ASP.NET MVC Architecture

Here, you will learn an overview of MVC architecture.

The MVC architectural pattern has existed for a long time in software engineering. All most all the languages use MVC with slight variation, but conceptually it remains the same.

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

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

Model: Model represents the shape of the data. A class in C# is used to describe a model. Model objects store data retrieved from the database.

Model represents the data.

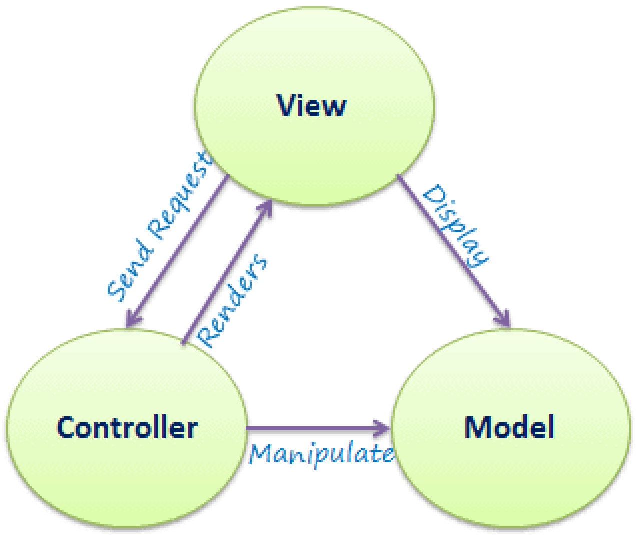
View: View in MVC is a user interface. View display model data to the user and also enables them to modify them. View in ASP.NET MVC is HTML, CSS, and some special syntax (Razor syntax) that makes it easy to communicate with the model and the controller.

View is the User Interface.

Controller: The controller handles the user request. Typically, the user uses the view and raises an HTTP request, which will be handled by the controller. The controller processes the request and returns the appropriate view as a response.

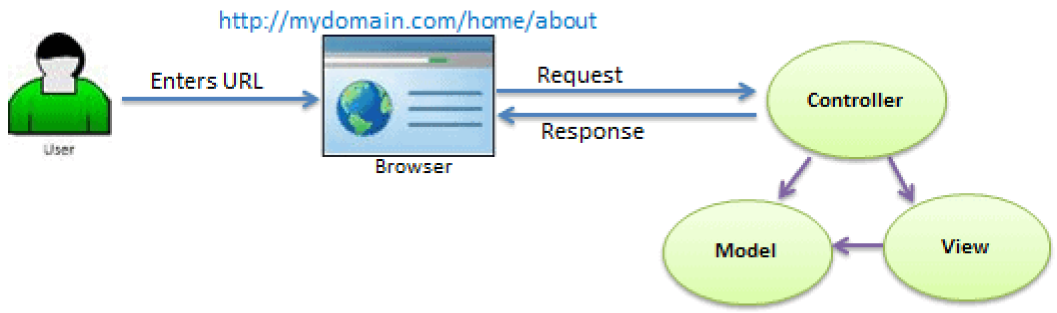
Controller is the request handler.

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



MVC Architecture

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



Request Flow in MVC Architecture

As per the above figure, when a user enters a URL in the browser, it goes to the webserver and routed to a controller. A controller executes related view and models for that request and create the response and sends it back to the browser.

Points to Remember

1. MVC stands for Model, View and Controller.
2. Model represents the data
3. View is the User Interface.
4. Controller is the request handler.

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 Framework	Released 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 update

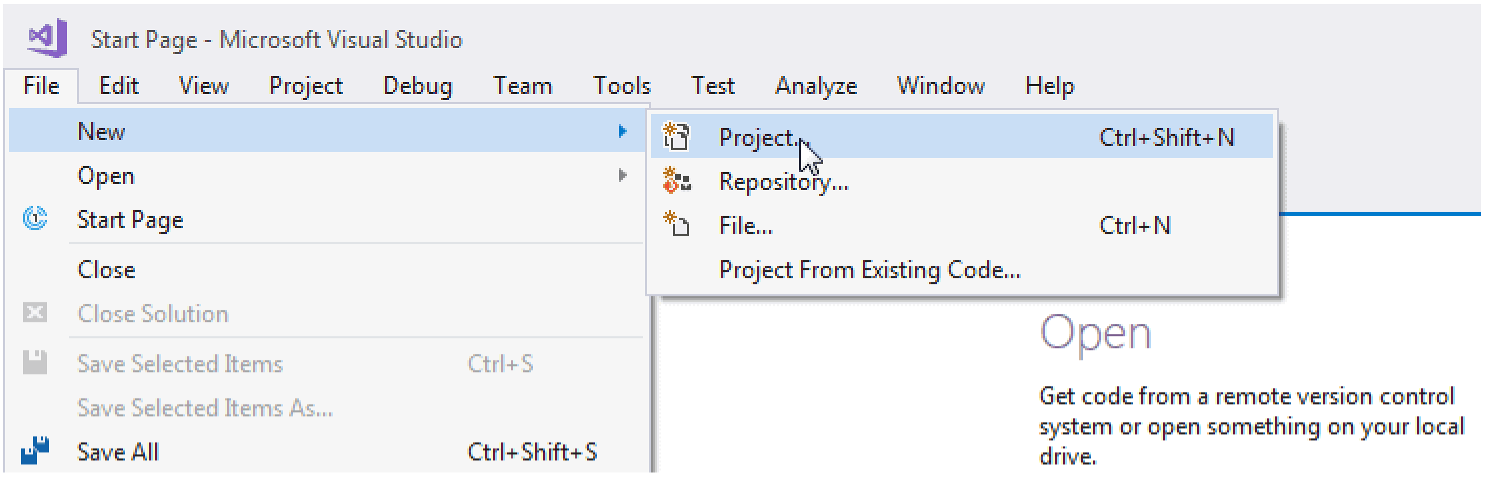
Create ASP.NET MVC Application

In this section, we will create a new MVC web application using Visual Studio and understand the basic building blocks of the ASP.NET MVC Application.

We are going to use ASP.NET MVC v5.2, and Visual Studio 2017 community edition, and .NET Framework 4.6 to create our first MVC application.

Download the latest version of Visual Studio from visualstudio.microsoft.com/downloads.

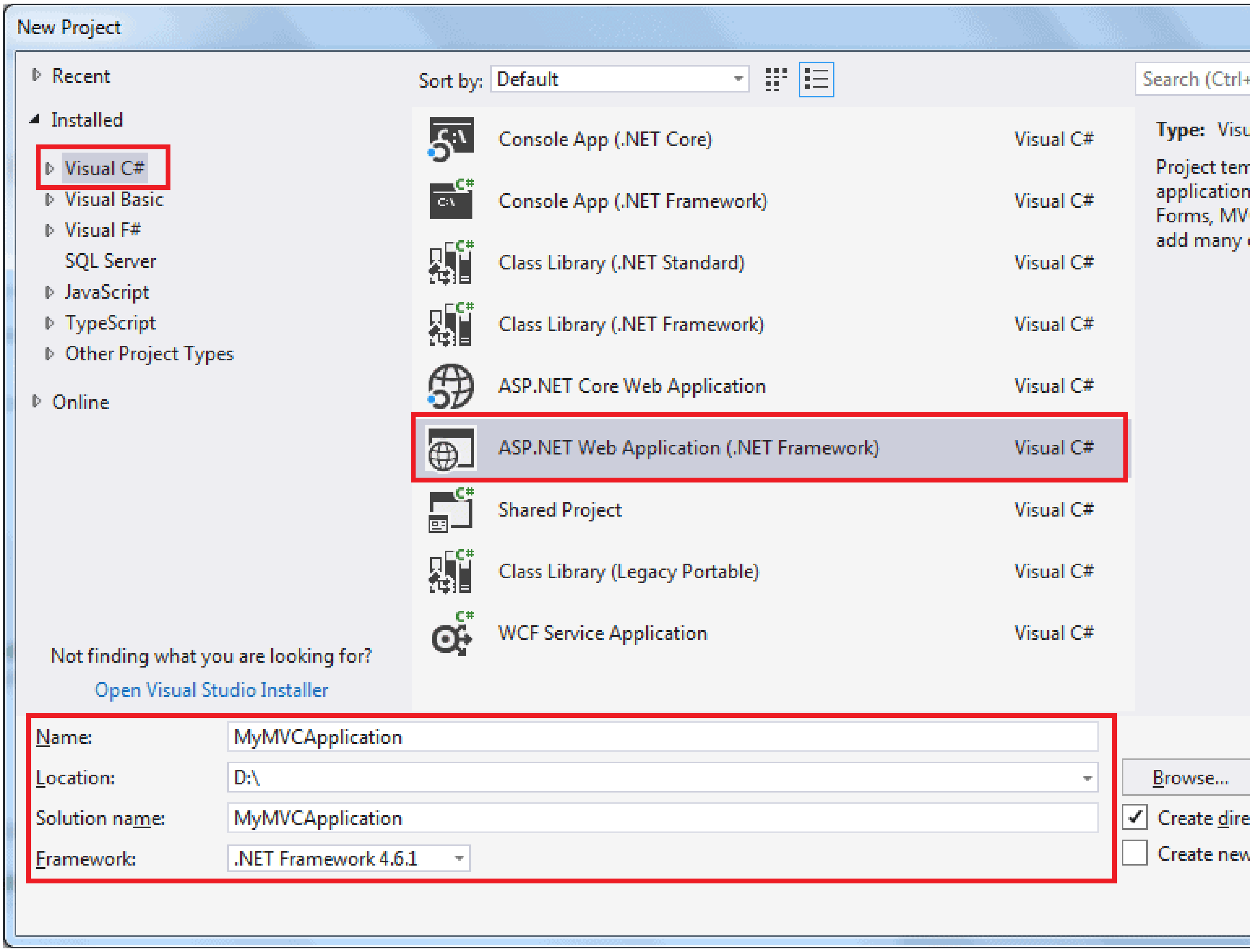
Open Visual Studio 2017 and select **File menu** -> **New** -> **Project**, as shown below.



Get code from a remote version control system or open something on your local drive.

Create a New Project in Visual Studio

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 an appropriate name for your application). Also, you can change the location of the MVC application by clicking on **Browse..** button. Finally, click **OK.**


New Project

- Recent
- Installed
  - Visual C#
  - Visual Basic
  - Visual F#
  - SQL Server
  - JavaScript
  - TypeScript
  - Other Project Types
- Online

Not finding what you are looking for?
Open Visual Studio Installer

Sort by: Default

Console App (.NET Core)	Visual C#
Console App (.NET Framework)	Visual C#
Class Library (.NET Standard)	Visual C#
Class Library (.NET Framework)	Visual C#
ASP.NET Core Web Application	Visual C#
ASP.NET Web Application (.NET Framework)	Visual C#
Shared Project	Visual C#
Class Library (Legacy Portable)	Visual C#
WCF Service Application	Visual C#

Search (Ctrl+

Type: Visu

Project tem
application
Forms, MV
add many

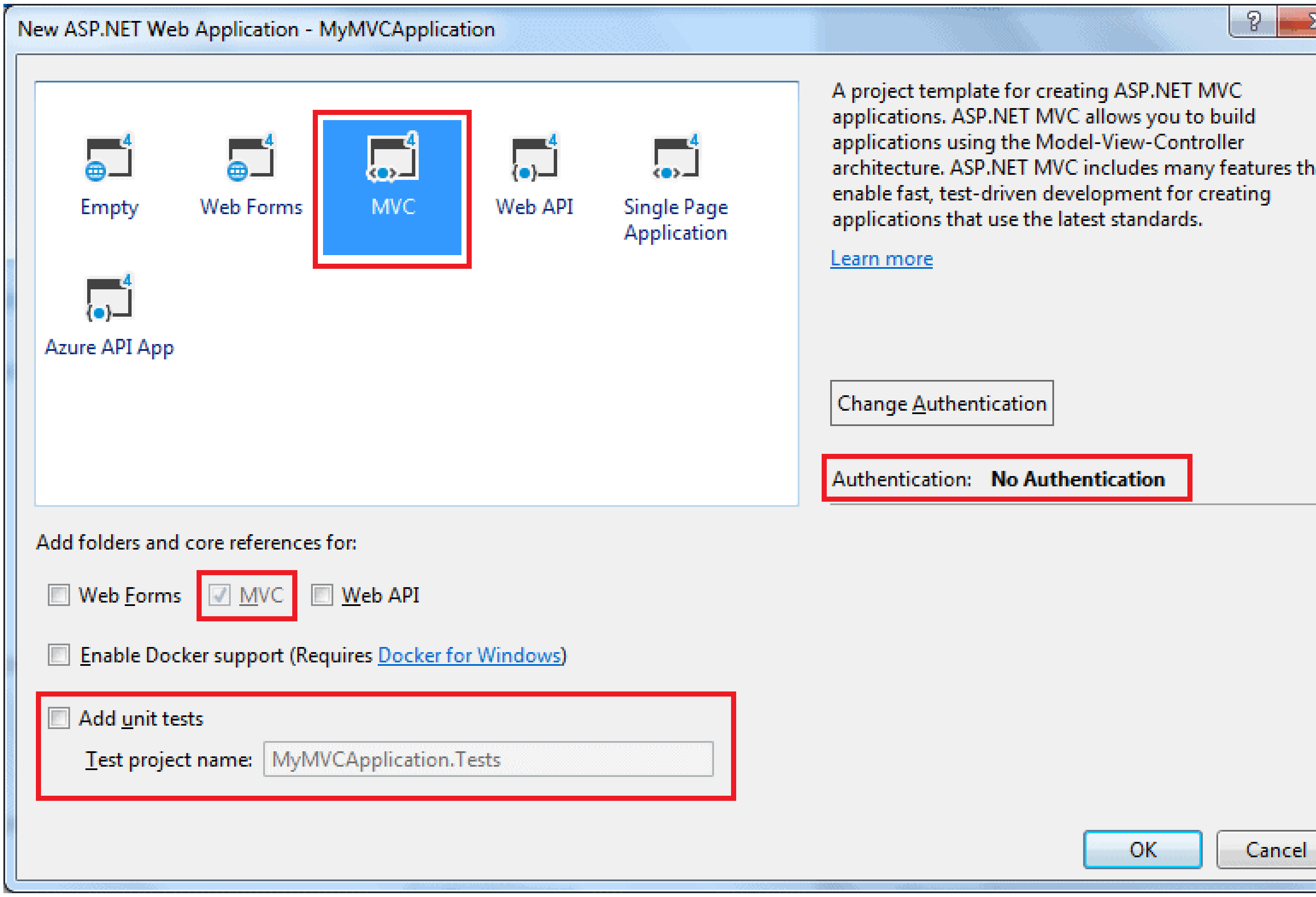
Name:	MyMVCApplication
Location:	D:\
Solution name:	MyMVCApplication
Framework:	.NET Framework 4.6.1

Browse...

[✓] Create dire

[ ] Create new

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


New ASP.NET Web Application - MyMVCApplication

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features th enable fast, test-driven development for creating applications that use the latest standards.

Authentication: **No Authentication**

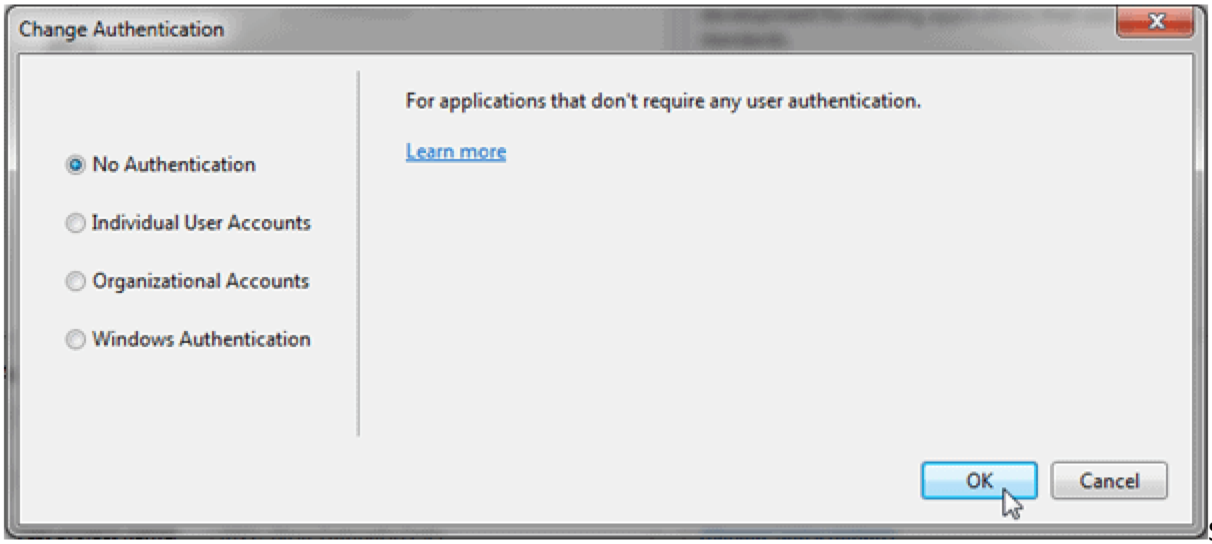
Add folders and core references for:

- [ ] Web Forms
- [x] MVC
- [ ] Web API
- [ ] Enable Docker support (Requires Docker for Windows)
- [ ] Add unit tests

Test project name: MyMVCApplication.Tests

Select MVC Project Template

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


Change Authentication

- [✓ No Authentication]
- Individual User Accounts
- Organizational Accounts
- Windows Authentication

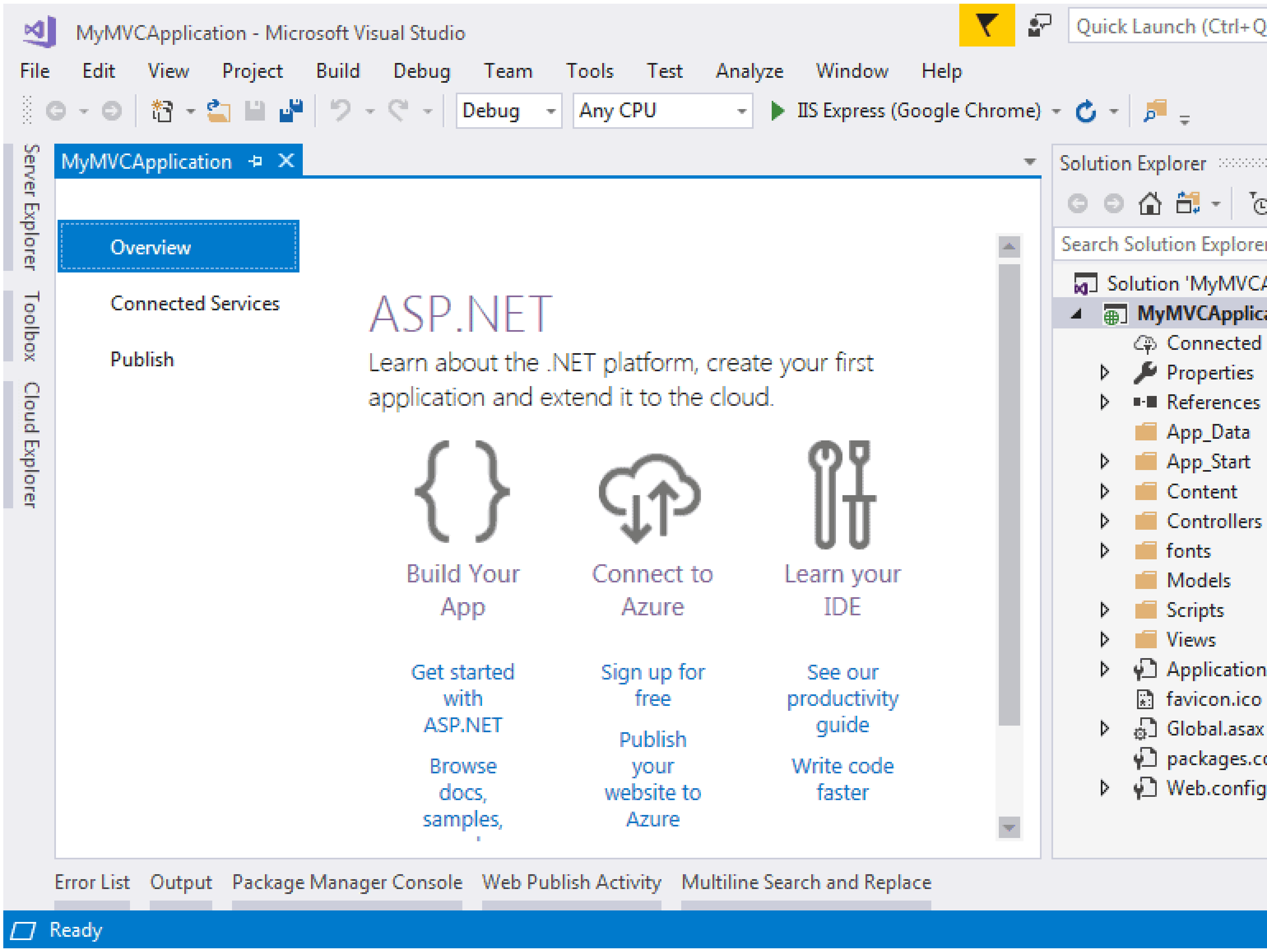
For applications that don't require any user authentication.

Select

Authenctication Type

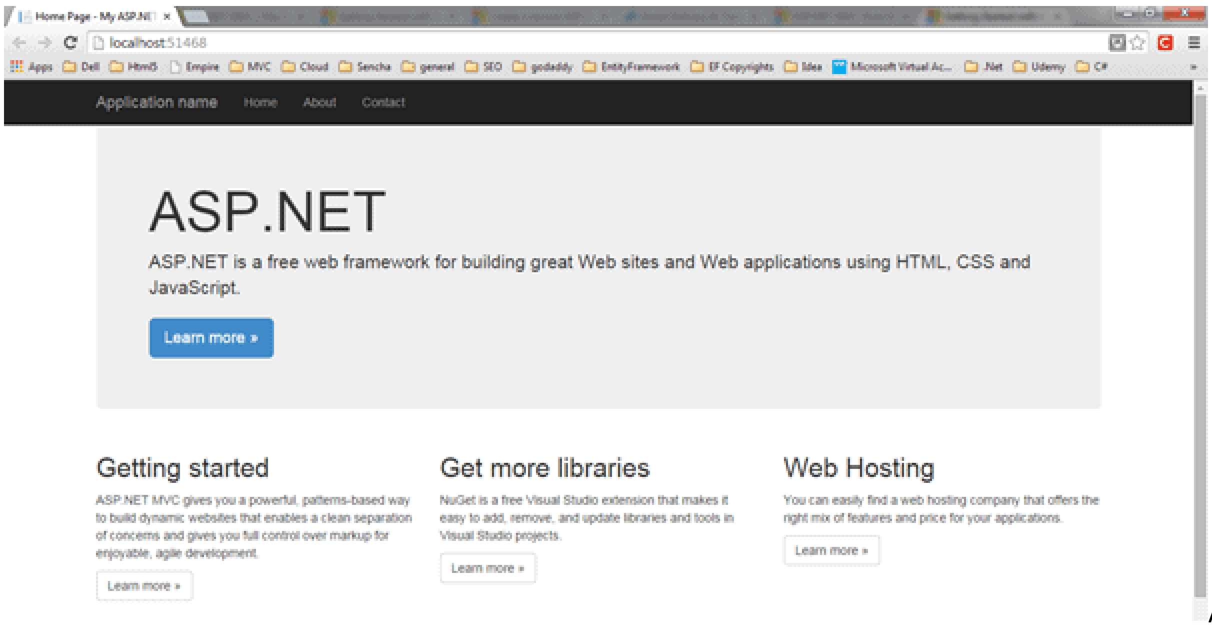
Here, we are keeping the default authentication for our application which is No Authentication. Click **OK** to continue.

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



MVC Project in Visual Studio

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



ASP.NET

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

Getting started

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

Get more libraries

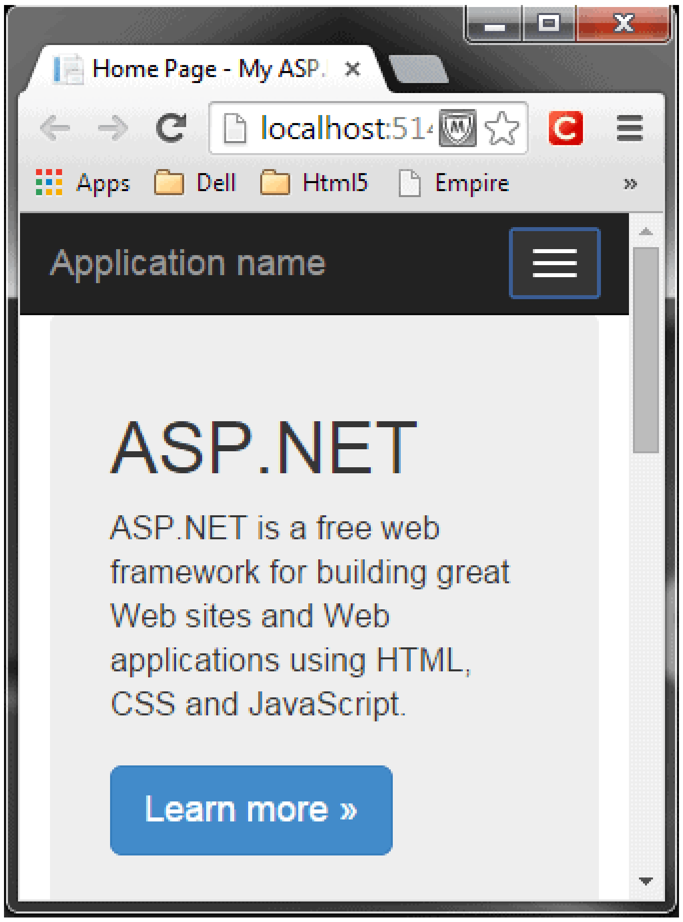
NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.

Web Hosting

You can easily find a web hosting company that offers the right mix of features and price for your applications.

ASP.NET MVC Application

MVC 5 project includes JavaScript and CSS files of bootstrap 3.0 by default. So you can create responsive web pages. This responsive UI will change its look and feel based on the screen size of the different devices. For example, the top menu bar will be changed in the mobile devices, as shown below.

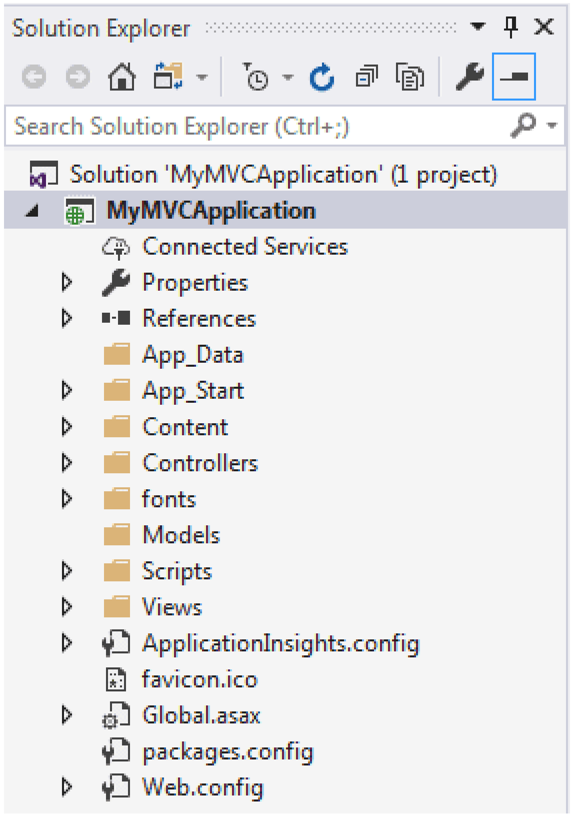


ASP.NET

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

ASP.NET MVC Folder Structure

Here, you will learn about the ASP.NET MVC project structure. Visual Studio creates the following folder structure of the ASP.NET MVC application by default.



MVC Folder Structure

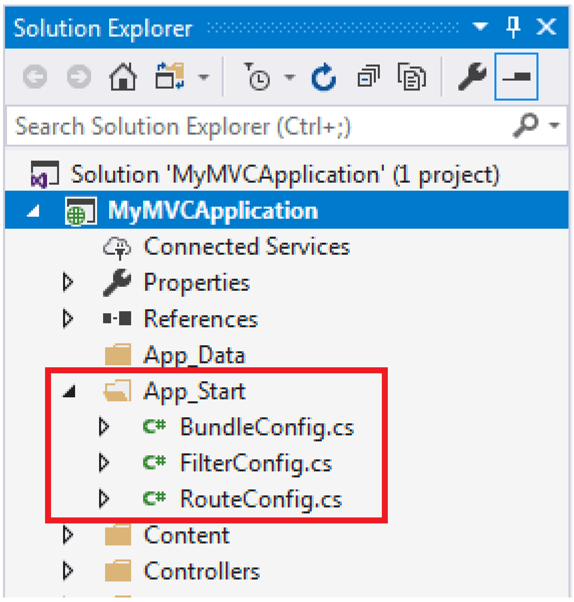
Let's see significance of each folder.

App_Data

The 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

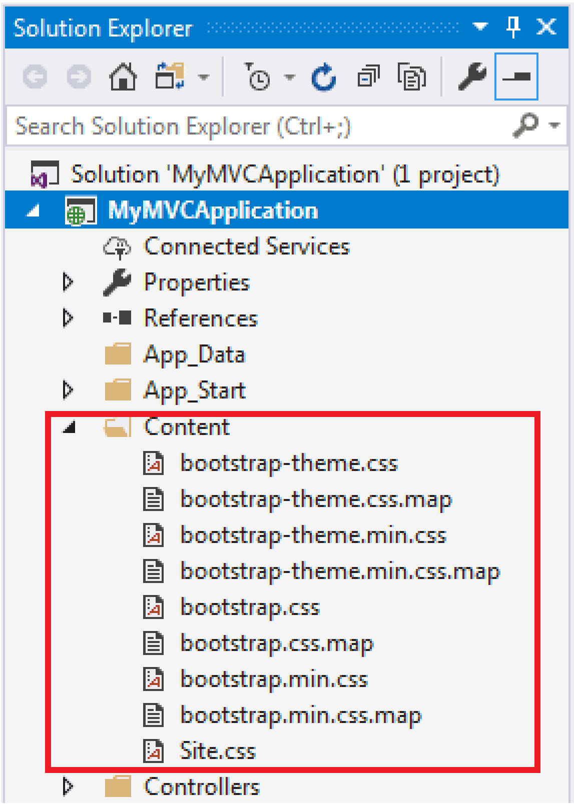
The App_Start folder can contain class files that will be executed when the application starts. Typically, these would be config files like AuthConfig.cs, BundleConfig.cs, FilterConfig.cs, RouteConfig.cs etc. MVC 5 includes BundleConfig.cs, FilterConfig.cs and RouteConfig.cs by default. We will see the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



App_Start Folder

Content

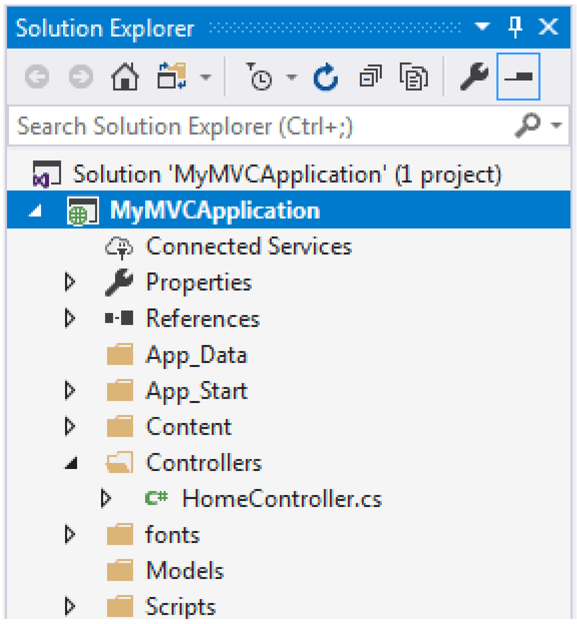
The Content folder contains static files like CSS files, images, and icons files. MVC 5 application includes bootstrap.css, bootstrap.min.css, and Site.css by default.



Content Folder

Controllers

The Controllers folder contains class files for the controllers. A `Controller` 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.

