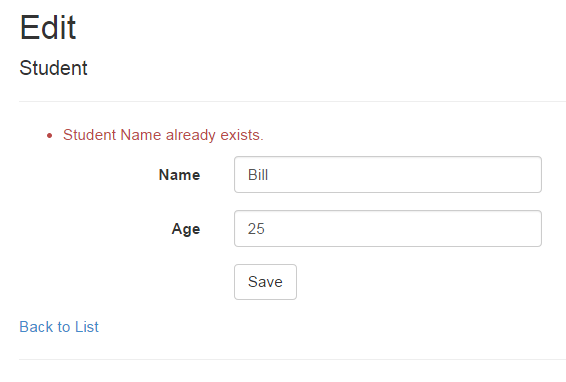
ModelState.AddModelError(string.Empty, "Student Name already exists.");

return View(std);

}

}

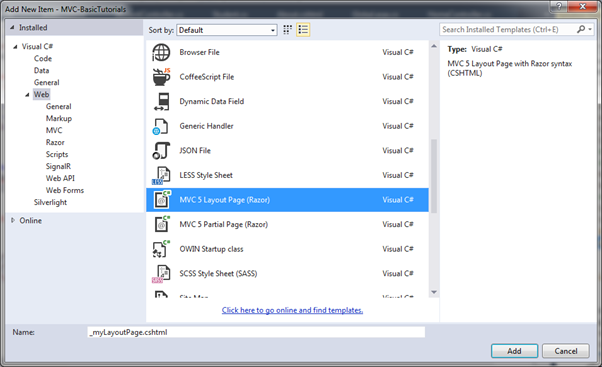
As you can see in the above code, we have added custom error messages using the ModelState.AddModelError method. The ValidationSummary method will automatically display all the error messages added into ModelState.

Display error message using ValidationSymmary

Create Layout View

To create a new layout view in Visual Studio, right click on shared folder -> select Add -> click on **New Item..**

In the Add New Item dialogue box, select MVC 5 Layout Page (Razor) and give the layout page name as "\_myLayoutPage.cshtml" and click **Add**.

Rendering Methods

You will see \_myLayoutPage.cshtml as shown below.

\_myLayoutPage.cshtml:

<!DOCTYPE html>

<html>

<head>

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

<title>@ViewBag.Title</title>

</head>

<body>

<div>

@RenderBody()

</div>

</body>

</html>

Now, add the <footer> tag with the RenderSection("footer",true) method alongwith some styling as shown below. Please notice that we made this section as required. This means any view that uses \_myLayoutPage as its layout view must include a footer section.

Example: Adding RenderSection

<!DOCTYPE html>

<html>

<head>

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

<title>@ViewBag.Title</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

<div>

@RenderBody()

</div>

<footer class="panel-footer">

@RenderSection("footer", true)

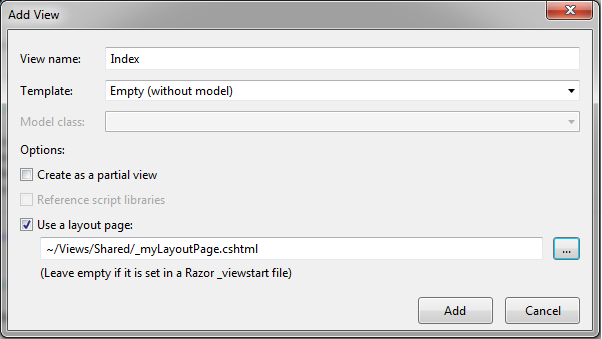
</footer>

</body>

</html>

Now, let's use this \_myLayoutPage.cshtml with the Index view of HomeController.

You can add an empty Index view by right clicking on Index action method of HomeController and select Add View. Select Empty as a scaffolding template and \_myLayoutPage.cshtml as layout view and click Add.

Add Index View

This will create Index.cshtml as shown below.

Index view:

@{

ViewBag.Title = "Home Page";

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

}

<h2>Index</h2>

So now, we have created Index view that uses our \_myLayoutPage.cshtml as a layout view. We will now add footer section along with some styling because \_myLayoutPage requires footer section.

