**What is ASP.NET MVC?**

The ASP.NET MVC is a web application development framework provided by Microsoft which is built on top of the .NET Framework. We can use this ASP.NET MVC Framework to develop web applications that provide a clean separation of code. The ASP.NET MVC framework is the most extensible and customizable framework provided by Microsoft.

The ASP.NET MVC Framework is based on MVC (Model-View-Controller) Design Pattern. So the point that I need to highlight here is ASP.NET MVC is a Framework whereas MVC is a Design Pattern.

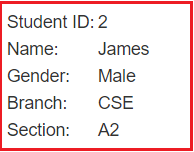
The ASP.NET MVC Framework is not built from ground zero. You can consider it as an alternative approach to our traditional ASP.NET Web Forms Framework. As it is built on top of the .NET Framework, developers enjoy almost all the ASP.NET features while working with the MVC application.

##### ****What is MVC?****

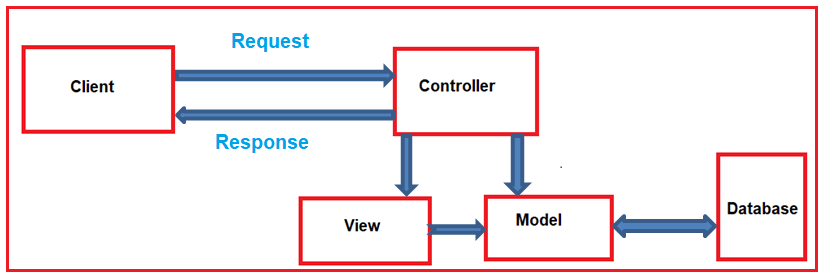
MVC stands for Model View and Controller. It is an architectural design pattern that means this design pattern is used at the architecture level of an application. So, the point that you need to remember is MVC is not a programming language, MVC is not a Framework, it is a design pattern. When we design an application, first we create the architecture of that application, and MVC plays an important role in the architecture of that particular application.

##### ****How does MVC Design Pattern works?****

Let us see an example to understand how the MVC pattern works in the ASP.NET MVC application. For example, we want to design an application, where we need to display the student details on a web page as shown below.



So, when we issue a request something like “**http://dotnettutorials.net/student/details/2**” from a web browser then the following things are happening in order to handle the request.



The controller is the component in the ASP.NET MVC application that actually receives the incoming HTTP request and then handles that request. In order to handle the incoming HTTP request, the controller does several things as follows.

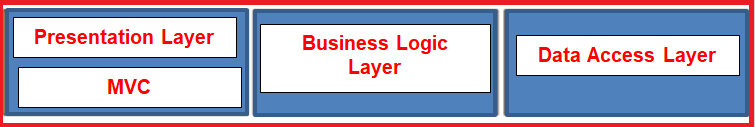
1. The controller creates the model object that is required by a view. The model is the component in the MVC design pattern that contains a set of classes to represent the domain data or business data as well as logic to manage the data.
2. The controller then selects a view to render the domain data or business data. The point that you need to remember is, while selecting a view, it is the responsibility of the controller to pass the model data.
3. In the MVC Design Pattern, the one and only responsibility of a view is to display the model data. So, the responsibility of a view is to generate the necessary HTML which will render the model data or business data. Once the HTML is generated by the view, then that HTML is then sent to the client via the controller who initially made the request.

**Where MVC is used in the real-time three-layer application?**

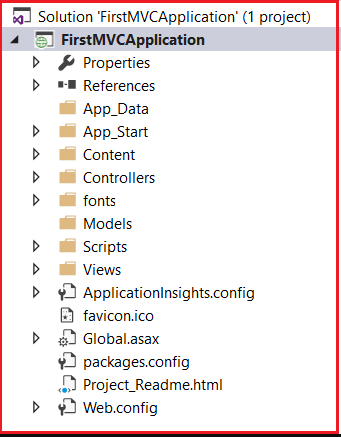
In general, a real-time application may consist of the following layers

1. **Presentation Layer:** This layer is responsible for interacting with the user.
2. **Business Layer:** This layer is responsible for implementing the core business logic of the application.
3. **Data Access Layer:** This layer is responsible for interacting with the database to perform the CRUD operations.

The MVC design pattern is basically used to implement the Presentation Layer of the application. Please have a look at the following diagram.



**MVC file and folder structure**



##### App\_Data:

The App\_Data folder of an ASP.NET MVC application contains application-related data files like .mdf files, LocalDB, XML files, etc. The most important point that you need to remember is IIS is never going to serve files from this App\_Data folder.

##### App\_Start:

The App\_Start folder of an ASP.NET MVC application contains configuration-related class files which are needed to be executed at the time of application starts. The classes like BundleConfig, FilterConfig, RouteConfig, IdentityConfig, etc are stored within this folder. We will discuss the use of each of these class files in detail in our upcoming articles.

##### Content:

The Content Folder of an ASP.NET MVC application contains static files such as image files, CSS files, icons files, etc. When we create a new ASP.NET MVC 5 application, then by default bootstrap.css, Site.css, and bootstrap.min.css files are included by Visual Studio as shown in the image below.

##### Global.asax:

The Global.asax file in an ASP.NET MVC application allows us to write the code that we want to run at the application level or you can say global level, such as Application\_BeginRequest, Application\_Error, Application\_Start, Session\_Start, Session\_End, etc. In our upcoming articles, we will discuss the use of these application-level events in detail.

##### Packages.config:

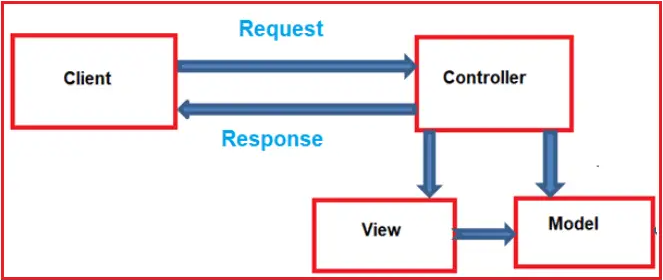
The Packages.config file in an ASP.NET MVC application is managed by NuGet Package Manager which will keep track of what packages and versions have been installed in your application.

##### Web.config:

The Web.config file of an ASP.NET MVC application is one of the most useful and important files which contains the application-level configurations such as connection strings, global variables, etc.

##### What is a Controller in an ASP.NET MVC application?

A controller in an ASP.NET MVC Application is a class having a set of public methods. These public methods of the controller class are called action methods or simple actions. These action methods in the ASP.NET MVC application are going to handle the incoming HTTP Requests.



As shown in the above diagram, the Controller is the component that is going to receive the incoming HTTP Request and then process that request. While processing the request, the controller does several things. It works with the model. Then it selects a view and if needed then passes the model object to the view. The view then generates the necessary HTML and the controller then sends the HTML back to the client who initially made the request. So, we can consider that the Controller is the heart of the MVC application.

##### What are ASP.NET MVC Views?

In the MVC pattern, the view component contains the logic to represent the model data as a user interface with which the end-user can interact. Typically, it creates the user interface with the data from the model provided to it by the controller. So you can consider the Views in ASP.NET MVC as HTML templates embedded with Razor syntax which generates HTML content that sends to the client.

##### What are the Models in ASP.NET MVC?

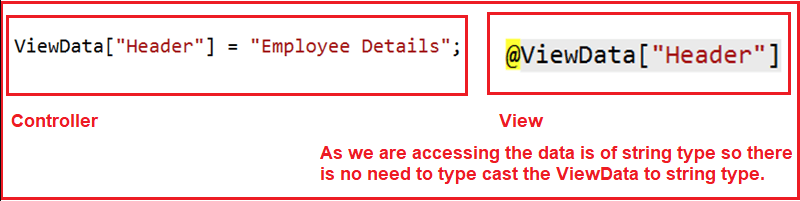
The Models in ASP.NET MVC application are the component that contains a set of classes that are used to represent the business data (or domain data) as well as logic to manage the business data. So in simple words, we can say that the model in ASP.NET MVC is used to manage the domain data i.e. the state of the application in memory.

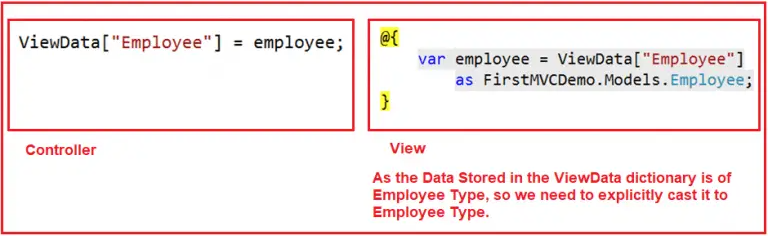
**What is ViewData in ASP.NET MVC?**

The ViewData in ASP.NET MVC Framework is a mechanism to pass the data from a controller action method to a view.



So we can say that the **ViewData in ASP.NET MVC** Framework is a dictionary object. As it is a dictionary object, it is going to store the data in the form of **key-value pairs** where each **key must be a string** and the value that we are passing to the dictionary is going to be stored in the form of an **object type**.





