**Filters in ASP.NET MVC**

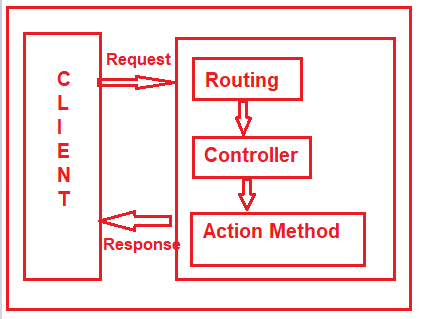
**Filters in ASP.NET MVC Application**

In this article, I am going to give you an overview of **Filters in the ASP.NET MVC** Application. Filters are one of the most important concepts in ASP.NET MVC Application and as a developer, you should be aware of this concept. So, from this and in few upcoming articles I am going to discuss ASP.NET MVC Filters in Detail. As part of this article, we are going to discuss the following pointers.

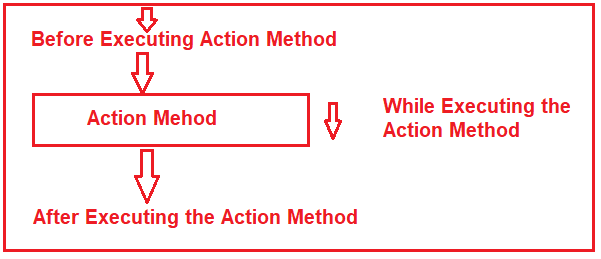
1. **What are the Filters in ASP.NET MVC?**
2. **Why do we need Filters in MVC?**
3. **What are the different types of MVC Filters?**
4. **Different ways to configure ASP.NET MVC Filters**
5. **What is the need for Custom Filters in the MVC application?**

**What are Filters in ASP.NET MVC Application?**

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 which is going to be executed either 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 are used to perform the following common functionalities in your application.

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

**What are the Different Types of Filters available in 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 occurred in your application.

**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**

**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 the five categories (Authentication Filter, Authorization Filter, Action Filter, Result Filter, and Exception Filter) of Filters.

**Where we can 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 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 is going to be executed when the application starts. When you register a filter at the Global level, then it is applicable 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, then it is applicable 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

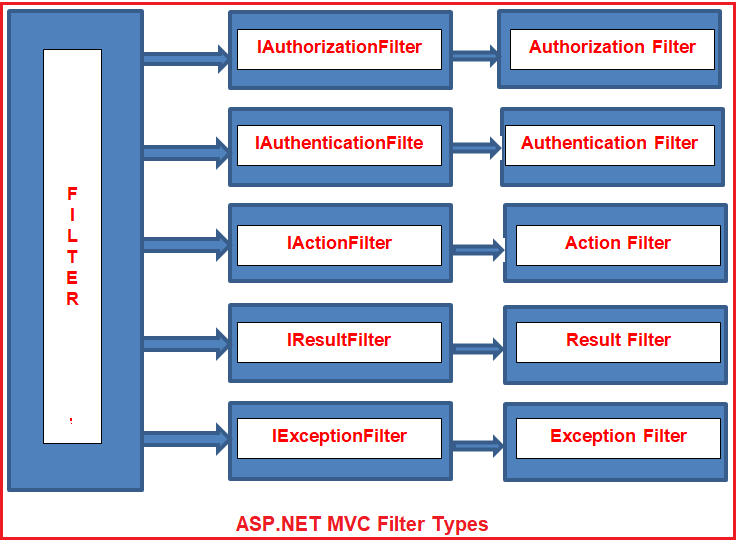
**return** View**()**;

**}**

**}**

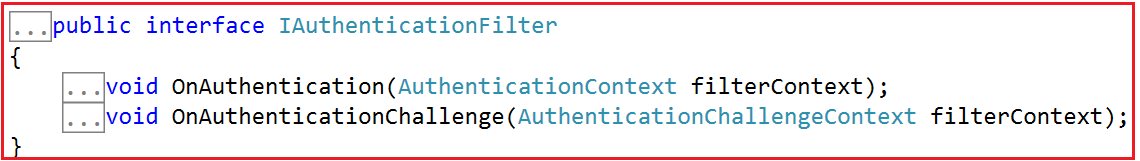
**Role and Responsibility of MVC Filters:**

As we already discussed we have five types of Filters (Authentication Filter, Authorization Filter, Action Filter, Result Filter, and Exception Filter) in the ASP.NET MVC application. Let us discuss the overview of each filter. Here we only discuss the purpose and when that filter is going to be executed and from our next article onwards we will discuss each filter in detail.



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

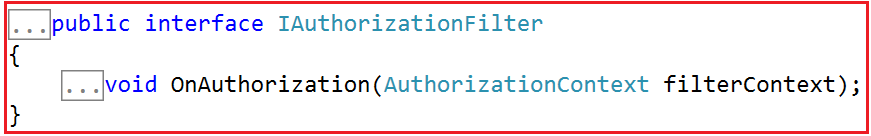
The Authentication filter is the first filter that is going to be executed before executing any other filter or action method. This filter checks that the user from where the request is coming is a valid user or not. The Authentication filters in ASP.NET MVC Framework implements the **IAuthenticationFilter** 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 Custom Authentication Filter then you need to implement the **IAuthenticationFilter** interface. In a later article, we will discuss how to Create [**Custom Authentication Filters**](https://dotnettutorials.net/lesson/customizing-authentication-filter-mvc/)with real-time examples.

**Authorization Filters in ASP.NET MVC**

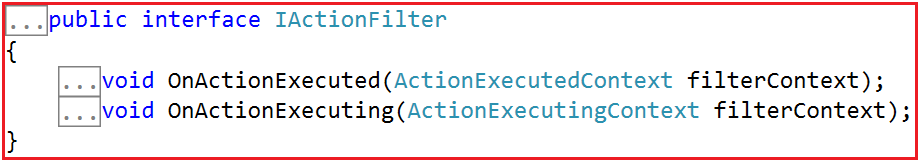
The Authorization Filters are executed after the Authentication Filter. This filter is used to check whether the user has the right to access a particular resource or page. The built-in **AuthorizeAttribute** and **RequireHttpsAttribute** are examples of Authorization Filters. The Authorization Filters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



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. We will discuss [**Authorization Filters**](https://dotnettutorials.net/lesson/authorization-filter-mvc/)with real-time examples in our upcoming articles.

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

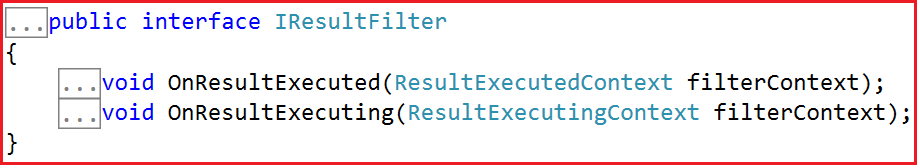
The Action Filters in ASP.NET MVC Application will be executed before the action method starts executing or after the action has been executed. So, if you want to execute some custom logic that is going to be executed before or after an action method is executed, then you need to use the Action Filters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



The Action filters implement the **IActionFilter** interface that has 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. We will discuss [**Action Filters in MVC**](https://dotnettutorials.net/lesson/custom-action-filters-mvc/) with real-time examples in our upcoming articles.

**Result Filters in ASP.NET MVC:**

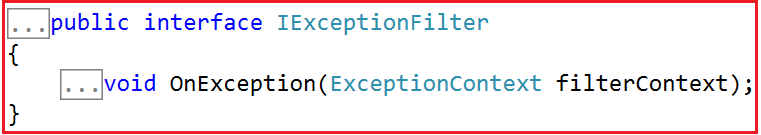
The Result filters in the ASP.NET MVC application are executed before or after generating the result for an action. Action Result type can be ViewResult, PartialViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult and EmptyResult which derives from the ActionResult 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 Result Filters. The Result Filters in MVC implements the **IResultFilter** interface. The definition of the IResultFilter interface is given below.



The Result filters implement the **IResultFilter** interface that has 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. We will discuss [**Result Filters**](https://dotnettutorials.net/lesson/customizing-outputcache-attribute-mvc/)in detail with real-time examples in our upcoming articles.

**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 in-built [**HandleErrorAttribute**](https://dotnettutorials.net/lesson/exception-filter-mvc/)is an example of Exception Filters. 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. The definition of **IExceptionFilter** is given below.



# Exception Filter in ASP.NET MVC

# 

## ****Exception Filter in ASP.NET MVC Application****

In this article, I am going to discuss **Exception Filter in the ASP.NET MVC** **Application** with examples. Please read our previous article where we discussed the basics of [**Filters in ASP.NET MVC**](https://dotnettutorials.net/lesson/filters-in-mvc/). At the end of this article, you will understand what exactly an Exception Filter is and when and how to use Exception Filters in ASP.NET MVC Application.

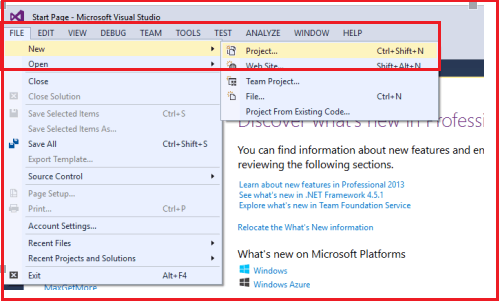
##### ****What is Exception Filter in ASP.NET MVC Application?****

The Exception Filter in the ASP.NET MVC Application is used to handle any exceptions that occur during the ASP.NET MVC Request processing pipeline. 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.

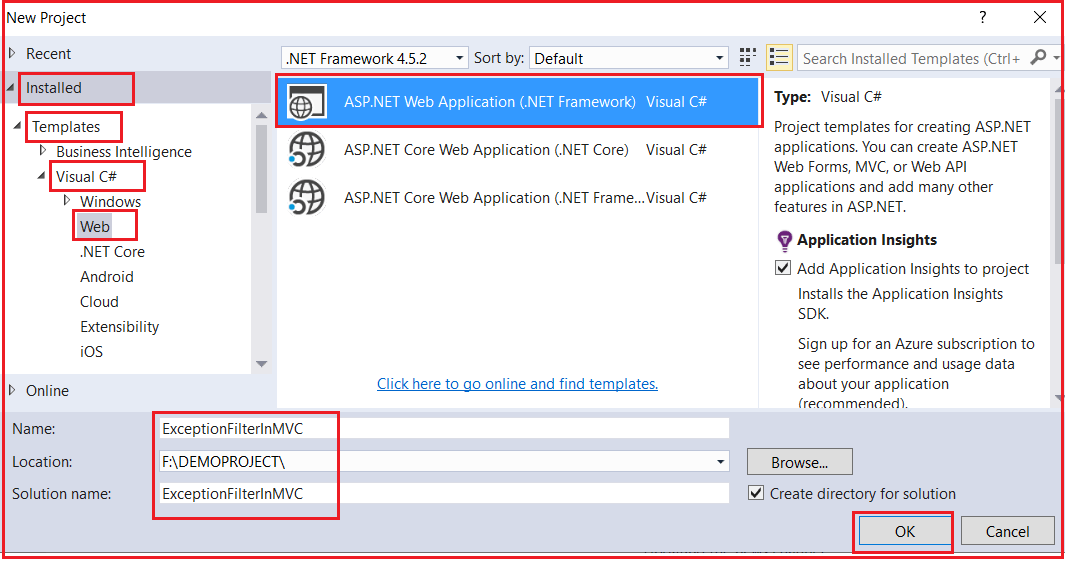
As part of this article, we will see how to use the **HandleError**attribute to display friendly error pages to the end-user when there is an unhandled exception occurs during the request processing pipeline. Let us understand this with an example.

##### ****Create a new Empty MVC application:****

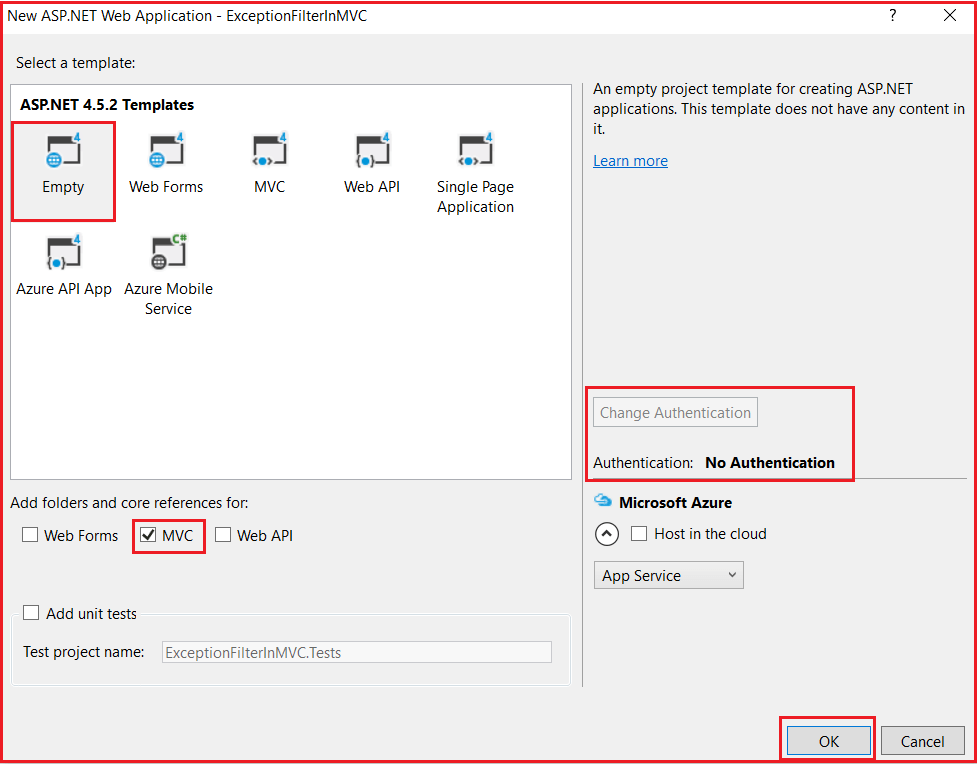
Open visual studio and then select **File => New => Project**option from the context menu as shown in the below image.



Once you click on the “**Project**” link, a new window will open. From that window, we need to select the “**Web**” templates from the left pane. From the middle pane, select the “**ASP.NET Web Application**“. Then provide a meaningful name to the project such as “**ExceptionFilterInMVC**”. Finally, click on the “**OK**” button as shown in the below image



Once you click on the “**OK”** button a new dialog window will open with the name “**New ASP.NET Project**” for selecting project Templates as shown in the below image.



From the above window, we need to choose the “**Empty”** and “**MVC”** project template with the Authentication type as “**No Authentication”** and then click on the “**OK**” button. Once you click on the **OK** button it will take some to time create the project for us.

##### ****Creating Controller:****

Create an Empty MVC5 controller with the name HomeController within the Controllers Folder. Once you create the Controller copy and paste the following code into it.

**using** *System;*

**using** *System.Web.Mvc;*

**namespace** *ExceptionFilterInMVC.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**throw** new Exception**(**"Something went wrong"**)**;

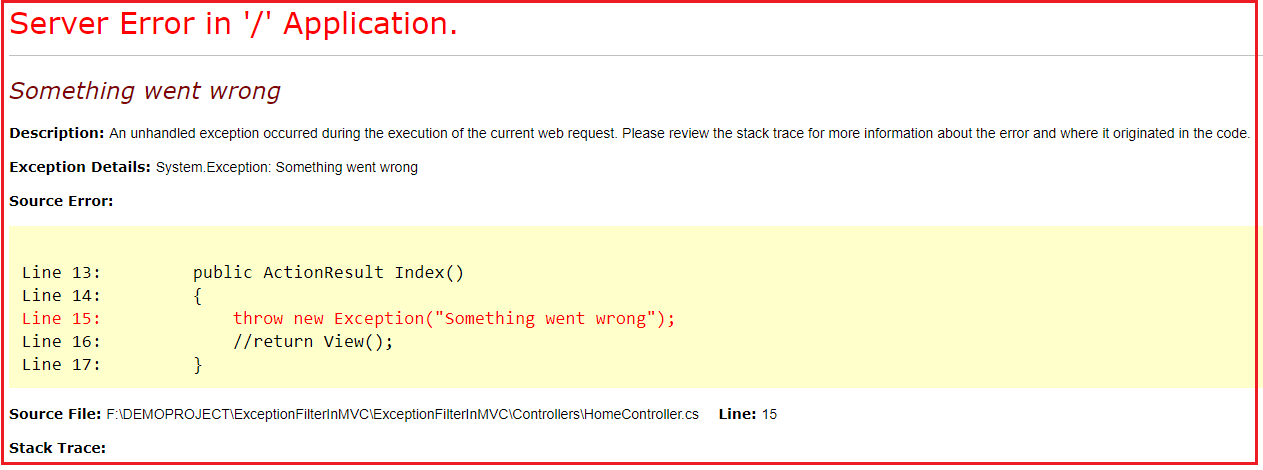
//return View();

**}**

**}**

**}**

As you can see, from the **Index**() action method, we intentionally throw an exception. As we have not handled this exception, so when we run the application, we will get the default **“yellow screen of death”** error page as shown in the below image.



The above error page does not make any sense for the end-user to understand. Now, let us see how to replace the above yellow screen of death error page with a friendly error page that can be understood by the end-user.

##### ****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.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Unknwo Error**</h1>**

**<h2>**An unknown error has occurred. We are working on it. Please try after some time**</h2>**

**</hgroup>**

**</body>**

**</html>**

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

To enable Custom Errors for your application, open the web.config file that is present in the root directory and then adds the following “**customErrors**” element under the “**<system.web>**” section.

**<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.

##### ****Applying at Controller Level:****

Please modify the Home Controller as shown below to apply the Handle Error attribute to handle the unhandled exception that occurred during the execution of this Controller actions.

**using** *System;*

**using** *System.Web.Mvc;*

**namespace** *ExceptionFilterInMVC.Controllers*

**{**

**[**HandleError**]**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**throw** new Exception**(**"Something went wrong"**)**;

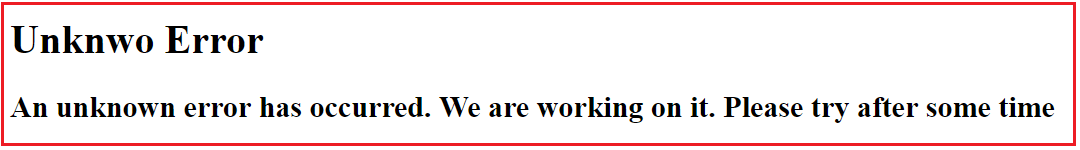
//return View();

**}**

**}**

**}**

Now run the application and you will see that instead of the yellow screen of death error page, you will now get the generic error page as shown below.



##### ****Using HandleError Globally in ASP.NET MVC:****

If you want to use the **HandleError** attribute globally, then you need to register it within the **GlobalFilters**. You can do this within the **Application\_Start()** method of the **Global.asax file** as shown below.

**using** *System.Web.Mvc;*

**using** *System.Web.Routing;*

**namespace** *ExceptionFilterInMVC*

**{**

**public** **class** MvcApplication : System.Web.HttpApplication

**{**

**protected** **void** Application\_Start**()**

**{**

AreaRegistration.RegisterAllAreas**()**;

RouteConfig.RegisterRoutes**(**RouteTable.Routes**)**;

//Adding Handle Error attribute Globally

GlobalFilters.Filters.Add**(**new HandleErrorAttribute**())**;

**}**

**}**

**}**

Now remove the HandleError Attribute from the Controller. With the above changes in place, now it is going to handle all the exceptions raised by all the action methods of all the controllers and return the error view which is present inside the shared folder.

##### ****How to register Filters Using FilterConfig in ASP.NET MVC?****

You can also register the Filters globally by using the FilterConfig class. Let see how we can do this. Add a class file with the name **FilterConfig.cs** within the **App\_Start** folder of your application. Once you create the class file then copy and paste the following code into it.