Index view:

@{

ViewBag.Title = "Home Page";

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

}

<div class="jumbotron">

<h2>Index</h2>

</div>

<div class="row">

<div class="col-md-4">

<p>This is body.</p>

</div>

@section footer{

<p class="lead">

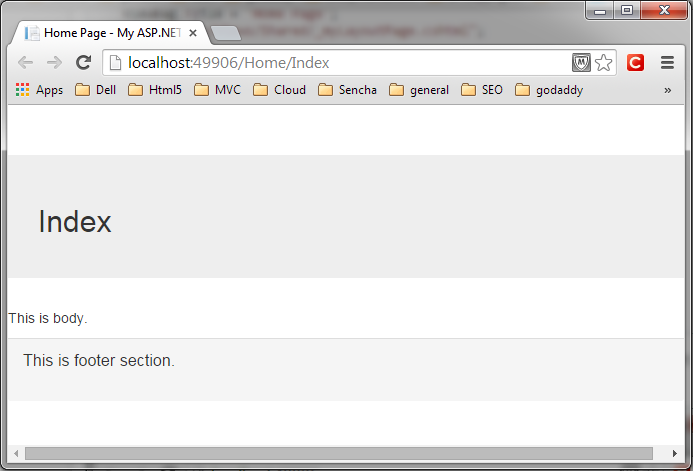
This is footer section.

</p>

}

</div>

Now, run the application and you will see Index view will contain body and footer part as shown below.

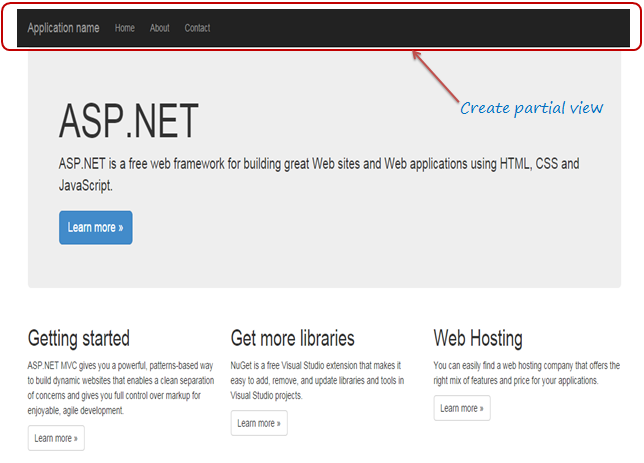


# Partial View

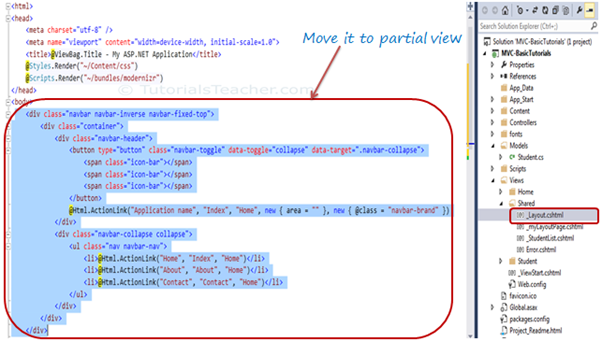
In this section you will learn about partial views in ASP.NET MVC.

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

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

Partial View

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

Partial Views

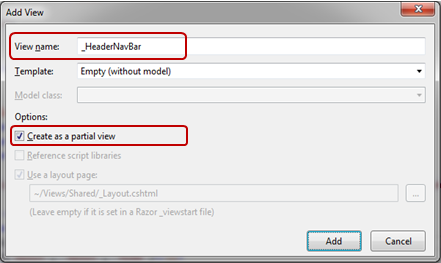
## Create a New Partial View

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

 Note:

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

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

Partial Views

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

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

Example: Partial View \_HeaderNavBar.cshtml

<div class="navbar navbar-inverse navbar-fixed-top">

<div class="container">

<div class="navbar-header">

<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

@Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav">

<li>@Html.ActionLink("Home", "Index", "Home")</li>

<li>@Html.ActionLink("About", "About", "Home")</li>

<li>@Html.ActionLink("Contact", "Contact", "Home")</li>

</ul>

</div>

</div>

</div>

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

## Render Partial View

You can render the partial view in the parent view using html helper methods: Partial() or RenderPartial() or RenderAction(). Each method serves different purposes. Let's have an overview of each method and then see how to render partial view using these methods.