**Controller**

using FirstMVCDemo.Models;

using System.Web.Mvc;

namespace FirstMVCDemo.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer();

Employee employee = employeeBL.GetEmployeeDetails(102);

ViewData["Employee"] = employee;

ViewData["Header"] = "Employee Details";

return View();

}

}

}

**View**

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Page Title</title>

</head>

<body>

@{

var employee = ViewData["Employee"]

as FirstMVCDemo.Models.Employee;

}

<h2>@ViewData["Header"]</h2>

<table style="font-family:Arial">

<tr>

<td>Employee ID:</td>

<td>@employee.EmployeeId </td>

</tr>

<tr>

<td>Name:</td>

<td>@employee.Name</td>

</tr>

<tr>

<td>Gender:</td>

<td>@employee.Gender</td>

</tr>

<tr>

<td>City:</td>

<td>@employee.City</td>

</tr>

<tr>

<td>Salary:</td>

<td>@employee.Salary</td>

</tr>

<tr>

<td>Address:</td>

<td>@employee.Address</td>

</tr>

</table>

</body>

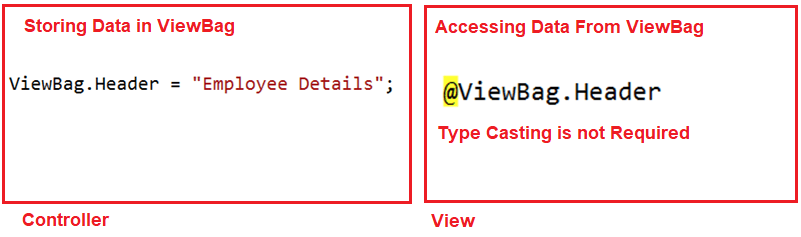
</html>

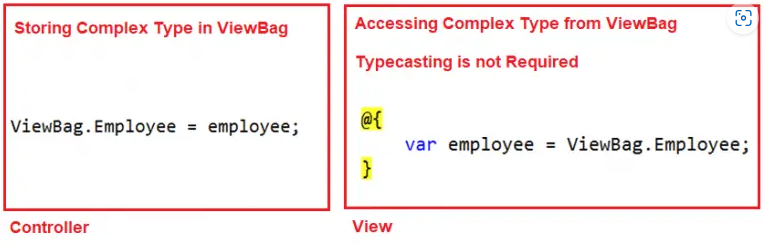
##### ****What is ViewBag in ASP.NET MVC?****

The ViewBag in ASP.NET MVC Framework is one of the mechanisms to pass the data from a controller action method to a view.



As you can see in the above image, the ViewBag is a dynamic property (a new feature introduced in C# 4.0). Dynamic data type means at runtime based on the value; it will decide the data type.





**Controller**

using FirstMVCDemo.Models;

using System.Web.Mvc;

namespace FirstMVCDemo.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer();

Employee employee = employeeBL.GetEmployeeDetails(101);

ViewBag.Employee = employee;

ViewBag.Header = "Employee Details";

return View();

}

}

}

**View**

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Page Title</title>

</head>

<body>

@{

var employee = ViewBag.Employee;

}

<h2>@ViewBag.Header</h2>

<table style="font-family:Arial">

<tr>

<td>Employee ID:</td>

<td>@employee.EmployeeId </td>

</tr>

<tr>

<td>Name:</td>

<td>@employee.Name</td>

</tr>

<tr>

<td>Gender:</td>

<td>@employee.Gender</td>

</tr>

<tr>

<td>City:</td>

<td>@employee.City</td>

</tr>

<tr>

<td>Salary:</td>

<td>@employee.Salary</td>

</tr>

<tr>

<td>Address:</td>

<td>@employee.Address</td>

</tr>

</table>

</body>

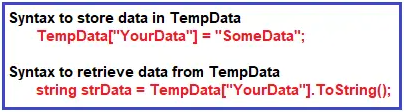
</html>

##### ****What is TempData in ASP.NET MVC?****

##### The limitation of both ViewData and ViewBag is they are limited to one HTTP request only. So, if redirection occurs then their values become null means they will lose the data they hold. In many real-time scenarios, we may need to pass the data from one HTTP Request to the next subsequent HTTP Request. For example, we may need to pass the data from one controller to another controller or one action method to another action method within the same controller. Then in such situations like this, we need to use TempData.

The TempData in ASP.NET MVC Framework is one of the mechanisms to pass a small amount of temporary data from a controller action method to a view as well as from a controller action method to another action method either within the same controller or to a different controller. TempData value will become null once the subsequent request is completed by default. But if you want then you can also change this default behavior. If you have a look at the definition Controller class, then you will find the following signature of the TempData property.





**Let’s understand TempData**

1st Request: <http://localhost:xxxxx/Employee/Method1>

2nd Request: <http://localhost:xxxxx/Employee/Method2>

3rd Request: <http://localhost:xxxxx/Employee/Method3>

As you can see in the above example, we add **Name and Age** in TempData in the first request, and in the second subsequent request, we access the data from the TempData which we stored in the first request. However, we can’t get the same data in the third request because TempData will be cleared out after the second request.

##### ****How to retain TempData values in the consecutive request?****

In order to retain the TempData value in the third consecutive request, we need to call **TempData.Keep()** method. Let’s see the use of **TempData.Keep()** method  with an example.

namespace FirstMVCDemo.Controllers

{

public class EmployeeController : Controller

{

public ActionResult Method1()

{

TempData["Name"] = "Pranaya";

TempData["Age"] = 30;

return View();

}

public ActionResult Method2()

{

string Name;

int Age;

if (TempData.ContainsKey("Name"))

Name = TempData["Name"].ToString();

if (TempData.ContainsKey("Age"))

Age = int.Parse(TempData["Age"].ToString());

TempData.Keep();

// do something with userName or userAge here

return View();

}

public ActionResult Method3()

{

string Name;

int Age;

if (TempData.ContainsKey("Name"))

Name = TempData["Name"].ToString();

if (TempData.ContainsKey("Age"))

Age = int.Parse(TempData["Age"].ToString());

// do something with userName or userAge here

return View();

}

}

}

# **Strongly Typed Views in ASP.NET MVC**

# In ASP.NET MVC, we can pass the data from the controller action method to a view in many different ways like ViewBag, ViewData, TempData and strongly typed model object. If we pass the data to a View using ViewBag, TempData, or ViewData, then that view becomes a loosely typed view. Here we will discuss how to create a strongly typed view in the ASP.NET MVC application.

##### ****Creating Strongly Typed View in MVC****

In order to create a strongly typed view in ASP.NET MVC application, we need to pass the model object as a parameter to the View() extension method. The Controller base class provide us the following overloaded versions of View() extension method which we can use to pass the model data from the controller action method to a view.

protected internal ViewResult View(object model)

protected internal ViewResult View(string viewName)

protected internal ViewResult View(string viewName, string masterName)

protected internal ViewResult View(string viewName, object model)

protected internal virtual ViewResult View(string viewName, string masterName, object model)

protected internal ViewResult View(IView view)

protected internal virtual ViewResult View(IView view, object model)

We are going to use the overloaded version which takes only the model object as the input parameter. As the input parameter is of object type, so we can pass any data. Modify the Index action method of the Home Controller as shown below to pass the Employee object as a parameter to the View extension method.

using FirstMVCDemo.Models;

using System.Web.Mvc;

namespace FirstMVCDemo.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

EmployeeBusinessLayer employeeBL = new EmployeeBusinessLayer();

Employee employee = employeeBL.GetEmployeeDetails(101);

ViewBag.Header = "Employee Details";

return View(employee);

}

}

}

##### ****Changes in Index.cshtml View****

**@model FirstMVCDemo.Models.Employee**

In order to create a strongly typed view in ASP.NET MVC Application, we need to specify the model type within the view by using the **@model** directive. As here, the Employee class is going to be the model so we need to specify the model directive as shown below.

@model FirstMVCDemo.Models.Employee

<!DOCTYPE html>

<html>

<head>

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

<title>Page Title</title>

</head>

<body>

<h2>@ViewBag.Header</h2>

<table style="font-family:Arial">

<tr>

<td>Employee ID:</td>

<td>@Model.EmployeeId </td>

</tr>

<tr>

<td>Name:</td>

<td>@Model.Name</td>

</tr>

<tr>

<td>Gender:</td>

<td>@Model.Gender</td>

</tr>

<tr>

<td>City:</td>

<td>@Model.City</td>

</tr>

<tr>

<td>Salary:</td>

<td>@Model.Salary</td>

</tr>

<tr>

<td>Address:</td>

<td>@Model.Address</td>

</tr>

</table>

</body>

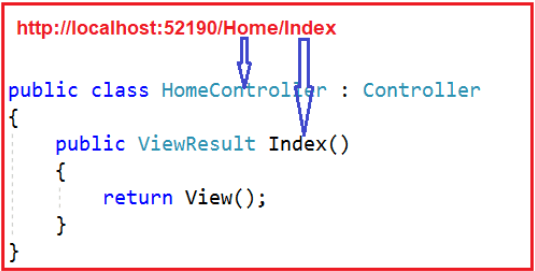
</html>

**Advantages of using Strongly Typed View in ASP.NET MVC Application**

1. Strongly Typed View in ASP.NET MVC provides compile-time error checkingas well as intelligence support.
2. If we misspell the property name, then it comes to know at compile time rather than at runtime.

##### ****What is Routing in ASP.NET MVC?****

The ASP.NET MVC Routing module is responsible for mapping the incoming browser requests (i.e. the incoming URL or incoming HTTP Requests) to a particular controller action method. This mapping is done by the routing rules defined for your application.



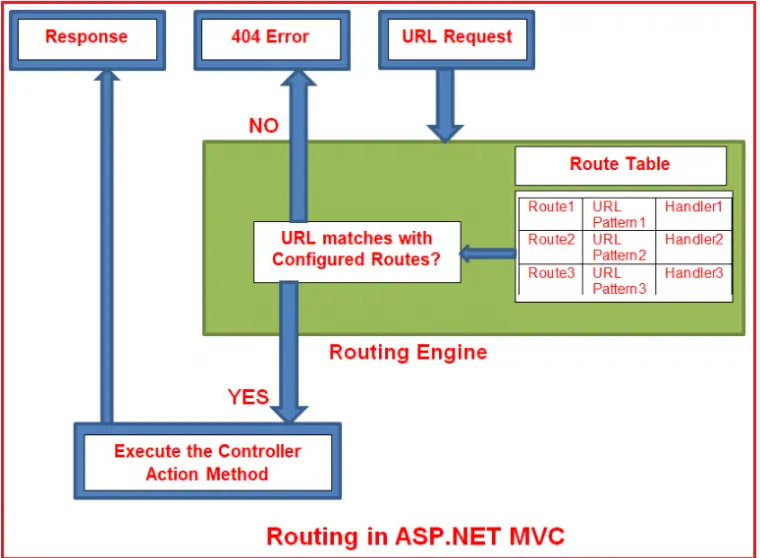
**What are the different types of Routing supported by ASP.NET MVC?**

In the ASP.NET MVC application, we can define routes in two ways. They are as follows:

1. **Convention Based Routing**
2. **Attribute-Based Routing.**

##### ****How Does the Routing work in ASP.NET MVC Application?****

Please have a look at the following diagram which illustrates the Routing Process in the ASP.NET MVC Application.



