ASP.NET MVC 5

MVC Architecture

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

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

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

Model: Model represents shape of the data and business logic. It maintains the data of the application. Model objects retrieve and store model state in a database.

Model is a data and business logic.

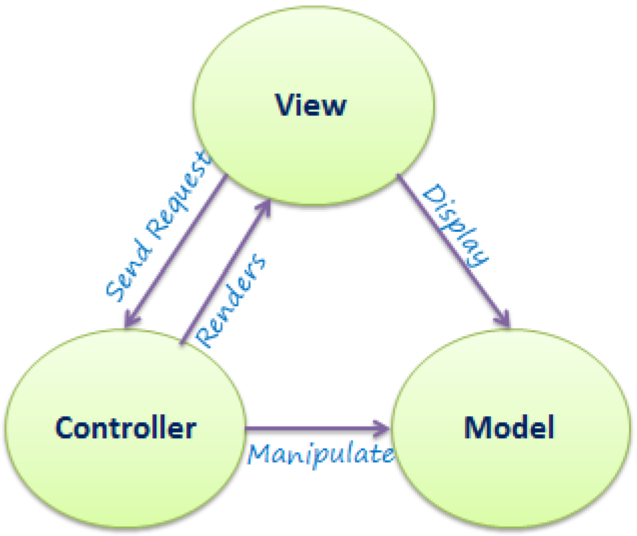
View: View is a user interface. View display data using model to the user and also enables them to modify the data.

View is a User Interface.

Controller: Controller handles the user request. Typically, user interact with View, which in-turn raises appropriate URL request, this request will be handled by a controller. The controller renders the appropriate view with the model data as a response.

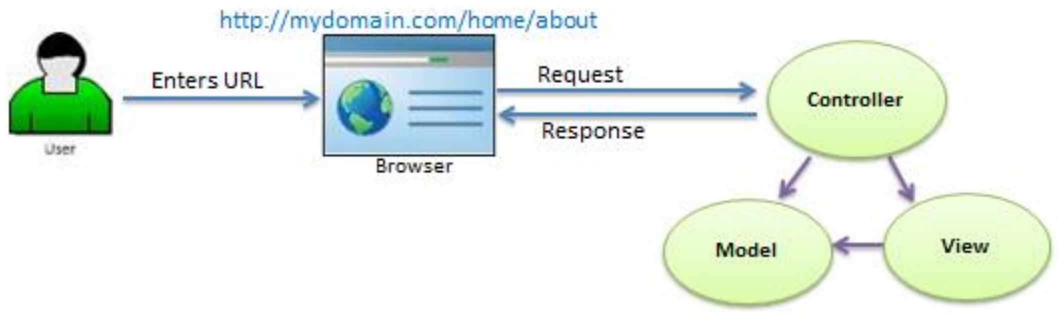
Controller is a request handler.

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



MVC Architecture

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



Request/Response in MVC Architecture

As per the above figure, when the user enters a URL in the browser, it goes to the server and calls appropriate controller. Then, the Controller uses the appropriate View and Model and creates the response and sends it back to the user

Create First ASP.NET MVC Application

In this section, we will create a new MVC 5 application with Visual Studio 2013 for Web and understand the basic building blocks of a MVC Application.

First of all, setup a development environment to develop an ASP.NET MVC 5 application.

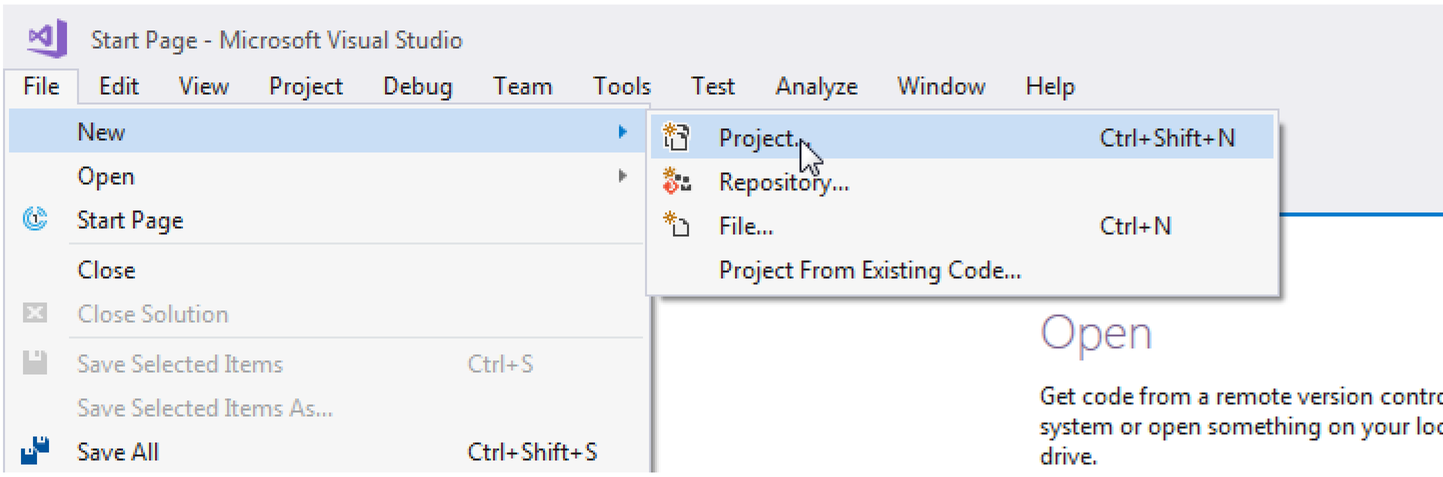
Setup Development Environment

You can develop ASP.NET MVC application with appropriate version of Visual Studio and .NET framework, as you have seen in the previous section of version history.

Here, we will use MVC v5.2, Visual Studio 2017 Community edition and .NET framework 4.6 to create our first MVC application.

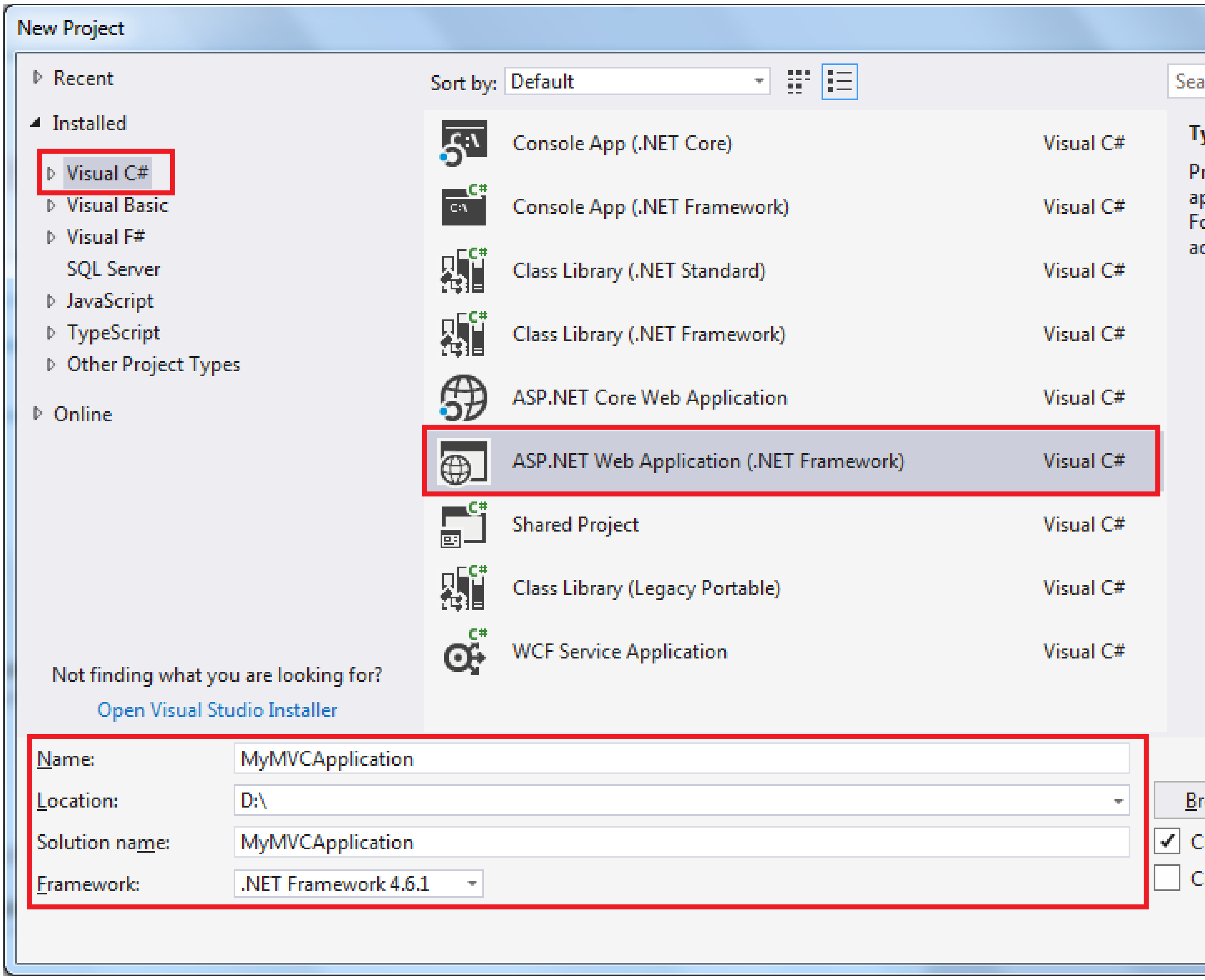
Create first simple MVC application

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


Start Page - Microsoft Visual Studio
File
Edit
View
Project
Debug
Team
Tools
Test
Analyze
Window
Help
New
Open
Start Page
Close
Close Solution
Save Selected Items Ctrl+S
Save Selected Items As...
Save All Ctrl+Shift+S
Project... Ctrl+Shift+N
Repository...
File... Ctrl+N
Project From Existing Code...
Open
Get code from a remote version contro
system or open something on your loc
drive.

Create a New Project in Visual Studio

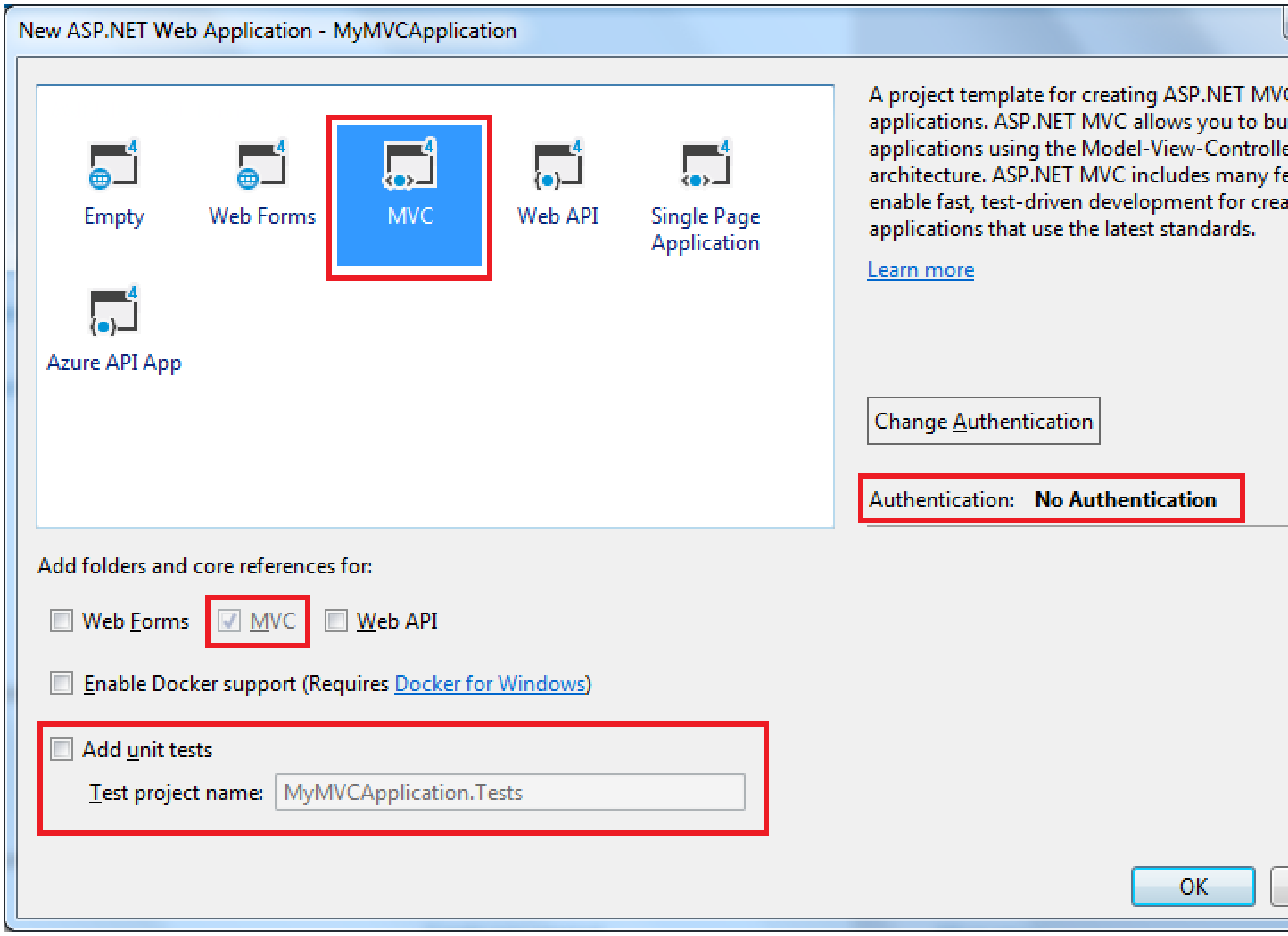
From the **New Project** dialog as shown below, expand Visual C# node and select **Web**in the left pane, and then select **ASP.NET Web Application (.NET Framework)** in the middle pane. Enter the name of your project `MyMVCApplication`. (You can give any appropriate name for your application). Also, 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
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#
Not finding what you are looking for?
Open Visual Studio Installer

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

Create MVC Project in Visual Studio

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 MV
applications. ASP.NET MVC allows you to bu
applications using the Model-View-Controlle
architecture. ASP.NET MVC includes many fe
enable fast, test-driven development for crea
applications that use the latest standards.