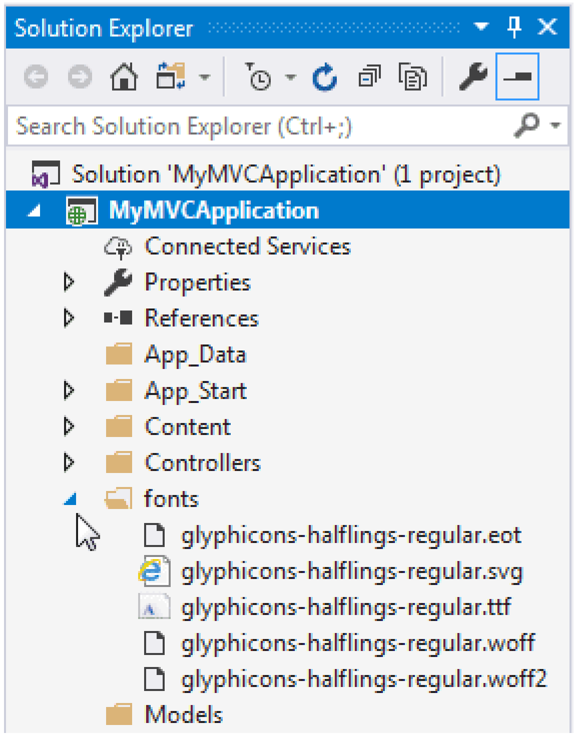


Controller Folder

ADVERTISEMENT

fonts

The Fonts folder contains custom font files for your application.

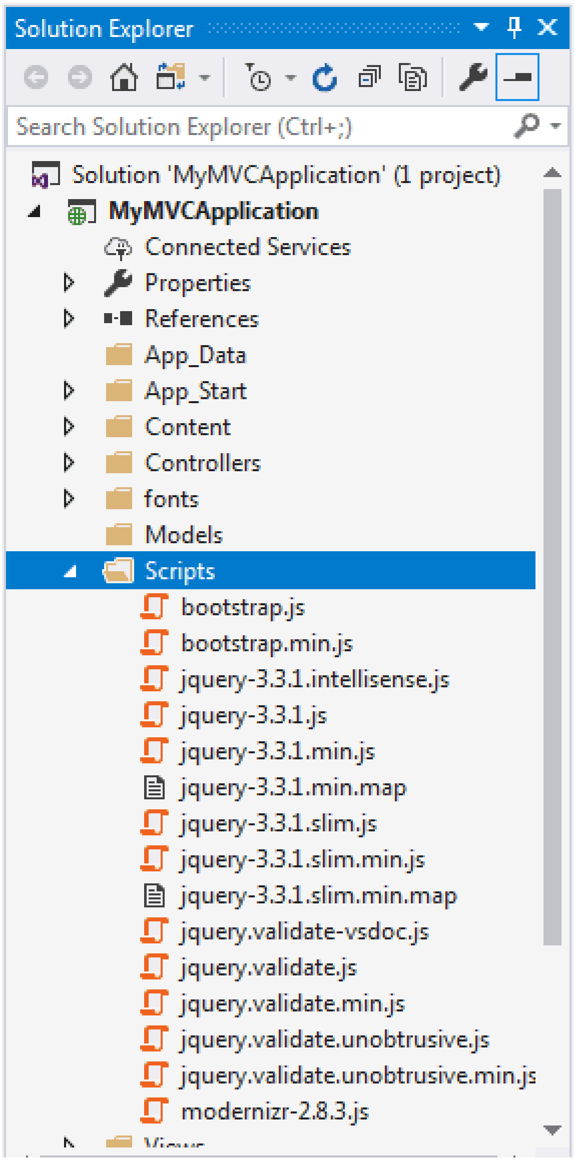

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'MyMVCApplication' (1 project)
MyMVCApplication
Connected Services
Properties
References
App_Data
App_Start
Content
Controllers
fonts
glyphicons-halflings-regular.eot
glyphicons-halflings-regular.svg
glyphicons-halflings-regular.ttf
glyphicons-halflings-regular.woff
glyphicons-halflings-regular.woff2
Models
Fonts folder

Models

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

Scripts

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



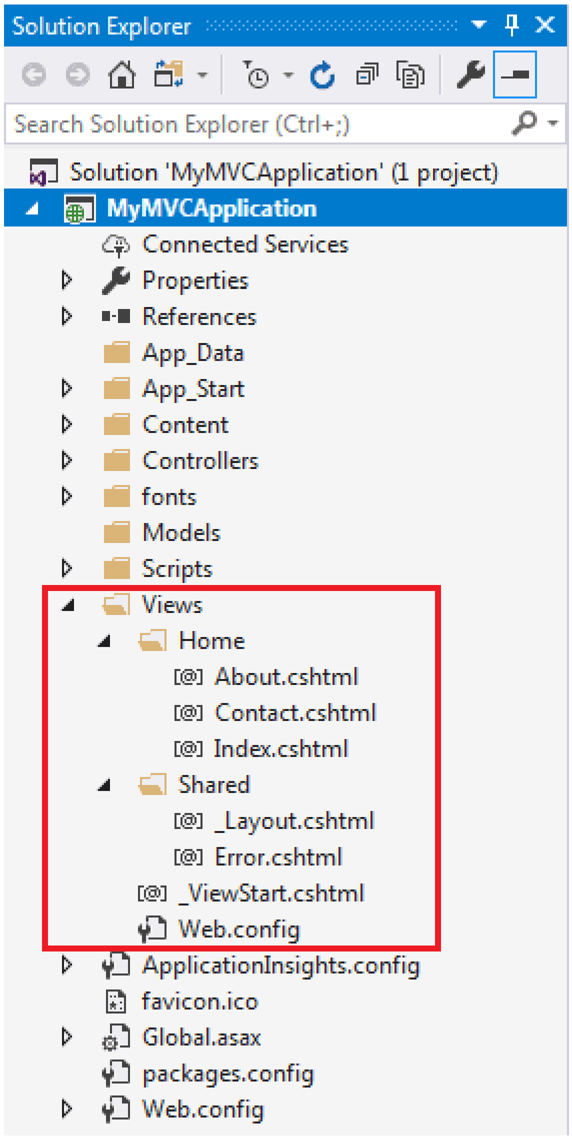
Scripts Folder

Views

The 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.

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

The Shared folder under the View folder contains all the views shared among different controllers e.g., layout files.



View Folder

Additionally, MVC project also includes the following configuration files:

Global.asax

Global.asax file 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 track 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.

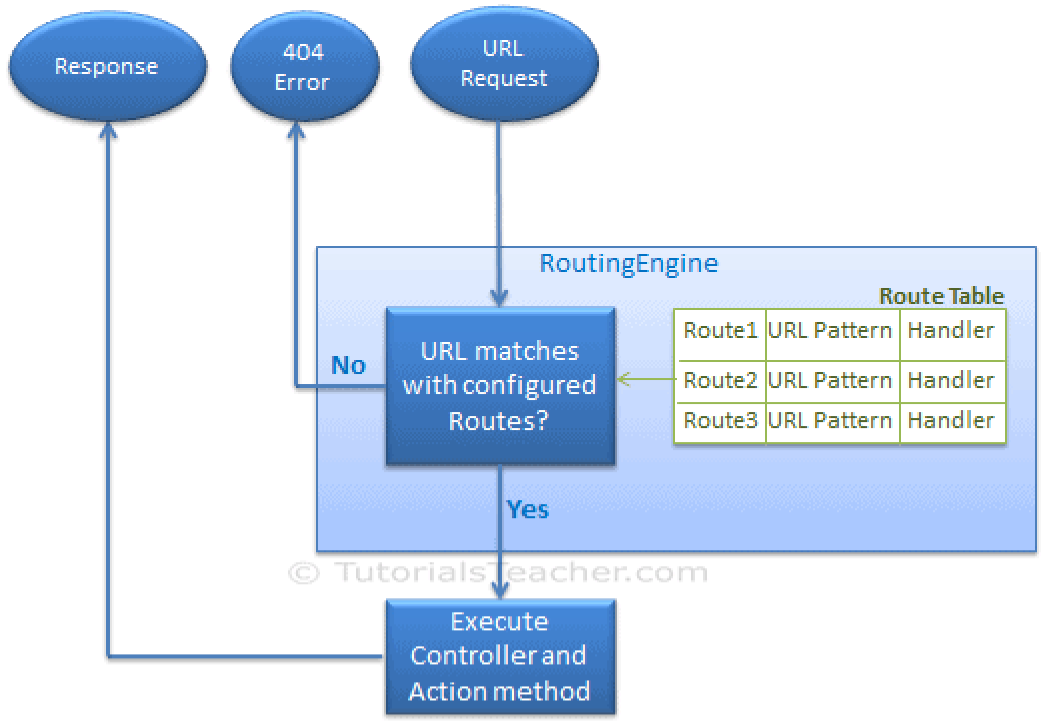
Routing is not specific to the MVC framework. It can be used with ASP.NET Webform application or MVC application.

ASP.NET introduced Routing to eliminate the needs of mapping each URL with a physical file. Routing enables us to define a 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 the 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 the Routing engine to determine appropriate handler class or file for an incoming request.

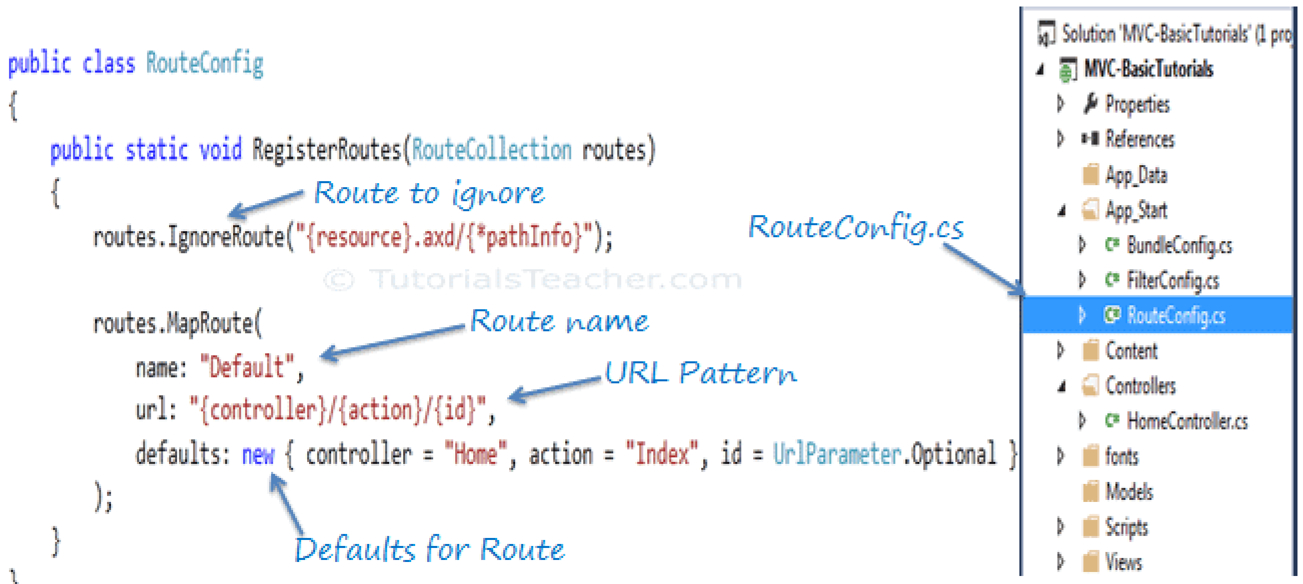
The following figure illustrates the Routing process.


Response
404 Error
URL Request
RoutingEngine
Route Table
Route1 URL Pattern Handler
Route2 URL Pattern Handler
Route3 URL Pattern Handler
URL matches with configured Routes?
No
Yes
© TutorialsTeacher.com
Execute Controller and Action method

Routing in MVC

Configure a Route

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



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

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

Configure

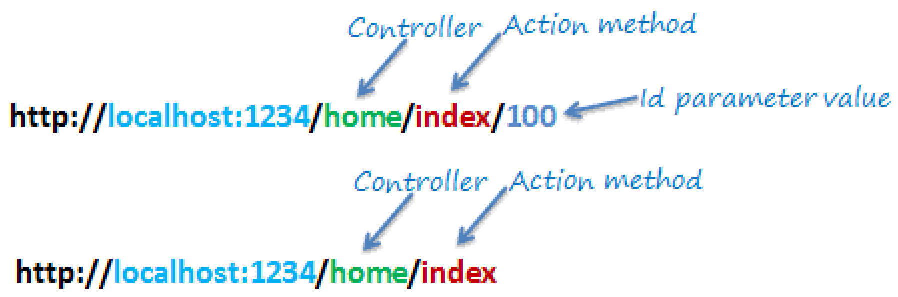
Routes in MVC

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 specify which controller, action method, or value of id parameter should be used if they do not exist in the incoming request URL.

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

URL Pattern

The URL pattern is considered only after the 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 a controller name. The same way, anything after the controller name would be considered as action name and then the value of id parameter.


Controller
Action method
http://localhost:1234/home/index/100
Id parameter value
Controller
Action method
http://localhost:1234/home/index

Routing in MVC

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

The following table shows which Controller, Action method, and Id parameter would handle different URLs considering the 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 the 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

Copy

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

MVC framework evaluates each route in sequence. It starts with the first configured route, and if incoming URL doesn't satisfy the URL pattern of the route, then it will evaluate the second route and so on. In the above example, routing engine will evaluate the `Student` route first and if incoming URL doesn't start with `/students` then only it will consider the second route which is the default route.

The following table shows how different URLs will be mapped to the `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 the parameter by configuring route constraints. For example, the following route applies a limitation on the id parameter that the id's value must be numeric.

Example: Route Constraints

Copy

```
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

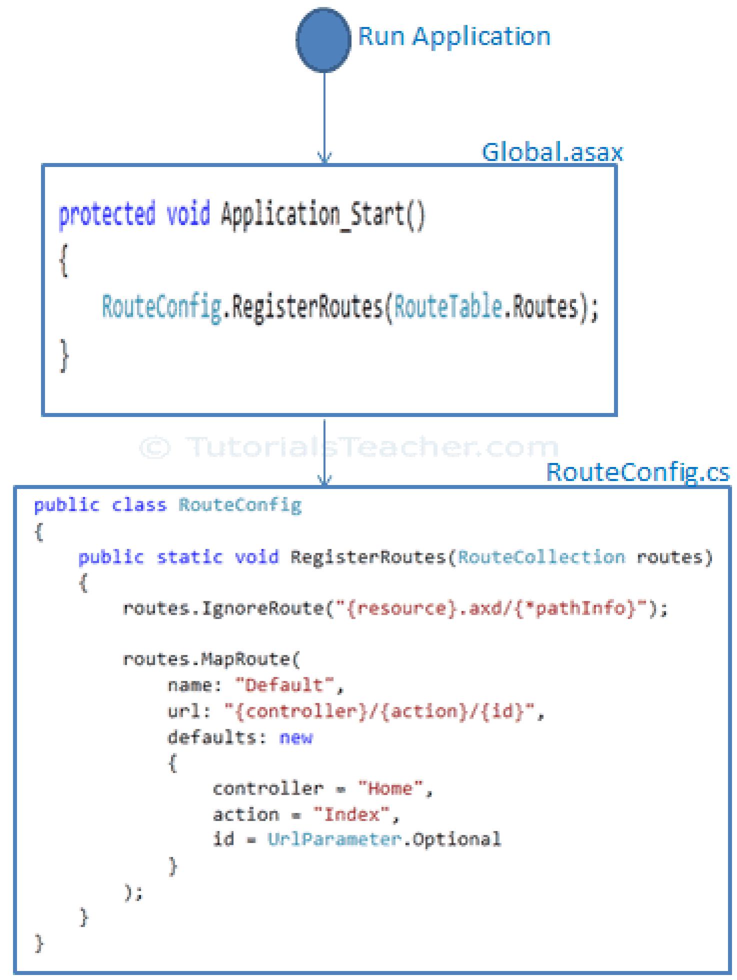
Now, after configuring all the routes in the `RouteConfig` class, you need to register it in the `Application_Start()` event in the `Global.asax` so that it includes all your routes into the `RouteTable`.

Example: Route Registration

Copy

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        RouteConfig.RegisterRoutes(RouteTable.Routes);
    }
}
```

The following figure illustrate Route registration process.



```
protected void Application_Start()
{
    RouteConfig.RegisterRoutes(RouteTable.Routes);
}
```

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

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

Register Route

Thus, routing plays important role in MVC framework.

Points to Remember :

1. Routing plays important role in the 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 the domain name.
3. Routes can be configured in RouteConfig class. Multiple custom routes can also be configured.
4. Route constraints apply restrictions on the value of parameters.
5. Route must be registered in Application_Start event in Global.ascx.cs file.

Controllers in ASP.NET MVC

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

The Controller in MVC architecture handles any incoming URL request. The `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, the home page controller name must be `HomeController`, and for the student page, it must be the `StudentController`. Also, every controller class must be located in the `Controller` folder of the MVC folder structure.

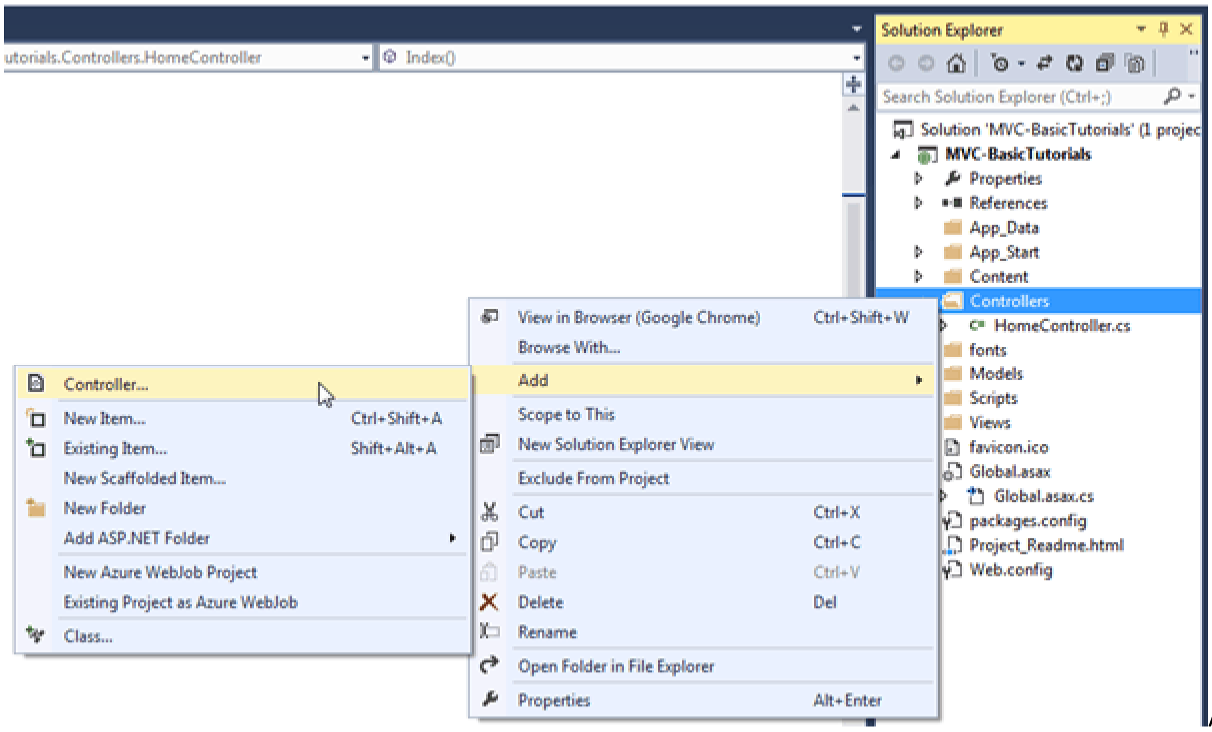
Adding a New Controller

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

TIPS MVC will throw "The resource cannot be found" error when you do not append "Controller" to the controller class name.

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

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

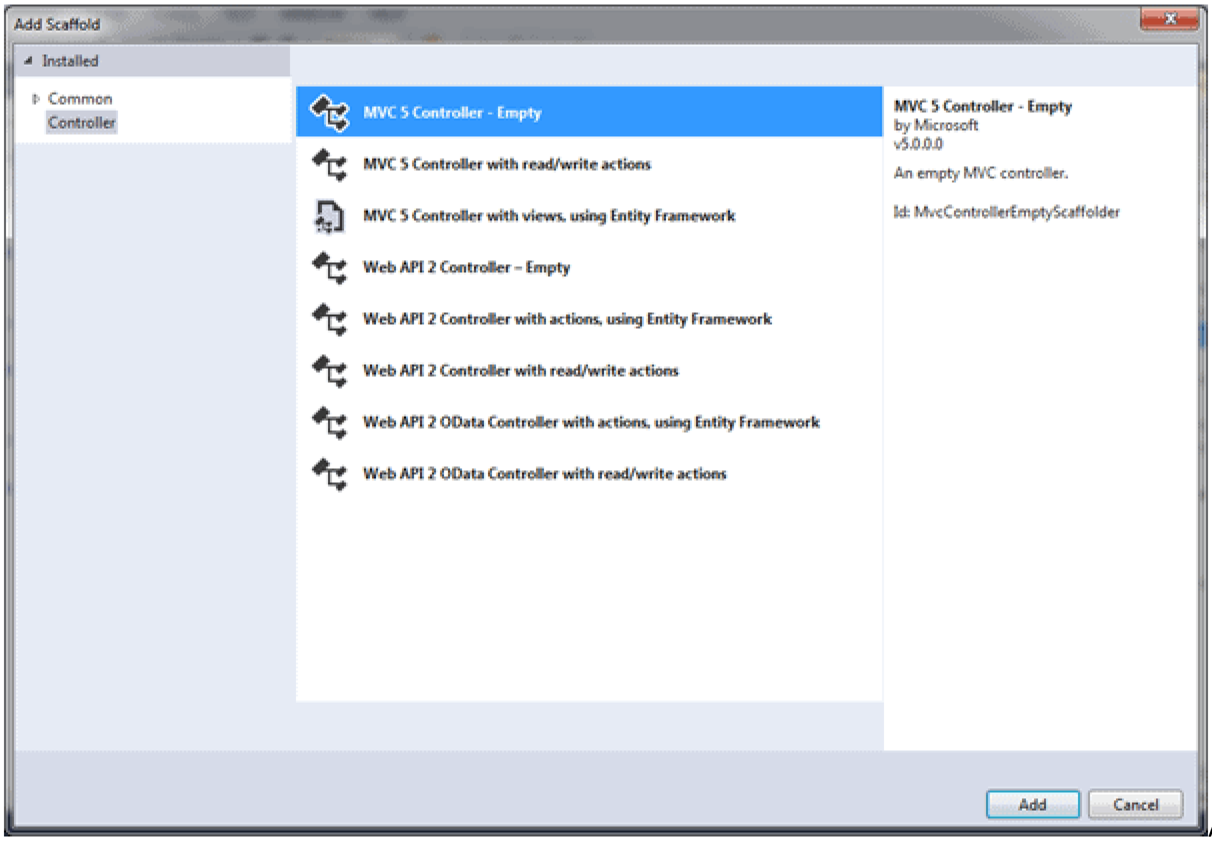


Add New Controller

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 the MVC framework. You can develop a customized scaffolding template using T4 templates as per your architecture and coding standards.

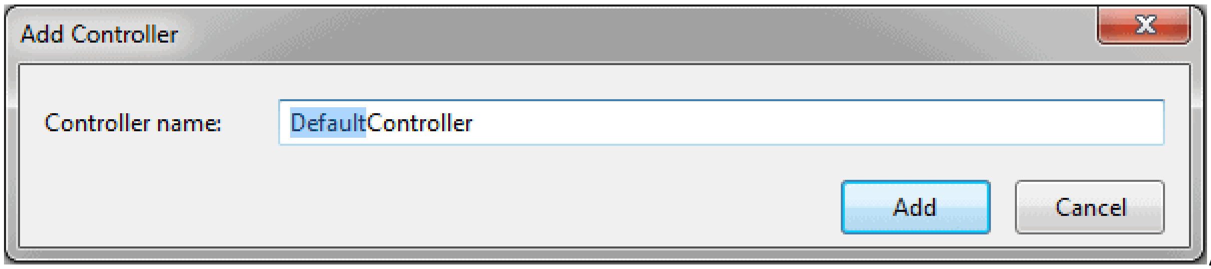


MVC 5 Controller - Empty
MVC 5 Controller with read/write actions
MVC 5 Controller with views, using Entity Framework
Web API 2 Controller – Empty
Web API 2 Controller with actions, using Entity Framework
Web API 2 Controller with read/write actions
Web API 2 OData Controller with actions, using Entity Framework
Web API 2 OData Controller with read/write actions

MVC 5 Controller - Empty
by Microsoft
v5.0.0.0
An empty MVC controller.
Id: MvcControllerEmptyScaffolder

Adding Controller

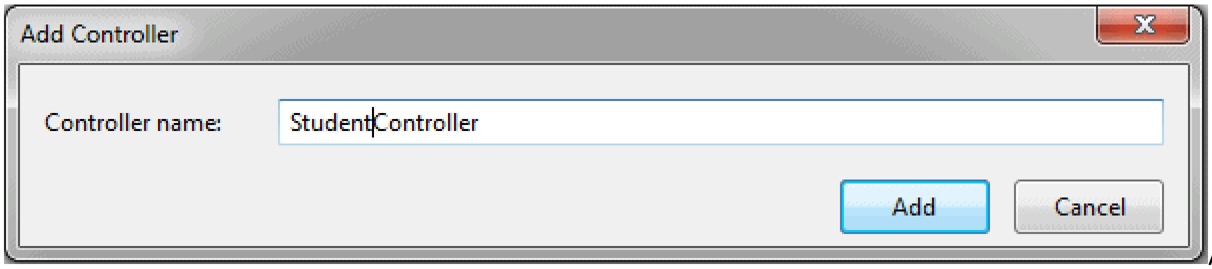
Add Scaffold dialog contains different templates to create a new `controller`. We will learn about other templates later. For now, select `"MVC 5 Controller - Empty"` and click `Add`. It will open the `Add Controller` dialog, as shown below



Controller name: DefaultController

Adding Controller

In the **Add Controller** dialog, enter the name of the controller. Remember, the controller name must end with `Controller`. Write `StudentController` and click **Add**.



Adding Controller

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

Example: Controller

Copy

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

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

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

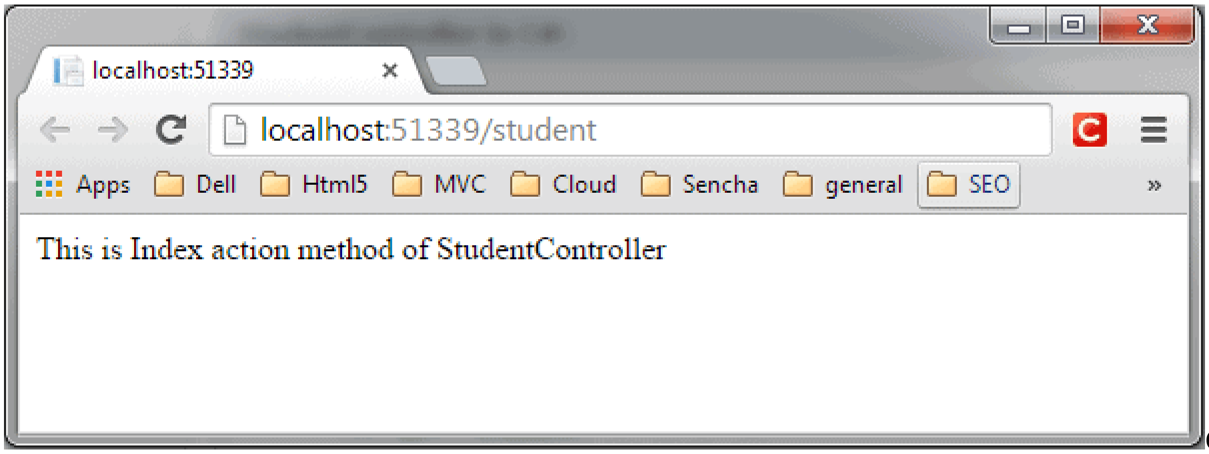
Now, we will return a dummy string from the Index action method of above the `StudentController`. Changing the return type of Index method from `ActionResult` to string and returning dummy string is shown below. You will learn about the `ActionResult` in the next section.

Example: Controller

Copy

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

namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public string Index()
        {
                return "This is Index action method of StudentController";
        }
    }
}
```

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



This is Index action method of StudentController

Controller

Points to Remember :

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

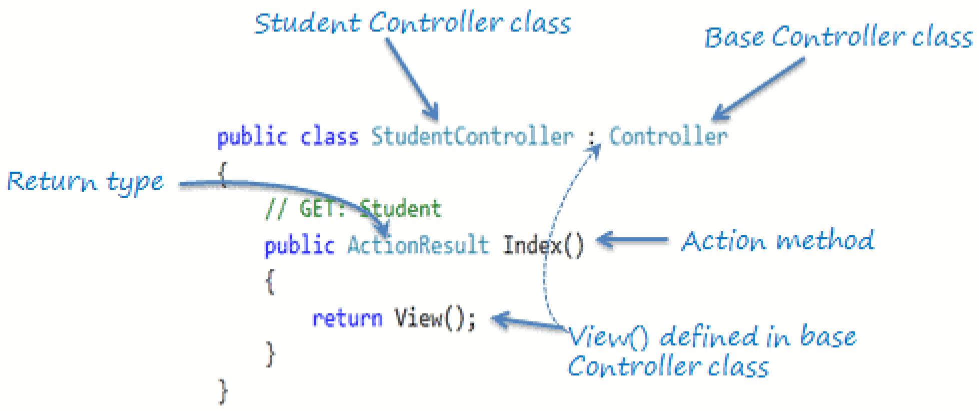
Action method

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

All the public methods of the `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 illustrates the `Index()` action method in the `StudentController` class.



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

Action Method

As you can see in the above figure, the `Index()` method is public, and it returns the `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 a default action method as per the configured route in the `RouteConfig` class. By default, the `Index()` method is a default action method for any controller, as per configured default root, as shown below.

Default Route

Copy

```
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 the `RouteConfig` class.

ActionResult

MVC framework includes various `Result` classes, which can be returned from an action method. The 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 the return type of action method that returns any result listed above. However, you can specify the appropriate result class as a return type of action method.

The `Index()` method of the `StudentController` in the above figure uses the `View()` method to return a `ViewResult` (which is derived from the `ActionResult` class). The base `Controller` class includes the `View()` method along with other methods that return a 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	Represents a JavaScript script.	JavaScript()
JsonResult	Represents JSON that can be used in AJAX.	Json()
RedirectResult	Represents a redirection to a new URL.	Redirect()
RedirectToRouteResult	Represents redirection to another route.	RedirectToRoute()
PartialViewResult	Represents the partial view result.	PartialView()
HttpUnauthorizedResult	Represents HTTP 403 response.	

As you can see in the above table, the `View()` method returns the `ViewResult`, the `Content()` method returns a string, the `File()` method returns the content of a file, and so on. Use different methods mentioned in the above table to return a different type of result 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 below.