**using** *System.Web.Mvc;*

**namespace** *ExceptionFilterInMVC.App\_Start*

**{**

**public** **class** FilterConfig

**{**

**public** **static** **void** RegisterGlobalFilters**(**GlobalFilterCollection filters**)**

**{**

filters.Add**(**new HandleErrorAttribute**())**;

**}**

**}**

**}**

##### ****Modifying the Global.asax file:****

Modify the **Application\_Start** event of the **Global.asax** file as shown below. Here we are just calling the RegisterGlobalFilters method of FilterConfig class.

**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**)**;

**}**

**}**

**}**

With the above changes, now run the application and it should display the error page as expected.

##### ****How to**** ****display error detail in the Error view?****

If you want to display the error details in the error view, then you need to make the error view a strongly typed view of the Model **System.Web.Mvc.HandleErrorInfo**. Then as usual by using the **@Model** keyword, you can access the necessary properties to display errors. So, modify the **Error.cshtml** view which is present in the Shared folder as shown below.

@{

Layout = null;

}

@model System.Web.Mvc.HandleErrorInfo

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Erro Occured **</h1>**

**<h2>**Controller Name: @Model.ControllerName**</h2>**

**<h2>**Action Name: @Model.ActionName**</h2>**

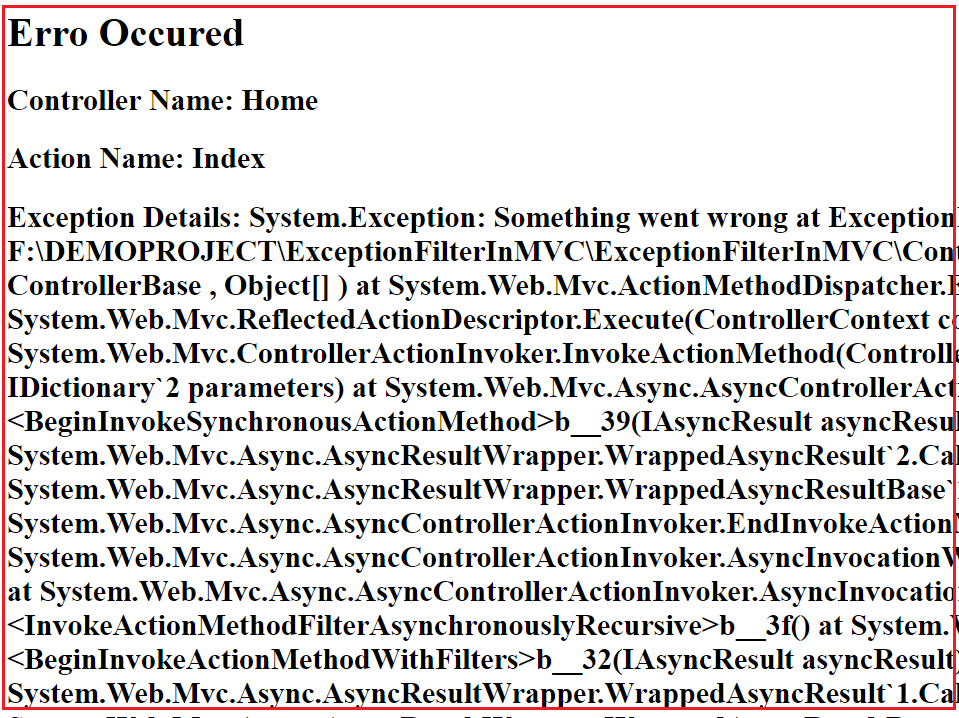
**<h2>**Exception Details: @Model.Exception**</h2>**

**</hgroup>**

**</body>**

**</html>**

Now run the application and navigate to **Home/Index** it will display the following error page.



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

Let understand this with an example. First, add two views in the shared folder. Create **NullReference.cshtml** view within the Shared folder. Then copy and paste the following code into it.

@{

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>**

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

Please modify the Home Controller as shown below. As you can see we have applied the HandleError attribute at the Controller level. 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.

**[**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"**)**;

**}**

**public** ActionResult TestMethod1**()**

**{**

**throw** new NullReferenceException**()**;

**}**

**public** ActionResult TestMethod2**()**

**{**

**throw** new DivideByZeroException**()**;

**}**

**}**

##### ****What are the Limitations of HandleErrorAttribute in MVC?****

Following are the limitations of the built-in HandleError attribute in MVC.

1. We cannot log the error anywhere using the built-in HandleError attribute.
2. It is not possible to handle the exceptions raised outside the controllers. For example- we cannot handle the exception because of the invalid URL.
3. Scenario-based Exception Handling is not possible. For example – display one error page when the request comes via AJAX and show a different error page when it comes via other than AJAX.

To overcome all the above-mentioned problems we need to create a Custom Handle Error Attribute. So, in our next article, I am going to discuss how to create [**Custom Exception Filter in ASP.NET MVC**](https://dotnettutorials.net/lesson/custom-exception-filter-mvc/) Application. Here, in this article, I try to explain the **Exception Filter in ASP.NET MVC** Application step by step with a simple example. I hope you understood how to use Handle Error Attribute in the ASP.NET MVC application.

# Custom Exception Filter in ASP.NET MVC

## ****Custom Exception Filter in ASP.NET MVC Application****

In this article, I am going to discuss **Custom Exception Filter in ASP.NET MVC** Application. Please read our previous article before proceeding to this article where we discussed the [**Exception Filter**](https://dotnettutorials.net/lesson/exception-filter-mvc/) i.e. the built-in exception filter attribute **HandleError**. At the end of this article, you will understand what is the need for Custom Exception Filter and how to create and use Custom Exception Filter in ASP.NET MVC Application. We are going to work with the same example that we created in our previous article.

##### ****What is the need for Custom Exception Filter in ASP.NET MVC Application?****

As we already discussed in our previous article, we cannot do the following things using the built-in HandleErrorr attribute

1. We cannot log the Exception anywhere using the HandleError attribute.
2. It is not possible to handle the exceptions raised outside the controllers.
3. Scenario-based Exception Handling is not possible.

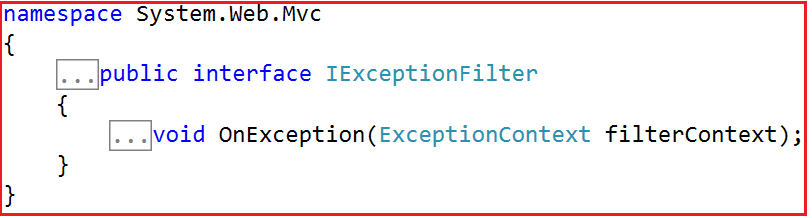
Let us see how to overcome the above problems by creating a Custom Exception Filter in MVC. Here in this article, I will show you how to log the exception in a text file but if you want then you can log the exception in the database too. Here our focus is on how to create the custom exception filter not how to log the exception.

##### ****Adding Log Folder:****

Add a folder with the name Log into the project’s root directory. In this folder, we are going to log all the exceptions raised in our application using a text (i.e. Log.txt) file.

##### ****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"

**}**;

**}**

**}**

**}**

**}**

As you can see, the above class is derived from the **FilterAttribute**class. This is required as we want to use this class as a Filter attribute. This class is also implementing the **IExceptionFilter** interface and provides an implementation for **OnException()**method. This method contains the logic for storing the exception information in a text (i.e. .txt) file. First, it checks whether the exception has been handled or not by using the ExceptionHandled property. If the exception is not handled then it fetches the exception message (filterContext.Exception.Message), stack trace (filterContext.Exception.StackTrace), controller name (filterContext.RouteData.Values[“controller”].ToString()), and action method name (filterContext.RouteData.Values[“action”].ToString()) from the filterContext object. Then it stores the error info into the text file. Then it changes the ExceptionHandled property to true and redirects to the error page.

##### ****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. If the FilterConfig class is not already present there then just create a class file with the name **FilterConfig.cs** within the **App\_Start** folder.

###### **FilterConfig.cs**

**using** *System.Web.Mvc;*

**using** *ExceptionFilterInMVC.Models;*

**namespace** *ExceptionFilterInMVC.App\_Start*

**{**

**public** **class** FilterConfig

**{**

**public** **static** **void** RegisterGlobalFilters**(**GlobalFilterCollection filters**)**

**{**

filters.Add**(**new LogCustomExceptionFilter**())**;

**}**

**}**

**}**

**Note:** While creating the project, if you have selected the Empty MVC Template then the above class (i.e. FilterConfig) will not present with you, but if you have selected any readymade MVC template then the above class will be available with your application. If this class file is not present then just create a class file with the name FilterConfig within the App\_Start folder. This FilterConfig class is the centralized place where we used to add the Filters globally for our application. This will be instantiated when the Web application starts.

##### ****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**)**;

**}**

**}**

**}**

##### ****Modifying the Error.cshtml view:****

Open the **Error.cshtml** file which is present inside the Shared folder which is inside the views folder. If **Error.cshtml** file is not present then just create it. Once you open the **Error.cshtml** file then copy and paste the following code in it.

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Unknwo Error**</h1>**

**<h2>**An unknown error has occurred. We are working on it. Please try after some time**</h2>**

**</hgroup>**

**</body>**

**</html>**

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

Please modify the Home Controller as shown below to throw different types of exceptions from different action methods.

**using** *System;*

**using** *System.Web.Mvc;*

**using** *ExceptionFilterInMVC.Models;*

**namespace** *ExceptionFilterInMVC.Controllers*

**{**

//[LogCustomExceptionFilter]

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**throw** new Exception**(**"Something went wrong"**)**;

**}**

**public** ActionResult About**()**

**{**

**throw** new NullReferenceException**()**;

**}**

**public** ActionResult Contact**()**

**{**

**throw** new DivideByZeroException**()**;

**}**

**}**

**}**

That’s it. We are done with our implementation. Now, run the application and try to execute all the action methods, and then check the **Log.txt** file which should be created inside the **Log** folder.

**Error Pages Based on Status Code in ASP.NET MVC**

**Error Pages Based on Status Code in ASP.NET MVC**

In this article, I am going to discuss **how to display different Error Pages Based on Status Code** in the ASP.NET MVC application. Please read our previous two articles as this is a continuation article to our previous articles. In our previous two articles, we discussed [**Exception Filters in MVC**](https://dotnettutorials.net/lesson/exception-filter-mvc/)and how to create and use [**Custom Exception Filter**](https://dotnettutorials.net/lesson/custom-exception-filter-mvc/) in ASP.NET MVC applications.

**Creating Error Controller:**

Create a controller with the name ErrorController within the Controllers folders. Once you create the Error Controller then copy and paste the following code in it.

**using** *System.Web.Mvc;*

**namespace** *ExceptionFilterInMVC.Controllers*

**{**

**public** **class** ErrorController : Controller

**{**

**public** ActionResult PageNotFoundError**()**

**{**

**return** View**()**;

**}**

**public** ActionResult UnauthorizedError**()**

**{**

**return** View**()**;

**}**

**public** ActionResult InternalServerError**()**

**{**

**return** View**()**;

**}**

**public** ActionResult GenericError**()**

**{**

**return** View**()**;

**}**

**}**

**}**

Note: If you have any specific status code other than the above, then you can also create that specific status code action method.

**Creating Error Views:**

Now, we need to create the respective error views.

**PageNotFoundError.cshtml**

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**PageNotFound Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Page Not Found Error**</h1>**

**<h2>**The Page you are trying to access is no longer available. Kindly check and submit the URL again**</h2>**

**</hgroup>**

**</body>**

**</html>**

**UnauthorizedError.cshtml**

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Unauthorized Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Unauthorized Error**</h1>**

**<h2>**You donot have the permission to access this page. Kindly contact with your admin.**</h2>**

**</hgroup>**

**</body>**

**</html>**

**InternalServerError.cshtml**

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Internal Server Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Internal Server Error**</h1>**

**<h2>**Some Internal Server error Occurred while processing your request. Kindly try after some time.**</h2>**

**</hgroup>**

**</body>**

**</html>**

**GenericError.cshtml**

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

**<title>**Generic Error**</title>**

**</head>**

**<body>**

**<hgroup>**

**<h1>**Generic Error**</h1>**

**<h2>**An unknown error has occurred. We are working on it. Please try after some time**</h2>**

**</hgroup>**

**</body>**

**</html>**

**Registering the Error Views based on Status Code:**

Our requirement is to register the respective error pages based on their status code. If the error status code is not available then we need to display the Generic Error page to the end-user. To do this, open the web.config file and modify the following CustomError elements within the System.Web as shown below.

**<customErrors** mode="On" defaultRedirect="~/Error/GenericError"**>**

**<error** statusCode="404" redirect="~/Error/PageNotFoundError"**/>**

**<error** statusCode="500" redirect="~/Error/InternalServerError"**/>**

**<error** statusCode="401" redirect="~/Error/UnauthorizeddError"**/>**

**</customErrors>**

That’s it. Now run the application and navigates to a URL that does not exist and you will see that 404 custom error page will display. I hope now you understood how to display different error pages based on the Status code in ASP.NET MVC Application. In the next article, I am going to discuss the [**Child Action Only Attribute in ASP.NET MVC**](https://dotnettutorials.net/lesson/childactiononly-attribute-mvc/) Application with one real-time example.

**ChildActionOnly Attribute in ASP.NET MVC**

**ChildActionOnly Attribute in ASP.NET MVC Application**

In this article, I am going to discuss the **ChildActionOnly Attribute in ASP.NET MVC** Application. Please read our previous article before proceeding to this article where we discussed [**how to display custom error pages based on the status code**](https://dotnettutorials.net/lesson/error-pages-based-on-status-code-in-mvc/) in ASP.NET MVC Application.

**What is ChildActionOnly Attribute in ASP.NET MVC?**

Suppose you have a scenario where you have one action method and you don’t want that action method to be invoked via URL rather you want that action method to be invoked by other actions of your application. Then in such scenarios, **ChildActionOnly Attribute**can be handy. So, when we decorate an action method with the ChildActionOnly attribute, then it is called child action in ASP.NET MVC Application and child action methods are only accessible by a child request. That means once an action method becomes a child action, then it will not respond to the URL requests rather it will be invoked by other action methods of your application. Let us understand this with an example.

**Step1:** Create an empty ASP.NET MVC 5 application

**Step2:** Add HomeController. Copy and paste the following code. As you can see, here, we decorate the Countries action method with the ChildActionOnly attribute. Now, this method is accessible only by the child request. A runtime exception will be thrown if we try to access this method through a URL request.

**public** **class** HomeController : Controller

**{**

// Public action method that can be invoked using a URL request

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

// This method is accessible only by a child request. A runtime

// exception will be thrown if a URL request is made to this method

**[**ChildActionOnly**]**

**public** ActionResult Countries**(**List**<**String**>** countryData**)**

**{**

**return** View**(**countryData**)**;

**}**

**}**

**Step3:** Right click on the **“Countries()”** action method and add the **“Countries”** view. This view will render the given list of countries as an unordered list.

@model List**<string>**

@foreach (string country in Model)

{

**<ul>**

**<li>**

**<b>**

@country

**</b>**

**</li>**

**</ul>**

}

**Step4:** Right click on the **“Index()”** action method and add the **“Index”** view. Copy and paste the following code in it. Notice that, here we are using the Action HTML helper method to invoke the childaction.

**<h2>**Countries List**</h2>**

@Html.Action("Countries", new { countryData = new List**<string>**() { "US", "UK", "India" } })

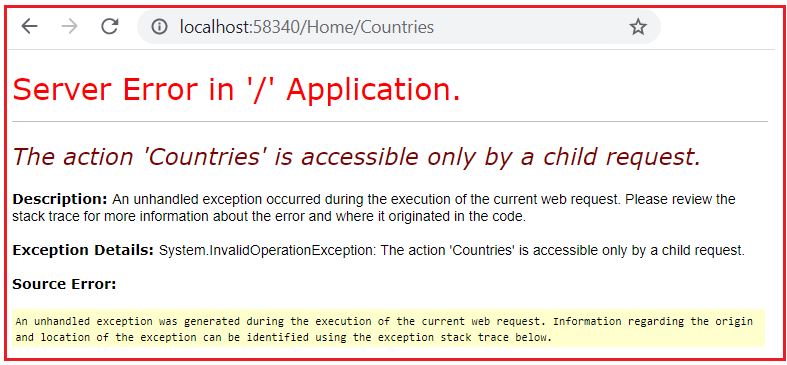
**Please Note:** The Child actions can also be invoked using the **“RenderAction()**” HTML helper as shown below.

@**{**

Html.RenderAction**(**"Countries", new **{** countryData = new List**<**string**>()** **{** "US", "UK", "India" **}** **})**;

**}**

Now run the application and navigate to the **Home/Index** action method and internally Index view make a call to the Countries child action which will return the data and those data you can see on the Index view as expected. Now, let’s try to access the Countries action method from the URL by making a request to Home/Countries and see what happens. We will get the following error page which clearly saying that the action Countries is accessible only by a child request.



**Points to remember while working with ChildActionOnly Attribute in ASP.NET MVC**

1. The child action methods will not be responded to incoming URL requests. If you try to invoke the child actions using URL, then you will get a runtime error saying – **Child action is accessible only by a child request**.
2. You can only access the child action methods by making a child request from a view either by using the “**Action()**” or “**RenderAction()**” HTML helper methods.
3. The most important point that you need to remember is, an action method doesn’t need to have the **ChildActionOnly** attribute to be used as a child action. You can access the normal action methods by making a child request. You need to use the **ChildActionOnly** attribute only if you want to prevent the action method to be invoked using URL.
4. Child action methods are different from the NonAction methods in MVC application. The difference is that the NonAction methods cannot be invoked as a child request either by using the Action() or RenderAction() HTML helpers.
5. The main advantage of Child Action method is that you can cache portions of a view. We will discuss this in [**OutputCache Attribute**](https://dotnettutorials.net/lesson/outputcache-attribute-mvc/) article.

# OutputCache Attribute in MVC

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

In this article, I am going to discuss the **OutputCache Attribute in the ASP.NET MVC** application with examples. Please read our previous article before proceeding to this article where we discussed how to use the [**Child Action Only Attribute in the ASP.NET MVC**](https://dotnettutorials.net/lesson/childactiononly-attribute-mvc/) application. The **OutputCache attribute** belongs to the Results Filter category.

##### ****Why do we need OutputCache Attribute in ASP.NET MVC?****

In order to implement Caching in ASP.NET MVC Application, we need the OutputCache Attribute. The OutputCache Attribute in ASP.NET MVC Application is used to cache the content returned by a controller action method for a specific time period, so that, if the subsequent request comes within that time period, then the content is going to be returned from the cache memory. So, with the help of caching, we can drastically improve the performance of an ASP.NET MVC Application. This filter can also be applied to the Action Method or on the controller.

The OutputCache Attribute has several properties like CacheProfile, Duration, Location, VaryByParam, VaryByHeader, NoStore, etc. We will discuss the use of each property.

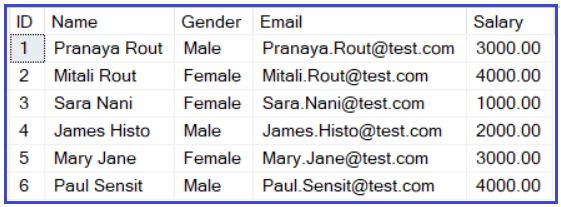
##### ****Why Caching?****

As we already discussed to improve the performance of an application we need caching. For example, if we have an ASP.NET MVC application, which displays the list of states or a list of cities or a list of countries, etc, like some master that which does not change that frequently. Now if we want to retrieve all the above master data from a database, then we need to execute the database query each and every time the user invokes the controller action method and then return the view.

In such scenarios, we can take advantage of the **OutputCache Attribute in ASP.NET MVC** to avoid executing the action method each and every time whenever the user wants to retrieve the view. Here, the view is going to be returned from the cache memory instead of being regenerated by the controller action method.

##### ****Example: OutputCache Attribute in ASP.NET MVC****

Let us understand OutputCache Attribute in ASP.NET MVC Application with an example. We are going to use the following Employee table in this example.



Please use the below SQL script to create and populate the Employee table with the required sample data.