In simple words, we can say that Routing in ASP.NET MVC is a pattern matching mechanism that handles the incoming HTTP request (i.e. incoming URL) and figures out what to do with that incoming HTTP request.

When the client makes a request i.e. makes an HTTP Request, then that request is first received by the Routing Engine. Once the Routing engine receives an HTTP Request, then it figures out the URL Pattern of the incoming request and checks if that URL pattern is present in the Route table. If it found a matching URL pattern for the incoming request in the Route Table, then it fetches the corresponding handler information and forwards the request to the appropriate controller and action method. If there is no match found in the routing table for the incoming HTTP request URL Pattern, then it simply returns a 404 HTTP status code. The routing functionality is implemented in the **System.Web.Routing**.

##### ****How to Configure a Route in ASP.NET MVC?****

Every ASP.NET MVC application must configure (register) at least one route in the RouteConfig class and by default, the ASP.NET MVC Framework provides one default route. But you can configure as many routes as you want. You can register a route within the **RegisterRoutes** method of **RouteConfig** class, which you can find with the **RouteConfig.cs** class file under the **App\_Start** folder. You will find the following code in the RouteConfig class.

namespace WebApplication1

{

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 }

);

}

}

}

As you can see in the above code, the Routing is configured using the **MapRoute()** extension method of the **RouteCollection** class, where the Route name is “**Default**” and the URL pattern is “**{controller}/{action}/{id}**“.

The Defaults value for the **controller** is **Home**, and the default **action** method is **Index** and the **id** parameter is **optional**. This Route information i.e. the Route Name, URL Pattern, and handler information i.e. the controller name, action method name are stored in the Route table at the application start-up i.e. when the application runs for the first time.

##### ****Creating Custom Routes in ASP.NET MVC Application****

namespace WebApplication1

{

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 = "About", id = UrlParameter.Optional }

);

routes.MapRoute(

name: "Employee",

url: " Employee/{id}",

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

);

}

}

}

The following URLs will be mapped to the Employee route.

1. [**http://localhost:53605/Employee**](http://localhost:53605/Employee)
2. [**http://localhost:53605/Employee/Index**](http://localhost:53605/Employee/Index)
3. [**http://localhost:53605/Employee/Index/3**](http://localhost:53605/Employee/Index/3)

##### ****Route Constraints in ASP.NET MVC Application?****

The Route Constraint in ASP.NET MVC Routing allows us to apply a regular expression to a URL segment to restrict whether the route will match the request. In simple words, we can say that the Route constraint is a way to put some validation around the defined route. Suppose you have defined the following route in your application.

namespace WebApplication1

{

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 }

);

}

}

}

Now you want to restrict the incoming request URL with numeric id only. Now let’s see how we can do this with the help of regular expression in the ASP.NET MVC Application.

namespace WebApplication1

{

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 }

constraints: new { id = @"\d+" } //Restriction for id

);

}

}

}

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

##### ****Route Constraint for Restricting Controller and Actions****

Suppose you want to restrict the user for those URLs that have controller name with **H prefix** and action name should be only **Details or About**. Now let’s see how we can achieve this with the help of regular expression.

namespace WebApplication1

{

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 },

new { controller = "^H.\*", action = "^Details$|^About$" } //Restriction for controller and action

);

}

}

}

Now for this route, the routing engine will consider only those URLs which have controller name with **H prefix,** and action names should be only **Details** or **Index.** such as **http://dotnettutorials.net/Home/Index**, **http://dotnettutorials.net/Home/Details**, and **http://dotnettutorials.net/**, **http:// dotnettutorials.net/Home** else it will consider that URL is not matched with this route.

##### ****What is Attribute Routing in ASP.NET MVC?****

If we are defining Routes by using the [Route] attribute is called Attribute Routing. It provides you more control over the URIs by defining routes directly on actions and controllers in your ASP.NET MVC application.

##### ****Enabling Attribute Routing in ASP.NET MVC Application****

namespace WebApplication1

{

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

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

//Enabling attribute routing

routes.MapMvcAttributeRoutes();

routes.MapRoute(

name: "Default",

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

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

);

}

}

}

##### ****Using Route Attribute in ASP.NET MVC****

[HttpGet]

[Route("students/{studentID}/courses")]

public ActionResult GetStudentCourses(int studentID)

{

List<string> CourseList = new List<string>();

if (studentID == 1)

CourseList = new List<string>() { "ASP.NET", "C#.NET", "SQL Server" };

else if (studentID == 2)

CourseList = new List<string>() { "ASP.NET MVC", "C#.NET", "ADO.NET" };

else if (studentID == 3)

CourseList = new List<string>() { "ASP.NET WEB API", "C#.NET", "Entity Framework" };

else

CourseList = new List<string>() { "Bootstrap", "jQuery", "AngularJs" };

ViewBag.CourseList = CourseList;

return View();

}

With the above changes in place, you can access the above action by using **students/{studentID}/courses**URL**.**Here, **studentID**will be replaced with the values. Now navigate to the following URL and you should get the output as expected.

**http://localhost:58316/students/2/courses**

**Advantages of using Attribute Routing in ASP.NET MVC5?**

* It gives us more control over the URIs than convention-based routing. Creating URI patterns like hierarchies of resources (For example, students have courses, Departments have employees) is very difficult with convention-based routing.
* Reduces the chances for errors, if a route is modified incorrectly in RouteConfig.cs then it may affect the entire application’s routing.
* May decouple controller and action names from route entirely.
* Easy to map two routes pointing to the same action.

## **Route Prefix in ASP.NET MVC Attribute Routing**

## Let’s understand the need and use of Route Prefix with one example. Let’s modify the StudentController class as shown below.

namespace AttributeRoutingDemoInMVC.Controllers

{

public class StudentsController : Controller

{

static List<Student> students = new List<Student>()

{

new Student() { Id = 1, Name = "Pranaya" },

new Student() { Id = 2, Name = "Priyanka" },

new Student() { Id = 3, Name = "Anurag" },

new Student() { Id = 4, Name = "Sambit" }

};

[HttpGet]

[Route("students")]

public ActionResult GetAllStudents()

{

return View(students);

}

[HttpGet]

[Route("students/{studentID}")]

public ActionResult GetStudentByID(int studentID)

{

Student studentDetails = students.FirstOrDefault(s => s.Id == studentID);

return View(studentDetails);

}

[HttpGet]

[Route("students/{studentID}/courses")]

public ActionResult GetStudentCourses(int studentID)

{

List<string> CourseList = new List<string>();

if (studentID == 1)

CourseList = new List<string>() { "ASP.NET", "C#.NET", "SQL Server" };

else if (studentID == 2)

CourseList = new List<string>() { "ASP.NET MVC", "C#.NET", "ADO.NET" };

else if (studentID == 3)

CourseList = new List<string>() { "ASP.NET WEB API", "C#.NET", "Entity Framework" };

else

CourseList = new List<string>() { "Bootstrap", "jQuery", "AngularJs" };

ViewBag.CourseList = CourseList;

return View();

}

}

}

All the routes in the **StudentsController** start with the same prefix i.e. **students.** That means “**students**” is the common prefix for all the routes in Student Controller.

Here, instead of writing the common prefix “**students**” at each action method, we can specify the common prefix **students**for the entire Student Controller (for all the action methods of student controller) using the **[RoutePrefix]** attribute at the controller level as shown below.

namespace AttributeRoutingDemoInMVC.Controllers

{

[RoutePrefix("students")]

public class StudentsController : Controller

{

static List<Student> students = new List<Student>()

{

new Student() { Id = 1, Name = "Pranaya" },

new Student() { Id = 2, Name = "Priyanka" },

new Student() { Id = 3, Name = "Anurag" },

new Student() { Id = 4, Name = "Sambit" }

};

[HttpGet]

[Route]

//This will be translated to /students

public ActionResult GetAllStudents()

{

return View(students);

}

[HttpGet]

[Route("{studentID}")]

//This will be translated to /students/2

public ActionResult GetStudentByID(int studentID)

{

Student studentDetails = students.FirstOrDefault(s => s.Id == studentID);

return View(studentDetails);

}

[HttpGet]

[Route("{studentID}/courses")]

//This will be translated to /students/2/course

public ActionResult GetStudentCourses(int studentID)

{

List<string> CourseList = new List<string>();

if (studentID == 1)

CourseList = new List<string>() { "ASP.NET", "C#.NET", "SQL Server" };

else if (studentID == 2)

CourseList = new List<string>() { "ASP.NET MVC", "C#.NET", "ADO.NET" };

else if (studentID == 3)

CourseList = new List<string>() { "ASP.NET WEB API", "C#.NET", "Entity Framework" };

else

CourseList = new List<string>() { "Bootstrap", "jQuery", "AngularJs" };

ViewBag.CourseList = CourseList;

return View();

}

}

}

As you can see in the above code, we applied the **[RoutePrefix(“students”)]** attribute at the Controller level. This Route Prefix attribute eliminates the need to repeat the common prefix “**students**” on each and every controller action method.

##### ****Override Route Prefix Attribute in ASP.NET MVC Attribute Routing****

[Route("tech/teachers")]

public ActionResult GetTeachers()

{

List<Teacher> teachers = new List<Teacher>()

{

new Teacher() { Id = 1, Name = "James" },

new Teacher() { Id = 2, Name = "Patrik" },

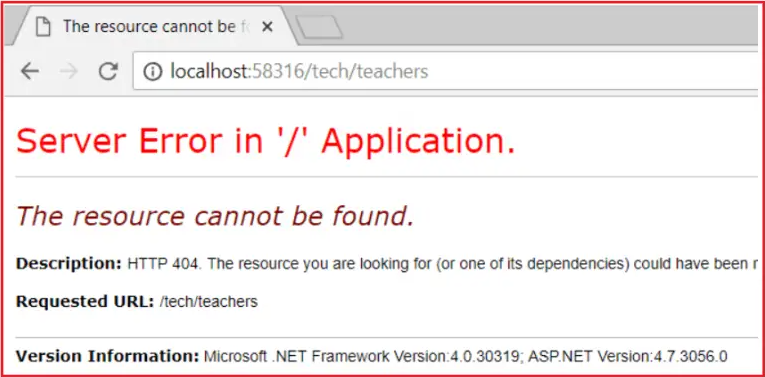
new Teacher() { Id = 3, Name = "Smith" }

};

return View(teachers);

}

If we use the **[Route]** attribute on the **GetTeachers()** method as shown in the above code and when we navigate to **tech/teachers**URL, we will get the following error.