Authentication: **No Authentication**

Add folders and core references for:

☐ Web Forms ☑ MVC ☐ Web API

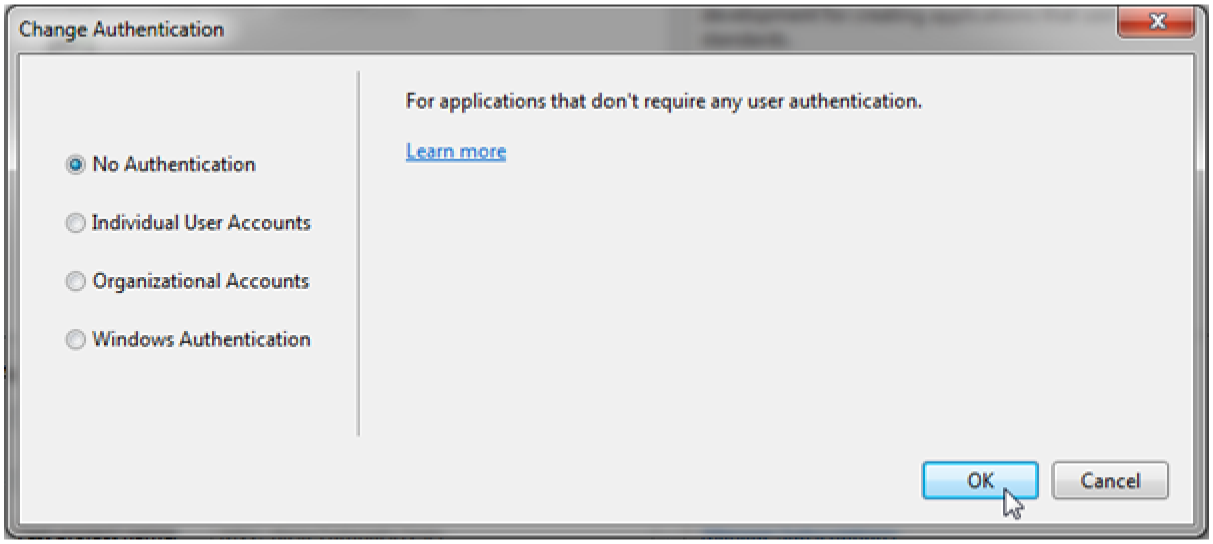
☐ Enable Docker support (Requires Docker for Windows)

☐ Add unit tests

Test project name: MyMVCApplication.Tests

Create MVC Application

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

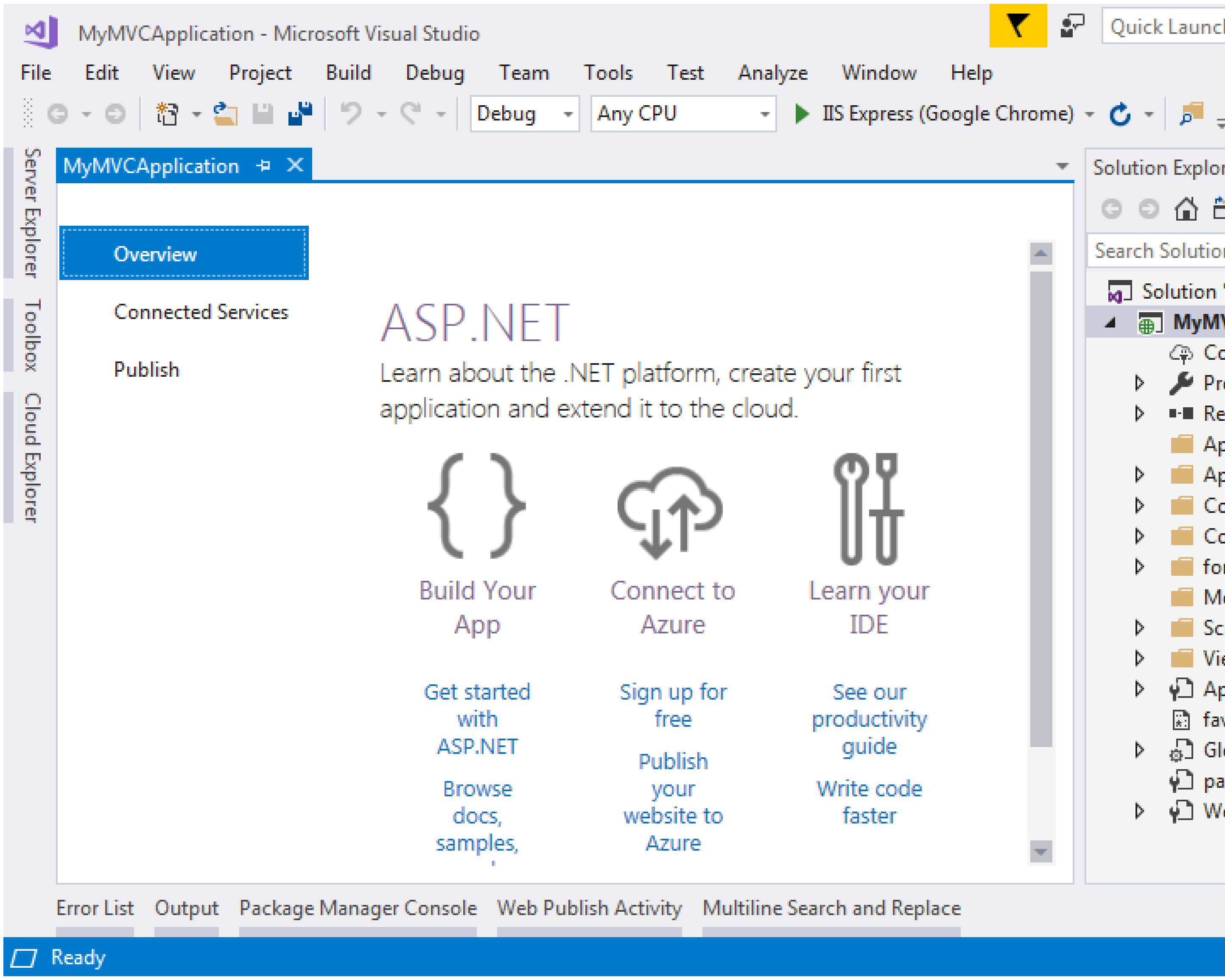


For applications that don't require any user authentication.

Select Authenctication Type

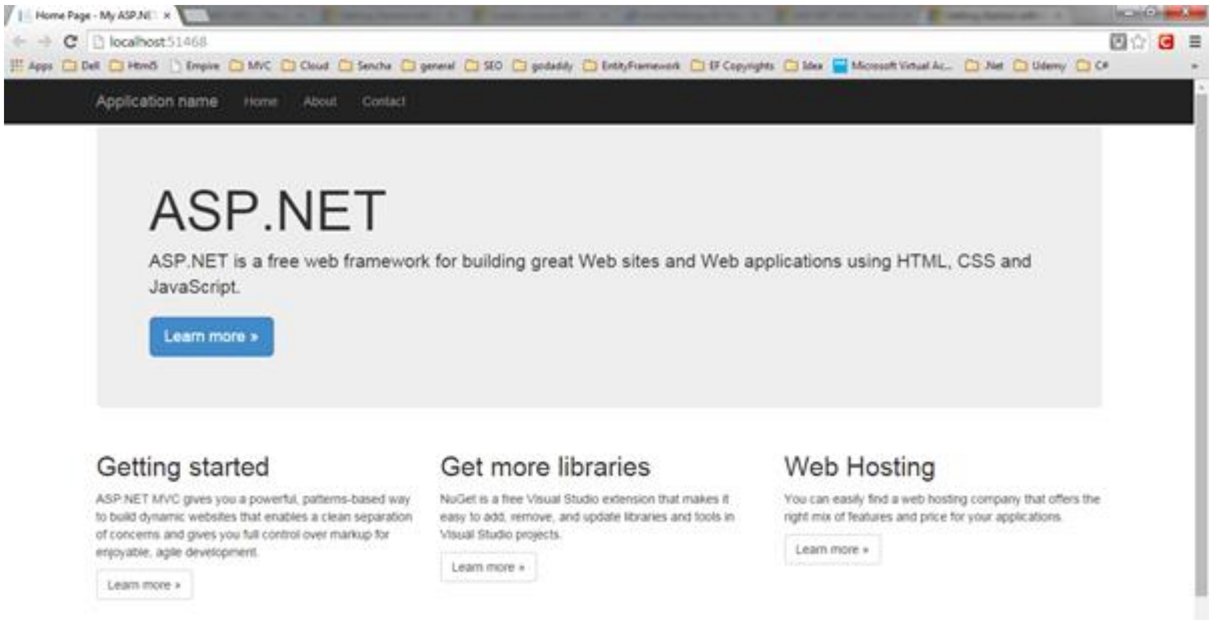
Here, we are keeping the default authentication for our application which is No Authentication. Click **OK** to continue.

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



First MVC Application

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



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.

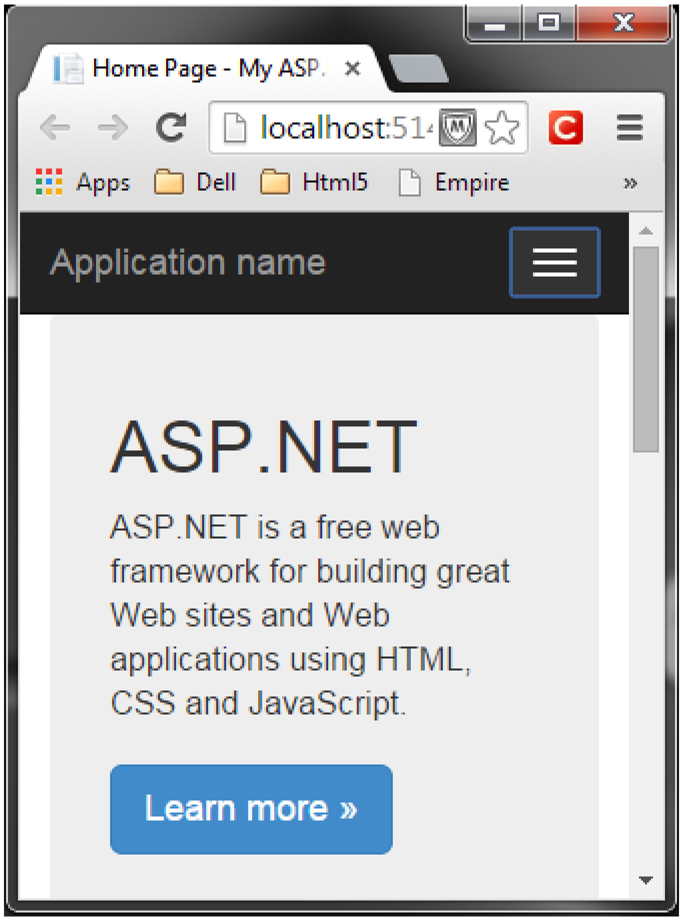
Get more libraries

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

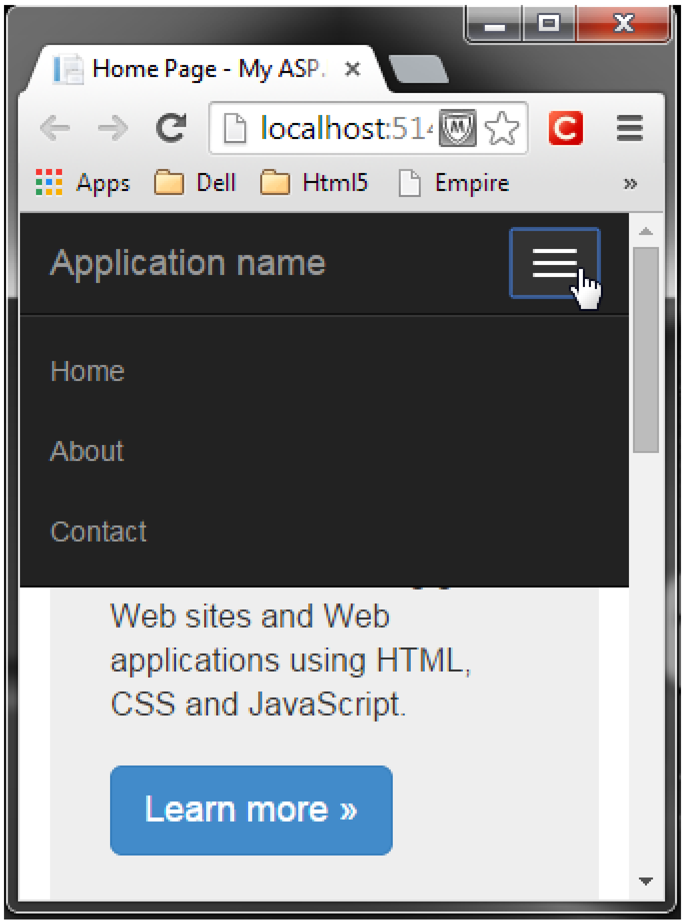
You can easily find a web hosting company that offers the right mix of features and price for your applications.