-- Create a Employee Table

**Create** **table** Employee

(

**ID** int identity primary key,

Name nvarchar(50),

Gender nvarchar(10),

Email nvarchar(50),

Salary decimal(18,2)

)

-- Insert some test data for testing purpose

**Insert** **into** Employee values('Pranaya Rout', 'Male', 'Pranaya.Rout@test.com', 3000)

**Insert** **into** Employee values('Mitali Rout', 'Female', 'Mitali.Rout@test.com', 4000)

**Insert** **into** Employee values('Sara Nani', 'Female', 'Sara.Nani@test.com', 1000)

**Insert** **into** Employee values('James Histo', 'Male', 'James.Histo@test.com', 2000)

**Insert** **into** Employee values('Mary Jane', 'Female', 'Mary.Jane@test.com', 3000)

**Insert** **into** Employee values('Paul Sensit', 'Male', 'Paul.Sensit@test.com', 4000)

##### ****Create a new MVC application:****

Create a new ASP.NET MVC Project. Once you create the ASP.NET MVC 5 Application, then create a new folder by Right-Clicking on the Project and select Add => New Folder and Rename the folder name as DAL

##### Adding ADO.NET Data Model inside DAL Folder

Now add ADO.NET Entity Data Model and follow the Database First Approach and create the EDMX file for our Employee table. Once you create the EDMX file, then the Following is the auto-generated EMPLOYEE model generated by Entity Framework.

**namespace** *CachinginMVC.DAL*

**{**

**using** System;

**using** System.Collections.Generic;

**public** **partial** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Email **{** **get**; **set**; **}**

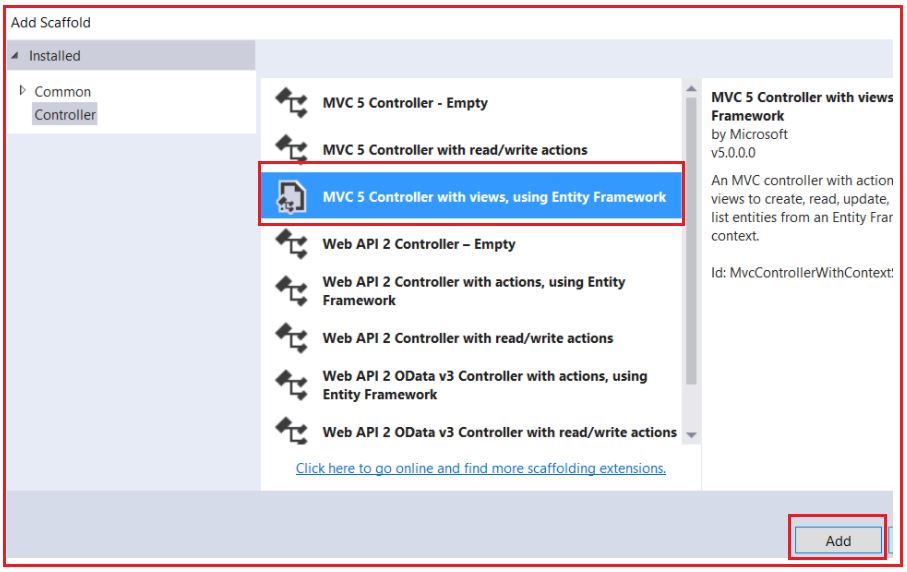
**public** Nullable**<decimal>** Salary **{** **get**; **set**; **}**

**}**

**}**

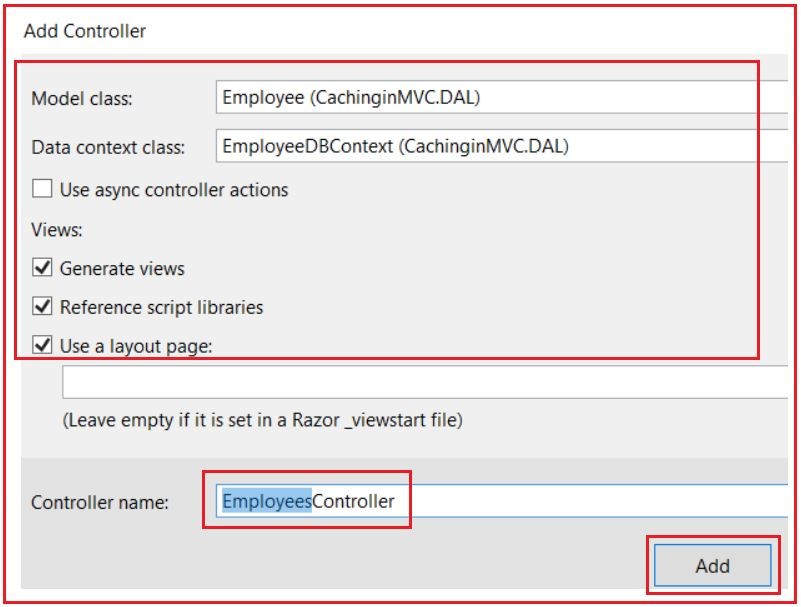
###### **Adding EmployeeController.**

Right-click on the Controllers folder and then select **Add => Controller.**Then select MVC5 Controller with views, using Entity Framework as shown in the below image.



Once we click on Add button one popup will open. Provide the following details

1. **Model Class: Employee (CachinginMVC.DAL)**
2. **Data Context Class: EmployeeDBContext (CachinginMVC.DAL)**
3. **Controller Name: EmployeesController**
4. Rest is as it is and click on the **Add** button as shown in the below image.



Modify the **Index()** action method in **EmployeesController** as shown below. Notice that, we are using the **OutPutCache** attribute to cache the content returned by the **Index()**action method for 10 seconds. The Duration property of the OutputCache attribute takes the value in seconds and then caches the result in the memory for that many seconds.

**[**OutputCache**(**Duration = 10**)]**

**public** ActionResult Index**()**

**{**

System.Threading.Thread.Sleep**(**3000**)**;

**return** View**(**db.Employees.ToList**())**;

**}**

##### ****Modify code in the “Index.cshtml” view as shown below.****

@model IEnumerable**<CachinginMVC.DAL.Employee>**

@{

ViewBag.Title = "Index";

}

**<p>**

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

**</p>**

**<p>**

**<b>**Employee List retrieved @@ @DateTime.Now.ToString()**</b>**

**</p>**

**<table** class="table"**>**

**<tr>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th>**

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

**</th>**

**<th></th>**

**</tr>**

@foreach (var item in Model)

{

**<tr>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

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

**</td>**

**<td>**

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

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

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

**</td>**

**</tr>**

}

**</table>**

##### ****Changes to RouteConfig.cs file****

**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 = "Employees", action = "Index", id = UrlParameter.Optional **}**

**)**;

**}**

**}**

When we navigate to **/Employees/Index**, the view output is cached for 10 seconds. If we refresh the view, within 10 seconds, then we will get the cached response of that view. After 10 seconds, the cache expires, the code is executed again and the output is cached for another 10 seconds.

##### ****Caching specific portion of a view using ChildActionOnly attribute:****

**Step1:** Remove OutputCache attribute and the line which calls Thread.Sleep(), from the Index() action method in EmployeeController. After the changes, the Index() action method should be as shown below.

**public** ActionResult Index**()**

**{**

**return** View**(**db.Employees.ToList**())**;

**}**

Add GetEmployeeCount() action method to EmployeeController. Notice that, this method is decorated with OutputCache and ChildActionOnly attributes. Child actions can be used to implement partial caching, although not necessary. In this case, even if the ChildActionOnly attribute is removed, a portion of the view will be cached as expected

**[**ChildActionOnly**]**

**[**OutputCache**(**Duration = 10**)]**

**public** string GetEmployeeCount**()**

**{**

**return** "Employee Count = " + db.Employees.Count**()**.ToString**()** + "@ " + DateTime.Now.ToString**()**;

**}**

Copy and paste the following code, just below the closing table tag in Index.cshtml view.

**<br /><br />**  
**<b> @Html.Action(“GetEmployeeCount”) </b>**

Navigate to **/Employee/Index**. Notice that, every time we refresh the page, the time in the section of the page that displays the employee list changes, but not the time, that displays the employee count. This proves that only a portion of the view is cached.

# VaryByParam Location and CacheProfiles in OutputCache Attribute in MVC

## ****VaryByParam Location and CacheProfiles in OutputCache Attribute in MVC****

In this article, I am going to discuss the **VaryByParam Location and CacheProfiles** **Properties** of **OutputCache Attribute in MVC**. Please read our previous article before proceeding to this article where we discussed the basics of [**OutputCache Attribute in MVC**](https://dotnettutorials.net/lesson/outputcache-attribute-mvc/) application with an example. We are going to work with the same example that we started in our previous article.

##### ****VarByParam Option of OutputCache Attribute in ASP.NET MVC:****

If you want to implement different cache versions of the output based on some input parameters then you need to use the VarByParam option of the OutputCache attribute in the ASP.NET MVC Application. For example, we have a detailed action method which will display the employee details based on the employee id. That means this method takes employee id as the input parameter and based on that input parameter it displays that employee information. Now we want to create a different cache version of the details view based on the employee id.

In situations like above, we can make use of the **VaryByParam** option of the **OutputCache** attribute. This **VaryByParam** property creates a different cached version of the same content when a form parameter or query string parameter varies. Following is the implementation of Details action.

**[**OutputCache**(**Duration = **int**.MaxValue, VaryByParam = "id"**)]**

**public** ActionResult Details**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** new HttpStatusCodeResult**(**HttpStatusCode.BadRequest**)**;

**}**

Employee employee = db.Employees.Find**(**id**)**;

**if** **(**employee == **null)**

**{**

**return** HttpNotFound**()**;

**}**

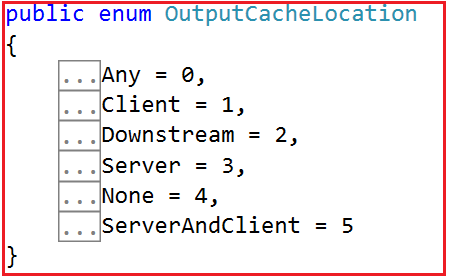
**return** View**(**employee**)**;

**}**

The Details() action method includes a **VaryByParam** property with the value “Id”. When different values of the Id parameter are passed to the controller action, different cached versions of the Details view are generated. So, in simple words, we can say that **VaryByParam**means based on the input parameter, it will store different versions of the view in the cache.

##### ****Location Property with OutputCache Attribute:****

The Location property of OutputCache Attribute is used to specify the location where the output is cached. The Location property takes the values from **OutputCacheLocation** enum. The definition of **OutputCacheLocation** enum is shown below.



As per your business requirement and security, you can cache the data in the appropriate location as follows:

1. **Any:** This is the default value for Location Property. If we have not specified the Location Property in OutputCache Attribute then the value is going to be Any. In this case, the output cache can be located on the browser client (from where the request is originated) on a proxy server (or any other server) participating in the request, or on the resource server where the request was processed.
2. **Client:** If you set the Location value to Client, then the output cache is going to be stored on the browser client from where the request is originated.
3. **Downstream:** When you set the Location value to Downstream, then the output cache is going to be stored in any HTTP 1.1 cache-capable devices other than the origin server. This includes proxy servers and the client that made the request.
4. **Sever:**When you set the Location value as Server, then the output cache is going to be stored on the Web server where the request was processed.
5. **None:** When you set the Location value to None, then the output cache is disabled for the requested page.
6. **ServerAndClient:** When you set the Location value as ServerAndClient, then the output cache can be stored either at the origin server or at the requesting client. Proxy servers are not allowed to cache the response.

##### ****Modify the Index action method as shown below to use the Location property:****

In the below example, we have set the location property value as client and hence, now the output cache is going to be stored on the client machine.

**[**OutputCache**(**Duration = 10, Location = System.Web.UI.OutputCacheLocation.Client**)]**

**public** ActionResult Index**()**

**{**

**return** View**(**db.Employees.ToList**())**;

**}**

##### ****CacheProfiles Properties of OutputCache Attribute in ASP.NET MVC:****

To cache the data returned by **Index**() action method, for 60 seconds, we would use **[OutputCache]** attribute as shown below.

**[**OutputCache**(**Duration = 60**)]**

**public** ActionResult Index**()**

**{**

**return** View**(**db.Employees.ToList**())**;

**}**

In the example above, we specified the **OutputCache**settings within the code i.e. at the action method level. The problem with this approach is that

1. If you have to apply the same cache settings to several methods then you need to write the same code at multiple places resulting code duplicated.
2. Later, if you want to change the cache settings, then you need to change it at several places. As a result, maintaining the application code becomes complicated. Also, changing the application code requires to build and re-deployment.

**To overcome the above disadvantages**, the cache settings can be specified in the **web.config** file using **cache profiles**.

###### **Step1: Specify the cache settings in web.config using cache profiles**

**<system.web>**

**<caching>**

**<outputCacheSettings>**

**<outputCacheProfiles>**

**<clear/>**

**<add** name="1MinuteCache" duration="60" varyByParam="none"**/>**

**</outputCacheProfiles>**

**</outputCacheSettings>**

**</caching>**

**</system.web>**

###### **Step2: Reference the cache profile in application code**

**[**OutputCache**(**CacheProfile = "1MinuteCache"**)]**

**public** ActionResult Index**()**

**{**

**return** View**(**db.Employees.ToList**())**;

**}**

The cache settings are now read from one central location i.e. from the **web.config** file and this is possible due to the cache profiles.

###### **Advantage of using Cache Profiles in ASP.NET MVC Application:**

1. You have only one place to change the cache settings if required. As a result, maintaining the caching is much easier.
2. As the changes are done in the web.config, we don’t require to build and redeploy the application.

# Custom OutputCache Attribute in ASP.NET MVC

## ****Custom OutputCache Attribute in ASP.NET MVC****

In this article, I am going to discuss **How To Create Custom OutputCache Attribute in ASP.NET MVC** application. Please read our previous article where we discussed how to use [**VarByParam, Location, and Output Cache Profile**](https://dotnettutorials.net/lesson/varybyparam-location-and-cacheprofiles/) options of Outputcache Attribute with examples. We are also going to work with the same example that we worked on within our previous two articles.

##### ****Using Cache Profiles with Child Action method:****

In the following example, the GetEmployeeCount() method is decorated with two attributes i.e. ChildActionOnly and OutputCache. As you can see the OutputCache attribute uses the CacheProfile. If you remember in our previous article we create the cache profile in the web.config file.

**[**ChildActionOnly**]**

**[**OutputCache**(**CacheProfile = "1MinuteCache"**)]**

**public** string GetEmployeeCount**()**

**{**

**return** "Employee Count = " + db.Employees.Count**()**.ToString**()**

+ "@ " + DateTime.Now.ToString**()**;

**}**

When you use the Cache Profiles with child action methods, then you will get an error saying –

**Duration must be a positive number.**There are many different ways to make the cache profiles work with child action methods. Here, in this article, I am going to create a Custom Outputcache Attribute to make the cache profiles work with the child action methods.

###### **Create a custom OutputCache attribute that loads the cache settings from the cache profile in web.config.**

**Step1:** Right-click on the project name in solution explorer, and add a folder with the name **Common**.   
**Step2:** Right-click on the **“Common”** folder and add a class file with the name **PartialCacheAttribute.cs**   
**Step3:** Copy and paste the following code. Notice that, I have named the custom **OutputCache**attribute as **PartialCacheAttribute**.

**using** *System.Web.Configuration;*

**using** *System.Web.Mvc;*

**namespace** *CachinginMVC.Common*

**{**

**public** **class** PartialCacheAttribute : OutputCacheAttribute

**{**

**public** PartialCacheAttribute**(**string cacheProfileName**)**

**{**

OutputCacheSettingsSection cacheSettings = **(**OutputCacheSettingsSection**)**WebConfigurationManager.GetSection**(**"system.web/caching/outputCacheSettings"**)**;

OutputCacheProfile cacheProfile = cacheSettings.OutputCacheProfiles**[**cacheProfileName**]**;

Duration = cacheProfile.Duration;

VaryByParam = cacheProfile.VaryByParam;

VaryByCustom = cacheProfile.VaryByCustom;

**}**

**}**

**}**

**Step4:** Use **PartialCacheAttribute** on the child action method and pass the cache profile name that is created in the web.config file. Please note that**PartialCacheAttribute** is in **CachinginMVC.Common** namespace.

**[**ChildActionOnly**]**

**[**PartialCache**(**"1MinuteCache"**)]**

**public** string GetEmployeeCount**()**

**{**

**return** "Employee Count = " + db.Employees.Count**()**.ToString**()**

+ "@ " + DateTime.Now.ToString**()**;

**}**

# ValidateInput Attribute in ASP.NET MVC

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

In this article, I am going to discuss the **ValidateInput Attribute in ASP.NET MVC** Application. Please read our previous article where we discussed [**How To Create Custom OutputCache Attribute in MVC**](https://dotnettutorials.net/lesson/customizing-outputcache-attribute-mvc/) Application.

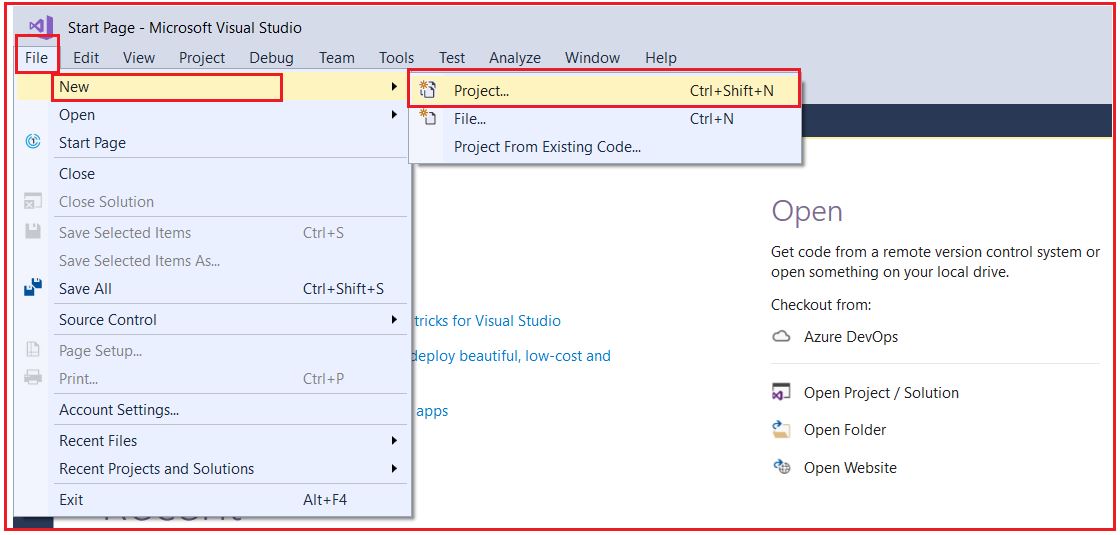
##### ****What is ValidateInput Attribute in ASP.NET MVC?****

The ValidateInput Attribute in MVC is used to allow sending HTML content or codes to the server which is by default disabled by ASP.NET MVC Framework to avoid XSS (Cross-Site Scripting) attacks. This attribute is used to enable or disable request validation. By default, request validation is enabled in ASP.NET MVC Framework.