But if we navigate to **/students/tech/teachers** then we will get the output as expected that is the list of teachers. This is because the **[RoutePrefix(“students”)]**attribute on StudentsController.  To override the RoutePrefix attribute, we need to use the **~** (tilde) symbol while defining the route as shown in the below code.

[Route("~/tech/teachers")]

public ActionResult GetTeachers()

{

List<Teacher> teachers = new List<Teacher>()

{

new Teacher() { Id = 1, Name = "James" },

new Teacher() { Id = 2, Name = "Patrik" },

new Teacher() { Id = 3, Name = "Smith" }

};

return View(teachers);

}

With this change **GetTeachers()** action method is now mapped to URI “**/tech/teachers**” as expected.

## **ASP.NET MVC Attribute Routing with Optional Parameter**

You can also define a URI parameter as optional by adding a **question mark (“?”)** to the route parameter.  If you have not specified any value for the studentName parameter in the URL, then it will store a null value in it and if you specified any value in the URL for the studentName parameter, then that value will be mapped to the studentName Parameter.

[Route("MVCTest/{studentName ?}")]

##### ****Route Names in Attribute Routing****

namespace AttributeRoutingDemoInMVC.Controllers

{

[Route("menu", Name = "mymenu")]

public class MenuController : Controller

{

public ActionResult MainMenu()

{

ViewBag.Message = "Menu Page";

return View();

}

}

}

You could generate a link using **Url.RouteUrl**:

<a href="@Url.RouteUrl("mymenu")">Main menu</a>

##### ****ActionLink HTML Helper in ASP.NET MVC****

Let us understand the need and use of ActionLink HTML Helper in ASP.NET MVC Application with an example. We want to display all the employees in a bulleted list as shown in the below image. Please have a look at the employee’s name, here we rendering the employee name as hyperlinks.



When we click on the above name hyperlink, then we need to redirect to the employee details page where we will display the full details of the employee as shown in the below image.



namespace CRUD\_OperationsInMVC.Controllers

{

public class EmployeeController : Controller

{

public ActionResult Index()

{

EmployeeDBContext dbContext = new EmployeeDBContext();

List<Employee> empList = dbContext.Employees.ToList();

return View(empList);

}

public ActionResult Details(int id)

{

EmployeeDBContext dbContext = new EmployeeDBContext();

Employee employee = dbContext.Employees.FirstOrDefault(x => x.EmployeeId == id);

return View(employee);

}

}

}

##### ****Index View****

@model IEnumerable<CRUD\_OperationsInMVC.Models.Employee>

<div style="font-family:Arial">

@{

ViewBag.Title = "Employee List";

}

<h2>Employee List</h2>

<ul>

@foreach (Employee employee in @Model)

{

<li>@Html.ActionLink(employee.Name, "Details", new { id = employee.EmployeeId })</li>

}

</ul>

</div>

##### ****Details view****

@model CRUD\_OperationsInMVC.Models.Employee

<div>

<h4>Employee Details</h4>

<hr />

<dl class="dl-horizontal">

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.Name)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.Gender)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.City)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.Salary)

</dd>

</dl>

</div>

<p>

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

</p>

##### ****What are HTML Helpers in ASP.NET MVC?****

An HTML Helper in ASP.NET MVC is an extension method of the HTML Helper class, which is used to generate HTML content in a view.  For example, if you want to generate a textbox with id=”firstname” and name=”firstname” then you can type all the required HTML in a view as shown below.

**<input type=”text” name=”firtsname” id=”firstname” />**

But in ASP.NET MVC, you can use the following “TextBox” HTML helper method in view to generating a text box.

**@Html.TextBox(“firstname”)**

**@Html.TextBox(“firstname”, “Pranaya”)**

At runtime, the above TextBox HTML helper method generates the following HTML

**<input id=”firstname” name=”firstname” type=”text” value=”Pranaya” />**

It is also possible to set the HTML attributes of a text box. If you want to do so, you need to use the following overloaded version of the TextBox HTML helper method.

**@Html.TextBox("firstname", "Pranaya", new { style = "background - color:Red; color: White; font - weight:bold", title = "Please enter your first name" })**

Notice here that we are passing the HTML attributes title and style as an anonymous type to the TextBox helper method. Some of the HTML attributes are reserved keywords. For example, read-only, class, etc. If you want to use these attributes within a Helper method, you need to prefix them with the “@” symbol, as shown in the example below.

**@Html.TextBox("firstname", "Pranaya", new { @class = "redtextbox", @readonly ="true" })**

If you want to generate a label for “Name” using the HTML helper method, then use the following HTML Helper method.

**@Html.Label(“Name”, “Name”)**

If you want to generate a textbox to enter a password, use the HTML Helper method.

**@Html.Password(“Password”)**

If you want to generate a multi-line textbox using Helper methods with 6 rows and 30 columns, then use the following HTML Helper method

**@Html.TextArea(“Comments”, “”, 6, 30, null)**

If you want to generate a hidden field, then use the following HTML Helper method

**@Html.Hidden(“id”)**

##### ****TextBoxFor() HTML Helper Method in MVC****

##### TextBoxFor() HTML Helper Method is a lambda expression which specifies the EmployeeName property of the Model object to bind with the textbox.

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

##### ****Differences between Html.TextBox and Html.TextBoxFor in ASP.NET MVC application?****

As we already discussed that the **@Html.TextBox()** is a loosely typed helper method whereas the **@Html.TextBoxFor()** is a strongly typed helper method.

##### The Html.TextBox() Helper method is not strongly typed and hence they don’t require a strongly typed view. This means that we can hardcode whatever name we want. On the other hand the Html.TextBoxFor() HTML Helper method is a strongly typed method and hence it requires a strongly typed view and the name should be given using the lambda expression.

The Strongly typed HTML helper methods also provide compile-time error checking. In real-time applications, we mostly prefer to use strongly typed HTML Helper methods.

##### ****DropDownList HTML Helper Method in ASP.NET MVC Application****

@Html.DropDownList("Departments", new List<SelectListItem>

{

new SelectListItem { Text = "IT", Value = "1", Selected=true},

new SelectListItem { Text = "HR", Value = "2"},

new SelectListItem { Text = "Payroll", Value = "3"}

}, "Select Department")

**How to set the Dropdown list values from the database in the ASP.NET MVC Application?**

namespace HTML\_HELPER.Models

{

public class Department

{

public int Id { get; set; }

public string Name { get; set; }

}

}

namespace HTML\_HELPER.Controllers

{

public class EmployeeController : Controller

{

public ActionResult Index()

{

//Get the data from the database

//Here we are creating Department list

List<Department> ListDepartments = new List<Department>()

{

new Department() {Id = 1, Name="IT" },

new Department() {Id = 2, Name="HR" },

new Department() {Id = 3, Name="Payroll" },

};

// Retrieve departments and build SelectList

ViewBag.Departments = new SelectList(ListDepartments, "Id", "Name");

return View();

}

}

}

**Or you can also do the same thing in the following way**

namespace HTML\_HELPER.Controllers

{

public class EmployeeController : Controller

{

public ActionResult Index()

{

List<SelectListItem> items = new List<SelectListItem>();

items.Add(new SelectListItem { Text = "IT", Value = "1" });

items.Add(new SelectListItem { Text = "HR", Value = "2" });

items.Add(new SelectListItem { Text = "Payroll", Value = "2" });

ViewBag.Departments = items;

return View();

}

}

}

@Html.DropDownList("Departments", @ViewBag.Departments as List<SelectListItem>, "Select Department", new { @class = "form - control"})

##### ****RadioButtonList HTML Helper Method in ASP.NET MVC Application****

Models

public class About

{

public string SelectedGender { get; set; }

public List<Gender> Genders { get; set; }

}

public class Gender

{

public int Id { get; set; }

public string Name { get; set; }

}

Controller

public ActionResult About()

{

List<Gender> ListDepartments = new List<Gender>()

{

new Gender() {Id = 1, Name="Male" },

new Gender() {Id = 2, Name="Female" },

new Gender() {Id = 3, Name="Transgender" },

};

About objAbout=new About();

objAbout.SelectedGender = "1";

objAbout.Genders = ListDepartments; ;

return View(objAbout);

}

**View**

@model WebApplication1.Models.About

@foreach (var gender in Model.Genders)

{

@Html.RadioButtonFor(m => m.SelectedGender, gender.Id)@gender.Name

}

**What are Partial Views in MVC Application?**

The Partial Views in ASP.NET MVC Application are the views that are rendered within another view. The HTML output generated by the partial view is rendered into the calling (or parent) view. Like views, partial views use the *.*cshtml file extension.