### Html.Partial()

@Html.Partial() helper method renders the specified partial view. It accept partial view name as a string parameter and returns MvcHtmlString. It returns html string so you have a chance of modifing the html before rendering.

The following table lists overloads of the Partial helper method:

| Helper Method | Description |
| --- | --- |
| *MvcHtmlString* Html.Partial(string partialViewName) | Renders the given partial view content in the referred view. |
| *MvcHtmlString* Html.Partial(string partialViewName,object model) | Renders the partial view content in the referred view. Model parameter passes the model object to the partial view. |
| *MvcHtmlString* Html.Partial(string partialViewName, ViewDataDictionary viewData) | Renders the partial view content in the referred view. View data parameter passes view data dictionary to the partial view. |
| *MvcHtmlString* Html.Partial(string partialViewName,object model, ViewDataDictionary viewData) | Renders the partial view content in the referred view. Model parameter passes the model object and View data passes view data dictionary to the partial view. |

### Html.RenderPartial()

The RenderPartial helper method is same as the Partial method except that it returns void and writes resulted html of a specified partial view into a http response stream directly.

| Helper method | Description |
| --- | --- |
| RenderPartial(String partialViewName) | Renders the specified partial view |
| RenderPartial(String partialViewName, Object model) | Renders the specified partial view and set the specified model object |
| RenderPartial(String partialViewName, ViewDataDictionary viewData) | Renders the specified partial view, replacing its ViewData property with the specified ViewDataDictionary object. |
| RenderPartial(String partialViewName, Object model, ViewDataDictionary viewData) | Renders the specified partial view, replacing the partial view's ViewData property with the specified ViewDataDictionary object and set the specified model object |

### Html.RenderAction()

The RenderAction helper method invokes a specified controller and action and renders the result as a partial view. The specified Action method should return PartialViewResult using the Partial() method.

| **Name** | **Description** |
| --- | --- |
| RenderAction(String actionName) | Invokes the specified child action method and renders the result in the parent view. |
| RenderAction(String actionName, Object routeValue) | Invokes the specified child action method using the specified parameters and renders the result inline in the parent view. |
| RenderAction(String actionName, String controllerName) | Invokes the specified child action method using the specified controller name and renders the result inline in the parent view. |
| RenderAction(String actionName, RouteValueDictionary routeValues) | Invokes the specified child action method using the specified parameters and renders the result inline in the parent view. |
| RenderAction(String actionName, String controllerName, Object routeValue) | Invokes the specified child action method using the specified parameters and controller name and renders the result inline in the parent view. |
| RenderAction(String actionName, String controllerName, RouteValueDictionary routeValues) | Invokes the specified child action method using the specified parameters and controller name and renders the result inline in the parent view. |

So now, we can use any of the above rending methods to render the \_HeaderNavBar partial view into \_Layout.cshtml. The following layout view renders partial view using the RenderPartial() method.

Example: Html.RenderPartial()

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - My ASP.NET Application</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

@{

**Html.RenderPartial("\_HeaderNavBar");**

}

<div class="container body-content">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

</footer>

</div>

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

@RenderSection("scripts", required: false)

</body>

</html>

 Note:

RenderPartial returns void, so a semicolon is required at the end and so it must be enclosed in the braces.

The following layout view uses the Partial method to render partial view\_HeaderNavBar.cshtml.