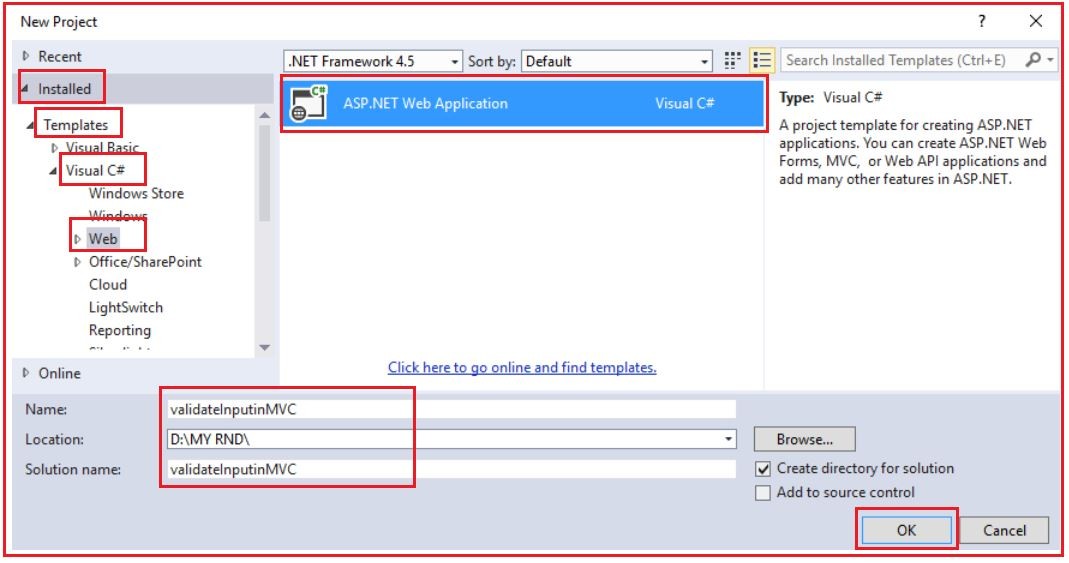
##### ****Example: ValidateInput Attribute in ASP.NET MVC****

Let’s understand ValidateInput Attribute in ASP.NET MVC Application with an example.

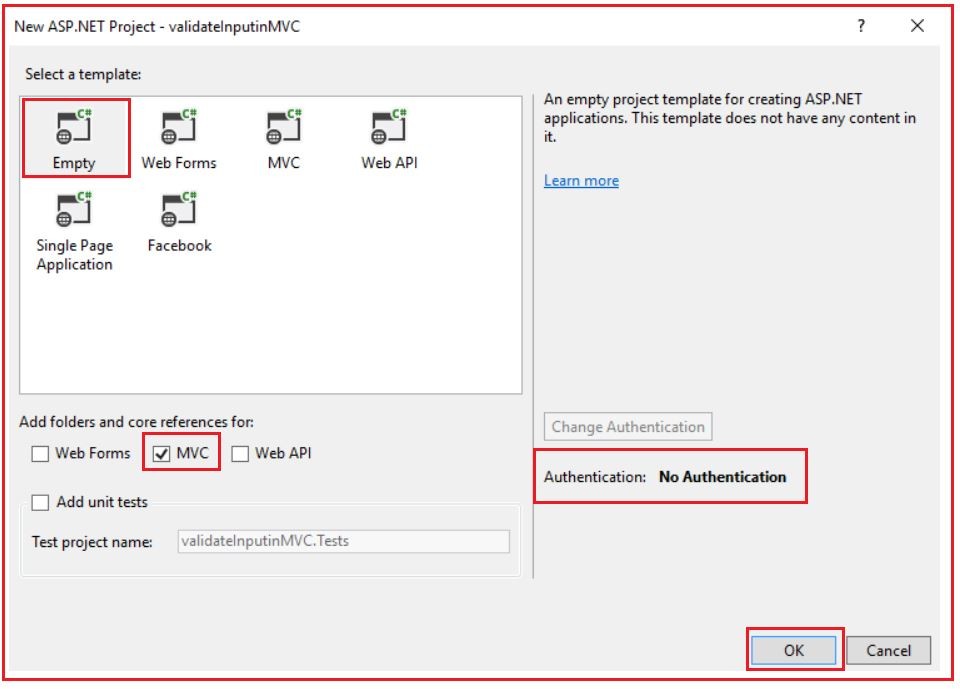
**Step1:** Create a new ASP.NET MVC 5 application using the Empty template. Open Visual Studio and create a New Project. Select File => New => Project option as shown in the below image.



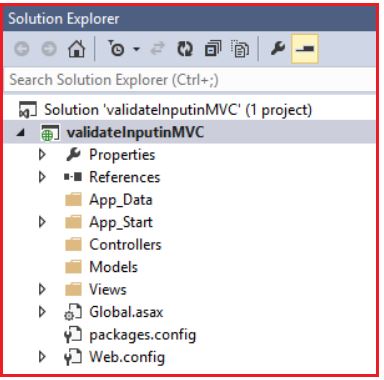
After clicking on the “**Project**” link a new dialog will pop up. In that we are going to select web templates from the left pane after selecting the web template, we find only one project template in its “**ASP.NET Web Application**” just select that. After selecting the project template next we are going to name the project “**validateInputinMVC**” and clicking on the **OK** button as shown in the below image.



Once you click on the **OK** button a new dialog will pop up with the Name “**New ASP.NET Project**” for selecting project Templates. In this dialog, we are going to choose the **Empty** project template and then we are choosing the **MVC** checkbox and click on the **OK** button as shown in the below image.

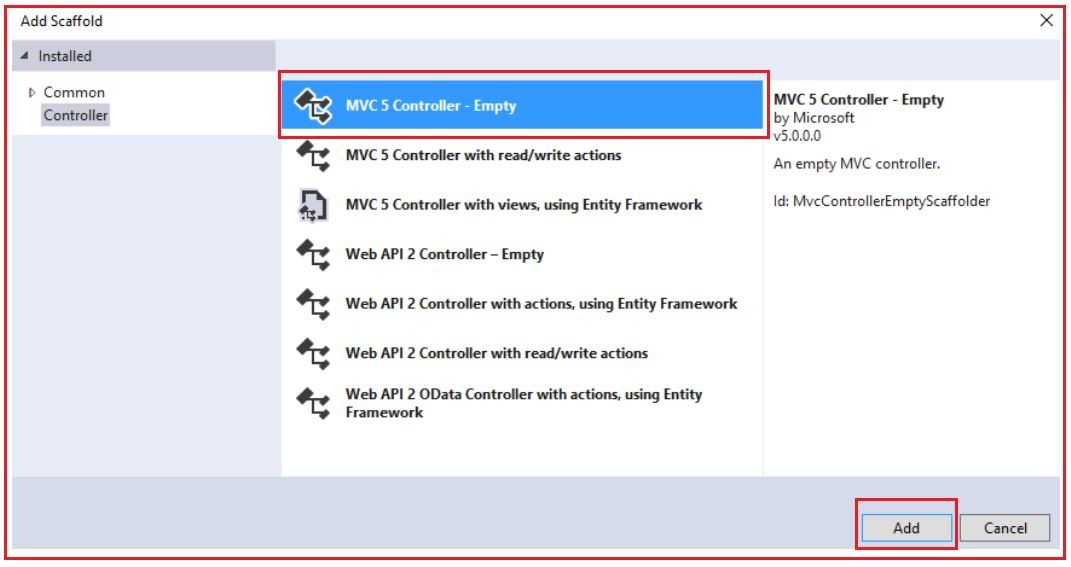


Once you click on the **OK** button. it will take some to time create the project for us with the following file and folder structure.

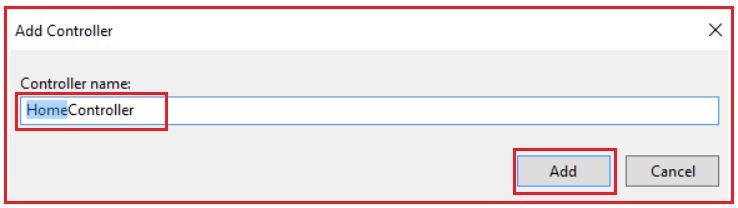


##### ****Step2: Add a HomeController.****

Right-click on the Controllers folder and select controller which will open a pop-up for adding the controller. Here, select “**MVC 5 Controller – Empty”** and click on the **Add** button as shown in the below image.



Once you click on the Add button, it will open a new pop-up for proving the controller name. Here, provide the controller name as Home and click on the Add button as shown in the below image.



Once you click on the Add button, it will add the HomeController. Then copy and paste the following code within the HomeController

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**public** string Index**(**string comments**)**

**{**

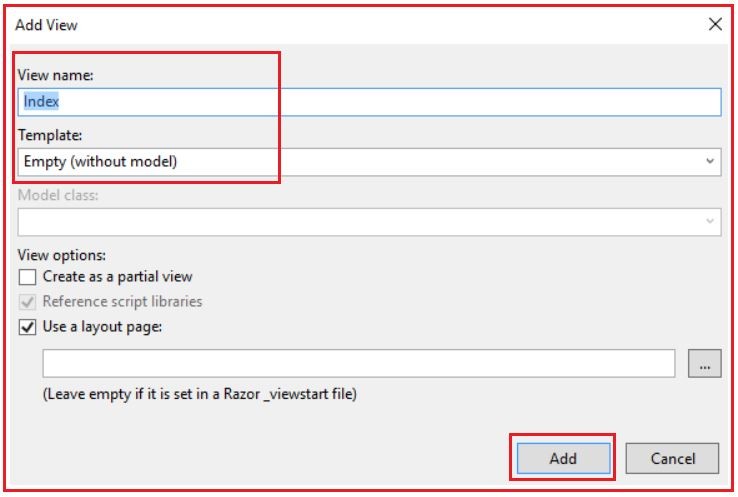
**return** "Your Comments: " + comments;

**}**

**}**

###### **Step3: Add Index.cshtml view.**

Right-click on the Index action method and select Add View which will open a pop-up as shown in the below image. Here, we are going with the default values and click on the Add button.



Once you click on the Add button, it will create the Index.cshtml view. Copy and paste the following code into it.

@{

ViewBag.Title = "Index";

Layout = null;

}

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

@using (Html.BeginForm())

{

**<b>**Comments:**</b>**

**<br/>**

@Html.TextArea("comments")

**<br/>**

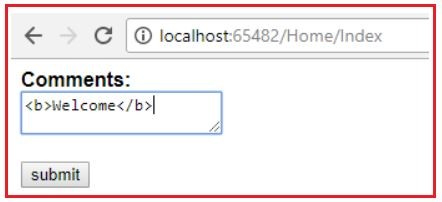
**<br/>**

**<input** type="submit" value="submit"**/>**

}

**</div>**

**Step4:**Run the application and navigate to**/Home/Index**. Type the text **<b>Welcome</b>** in the “**Comments**” textbox and click “**Submit**” as shown in the below image.



###### **Notice that, we get the following error –**



This is because, by default, request validation is turned on in ASP.NET MVC Application and does not allow you to submit any HTML, to prevent XSS (Cross-site scripting attacks). However, in some cases, we may want the user to be able to submit HTML tags like <b>,<u>, etc. For this to happen, we need to turn off request validation, by decorating the action method with the **ValidateInput** attribute and set the value as false as shown in the below code.

**namespace** *validateInputinMVC.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**[**ValidateInput**(false)]**

**public** string Index**(**string comments**)**

**{**

**return** "Your Comments: " + comments;

**}**

**}**

**}**

At this point, we should be able to submit comments, with HTML tags in them. In the next article, I am going to discuss the [**RequireHttps Attribute in ASP.NET MVC**](https://dotnettutorials.net/lesson/requirehttps-attribute-mvc/) Application. Here, in this article, I try to explain the **ValidateInput Attribute in ASP.NET MVC** application with Examples. I hope this ValidateInput Attribute in MVC article will help you with your need. I would like to have your feedback. Please post your feedback, question, or comments about this ValidateInput Attribute in MVC article.

# RequireHttps Attribute in ASP.NET MVC

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

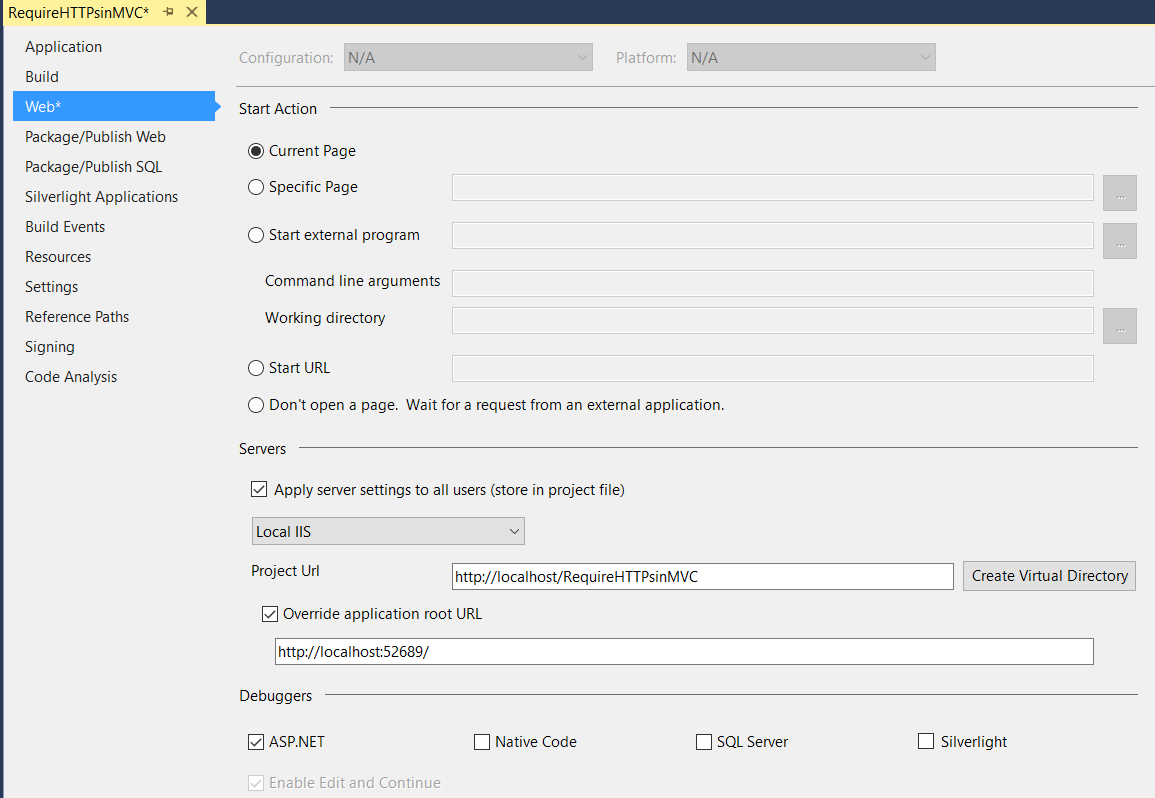
In this article, I am going to discuss the **RequireHttps Attribute in ASP.NET MVC** Application with Examples. Please read our previous article, where we discussed the [**ValidateInput attribute in ASP.NET MVC**](https://dotnettutorials.net/lesson/validateinput-attribute-mvc/).

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

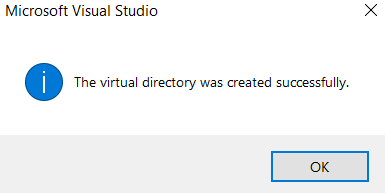
The RequireHttps Attribute in ASP.NET MVC forces an unsecured HTTP request to be re-sent over HTTPS. Let’s understand [RequireHttps] attribute with an example. To understand this let’s create an empty MVC application with the name RequireHTTPsinMVC.

##### ****Set the local IIS to run this project:****

Once you created the project, let’s set the local IIS to run this project.  To do so right-click on the project and click on properties, which will open the properties window for us. Then click on the **web** tab, Choose **Local IIS**, Check the **Override application root URL**checkbox and then click on **Create Virtual Directory** button as shown in the below image.



Once we click on the Create Virtual Directory button it will display a popup as shown below

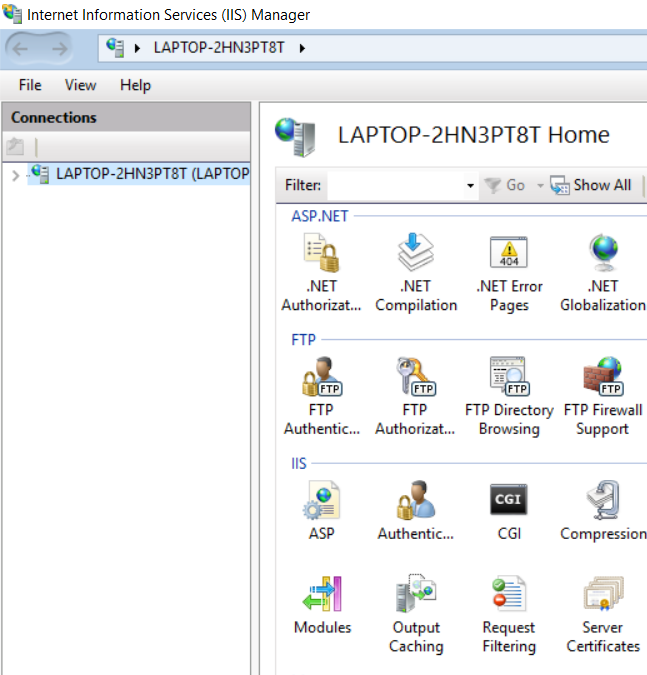


Just click OK, and it will create the application in IIS.

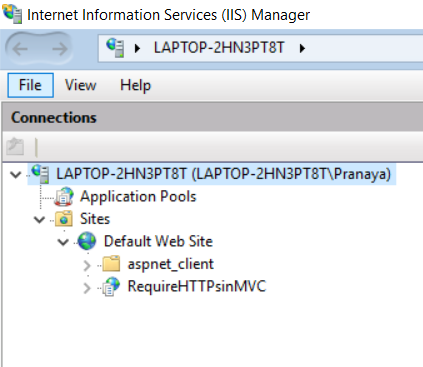
**Note:** You have to run the visual studio in Administrator mode to do so otherwise it will not all you to create the virtual directory in IIS.

##### ****How to see the website in IIS?****

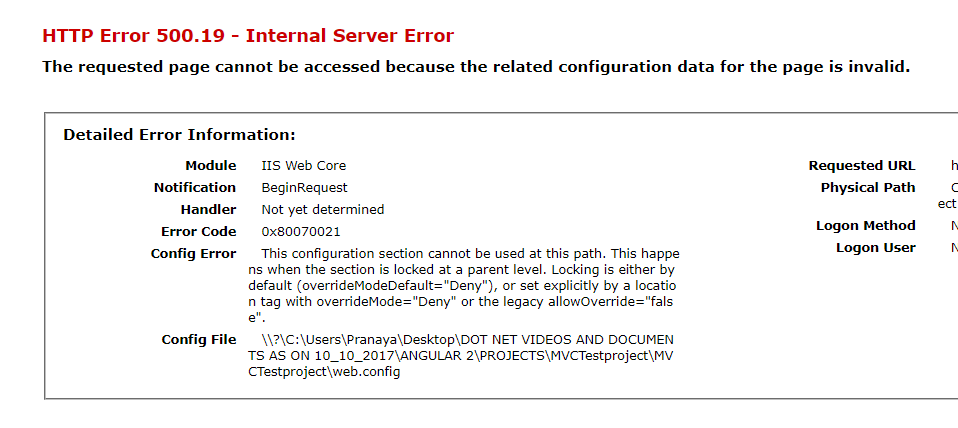
Open the Run window. Type inetmgr and click ok, it will open IIS as shown below



**To see our application just expand the node as shown below**



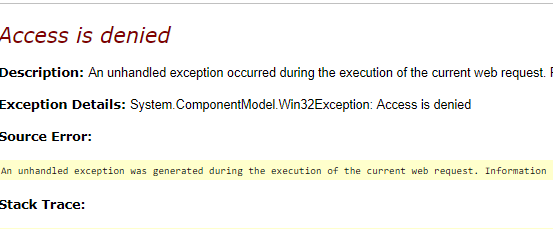
**Save the application and run the application from the visual studio.**If you are getting the below error, then you need to install a few windows configuration things.