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, top menu bar will be changed in the mobile devices as shown below.



ASP.NET

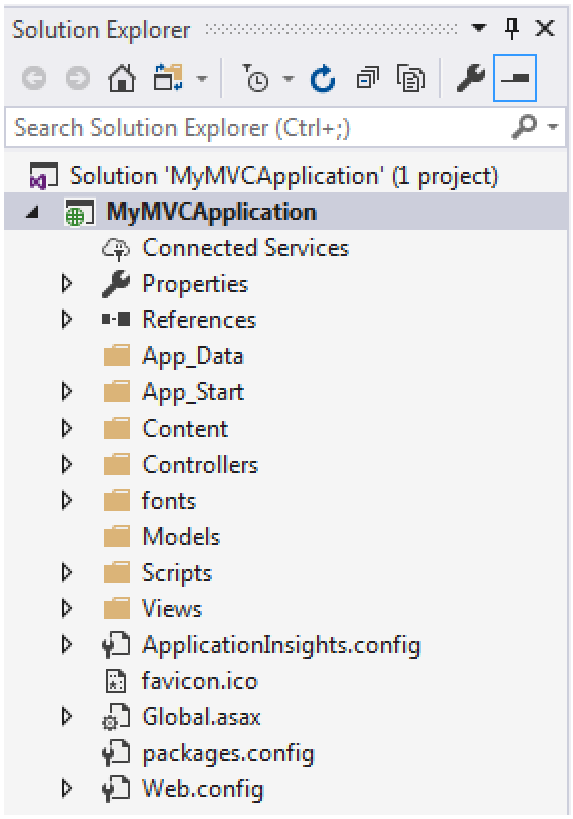
ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.



Web sites and Web applications using HTML, CSS and JavaScript.

ASP.NET MVC Folder Structure

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



MVC Folder Structure

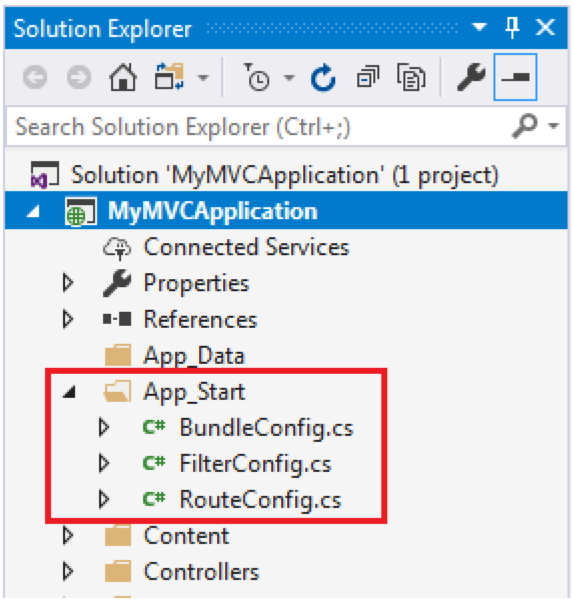
Let's see significance of each folder.

App_Data

App_Data folder can contain application data files like LocalDB, .mdf files, xml files and other data related files. IIS will never serve files from App_Data folder.

App_Start

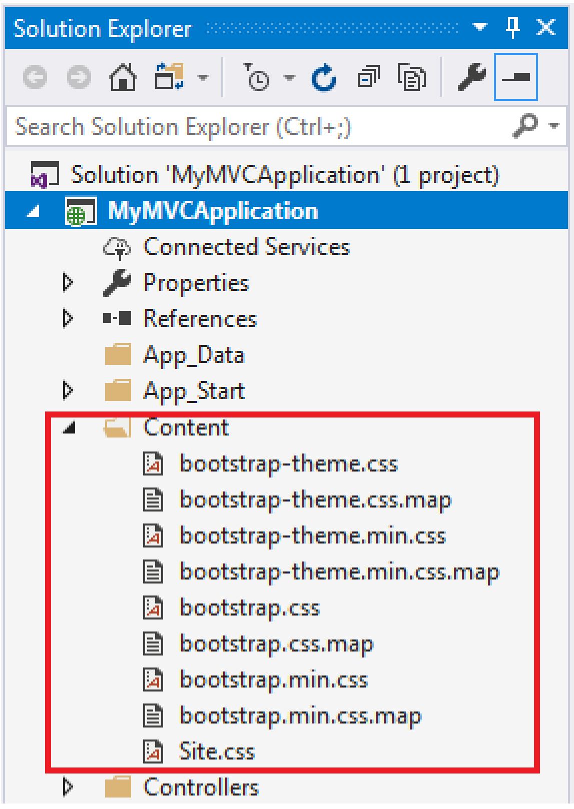
App_Start folder can contain class files which will be executed when the application starts. Typically, these would be config files like AuthConfig.cs, BundleConfig.cs, FilterConfig.cs, RouteConfig.cs etc. MVC 5 includes BundleConfig.cs, FilterConfig.cs and RouteConfig.cs by default. We will see significance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



App_Start Folder

Content

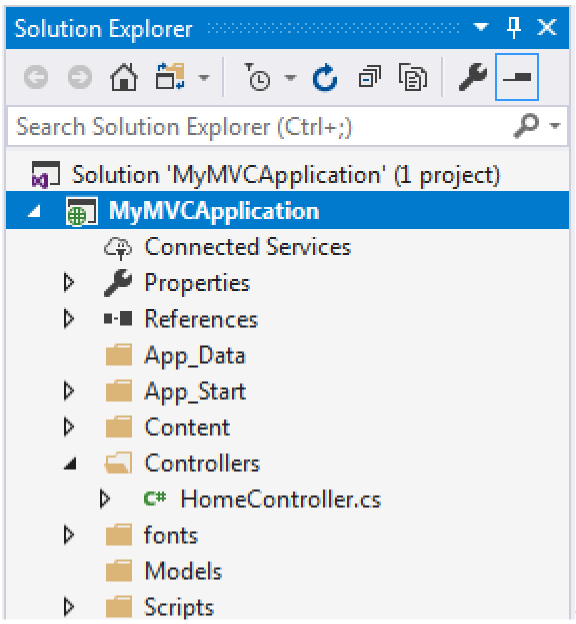
Content folder contains static files like css files, images and icons files. MVC 5 application includes bootstrap.css, bootstrap.min.css and Site.css by default.



Content Folder

Controllers

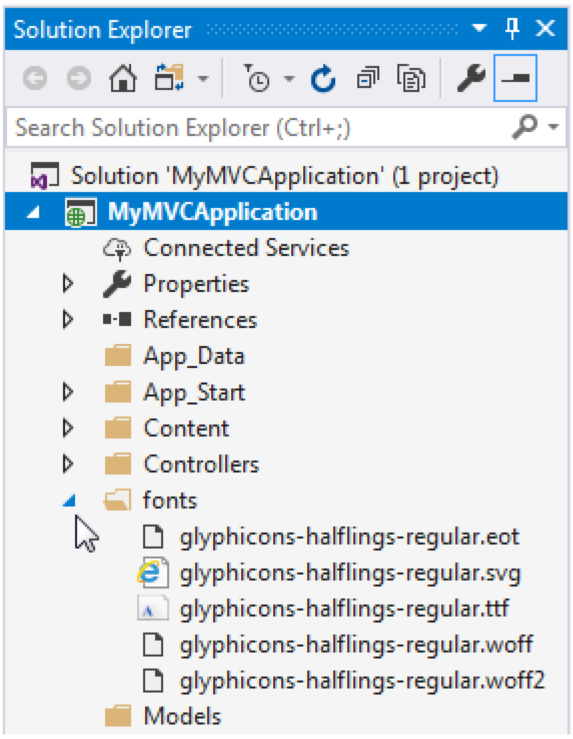
Controllers folder contains class files for the controllers. Controllers handles users' request and returns a response. MVC requires the name of all controller files to end with "Controller". You will learn about the controller in the next section.



Controller Folder

fonts

Fonts folder contains custom font files for your application.

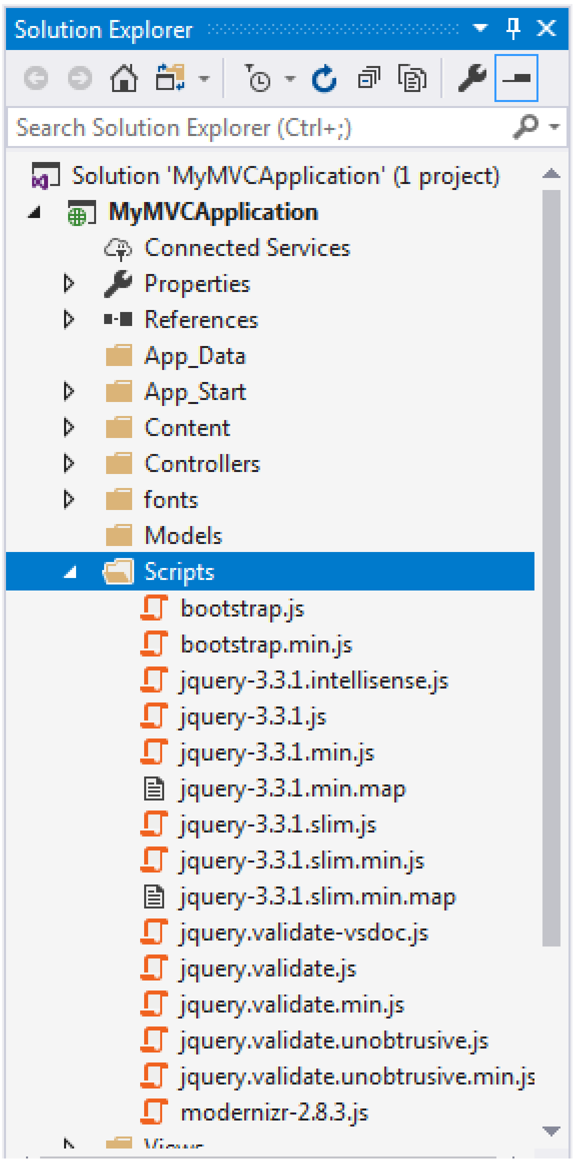

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'MyMVCApplication' (1 project)
MyMVCApplication
Connected Services
Properties
References
App_Data
App_Start
Content
Controllers
fonts
glyphicons-halflings-regular.eot
glyphicons-halflings-regular.svg
glyphicons-halflings-regular.ttf
glyphicons-halflings-regular.woff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

Models

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

Scripts

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



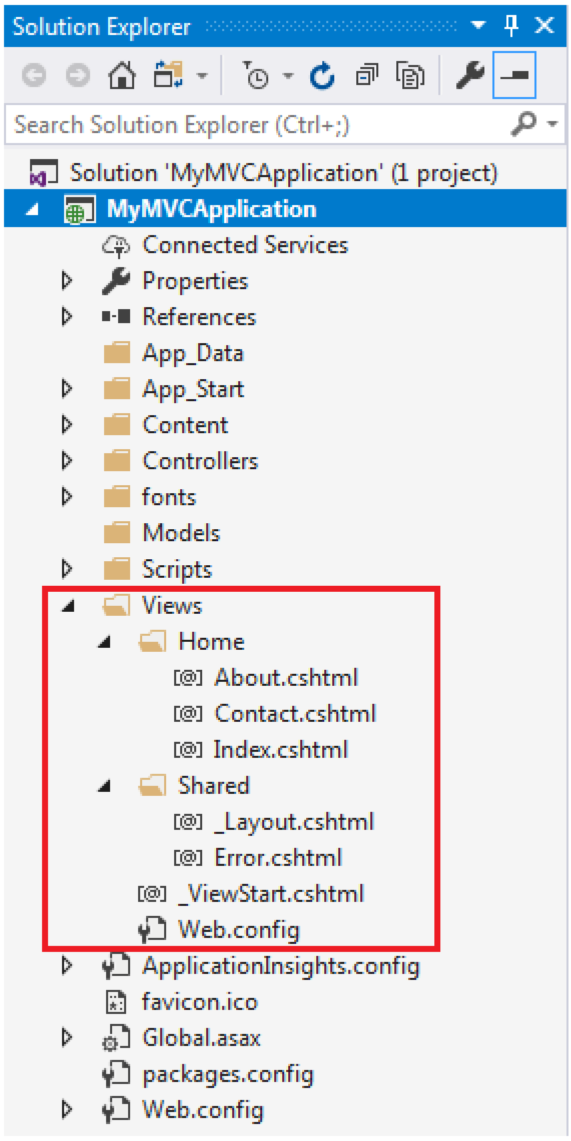
Scripts Folder

Views

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

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

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



View Folder

Additionally, MVC project also includes following configuration files:

Global.asax

Global.asax allows you to write code that runs in response to application level events, such as Application_BeginRequest, application_start, application_error, session_start, session_end etc.