Example: Html.Partial()

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title - My ASP.NET Application</title>

@Styles.Render("~/Content/css")

@Scripts.Render("~/bundles/modernizr")

</head>

<body>

@Html.Partial("\_HeaderNavBar")

<div class="container body-content">

@RenderBody()

<hr />

<footer>

<p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

</footer>

</div>

@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

@RenderSection("scripts", required: false)

</body>

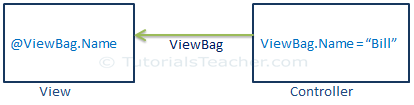
</html>

# ASP.NET MVC - ViewBag

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.

The following figure illustrates the ViewBag.

ViewBag

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.)

ou can assign any number of properties and values to ViewBag. If you assign the same property name multiple times to ViewBag, then it will only consider last value assigned to the property.

 Note:

ViewBag only transfers data from controller to view, not visa-versa. ViewBag values will be null if redirection occurs.

The following example demonstrates how to transfer data from controller to view using ViewBag.

Example: Set ViewBag in Action method

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();

}

}

}

In the above example, we want to display the total number of students in a view for the demo. So, we have attached the TotalStudents property to the ViewBag and assigned the student count using studentList.Count().

Now, in the Index.cshtml view, you can access ViewBag.TotalStudents property and display all the student info as shown below.

Example: Acess ViewBag in a View

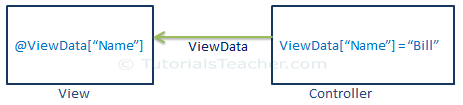
<label>Total Students:</label> @ViewBag.TotalStudents

ASP.NET MVC - 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.

ViewData

 Note:

ViewData only transfers data from controller to view, not vice-versa. It is valid only during the current request.

The following example demonstrates how to transfer data from controller to view using ViewData.

Example: ViewData in Action method

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();

}

In the above example, we have added a student list with the key "students" in the ViewData dictionary. So now, the student list can be accessed in a view as shown below.

Example: Access ViewData in a Razor 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.

Example: Add KeyValuePair in ViewData

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();

}

ViewData and ViewBag both use the same dictionary internally. So you cannot have ViewData Key matches with the property name of ViewBag, otherwise it will throw a runtime exception.

Example: ViewBag and ViewData

public ActionResult Index()

{

ViewBag.Id = 1;

ViewData.Add("Id", 1); // throw runtime exception as it already has "Id" key

ViewData.Add(new KeyValuePair<string, object>("Name", "Bill"));

ViewData.Add(new KeyValuePair<string, object>("Age", 20));

return View();

}

https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

1. ViewData transfers data from the Controller to View, not vice-versa.
2. ViewData is derived from ViewDataDictionary which is a dictionary type.
3. ViewData's life only lasts during current http request. ViewData values will be cleared if redirection occurs.
4. ViewData value must be type cast before use.
5. ViewBag internally inserts data into ViewData dictionary. So the key of ViewData and property of ViewBag must **NOT** match.

ASP.NET MVC - 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.

You can add a key-value pair in TempData as shown in the below example.

Example: TempData

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

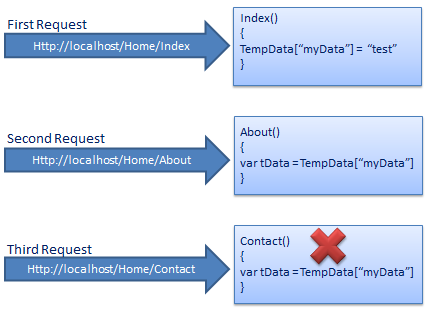
return View();

}

}

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

The following figure illustrates TempData.

TempData

https://www.tutorialsteacher.com/Content/images/tips.pngTempData internally uses session to store the data. So the data must be serialized if you decide you to switch away from the default Session-State Mode, and use State Server Mode or SQL Server Mode.

As you can see in the above example, we add test data in TempData in the first request and in the second subsequent request we access test data from TempData which we stored in the first request. However, 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.

Example: TempData.Keep()

public class HomeController : Controller

{

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();

}

}

https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

1. TempData can be used to store data between two consecutive requests. TempData values will be retained during redirection.
2. TemData is a TempDataDictionary type.
3. TempData internaly use Session to store the data. So think of it as a short lived session.
4. TempData value must be type cast before use. Check for null values to avoid runtime error.
5. TempData can be used to store only one time messages like error messages, validation messages.
6. Call TempData.Keep() to keep all the values of TempData in a third request.