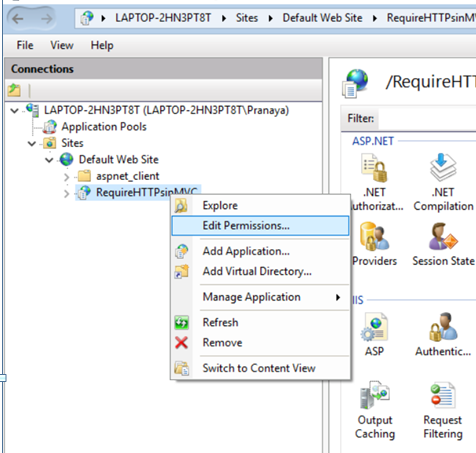
1. **Open Run Window**
2. **in the Run Window, enter “OptionalFeatures.exe”**
3. **in the features window, Click: “Internet Information Services”**
4. **Click: “World Wide Web Services”**
5. **Click: “Application Development Features”**
6. **Check the features.**

##### ****Now again run the application.****

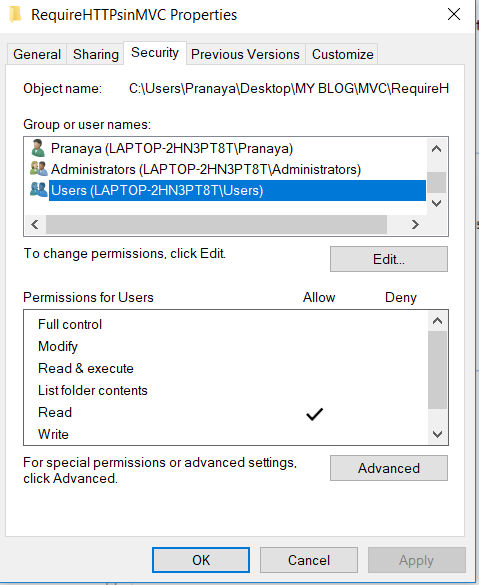
If you are getting below Access is Denied error then do the following things.



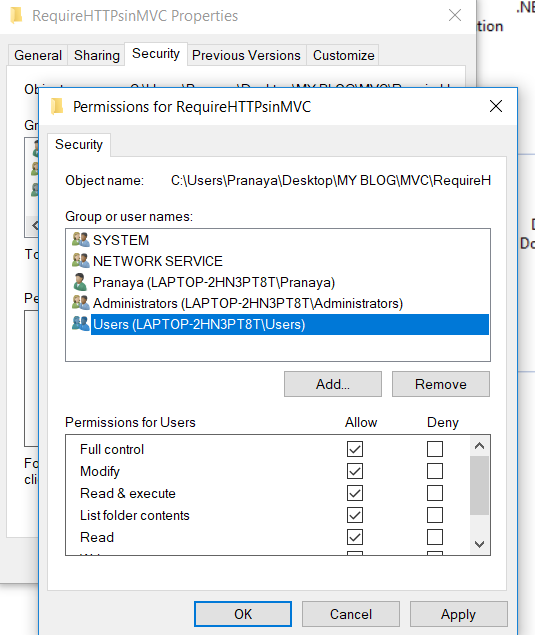
**Right-click on your project in IIS and click on Edit Permission as shown below.**



**Once we click on the Edit Permission the following popup will open. Select the Security tab as shown below.**



**Then select User and click on Edit button which will open the below popup.**

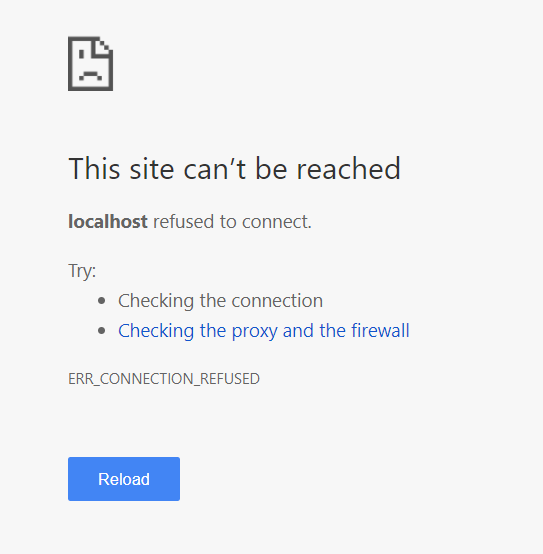


Select Users from Group or user names section, then check all the checkboxes in the Allow section, then click on Apply and then OK. Then click OK. Run the application and navigate to **/Home/Index** and it will work as expected.

##### ****Let’s run the application using the HTTPS protocol****

https://localhost/RequireHTTPsinMVC/Home/Index

When we run the application it will give us the below error



To run the application using Https we need SSL Certificate.

##### ****What are self-signed certificates?****

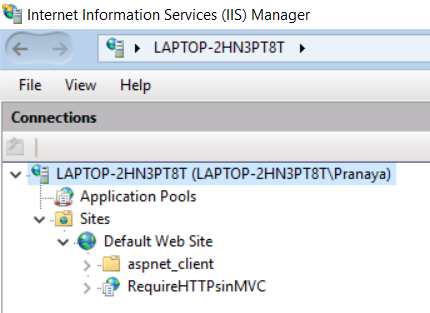
A self-signed certificate is an identity certificate that is signed by its own creator. Certificates are signed by Certificate Authority. In general self-signed certificates are fine for testing purposes and not for production use.

##### ****Creating self-signed certificates****

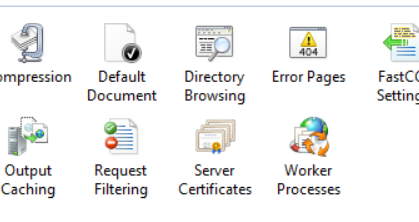
There are several ways to create self-signed test certificates. Let us explore the easier options available. The easiest and simpler approach is to use IIS to create these certificates.

In IIS

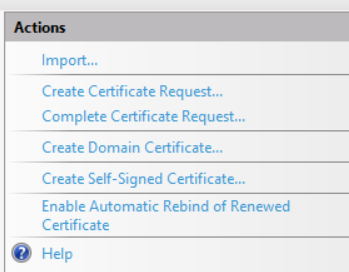
**1. Click on the “Server Name” as shown below**



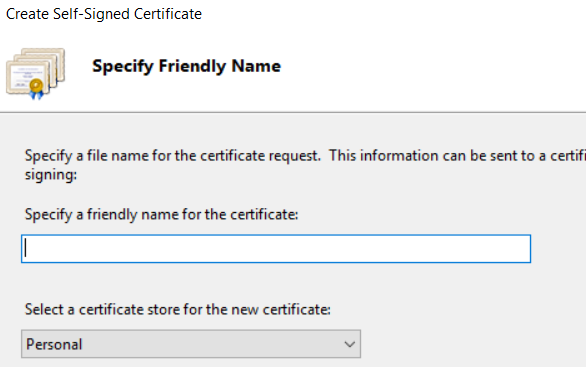
**2. Double click on “Server Certificates” feature**



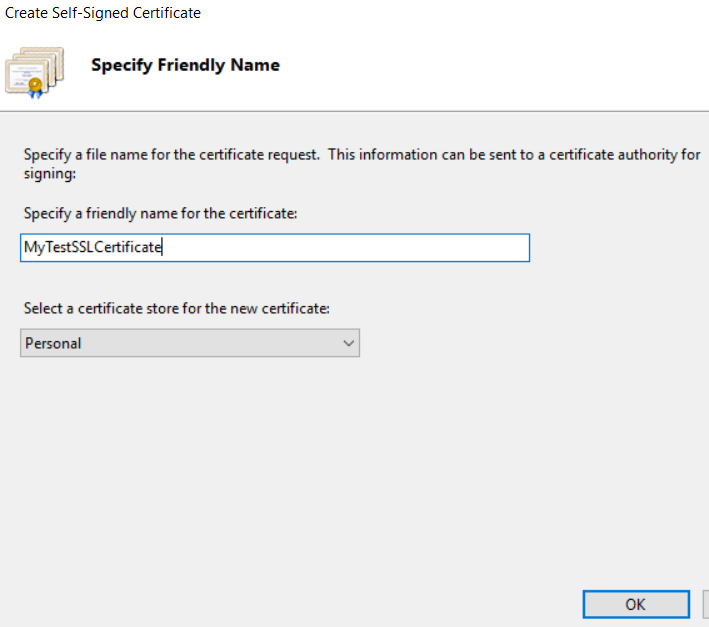
**Once we double click just go to the right-side panel as shown below**



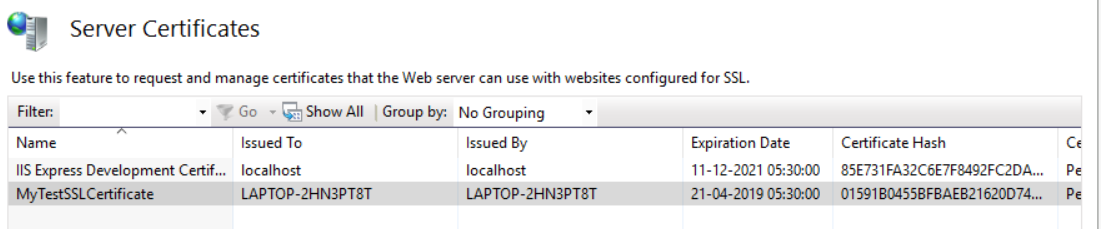
**3. Click on the “Create Self Signed Certificate” link, under “Actions” which will open the below popup.**



**4. Specify a friendly name for the certificate and click OK.**



The friendly name is not part of the certificate itself but is used by the server administrator to easily distinguish the certificate.  Once we click on the Ok button the SSL certificate will be added to the Server Certificate list as shown below.



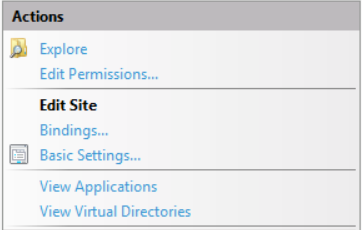
The next step is to Add HTTPS site binding if it is not already present. Open IIS

**Expand the “Server Name”**

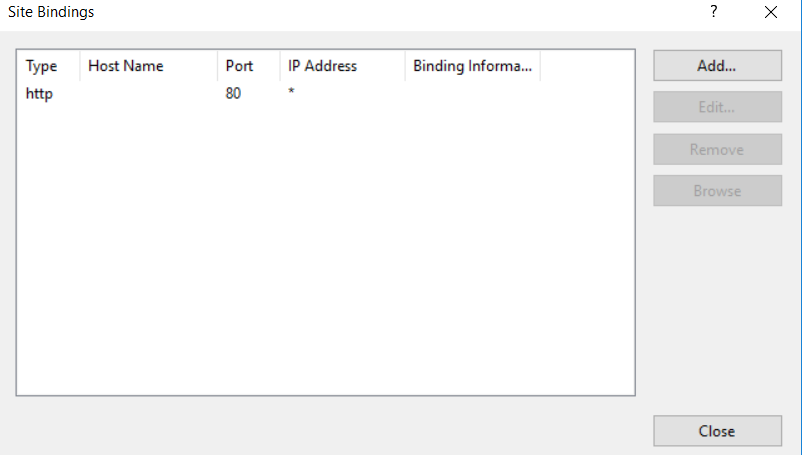
**Expand “Sites”**

**Select “Default Web Site”**

**Click “Binding” under “Edit Site” in the “Actions” pane as shown below.**

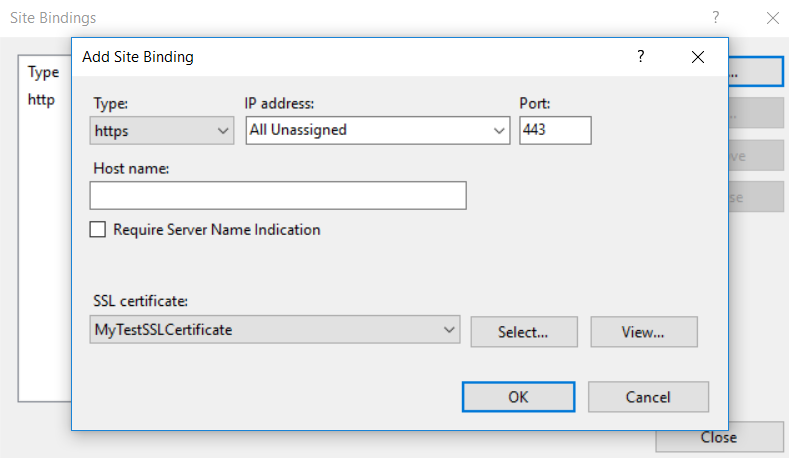


**Once we click on Bindings it will open the below popup**

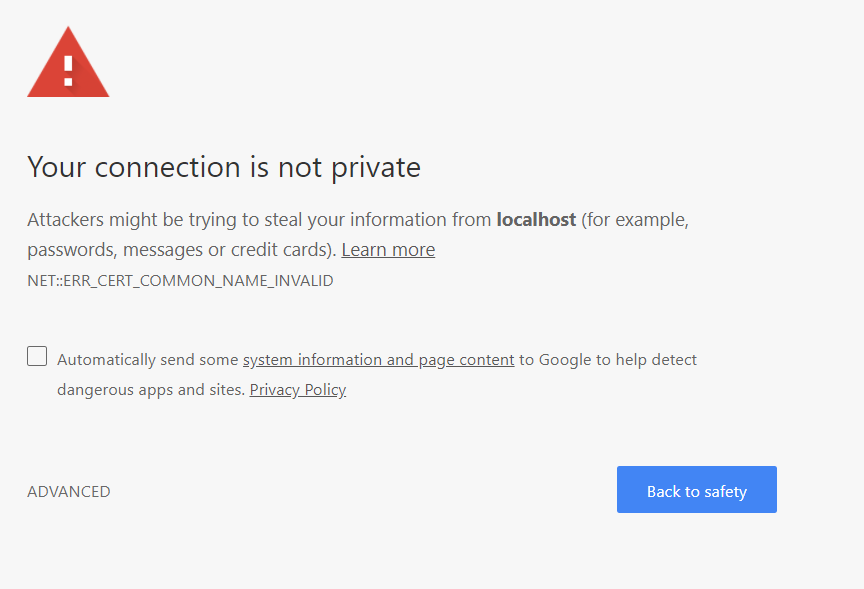


###### **In the “Site Bindings” window, click “Add”**

Once we click on the Add button the below popup will open



Select Type = “https” and the SSL Certificate as MyTestSSLCertificate and click “OK”. Click “Close” on the “Site Bindings” window. At this point, we will be able to access our application using both HTTP and HTTPS protocol. When the site is accessed over HTTPS, we may receive a browser warning about the authenticity of the website as shown below.



Then click on the ADVANCED option and then click on Proceed to localhost (unsafe) and it will display our page as expected. Suppose our requirement is to automatically transfer the Http request to Https then we need to decorate RequireHttps attribute either in the action method or in the controller level as shown below

**[**RequireHttps**]**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

**}**

**}**

Now run the application and see it autumnally transferred the HTTP request to https request.

# Custom Action Filters in ASP.NET MVC

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

In this article, I am going to discuss **how to create and use Custom Action Filters in ASP.NET MVC** Application. Please read our previous article, where we discussed the [**RequireHttps attribute in MVC**](https://dotnettutorials.net/lesson/requirehttps-attribute-mvc/) Application. At the end of this article, you will understand the following pointers.

1. **What are Action Filters in MVC?**
2. **Why do we need Action Filters in ASP.NET MVC?**
3. **Why do we need Custom Action Filters in MVC?**
4. **How to create Custom Action Filter in ASP.NET MVC Application?**

##### ****What are Action Filters in MVC?****

An Action Filter in ASP.NET MVC Application is an attribute that can be applied either at the action methods of a controller or at the controller level directly. So, basically, action filters allow us to execute some custom code or logic either before executing an action method or immediately after the action method completes its execution

##### ****Why do we need Action Filters in the ASP.NET MVC Application?****

As we already discussed, Filters in ASP.NET MVC are used to execute some custom code or logic at different levels of the request processing pipeline. The other advantage is that these Filters can be applied to multiple controllers or multiple action methods of different controllers which means it allows us to share the custom code or logic across Controllers.

For example, let’s say we want to execute some security code or some request and response logging code across the controllers. Then we can write a custom filter containing all those logic and apply that filter wherever you want to execute these custom logic. The following are some of the situations where you may use Filters.

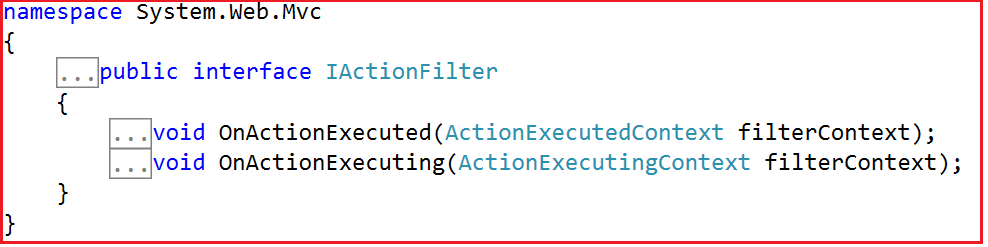
1. Caching
2. Logging
3. Error Handling
4. Authentication and Authorization, etc.

##### ****How to create Custom Action Filter in ASP.NET MVC Application?****

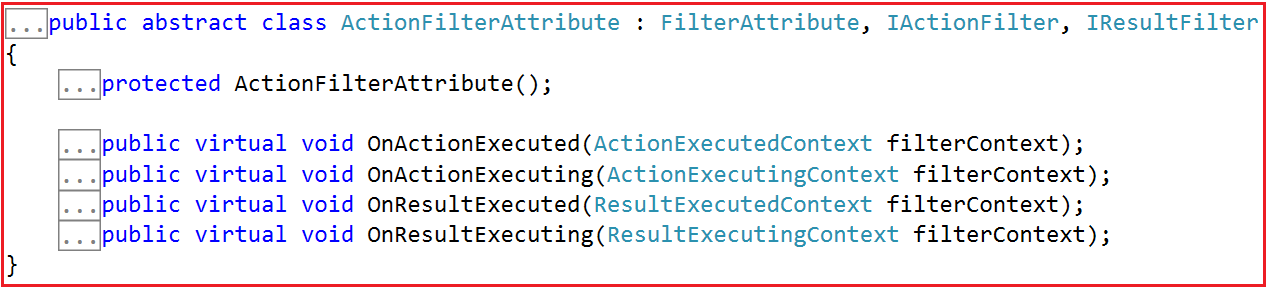
In the ASP.NET MVC application, we can create a custom action filter in two ways. They are as follows

1. First, by implementing the **IActionFilter** interface and inheriting the **FilterAttribute** class.
2. Second, by overriding the **ActionFilterAttribute** abstract class.

The **IActionFilter** interface includes the following methods to implement:



The **ActionFilterAttribute** abstract class includes the following methods which need to be overridden:



So, you can use any of the above approaches to create a Custom Action Filter in ASP.NET MVC Application. Let’s understand when the above four methods are going to be executed.

1. **OnActionExecuting(ActionExecutingContext filterContext)-** This method is going to be execute before executing a controller action method.
2. **OnActionExecuted(ActionExecutedContext filterContext)-** This method is called after a controller action method is executed.
3. **OnResultExecuting(ResultExecutingContext filterContext)-** This method is called before a controller action result is executed.
4. **OnResultExecuted(ResultExecutedContext filterContext)-** This method is called after a controller action result is executed.

##### ****Example:****

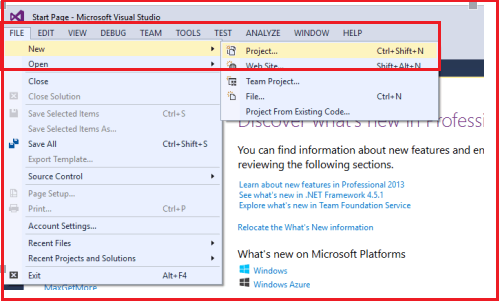
Let us consider a scenario of Logging. For every incoming request, we need to log some data to the files on the basis of some logic. If we don’t create this logic inside a custom filter, then we will have to write the same logic for each controller’s action. This mechanism will lead to two problems:

1. Duplication of code; and
2. Violation of the Single Responsibility Principles. That means action methods will now perform additional tasks of logging.