Packages.config

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

Web.config

Web.config file contains application level configurations.

Routing in MVC

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

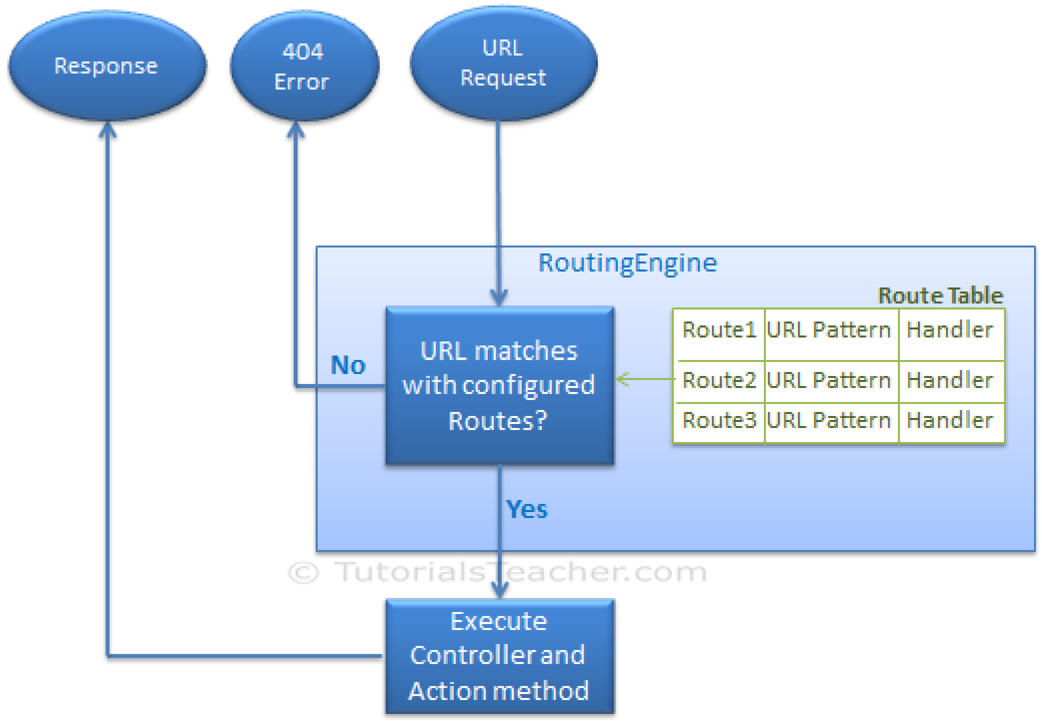
Routing is not specific to MVC framework. It can be used with ASP.NET Webform application or MVC application.

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

Route

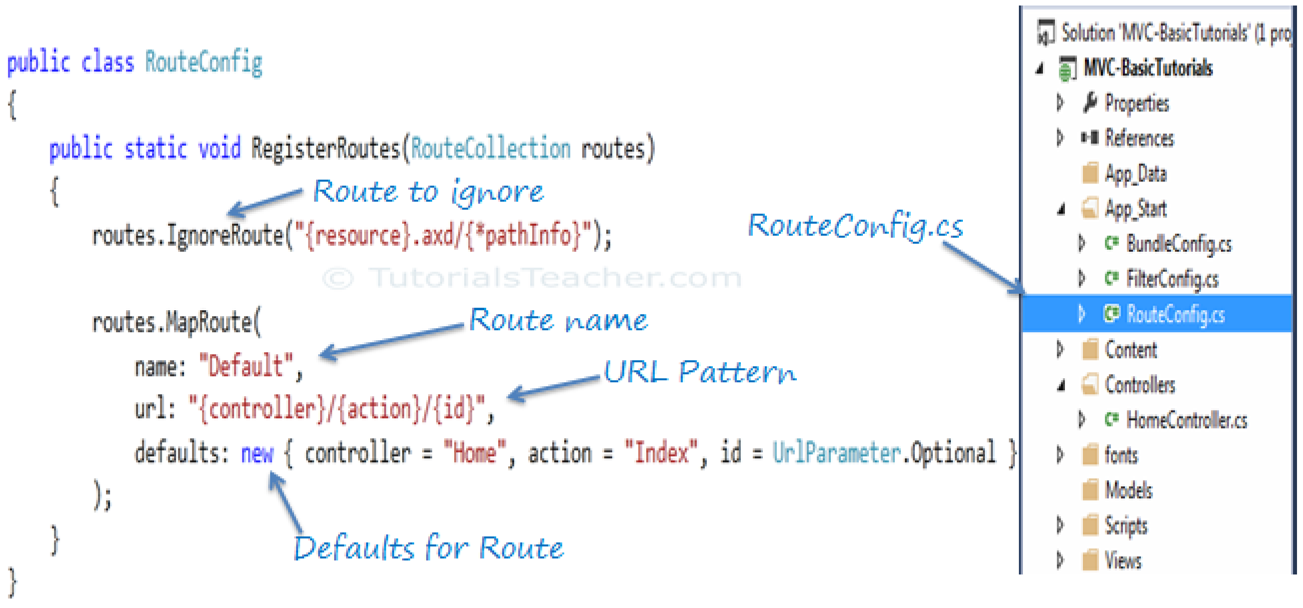
Route defines the URL pattern and handler information. All the configured routes of an application stored in RouteTable and will be used by Routing engine to determine appropriate handler class or file for an incoming request.

The following figure illustrates the Routing process.


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

Configure a Route

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



```
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 }
        );
    }
}
```

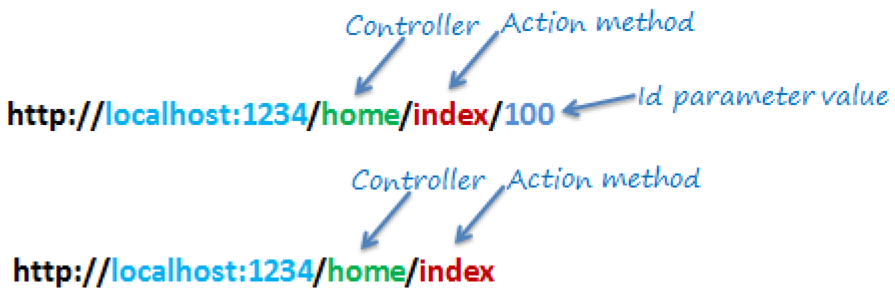
Configure Route in MVC

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

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

URL Pattern

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


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 domain name then the default controller and action method will handle the request. For example, http://lcoalhost:1234 would be handled by HomeController and Index method as configured in the defaults parameter.

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

URL	Controller	Action	Id
http://localhost/home	HomeController	Index	null
http://localhost/home/index/123	HomeController	Index	123
http://localhost/home/about	HomeController	About	null
http://localhost/home/contact	HomeController	Contact	null

URL	Controller	Action	Id
http://localhost/student	StudentController	Index	null
http://localhost/student/edit/123	StudentController	Edit	123

Multiple Routes

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

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

Example: Custom Routes

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

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

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

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

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

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

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

Route Constraints

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

Example: Route Constraints

```
routes.MapRoute(
        name: "Student",
        url: "student/{id}/{name}/{standardId}",
        defaults:  new  {  controller  =  "Student",  action  =  "Index",  id  =
UrlParameter.Optional,    name    =    UrlParameter.Optional,    standardId    =
UrlParameter.Optional },
        constraints: new { id = @"\d+" }
    );
```