Example: Action Method Parameters

Copy

```
[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 match. Visit model binding section for more information on it.

Points to Remember :

1. All the public methods in the Controller class are called Action methods.
2. The Action method has the following restrictions.
   - Action method must be public. It cannot be private or protected.
   - Action method cannot be overloaded.
   - Action method cannot be a static method.
3. ActionResult is a base class of all the result type which returns from Action method.
4. The base Controller class contains methods that returns appropriate result type e.g. View(), Content(), File(), JavaScript() etc.
5. The Action method can include Nullable type parameters.

Action Selectors

Action selector is the attribute that can be applied to the action methods. It helps the 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

The `ActionName` attribute allows us to specify a different action name than the method name, as shown below.

Example: Specify a different action name

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

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

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

NonAction

Use the `NonAction` attribute when you want public method in a controller but do not want to treat it as an action method.

In the following example, the `Index()` method is an action method, but the `GetStudent()` is not an action method.

Example: NonAction

Copy

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

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

ActionVerbs: HttpGet, HttpPost, HttpPut

The `ActionVerbs` selector is to handle different type of Http requests. The MVC framework includes HttpGet, HttpPost, HttpPut, HttpDelete, HttpOptions, and HttpPatch action verbs. You can apply one or more action verbs to an action method to handle different HTTP requests. If you don't apply any action verbs to an action method, then it will handle HttpGet request by default.

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 the message body.
OPTIONS	It represents a request for information about the communication options supported by the web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

Visit W3.org for more information on Http Methods.

The following example shows how to handle different types of HTTP requests in the `Controller` using ActionVerbs:

Example: Handle HTTP Requests in the Controller

Copy

```
public class StudentController : Controller
{
    public ActionResult Index() // handles GET requests by default
    {
        return View();
    }

    [HttpPost]
    public ActionResult PostAction() // handles POST requests by default
    {
        return View("Index");
    }
    [HttpPut]
    public ActionResult PutAction() // handles PUT requests by default
    {
        return View("Index");
    }

    [HttpDelete]
    public ActionResult DeleteAction() // handles DELETE requests by default
    {
        return View("Index");
    }

    [HttpHead]
    public ActionResult HeadAction() // handles HEAD requests by default
    {
        return View("Index");
    }

    [HttpOptions]
    public ActionResult OptionsAction() // handles OPTION requests by default
    {
        return View("Index");
    }

    [HttpPatch]
    public ActionResult PatchAction() // handles PATCH requests by default
    {
        return View("Index");
    }
}
```

You can also apply multiple action verbs using the `AcceptVerbs` attribute, as shown below.

Example: AcceptVerbs

Copy

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

Model in ASP.NET MVC

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

The model classes represents domain-specific data and business logic in the MVC application. It represents the shape of the data as public properties and business logic as methods.

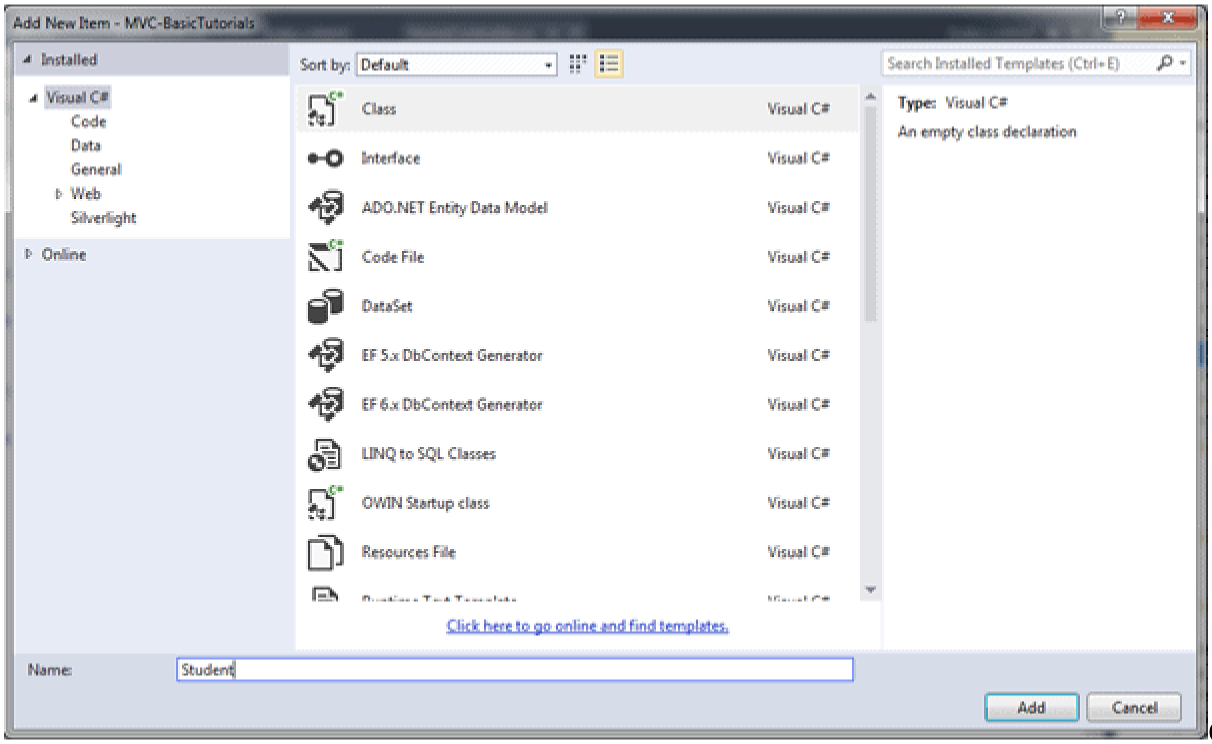
In the ASP.NET MVC Application, all the Model classes must be created in the Model folder.

Adding a Model Class

Let's create the model class that should have the required properties for the `Student` entity.

In the MVC application in Visual Studio, and right-click on the `Model` folder, select **Add** -> and click on **Class...**. It will open the **Add New Item** dialog box.

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


Add New Item - MVC-BasicTutorials
Installed
Visual C#
Code
Data
General
Web
Silverlight
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Class Visual C#
Interface Visual C#
ADO.NET Entity Data Model Visual C#
Code File Visual C#
DataSet Visual C#
EF 5.x DbContext Generator Visual C#
EF 6.x DbContext Generator Visual C#
LINQ to SQL Classes Visual C#
OWIN Startup class Visual C#
Resources File Visual C#
Type: Visual C#
An empty class declaration
Click here to go online and find templates.
Add
Cancel

Name: Student

Create Model Class

This will add a new `Student` class in model folder. We want this model class to store id, name, and age of the students. So, we will have to add public properties for `Id`, `Name`, and `Age`, as shown below.

Example: Model class

Copy

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

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

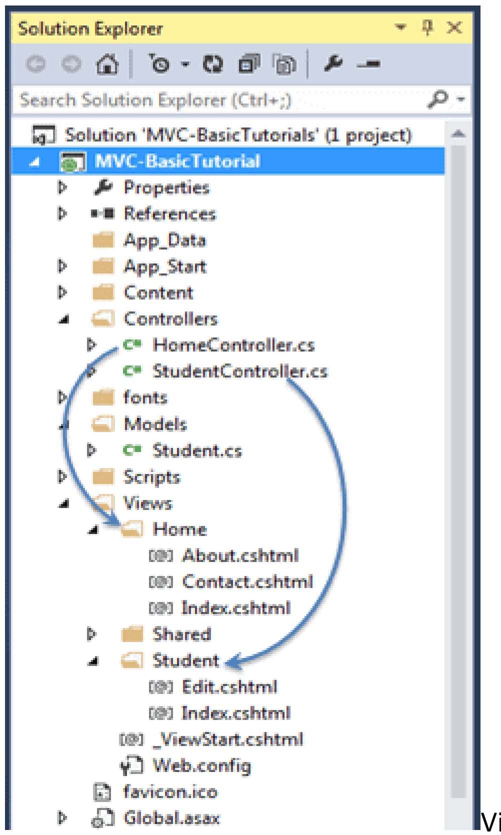
Create a View in ASP.NET MVC

In this section, you will learn how to create a view and use the model class in it in the ASP.NET MVC application.

A view is used to display data using the model class object. The **Views** folder contains all the view files in the ASP.NET MVC application.

A controller can have one or more action methods, and each action method can return a different view. In short, a controller can render one or more views. So, for easy maintenance, the MVC framework requires a separate sub-folder for each controller with the same name as a controller, under the **Views** folder.

For example, all the views rendered from the `HomeController` will resides in the **Views** > **Home** folder. In the same way, views for `StudentController` will resides in **Views** > **Student** folder, as shown below.



View folders for Controllers

Note:

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

Razor View Engine

Microsoft introduced the razor view engine to compile a view with a mix of HTML tags and server-side code. The special syntax for razor view maximizes the speed of writing code by minimizing the number of characters and keystrokes required when writing a view.

The razor view uses @ character to include the server-side code instead of the traditional <% %> of ASP. You can use C# or Visual Basic syntax to write server-side code inside the razor view.

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

File extension	Description
.cshtml	C# Razor view. Supports C# code with html tags.
.vbhtml	Visual Basic Razor view. Supports Visual Basic code with html tags.
.aspx	ASP.Net web form

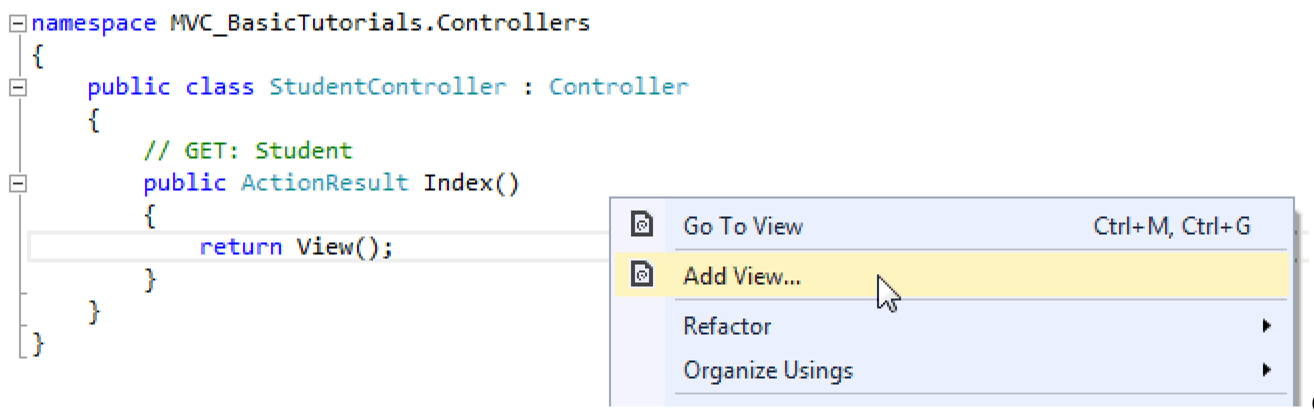
File extension	Description
.ascx	ASP.NET web control

Learn [Razor syntax](#) in the next section.

Creating a View

You can create a view for an action method directly from it by right clicking inside an action method and select **Add View...**.

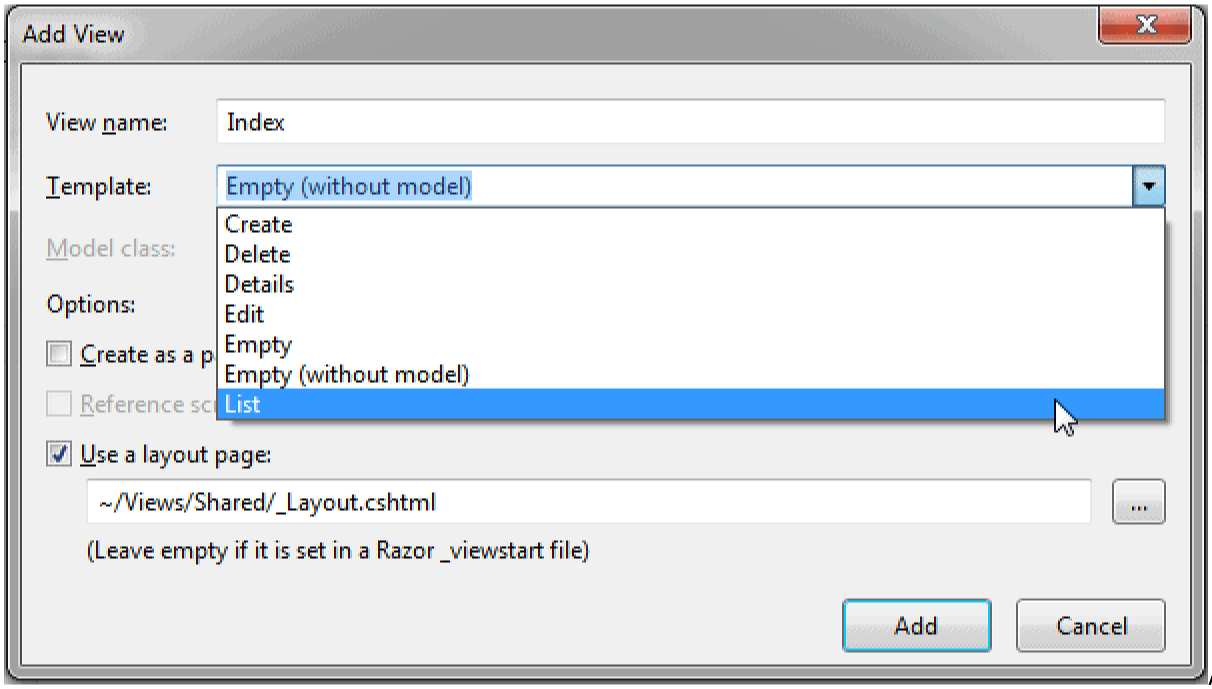
The following creates a view from the `Index()` action method of the `StudentContoller`, as shown below.



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

Create a View from Action Method

This will open the **Add View** dialogue box, shown below. It's good practice to keep the view name the same as the action method name so that you don't have to explicitly specify the view name in the action method while returning the view.


Add View

View name: Index

Template: Empty (without model)

Model class:

Options:

- [ ] Create as a partial view
- [ ] Reference script libraries
- [✓] Use a layout page:

~/Views/Shared/_Layout.cshtml

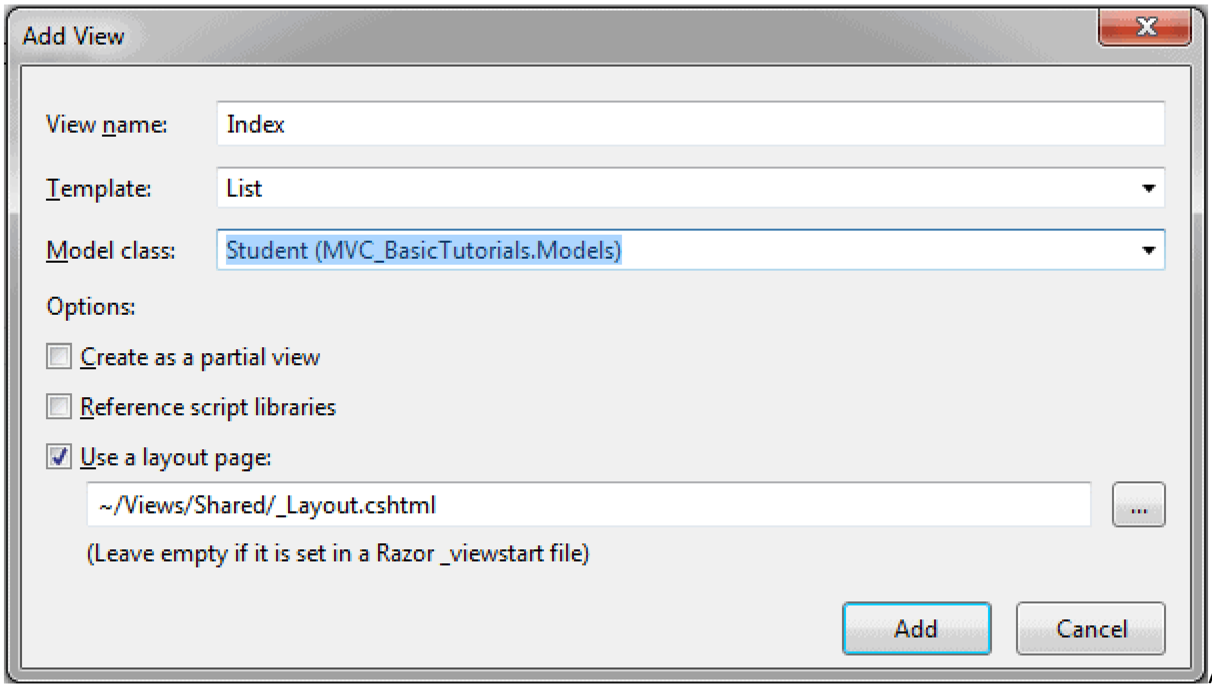
(Leave empty if it is set in a Razor _viewstart file)

Add a 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 the list of students in the view.

ADVERTISEMENT

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


Add View

View name: Index

Template: List

Model class: Student (MVC_BasicTutorials.Models)

Options:

- [ ] Create as a partial view
- [ ] Reference script libraries
- [✓] Use a layout page:

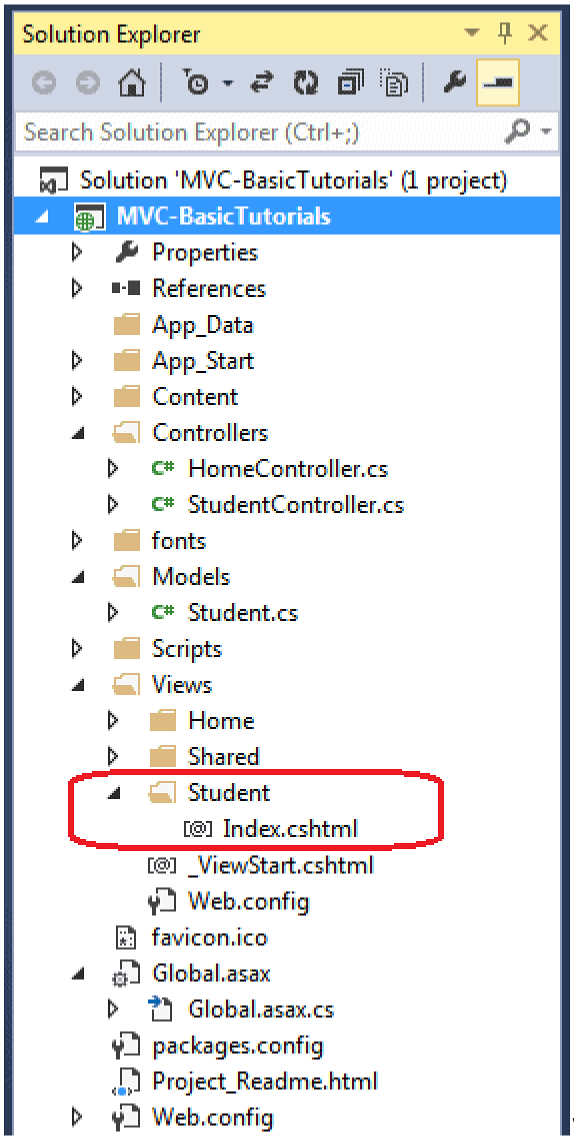
~/Views/Shared/_Layout.cshtml

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

Add a View

Check "Use a layout page" checkbox and select the default `_Layout.cshtml` page for this view and then click **Add** button.

This will create the `Index` view under **View** -> **Student** folder, as shown below:



View

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

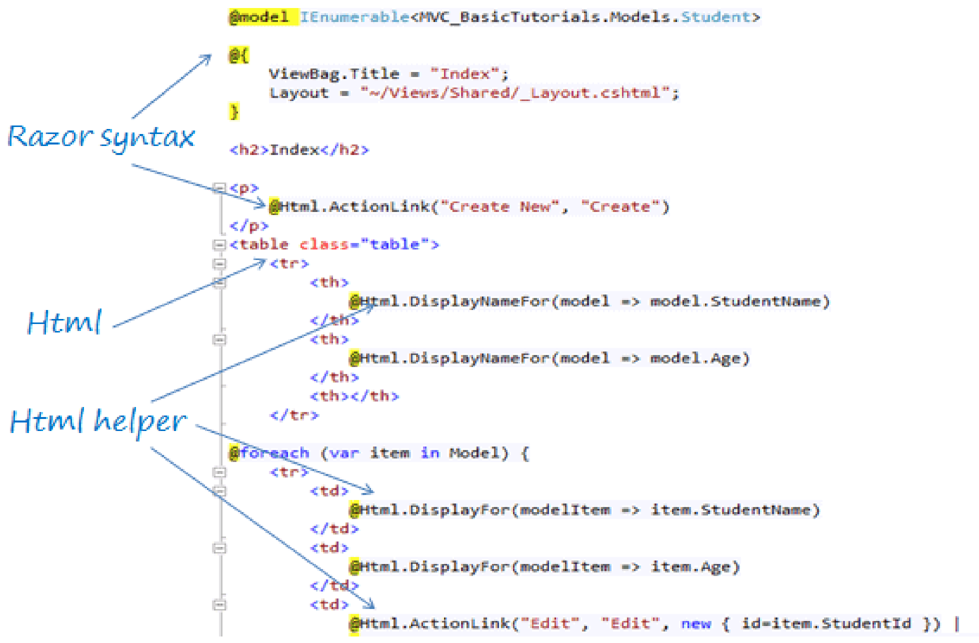
Views\Student\Index.cshtml:

Copy

```
@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>
```

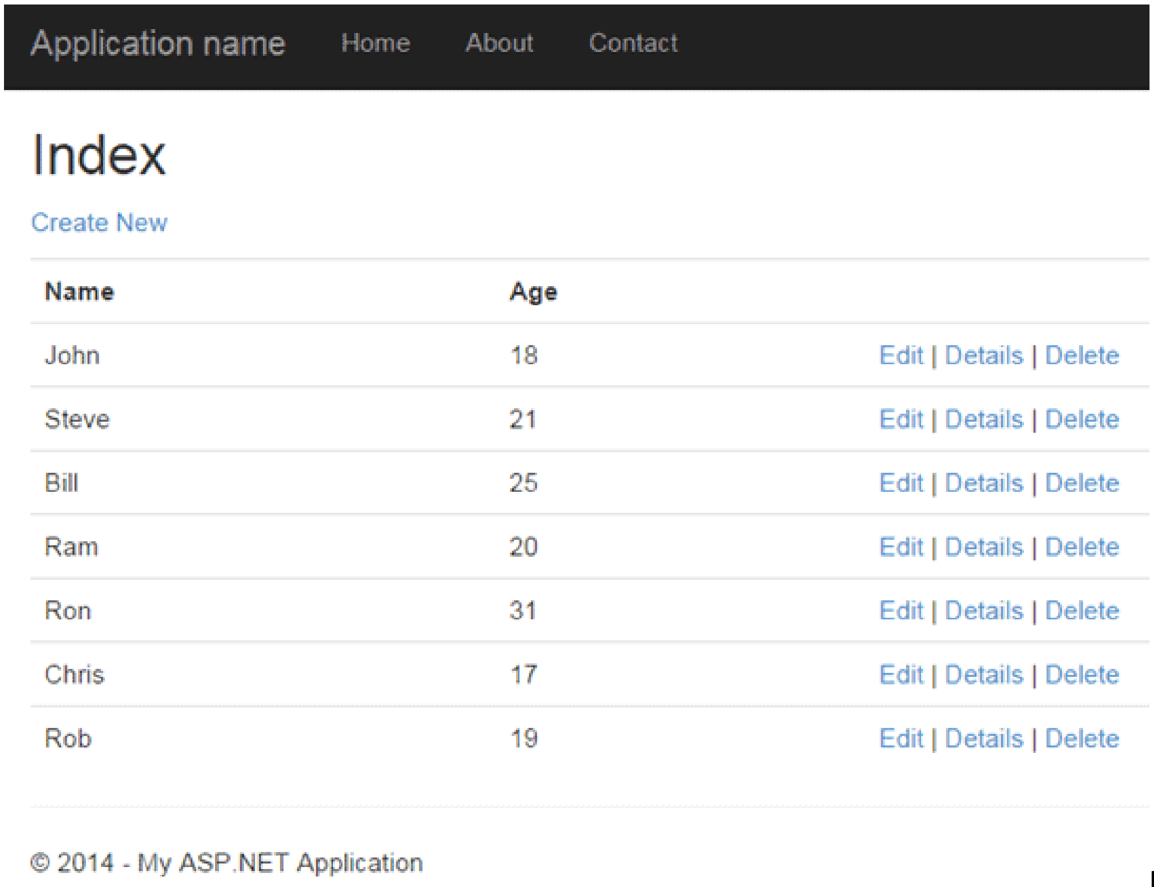
As you can see in the above `Index` view, it contains both HTML and razor codes. Inline razor expression starts with @ symbol. @Html is a helper class to generate HTML controls. You will learn razor syntax and HTML helpers in the coming sections.



```
@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 }) |
```

Index.cshtml

The above Index view would look as below when we run the application.



Index

Create New

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

© 2014 - My ASP.NET Application

Index View

Note:

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

Integrate Controller, View and Model

We have already created a Controller, a model and a view in the previous sections. Here, we will integrate them to run the application and see the result.

The following code snippet shows the `StudentController`, the `Student` model, and the `Index.cshtml` view created in the previous sections.

Example: StudentController

Copy

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

Example: Student Model class

Copy

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

Example: Index.cshtml View

Copy

```
@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 an action method to a view. As you can see in the above `Index.cshtml`, it uses `IEnumerable<Student>` as a model type. So we need to pass it from the `Index()` action method of the `StudentController` class, as shown below.

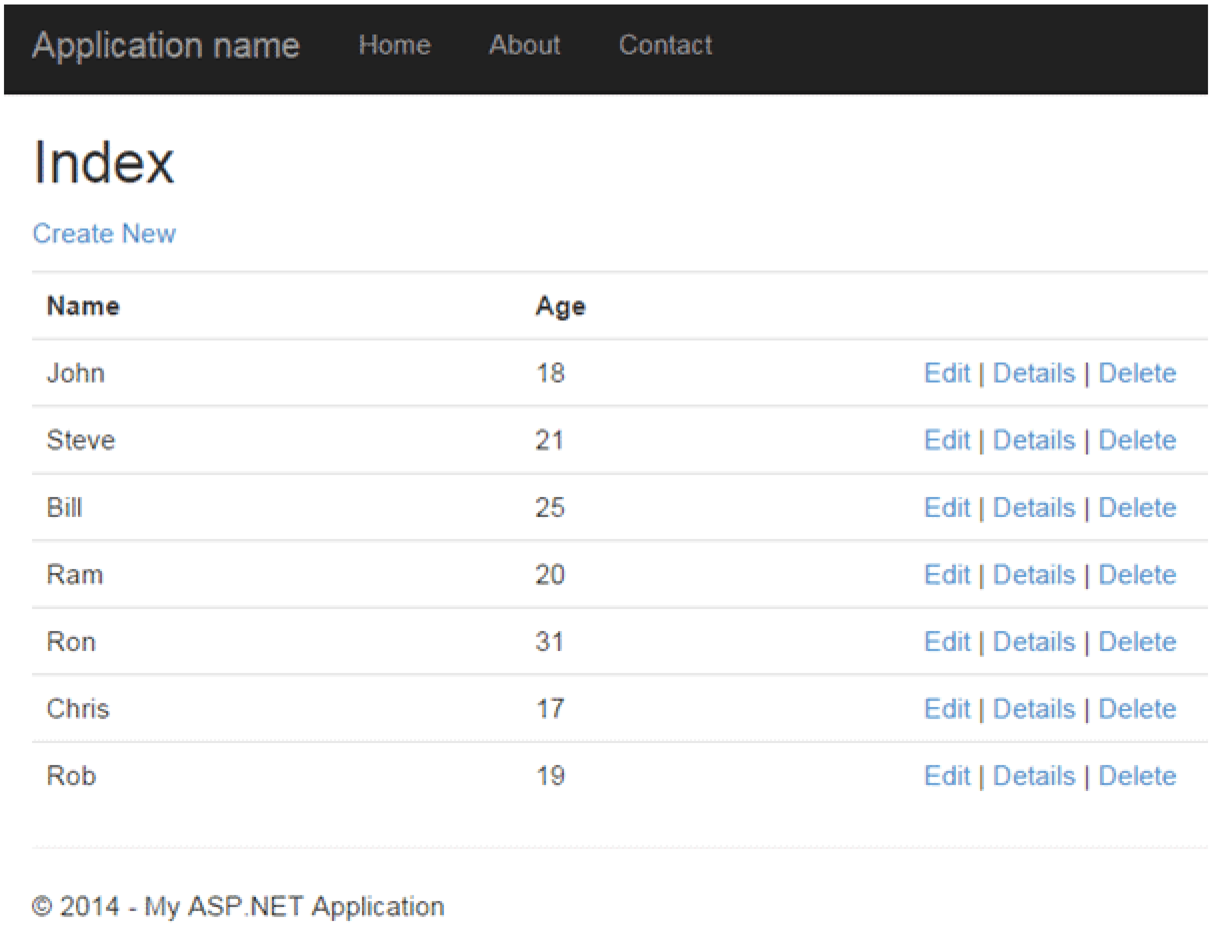
Example: Passing Model from Controller

Copy

```
public class StudentController : Controller
{
    static IList<Student> studentList = new List<Student>{
                new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,
                new Student() { StudentId = 2, StudentName = "Steve",  Age = 21 } ,
                new Student() { StudentId = 3, StudentName = "Bill",  Age = 25 } ,
                new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,
                new Student() { StudentId = 5, StudentName = "Ron" , Age = 31 } ,
                new Student() { StudentId = 4, StudentName = "Chris" , Age = 17 } ,
                new Student() { StudentId = 4, StudentName = "Rob" , Age = 19 }
            };
    // GET: Student
    public ActionResult Index()
    {
        //fetch students from the DB using Entity Framework here
        return View(studentList);
    }
}
```

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

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



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

© 2014 - My ASP.NET Application

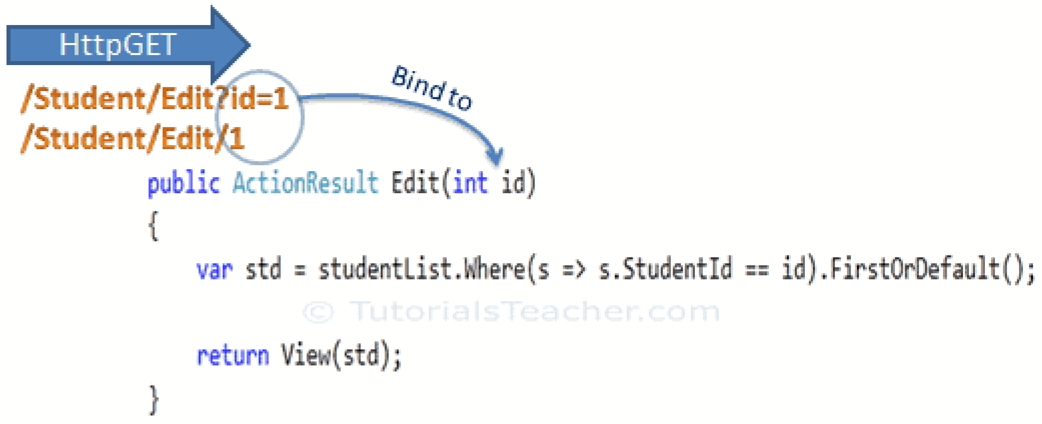
Bind Query String to an Action Method Parameters in MVC

Here, you will learn about to bind a model object to an action method parameters in the ASP.NET MVC application.

The model binding refers to converting the HTTP request data (from the query string or form collection) to an action method parameters. These parameters can be of primitive type or complex type.

Binding to Primitive Type

The HTTP GET request embeds data into a query string. MVC framework automatically converts a query string to the action method parameters provided their names are matching. For example, the query string `id` in the following GET request would automatically be mapped to the `Edit()` action method's `id` parameter.



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

    return View(std);
}
```

Model Binding

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

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

For example, the query string parameters of an HTTP request `http://localhost/Student/Edit?id=1&name=John` would map to `id` and `name` parameters 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. It will automatically convert the input fields data on the view to the properties of a complex type parameter of an action method in HttpPost request if the properties' names match with the fields on the view.

Example: Model classes in C#

```
 public class Student

{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
    public Standard standard { get; set; }
}

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

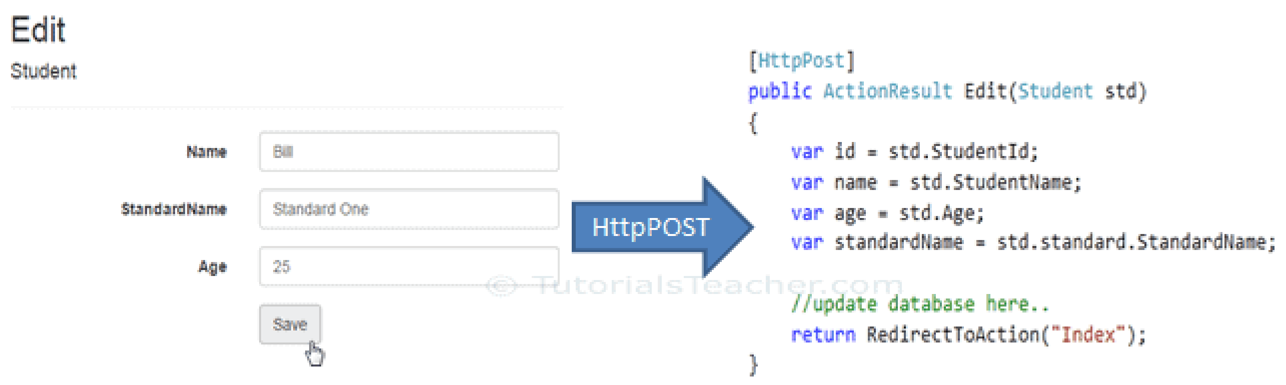
Now, you can create an action method which includes the Student type parameter. In the following example, Edit action method (HttpPost) includes Student type parameter.