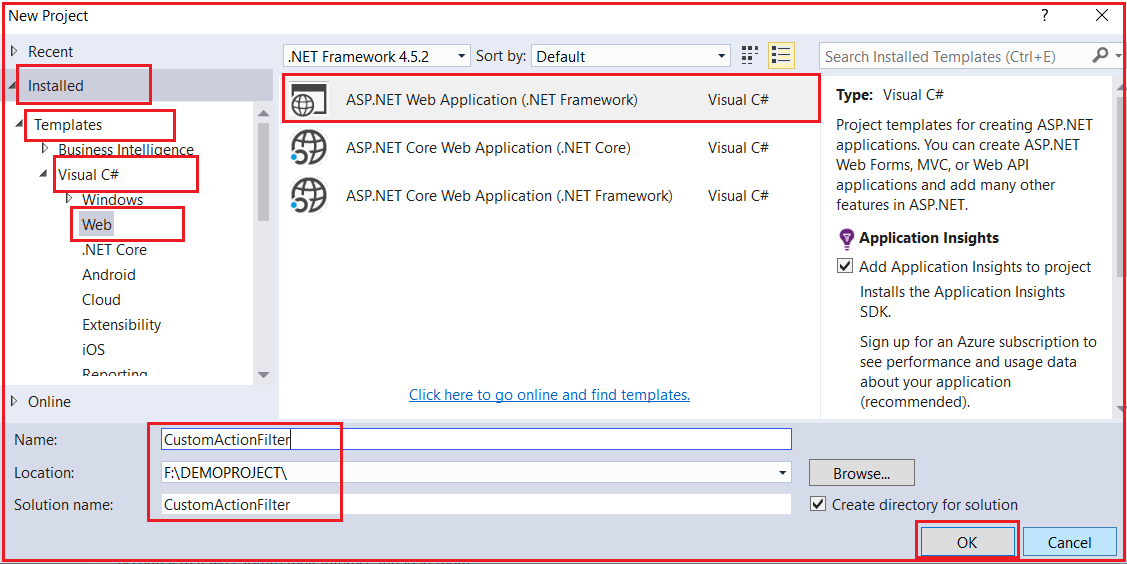
We can solve the above two problems by putting the logging logic inside a custom action filter and applying the filter at all the controllers’ levels.

##### ****Creating a new ASP.NET MVC Application****

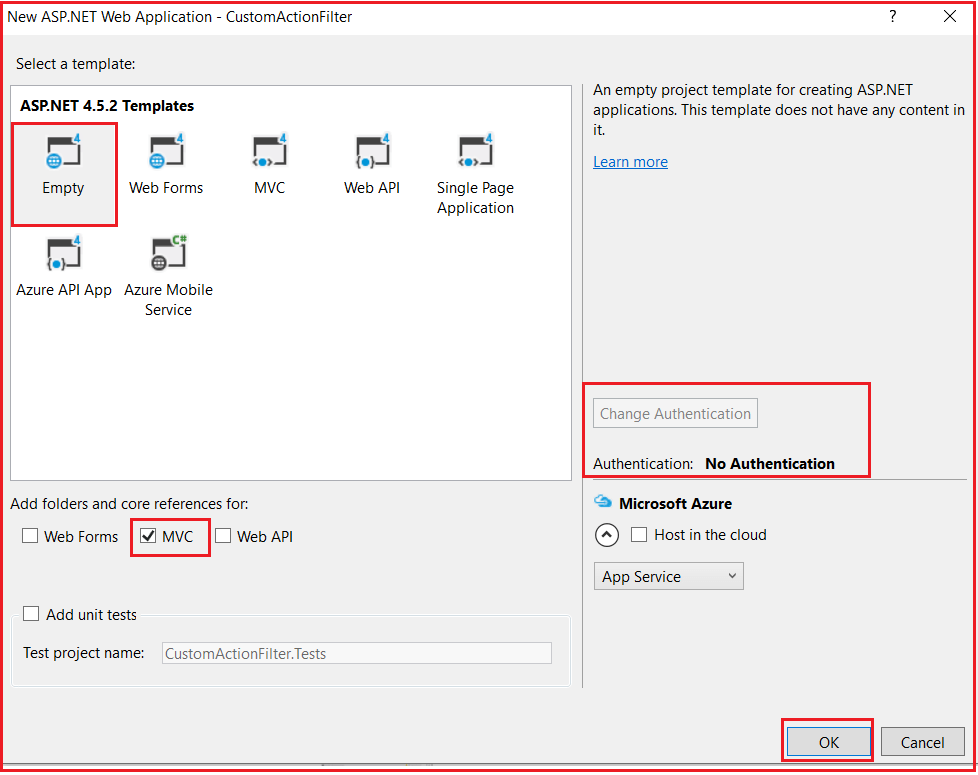
Open visual studio and select **File => New => Project**from the context menu as shown in the below image.



Once you click on the “**Project**” link, a new dialog will pop up. From that window, we are going to choose “**Web**” templates from the left pane. From the middle pane, we need to select “**ASP.NET Web Application**“. Then provide a meaningful name to the project such as “**CustomActionFilter**”. Finally, click on the “**OK**” button as shown in the below image



Once you click on the “**OK”** button a new dialog will pop up with the name “**New ASP.NET Project**” for selecting project Templates as shown in the below image.



In this dialog, we are going to choose the “**Empty”** and “**MVC”** project template with the Authentication type as “**No Authentication”** and then click on the “**OK**” button. Once you click on the OK button it will take some time to create the project for us.

##### ****Adding Log Folder:****

Let’s add a folder called Log into the project’s root directory. To do so, just right-click on the “CustomActionFilter” project and then select **Add** ➜ “**New Folder**“ and name it as Log. In this folder, we are going to create a text file with the **Log.txt** where we will store the Log data.

##### ****Creating Custom Action Filter in MVC:****

Create a class file with the name LogFilter.cs within the Models folder and then copy and paste the following code in it.

**using** *System.Web;*

**using** *System;*

**using** *System.Web.Mvc;*

**using** *System.IO;*

**using** *System.Web.Routing;*

**namespace** *CustomActionFilter.Models*

**{**

**public** **class** LogFilter : FilterAttribute, IActionFilter

**{**

**public** **void** OnActionExecuted**(**ActionExecutedContext filterContext**)**

**{**

Log**(**"OnActionExecuted", filterContext.RouteData**)**;

**}**

**public** **void** OnActionExecuting**(**ActionExecutingContext filterContext**)**

**{**

Log**(**"OnActionExecuting", filterContext.RouteData**)**;

**}**

**private** **void** Log**(**string methodName, RouteData routeData**)**

**{**

var controllerName = routeData.Values**[**"controller"**]**;

var actionName = routeData.Values**[**"action"**]**;

string message = methodName + " Controller:" + controllerName + " Action:" + actionName + " Date: "

+ DateTime.Now.ToString**()** + Environment.NewLine;

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

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

**}**

**}**

**}**

Here we use the first version of creating a Custom Action Filter. That is by implementing the **IActionFilter** interface and inheriting from **FilterAttribute** class so that we can use this class as a Filter.

##### ****Using the Custom Action Filter in ASP.NET MVC:****

Create a Controller with the name Home Controller and then copy and paste the following code into it.

**using** *System.Web.Mvc;*

**using** *CustomActionFilter.Models;*

**namespace** *CustomActionFilter.Controllers*

**{**

**[**LogFilter**]**

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

**return** View**()**;

**}**

**}**

**}**

##### ****Creating the Index and Contact Views:****

###### **Index.cshtml**

@{

ViewBag.Title = "Index";

}

**<h2>**Index View**</h2>**

###### **Contact.cshtml**

@{

ViewBag.Title = "Contact";

}

**<h2>**Contact View**</h2>**

Now run the application and navigate to different pages and see the text file.

##### ****Creating a Custom Action Filter using ActionFilterAttribute:****

Let us modify the LogFilter class as shown below to override the ActionFilterAttribute.

**using** *System.Web;*

**using** *System;*

**using** *System.Web.Mvc;*

**using** *System.IO;*

**using** *System.Web.Routing;*

**namespace** *CustomActionFilter.Models*

**{**

**public** **class** LogFilter : ActionFilterAttribute

**{**

**public** **override** **void** OnActionExecuted**(**ActionExecutedContext filterContext**)**

**{**

Log**(**"OnActionExecuted", filterContext.RouteData**)**;

**}**

**public** **override** **void** OnActionExecuting**(**ActionExecutingContext filterContext**)**

**{**

Log**(**"OnActionExecuting", filterContext.RouteData**)**;

**}**

**public** **override** **void** OnResultExecuting**(**ResultExecutingContext filterContext**)**

**{**

Log**(**"OnResultExecuting", filterContext.RouteData**)**;

**}**

**public** **override** **void** OnResultExecuted**(**ResultExecutedContext filterContext**)**

**{**

Log**(**"OnResultExecuted", filterContext.RouteData**)**;

**}**

**private** **void** Log**(**string methodName, RouteData routeData**)**

**{**

var controllerName = routeData.Values**[**"controller"**]**;

var actionName = routeData.Values**[**"action"**]**;

string message = methodName + " Controller:" + controllerName + " Action:" + actionName + " Date: "

+ DateTime.Now.ToString**()** + Environment.NewLine;

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

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

**}**

**}**

**}**

As you can see, now the LogFilter class is derived from the **ActionFilterAttribute** abstract class and we also override all four methods. Now, it logs before and after the action method or result executes. You can apply the Log attribute to any Controller or action method where you want to log the action. Run the application and open the Log.txt file which is inside Log Folder and checks the log that is generated by the Log Filter.

# Filter Overrides in ASP.NET MVC

# 

## ****Filter Overrides in ASP.NET MVC Application****

ASP.NET MVC 5 and ASP.NET Web API 2 added a very important feature called Filter Overrides. A good definition of filter overrides is available in release notes, “**You can now override which filters apply to a given action method or controller, by specifying an override filter. Override filters specify a set of filter types that should not run for a given scope (action or controller). This allows you to add global filters, but then exclude some from specific actions or controllers**“.

Let’s simplify this definition, ASP.NET MVC 5 has introduced a new feature called Filter Override, which allows us to clear or hide certain Filters created in a higher scope. For example, if we created a global action filter then they are applicable to all controllers or if we apply the filter at controller level then they are applicable to all actions of that controller, but we could override those filters on a case by case basis at the controller action level. This allows us to set global or controller filters that apply almost all cases and just override them in few, specific places where those filters don’t apply.

In this article, I am going to discuss the problem that filter overrides is trying to solve with an example. I will also show you a bug in MVC 5 regarding filter overrides and how to solve that bug in MVC 5.

###### **As we know there are the following five types of filters available with MVC:**

1. Authentication filters
2. Authorization filters
3. Action filters
4. Result filters
5. Exception filters

###### **So we have five type filter overrides corresponding to this:**

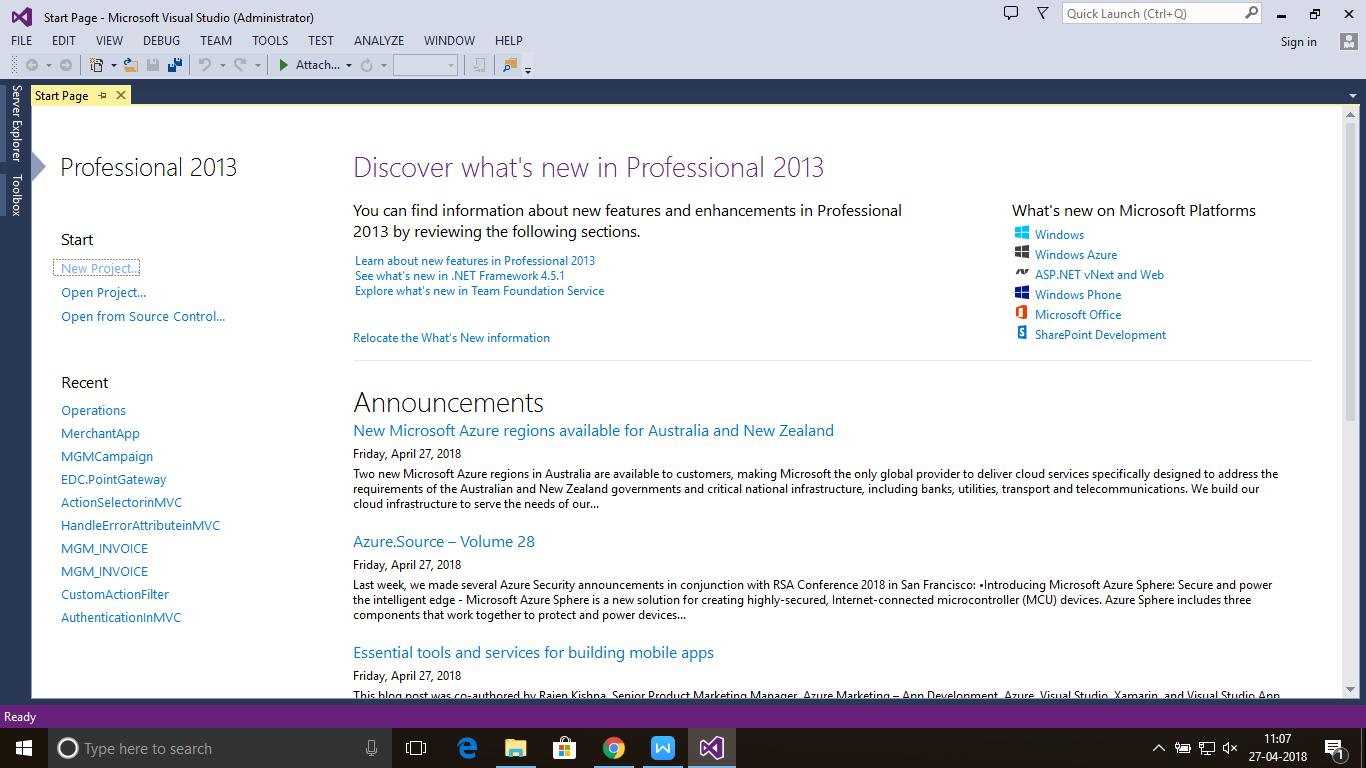
1. OverrideAuthenticationAttribute
2. OverrideAuthorizationAttribute
3. OverrideActionFiltersAttribute
4. OverrideResultAttribute
5. OverrideExceptionAttribute

We can mark any action method with an override filter attribute that essentially clears all filters in an upper scope (in other words controller level or global level).

Let’s understand this with an example, if we created a controller-level action filter or global action filter, now I have an action method on which I do not want to action filter. In this case, the Filter Override feature is very useful.

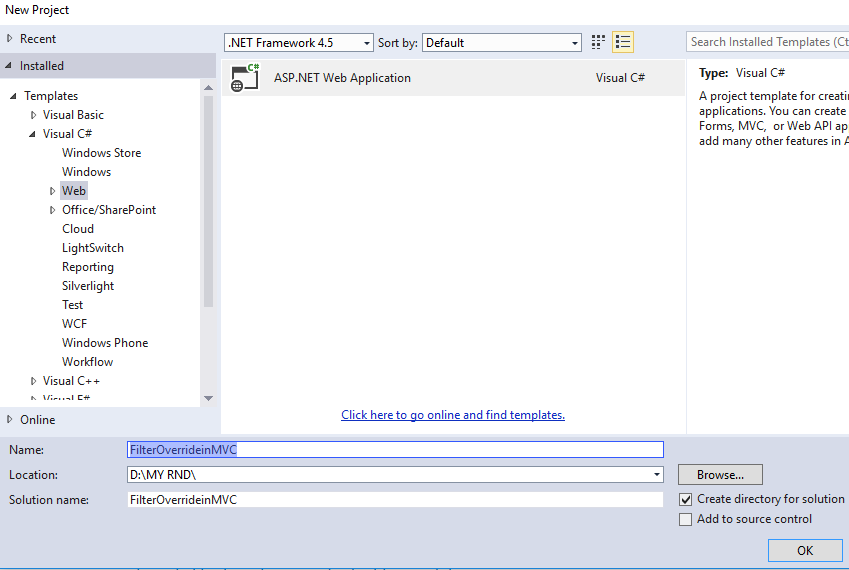
##### ****Create New Asp.Net MVC Application****

From the Visual Studio, Start page click on **“New Project“** link.

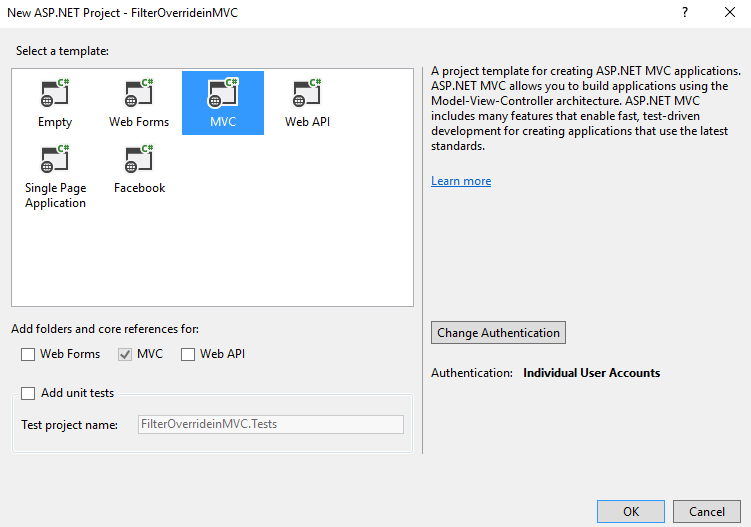


After clicking on “New Project” link a new dialog will pop up.

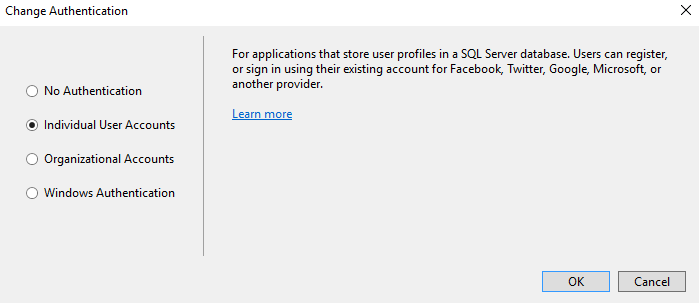
In that we are going to select web templates from the left pane after selecting web template, we find only one project template in it “**ASP.NET Web Application**” just select that.



After selecting this project template next we are going to name the project as **“FilterOverrideinMVC“** and clicking on the OK button a new dialog will pop up with Name **“New ASP.NET Project“** for selecting project Templates.



In this dialog, we are going to choose MVC project template and then we are going to choose Authentication type for doing that just click on Change Authentication button, a new dialog will pop up with the name “Change Authentication” here we are going to choose Individual User Accounts. Then click on OK button as shown below.

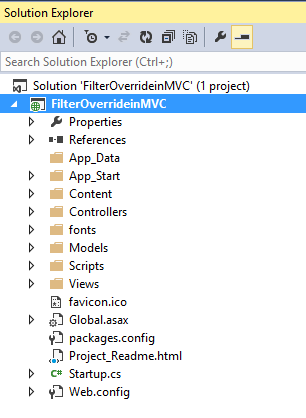


###### **Note: – Individual User Accounts**

If you choose this option for Authentication of application then your application will be configured to use ASP.NET identity [ASP.NET Membership] where User register in the application and then sign in using credentials and also User sign in via social accounts such as Facebook, Twitter, Google, Microsoft and other providers. All user data will be stored in the SQL server database.

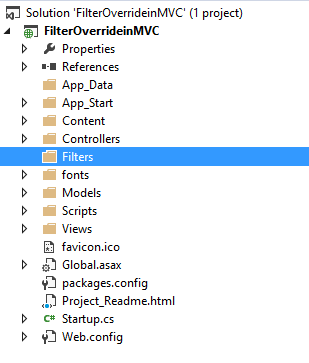
After selecting Authentication type as Individual User Accounts click on OK Button.

After creating the project it will show the folder structure as shown below.



###### **After creating project first thing we are going add Filter folder in Project.**

For adding Folder just right-click on “FilterOverrideinMVC” and then select Add and inside that select “New Folder” then rename the folder name as Filters as shown below.