# ScriptBundle in ASP.NET MVC

We have learned how bundling technique works in ASP.NET MVC. Here, we will learn how to create a bundle of multiple JavaScript files in one http request.

ASP.NET MVC API includes ScriptBundle class that does JavaScript minification and bundling.

Open App\_Start\BundleConfig.cs file in the MVC folders. The BundleConfig.cs file is created by MVC framework by default. You should write your all bundling code in the BundleConfig.RegisterBundles() method. (you can create your own custom class instead of using BundleConfig class, but it is recommended to follow standard practice.) The following code shows a portion of the RegisterBundles method.

Example: BundleConfig.RegisterBundle()

using System.Web;

using System.Web.Optimization;

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

// create an object of ScriptBundle and

// specify bundle name (as virtual path) as constructor parameter

ScriptBundle scriptBndl = new ScriptBundle("~/bundles/bootstrap");

//use Include() method to add all the script files with their paths

scriptBndl.Include(

"~/Scripts/bootstrap.js",

"~/Scripts/respond.js"

);

//Add the bundle into BundleCollection

bundles.Add(scriptBndl);

BundleTable.EnableOptimizations = true;

}

}

In the above example, we have created a bundle of two JavaScript files, bootstrap.js and respond.js using ScriptBundle for demo purposes.

1. First of all create an instance of ScriptBundle class by specifing the bundle name as a constructor parameter. This bundle name is a virtual path starting with ~/. You can give anything in virtual path but it's recommended to give a path that will be easy to identify as a bundle. Here, we have given "~/bundles/bootstrap" path, so that we can easily identify that this bundle includes bootstrap related files.
2. Use Include method to add one or more JS files into a bundle with its relative path after root path using ~ sign.
3. Final, add the bundle into BundleCollection instance, which is specified as a parameter in RegisterBundle() method.
4. Last, BundleTable.EnableOptimizations = true enables bundling and minification in debug mode. If you set it to false then it will not do bundling and minification.

You can also use IncludeDirectory method of bundle class to add all the files under particular directory as shown below.

ScriptBundle Example:

public static void RegisterBundles(BundleCollection bundles)

{

bundles.Add(new ScriptBundle("~/bundles/scripts").IncludeDirectory("~/Scripts/","\*.js",true));

}

Thus, you can create a bundle of JavaScript files using ScriptBundle. MVC framework invokes BundleConfig.RegisterBundle() method from the Application\_Start event in Global.asax.cs file, so that it can add all the bundles into BundleCollection at the starting of an application.

Example: Invoke RegisterBundle() in Application\_Start event

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

}

## Using Wildcards

Sometime third party script files includes versions in a name of script file. So it is not advisable to changes the code whenever you upgrade the version of script file. With the use of wildcards, you don't have to specify a version of a script file. It automatically includes files with the version available.

For example, Jquery files includes the version in a name. So you can use {version} wildcard to pickup a version based on available version.

Example: Wildcard with bundle

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

bundles.Add(new ScriptBundle("~/bundles/jquery")

.Include( "~/Scripts/jquery-{version}.js"));

}

}

Now, it will pick up jquery file added in a project. If you have included jquery-1.7.1.js then it will render this file and when you upgrade jquery file to jquery-1.10.2.js then it will automatically render 1.10 version file without changing or compiling code.

## Using CDN

You can also use Content Delivery Network to load script files. For example, you can load jquery library from CDN as shown below.

Example: Load files from CDN

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

var cdnPath = "http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1.min.js";

bundles.Add(new ScriptBundle("~/bundles/jquery", cdnPath)

.Include( "~/Scripts/jquery-{version}.js"));

}

}

In the above code, jquery will be requested from the CDN while in release mode and in the debug mode, jquery library will be loaded from a local source. Please note that you should have a fallback mechanism to deal with a CDN request failure.

Now, let's see how to use the bundle into a razor view.