**How Can We Call/Display a Partial View?**

We can call or display a partial view within a view mainly in five ways. They are as follows:

1. **Html.RenderPartial**
2. **Html.Partial**
3. **Html.RenderAction**
4. **Html.Action**
5. **jQuery load function**

**Model**

public class Product

{

public long ProductID { get; set; }

public string Name { get; set; }

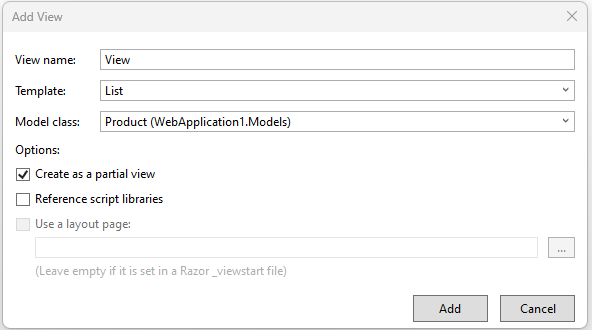
public string Category { get; set; }

public string Description { get; set; }

public decimal Price { get; set; }

}

**Partial View**

****

@model IEnumerable<WebApplication1.Models.Product>

<p>

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

</p>

<table class="table">

<tr>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

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

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

</td>

</tr>

}

</table>

**Index View**

@model IEnumerable<WebApplication1.Models.Product>

@{

ViewBag.Title = "Home Page";

}

@Html.Partial("ProductDetails", Model)

**Controller**

public ActionResult Index()

{

List<Product> products = new List<Product>()

{

new Product { ProductID =1, Name ="Product 1", Category = "Category 1", Description ="Description 1", Price = 10m},

new Product { ProductID =2, Name ="Product 2", Category = "Category 1", Description ="Description 2", Price = 20m},

new Product { ProductID =3, Name ="Product 3", Category = "Category 1", Description ="Description 3", Price = 30m},

new Product { ProductID =4, Name ="Product 4", Category = "Category 2", Description ="Description 4", Price = 40m},

new Product { ProductID =5, Name ="Product 5", Category = "Category 2", Description ="Description 5", Price = 50m},

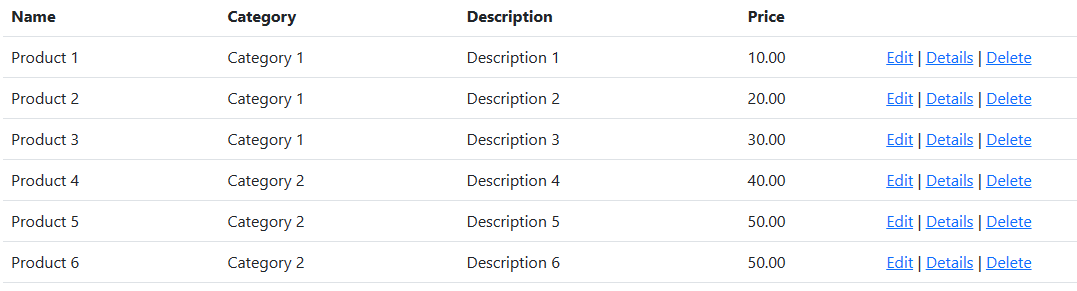
new Product { ProductID =6, Name ="Product 6", Category = "Category 2", Description ="Description 6", Price = 50m}

};

return View(products);

}

**Output**

****

## **Different Ways to Render Partial View in ASP.NET MVC**

Method 1:

@Html.Partial("ProductDetails", Model)

It works when you have the partial view located in the Shared folder. If your partial view is located in a different folder, then you will have to mention the full path of view as shown below.

@Html.Partial("~/Views/Shared/ProductDetails.cshtml", Model)

Method 2:

@{Html.RenderPartial("ProductDetails", Model);}

It works when you have the partial view located in the Shared folder. If your partial view is located in a different folder, then you will have to mention the full path of view as shown below.

@{Html.RenderPartial("~/Views/Home/ProductDetails.cshtml", Model);}

Method 3: Return To Partial View

public ActionResult Index()

{

List<Product> products = new List<Product>()

{

new Product { ProductID =1, Name ="Product 1", Category = "Category 1", Description ="Description 1", Price = 10m},

new Product { ProductID =2, Name ="Product 2", Category = "Category 1", Description ="Description 2", Price = 20m},

new Product { ProductID =3, Name ="Product 3", Category = "Category 1", Description ="Description 3", Price = 30m},

new Product { ProductID =4, Name ="Product 4", Category = "Category 2", Description ="Description 4", Price = 40m},

new Product { ProductID =5, Name ="Product 5", Category = "Category 2", Description ="Description 5", Price = 50m},

new Product { ProductID =6, Name ="Product 6", Category = "Category 2", Description ="Description 6", Price = 50m}

};

return PartialView("ProductDetails", products);

}

Output:



## **Razer View Syntax in MVC**

## In this article, we will discuss razor view syntax in the ASP.NET MVC application. Use @ symbol to switch between C# code and HTML.

@for (int i = 1; i <= 10; i++)

{

<b>@i</b>

}

**Output:**  
**1 2 3 4 5 6 7 8 9 10**

##### ****Why do we need Data Annotation Attributes in ASP.NET MVC?****

##### Now a day’s, it’s a challenging job for a web developer to validate the user input for any Web application. As web developers, we not only validate the business logic at the client-side that is in the browser, but also we need to validate the business logic running on the Server. That means as a developer we need to validate the business logic both at the client-side as well as server-side.

The client-side validation of the business logic gives the users immediate feedback on the information they entered into a web page and which is an expected feature in today’s web applications. Along the same line, the server-side validation logic is in place because we never trust the information coming from the network.

###### The System.ComponentModel.DataAnnotations assembly has many built-in validation attributes, for example:

1. **Required**
2. **Range**
3. **RegularExpression,**
4. **Compare**
5. **StringLength**
6. **Data type**

##### ****Required Attribute in ASP.NET MVC****

[Required]

public string FirstName

{

get;

set;

}

[Required(ErrorMessage = "First Name is Required")]

public string FirstName

{

get;

set;

}

##### ****StringLength Attribute in ASP.NET MVC****

[Required(ErrorMessage = "Last Name is Required")]

[StringLength(30)]

public string LastName

{

get;

set;

}

[Required(ErrorMessage = "Last Name is Required")]

[StringLength(30, MinimumLength = 4, ErrorMessage = "Last name should be between 4 and 30 characters")]

public string LastName

{

get;

set;

}

##### ****Regular Expression Attribute in ASP.NET MVC****

[Required(ErrorMessage = "Email id is required")]

[RegularExpression(@"\A(?:[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)\*@(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?)\Z",ErrorMessage = "Please enter a valid email address")]

public string EmailId

{

get;

set;

}

##### ****Range Attribute in ASP.NET MVC Application****

[Range(25, 60, ErrorMessage = "Age must be between 25 and 60")]

public int Age

{

get;

set;

}

#### **DataType Attribute in ASP.NET MVC**

#### DataType Attribute in ASP.NET MVC Framework enables us to provide the runtime information about the specific purpose of the properties. For example, a property of type string can have various scenarios as it might hold an Email address, URL, or password. There are various data types that include Currency, Date, Time, Password and MultilineText, etc. Let’s see some of the examples of using the DataType attribute.

[DataType(DataType.PostalCode, ErrorMessage = "Please Enter a valid PIN/ZIP Code")]

public string PostalCode

{

get;

set;

}

[DataType(DataType.Url, ErrorMessage = "Please Enter a valid URL")]

public string URL

{

get;

set;

}

[DataType(DataType.Password)]

public string Password

{

get;

set;

}

[DataType(DataType.PhoneNumber, ErrorMessage = "Please Enter a valid Phone Number")]

public string Mobile

{

get;

set;

}

#### **Compare** **Attribute in ASP.NET MVC Application**

[DataType(DataType.Password)] 1

[Required(ErrorMessage = "Password is Required")]

public string Password

{

get;

set;

}

[DataType(DataType.Password)]

[Required(ErrorMessage = "Confirm Password is Required")]

[Compare("Password", ErrorMessage = "Password and Confirm Password do not match"]

public string ConfirmPassword

{

get;

set;

}

#### **Validation Message and Validation Summary in ASP.NET MVC**

#### 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.

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

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

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

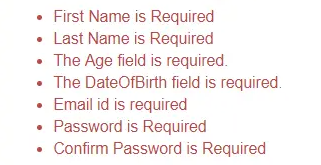
The ValidationMessageFor() method will only display an error if you have configured the DataAnnotations attribute to the specifed property in the model class.

##### ****ValidationSummary Attribute in ASP.NET MVC Application****

The ValidationSummary helper method generates an unordered list (ul element) of validation messages that are in the ModelStateDictionary object. The ValidationSummary can be used to display all the error messages for all the fields. It can also be used to display custom error messages.

@Html.ValidationSummary(false, "Please fix the following errors and then submit the form")

The following figure shows how ValidationSummary displays the error messages. Now run the application and submit the form without filling the data. It will show the errors as shown below

****

**Form Example**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

<div class="form-horizontal">

<hr />

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

<div class="form-group">

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

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

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

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

</div>

</div>

<div class="form-group">

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

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

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

@Html.ValidationMessageFor(model => model.LastName, "", 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="Create" class="btn btn-danger" />

</div>

</div>

</div>

}

## **Attributes in ASP.NET MVC Application**

I am going to discuss **Built-in Attributes in ASP.NET MVC** applications.

* Display
* DisplayName
* DisplayFormat
* ScaffoldColumn
* DataTypeAttribute,
* DisplayColumnAttribute
* UIHint
* HiddenInput
* ReadOnly

**Model**

public class Employee

{

public string FullName { get; set; }

public DateTime? HireDate { get; set; }

public string Gender { get; set; }

public int? Salary { get; set; }

public string EmailAddress { get; set; }

public string PersonalWebSite { get; set; }

}

**View**

@model WebApplication1.Models.Employee

<div>

<h4>Employee Details</h4>

<dl class="dl-horizontal">

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.FullName)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.HireDate)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.Gender)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.EmailAddress)

</dd>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.Salary)

</dd>

<dt>

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

</dt>

<dd>

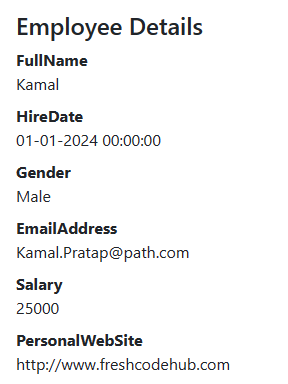
@Html.DisplayFor(model => model.PersonalWebSite)

</dd>

</dl>

</div>

**Output**



Notice that the output is not that pretty. We can control the display of data in a view using display attributes that are found in the **System.ComponentModel.DataAnnotations** namespace. We need to create a “**Employee**” class and we need to decorate that class with the display attributes.