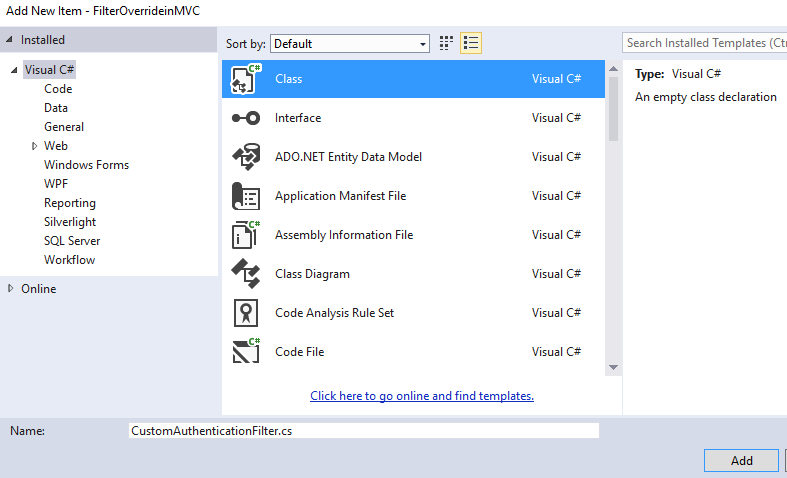


After adding Filter folder the next thing we are going add Filters in this folder to validate User is Logged in to the application or not.

##### ****Adding Authentication Filter****

We are going to add Filter in filters folder with name CustomAuthenticationFilter and this filter will inherit a class FilterAttribute and IAuthenticationFilter, and in this filter we are just going to check Session is IsNullOrEmpty if it is NULL or Empty then we are going to redirect it to Error View else if it is not NULL or Empty then it will allow executing Action Method.

Right click on Filters folder and Click on Add => Class, then provide the class name as “**CustomAuthenticationFilter**” and click on Add as shown below



**Copy and paste the below code in CustomAuthenticationFilter class**

**using** *System.Web.Mvc;*

**using** *System.Web.Mvc.Filters;*

**namespace** *FilterOverrideinMVC.Filters*

**{**

**public** **class** CustomAuthenticationFilter : FilterAttribute, IAuthenticationFilter

**{**

**public** **void** OnAuthentication**(**AuthenticationContext filterContext**)**

**{**

**if** **(**string.IsNullOrEmpty**(**Convert.ToString**(**filterContext.HttpContext.Session**[**"UserID"**])))**

**{**

filterContext.Result = new ViewResult

**{**

ViewName = "Error"

**}**;

**}**

**}**

**public** **void** OnAuthenticationChallenge**(**AuthenticationChallengeContext filterContext**)**

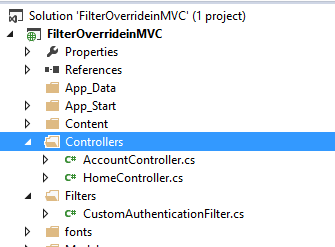
**{**

**}**

**}**

**}**

In next step we are going to apply **CustomAuthenticationFilter Filter** on HomeController, this Controller is created by default when we have created the project.



In this step, we are going to apply CustomAuthenticationFilter Filter to Home controller after applying this filter.

**Applying CustomAuthenticationFilter Filter on HomeController**

The User who is going to access this controller must have **Session[“UserID”]** if it is NULL or empty then it is going to redirect it to Error View.

**namespace** *FilterOverrideinMVC.Controllers*

**{**

**[**CustomAuthenticationFilter**]** //Applied Custom Authentication Filter

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

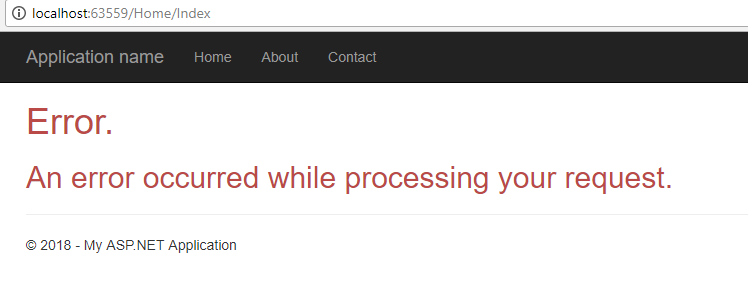
**}**

**}**

**}**

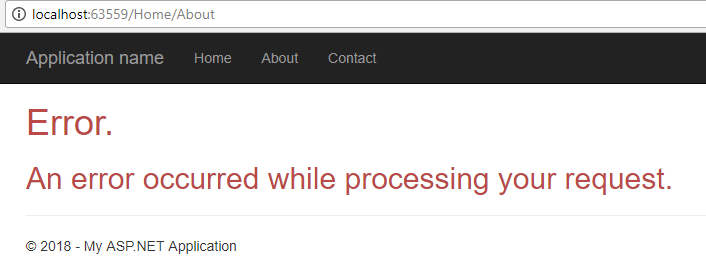
Now let’s Save and Run this Project and access URL:- http://localhost:####/Home/Index

###### **It will redirect it to Error View as shown below.**



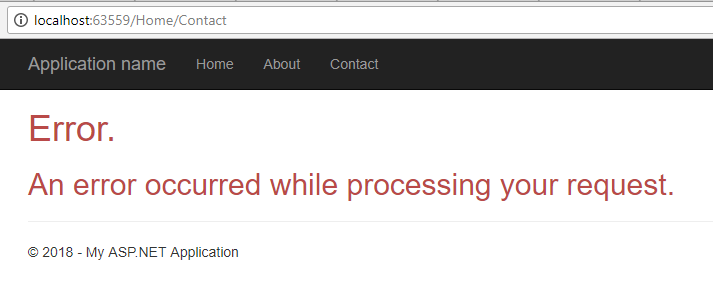
Meanwhile, if we are going to access URL: – http://localhost:xxxx/Home/About

It will redirect it to the Error View as shown below.



Meanwhile, if we are going to access URL: – http://localhost:3025/Home/Contact

It will redirect it to the Error View as shown below.



##### ****Applying OverrideAuthentication Filter on About Action Method****

In this part we are going to apply [OverrideAuthentication] filter on About Action Method.

**namespace** *FilterOverrideinMVC.Controllers*

**{**

**[**CustomAuthenticationFilter**]** //Applied Custom Authentication Filter

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**OverrideAuthentication**]** //Applied override Authentication Filter

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

**}**

**}**

**}**

###### **Scenario 1:-**

If we are going to access Index Action Method still it is going to redirect to Error View because CustomAuthenticationFilter is applied on Controller it means it is applied to all Action Methods inside that Controller.

###### **Scenario 2:-**

If we are going to access Contact Action Method still it is going to redirect to Error View because CustomAuthenticationFilter is applied on Controller it means it is applied to all Action Methods inside that Controller.

###### **Scenario 3:-**

Now we are going to access About Action Method it still redirect to Error View after applying [OverrideAuthentication] Filter on this Action Method,

**Actually, this is a bug in ASP.NET MVC 5 with Filter Overrides that has been fixed in the ASP.NET MVC 5.1 Preview. To get this to work in my example, I created my own [OverrideAuthorization] Attribute that implements IOverrideFilter, etc. As shown below**

Right click on Filters folder and add a class file with name OverrideAuthenticationFilter and copy and paste the below code

**using** *System.Web.Mvc;*

**using** *System.Web.Mvc.Filters;*

**namespace** *FilterOverrideinMVC.Filters*

**{**

**public** **class** OverrideAuthenticationFilter : FilterAttribute, IOverrideFilter

**{**

**public** Type FiltersToOverride

**{**

**get**

**{**

**return** typeof**(**IAuthenticationFilter**)**;

**}**

**}**

**}**

**}**

###### **Now apply this filter to our About method of Home controller as shown below**

**namespace** *FilterOverrideinMVC.Controllers*

**{**

**[**CustomAuthenticationFilter**]** //Applied Custom Authentication Filter

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

//[OverrideAuthentication] // Applied Override Authentication Filter will not work in MVC 5

**[**OverrideAuthenticationFilter**]** //Customizing Override Authentication Filter to work in MVC 5

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

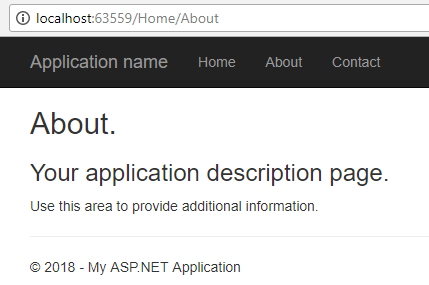
**}**

**}**

**}**

Now let’s Save and Run this Project and access URL: – http://localhost:####/Home/About

###### **It will redirect show the About View as shown below.**

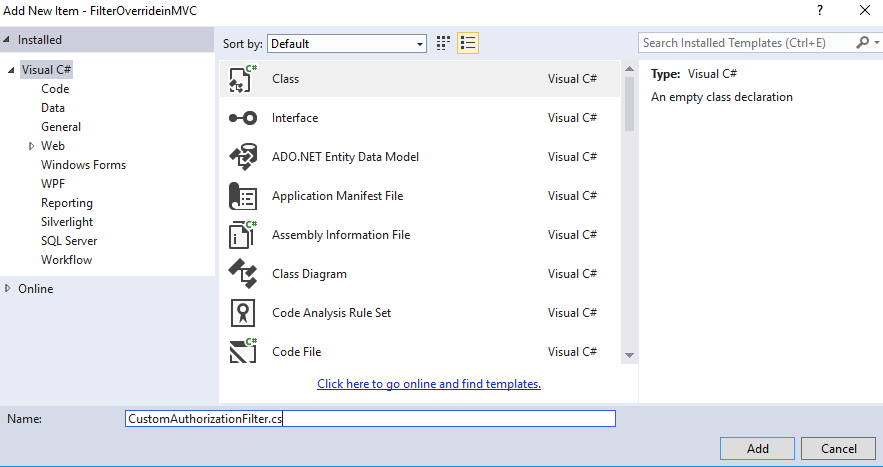


Note: – if we want to override Authentication filter then must only apply [OverrideAuthentication] filter but if we apply other filter such [OverrideAuthorization] or [OverrideActionFilters] or [OverrideResultFilters] or [OverrideExceptionFilters] will not work on it.

##### ****Adding Authorization Filter****

We are going to add Filter in filter folder with name CustomAuthorizationFilter and this filter will inherit a class FilterAttribute and IAuthorizationFilter Interface, and in this filter we are just going to check Session is IsNullOrEmpty if it is NULL or Empty then we are going to redirect it to Error View else if it is not NULL or Empty then it will allow executing Action Method.

First Right click on Filters folder then select Add => Class and provide the class name as CustomAuthorizationFilter and click on add button as shown below



###### **Copy and paste the below code**

**using** *System.Web.Mvc;*

**namespace** *FilterOverrideinMVC.Filters*

**{**

**public** **class** CustomAuthorizationFilter : FilterAttribute, IAuthorizationFilter

**{**

**public** **void** OnAuthorization**(**AuthorizationContext filterContext**)**

**{**

**if** **(**string.IsNullOrEmpty**(**Convert.ToString**(**filterContext.HttpContext.Session**[**"UserID"**])))**

**{**

filterContext.Result = new ViewResult

**{**

ViewName = "Error"

**}**;

**}**

**}**

**}**

**}**

**In next step we are going to apply CustomAuthorizationFilter Filter on HomeController, this Controller is created by default when we have created project.**

**namespace** *FilterOverrideinMVC.Controllers*

**{**

**[**CustomAuthorizationFilter**]** //Applied custom Authorization Filter

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**OverrideAuthorization**]** //Applied override Authorization

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

**}**

**}**

**}**

###### **Scenario1:-**

If we are going to access Index Action Method it is going to redirect to Error View because CustomAuthorizationFilter is applied on Controller it means it is applied to all Action Methods inside that Controller.

###### **Scenario2:-**

If we are going to access Contact Action Method it is going to redirect to Error View because CustomAuthorizationFilter is applied on Controller it means it is applied to all Action Methods inside that Controller.

###### **Scenario3:-**

Now we are going to access About Action Method it is still redirected to Error after applying the OverrideAuthentication Filter in About Action method.

**Actually, this is a bug in ASP.NET MVC 5 with Filter Overrides that has been fixed in the ASP.NET MVC 5.1 Preview. To get this to work this example, we need to create custom [OverrideAuthorization] Attribute that implements IOverrideFilter, etc. As shown below**

Right click on Filters folder and add a class file with name OverrideAuthenticationFilter and copy and paste the below code

**namespace** *FilterOverrideinMVC.Filters*

**{**

**public** **class** OverrideAuthorizationFilter : FilterAttribute, IOverrideFilter

**{**

**public** Type FiltersToOverride

**{**

**get**

**{**

**return** typeof**(**IAuthorizationFilter**)**;

**}**

**}**

**}**

**}**

###### **Now apply this filter to our About method of Home controller as shown below**

**namespace** *FilterOverrideinMVC.Controllers*

**{**

**[**CustomAuthorizationFilter**]** //Applied custom Authorization Filter

**public** **class** HomeController : Controller

**{**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

// [OverrideAuthorization] //Applied override Authorization will not work in MVC 5

**[**OverrideAuthorizationFilter**]** //Customizing Override Authorization Filter to work in MVC 5

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page."

**return** View**()**;

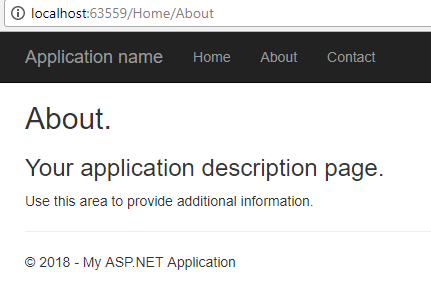
**}**

**}**

**}**

Now let’s Save and Run this Project and access URL: – http://localhost:####/home/About

**It will redirect show the About View as shown below.**



**Note: In the same way we can override Action filter, Exception Filter, and Result Filter.**

**Note**: The Override does not work properly with MVC version 5.0 due to some internal bug. This bug was resolved in MVC version 5.1 (preview).

In the next article, I am going to discuss the [**Authorization Filter in MVC**](https://dotnettutorials.net/lesson/authorization-filter-mvc/)Application.

**SUMMARY:**The Filter Overrides in ASP.NET MVC 5 are very useful when we are implementing a global or controller level filter and we do not want to apply an action filter on some Action methods in the controller. This feature is useful for removing the headache of applying filters for each and every action where we need to exclude only a few actions.

**Authorization Filter in ASP.NET MVC**

**Authorization Filter in ASP.NET MVC Application**

In this article, I am going to the **Authorization Filter in ASP.NET MVC Application**. As part of this article, we are going to discuss the following pointers in detail**.**

1. **Why we need Authorization Filter in MVC?**
2. **What are Authorize and AllowAnonymous action filter in MVC?**
3. **Understanding Authorization Filters in ASP.NET MVC.**
4. **How to use the AllowAnonymous Attribute in ASP.NET MVC?**

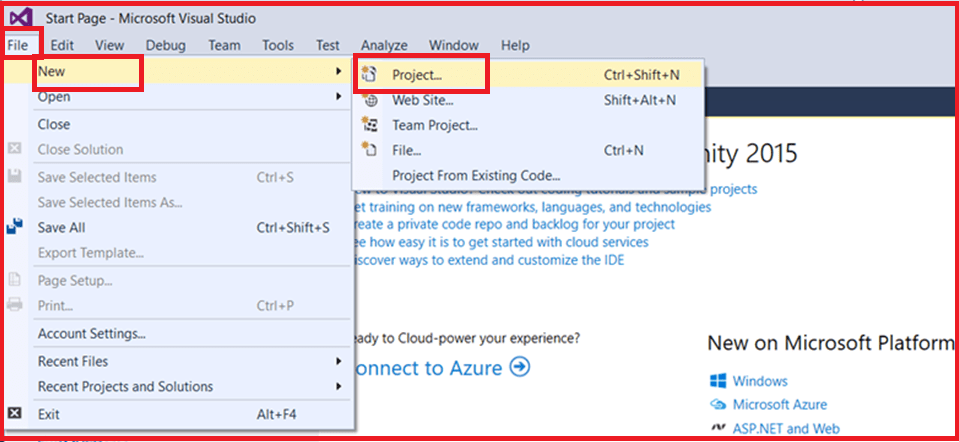
**Why do we need Authorization Filter in ASP.NET MVC?**

By default, in the ASP.NET MVC application, all the action methods of all controllers can be accessed by both authenticated and anonymous users. But if you want the action methods to be available only for the authenticated and authorized users, then you need to use the Authorization Filter in ASP.NET MVC.

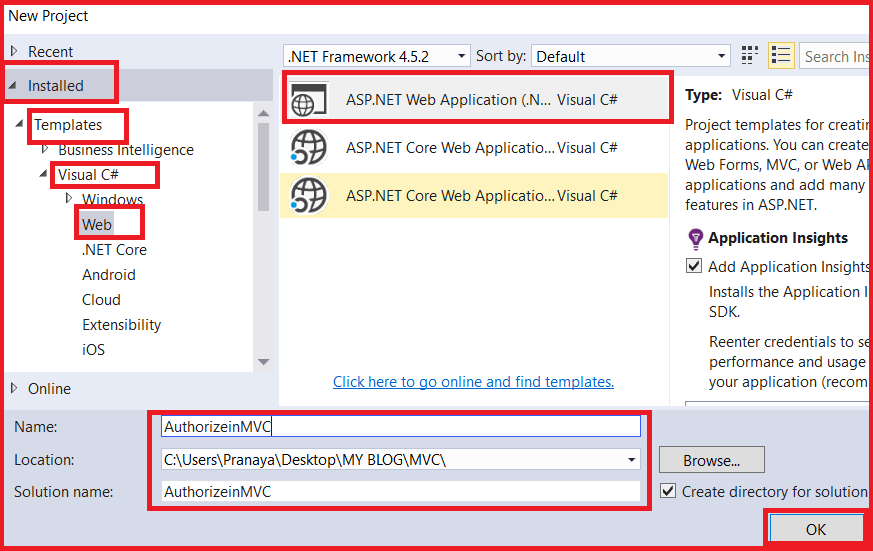
The Authorization Filter provides two built-in attributes i.e. Authorize and AllowAnonymous which we can use as per our business requirement.  Let us understand the “Authorize” and “AllowAnonymous” filters with an example.

**Understanding Authorization Filters in ASP.NET MVC Application:**

In order to understand the Authorization Filters, let’s create a new ASP.NET MVC Application. Open the Visual Studio in Administrator mode and then select **File => New Project** as shown in the below image

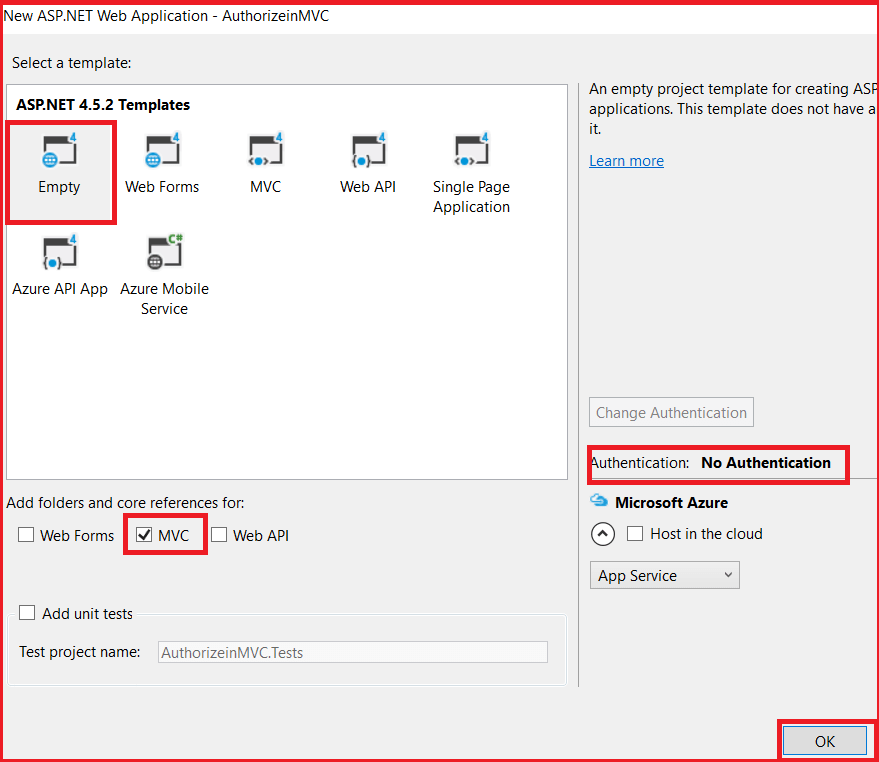


Once you click on the **Project** link the **New Project** dialogue window will open. From the **New Project** window select **Web** tab which is under the **Visual C#** tab which is again under the “**Installed – Templates**” section. From the middle pane select the **ASP.NET Web Application** and name the project as “**AuthorizeinMVC**” and then click on the “**OK**” button as shown in the below image



Once you click on the **OK** button, then a new dialogue window will open with the name **New ASP.NET Project** for selecting the **Project Templates.** From that window select the **Empty**project template as we are going to do everything from scratch. Again From the **add folder and core reference** section select the **MVC** Checkbox as we are going to create an MVC application.