Example: Action Method with Class Type Parameter

Copy

```
[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");
}
```

Thus, the MVC framework will automatically map Form collection values to the Student type parameter when the form submits an HTTP POST request to the `Edit()` action method, as shown below.



```
[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");
}
```

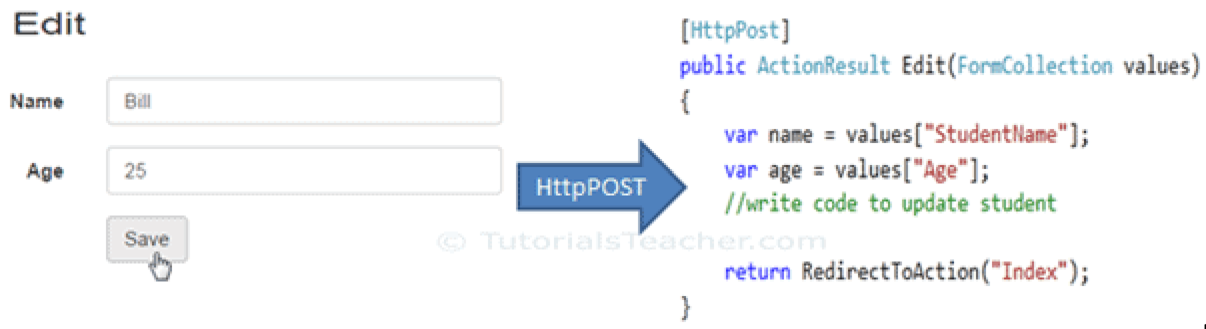
Model Binding to Complex Type

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

ADVERTISEMENT

FormCollection

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



```
[HttpPost]
public ActionResult Edit(FormCollection values)
{
    var name = values["StudentName"];
    var age = values["Age"];
    //write code to update student

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

Model Binding to FormCollection

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 of a model should include or exclude in binding.

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

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 exclude the properties, as shown below.

Example: Exclude Properties in Binding

Copy

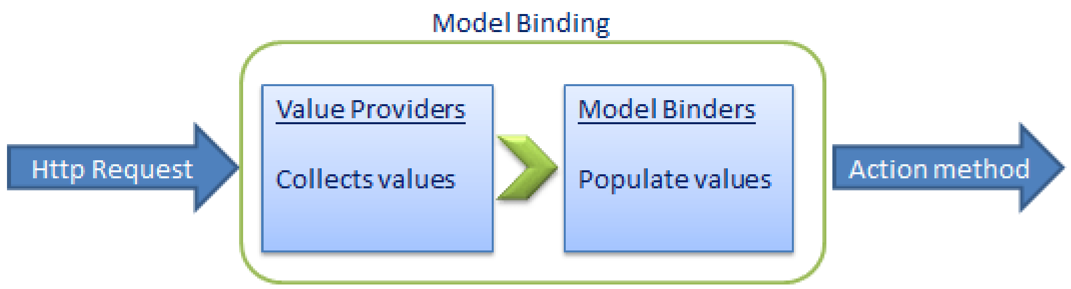
```
[HttpPost]
public ActionResult Edit([Bind(Exclude = "Age")] Student std)
{
    var name = std.StudentName;

    //write code to update student

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

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

Model Binding Process

As you have seen, that the ASP.NET MVC framework 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, it populates primitive type or a complex type with these values.

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



Model Binding in ASP.NET MVC

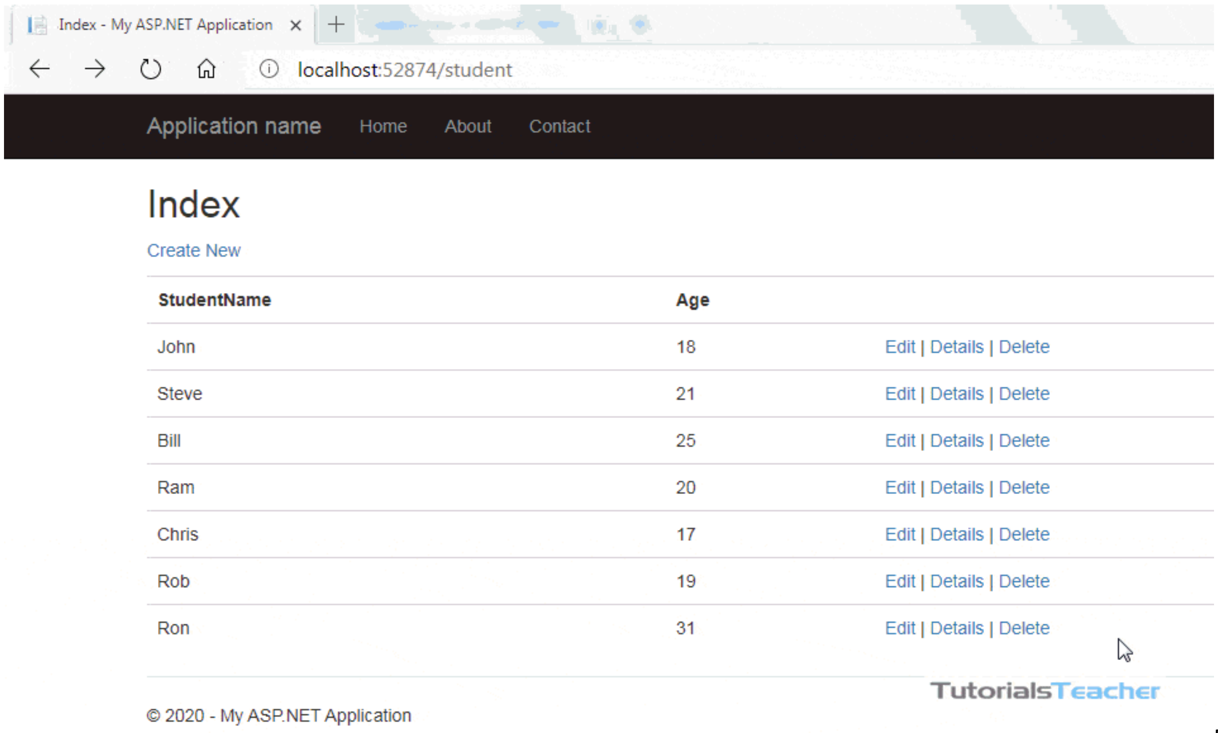
Default value provider collection evaluates values from the following sources:

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

MVC includes DefaultModelBinder class which effectively binds most of the model types.

Create Edit View in ASP.NET MVC

We created the list view in the Integrate Model, View, Controller chapter. Here, you will learn how to create the edit view where the users can edit the data. The following illustrates the steps involved in editing a student's record.



Index - My ASP.NET Application

localhost:52874/student

Application name Home About Contact

Index

Create New

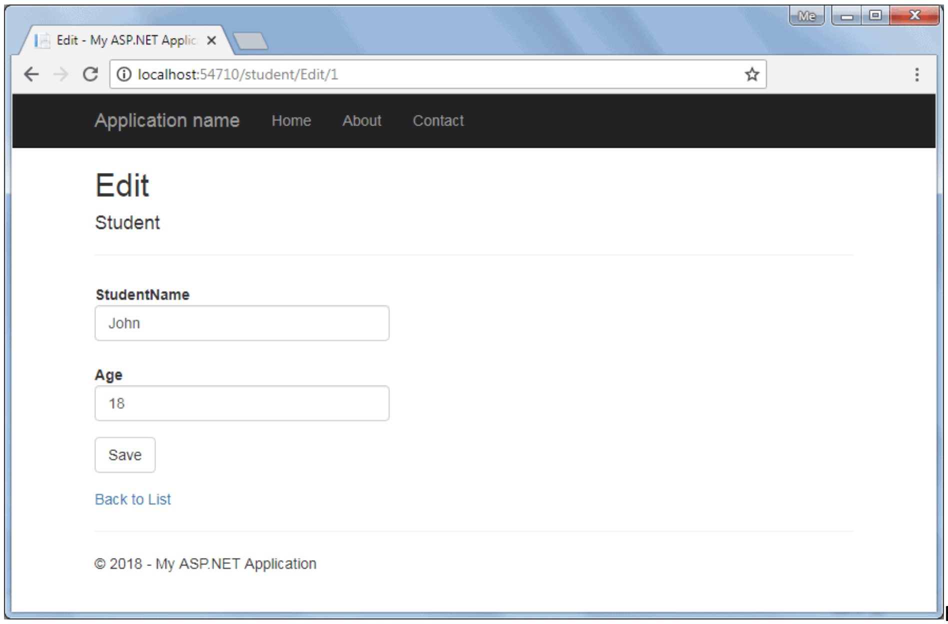
StudentName	Age	
John	18	Edit | Details | Delete
Steve	21	Edit | Details | Delete
Bill	25	Edit | Details | Delete
Ram	20	Edit | Details | Delete
Chris	17	Edit | Details | Delete
Rob	19	Edit | Details | Delete
Ron	31	Edit | Details | Delete

TutorialsTeacher

© 2020 - My ASP.NET Application

Editing Steps in ASP.NET MVC Application

The edit view will be rendered on the click of the `Edit` link in the student list view, which we already created the student list view in the Create a View chapter. Here, we will build the following edit view in order to edit a student record.



Edit - My ASP.NET Applic

localhost:54710/student/Edit/1

Application name Home About Contact

Edit

Student

StudentName

John

Age

18

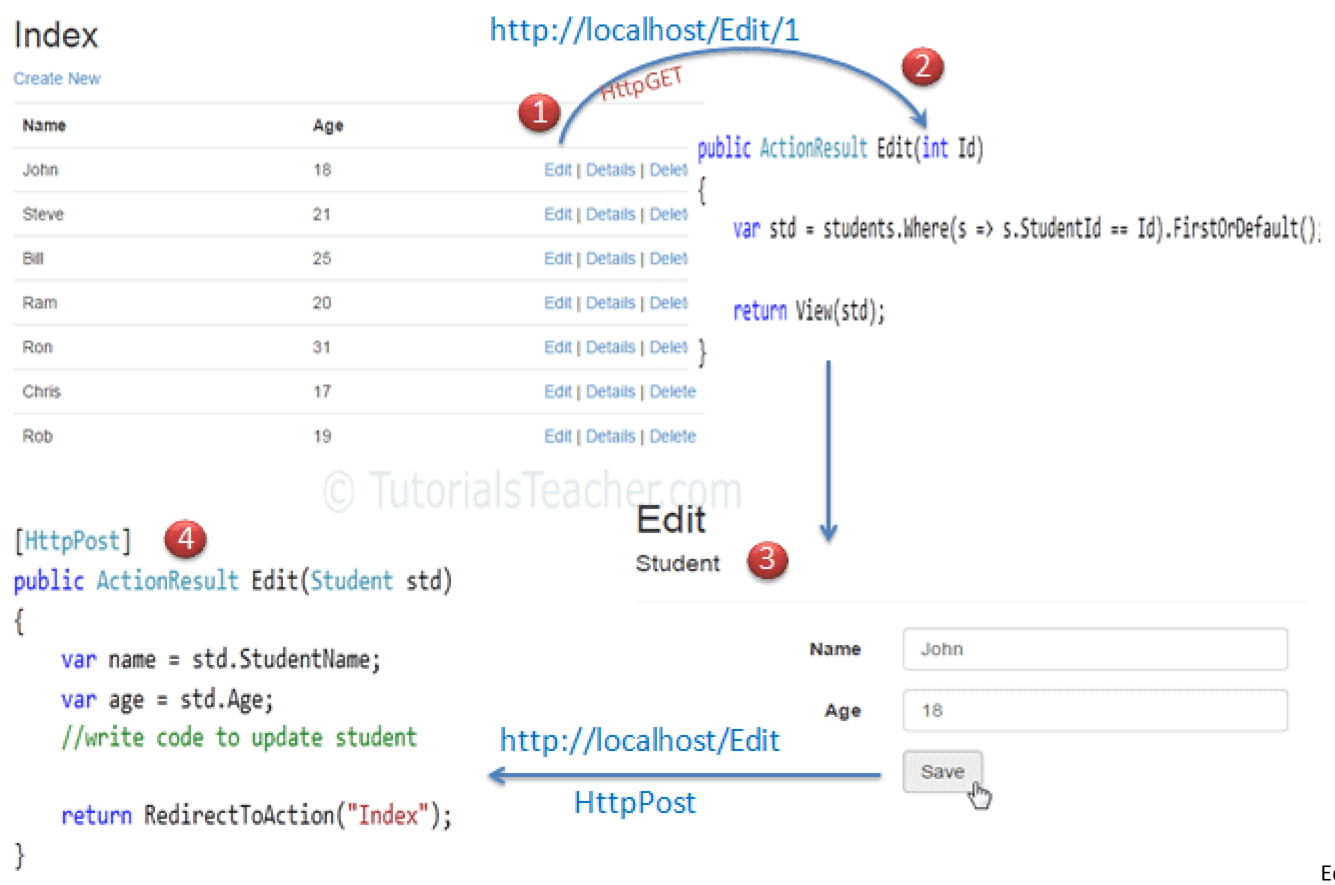
Save

Back to List

© 2018 - My ASP.NET Application

Edit View

The following figure describes how the edit functionality would work in ASP.NET MVC application.



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

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

    return View(std);
}
```

Name	John
Age	18

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

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

Editing

Steps in ASP.NET MVC App

The above figure illustrates the following steps.

1. The user clicks on the `Edit` link in the student list view, which will send the `HttpGET` request `http://localhost/student/edit/{Id}` with corresponding `Id` parameter in the query string. This request will be handled by the HttpGET action method `Edit()`. (by default action method handles the `HttpGET` request if no attribute specified)

2. The `HttpGet` action method `Edit()` 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 to edit the data using the `Edit` view in ASP.NET MVC.

So let's start to implement the above steps.

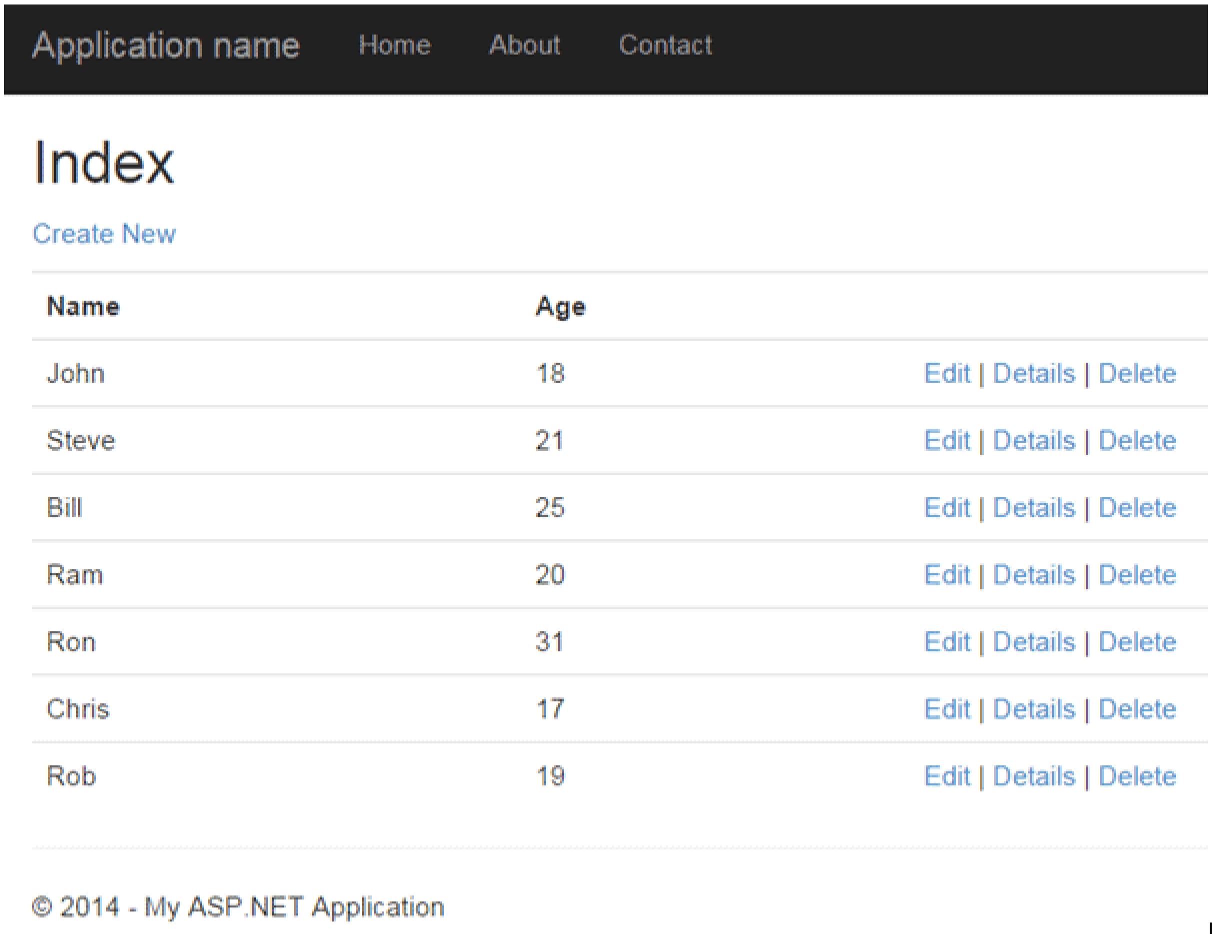
The following is the `Student` model class.

Example: Model Class

```
namespace MVCTutorials.Controllers

{
    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 the student list view in the [Create a View]() chapter, which includes the Edit action links for each `Student`, as shown below.



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

© 2014 - My ASP.NET Application

List View

In the above list view, edit links send `HttpGet` request to the `Edit()` action method of the `StudentController` with corresponding `StudentId` in the query string. For example, an edit link with a student `John` will append a `StudentId` to the request url because John's `StudentId` is `1` e.g. `http://localhost:<port number>/edit/1`.

Step 2:

Now, create a HttpGET action method `Edit(int  id)` in the `StudentController`, as shown below.

Example: HttpGet Edit() Action method - C#

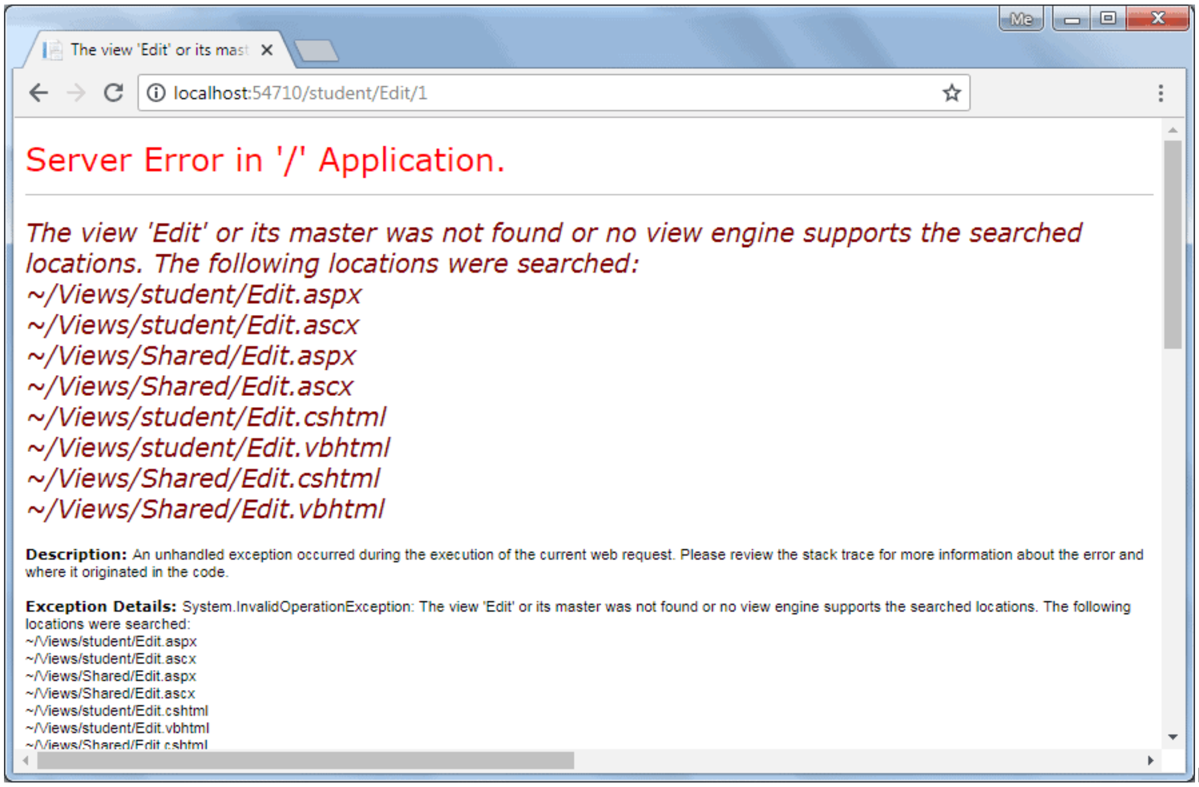
Copy

```
using MVCTutorials.Models;

namespace MVCTutorials.Controllers
{
    public class StudentController : Controller
    {
        static IList<Student> studentList = new List<Student>{
                new Student() { StudentId = 1, StudentName = "John", Age = 18 } ,
                new Student() { StudentId = 2, StudentName = "Steve",  Age = 21 } ,
                new Student() { StudentId = 3, StudentName = "Bill",  Age = 25 } ,
                new Student() { StudentId = 4, StudentName = "Ram" , Age = 20 } ,
                new Student() { StudentId = 5, StudentName = "Ron" , Age = 31 } ,
                new Student() { StudentId = 4, StudentName = "Chris" , Age = 17 } ,
                new Student() { StudentId = 4, StudentName = "Rob" , Age = 19 }
            };

        // GET: Student
        public ActionResult Index()
        {
            //fetch students from the DB using Entity Framework here

            return View(studentList.OrderBy(s => s.StudentId).ToList());
        }

        public ActionResult Edit(int Id)
        {
            //here, get the student from the database in the real application

            //getting a student from collection for demo purpose
            var std = studentList.Where(s => s.StudentId == Id).FirstOrDefault();

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

The HttpGet `Edit()` action method must perform two tasks. First, it should fetch a student data from the underlying data source, whose `StudentId` matches the parameter `Id`. Second, it should render the `Edit` view with the data, so that the user can edit it.

In the above `Edit()` action method, a LINQ query is used to get a `Student` from the `studentList` collection whose `StudentId` matches with the parameter `Id`, and then pass that `std` object into `View(std)` to populate the edit view with this data. In a real-life application, you can get the data from the database instead of the sample collection.

At this point, if you run the application and click on the `Edit` link in the student list view, then you will get the following error.



Server Error in '/' Application.

The view 'Edit' or its master was not found or no view engine supports the searched locations. The following locations were searched:
~/Views/student/Edit.aspx
~/Views/student/Edit.ascx
~/Views/Shared/Edit.aspx
~/Views/Shared/Edit.ascx
~/Views/student/Edit.cshtml
~/Views/student/Edit.vbhtml
~/Views/Shared/Edit.cshtml
~/Views/Shared/Edit.vbhtml

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.InvalidOperationException: The view 'Edit' or its master was not found or no view engine supports the searched locations. The following locations were searched:
~/Views/student/Edit.aspx
~/Views/student/Edit.ascx
~/Views/Shared/Edit.aspx
~/Views/Shared/Edit.ascx
~/Views/student/Edit.cshtml
~/Views/student/Edit.vbhtml
~/Views/Shared/Edit.cshtml

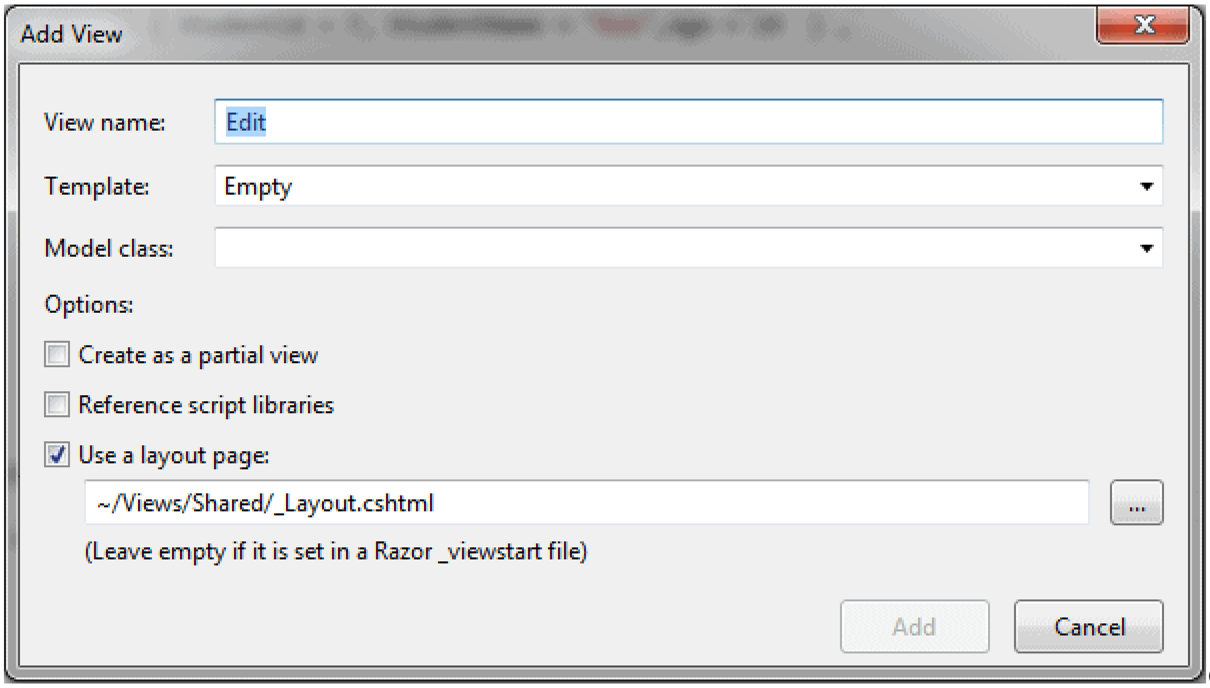
Edit View Error

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

ADVERTISEMENT

Step 3:

To create Edit view, right-click in the `Edit()` action method and click on **Add View...**. It will open Add View dialogue, as shown below.

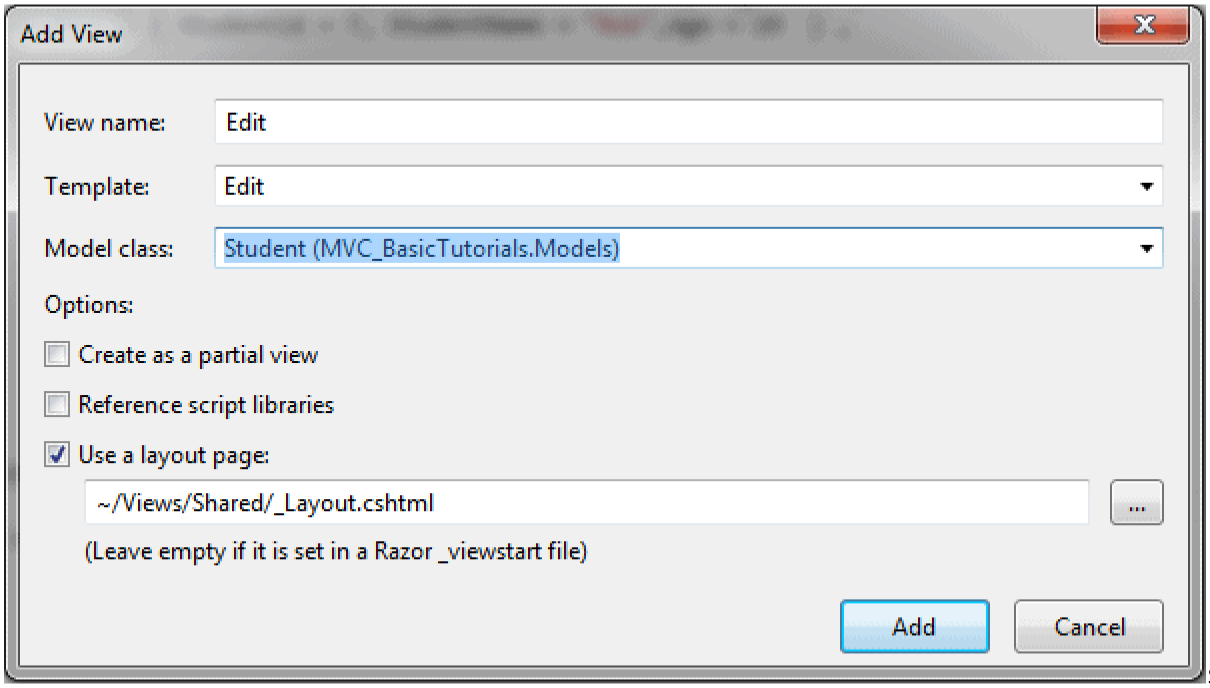


View name: Edit
Template: Empty
Model class:
Options:
☐ Create as a partial view
☐ Reference script libraries
☑ Use a layout page:
~/Views/Shared/_Layout.cshtml
(Leave empty if it is set in a Razor _viewstart file)

Create Edit View

In the Add View dialogue, keep the view name as `Edit`.

Select `Edit` Template and `Student` Model class from dropdown, as shown below.



View name: Edit

Template: Edit

Model class: Student (MVC_BasicTutorials.Models)

Options:

- [ ] Create as a partial view
- [ ] Reference script libraries
- [x] Use a layout page:

~/Views/Shared/_Layout.cshtml

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

Select Edit Template and Model

Click **Add** button to generate the `Edit.cshtml` view under **/View/Student** folder, as shown below.

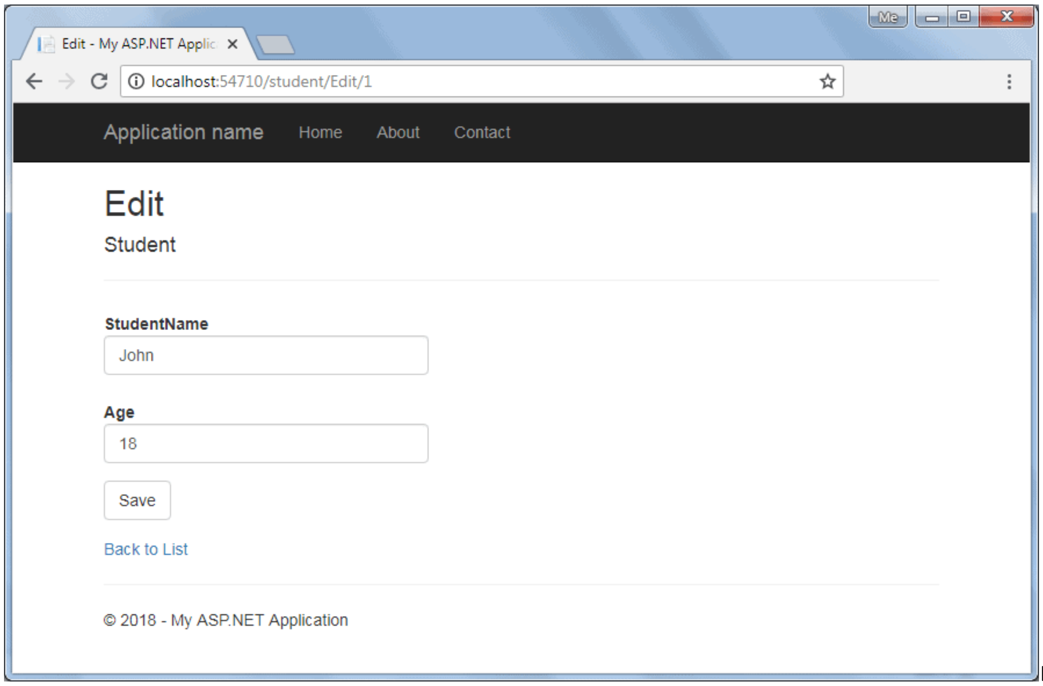
/View/Student/Edit.cshtml

Copy

```
@model MVCTutorials.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 the `HtmlHelper` method `Html.BeginForm()` to create the HTML form tag. `Html.BeginForm` sends a `HttpPost` request by default. This will display a `Student` data when you click an edit link in the student list view, as shown below.



Edit

Student

StudentName

John

Age

18

Save

Back to List

© 2018 - My ASP.NET Application

Edit View

You can now edit the data and click on the Save button. The Save button should send the HttpPOST request because we need to submit the form data as a part of the request body as a `Student` object.