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

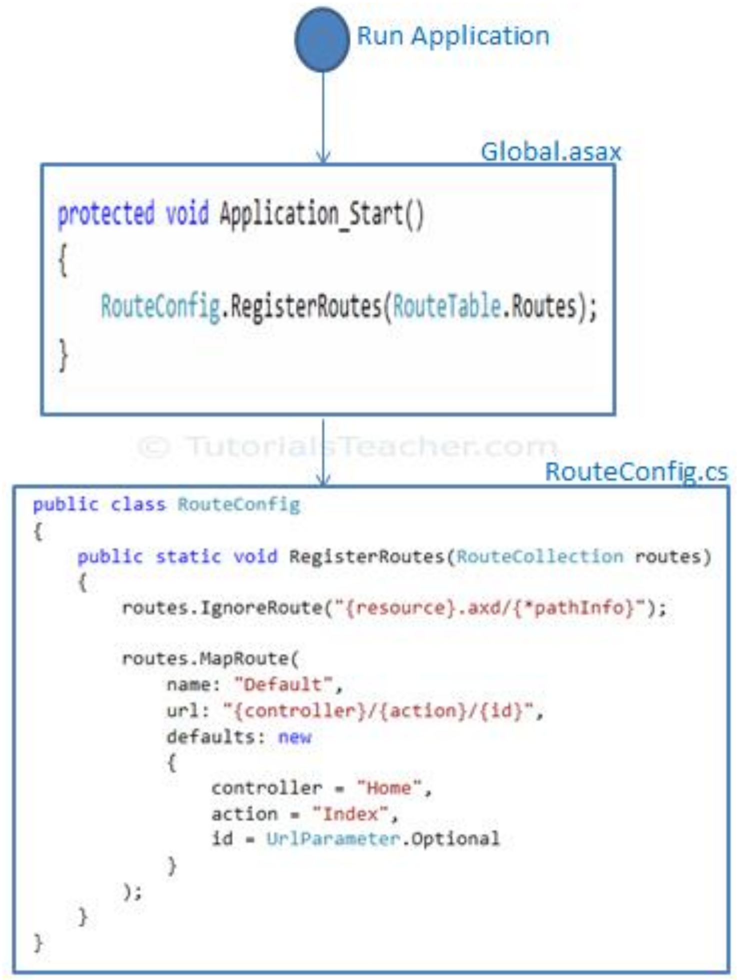
Register Routes

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

Example: Route Registration

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

The following figure illustrate Route registration process.



```
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.

Controller

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

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

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

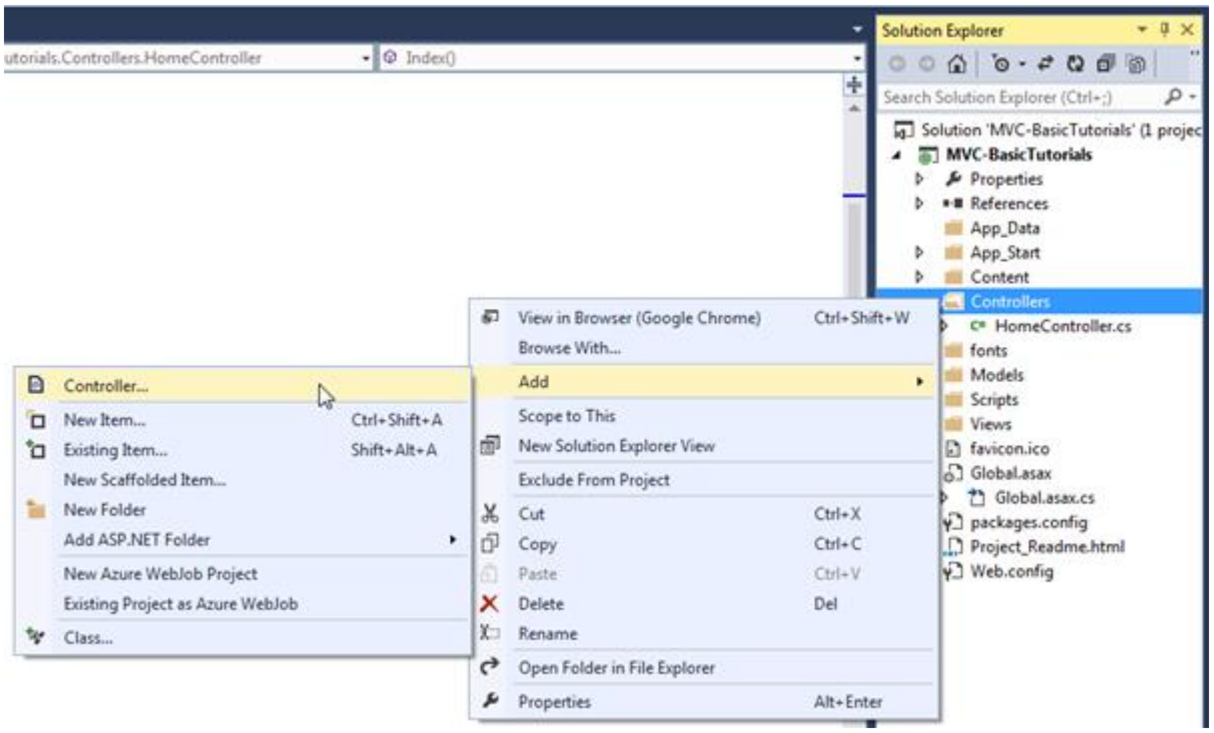
Adding a New Controller

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

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 in turn created a default HomeController. Here, we will create a new StudentController.

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

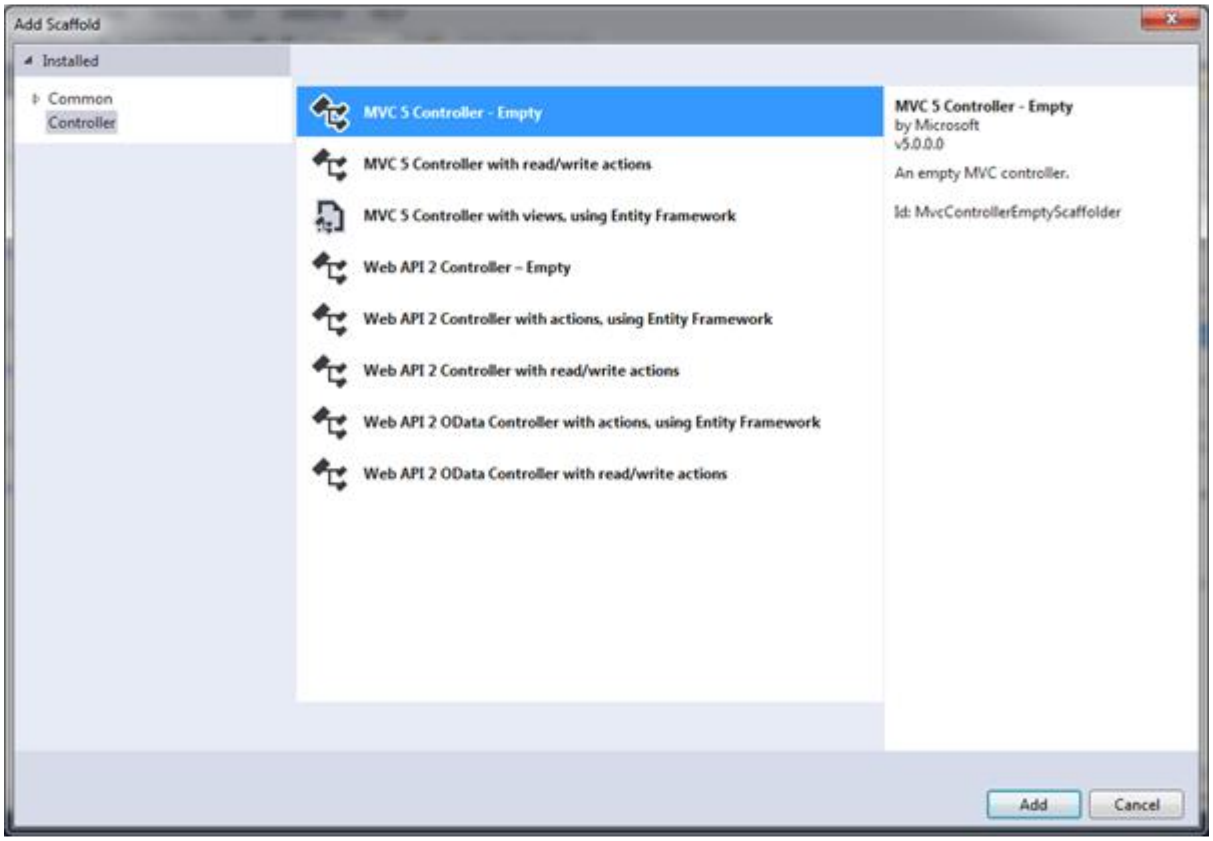


Add New Controller

This opens Add Scaffold dialog as shown below.

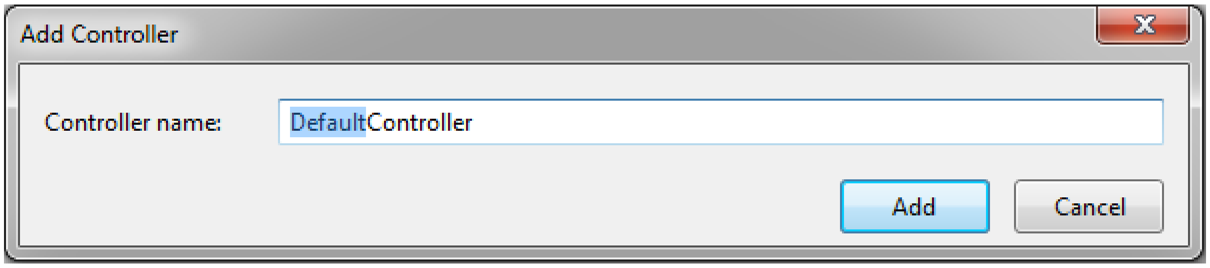
Note:

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


Add Scaffold
Installed
Common
Controller
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
Add
Cancel

Adding Controller

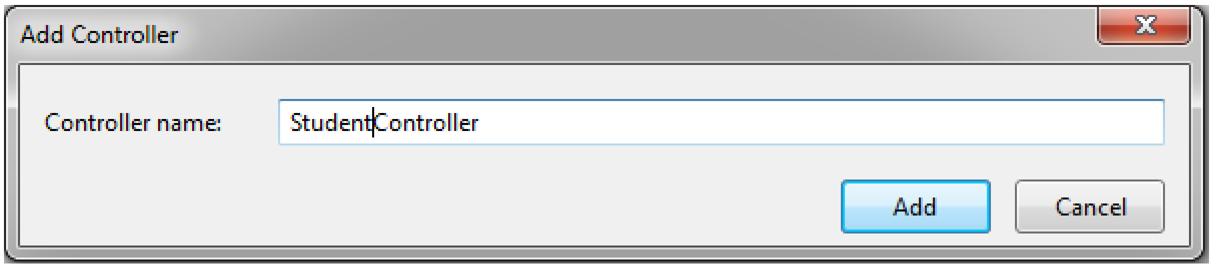
Add Scaffold dialog contains different templates to create a new **controller**. We will learn about other templates later. For now, select **"MVC 5 Controller - Empty"** and click **Add**. It will open Add Controller dialog as shown below



Controller name: DefaultController

Adding Controller

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



Adding Controller

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

Example: Controller

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

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

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

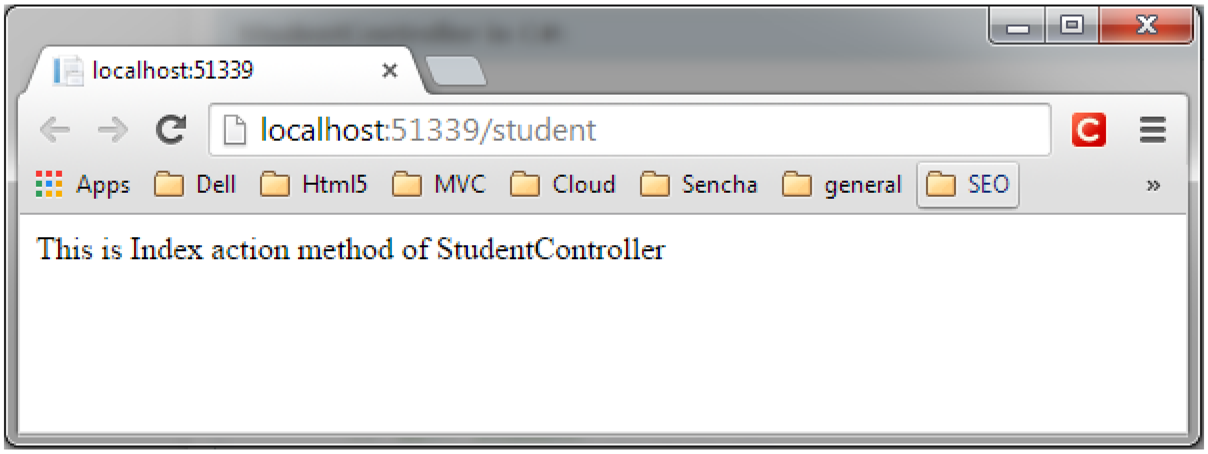
Now, we will return a dummy string from Index action method of above StudentController. Changing the return type of Index method from ActionResult to string and returning dummy string is shown below. You will learn about ActionResult in the next section.

Example: Controller

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

```
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public 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 StudentController class, 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

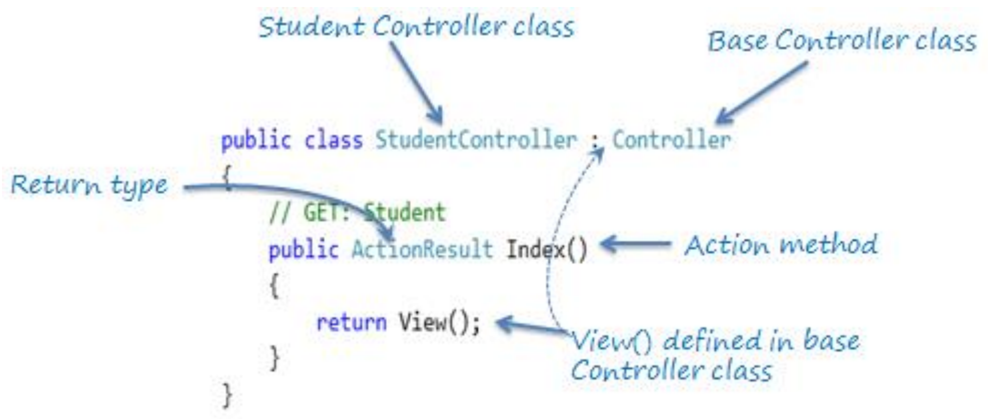
Action method

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

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

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

The following is an example of Index action method of StudentController



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

Action Method

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

Default Action Method

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

Default Route:

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

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

ActionResult

MVC framework includes various result classes, which can be return from an action methods. There result classes represent different types of responses such as html, file, string, json, javascript etc. The following table lists all the result classes available in ASP.NET MVC.

Result Class	Description
ViewResult	Represents HTML and markup.
EmptyResult	Represents No response.

Result Class	Description
ContentResult	Represents string literal.
FileContentResult/ FilePathResult/ FileStreamResult	Represents the content of a file
JavaScriptResult	Represent a JavaScript script.
JsonResult	Represent JSON that can be used in AJAX
RedirectResult	Represents a redirection to a new URL
RedirectToRouteResult	Represent another action of same or other controller
PartialViewResult	Returns HTML from Partial view
HttpUnauthorizedResult	Returns HTTP 403 status

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

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

Result Class	Description	Base Controller Method
ViewResult	Represents HTML and markup.	View()
EmptyResult	Represents No response.	
ContentResult	Represents string literal.	Content()
FileContentResult, FilePathResult, FileStreamResult	Represents the content of a file	File()
JavaScriptResult	Represent a JavaScript script.	JavaScript()
JsonResult	Represent JSON that can be used in AJAX	Json()
RedirectResult	Represents a redirection to a new URL	Redirect()

Result Class	Description	Base Controller Method
RedirectToRouteResult	Represent another action of same or other controller	RedirectToRoute()
PartialViewResult	Returns HTML	PartialView()
HttpUnauthorizedResult	Returns HTTP 403 status	

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

Action Method Parameters

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

Example: Action method parameters

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

    return RedirectToAction("Index");
}

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

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

Please note that action method paramter can be Nullable Type.

By default, the values for action method parameters are retrieved from the request's data collection. The data collection includes name/values pairs for form data or query string values or cookie values. Model binding in ASP.NET MVC automatically maps the URL query string or form data collection to the action method parameters if both names are matching.