**Model**

public class Employee

{

[DisplayName("Full Name")]

public string FullName { get; set; }

[DataType(DataType.Date)]

public DateTime? HireDate { get; set; }

[DisplayFormat(NullDisplayText = "Gender not specified")]

public string Gender { get; set; }

[DataType(DataType.Currency)]

public int? Salary { get; set; }

[DataType(DataType.EmailAddress)]

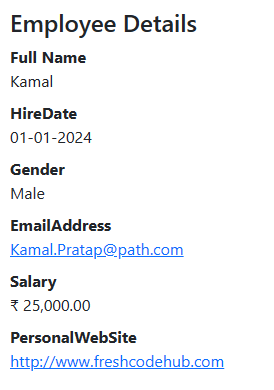
public string EmailAddress { get; set; }

[DataType(DataType.Url)]

public string PersonalWebSite { get; set; }

}

**Output**

****

**Understanding Action Results**

Action Result is the return type of an action method. A controller action returns something called an action result. An action result is what a controller action returns in response to a browser request.

The ASP.NET MVC framework supports several types of action results including:

1. **ViewResult** - Represents HTML and markup.
2. **EmptyResult** - Represents no result.
3. **RedirectResult** - Represents a redirection to a new URL.
4. **JsonResult** - Represents a JavaScript Object Notation result that can be used in an AJAX application.
5. **JavaScriptResult** - Represents a JavaScript script.
6. **ContentResult** - Represents a text result.
7. **FileContentResult** - Represents a downloadable file (with the binary content).
8. **FilePathResult** - Represents a downloadable file (with a path).
9. **FileStreamResult** - Represents a downloadable file (with a file stream).

**Why is ActionResult an abstract class in ASP.NET MVC?**

It’s because different controller action methods can return different types of results as per the business needs and still the ASP.NET MVC Framework handles them properly. If you mention the return type of an action method as ActionResult, then this action method can return any type which is derived from the ActionResult abstract class.

**View Result in ASP.NET MVC Application**

public ActionResult Index()

{

Employee employee = new Employee();

employee.FullName = "Kamal";

employee.HireDate = Convert.ToDateTime("01-01-2024");

employee.Gender = "Male";

employee.Salary = 25000;

employee.EmailAddress = "Kamal.Pratap@path.com";

employee.PersonalWebSite = "http://www.freshcodehub.com";

return View(employee);

}

**Partial View Result in ASP.NET MVC Application**

public ActionResult Index()

{

Employee employee = new Employee();

employee.FullName = "Kamal";

employee.HireDate = Convert.ToDateTime("01-01-2024");

employee.Gender = "Male";

employee.Salary = 25000;

employee.EmailAddress = "Kamal.Pratap@path.com";

employee.PersonalWebSite = "http://www.freshcodehub.com";

return PartialView("EmployeeDetails", employee);

}

##### JSON Result in ASP.NET MVC

The JSON result is one of the most important Action results in the ASP.NET MVC application. This action result returns the data in JSON Format i.e. in the form of key-value pairs. And moreover, we need to call this method using Ajax from a view.

[HttpGet]

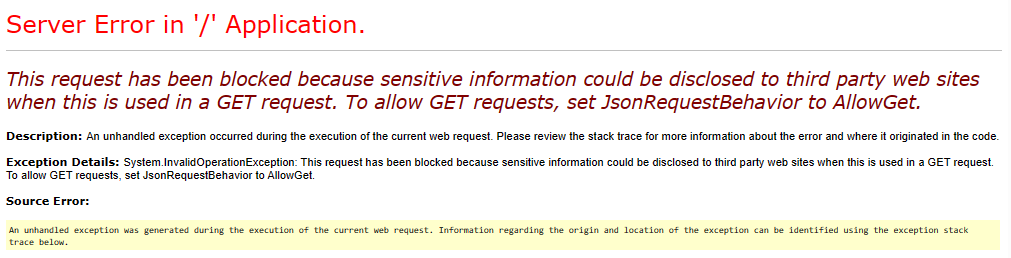
public JsonResult Index()

{

return Json(new { Name = "John Smith", ID = 4, DateOfBirth = new DateTime(1999, 12, 31) });

}

Now, run the application and navigates to **Home/Index** in the URL and you will get the following error.

****

MVC is trying to protect you here; it doesn’t want you to share JSON information over a GET request because it could potentially contain sensitive information. This is attempting to protect you from an exploit known as JSON Hijacking. You can turn off this error by modifying the Index method as shown below.

[HttpGet]

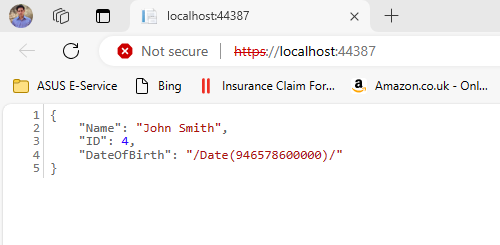
public JsonResult Index()

{

return Json(new { Name = "John Smith", ID = 4, DateOfBirth = new DateTime(1999, 12, 31) },

JsonRequestBehavior.AllowGet);

}

****

[HttpGet]

public ActionResult Index()

{

var persons = new List<Person>

{

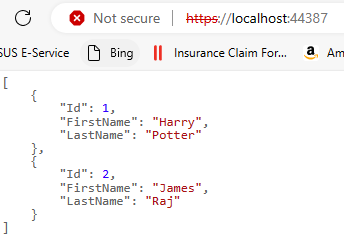
new Person{Id=1, FirstName="Harry", LastName="Potter"},

new Person{Id=2, FirstName="James", LastName="Raj"}

};

return Json(persons, JsonRequestBehavior.AllowGet);

}

****

**JavaScriptResult in Asp.Net MVC**

**Controller**

public ActionResult Index()

{

return View();

}

[HttpGet]

public JavaScriptResult WarningMessage()

{

var msg = "alert('Are you sure want to Continue?');";

return new JavaScriptResult() { Script = msg };

}

**View**

@{

ViewBag.Title = "Home Page";

}

<script src="//code.jquery.com/jquery-1.10.2.js"></script>

<script>

$(document).ready(function () {

$("button").click(function () {

$.getScript("/Home/WarningMessage");

});

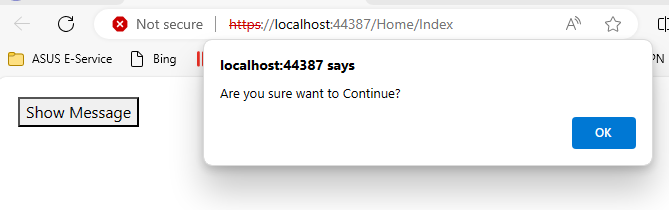
});

</script>

<button>Show Message</button>

**Output**

And when you click on the Show Message button, it will display the popup as shown in the below image.

****

**File Result in Asp.Net MVC**

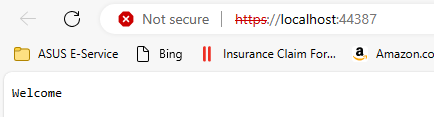
If you want to return a file from your action method then you need to use either FileResult or ActionResult as the return type of your action method.

public FileResult Index()

{

return File(Url.Content("~/Content/SimpleText.txt"), "text/plain");

}

****

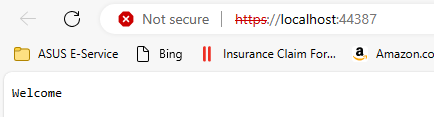
public FileResult Index()

{

byte[] fileBytes = System.IO.File.ReadAllBytes(Server.MapPath("~/Content/SimpleText.txt"));

return File(fileBytes, "text/plain");

}

****

**Content Result in Asp.Net MVC**

The Content Result in ASP.NET MVC returns different content formats to the view like HTML format, JavaScript format, or any other format. We need to use the ContentResult in the ASP.NET MVC Application when we want to allow the action to specify what should it returned. All we need to do is specify the content and MIME type.

public ContentResult Index()

{

return Content("<h3>Here's a custom content header</h3>", "text/html", System.Text.Encoding.UTF8);

}

****

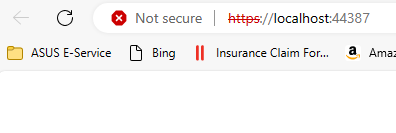
**Empty Result in Asp.Net MVC**

public EmptyResult Index()

{

return new EmptyResult();

}

****

**Redirect Result in Asp.Net MVC**

Suppose, you want to redirect to a specific URL, then you need to use the Redirect method and this method takes the URL to recirect.

public RedirectResult Index()

{

return Redirect("https://dotnettutorials.net");

}

**Redirect To Route Result in Asp.Net MVC**

The RedirectToRouteResult is used whenever we need to go from one action method to another action method within the same or different controller in ASP.NET MVC Application.

public RedirectToRouteResult Index()

{

return RedirectToRoute(new { controller = "Home", action = "About" });

}

**Redirect To Action Result in Asp.Net MVC**

The RedirectToAction Result in ASP.NET MVC is returning the result to a specified controller and action method. Controller name is optional in RedirectToAction method. If not mentioned, the Controller name redirects to a mentioned action method in the current Controller. Suppose the action name is not available but mentioned in the current controller, then it will show a 404 error page.

public ActionResult Index()

{

return RedirectToAction("About", "Home");

}

**Status Results in ASP.NET MVC**

HttpStatusCodeResult in ASP.NET MVC Framework returns an HTTP status code to the browser, along with a custom message to be displayed.

public HttpStatusCodeResult UnauthorizedStatusCode()

{

return new HttpStatusCodeResult(HttpStatusCode.Unauthorized, "You are not authorized to access this controller action.");

}

public HttpStatusCodeResult BadGateway()

{

return new HttpStatusCodeResult(HttpStatusCode.BadGateway, "I have no idea what this error means.");

}

##### ****What are Action Selectors in ASP.NET MVC?****

The Actions are the public methods of a controller in ASP.NET MVC Application that responds to incoming HTTP Requests. The Action Selectors in ASP.NET MVC are the attributes that can be applied to the action methods of a controller and are used to control which action method gets invoked in response to a particular request. That means Action Selectors in ASP.NET MVC Framework help the routing engine to select the correct action method to handle a particular request.