## Include ScriptBundle in Razor View

We have create a script bundle above. Now, we will learn how to include bundle into razor view.

The script bundles can be included using static Scripts class. Use Scripts.Render() method to include specified script bundle at runtime.

Example: Scripts.Render()

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>@ViewBag.Title</title>

**@Scripts.Render("~/bundles/bootstrap")**

</head>

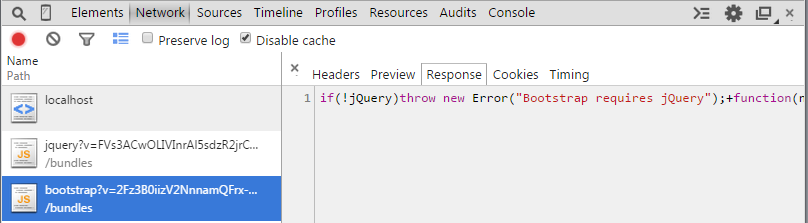
<body>

@\*html code removed for clarity \*@

</body>

</html>

Now, if you run the above example then you will find two script files is combined, minified and loaded in a single request. Please make sure that you have set debug = false in web.config <compilation debug="false" targetFramework="4.5"/>

Load Bundle in Browser

As you can see in the above figure that bootstrap bundle is loaded in a single request. It has also combined and minified two JS files for bootstrap.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

1. Bundling and Minification minimize static script or css files loading time therby minimize page loading time.
2. ScriptBundle does minification of JavaScript files.
3. Create script or css bundles in BundleConfig class included in App\_Start folder.
4. Use wildcard {version} to render available version files at runtime.
5. Use Scripts.Render("bundle name") method to include script bundle in a razor view.

# StyleBundle

You have learned how to create a bundle of JavaScript files in the previous section. Here, you will learn how to create a bundle of style sheet files (CSS).

ASP.NET MVC API includes StyleBundle class that does CSS minification and bundling. StyleBundle is also derived from a Bundle class so it supports same methods as ScriptBundle.

As mentioned in the previous section, you should create bundles of script and css files in the RegisterBundles() method of BundleConfig class contained in App\_Start -> BundleConfig.cs file.

The following code shows a portion of the RegisterBundles() method.

Use ScriptsInclude or IncludeDerictory method to add css files into bundle as shown below:

Example: StyleBundle

public class BundleConfig

{

public static void RegisterBundles(BundleCollection bundles)

{

bundles.Add(new StyleBundle("~/bundles/css").Include(

"~/Content/bootstrap.css",

"~/Content/site.css"

));

// add ScriptBundle here..

}

}

As you can see in the above example, we have created StyleBundle instance with bundle name as virtual path. The bundle name (virtual path) must start with ~/. Use Include() or IncludeDirectory() method with css file names as a string.

You can use wildcard and CDN path the same way as ScriptBundle as shown in the previous section.

## Include Style Bundle in Razor View

You can use StyleBundle in a layout view and render bunch of css files in a single request using static Styles class. Styles is a helper class to render css bundles.

Example: Include Style Bundle in View

<!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("~/bundles/css")**

</head>

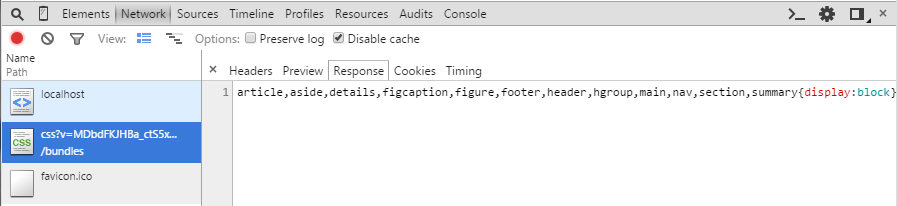
<body>

@\*html code removed for clarity \*@

</body>

</html>

As shown in the above example, use Styles.Render() method to include specified css bundle at runtime. Open developer tool of the browser and check that it has minified and loaded css files as shown below.

Load Bundle in Browser

Learn how to set image path in StyleBundle.