Action Selectors

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

1. ActionName
2. NonAction
3. ActionVerbs

ActionName

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

Example: ActionName

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

    }

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

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

NonAction

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

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

Example: NonAction

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

    }

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

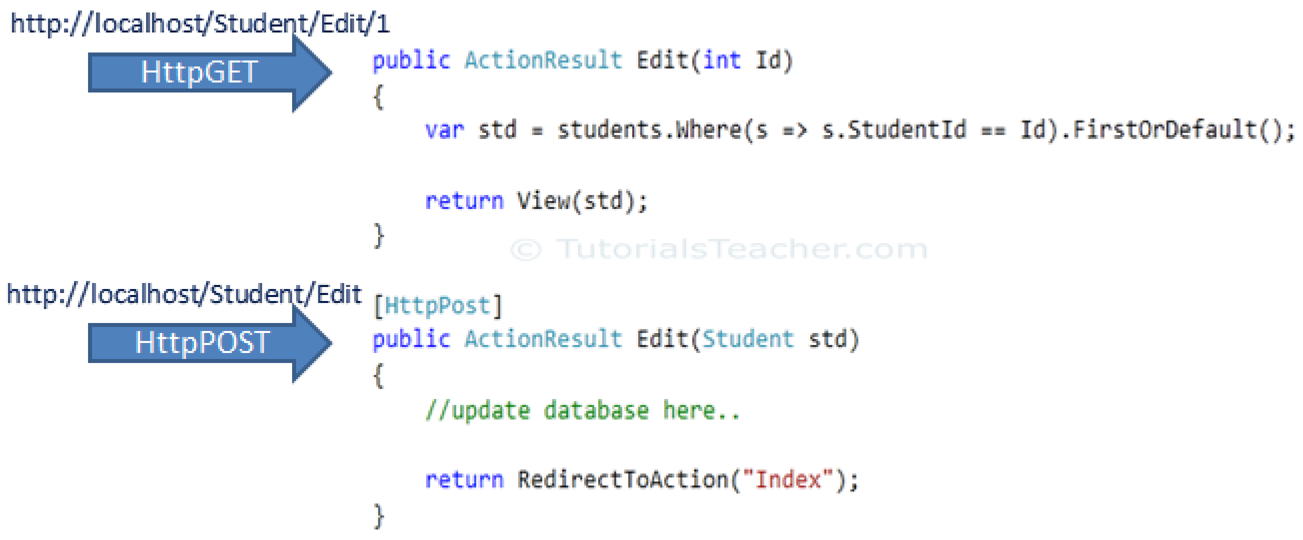
ActionVerbs

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

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

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

The following figure illustrates the HttpGET and HttpPOST action verbs.


http://localhost/Student/Edit/1
HttpGET
http://localhost/Student/Edit
HttpPOST

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

    return View(std);
}
```

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

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

ActionVerbs

The following table lists the usage of http methods:

Http method	Usage
GET	To retrieve the information from the server. Parameters will be appended in the query string.
POST	To create a new resource.
PUT	To update an existing resource.
HEAD	Identical to GET except that server do not return message body.
OPTIONS	OPTIONS method represents a request for information about the communication options supported by web server.
DELETE	To delete an existing resource.
PATCH	To full or partial update the resource.

Visit W3.org for more information on Http Methods.

The following example shows different action methods supports different ActionVerbs:

Example: ActionVerbs

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

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


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

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

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

    [HttpOptions]
    public ActionResult OptionsAction()
    {
        return View("Index");
    }

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

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

Example: AcceptVerbs

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

Model in ASP.NET MVC

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

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

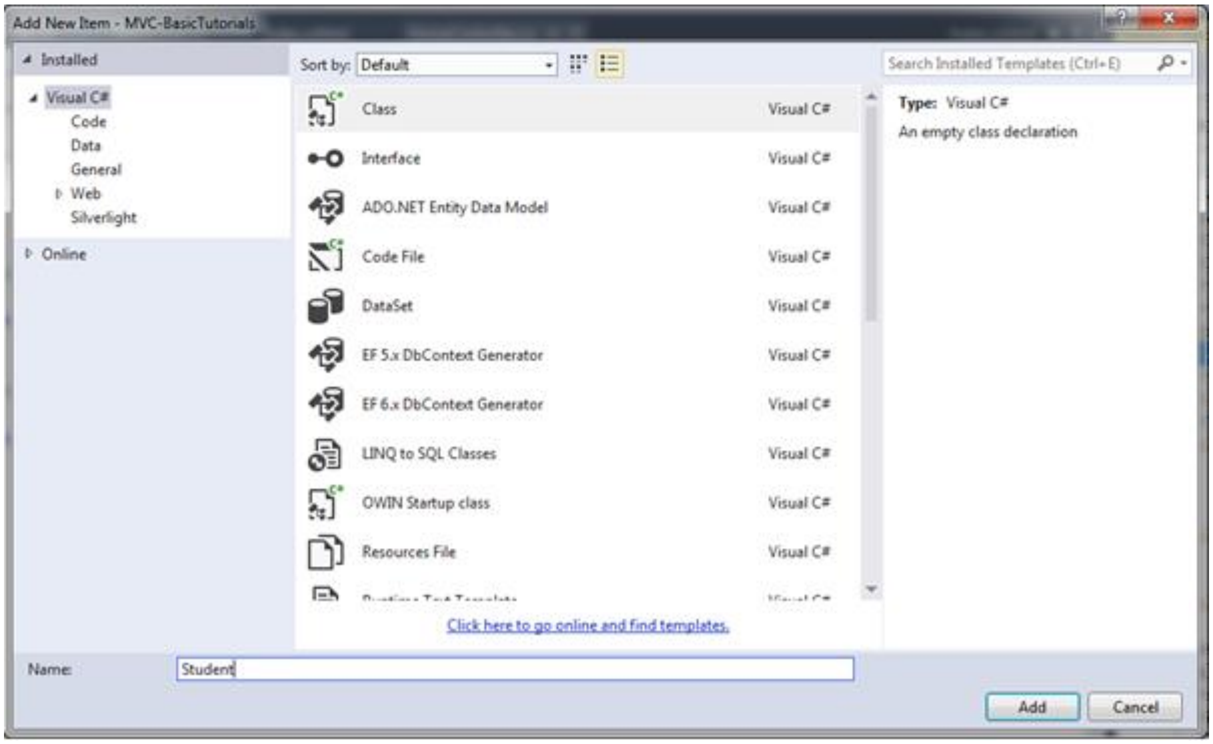
Model class holds data in public properties. All the Model classes reside in the Model folder in MVC folder structure.

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

Adding a Model

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

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


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

Type: Visual C#
An empty class declaration

Name: Student

Create Model Class

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

Example: Model class

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

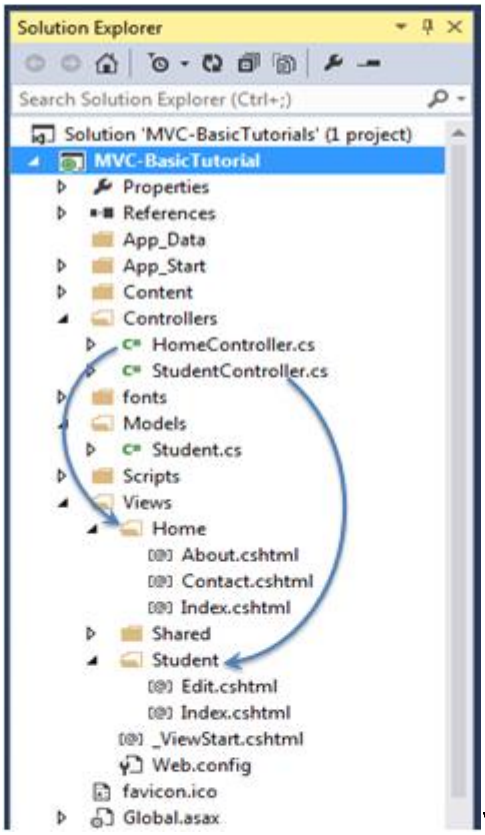
So in this way, you can create a model class which you can use in View. You will learn how to implement validations using model later.

View in ASP.NET MVC

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

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

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



View folders for Controllers

Note:

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

Razor View Engine

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

ASP.NET MVC supports following types of view files:

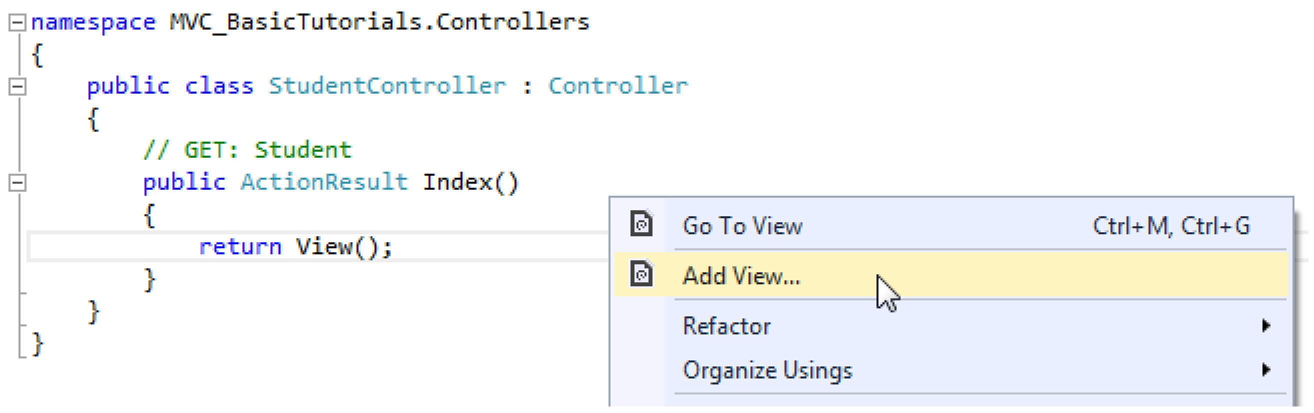
View file extension	Description
.cshtml	C# Razor view. Supports C# with html tags.
.vbhtml	Visual Basic Razor view. Supports Visual Basic with html tags.

View file extension	Description
.aspx	ASP.Net web form
.ascx	ASP.NET web control

Create New View

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

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

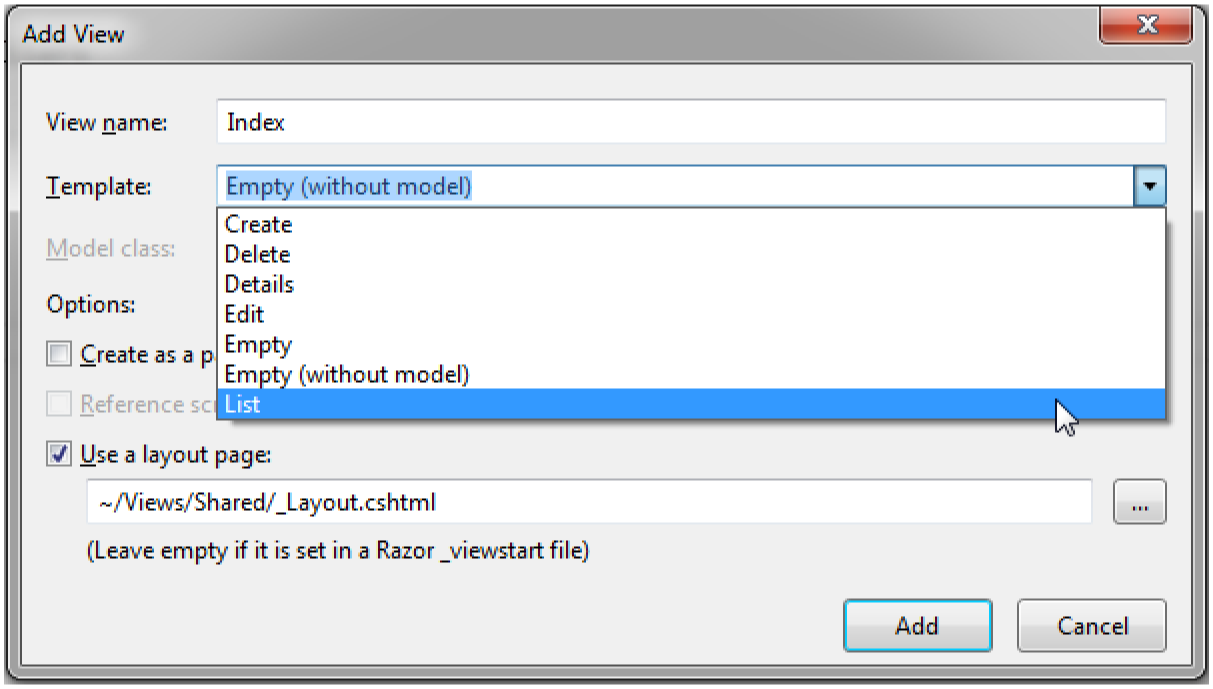


```
namespace MVC_BasicTutorials.Controllers
{
    public class StudentController : Controller
    {
        // GET: Student
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Create a View

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

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


Add View

View name: Index

Template: Empty (without model)

Create
Delete
Details
Edit
Empty
Empty (without model)
List

Model class:

Options:

☐ Create as a p

☐ Reference sc

☑ Use a layout page:

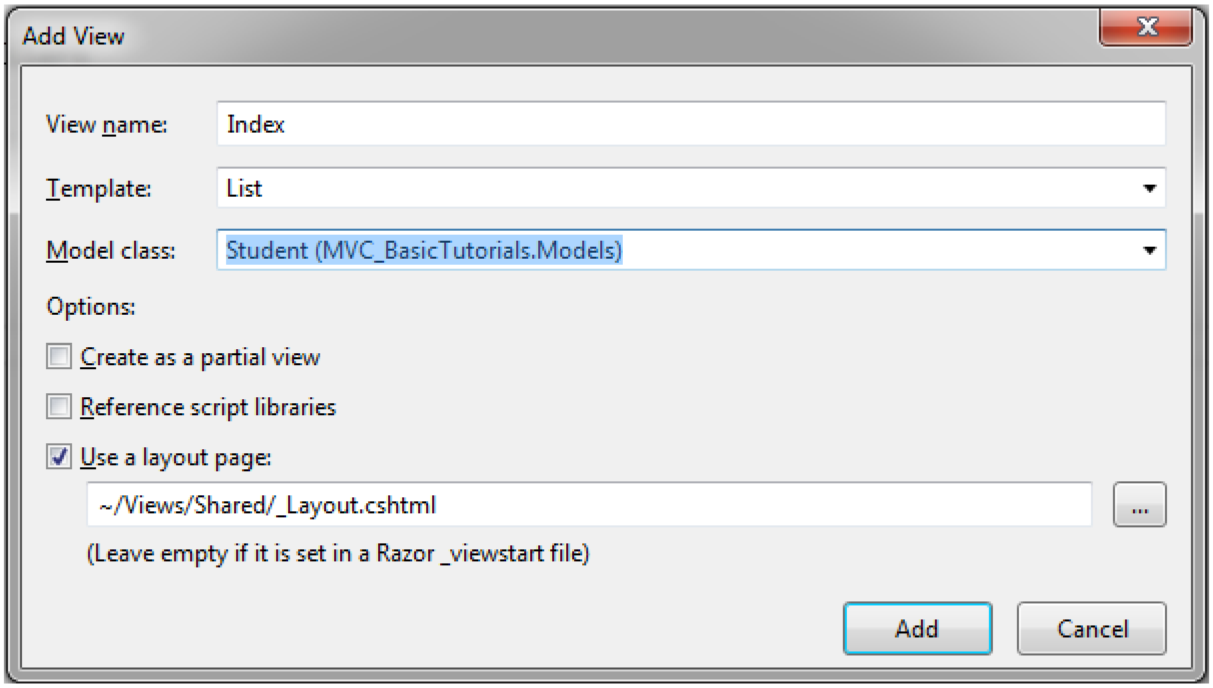
~/Views/Shared/_Layout.cshtml

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

Add | Cancel

View

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


Add View

View name: Index

Template: List

Model class: Student (MVC_BasicTutorials.Models)

Options:

☐ Create as a partial view

☐ Reference script libraries

☑ Use a layout page:

~/Views/Shared/_Layout.cshtml

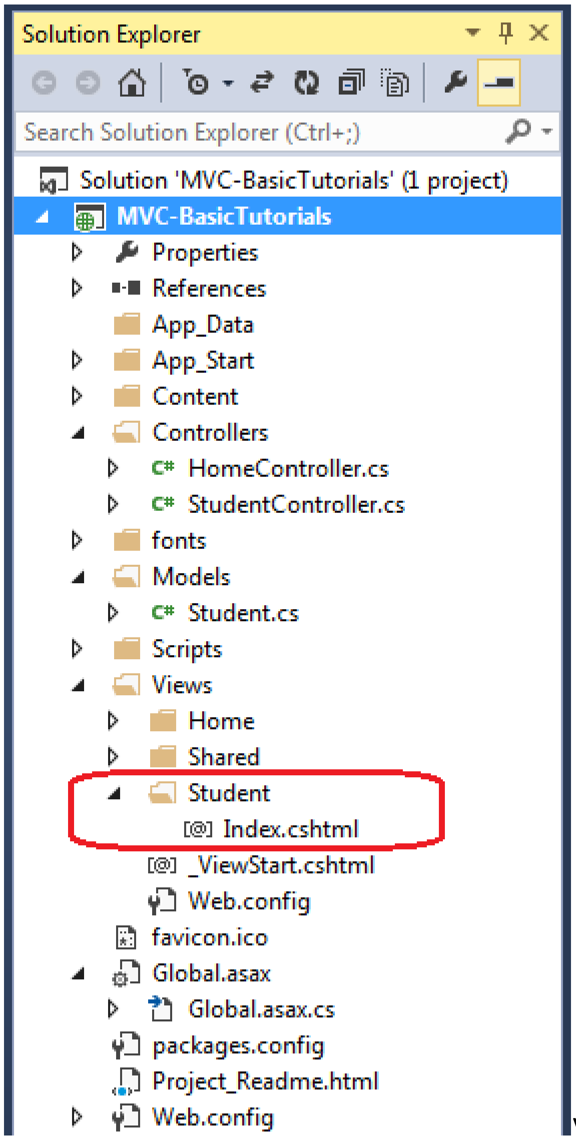
(Leave empty if it is set in a Razor _viewstart file)

Add | Cancel

View

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

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

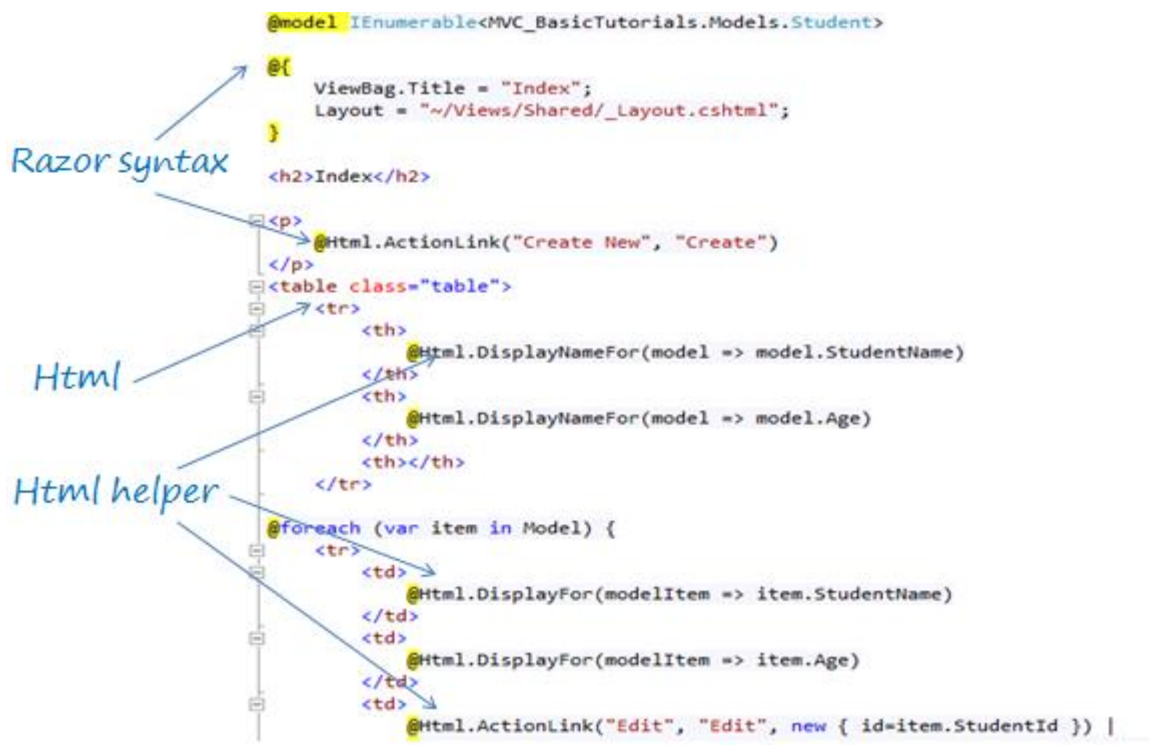


View

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

Views\Student\Index.cshtml:

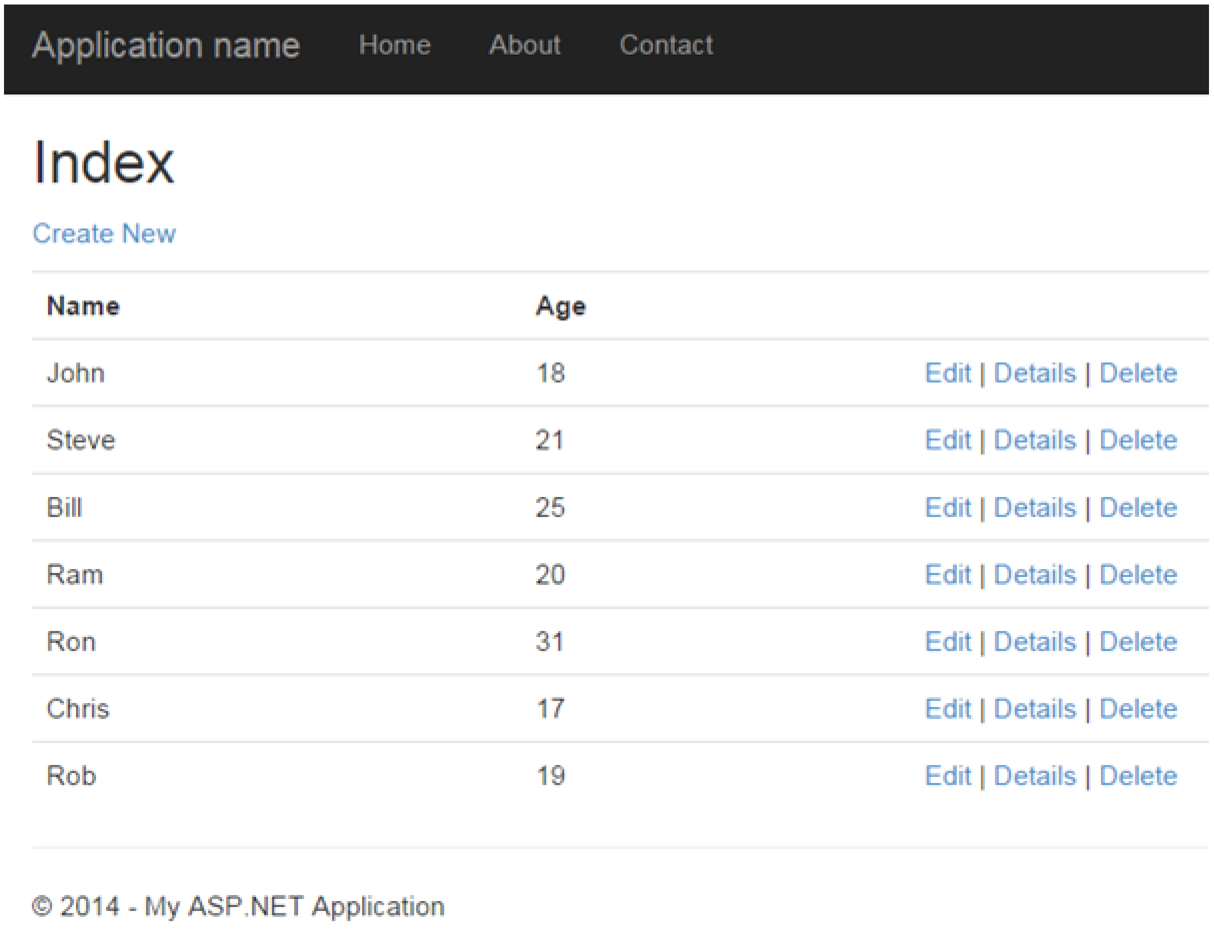
```
@model IEnumerable<MVC_BasicTutorials.Models.Student>

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

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Age)
        </th>
        <th></th>
    </tr>

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

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



```
@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 like below.