##### ****Understanding ActionName Action Selector in ASP.NET MVC****

The **ActionName** action selector in the ASP.NET MVC Application is used when we want to invoke an action method with a different name, than what is already given to the action method. To understand the need and use of the **ActionName** selector, let’s modify the **HomeController** as shown below.

public class HomeController : Controller

{

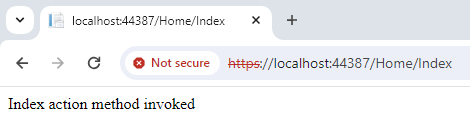
public string Index()

{

return "Index action method invoked";

}

**}**

****

The URL “**/Home/Index**“ would invoke the Index() action method in HomeController. If you want to invoke the**Index()** action method, with the URL “**/Home/List”,** then you need to decorate the action method with the ActionName attribute as shown below.

public class HomeController : Controller

{

[ActionName("List")]

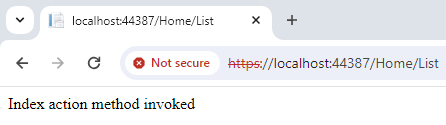
public string Index()

{

return "Index action method invoked";

}

}

****

Now, if you navigate to **/Home/Index**, you will get an error – “**The resource cannot be found**“. At the moment, the **Index**() action method is returning a string, but if it returns a view, should the view be named – **Index or List**? The List should be the view name.

public class HomeController : Controller

{

[ActionName("List")]

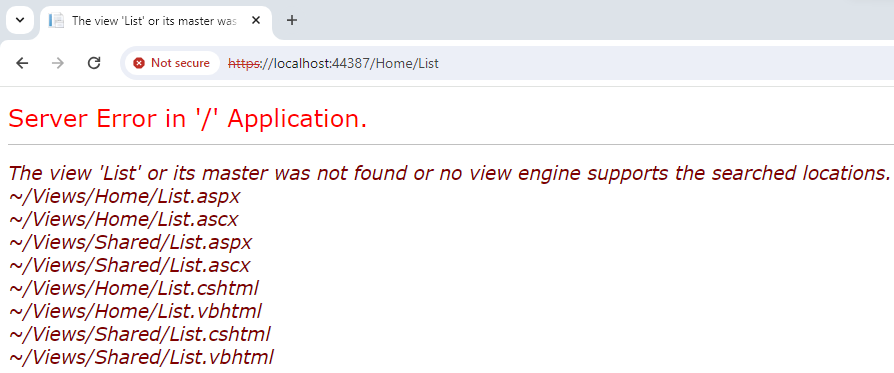
public ActionResult Index()

{

return View();

}

}

****

##### ****Action Verb Selector in ASP.NET MVC Application****

We need to use the **Action Verb Selector**when we want to control the invocation of an action method based on the request type in the ASP.NET MVC Application. We can define two different action methods with the same name but one action method responds to an HTTP Get request while the other action method responds to an HTTP Post request.

public class HomeController : Controller

{

EmployeeDBContext dbContext = new EmployeeDBContext();

public ActionResult Index()

{

List<Employee> ListEmployees = dbContext.Employees.ToList();

return View(ListEmployees);

}

[AcceptVerbs(HttpVerbs.Get)]

public ActionResult Edit(int id)

{

Employee employee = dbContext.Employees.Where(x => x.EmployeeId == id).FirstOrDefault();

return View(employee);

}

[AcceptVerbs(HttpVerbs.Post)]

public ActionResult Edit(Employee employee)

{

if (ModelState.IsValid)

{

dbContext.Entry(employee).State = EntityState.Modified;

dbContext.SaveChanges();

return RedirectToAction("Index");

}

return View(employee);

}

}

##### ****HttpGet and HttpPost Attribute in ASP.NET MVC****

Instead of using **[AcceptVerbs(HttpVerbs.Get)]** and **[AcceptVerbs(HttpVerbs.Post)]** attribute, we can also use HttpGet and HttpPost attributes as shown in the below code. This is an alternative approach to using the AcceptVerbs attribute and the behavior is going to be the same i.e. the action method which decorates with the HttpGet attribute will only respond to GET Request whereas the action method which decorates with HttpPost attribute will only respond to the Post Request.

public class HomeController : Controller

{

EmployeeDBContext dbContext = new EmployeeDBContext();

public ActionResult Index()

{

List<Employee> ListEmployees = dbContext.Employees.ToList();

return View(ListEmployees);

}

[HttpGet]

public ActionResult Edit(int id)

{

Employee employee = dbContext.Employees.Where(x => x.EmployeeId == id).FirstOrDefault();

return View(employee);

}

[HttpPost]

public ActionResult Edit(Employee employee)

{

if (ModelState.IsValid)

{

dbContext.Entry(employee).State = EntityState.Modified;

dbContext.SaveChanges();

return RedirectToAction("Index");

}

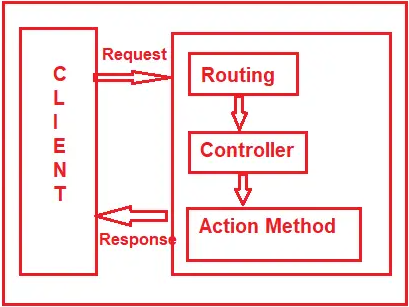
return View(employee);

}

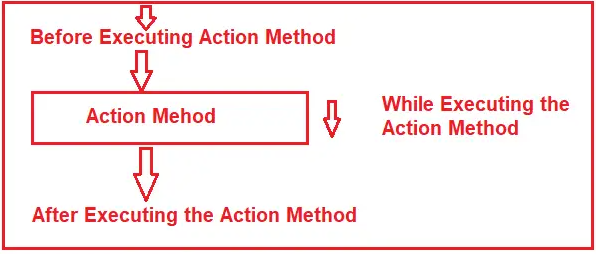
}

##### ****What are Filters in ASP.NET MVC Applications?****

As of now, we discussed when a client makes a request, then that request comes to the Routing Engine, and then the Routing Engine navigates that Request to the Controller. The controller then selects the appropriate action method to execute. So, it is the Controller action method that is going to handle the incoming request and send the response back to the client who initially made the request, as shown in the below image.



But what will you do if you want to execute some code or logic either before or after the action method executed as shown in the below image?



If that is your requirement, then you need to use Filters in the ASP.NET MVC application. The Filters in ASP.NET MVC Framework are the attribute that allows us to inject some logic or code that will be executed before or after an action method is invoked.

**Why do we need to use Filters in the ASP.NET MVC Applications?**

Basically, ASP.NET MVC Filters perform the following common functionalities in your application.

1. Caching
2. Logging
3. Error Handling
4. Authentication and Authorization, etc.

**What are the Predefined Filters?**

Some of the filters are already built by the ASP.NET MVC framework, and they are ready to be used. For example,

1. **Authorize**
2. **ValidateInput**
3. **HandleError**
4. **RequireHttps**
5. **OutputCache, etc**

**What are the Different Types of Filters available in the ASP.NET MVC Framework?**

The ASP.NET MVC 5 framework provides five different types of Filters. They are as follows

1. **Authentication Filter (Introduced in MVC 5)**
2. **Authorization Filter**
3. **Action Filter**
4. **Result Filter**
5. **Exception Filter**

**Note:**This is also the order of the execution of Filters if more than one filter is applied. But the point that you need to remember is the Exception Filter can be executed at any point in time when there is an unhandled exception occurring in your application.

##### ****Can we Create Custom Filters in MVC?****

Yes, we can create custom filters in MVC. If the built-in filters do not serve our purpose, then we can create our own custom filter as per our business requirements. We can create the Custom Filter for all five categories (Authentication Filter, Authorization Filter, Action Filter, Result Filter, and Exception Filter) of Filters.

**Where can we configure filters in ASP.NET MVC?**

We can configure the filters at three different levels of our application. They are as follows

1. Global Level (Applicable to all controllers and all action methods)
2. Controller Level (Applicable to all the action methods of the particular controller)
3. Action Level (Applicable to the specific action methods)

##### ****Configuring Filters at the Global Level in ASP.NET MVC****

Here you need to register the Filter within the **Application\_Start()** method of **Global.asax.cs** file as shown below. As we know, this is the first method of our application, which will be executed when the application starts. When you register a filter at the Global level, it applies to all the Action Methods of all the Controllers of your MVC application

protected void Application\_Start()

{

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

}

##### ****Configuring Filters at Controller Level in ASP.NET MVC****

Here, you need to apply the filter at the top of the controller name, as shown below. When you apply the filter at the Controller level, it applies to all the action methods of that controller only.

[Authorize(Roles = "Admin")]

public class AdminController : Controller

{

//Code

}

##### ****Configuring Filters at Action Level in ASP.NET MVC****

Here, you need to apply the filter on the top of the action method name, as shown below. When you apply the filter to a particular action method, then it is only applicable to that particular action method.

public class UserController : Controller

{

[Authorize(Users = "User1,User2")]

public ActionResult LinkToLogin(string provider)

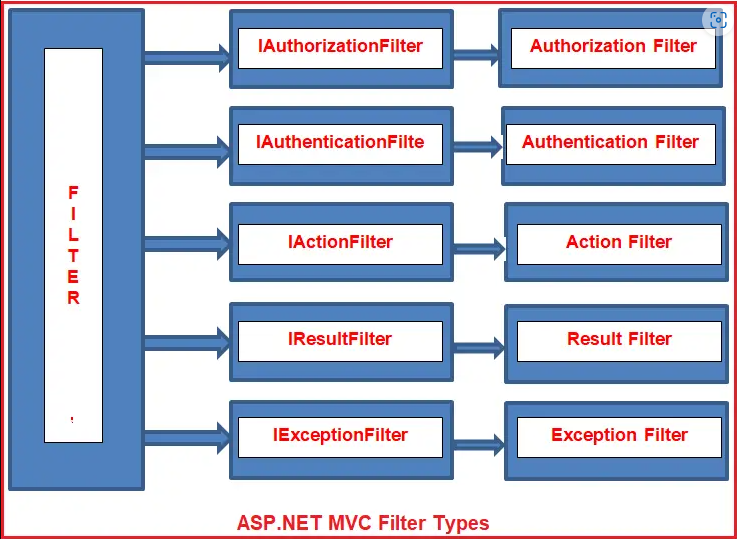
{

// Code

return View();

}

}



##### ****Authentication Filter in ASP.NET MVC****

The Authentication filter is the first one to execute before executing any other filter or action method. This filter checks whether the user from where the request is coming is a valid user or not. The Authentication filters in ASP.NET MVC Framework implement the **IAuthenticationFilter** interface. This filter is introduced with ASP.NET MVC5. The **IAuthenticationFilter** interface is used to create a Custom Authentication filter. The definition of the **IAuthenticationFilter** interface is given below.