Step 4:

Now, write HttpPost action method `Edit()` to save the edited student object, as shown below. So, there will be two `Edit()` action methods, HttpGet and HttpPost action methods.

Example: Controller Class in C#

Copy

```
using MVCTutorials.Models;

namespace MVCTutorials.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.OrderBy(s => s.StudentId).ToList());
        }

        public ActionResult Edit(int Id)
        {
            //here, get the student from the database in the real application

            //getting a student from collection for demo purpose
            var std = studentList.Where(s => s.StudentId == Id).FirstOrDefault();

            return View(std);
        }

        [HttpPost]
        public ActionResult Edit(Student std)
        {
            //update student in DB using EntityFramework in real-life application

            //update list by removing old student and adding updated student for demo purpose
            var student = studentList.Where(s => s.StudentId == std.StudentId).FirstOrDefault();
            studentList.Remove(student);
            studentList.Add(std);

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

In the above example, the HttpPost `Edit()` action method requires an object of the `Student` as a parameter. The `Edit()` view will bind the form's data collection to the student model parameter because it uses HTML helper methods `@Html.EditorFor()` for each properties to show input textboxes. Visit Model Binding section to know how MVC framework binds form data to action method parameter.

After updating the data in the DB, redirect back to the `Index()` action method to show the updated student list.

In this way, you can provide edit functionality using a default scaffolding Edit template.

Razor Syntax

Razor is one of the view engines supported in ASP.NET MVC. Razor allows you to write a 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 the following Characteristics:

- **Compact**: Razor syntax is compact, enabling you to minimize the number of characters and keystrokes required to write 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.

Inline expression

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

C# Razor Syntax

Copy

```
<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 lines of server-side code enclosed in braces `@{ ... }`. Each line must ends with a semicolon the same as C#.

Example: Server side Code in Razor Syntax

Copy

```
@{
    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

Copy

```
@{
    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

Copy

```
@{
    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!
```

ADVERTISEMENT

if-else condition

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

Example: if else in Razor

Copy

```
@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

Copy

```
@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

Copy

```
@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

Copy

```
@{
    string str = "";

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

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

Output:

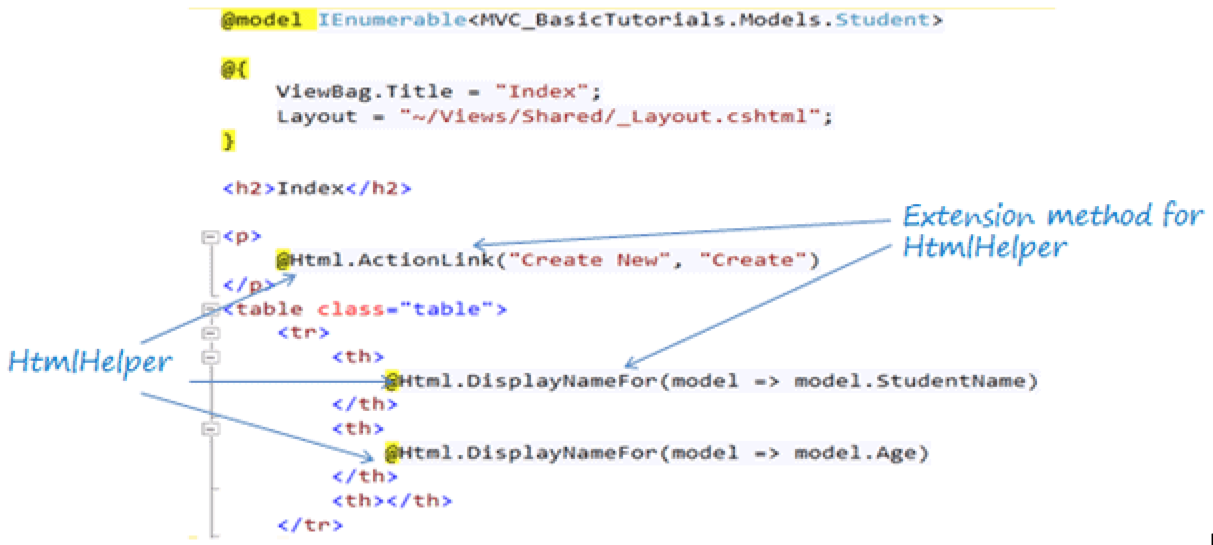
```
Hello World!
```

HTML Helpers

Here, you will learn what HTML helpers are and how to use them in the razor view.

The `HtmlHelper` class renders HTML controls in the razor view. It binds the model object to HTML controls to display the value of model properties into those controls and also assigns the value of the controls to the model properties while submitting a web form. So always use the `HtmlHelper` class in razor view instead of writing HTML tags manually.

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



```
@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>
```

HTML Helpers

In the above figure, **@Html** is an object of the `HtmlHelper` class. (@ symbol is used to access server-side object in razor syntax). Html is a property of the `HtmlHelper` class included in base class of razor view `WebViewPage`. The `ActionLink()` and `DisplayNameFor()` are extension methods included in the `HtmlHelper` class.

The `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 the `HtmlHelper` methods and HTML control each method renders.

Extension Method	Strongly Typed Method	Html Control
Html.ActionLink()	NA	<a></a>
Html.TextBox()	Html.TextBoxFor()	<input type="textbox">
Html.TextArea()	Html.TextAreaFor()	<input type="textarea">
Html.CheckBox()	Html.CheckBoxFor()	<input type="checkbox">
Html.RadioButton()	Html.RadioButtonFor()	<input type="radio">
Html.DropDownList()	Html.DropDownListFor()	<select> <option> </select>

Extension Method	Strongly Typed Method	Html Control
Html.ListBox()	Html.ListBoxFor()	multi-select list box: <select>
Html.Hidden()	Html.HiddenFor()	<input type="hidden">
Html.Password()	Html.PasswordFor()	<input type="password">
Html.Display()	Html.DisplayFor()	HTML text: ""
Html.Label()	Html.LabelFor()	<label>
Html.Editor()	Html.EditorFor()	Generates Html controls based on data type of specified model property e.g. textbox for string property, numeric field for int, double or other numeric type.

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.

Create a Textbox in ASP.NET MVC

The HtmlHelper class includes two extension methods `TextBox()` and `TextBoxFor<TModel, TProperty>()` that renders the HTML textbox control `<input type="text">` in the razor view.

It is recommended to use the generic `TextBoxFor<TModel, TProperty>()` method, which is less error prons and performs fast.

We will use the following `Student` model class throughout this article.

Example: Student Model

Copy

```
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; }
}
```

Html.TextBoxFor()

The `TextBoxFor<TModel, TProperty>()` is the generic extension method that creates `<input type="text">` control. The first type parameter is for the model class, and second type parameter is for the property.

TextBoxFor() Signature

Copy

```
public static MvcHtmlString TextBoxFor<TModel,TProperty> (this
HtmlHelper<TModel>> htmlHelper, Expression<Func<TModel,TProperty>> expression,
object htmlAttributes);
```

There are other overloads of the `TextBoxFor()` method. Visit docs.microsoft.com to know all the overloads of TextBoxFor() method.

The following example shows how to render a textbox for the `StudentName` property of the `Student` model.

Example: TextBoxFor() in Razor View

Copy

```
@model Student

@Html.TextBoxFor(m => m.StudentName)
```

In the above example, the lambda expression `m => m.StudentName` specifies the `StudentName` property to bind with a textbox. It generates an input text element with id and name attributes, as shown below.

Html Result:

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

The following example renders a textbox with the `class` attribute.

Example: TextBoxFor() in Razor View

Copy

```
@model Student

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

Html Result:

```
<input class="form-control" id="StudentName" name="StudentName" type="text"
value="" />
```

ADVERTISEMENT

Html.TextBox()

The `TextBox()` method creates <input type="text" > HTML control with the specified name, value, and other attributes.

TextBoxFor() Signature

Copy

```
public static MvcHtmlString TextBox(this HtmlHelper htmlHelper, string name,
string value, object htmlAttributes)
```

Visit docs.microsoft.com to know all the overloads of TextBox() method.

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

Example: Html.TextBox() in Razor View

Copy

```
@model Student

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

Html Result:

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

Create TextArea in ASP.NET MVC

The `HtmlHelper` class includes two extension methods to render multi-line `<textarea>` HTML control in a razor view: `TextArea()` and `TextAreaFor<TModel, TProperty>()`. By default, it creates a textarea with rows=2 and cols=20.

We will use the following `Student` model class throughout this article.

Example: Student Model

Copy

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

Html.TextAreaFor()

The `TextAreaFor<TModel, TProperty>()` is the generic extension method that creates `<textarea></textarea>` control.

It is recommended to use the generic `TextAreaFor<TModel, TProperty>()` method, which is less error prons and performs fast.

TextAreaFor() Signature

Copy

```
public static MvcHtmlString TextAreaFor<TModel,TProperty> (this HtmlHelper<TModel>> htmlHelper, Expression<Func<TModel,TProperty>> expression, object htmlAttributes);
```

Visit docs.microsoft.com to know all the overloads of TextAreaFor().

The following example creates and binds the `Description` property to a textarea control in the MVC view.

Example: TextAreaFor() in Razor View

```
@model Student

@Html.TextAreaFor(m => m.Description)
```

Html Result:

```
<textarea cols="20" id="Description" name="Description" rows="2"></textarea>
```

The following example renders a textarea with the `class` attribute.

Example: TextAreaFor() in Razor View

```
@model Student

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

In the above example, the first parameter `m => m.Description` is a lambda expression that specifies the model property to bind with the textarea element. The second parameter specifies the class attribute.

Html Result:

```
<textarea class="form-control" cols="20" id="Description" name="Description" rows="2"></textarea>
```

ADVERTISEMENT

Html.TextArea()

The `Html.TextArea()` method creates a `<textarea>` HTML control with specified name, value and html attributes.

The `TextArea()` method is a loosely typed method because the name parameter is a string. The name parameter can be a property name of the model class.

Example: Html.TextArea() in Razor View

Copy

```
@model Student

@Html.TextArea("Description", "This is dummy description.", new { @class = "form-control" })
```

Html Result:

```
<textarea class="form-control" id="Description" name="Description" rows="2"cols="20">This is dummy description.</textarea>
```

Create Checkbox in ASP.NET MVC

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

We will use the following `Student` model class throughout this article.

Example: Student Model

Copy

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

Html.CheckBoxFor()

The `CheckBoxFor<TModel, TProperty>()` extension method generates `<input type="checkbox">` control for the model property specified using a lambda expression.

Visit docs.microsoft.com to know all the overloads of CheckBoxFor() method.

Example: Html.CheckBoxFor() in Razor View

Copy

```
@model Student

@Html.CheckBoxFor(m => m.isActive)
```

Html Result:

```
<input data-val="true"
        data-val-required="The isActive field is required."
        id="isActive"
        name="isActive"
        type="checkbox"
        value="true" />

<input name="isActive" type="hidden" value="false" />
```

In the above example, the first parameter is a lambda expression that specifies the model property to bind with the checkbox element. We have specified isActive property in the above example.

Notice that it has generated an additional hidden field with the same name and `value=false`. When you submit a form with a checkbox, the value is posted only if a checkbox is checked. So, if you leave the checkbox unchecked, then nothing will be sent to the server. Sometimes, you would want `false` to be sent to the server. Because, an hidden input has the same name, it will send `false` to the server if checkbox is unchecked.

Html.CheckBox()

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

Example: Html.CheckBox() in Razor View

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

Html Result:

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

Create Radio buttons in ASP.NET MVC

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

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

We will use the following `Student` model class throughout this article.

Example: Student Model

Copy

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

Html.RadioButtonFor()

The `Html.RadioButtonFor<TModel, TProperty>()` extension method is a strongly typed extension method. It generates `<input type="radio">` control for the property specified using a lambda expression.

Visit docs.microsoft.com to know all the overloads of RadioButtonFor().

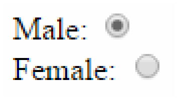
Example: Html.RadioButtonFor() in Razor View

```
@model Student


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

Html Result:

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

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

In the above example, the first parameter is a lambda expression that specifies the model property to be bind with a radio button control. We have created two radio buttons for the `Gender` property in the above example. So, it generates two `<input type="RadioButton">` controls with id and name set to property name `Gender`. The second parameter is a value that will be sent to the server when the form is submitted, here `Male` will be sent if the first radio button selected, and `Female` will be sent if the second radio button selected.



ADVERTISEMENT

RadioButton()

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

Visit docs.microsoft.com to know all the overloads of RadioButton() method.

Example: Html.RadioButton() in Razor View

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

Html Result:

```
Male: <input checked="checked"
        id="Gender"
        name="Gender"
        type="radio"
        value="Male" />

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

Create DropdownList in ASP.NET MVC

Learn how to generate the dropdownlist HTML control using the `HtmlHelper` in a razor view.

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

We will use the following `Student` model class and `Gender` enum.

Example: Student Model

Copy

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

public enum Gender
{
    Male,
    Female
}
```

Html.DropDownListFor()

The `Html.DropDownListFor<TModel,TProperty>` extension method is a strongly typed extension method generates `<select>` element for the property specified using a lambda expression.

Visit docs.microsoft.com to know all the overloads of DropDownListFor method.

The following example creates dropdown list for the above `StudentGender` property.

Example: Html.DropDownListFor() in Razor View

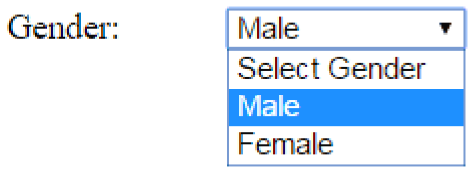
Copy

```
@using MyMVCApp.Models

@model Student

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

Html Result:

```
<select class="form-control" id="StudentGender" name="StudentGender">
    <option>Select Gender</option>
    <option>Male</option>
    <option>Female</option>
</select>
```

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



ADVERTISEMENT

Html.DropDownList()

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

Visit docs.microsoft.com to know all the overloads of DropDownList() method.

Example: Html.DropDownList() in Razor View

Copy

```
@using MyMVCApp.Models

@model Student

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

Html Result:

```
<select class="form-control" id="StudentGender" name="StudentGender">
    <option>Select Gender</option>
    <option>Male</option>
    <option>Female</option>
</select>
```

In the above example, the first parameter is a property name for which we want to display list items. The second parameter is a list of values to be included in the dropdown list. We have used Enum methods to get the `Gender` values. The third parameter is a label, which will be the first list item, and the fourth parameter is for HTML attributes like CSS to be applied on the dropdownlist.

Create a Hidden Field in ASP.NET MVC

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

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

We will use the following `Student` model class throughout this article.

Example: Student Model

Copy

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

Html.HiddenFor()

The `Html.HiddenFor<TModel, TProperty>` extension method is a strongly typed extension method generates a hidden input element for the model property specified using a lambda expression.

Visit docs.microsoft.com to know all the overloads of HiddenFor() method.

Example: HiddenFor() in Razor View

Copy

```
@model Student

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

Html Result:

```
<input data-val="true"
            data-val-number="The field StudentId must be a number."
            data-val-required="The StudentId field is required."
            id="StudentId"
            name="StudentId"
            type="hidden"
            value="" />
```

In the above example, the first parameter in `HiddenFor()` method is a lambda expression that specifies the model property to be bind with the hidden field. We have specified the `StudentId` property in the above example. So, it generates an input text element with id & name set to the property name. The value attribute will be set to the value of the `StudentId` property.

Please notice that it has created `data-` HTML5 attribute, which is used for the validation in ASP.NET MVC.

ADVERTISEMENT

Html.Hidden()

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

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

Example: Html.Hidden() in Razor View

Copy

```
@model Student

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

Html Result:

```
<input id="StudentId"
            name="StudentId"
            type="hidden"
            value="1" />
```

Create Password field in ASP.Net MVC

The `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 User model with Password() and PasswordFor() method.

Example: User Model

Copy

```
public class User
{
    public int UserId { get; set; }
    public string Password { get; set; }
}
```

Html.PasswordFor()

The `Html.PasswordFor<TModel,TProperty>()` extension method is a strongly typed extension method. It generates a `<input type="password">` element for the model object property specified using a lambda expression.

Visit docs.microsoft.com to know all the overloads of PasswordFor() method.

Example: PasswordFor() in Razor View

Copy

```
@model User

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

Html Result:

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

In the above example, the first parameter in `PasswordFor()` method is a lambda expression that specifies the model property to be bind with the password textbox. We have specified the `Password` property. It generates the following result.

ADVERTISEMENT

Html.Password()

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

Visit docs.microsoft.com to know all the overloads of Password() method.

Example: Html.Password() in Razor View

```
 @model User


@Html.Password("Password")
```

Html Result:

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

HtmlHelper - Display HTML String

Learn how to create html string literal using the `HtmlHelper` class in razor view.

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

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

Example: Student Model

Copy

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

Html.DisplayFor()

The `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

Copy

```
@model Student

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

Html Result:

```
"Steve"
```

In the above example, we have specified `StudentName` property of Student model using lambda expression in the DisplayFor() method. So, it generates a html string with the StudentName value, `Steve`, in the above example.

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)`

Visit docs.microsoft.com to know all the [overloads of Display() method]()

Example: Html.Display() in Razor View

Copy

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

Html Result:

```
"Steve"
```

Create Label in ASP.Net MVC

The `HtmlHelper` class includes two extension methods to generate HTML label element: `Label()` and `LabelFor()`.

We will use the following `Student` [model class]().

Example: Student Model

Copy

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

Html.LabelFor()

The `Html.LabelFor<TModel,TProperty>()` helper method is a strongly typed extension method. It generates a html label element for the model object property specified using a lambda expression.

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

Example: LabelFor() in Razor View

Copy

```
@model Student

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

Html Result:

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

In the above example, we have specified the `StudentName` property using a lambda expression in the `LabelFor()` method. The `Display` attribute on the `StudentName` property will be used as a label.

ADVERTISEMENT

Label()

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

Visit MSDN to know all the [overloads of Label() method]()

Example: Html.Label() in Razor View

Copy

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

Html Result:

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

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

Example: Html.Label() in Razor View

Copy

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

Html Result:

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

Create HTML Controls for Model Class Properties using EditorFor()

ASP.NET MVC includes the method that generates HTML input elements based on the datatype. The `Html.Editor()` or `Html.EditorFor()` extension methods generate HTML elements based on the data type of the model object's property.

The following table list the data types and releted HTML elements:

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.

Example: Student Model

Copy

```
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; }
}
```

Html.EditorFor()

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

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

Example: EditorFor() in Razor view

Copy

```
@model Student

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

Html Result:

```
StudentId:      <input data-val="true" data-val-number="The field StudentId must
be a number." data-val-required="The StudentId field is required." id="StudentId"
name="StudentId" type="number" value="" />

Student Name:   <input id="StudentName" name="StudentName" type="text" value=""
/>

Age:            <input data-val="true" data-val-number="The field Age must be a
number." data-val-required="The Age field is required." id="Age" name="Age"
type="number" value="" />

Password:       <input id="Password" name="Password" type="text" value="" />

isNewlyEnrolled:<input class="check-box" data-val="true" data-val-required="The
isNewlyEnrolled field is required." id="isNewlyEnrolled" name="isNewlyEnrolled"
type="checkbox" value="true" />

                <input name="isNewlyEnrolled" type="hidden" value="false" />

DoB:            <input data-val="true" data-val-date="The field DoB must be a
date." data-val-required="The DoB field is required." id="DoB" name="DoB"
type="datetime" value="" />
```

In the above exampl, MVC framework generates an appropriate control based on the data type of a property, e.g. textbox for string type property, number field for int type property, checkbox for boolean property, etc.

Html.Editor()

The `Html.Editor()` method requires a string parameter to specify the property name. It creats a HTML element based on the datatype of the specified property, same as `EditorFor()` method.

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")
```

Exception Handling in ASP.NET MVC

Here you will learn how to handle exceptions in ASP.NET MVC application.

You may handle all possible exceptions in the action methods using try-catch blocks. However, there can be some unhandled exceptions that you want to log and display custom error messages or custom error pages to users.

When you create an MVC application in Visual Studio, it does not implement any exception handling technique out of the box. It will display an error page when an exception occurred.

For example, consider the following action method that throws an exception.

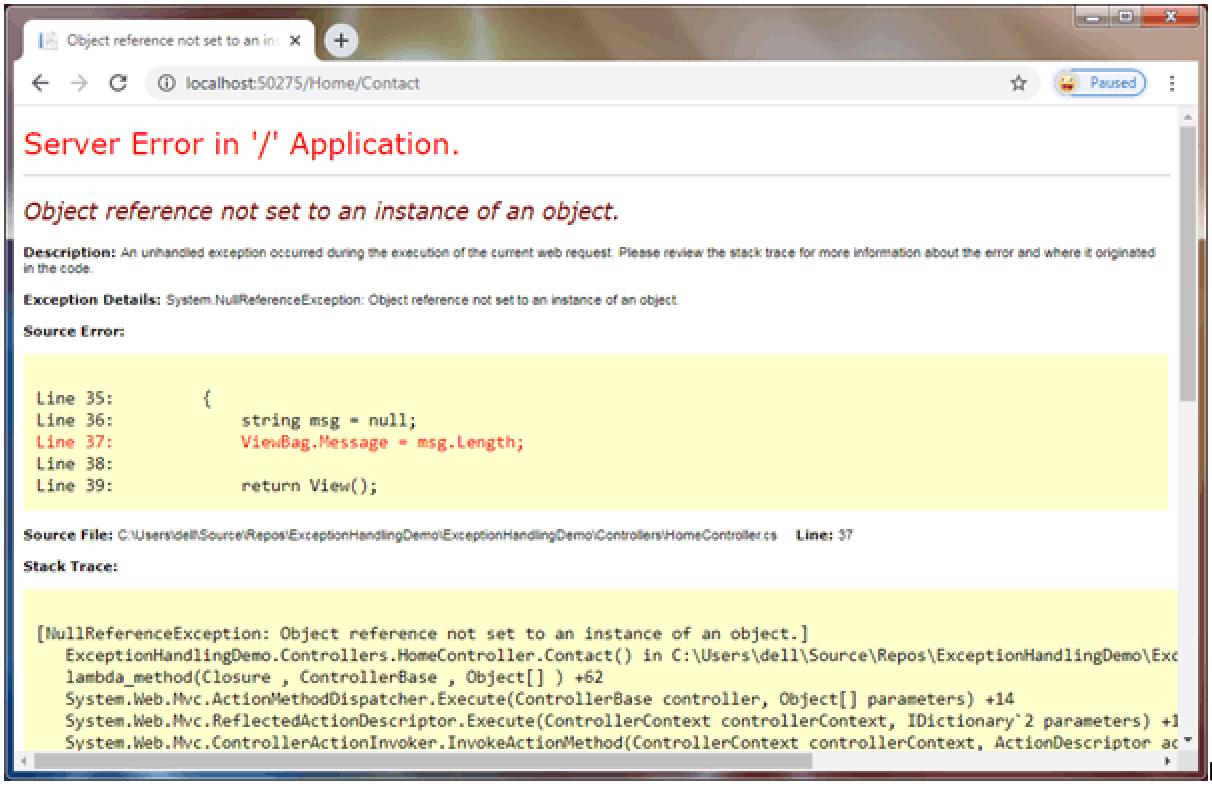
Example: Action Method

Copy

```
namespace ExceptionHandlingDemo.Controllers
```

```
{
    public class HomeController : Controller
    {
        public ActionResult Contact()
        {
            string msg = null;
            ViewBag.Message = msg.Length; // this will throw an exception

            return View();
        }
}
```

Navigating to `/home/contact` in the browser, and you will see the following yellow page (also known as the Yellow Screen of Death) that shows exception details such as exception type, line number and file name where the exception occurred, and stack trace.


Object reference not set to an in
localhost:50275/Home/Contact
Paused

Server Error in '/' Application.

Object reference not set to an instance of an object.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Exception Details: System.NullReferenceException: Object reference not set to an instance of an object.

Source Error:

```
Line 35:         {
Line 36:             string msg = null;
Line 37:             ViewBag.Message = msg.Length;
Line 38:
Line 39:             return View();
```

Source File: C:\Users\dell\Source\Repos\ExceptionHandlingDemo\ExceptionHandlingDemo\Controllers\HomeController.cs **Line:** 37

Stack Trace:

```
[NullReferenceException: Object reference not set to an instance of an object.]
   ExceptionHandlingDemo.Controllers.HomeController.Contact() in C:\Users\dell\Source\Repos\ExceptionHandlingDemo\Exc
   lambda_method(Closure , ControllerBase , Object[] ) +62
   System.Web.Mvc.ActionMethodDispatcher.Execute(ControllerBase controller, Object[] parameters) +14
   System.Web.Mvc.ReflectedActionDescriptor.Execute(ControllerContext controllerContext, IDictionary`2 parameters) +1
   System.Web.Mvc.ControllerActionInvoker.InvokeActionMethod(ControllerContext controllerContext, ActionDescriptor ac
```

Default Error Page in MVC

ASP.NET provides the following ways to handle exceptions:

1. Using `<customErrors>` element in web.config
2. Using `HandleErrorAttribute`
3. Overriding `Controller.OnException` method
4. Using `Application_Error` event of `HttpApplication`

<customErrors> Element in web.config

The `<customErrors>` element under `system.web` in web.config is used to configure error code to a custom page. It can be used to configure custom pages for any error code 4xx or 5xx. However, it cannot be used to log exception or perform any other action on exception.

Enable the `<customErrors>` in web.config, as shown below.

Example: Enable customErrors

Copy

```
<system.web>
    <customErrors mode="On"></customErrors>
</system.web>
```

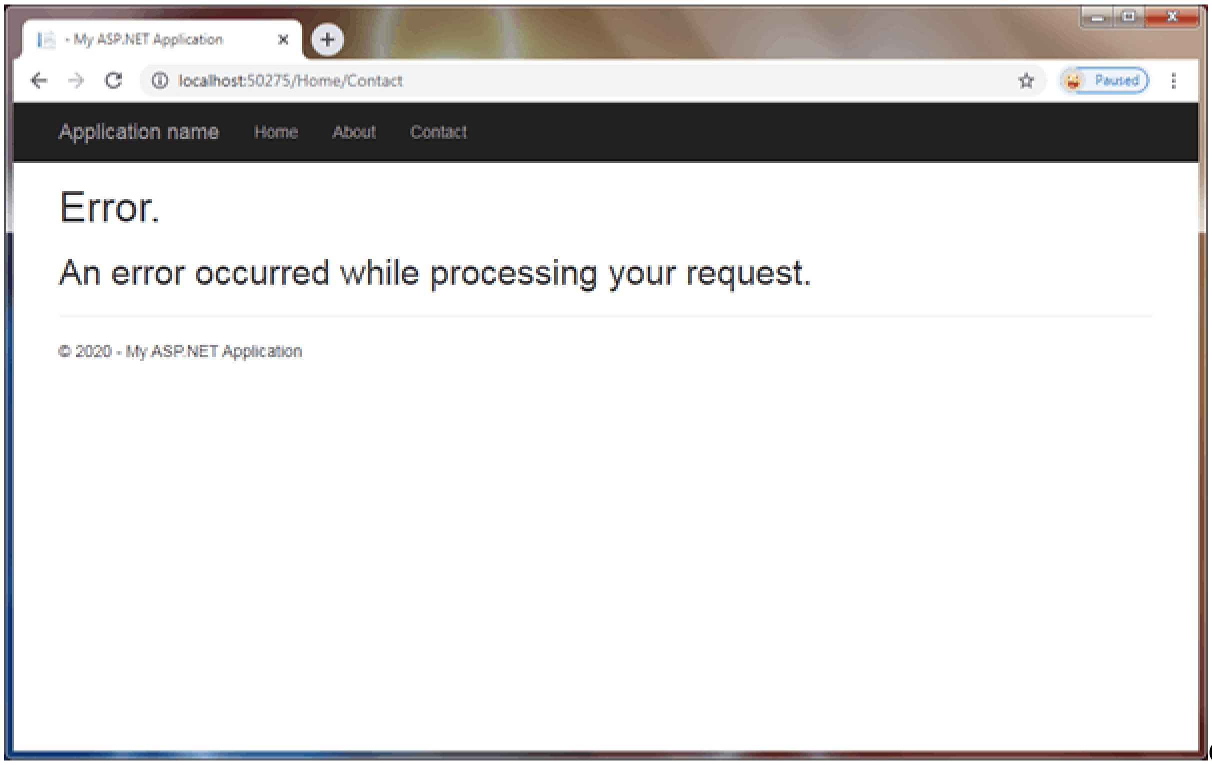
You also need to add `HandleErrorAttribute` filter in the `FilterConfig.cs` file.

Example: Add HandleErrorAttribute Filter

Copy

```
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

After enabling the customErrors mode to On, an ASP.NET MVC application will show the default custom error page, as shown below.



Error.

An error occurred while processing your request.

© 2020 - My ASP.NET Application

Custom Error Page

The above view is Error.cshtml in the Shared folder. It will be displayed on the 500 error code.

The `HandleErrorAttribute` filter set the Error.cshtml as the default view to display on an error occurred.

Learn more about handling exceptions using web.config customErrors in ASP.NET MVC.

ADVERTISEMENT

HandleErrorAttribute

The HandleErrorAttribute is an attribute that can be used to handle exceptions thrown by an action method or a controller. You can use it to display a custom view on a specific exception occurred in an action method or in an entire controller.

Note:

The HandleErrorAttribute attribute can only be used to handle the exception with status code 500. Also, it does not provide a way to log exceptions.

In order to use this attribute, you must add `HandleErrorAttribute` filter in the `FilterConfig.RegisterGlobalFilters()` method and also, set the mode attribute to On `<customErrors mode="On">` in web.config, as we did for the `customErrors` section above.

Now, let's apply [HandleError] attribute to the action method, as shown below.

Example: HandleErrorAttribute

Copy

```
public class HomeController : Controller
{
    [HandleError]
    public ActionResult Contact()
    {
        string msg = null;
        ViewBag.Message = msg.Length;

        return View();
    }
}
```

Above, we configured [HandleError] attribute on the Contact() action method. It will display Error.cshtml view from the Shared folder when an exception occurs. The [HandleError] set the Error.cshtml view as default view for any exceptions.

the [HandleError] can also be used to configure different pages for different types of exceptions, as shown below.

Example: Configure Views for Exceptions

Copy

```
public class HomeController : Controller
{
    [HandleError]
    [HandleError(ExceptionType        =typeof(NullReferenceException),        View
="~/Views/Error/NullReference.cshtml")]
    public ActionResult Contact()
    {
        string msg = null;
        ViewBag.Message = msg.Length;

        return View();
    }
}
```

Now, the above example will show NullReference.cshtml because it throws NullReferenceException.

The [HandleError] attribute has a limited scope and not recommended to use in most cases.

Overriding Controller.OnException Method

Another way to handle controller level exceptions is by overriding the `OnException()` method in the controller class. This method handles all your unhandled errors with error code 500.

It allows you to log an exception and redirect to the specific view. It does not require to enable the `<customErrors>` config in web.config.

Example: Handle Exceptions in the Controller

Copy

```
public class HomeController : Controller
{
    public ActionResult Contact()
    {
        string msg = null;
        ViewBag.Message = msg.Length;

        return View();
    }

    protected override void OnException(ExceptionContext filterContext)
    {
        filterContext.ExceptionHandled = true;

        //Log the error!!

        //Redirect to action
        filterContext.Result = RedirectToAction("Error", "InternalError");

        // OR return specific view
        filterContext.Result = new ViewResult
        {
            ViewName = "~/Views/Error/InternalError.cshtml"
        };
   }
}
```

Using Application_Error event of HttpApplication

The ideal way to log exception occurred in any part of your MVC application is to handle it in the Application_Error event in the global.asax file.

Example:

Copy

```
public class MvcApplication : System.Web.HttpApplication
{
    //other code removed for clarity

    protected void Application_Error()
    {
        var ex = Server.GetLastError();
        //log an exception
    }
}
```

The `Application_Error` event is fired on any type of exception and error codes. So, handle it carefully.