Index

Create New

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

© 2014 - My ASP.NET Application

Index View

Note:

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

Integrate Controller, View and Model

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

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

Example: StudentController

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

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

Example: Student Model class

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

Example: Index.cshtml to display student list

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

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

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.StudentName)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Age)
        </th>
        <th></th>
    </tr>

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

</table>
```

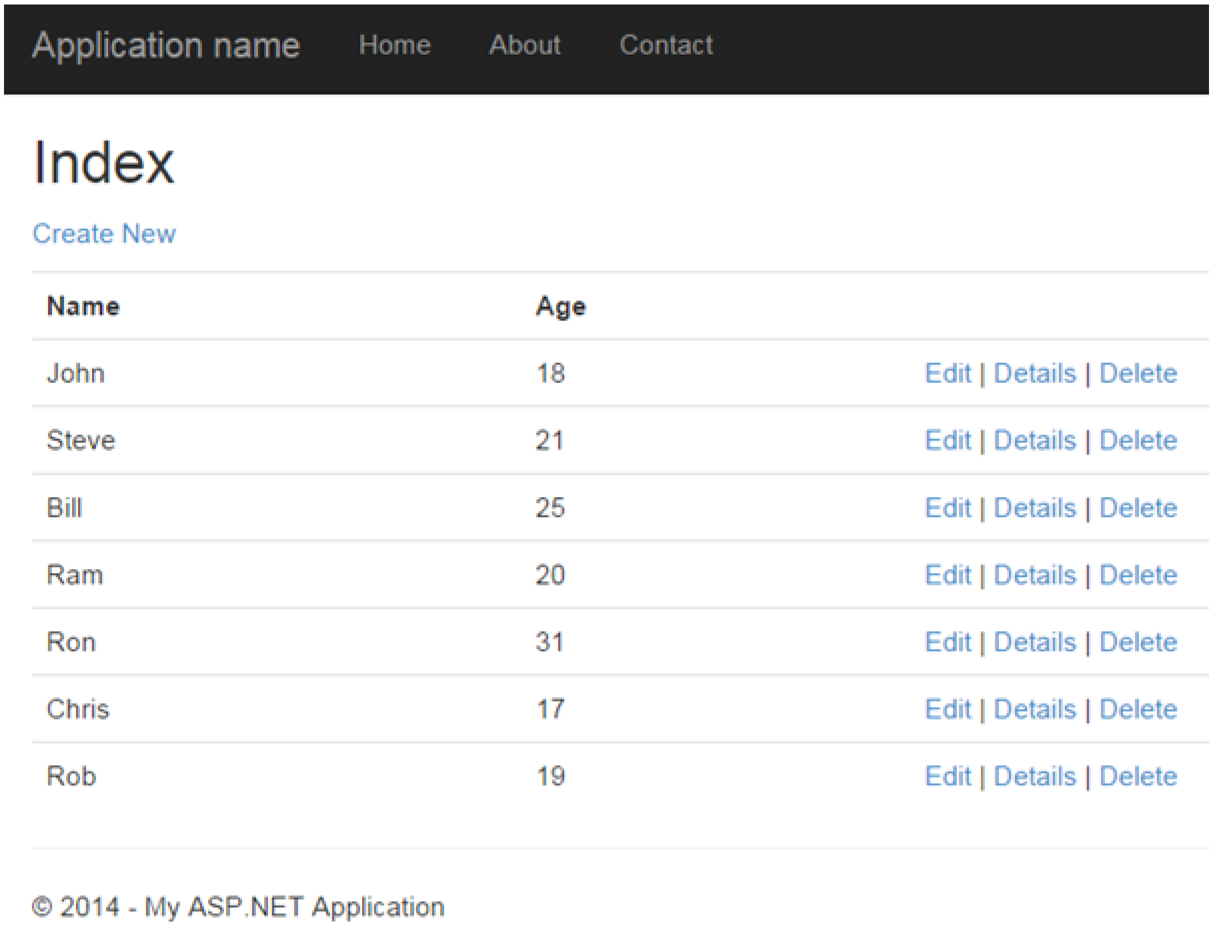
Now, to run it successfully, we need to pass a model object from controller to Index view. As you can see in the above Index.cshtml, it uses IEnumerable of Student as a model object. So we need to pass IEnumerable of Student model from the Index action method of StudentController class as shown below.

Example: Passing Model from Controller

```
public class StudentController : Controller
{
    // GET: Student
    public ActionResult Index()
    {
        var studentList = new List<Student>{
                            new Student() { StudentId = 1, StudentName = "John", Age
= 18 } ,
                            new Student() { StudentId = 2, StudentName = "Steve",  Age
= 21 } ,
                            new Student() { StudentId = 3, StudentName = "Bill",  Age
= 25 } ,
                            new Student() { StudentId = 4, StudentName = "Ram" , Age
= 20 } ,
                            new Student() { StudentId = 5, StudentName = "Ron" , Age
= 31 } ,
                            new Student() { StudentId = 4, StudentName = "Chris" , Age
= 17 } ,
                            new Student() { StudentId = 4, StudentName = "Rob" , Age
= 19 }
                        };
        // Get the students from the database in the real application

        return View(studentList);
    }
}
```

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

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



Index

Create New

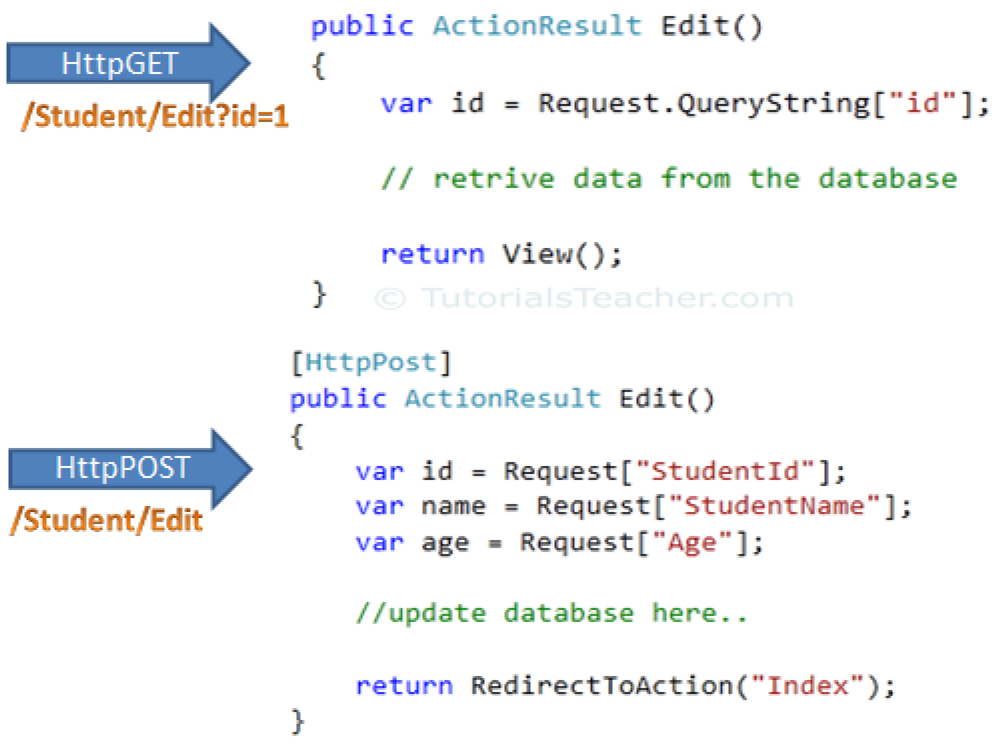
Name	Age	
John	18	Edit | Details | Delete
Steve	21	Edit | Details | Delete
Bill	25	Edit | Details | Delete
Ram	20	Edit | Details | Delete
Ron	31	Edit | Details | Delete
Chris	17	Edit | Details | Delete
Rob	19	Edit | Details | Delete

© 2014 - My ASP.NET Application

Model Binding

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

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



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

    // retrive data from the database

    return View();
}  © TutorialsTeacher.com

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

    //update database here..

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

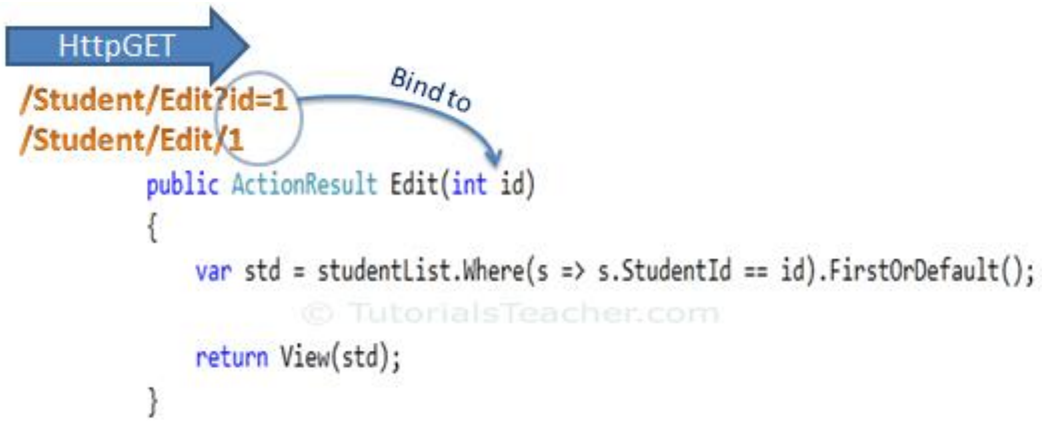
Accessing Request Data

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

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

Binding to Primitive type

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



```
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 paramters based on matching name.

For example, *http://localhost/Student/Edit?id=1&name=John* would map to id and name parameter of the following Edit action method.

Example: Convert QueryString to Action Method Parameters

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

    return View();
}
```

Binding to Complex type

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

Consider the following model classes.

Example: Model classes in C#

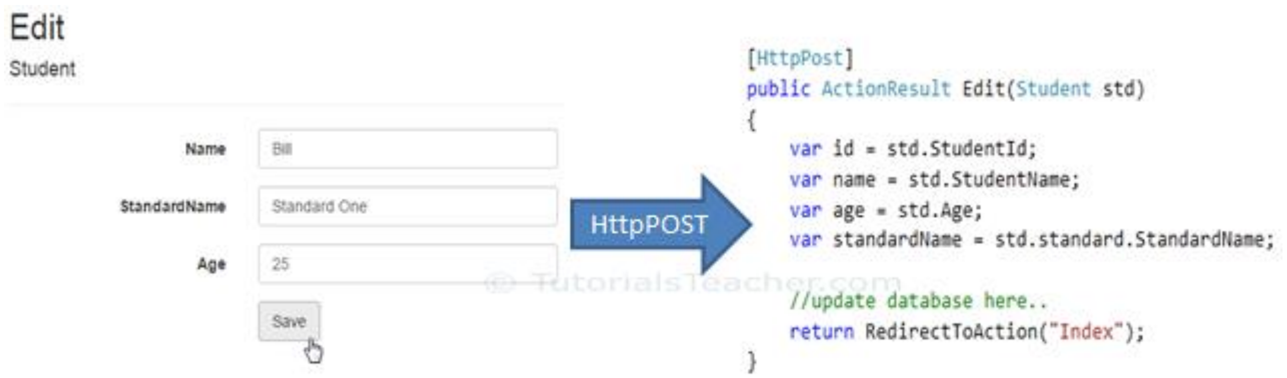
```
public class Student
{
    public int StudentId { get; set; }
    [Display(Name="Name")]
    public string StudentName { get; set; }
    public int Age { get; set; }
    public Standard standard { get; set; }
}

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

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

Example: Action Method with Class Type Parameter

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

    //update database here..

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

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



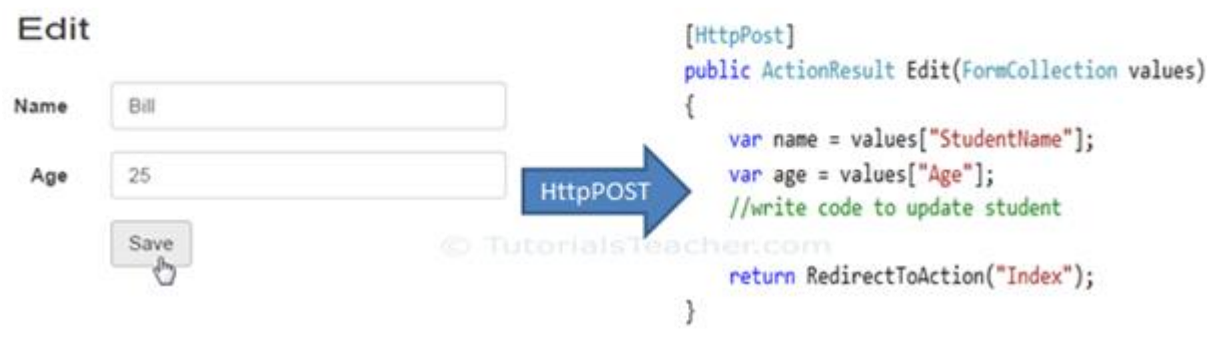
```
[HttpPost]
public ActionResult Edit(Student std)
{
    var id = std.StudentId;
    var name = std.StudentName;
    var age = std.Age;
    var standardName = std.standard.StandardName;

    //update database here..
    return RedirectToAction("Index");
}
```

Model Binding

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

FormCollection

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



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

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

Model Binding

Bind Attribute

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

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

Example: Binding Parameters

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

    //write code to update student

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

You can also use Exclude properties as below.

Example: Exclude Properties in Binding

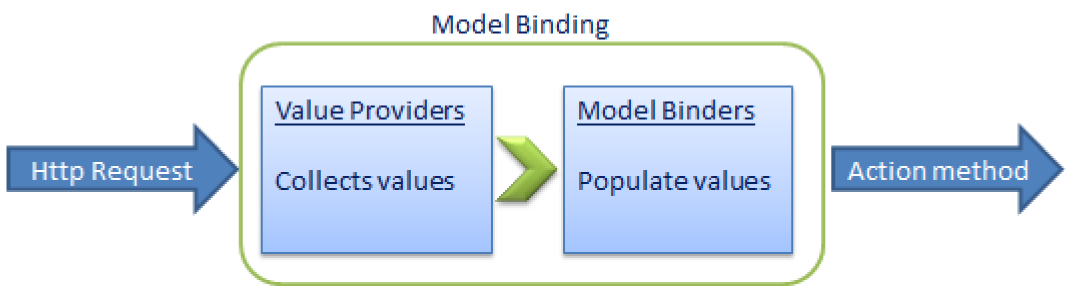
```
[HttpPost]
public ActionResult Edit([Bind(Exclude = "Age")] Student std)
{
    var name = std.StudentName;

    //write code to update student

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

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

Inside Model Binding

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

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



Model Binding in MVC

Default value provider collection evaluates values from the following sources:

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

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

Visit MSDN for detailed information on Model binding