As of now, there is no in-built Authentication Filter provided ASP.NET MVC Framework. If you want to create a Custom Authentication Filter, then you need to implement the **IAuthenticationFilter** interface.

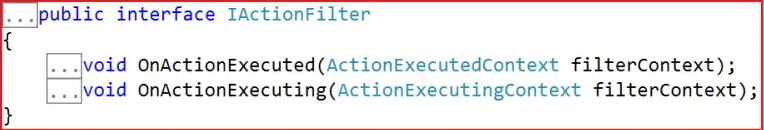
##### ****Authorization Filters in ASP.NET MVC****

The Authorization Filters are executed after the Authentication Filter. This filter checks whether the user has the right to access a particular resource or page. The built-in **AuthorizeAttribute** and **RequireHttpsAttribute** are examples of Authorization Filters. The Authorization Filters in ASP.NET MVC Framework implements the **IAuthorizationFilter** interface.

If you want to create a [**Custom Authorization Filter**](https://dotnettutorials.net/lesson/customizing-authorization-filter-mvc/), then you need to implement the **IAuthorizationFilter** interface.

##### ****Action Filters in ASP.NET MVC****

The Action Filters in ASP.NET MVC Application will be executed before the action method starts executing or after the action has been executed. So, if you want to execute some custom logic that will be executed before or after an action method is executed, then you need to use the Action Filters in MVC applications. The definition of the **IActionFilter** interface is given below.



The Action filters implement the **IActionFilter** interface with two methods, **OnActionExecuting** and **OnActionExecuted**. If you want to execute the Custom Logic before the action method starts executing, then you need to implement the OnActionExecuting method, and if you want to write custom logic after the action method is executed, then you need to implement the OnActionExecuted method.

**Result Filters in ASP.NET MVC**

The Result filters in the ASP.NET MVC application are executed before or after generating the result for an action. Action Result type can be ViewResult, PartialViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult, and EmptyResult, which derives from the ActionResult abstract class. Result filters are called after the Action filters. The in-built **[OutputCacheAttribute](https://dotnettutorials.net/lesson/outputcache-attribute-mvc/)**is an example of a Result Filter. The Result Filters in MVC implement the **IResultFilter** interface.

The Result filters implement the **IResultFilter** interface with two methods: **OnResultExecuting** and **OnResultExecuted**. If you want to execute the Custom Logic before generating the result, then you need to implement the **OnResultExecuting** method, and if you want to write custom logic after generating the result, then you need to implement the **OnResultExecuted**method. If you want to create a Custom Result Filter, then you need to implement the **IResultFilter** interface.

**Exception Filters in ASP.NET MVC**

The Exception filters are executed when there is an unhandled exception occurs during either the execution of actions or filters. The ASP.NET MVC Framework provides one in-built attribute called **HandleError** which is basically used to handle the unhandled exception in the MVC application.

The **IExceptionFilter** interface is used to create a [**Custom Exception Filter**](https://dotnettutorials.net/lesson/custom-exception-filter-mvc/), which provides the **OnException** method, which will be executed when there is an unhandled exception occurs during the actions or filters execution.

##### ****How to use Handle Error Attribute in ASP.NET MVC Application****

We can use the Handle Error attribute in three simple steps:

###### **Step1: Creating Error.cshtml view**

###### Create Shared Folder within the Views folder if it does not exist already. Then create one view with the name **Error.cshtml** within the shared folder. Once you create the **Error.cshtml** view then copy and paste the following code in it.

<!DOCTYPE html>

<html>

<head>

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

<title>Error</title>

</head>

<body>

<hgroup>

<h1>Error.</h1>

<h2>An error occurred while processing your request.</h2>

</hgroup>

</body>

</html>

###### **Step2: Enable Custom Errors in the web.config file**

**<customErrors mode=”On”></customErrors>**

###### **Step3: Apply Handle Error Attribute in ASP.NET MVC**

###### You can apply the HandleError attribute at all three different locations i.e. At the Action Method Level, at the Controller level, and Global Level.

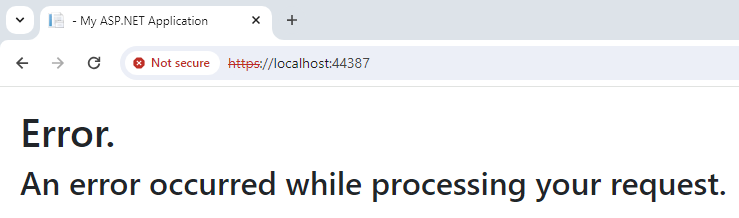
[HandleError]

public ActionResult Index()

{

throw new Exception("Something went wrong");

}

****

##### ****How to display Different Error Page for Different Exceptions?****

Create **NullReference.cshtml** view within the Shared folder.

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>Null Reference</title>

</head>

<body>

<hgroup>

<h1>Erro Occured </h1>

<h2>Null reference Exception occurred</h2>

</hgroup>

</body>

</html>

Create **DivideByZero.cshtml** view within the Shared Folder. Then copy and paste the following code in it.

@{

Layout = null;

}

<!DOCTYPE html>

<html>

<head>

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

<title>DivideByZero</title>

</head>

<body>

<hgroup>

<h1>Erro Occured </h1>

<h2>Divide by zero Exception occurred</h2>

</hgroup>

</body>

</html>

##### While applying the HandleError attribute, we have also checked the Exception type, and based on the Exception Type we have specified the view name. So, in that case, if the Exception type is ****DivideByZeroException**** then the **DivideByZero** view is going to be rendered. In the same line if the Exception type is ****NullReferenceException**** then the **NullReference** view is going to be rendered. Except for these two any other exception occurred then the **Error** view is going to be displayed.

##### ****Modifying the HomeController****

[HandleError(ExceptionType = typeof(DivideByZeroException), View = "DivideByZero")]

[HandleError(ExceptionType = typeof(NullReferenceException), View = "NullReference")]

[HandleError]

public class HomeController : Controller

{

public ActionResult Index()

{

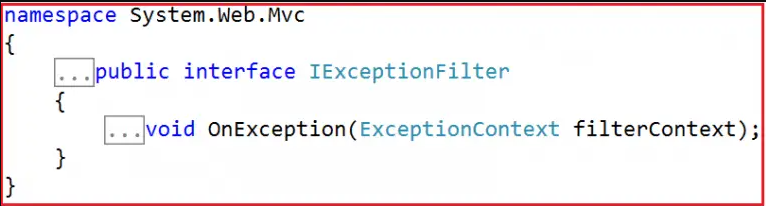
throw new Exception("Something went wrong");

}

##### }

##### ****Creating Custom Exception Filter in MVC****

Here we need to create a class file by implementing the **IExceptionFilter** interface. So let’s have a look at the definition of the **IExceptionFilter** interface as shown below.



As you can see this interface belongs to **System.Web.Mvc** namespace and having only a single method declaration. The Custom Exception class that we are going to create is going to implement the above **OnException** method only. So, create a class file with the name **LogCustomExceptionFilter.cs** within the **Models** folder and then copy and paste the following code in it.

using System;

using System.IO;

using System.Web;

using System.Web.Mvc;

namespace ExceptionFilterInMVC.Models

{

public class LogCustomExceptionFilter : FilterAttribute, IExceptionFilter

{

public void OnException(ExceptionContext filterContext)

{

if (!filterContext.ExceptionHandled)

{

var exceptionMessage = filterContext.Exception.Message;

var stackTrace = filterContext.Exception.StackTrace;

var controllerName = filterContext.RouteData.Values["controller"].ToString();

var actionName = filterContext.RouteData.Values["action"].ToString();

string Message = "Date :" + DateTime.Now.ToString() + ", Controller: " + controllerName + ", Action:" + actionName +

"Error Message : " + exceptionMessage

+ Environment.NewLine + "Stack Trace : " + stackTrace;

//saving the data in a text file called Log.txt

//You can also save this in a dabase

File.AppendAllText(HttpContext.Current.Server.MapPath("~/Log/Log.txt"), Message);

filterContext.ExceptionHandled = true;

filterContext.Result = new ViewResult()

{

ViewName = "Error"

};

}

}

}

}

##### ****How to Register Custom Exception Filter in ASP.NET. MVC****

As a filter, you can use it at three different levels i.e. at the action level, at the Controller level, and globally. Let’s register this filter globally so that it will be applicable for all the action methods of all controllers of our application. To register it globally, open the **FilterConfig** class which is present in the **App\_Start** folder, and then modify the class as shown below.

using System.Web.Mvc;

using ExceptionFilterInMVC.Models;

namespace ExceptionFilterInMVC.App\_Start

{

public class FilterConfig

{

public static void RegisterGlobalFilters(GlobalFilterCollection filters)

{

filters.Add(new LogCustomExceptionFilter());

}

}

}

##### ****Modifying the Global.asax****

As we know, the **Application\_Start()** method of **Global.asax** file is the first method that is going to be executed when the application starts. So, from here we need to call the **RegisterGlobalFilters()** method of **FilterConfig** class so that all our filters that are registered globally will be instantiated at the time of application startup. So, modify the **Application\_Start()** method of the **Global.asax** file as shown below.

using System.Web.Mvc;

using System.Web.Routing;

using ExceptionFilterInMVC.App\_Start;

namespace ExceptionFilterInMVC

{

public class MvcApplication : System.Web.HttpApplication

{

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

RouteConfig.RegisterRoutes(RouteTable.Routes);

//calling RegisterGlobalFilters to register filters globally

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

}

}

}