Recommendation

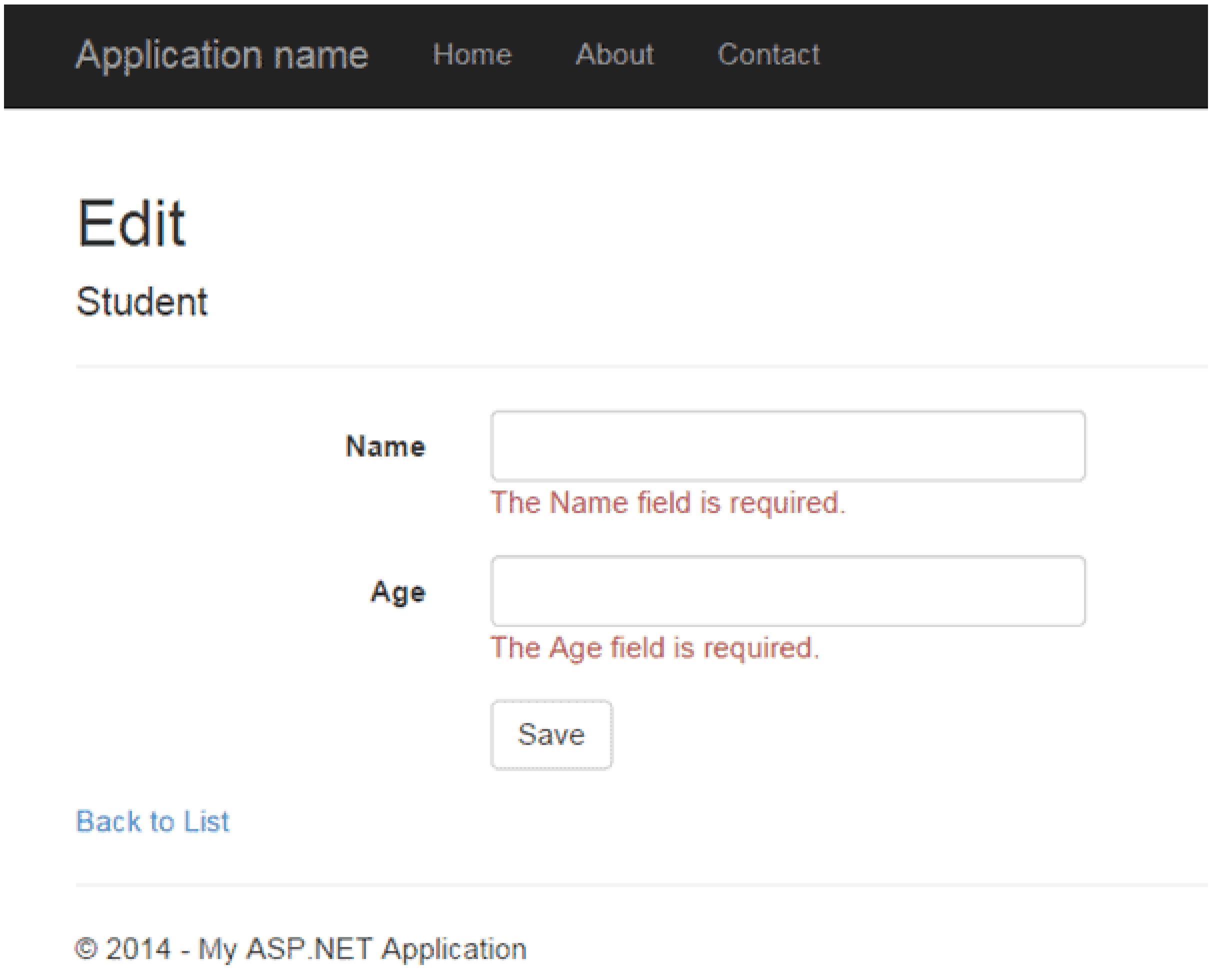
In most web applications, you should ideally log the exceptions and also show appropriate error messages or pages to the users. So, it is recommended to use the global `Application_Error` event to log all the exceptions along with `<customErrors>` element in web.config to redirect it to appropriate pages.

The above exception handling techniques will return the response with 200 status code. If you are concern to return specific error code in response then you have to use `<httpErrors>` element in web.config.

Implement Data Validation in MVC

Here, you will learn how to implement the data validation and display validation messages on the violation of business rules in an ASP.NET MVC application.

The following image shows how the validation messages will be displayed if `Name` or `Age` fields are blank while creating or editing data.



Edit

Student

Name

The Name field is required.

Age

The Age field is required.

© 2014 - My ASP.NET Application

Validation using Data Annotation Attributes

ASP.NET MVC includes built-in attribute classes in the System.ComponentModel.DataAnnotations namespace. These attributes are used to define metadata for ASP.NET MVC and ASP.NET data controls. You can apply these attributes to the properties of the model class to display appropriate validation messages to the users.

The following table lists all the data annotation attributes which can be used for validation.

Attribute	Usage
Required	Specifies that a property value is required.
StringLength	Specifies the minimum and maximum length of characters that are allowed in a string type property.
Range	Specifies the numeric range constraints for the value of a property.
RegularExpression	Specifies that a property value must match the specified regular expression.

Attribute	Usage
CreditCard	Specifies that a property value is a credit card number.
CustomValidation	Specifies a custom validation method that is used to validate a property.
EmailAddress	Validates an email address.
FileExtension	Validates file name extensions.
MaxLength	Specifies the maximum length of array or string data allowed in a property.
MinLength	Specifies the minimum length of array or string data allowed in a property.
Phone	Specifies that a property value is a well-formed phone number.

Let's see how to use these attributes to display validation messages on the view.

The following is the `Student` model class.

Example: Apply DataAnnotation Attributes

Copy

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

We want to implement validations for `StudentName` and `Age` property values. We want to make sure that users do not save empty `StudentName` or `Age` value. Also, age should be between 10 to 20.

The `Required` attribute is used to specify that the value cannot be empty. The `Range` attribute is used to specify the range of values a property can have. We will use the `Required` attribute on the `StudentName` to make it mandatory for the user to provide value and `Range` attribute to make sure the user enters value between 10 to 20, as shown below.

Example: Apply DataAnnotation Attributes

Copy

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

    [Range(10, 20)]
    public int Age { get; set; }
}
```

The above attributes define the metadata for the validations of the `Student` class. This alone is not enough for the validation. You need to check whether the submitted data is valid or not in the controller. In other words, you need to check the model state.

ADVERTISEMENT

Use the `ModelState.IsValid` to check whether the submitted model object satisfies the requirement specified by all the data annotation attributes. The following POST action method checks the model state before saving data.

Example: Edit Action methods:

Copy

```
public class StudentController : Controller
{
    public ActionResult Edit(int id)
    {
        var stud = ... get the data from the DB using Entity Framework

        return View(stud);
    }

    [HttpPost]
    public ActionResult Edit(Student std)
    {
        if (ModelState.IsValid) { //checking model state

            //update student to db

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

Now, create an edit view as shown here. The following is a generated edit view using the default scaffolding template.

Edit View: Edit.cshtml

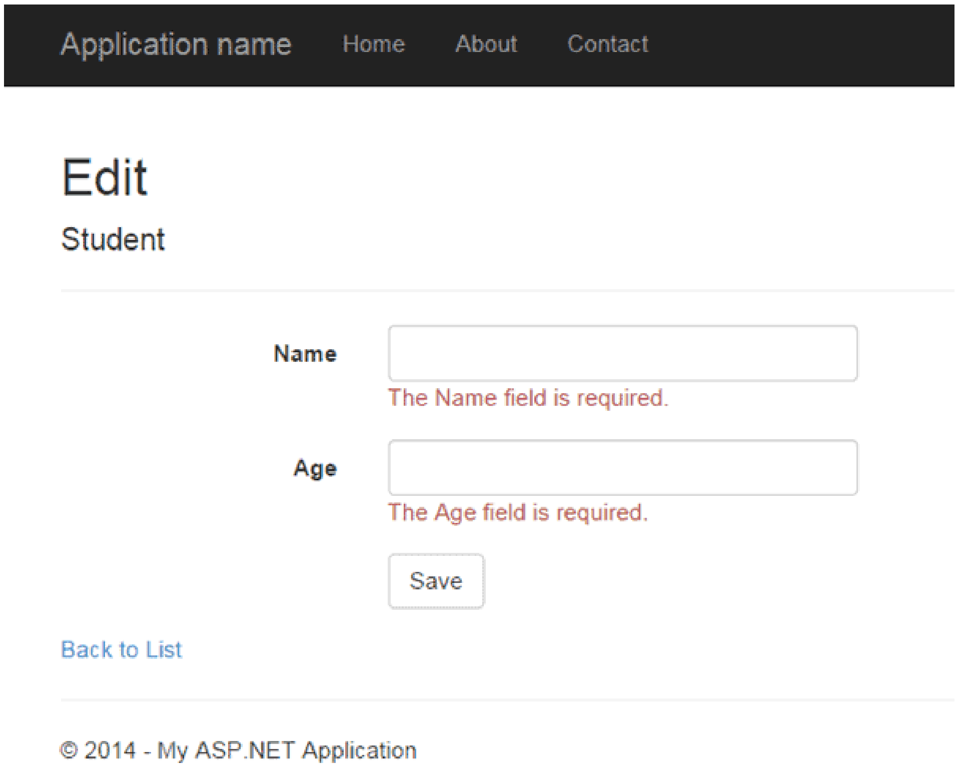
Copy

```
@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>
```

In the above view, it calls the HTML Helper method **ValidationMessageFor()** for every field and **ValidationSummary()** method at the top. The ValidationMessageFor() displays an error message for the specified field. The ValidationSummary() displays a list of all the error messages for all the fields.

In this way, you can display the default validation message when you submit a form without entering `StudentName` or `Age`, as shown below.



Edit

Student

Name

The Name field is required.

Age

The Age field is required.

© 2014 - My ASP.NET Application

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.

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

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

The following `Student` model class with the `Required` validation attribute on the `StudentName`.

Example: Student Model

Copy

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

The following view uses the `ValidationMessageFor()` method for the `StudentName`.

Example: ValidationMessageFor

Copy

```
@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 the `ValidationMessageFor()` method is a lambda expression to specify a property for which we want to show an error message. The second parameter is for custom error message if any, and the third parameter is for HTML attributes such as CSS, style, etc.

The above code will generate the following HTML when you run it.

Html Result:

Copy

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

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

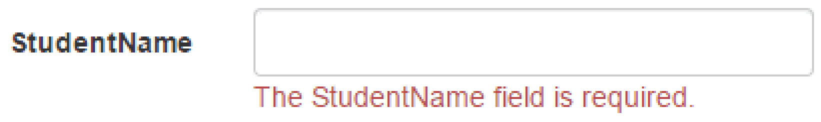
Now, when the user submits a form without entering a `StudentName` then ASP.NET MVC uses the data- attribute of HTML5 for the validation and the default validation message will be injected when validation error occurs, as shown below.

Html with Validation message:

Copy

```
<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.



Custom Error Message

You can display custom error messages instead of the default error message as above. You can provide a custom error message either in the data annotation attribute or in the `ValidationMessageFor()` method.

Use the `ErrorMessage` parameter of the data annotation attribute to provide your own custom error message, as shown below.

Example: Custom error message in the Model

Copy

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

You can also 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" })
```

It is recommended to use `ValidationMessageFor()` than `ValidationMessage()` because it is strongly typed and so performs fast and less error pron.

ASP.NET MVC: ValidationSummary

The ValidationSummary() extension method displays a summary of all validation errors on a web page as an unordered list element. It can also be used to display custom error messages.

The `ValidationMessageFor` displays an error message for an individual field, whereas the `ValidationSummary` displays all the error messages.

Consider the following `Student` model class with the `Required` and `Range` validation attributes.

Example: Student Model

Copy

```
public class Student
{
    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }
    [Range(10, 20)]
    public int Age { get; set; }
}
```

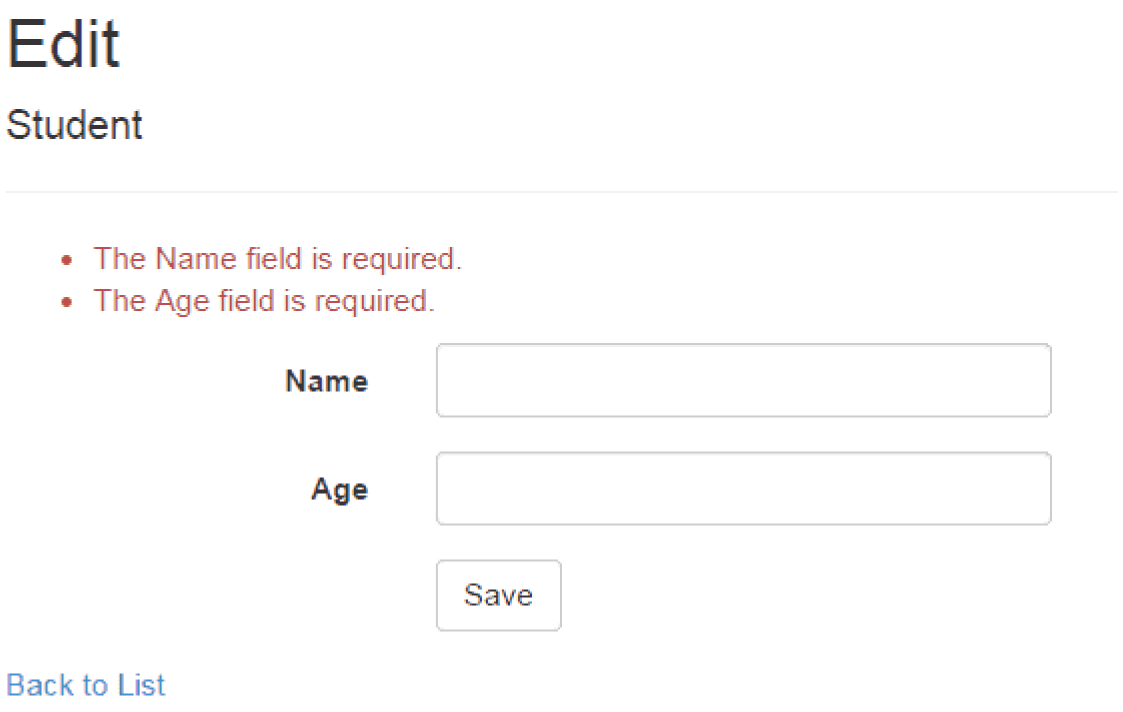
The following view uses the `ValidationSummary()` method to display all the error messages.

Example: ValidationMessageFor

Copy

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

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

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

Above, the first parameter of the `ValidationSummary()` is false, so it will display the field level errors as a summary. The second parameter is for the message. We don't want to provide a message there so specify an empty string. The third parameter is for HTML attributes such as CSS class for messages. The above will display the error messages as a summary shown below.


Edit

Student

- The Name field is required.
- The Age field is required.

Display Custom Error Messages

You can also display a custom error message using `ValidationSummary`.

Here, we will display a message if a student's name already exists in the database. So, in the HTTP Post action method, check the name in the database and add error message in the `ModelState` dictionary if the name already exists, as shown below.

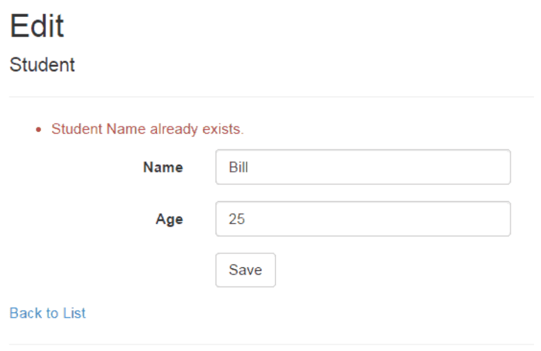
Example: Edit Action methods:

Copy

```
public class StudentController : Controller
{
    public ActionResult Edit(int id)
    {
        var stud = ... get the data from the DB using Entity Framework

        return View(stud);
    }

    [HttpPost]
    public ActionResult Edit(Student std)
    {
        if (ModelState.IsValid) { //checking model state
```

```
            //check whether name is already exists in the database or not
            bool nameAlreadyExists = * check database *

            if(nameAlreadyExists)
            {
                //adding error message to ModelState
                ModelState.AddModelError("name", "Student Name Already Exists.");

                return View(std);
            }

            return RedirectToAction("Index");
        }

        return View(std);
    }
}
```

Above, we added a custom error message using the `ModelState.AddModelError()` method. The `ValidationSummary()` method will automatically display all the error messages added into the `ModelState`.


Edit

Student

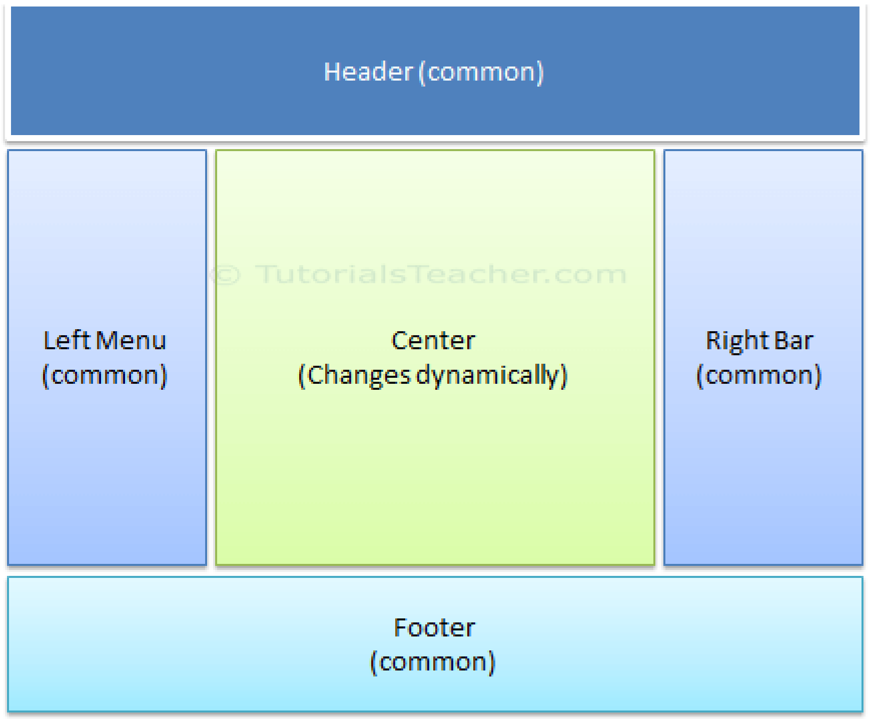
- Student Name already exists.

What is Layout View in ASP.NET MVC

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

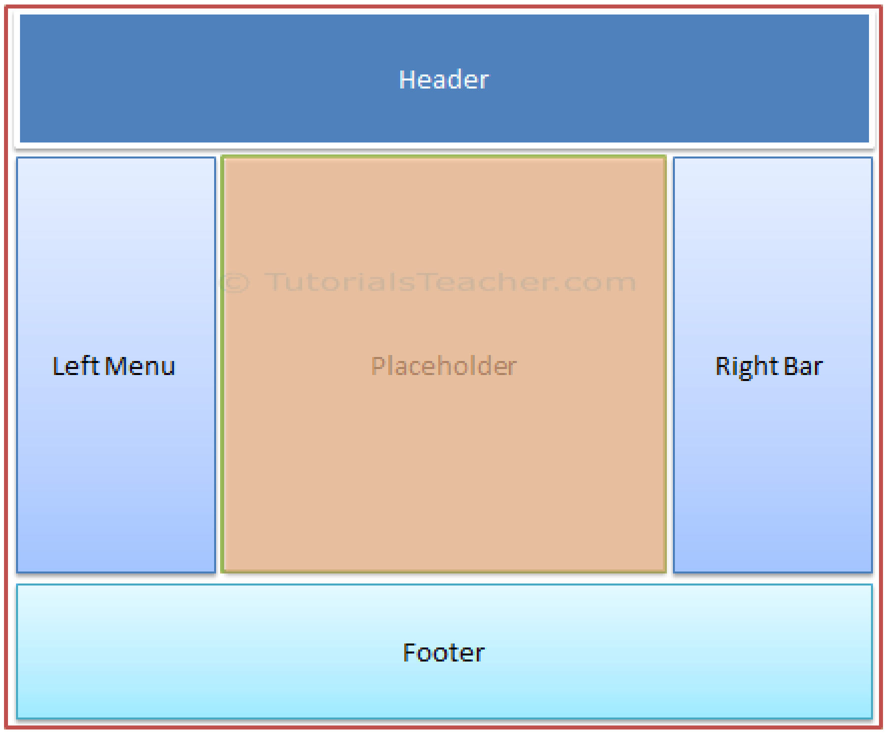
An application may contain a specific UI portion that remains the same throughout the application, such as header, left navigation bar, right bar, or footer section. ASP.NET MVC introduced a Layout view which contains these common UI portions so that we don't have to write the same code in every page.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

For example, an application UI may contain a header, left menu bar, right bar, and footer section that remains the same on every page. Only the center section changes dynamically, as shown below.



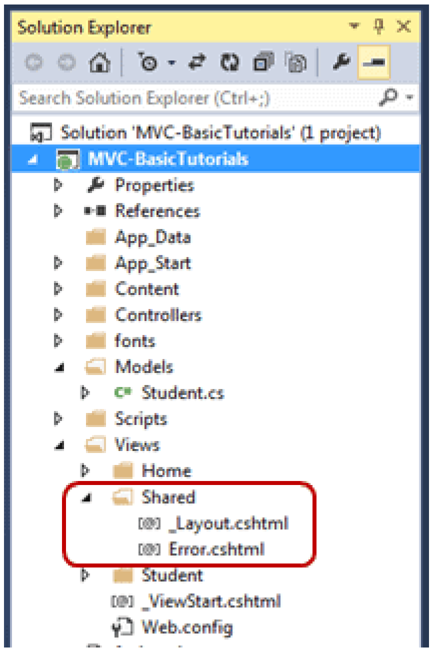
Sample Application UI Parts

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



Layout View

The layout view has the same extension as other views, .cshtml or .vbhtml. Layout views are shared with multiple views, so it must be stored in the `Shared` folder. By default, a layout view `_Layout.cshtml` is created when you [Create MVC application]() using Visual Studio, as shown below.



Layout Views in Shared Folder

The following is the default `_Layout.cshtml`.

_Layout.cshtml:

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

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

As you can see, the layout view contains HTML Doctype, head, and body tags. The only difference is a call to `RenderBody()` and `RenderSection()` methods. The child views will be displayed where the `RenderBody()` is called.

ADVERTISEMENT

Using Layout View

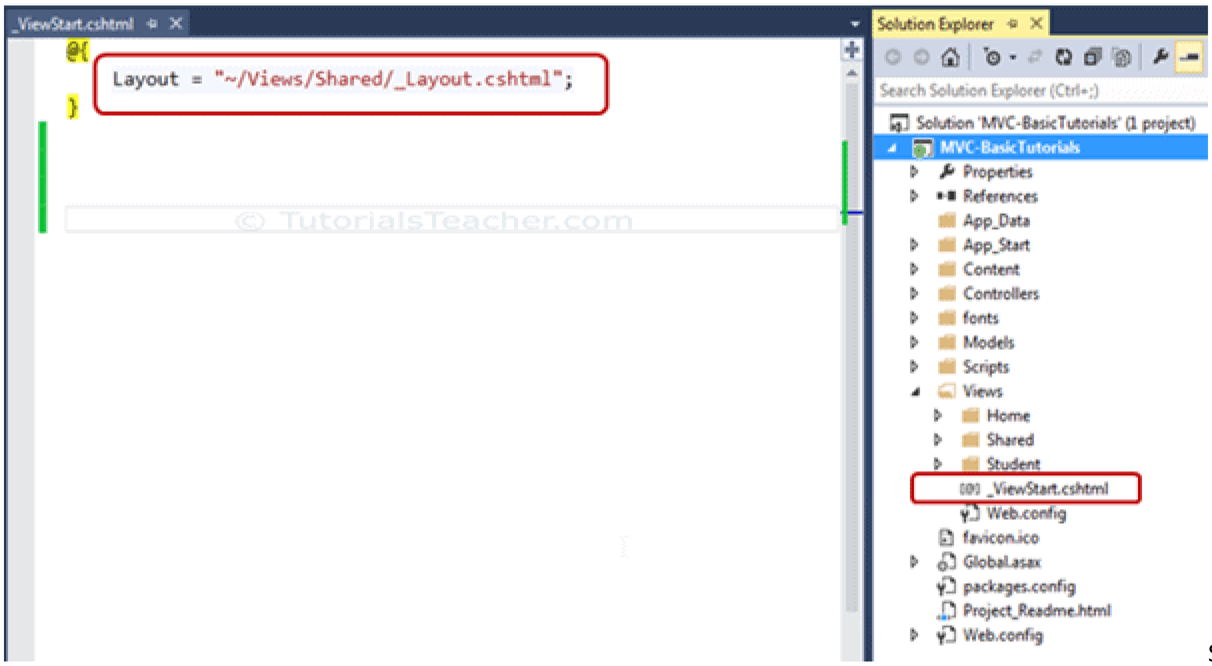
The views which will be displayed in a placeholder `RenderBody()` are called child views. There are multiple ways to specify which layout view will be used with which child views. You can specify it in a common `_ViewStart.cshtml`, in a child view, or in an action method.

ViewStart

The default `_ViewStart.cshtml` is included in the `Views` folder. It can also be created in all other `Views` sub-folders. It is used to specify common settings for all the views under a folder and sub-folders where it is created.

Set the `Layout` property to a particular layout view will be applicable to all the child views under that folder and its sub-folders.

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

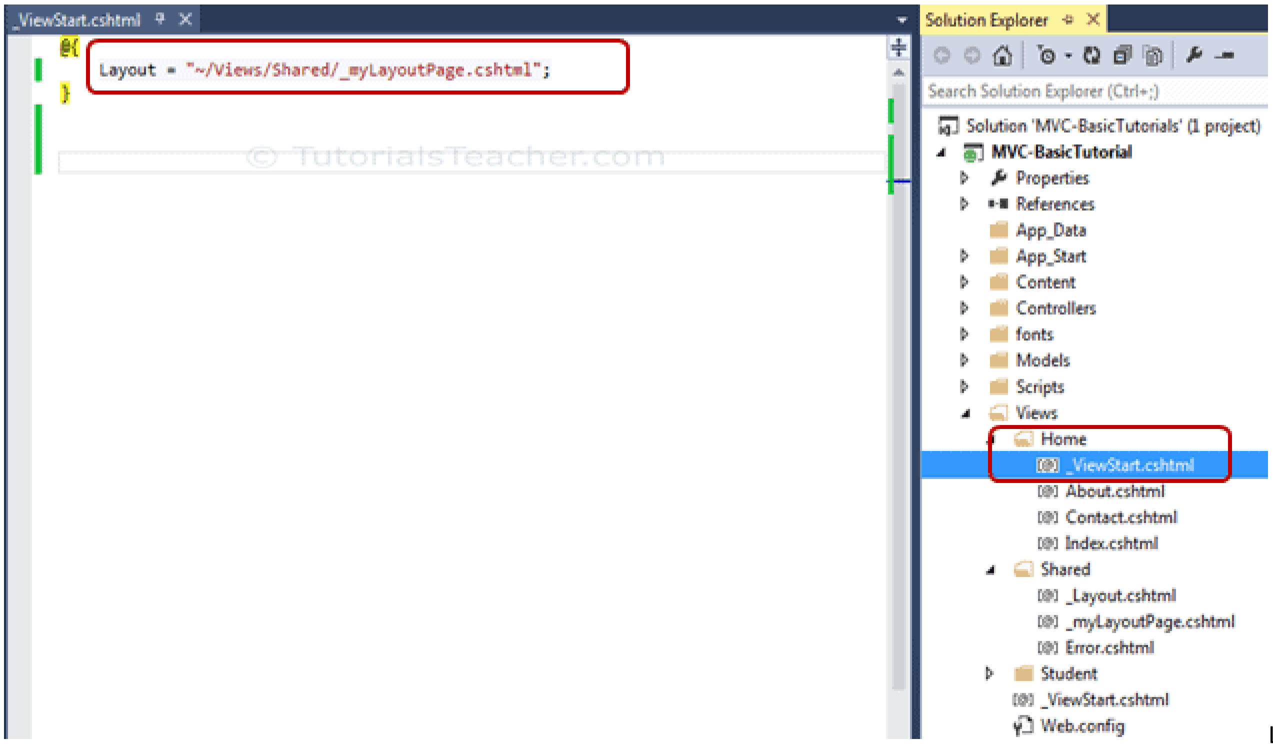


```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Setting Layout View in _ViewStart.cshtml

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

For example, the following `_ViewStart.cshtml` in the `Home` folder sets the `Layout` property to `_myLayoutPage.cshtml`. So now, `Index.cshtml`, `About.cshtml` and `Contact.cshtml` will display in the `_myLayoutPage.cshtml` instead of default `_Layout.cshml`.



```
@{
    Layout = "~/Views/Shared/_myLayoutPage.cshtml";
}
```

Layout View in Sub-folders

Specify Layout View in a Child View

You can also override the default layout view setting of `_ViewStart.cshtml` by setting the `Layout` property in each child view. For example, the following `Index.cshtml` view uses the `_myLayoutPage.cshtml` even if `_ViewStart.cshtml` sets the `_Layout.cshtml`.

Index.cshtml

Copy

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

<div class="jumbotron">
    <h1>ASP.NET</h1>
    <p class="lead">ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.</p>
    <p><a href="http://asp.net" class="btn btn-primary btn-lg">Learn more &raquo;</a></p>
</div>

<div class="row">
    <div class="col-md-4">
        <h2>Getting started</h2>
        <p>
            ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that
            enables a clean separation of concerns and gives you full control over markup
            for enjoyable, agile development.
        </p>
        <p><a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301865">Learn more &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Get more libraries</h2>
        <p>NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.</p>
        <p><a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301866">Learn more &raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Web Hosting</h2>
        <p>You can easily find a web hosting company that offers the right mix of features and price for your applications.</p>
        <p><a class="btn btn-default" href="http://go.microsoft.com/fwlink/?LinkId=301867">Learn more &raquo;</a></p>
    </div>
</div>
```

Specify Layout Page in Action Method

Specify the layout view name as a second parameter in the `View()` method, as shown below. By default, layout view will be searched in the `Shared` folder.

Example: Specify Layout View in Action Method

Copy

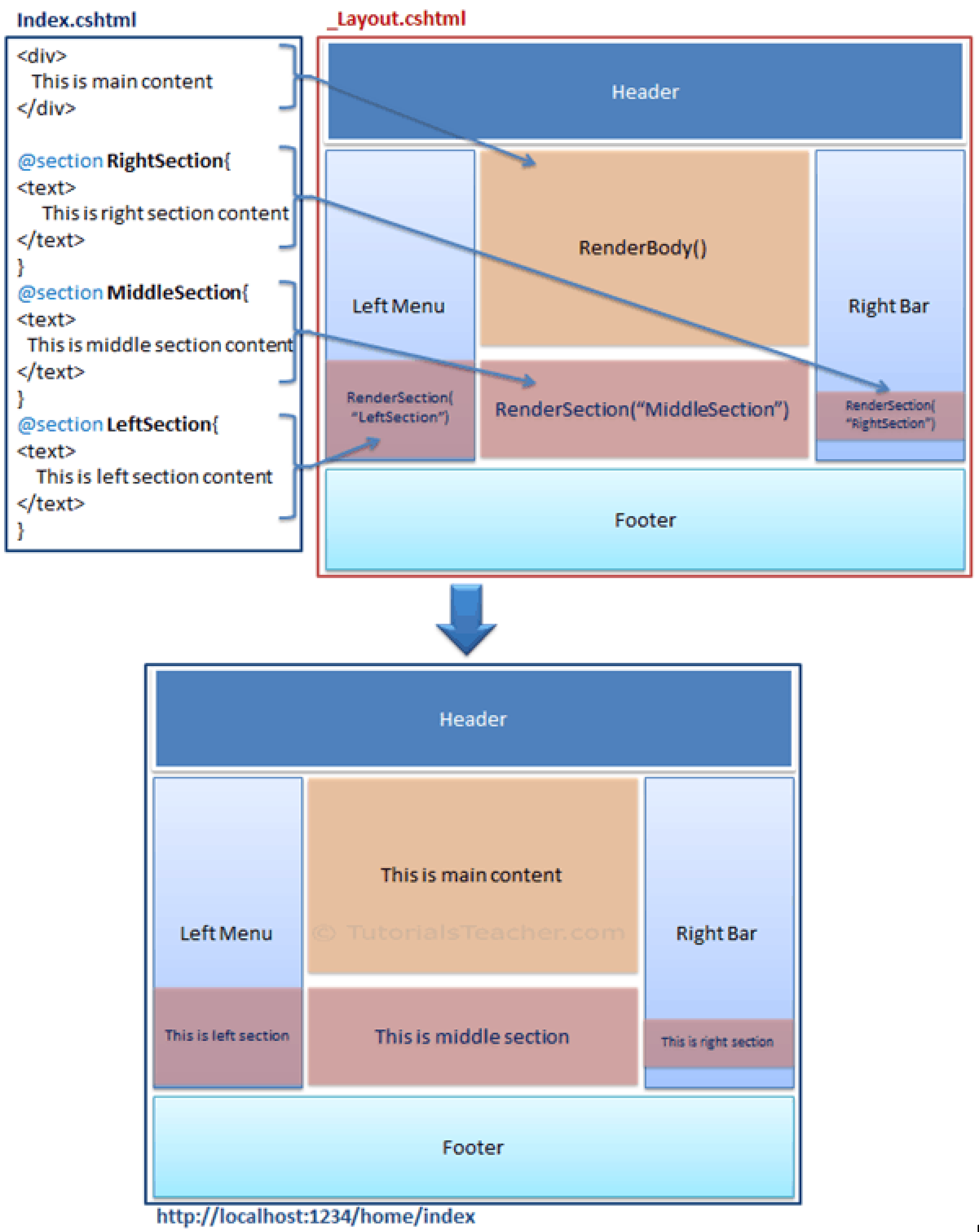
```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View("Index", "_myLayoutPage"); //set "_myLayoutView" as layout view
    }
}
```

Rendering Methods

ASP.NET MVC layout view renders child views using the following methods.

Method	Description
RenderBody()	Renders the portion of the child view that is not within a named section. Layout view must include the `RenderBody()` method.
RenderSection(string name)	Renders a content of named section and specifies whether the section is required.

The following figure illustrates the use of the `RenderBody()` and `RenderSection()` methods.