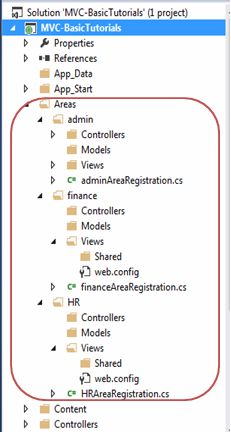
https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

1. Bundling and Minification minimize static script or css files loading time therby minimize page loading time.
2. MVC framework provides ScriptBundle, StyleBundle and DynamicFolderBundle classes.
3. StyleBundle does minification of CSS files.
4. Create script or css bundles in the BundleConfig class included in App\_Start folder.
5. Use wildcard {version} to render available version files at runtime.
6. Use Styles.Render("bundle name") method to include style bundles in a razor view.

# Area

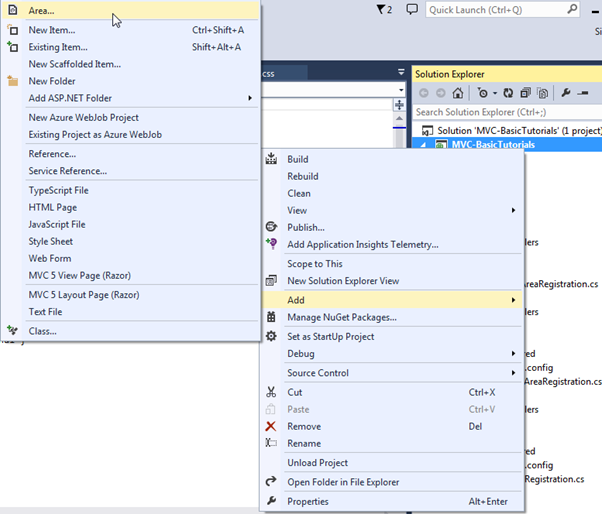
You have already learned that ASP.NET MVC framework includes separate folders for Model, View and Controller. However, large application can include a large number of controller, views and model classes. So to maintain a large number of views, models and controllers with the default ASP.NET MVC project structure can become unmanageable

ASP.NET MVC 2 introduced Area. Area allows us to partition large application into smaller units where each unit contains separate MVC folder structure, same as default MVC folder structure. For example, large enterprise application may have different modules like admin, finance, HR, marketing etc. So an Area can contain separate MVC folder structure for all these modules as shown below.

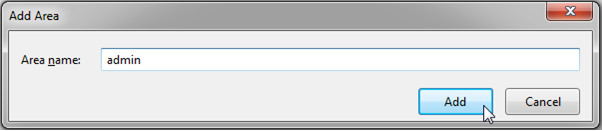
Area

## Create Area

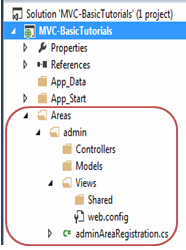
You can create an Area using ASP.NET MVC 5 and Visual Studio 2013 for web by right clicking on the project in the solution explorer -> Add -> Area..

Area

Enter Area name in Add Area dialogue box and click Add.

Area

This will add 'admin' folder under Area folder as shown below.

Area

As you can see, each area includes AreaRegistration class in {area name} + AreaRegistration.cs file.

The following is adminAreaRegistration class created with admin area.

Area Registration:

public class adminAreaRegistration : AreaRegistration

{

public override string AreaName

{

get

{

return "admin";

}

}

public override void RegisterArea(AreaRegistrationContext context)

{

context.MapRoute(

"admin\_default",

"admin/{controller}/{action}/{id}",

new { action = "Index", id = UrlParameter.Optional }

);

}

}

AreaRegistration class overrides RegisterArea method to map the routes for the area. In the above example, any URL that starts with **admin** will be handled by the controllers included in the admin folder structure under Area folder. For example, *http://localhost/admin/profile* will be handled by profile controller included in Areas/admin/controller/ProfileController folder.

Finally, all the area must be registered in Application\_Start event in Global.asax.cs as AreaRegistration.RegisterAllAreas();