```
<div>
  This is main content
</div>

@section RightSection{
<text>
    This is right section content
</text>
}
@section MiddleSection{
<text>
  This is middle section content
</text>
}
@section LeftSection{
<text>
    This is left section content
</text>
}
```

Rendering Methods

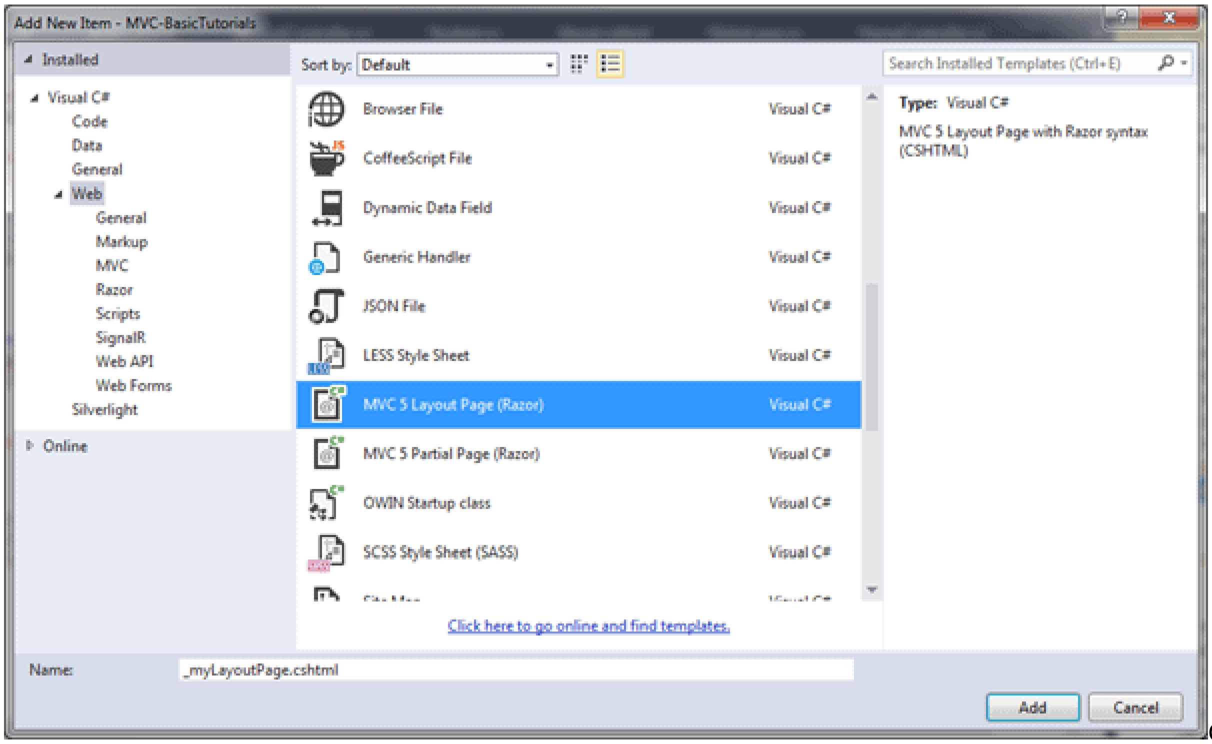
As you can see in the above figure, the `_Layout.cshtml` includes the `RenderBody()` method and `RenderSection()` method. Above, `Index.cshtml` contains the named sections using `@section` where the name of each section matches the name specified in the `RenderSection()` method of a layout view `_Layout.cshtml`, e.g. `@Section RightSection`. At run time, the named sections of `Index.cshtml`, such as `LeftSection`, `RightSection`, and `MiddleSection` will be rendered at appropriate place where the `RenderSection()` method is called. The rest of the `Index.cshtml` view, which is not in any of the named section, will be rendered in the `RenderBody()` is called.

Create a Layout View

You learned what is the layout view in ASP.NET MVC. Here you will learn how to create a layout view using Visual Studio.

You can create a layout view in any folder under the `Views` folder. However, it is recommended to create all the layout views in the `Shared` folder for easy maintenance purpose.

To create a new layout view in Visual Studio, right-click on the `Shared` folder -> select Add -> click on **New Item...**. This will open the **Add New Item** popup, as shown below.


Add New Item - MVC-BasicTutorials
Installed
Visual C#
Code
Data
General
Web
General
Markup
MVC
Razor
Scripts
SignalR
Web API
Web Forms
Silverlight
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Browser File Visual C#
CoffeeScript File Visual C#
Dynamic Data Field Visual C#
Generic Handler Visual C#
JSON File Visual C#
LESS Style Sheet Visual C#
MVC 5 Layout Page (Razor) Visual C#
MVC 5 Partial Page (Razor) Visual C#
OWIN Startup class Visual C#
SCSS Style Sheet (SASS) Visual C#
Type: Visual C#
MVC 5 Layout Page with Razor syntax (CSHTML)
Click here to go online and find templates.
Name: _myLayoutPage.cshtml
Add
Cancel
Create Layout View

In the **Add New Item** dialogue box, select `MVC 5 Layout Page (Razor)` template, and specify a layout view name as `_myLayoutPage.cshtml` and click **Add** to create it as shown below. Prefixing the underscore _ before layout view name is a common naming convention in ASP.NET MVC.

_myLayoutPage.cshtml

Copy

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
```

```
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

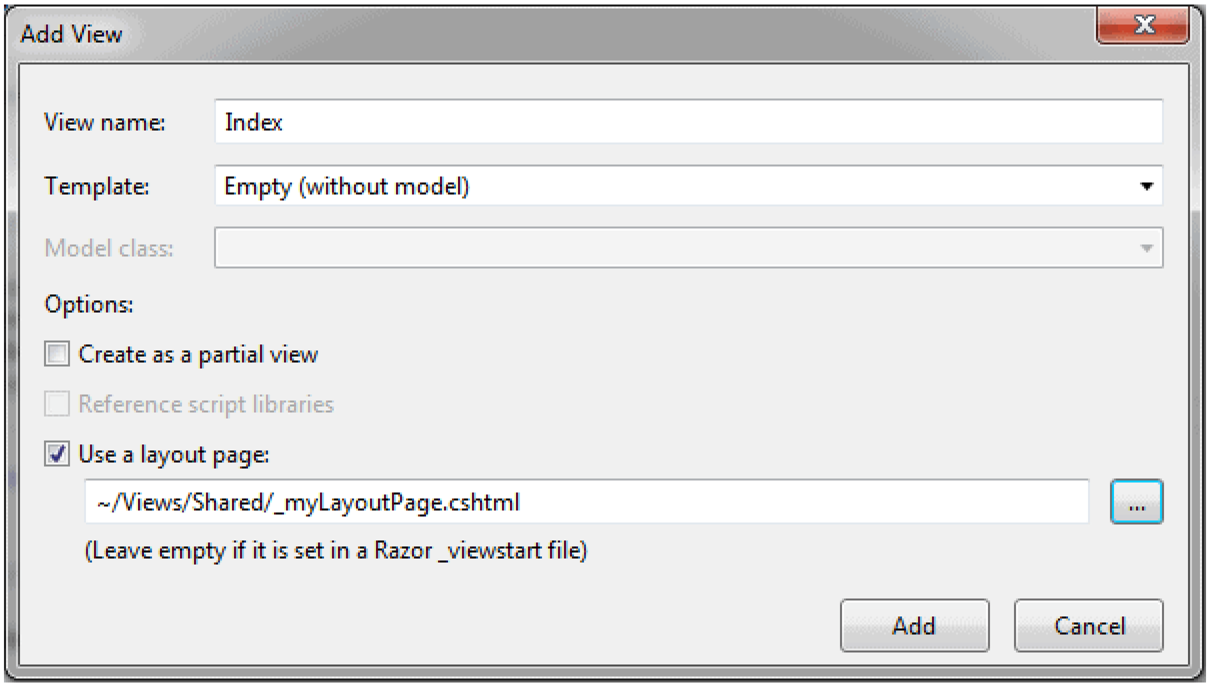
Now, let's add the common `<footer>` tag with the `RenderSection("footer",true)` method, as shown below. Please notice that we made this section as required. It means any view that uses the `_myLayoutPage` as its layout view must include a footer section.

Example: Adding RenderSection

Copy

```
<!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, [create a new child view]() and select `_myLayoutPage.cshtml` as a layout view, as shown below.



View name: Index

Template: Empty (without model)

Model class:

Options:

☐ Create as a partial view

☐ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_myLayoutPage.cshtml

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

This will create a new `Index.cshtml` as shown below.

Index.cshtml

Copy

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

<h2>Index</h2>
```

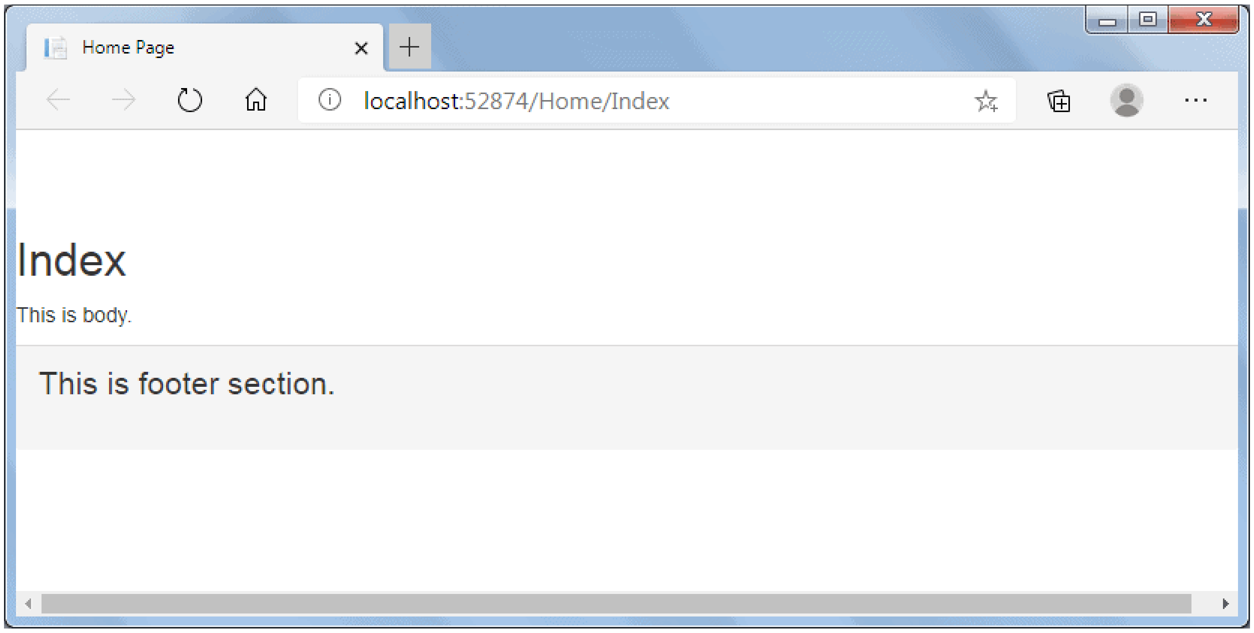
Now, add the footer section because `_myLayoutPage.cshtml` contains the mandatory footer section, as shown below.

Index.cshtml

Copy

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

<h2>Index</h2>
<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 that the `Index` view will be displayed in the `RenderBody()` method, and the footer section will be displayed in the `RenderSection("footer", true)`, as shown below.



Index

This is body.

This is footer section.

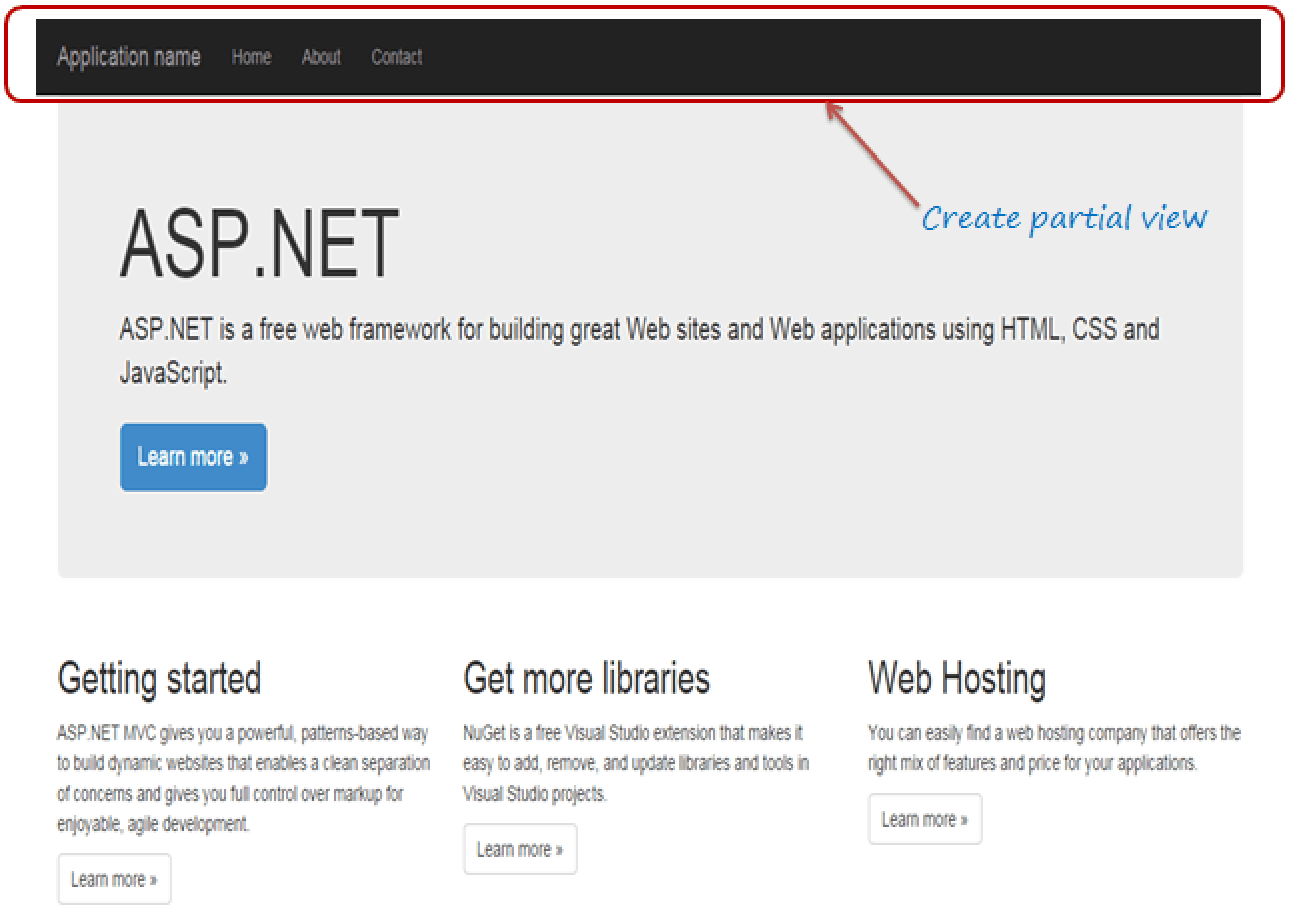
Thus, you can create a new layout view with a body and different sections.

Create and Render Partial Views

Here you will learn what is a partial view and how to use it in the ASP.NET MVC application.

A partial view is a reusable portion of a web page. It is `.cshtml` or `.vbhtml` file that contains HTML code. It can be used in one or more Views or Layout Views. You can use the same partial view at multiple places and eliminates the redundant code.

Let's create a partial view for the following menu, so that we can use the same menu in multiple layout views without rewriting the same code everywhere.



Application name Home About Contact

Create partial view

ASP.NET

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

Learn more »

Getting started

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

Learn more »

Get more libraries

NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.

Learn more »

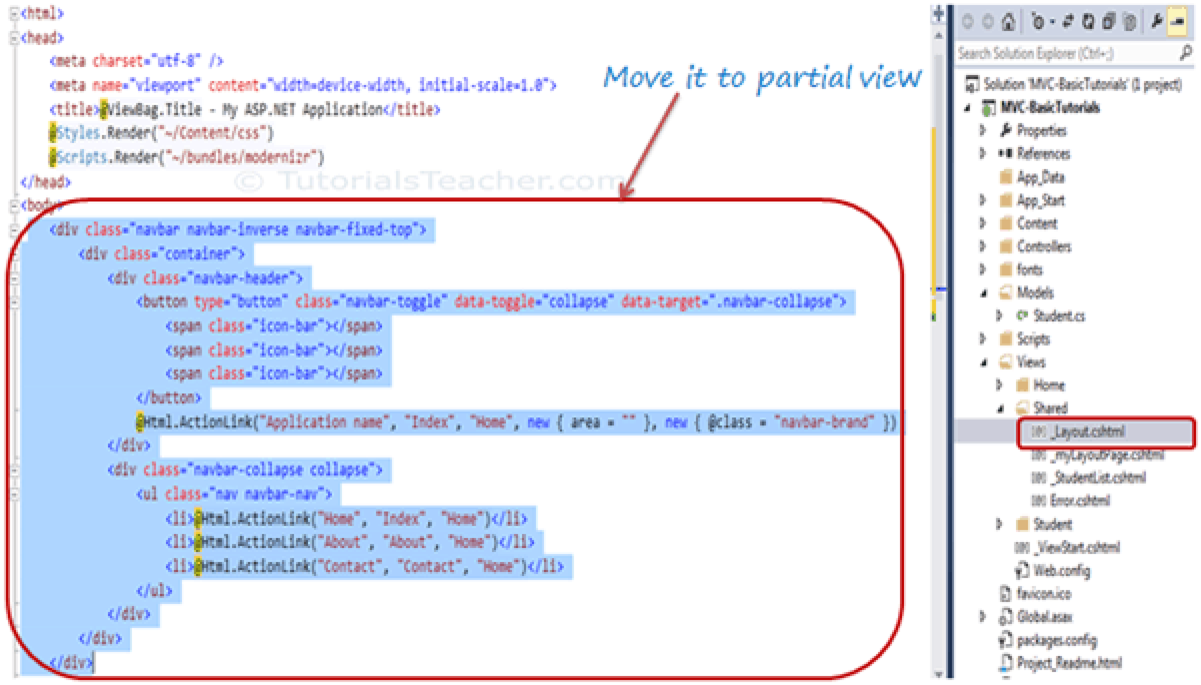
Web Hosting

You can easily find a web hosting company that offers the right mix of features and price for your applications.

Learn more »

Partial View

We created our first MVC application before. Open `_Layout.cshtml` file, and you will see the following HTML code for the above menu bar. We will cut and paste this code in a separate partial view.



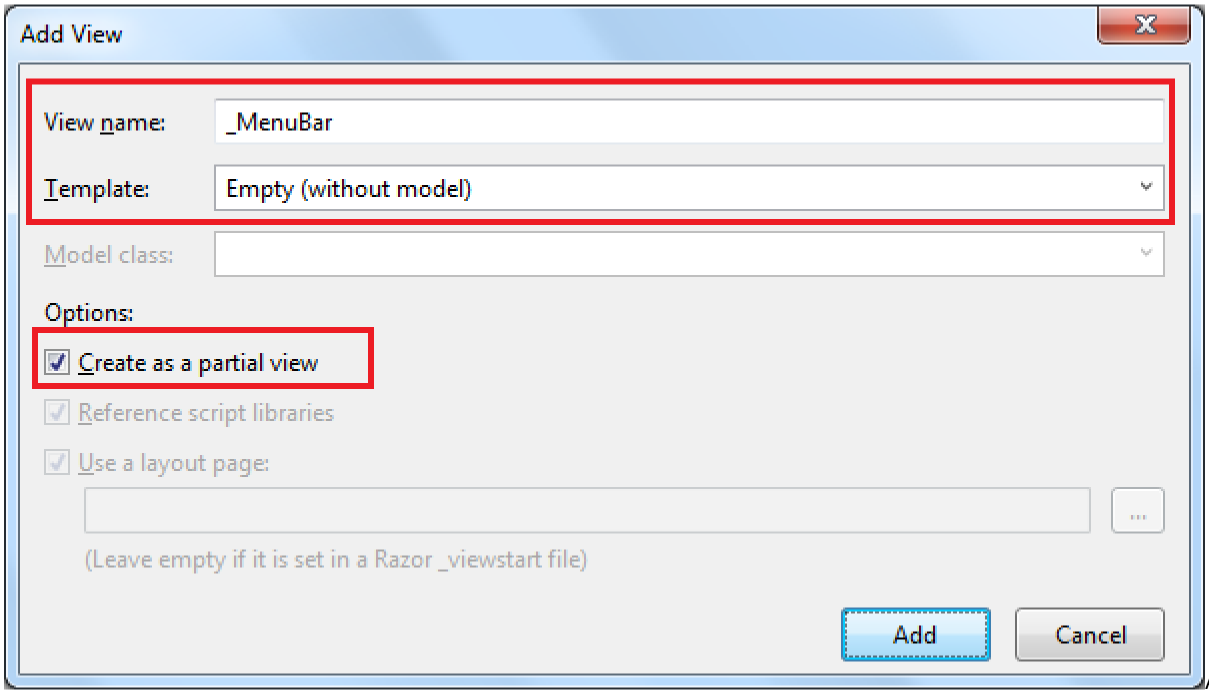
```
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - My ASP.NET Application</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    <div class="navbar navbar-inverse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

Partial Views

Create a New Partial View

To create a partial view, right click on the `Shared` folder -> click **Add** -> click **View..** to open the Add View popup, as shown below.

You can create a partial view in any `View` folder. However, it is recommended to create all your partial views in the `Shared` folder so that they can be used in multiple views.



View name: _MenuBar
Template: Empty (without model)
Model class:
Options:
[✓] Create as a partial view
[✓] Reference script libraries
[✓] Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)

Add Partial View

In the `Add New Item` popup, enter a partial view name, select "Create as a partial view" checkbox. We don't need not use any model for this partial view, so keep the Template dropdown as Empty (without model) and click on **Add** button. This will create an empty partial view in the `Shared` folder.

You can now cut the above code for the navigation bar and paste it in `_MenuBar.cshtml` as shown below:

_MenuBar.cshtml

Copy

```
<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.

ADVERTISEMENT

Rendering a Partial View

You can render the partial view in the parent view using the HTML helper methods: `@html.Partial()`, `@html.RenderPartial()`, and `@html.RenderAction()`.

Html.Partial()

The `@Html.Partial()` method renders the specified partial view. It accepts partial view name as a string parameter and returns `MvcHtmlString`. It returns an HTML string, so you have a chance of modifying the HTML before rendering.

Visit docs.microsoft.com to know the overloads of the Partial() method.

Now, include `_MenuBar` partial view in `_Layout.cshtml` using `@html.Partial("_MenuBar")`, as shown below.

_MenuBar

Copy

```
<!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("_MenuBar")

    @* you can modify result as below   *@
    @* var result = Html.Partial("_MenuBar") *@
    <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>
```

Html.RenderPartial()

The `@html.RenderPartial()` method is the same as the `@html.Partial()` method except that it writes the resulted HTML of a specified partial view into an HTTP response stream directly. So, you can modify it's HTML before render.

Visit docs.microsoft.com to know the overloads of the RenderPartial() method.

Example: Html.RenderPartial()

Copy

```
<!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("_MenuBar");
    }
    <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>
```

The `RenderPartial()` method returns void, so a semicolon is required at the end, and so it must be enclosed within the `@{ }`.

Html.RenderAction()

The `@html.RenderAction()` method executes the specified action method and renders the result. The specified action method must be marked with the `[ChildActionOnly]` attribute and return the `PartialViewResult` using the `PartialView()` method.

Visit docs.microsoft.com to know the overloads of the RenderAction() method.

To render a partial view using the `RenderAction()` method, first create an HttpGet action method and apply the `ChildActionOnly` attribute as shown below.

Example: Action Method Parameters

Copy

```
public class HomeController : Controller
{
    [ChildActionOnly]
    public ActionResult RenderMenu()
    {
        return PartialView("_MenuBar");
    }
}
```

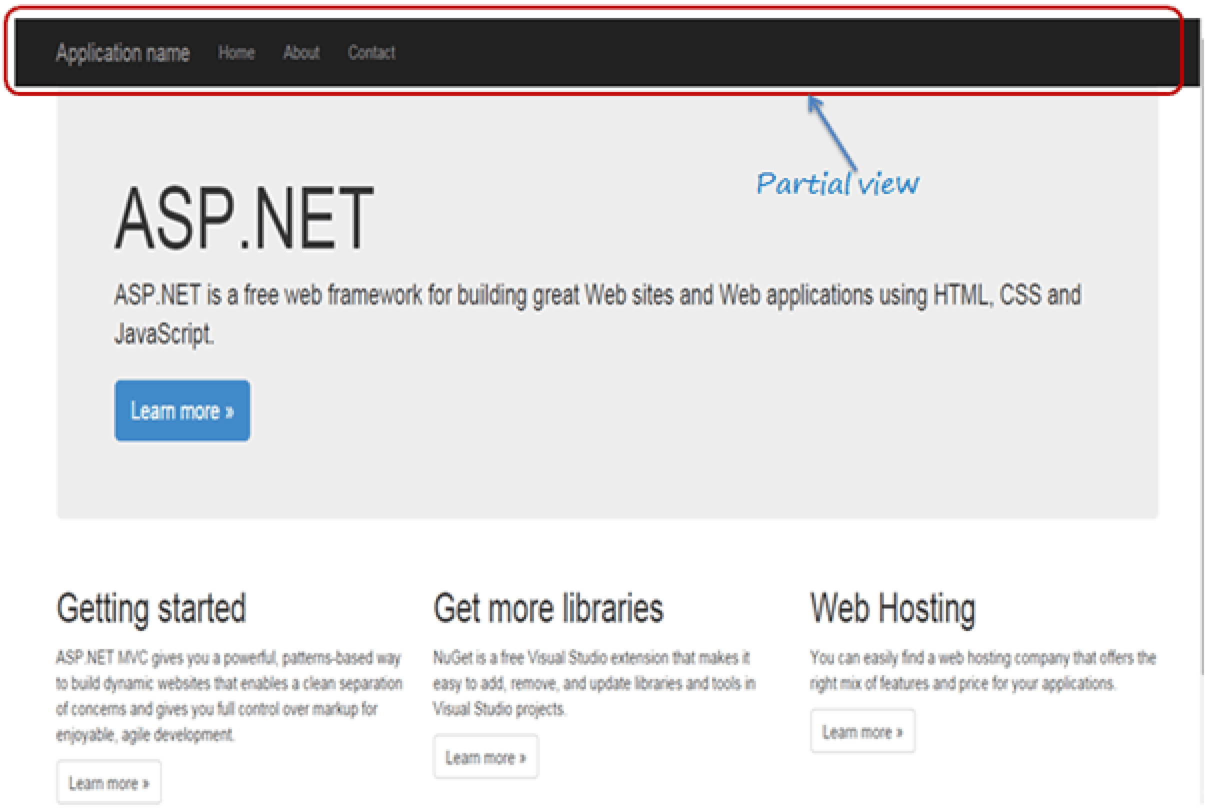
Now, call the `html.RenderAction("RenderMenu", "Home")` in the layout view, as shown below.

Example: Html.RenderPartial()

Copy

```
<!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.RenderAction("RenderMenu", "Home");
    }
    <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>
```

You will see the following result in the browser, irrespective of the rendering method you use.



ASP.NET

ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.

Getting started

ASP.NET MVC gives you a powerful, patterns-based way to build dynamic websites that enables a clean separation of concerns and gives you full control over markup for enjoyable, agile development.

Get more libraries

NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects.

Web Hosting

You can easily find a web hosting company that offers the right mix of features and price for your applications.

In this way, you can create a partial view for different portions of the web page in ASP.NET MVC application.

Difference between Html.Partial() and Html.RenderPartial() in ASP.NET MVC

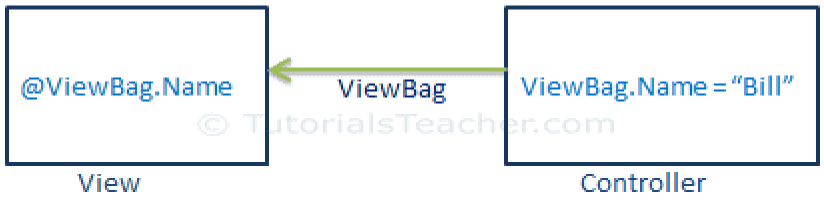
Html.Partial()	Html.RenderPartial()
Html.Partial returns html string.	Html.RenderPartial returns void.
Html.Partial injects the html string of the partial view into the main view.	Html.RenderPartial writes html in the response stream.
Performance is slow.	Perform is faster compared with HtmlPartial().
Html.Partial() need not to be inside the braces.	Html.RenderPartial must be inside braces @{ }.

ASP.NET MVC - ViewBag

The ViewBag in ASP.NET MVC is used to transfer temporary data (which is not included in the model) from the controller to the view.

Internally, it is a dynamic type property of the `ControllerBase` class which is the base class of the `Controller` class.

The following figure illustrates the ViewBag.



ViewBag Data Transfer

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.

TIPS You can assign a primitive or a complex type object as a value to ViewBag property.

You 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

Copy

```
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. So, we have attached the `TotalStudents` property to the `ViewBag` and assigned `studentList.Count()` value.

Now, in the `Index.cshtml` view, you can access `ViewBag.TotalStudents` property, as shown below.

Index.cshtml

Copy

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

Output:

```
Total Students: 5
```

Internally, ViewBag is a wrapper around ViewData. It will throw a runtime exception, if the ViewBag property name matches with the key of ViewData.

ViewBag Limitations

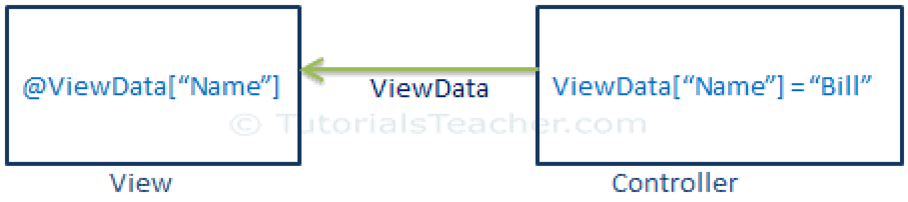
- ViewBag doesn't require typecasting while retrieving values from it. This can throw a run-time exception if the wrong method is used on the value.
- ViewBag is a dynamic type and skips compile-time checking. So, ViewBag property names must match in controller and view while writing it manually.

ASP.NET MVC - ViewData

In ASP.NET MVC, ViewData is similar to ViewBag, which transfers data from Controller to View. ViewData is of Dictionary type, whereas ViewBag is of dynamic type. However, both store data in the same dictionary internally.

ViewData is a dictionary, so it contains key-value pairs where each key must be a string.

The following figure illustrates the 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
 Copy
```
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, `ViewData["students"]` assigned to a `studentList` where `"students"` is a key and `studentList` is a value. You can now access `ViewData["students"]` in the view, as shown below.

Example: Access ViewData in a Razor View
 Copy
```
<ul>
@foreach (var std in ViewData["students"] as IList<Student>)
{
    <li>
        @std.StudentName
    </li>
}
</ul>
```

Above, we retrieve the value using `ViewData["students"]` and typecast it to an appropriate data type. You can also add `KeyValuePair` objects into the ViewData, as shown below.

Example: Add KeyValuePair in ViewData
 Copy
```
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
 Copy
```
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();
}
```

Points to Remember :

1. ViewData transfers data from the Controller to View, not vice-versa.
2. ViewData is a dictionary type.
3. ViewData's life only lasts during the current HTTP request. ViewData values will be cleared if redirection occurs.
4. ViewData value must be typecast to an appropriate type before using it.
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 is used to transfer data from view to controller, controller to view, or from one action method to another action method of the same or a different controller.

TempData stores the data temporarily and automatically removes it after retrieving a value.

TempData is a property in the ControllerBase class. So, it is available in any controller or view in the ASP.NET MVC application.

The following example shows how to transfer data from one action method to another using TempData.

Example: TempData

Copy

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        TempData["name"] = "Bill";

        return View();
    }

    public ActionResult About()
    {
        string name;

        if(TempData.ContainsKey("name"))
            name = TempData["name"].ToString(); // returns "Bill"

        return View();
    }

    public ActionResult Contact()
    {
        //the following throws exception as TempData["name"] is null
        //because we already accessed it in the About() action method
        //name = TempData["name"].ToString();

        return View();
    }
}
```

In the above example, we added data in the TempData in the `Index()` action method and access it in the `About()` action method. Assume that the user will go to the `Index` page first and then to the `About` page.

The following transfers data from an action method to a view.

Example: TempData

Copy

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        TempData["name"] = "Bill";

        return View();
    }

    public ActionResult About()
    {
        //the following throws exception as TempData["name"] is null
        //because we already accessed it in the Index.cshtml view
        //name = TempData["name"].ToString();

        return View();
    }

    public ActionResult Contact()
    {
        //the following throws exception as TempData["name"] is null
        //because we already accessed it in the Index.cshtml view
        //name = TempData["name"].ToString();

        return View();
    }
}
```

Above, we added data in the TempData in the `Index()` action method. So, we can access it in the `Index.cshtml` view, as shown below. Because we have accessed it in the index view first, we cannot access it anywhere else.

Index.cshtml

Copy

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

@TempData["name"]
```

ADVERTISEMENT

You can also transfer data from a view to controller, as shown below.

Index.cshtml

Copy

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

@{
    TempData["name"] = "Steve";
}
```

The above TempData can be accessed in the controller, as shown below.

Example: TempData

Copy

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

    public ActionResult About()
    {
        if(TempData.ContainsKey("name"))
            name = TempData["name"].ToString(); // returns "Bill"

        return View();
    }

    public ActionResult Contact()
    {
        //the following throws exception as TempData["name"] is null
        //because we already accessed it in the About() action method
        //name = TempData["name"].ToString();

        return View();
    }
}
```

Although, TempData removes a key-value once accessed, you can still keep it for the subsequent request by calling `TempData.Keep()` method.

The following example shows how to retain TempData value for the subsequent requests even after accessing it.

Example: TempData.Keep()

Copy

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        TempData["name"] = "Bill";
        return View();
    }

    public ActionResult About()
    {
        string name;

        if(TempData.ContainsKey("name"))
            name = TempData["name"] as string;

        TempData.Keep("name"); // Marks the specified key in the TempData for retention.

        //TempData.Keep(); // Marks all keys in the TempData for retention

        return View();
    }

    public ActionResult Contact()
    {
        string name;

        if(TempData.ContainsKey("name"))
            data = TempData["name"] as string;

        return View();
    }
}
```

ASP.NET MVC- Filters

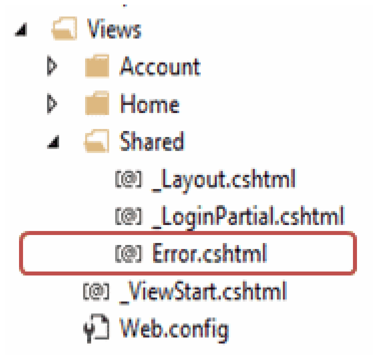
In ASP.NET MVC, a user request is routed to the appropriate controller and action method. However, there may be circumstances where you want to execute some logic before or after an action method executes. ASP.NET MVC provides filters for this purpose.

ASP.NET MVC Filter is a custom class where you can write custom logic to execute before or after an action method executes. Filters can be applied to an action method or controller in a declarative or programmatic way. Declarative means by applying a filter attribute to an action method or controller class and programmatic means by implementing a corresponding interface.

MVC provides different types of filters. The following table list filter types, built-in filters, and interface that must be implemented to create custom filters.

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

To understand the filter in detail, let's take an example of a built-in Exception filter. Exception filter executes when an unhandled exception occurs in your application. The `HandleErrorAttribute` class is a built-in exception filter class that renders the `Error.cshtml` by default when an unhandled exception occurs.



The following example demonstrates the use of `[HandError]` attribute on the controller class.

Example: Exception Filter

Copy

```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        //throw exception for demo
        throw new Exception("This is unhandled exception");
        return View();
    }

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

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

Above, the `[HandleError]` attribute applied to the `HomeController`. So, an error page `Error.cshtml` will be displayed if any action method of the `HomeController` throws an unhandled exception. Please note that unhandled exceptions are exceptions that are not handled by the try-catch blocks.

Filters applied to the controller will automatically be applied to all the action methods of a controller.

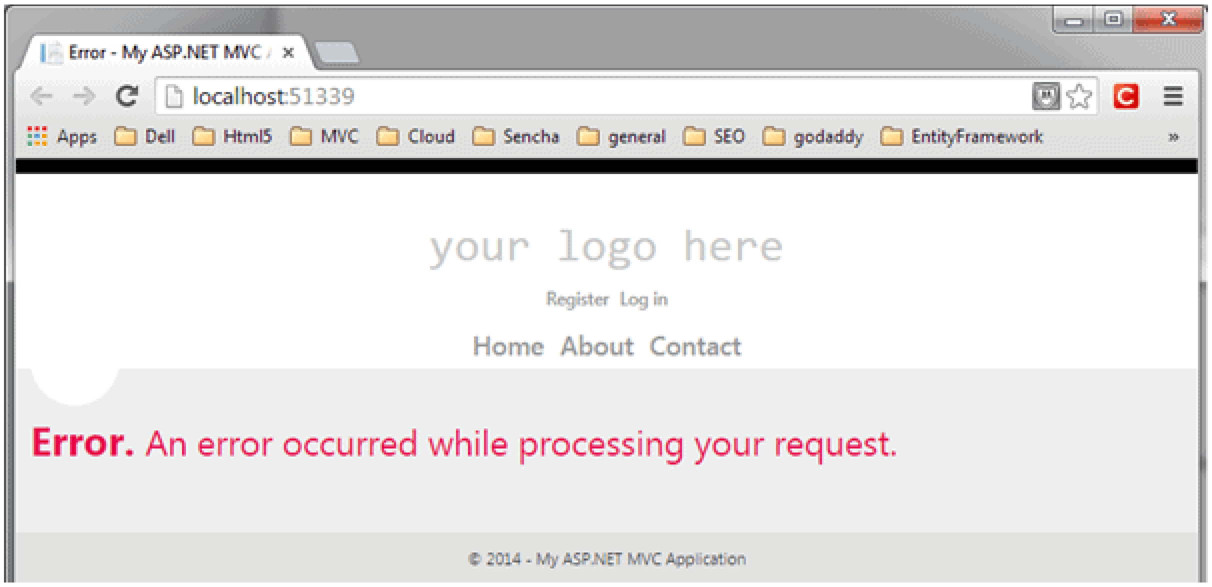
Please make sure that the `CustomError` mode is on in `System.web` section of web.config.

Example: Set CustomError Mode in web.config

Copy

```
<customErrors mode="On" />
```

Now, if you run the application, you would get the following error page because we throw an exception in the `Index()` action method for the demo purpose.



Error. An error occurred while processing your request.

© 2014 - My ASP.NET MVC Application

ADVERTISEMENT

Register Filters

Filters can be applied at three levels.

Global Level Filters

You can apply filters at a global level in the `Application_Start` event of the `global.asax.cs` file by using default `FilterConfig.RegisterGlobalFilters()` method. The global filters will be applied to all the controller and action methods of an application.

The `[HandleError]` filter is applied globally in the MVC application by default in every MVC application created using Visual Studio, as shown below.

Example: Register Global Filters

Copy

```
// MvcApplication class contains in Global.asax.cs file
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    }
}

// FilterConfig.cs located in App_Start folder
public class FilterConfig
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

Controller Level Filters

Filters can also be applied to the controller class. Controller level filters are applied to all the action methods. The following filter are applicable to all the action methods of the `HomeController`, but not on other controllers.

Example: Action Filters on Controller

Copy

```
[HandleError]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

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

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

}
```

Action Method Filters

One or more filters can also applied to an individual action method. The following filter applied only on the `Index()` action method.

Example: Filters on Action Method

Copy

```
public class HomeController : Controller
{
    [HandleError]
    public ActionResult Index()
    {
        return View();
    }

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

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

}
```

ASP.NET MVC - Action Filters

In the previous section, you learned about filters in MVC. In this section, you will learn about another filter type called action filters in ASP.NET MVC.

Action filter executes before and after an action method executes. Action filter attributes can be applied to an individual action method or to a controller. When an action filter is applied to a controller, it will be applied to all the controller's action methods.

The `OutputCache` is a built-in action filter attribute that can be applied to an action method for which we want to cache the output. For example, the output of the following action method will be cached for 100 seconds.

Example: ActionFilter

Copy

```
[OutputCache(Duration=100)]
public ActionResult Index()
{
    return View();
}
```

Create a Custom Action Filter

You can create a custom action filter in two ways, first, by implementing the `IActionFilter` interface and the `FilterAttribute` class. Second, by deriving the `ActionFilterAttribute` abstract class.

The IActionFilter interface include following methods to implement:

- void OnActionExecuted(ActionExecutedContext filterContext)
- void OnActionExecuting(ActionExecutingContext filterContext)

The ActionFilterAttribute abstract class includes the following methods to override:

- void OnActionExecuted(ActionExecutedContext filterContext)
- void OnActionExecuting(ActionExecutingContext filterContext)
- void OnResultExecuted(ResultExecutedContext filterContext)
- void OnResultExecuting(ResultExecutingContext filterContext)

As you can see, the `ActionFilterAttribute` class has four overload methods. It includes the `OnResultExecuted` and the `OnResultExecuting` methods, which can be used to execute custom logic before or after the result executes. Action filters are generally used to apply cross-cutting concerns such as logging, caching, authorization, etc.

The following example demonstrates creating a custom action filter class for logging.

Example: Custom ActionFilter for Logging

Copy

```
public class LogAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        Log("OnActionExecuted", filterContext.RouteData);
    }

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        Log("OnActionExecuting", filterContext.RouteData);
    }

    public override void OnResultExecuted(ResultExecutedContext filterContext)
    {
        Log("OnResultExecuted", filterContext.RouteData);
    }

    public override void OnResultExecuting(ResultExecutingContext filterContext)
    {
        Log("OnResultExecuting ", filterContext.RouteData);
    }
    private void Log(string methodName, RouteData routeData)
    {
        var controllerName = routeData.Values["controller"];
        var actionName = routeData.Values["action"];
        var message = String.Format("{0}- controller:{1} action:{2}", methodName,
                                                                    controllerName,
                                                                    actionName);
        Debug.WriteLine(message);
    }
}
```

Above, the `Log` class derived the `ActionFilterAttribute` class. It logs before and after the action method or result executes. You can apply the `Log` attribute to any controller or an action method where you want to log the execution of the action method.

Example: ActionFilter on Controller

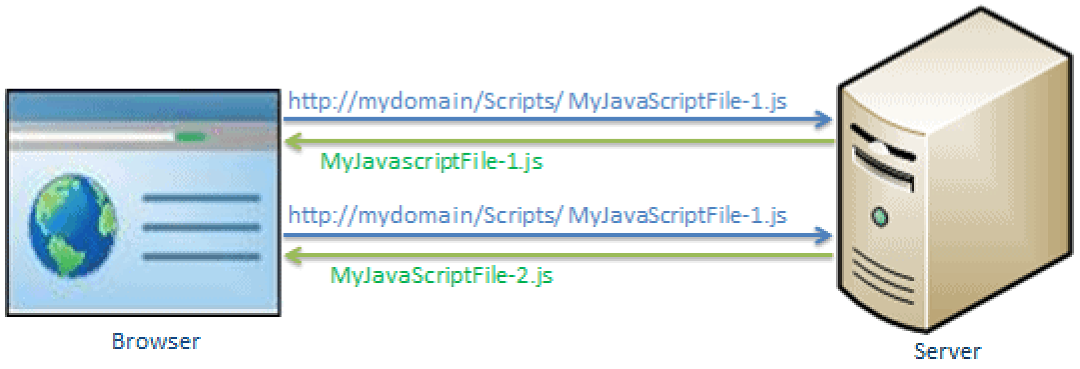
Copy

```
[Log]
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

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

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

Bundling and Minification

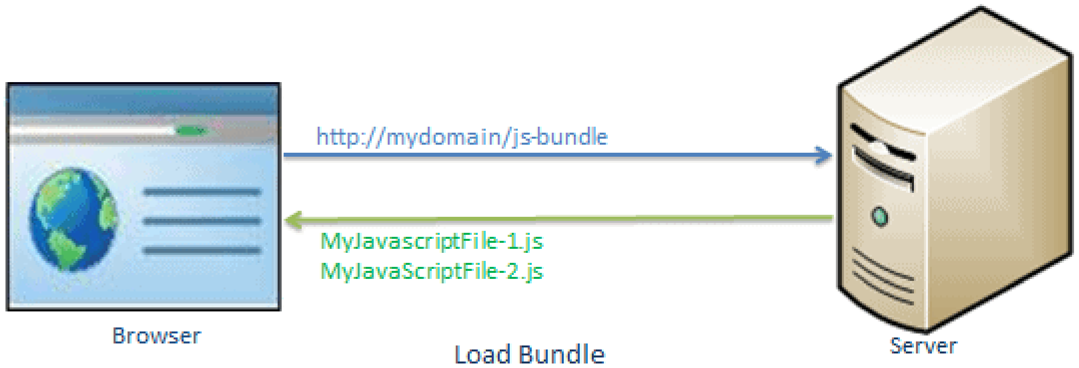
Bundling and minification techniques were introduced in MVC 4 to improve request load time. Bundling allows us to load the bunch of static files from the server in a single HTTP request.

The following figure illustrates the bundling technique:


http://mydomain/Scripts/ MyJavaScriptFile-1.js
MyJavascriptFile-1.js
http://mydomain/Scripts/ MyJavaScriptFile-1.js
MyJavaScriptFile-2.js
Browser
Server
Loading script files in separate requests

In the above figure, the browser sends two separate requests to load two different JavaScript file `MyJavaScriptFile-1.js` and `MyJavaScriptFile-2.js`.

The bundling technique in ASP.NET MVC allows us to load more than one JavaScript file, `MyJavaScriptFile-1.js` and `MyJavaScriptFile-2.js` in one request, as shown below.



Load Bundle

Minification

Minification technique optimizes script or CSS file size by removing unnecessary white space and comments and shortening variable names to one character.

For example, consider the following JavaScript function.

Example: JavaScript

Copy

```
sayHello = function(name){
    //this is comment
    var msg = "Hello" + name;
    alert(msg);
}
```

Minification will remove the unnecessary white spaces, comments, and shortening variable names to reduce the characters, which will reduce the size of the JavaScript file. The above JavaScript will be minimized as the following script.

Example: Minified JavaScript

Copy

```
sayHello=function(n){var t="Hello"+n;alert(t)}
```

Bundling and minification impact on the loading time of the page.

Bundle Types

MVC 5 includes following bundle classes in `System.web.Optimization` namespace:

ScriptBundle: ScriptBundle is responsible for JavaScript minification of single or multiple script files.

StyleBundle: StyleBundle is responsible for CSS minification of single or multiple style sheet files.

DynamicFolderBundle: Represents a Bundle object that ASP.NET creates from a folder that contains files of the same type.

Combine Script Files using ScriptBundle in ASP.NET MVC

Here, you will learn how to combine multiple JavaScript files and create a script bundle that can be returned in a single HTTP request in ASP.NET MVC.

The ScriptBundle class represents a bundle that does JavaScript minification and bundling. You can create style or script bundles in `BundleConfig` class under `App_Start` folder in an ASP.NET MVC project. (you can create your own custom class instead of using BundleConfig class, but it is recommended to follow standard practice.)

The following example demonstrates how to create a script bundle.

Example: Create Script Bundle

Copy

```
using System.Web;
using System.Web.Optimization;

public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/bs-jq-bundle").Include(
                      "~/Scripts/bootstrap.js",
                      "~/Scripts/jquery-3.3.1.js"));

        //the following creates bundles in debug mode;
        //BundleTable.EnableOptimizations = true;
    }
}
```

In the above example, we created a new bundle by creating an instance of the `ScriptBundle` class and specified the virtual path and bundle name in the constructor. The `~/bundles/` is a virtual path and `bs-jq-bundle` is a bundle name. Then, we added two js files, `bootstrap.js`, and `jquery-3.3.1.js` in this bundle. The `bundles.Add()` method is used to add new bundles into the `BundleCollection`. By default, the above `bs-jq-bundle` bundle will be created in the release mode. Use `BundleTable.EnableOptimizations = true` if you want to see bundles in the debug mode.

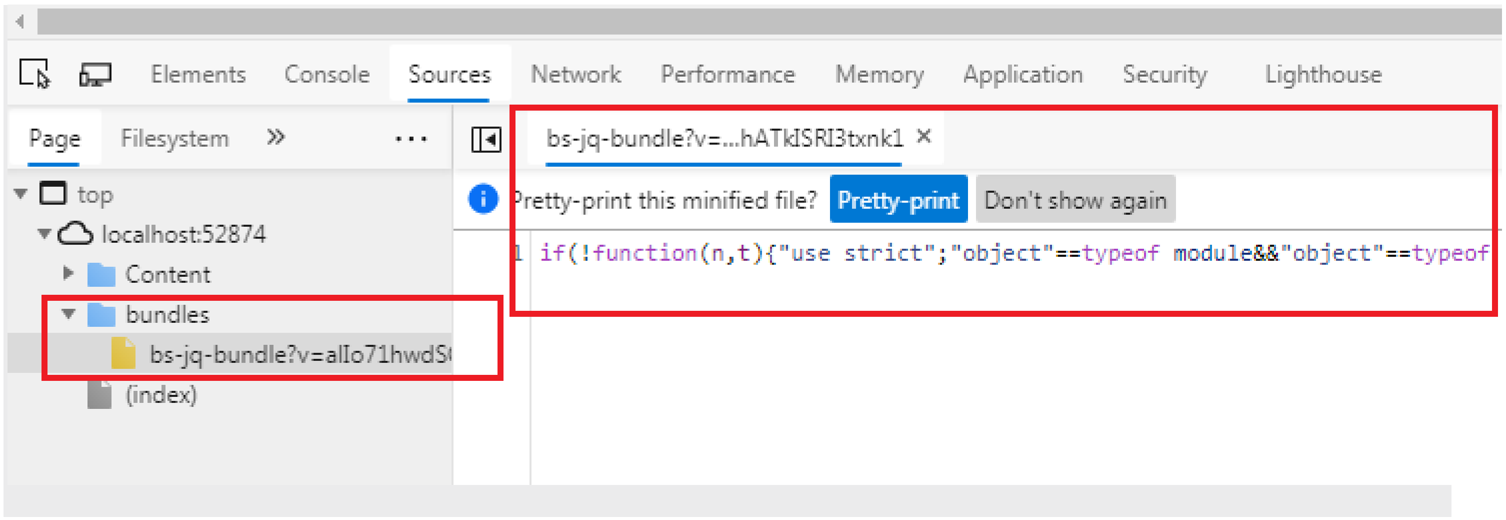
Now, to include the above `bs-jq-bundle` in your webpage, use `Scripts.Render()` method in the layout view, as shown below.

Example: Use Script Bundle

Copy

```
<!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, when you run the application in the release mode, you will see the bundle is created and loaded in a single request.



```
if(!function(n,t){"use strict";"object"==typeof module&&"object"==typeof
```

Include a Directory in Bundle

Use the `IncludeDirectory` method to add all the files under a particular directory in a bundle, as shown below.

ScriptBundle Example:

Copy

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/scripts")
                    .IncludeDirectory("~/Scripts/","*.js", true));
}
```

Using Wildcards

Most third party JavaScript files include a version in the name of the script file. For example, jQuery includes the version in the file name. The wildcard `{version}` will automatically pick up an available version file.

Example: Wildcard with bundle

Copy

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/jquery")
               .Include( "~/Scripts/jquery-{version}.js"));
    }
}
```

Using CDN

You can also create a bundle of the files from the Content Delivery Network (CDN), as shown below.

Example: Load files from CDN

Copy

```
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"));
    }
}
```

Note:

ASP.NET MVC framework calls the `BundleConfig.RegisterBundle()` from the `Application_Start` event in the `Global.asax.cs` file. So, all the bundles are added into the `BundleCollection` at the starting of an application.

StyleBundle - Combine CSS Files in ASP.NET MVC

Here you will learn how to combine multiple CSS (Cascading Style Sheet) files to return it in a single HTTP request.

ASP.NET MVC API includes StyleBundle class that does CSS minification and bundling. Same as the script bundle, all the style bundles should be created in the `BundleConfig` class. under the `App_Start` folder.

The following example shows how to combine multiple CSS files into a bundle.

Example: Create Style Bundle

Copy

```
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..

    }
}
```

In the above example, we created a new style bundle by creating an instance of the `StyleBundle` class and specified the virtual path and bundle name in the constructor. The `~/bundles/` is a virtual path and `css` is a bundle name. Then, we added two `.css` files, `bootstrap.css`, and `site.css` in this bundle. The `bundles.Add()` method is used to add new bundles into the `BundleCollection`. By default, the above `css` bundle will be created in the release mode. Use `BundleTable.EnableOptimizations = true` if you want to see bundles in the debug mode.

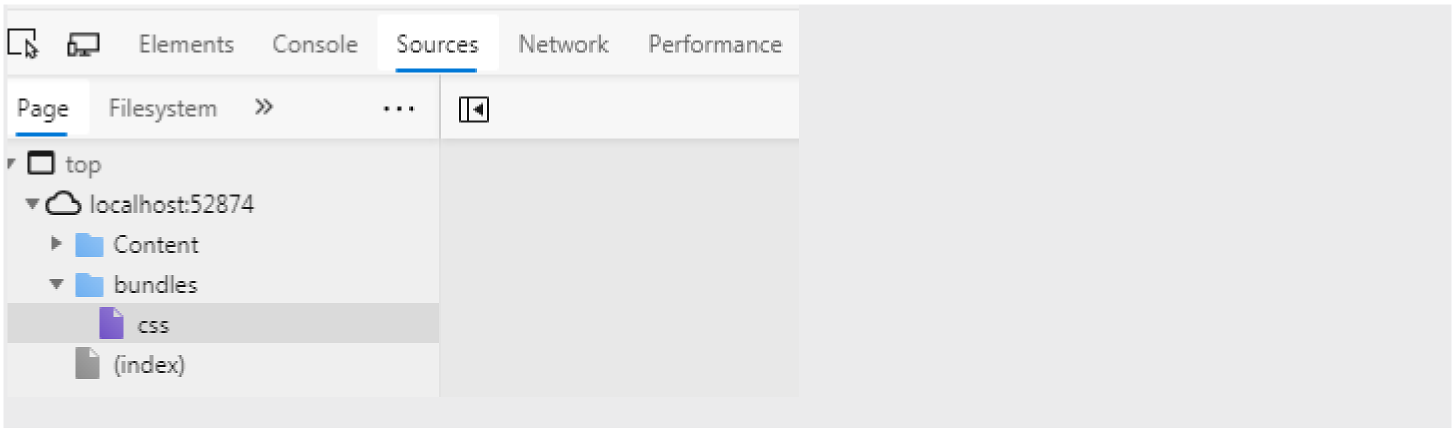
Now, to include the above `css` bundle in your webpage, use `Styles.Render()` method in the layout view, as shown below.

Example: Include Style Bundle in View

Copy

```
<!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>
```

Now, when you run the application in the release mode, you will see the bundle is created and loaded in a single request.



You can use the `IncludeDirectory()` method, version wildcard `{version}`, and CDN path the same way as ScriptBundle. Learn how to set image path in StyleBundle.

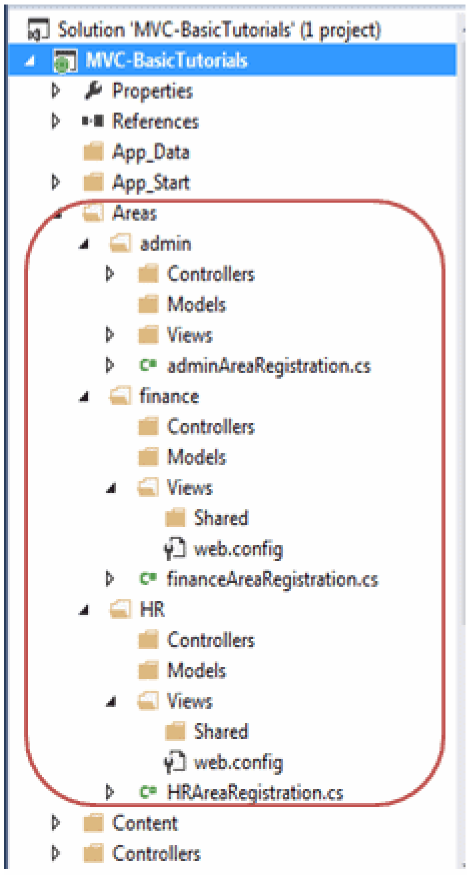
Note:

ASP.NET MVC framework calls the `BundleConfig.RegisterBundle()` from the `Application_Start` event in the `Global.asax.cs` file. So, all the bundles are added into the `BundleCollection` at the starting of an application.

Area in ASP.NET MVC

Here, you will learn what an area in ASP.NET MVC application is and how to create it.

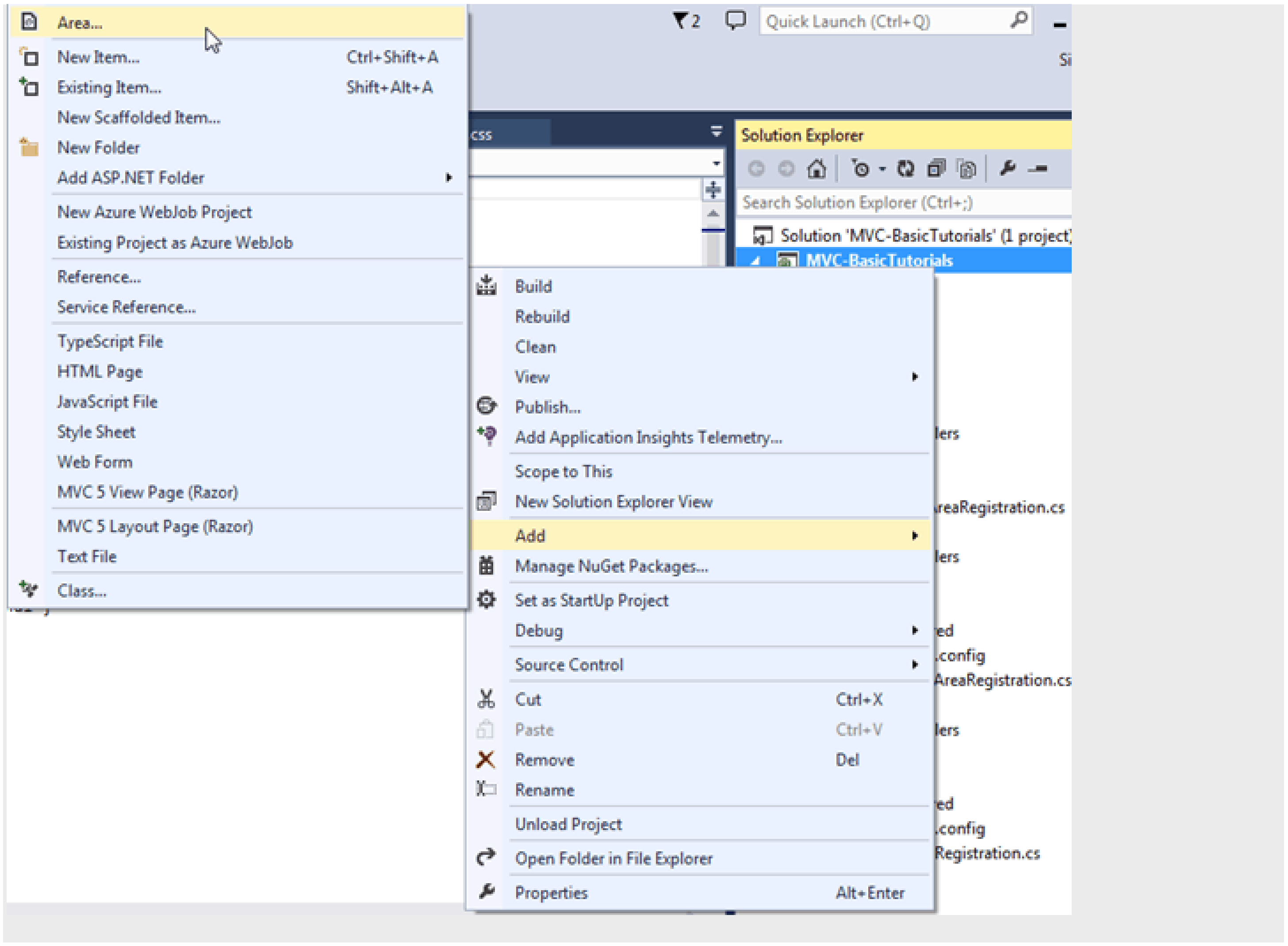
The large ASP.NET MVC application includes many controllers, views, and model classes. So it can be difficult to maintain it with the default ASP.NET MVC project structure. ASP.NET MVC introduced a new feature called Area for this. Area allows us to partition the large application into smaller units where each unit contains a separate MVC folder structure, same as the default MVC folder structure. For example, a large enterprise application may have different modules like admin, finance, HR, marketing, etc. So an Area can contain a separate MVC folder structure for all these modules, as shown below.



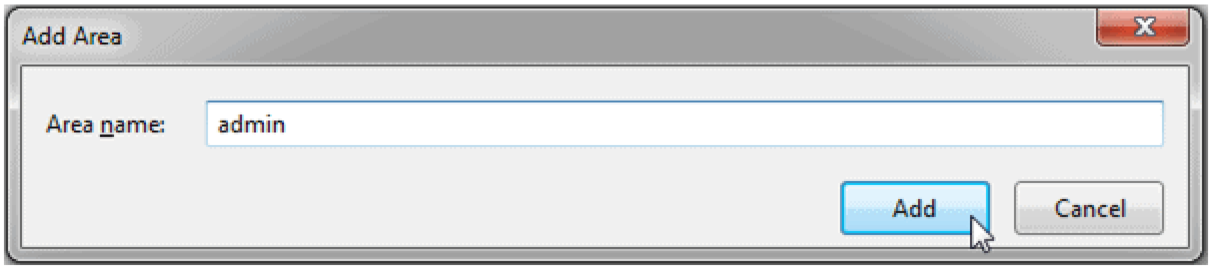
Multiple Areas in ASP.NET MVC Application

Creating an Area

You can create an area by right-clicking on the project in the solution explorer -> `Add` -> `Area..`, as shown below.

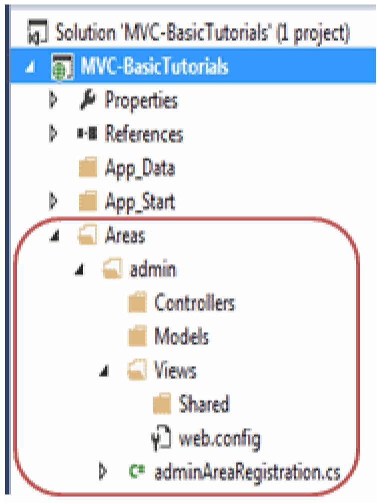


Enter the name of an area in the `Add Area` dialogue box and click on the Add button.



Area name: admin

This will add an `admin` folder under the `Area` folder, as shown below.



As you can see, each area includes the `AreaRegistration` class. The following is `adminAreaRegistration` class created with admin area.

Example: Area Registration

Copy

```
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 }
        );
    }
}
```

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

Finally, all the areas must be registered in the `Application_Start` event in `Global.asax.cs` as `AreaRegistration.RegisterAllAreas();`.

So in this way, you can create and maintain multiple areas for the large application.