Here we need to change the Authentication type for doing that just click on **Change Authentication** button. A new dialog will pop up with the name **Change Authentication** here we are going to choose **No Authentication** and then click on the **OK** button as shown in the below image.



Once you click on the OK button, then it will take some time to create the project for us.

**Creating the Home Controller:**

Right-click on the **“Controllers”** folder and add a controller with the name **HomeController**. Once you create the Home Controller then Copy and paste the following code**.**

**public** **class** HomeController : Controller

**{**

**public** ActionResult NonSecureMethod**()**

**{**

**return** View**()**;

**}**

**public** ActionResult SecureMethod**()**

**{**

**return** View**()**;

**}**

**}**

As you can see we create the above HomeController with two action methods i.e. NonSecureMethod and SecureMethod. We want the secure method to be accessed by authenticated users while the non-secure method to be accessed by anyone.

**Creating Login Controller**

Again, right-click on the **“Controllers”** folder and add a controller with the name **LoginController**. Once you create the Login controller then copy and paste the following code in it**.**

**public** **class** LoginController : Controller

**{**

**public** ActionResult Login**()**

**{**

**return** View**()**;

**}**

**}**

As you can see, we create the above Login Controller with one action method i.e. Login. Whenever an unauthenticated user wants to access the secure method then we need to redirect that user to the Login action method.

**Creating NonSecureMethod View**

Right-click on the **NonSecureMethod()** and then add a view with name **NonSecureMethod**. Once you create the view then copy and paste the following code in it.

@{

ViewBag.Title = "NonSecureMethod";

}

**<h2>**This method can be accessed by everyone as it is non-secure method**</h2>**

**Creating SecureMethod View**

Similarly, right-click on SecureMethod() and add a view with name SecureMethod. Then copy and paste the following code in **SecureMethod.cshtml** view

@{

ViewBag.Title = "SecureMethod";

}

**<h2>**This method needs to be access by authorized users as it SecureMethod**</h2>**

**Creating Login View**

Similarly Right-click on the **Login()** method of **Login Controller** and add the view with name **Login.cvshtml**. Then copy and paste the following code in **Login.cshtml** view

@{

ViewBag.Title = "Login";

}

**<h2>**Login Page**</h2>**

At this point, both authenticated and anonymous users can access both the “SecureMethod” and the “NonSecureMethod” method by using the following two URLs.  
**/Home/SecureMethod**  
**/Home/NonSecureMethod**

If you want the **“**SecureMethod**“** to be accessed only by the authenticated and authorized users, then you need to decorate this method with the **“**Authorize**“** attribute as shown below.

**[**Authorize**]**

**public** ActionResult SecureMethod**()**

**{**

**return** View**()**;

**}**

**Changing the web.config file**

Made the following changes in the **web.config** file. What we are doing here is, if the user is an unauthorized user, then we are just navigating that user to **Login** Page. So, add the following code under the **system.web** section of the **web.config** file.

**<**authentication mode="Forms"**>**

**<**forms loginUrl="/Home/Login"**><**/forms**>**

**<**/authentication**>**

That’s it. Now run the application and navigate to /**Home/SecureMethod**. Then you will see that it will redirect you to the Login page. On the other hand, you can access the NonSecure method. Now remove the **Authorize** attribute from the **SecureMethod**of **HomeController.**Then apply the **Authorize**attribute at the controller level as shown below.

[Authorize]

public class HomeController : Controller

{

public ActionResult NonSecureMethod()

{

return View();

}

public ActionResult SecureMethod()

{

return View();

}

}

When you apply the Authorize attribute at the controller level then it is applicable to all the action methods that are present within that controller.  Here all the action methods of Home Controller are now protected with the Authorize Attribute, So, now only the authenticated users can access both SecureMethod() and NonSecureMethod().

**How to use the AllowAnonymous Attribute in MVC?**

If you want to allow anonymous access to the NonSecureMethod of Home controller, then you need to decorate the AllowAnonymous attribute to that NonSecureMethod method as shown below. The AllowAnonymous attribute in MVC is used to skip the authorization which is enforced by Authorization Filter in MVC.

**[**Authorize**]**

**public** **class** HomeController : Controller

**{**

**[**AllowAnonymous**]**

**public** ActionResult NonSecureMethod**()**

**{**

**return** View**()**;

**}**

**public** ActionResult SecureMethod**()**

**{**

**return** View**()**;

**}**

**}**

Now, run the application and navigate to **/Home/NonSecureMethod** and you will see that it display the page as expected and when you navigate to **/Home/SecureMethod** then it will redirect you to the Login page.

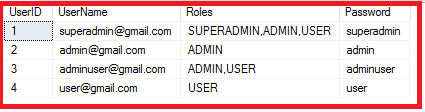
# Custom Authorization Filter in MVC

## ****Custom Authorization Filter in ASP.NET MVC****

In this article, I am going to discuss **Custom Authorization Filter in ASP.NET MVC Application** with an example. Please read our previous article before proceeding to this article where we discussed the basics of [**Authorization Filter in MVC**](https://dotnettutorials.net/lesson/authorization-filter-mvc/) application. There are certain scenarios in your projects on which you may need to customize the Authorization Attribute instead of using the built-in Authorization Attribute. So let us discuss with an example of when and how to customizing the authorization filter in ASP.NET MVC Application and mapping it to the default Authorize filter.

###### **SQL Script for this demo:**

We are going to use the following USERS table in this demo



Please use below SQL Script to create and populate the USERS table with the required data.

**CREATE** **TABLE** **USERS**

(

UserID **BIGINT** **PRIMARY** **KEY** **IDENTITY**,

UserName **VARCHAR**(100) NOT **NULL**,

Roles **VARCHAR**(100) NOT **NULL**,

[Password] **VARCHAR**(100) NOT **NULL**,

)

-- UserID is the auto-generated unique id

-- UserName is the unique value i.e. Email ID of the user

-- Roles can be ADMIN, SUPERADMIN OR USER

-- WE CAN PROVIDE 1 OR MORE USER SEPARATED BY ','

--INSERT THE FOLLOWING TEST DATA FOR TESTING PURPOSE

**INSERT** **INTO** **USERS** **VALUES**('superadmin@gmail.com','SUPERADMIN,ADMIN,USER', 'superadmin')

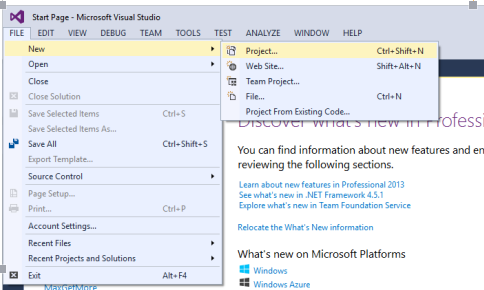
**INSERT** **INTO** **USERS** **VALUES**('admin@gmail.com','ADMIN', 'admin')

**INSERT** **INTO** **USERS** **VALUES**('adminuser@gmail.com','ADMIN,USER', 'adminuser')

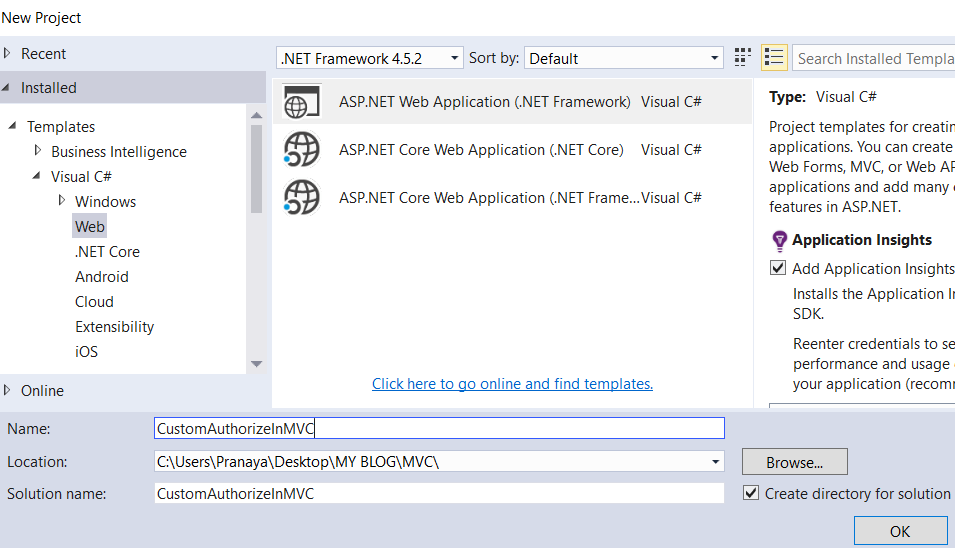
**INSERT** **INTO** **USERS** **VALUES**('user@gmail.com','USER', 'user')

##### ****Create a new ASP.NET MVC application:****

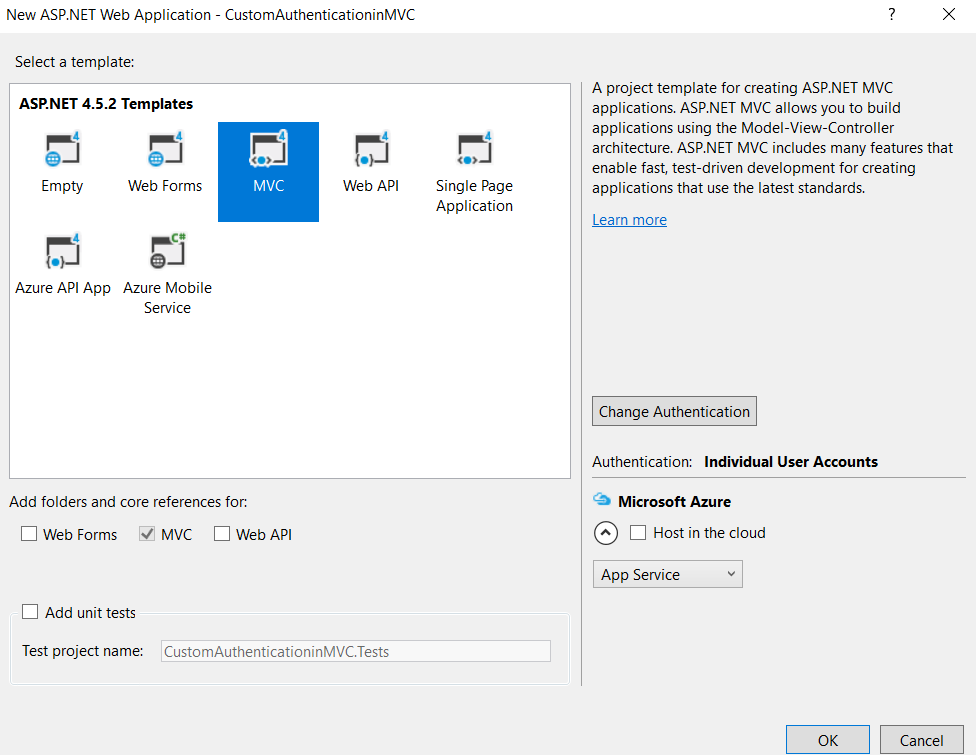
Open Visual Studio in Administrator mode and create a new project. To do so, **select File => New => Project**option as shown in the below image.



After clicking on the “Project” link a new dialog will pop up. From that window, we are going to select web templates from the left pane and from the middle pane, select the **“ASP.NET Web Application”** template. Provide a meaningful name such as “**CustomAuthenticationinMVC**” to your project and click on the **OK** button as shown below.



Once you click on the **OK** button a new dialog will pop up for selecting the project template. In this dialog, we are going to choose the **MVC** project template and then we are going to choose Authentication type. For selecting the Authentication type, just click on the **Change Authentication** button, a new dialog will pop up with the name “**Change Authentication**” here we are going to choose “**Individual User Accounts**” and then click on the **OK** button as shown below.



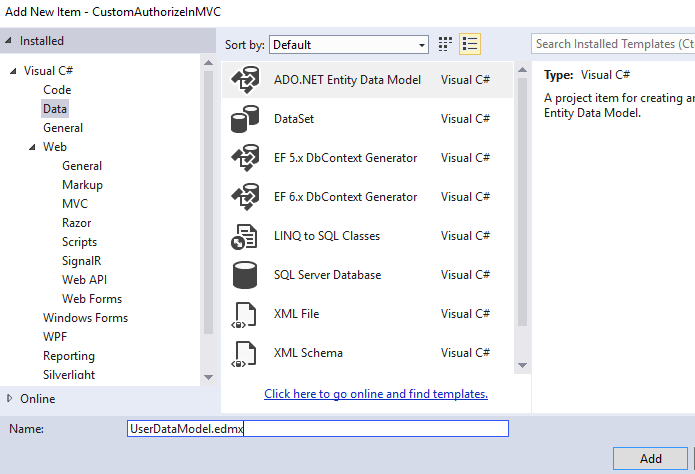
Once you click on the OK button, it will take some time to create the project for us.

##### ****Add a folder called DAL to the project.****

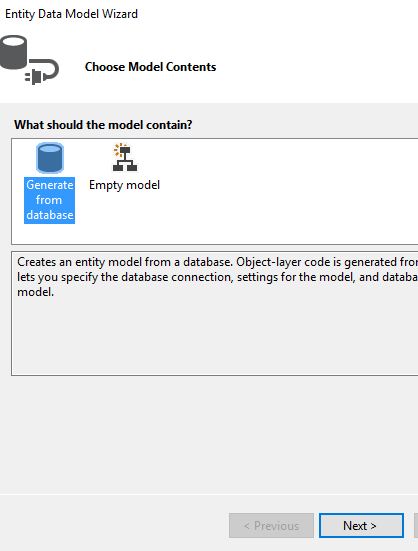
To create a folder, right-click on the **Project => Add => New Folder**option which will create a new folder and then rename the folder name as DAL

##### ****Adding ADO.NET Data Model****

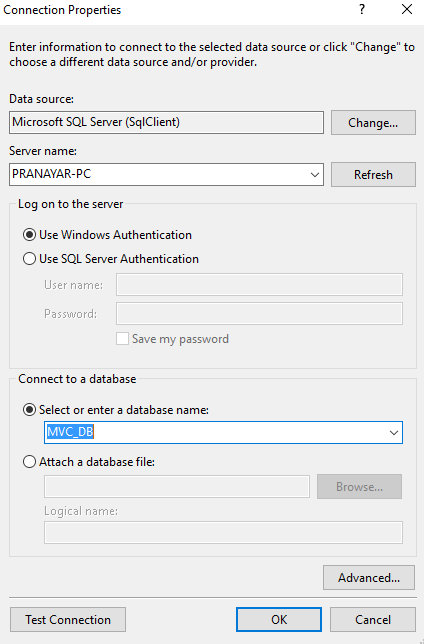
Right-click on the **DAL** folder then select **Add => New Item**from the context menu. Then select **Data** from the left pane and from the middle pane select **ADO.NET data Model**, Provide a meaningful name and click on the **Add** button as shown below.



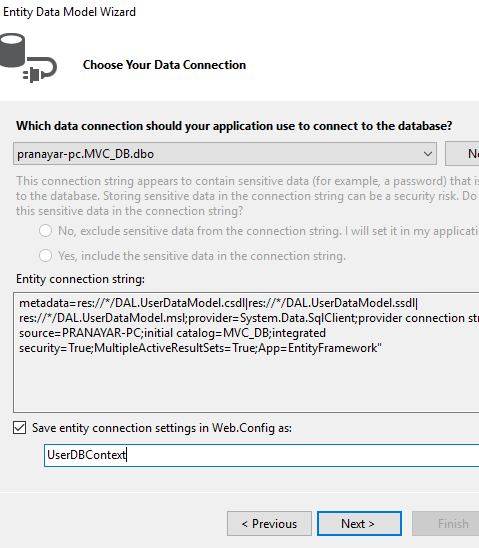
From the next window, select **Generate From Database** and click on the **Next** button as shown below. As our database is already created, so we are going to use the database first approach of entity framework. This is the reason why we choose the Generate From Database option.



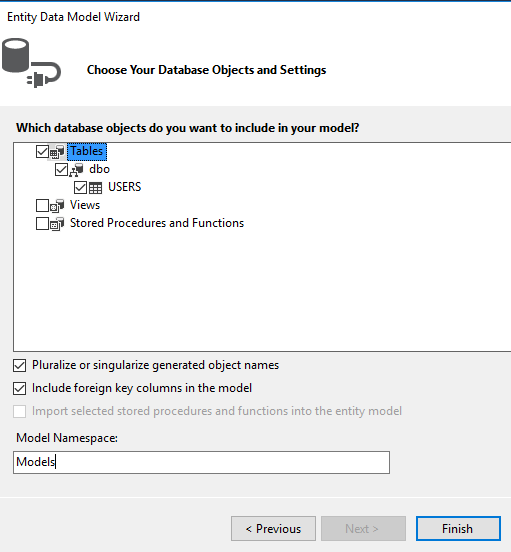
In the next screen, click on the **New Connection** and provide the necessary details, select the database where you created the **USERS** table and click on the **OK** button as shown below.



Provide a meaningful name for the **Connection String** that is going to create in **Web.config** file and click on the **Next** button as shown  in the below image.



From the Choose your database objects screen, choose the **USERS** object, provide the namespace and click on the **Finish** button as shown below.



That’s it. our model is ready.

##### ****Create ENUM to store user Roles****

For our needs, we will create the following Enum to declare roles: Right-click on the Models folder and add a class File with the name **Role.cs** and then copy and paste the following code

**public** enum Role

**{**

SUPERADMIN = 1,

ADMIN = 2,

USER = 3,

**}**

##### ****Create a Repository to perform DB Operations****

Right-click on the Models folder and add a class File with the name Repository.cs and copy and paste the following code

**public** **class** Repository

**{**

**public** USER GetUserDetails**(**string UserName, string Password**)**

**{**

USER user = new USER**()**;

**using** **(**UserDBContext db = new UserDBContext**())**

**{**

user = db.USERS.FirstOrDefault**(**u =**>** u.UserName.ToLower**()** == UserName.ToLower**()** &&

u.Password == Password**)**;

**}**

**return** user;

**}**

**}**

##### ****Modify the Login POST method****

Open AccountController that is present in Controllers Folder and Import the Following Namespaces

**using CustomAuthorizeInMVC.Models;**  
**using CustomAuthorizeInMVC.DAL;**  
**using System.Web.Security;**

Goto method called **Login(LoginViewModel model, string returnUrl)** whose type is POST. Modify the Login POST method as shown below

**[**HttpPost**]**

**[**AllowAnonymous**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult Login**(**LoginViewModel model, string returnUrl**)**

**{**

**if** **(**!ModelState.IsValid**)**

**{**

**return** View**(**model**)**;

**}**

Repository repo = new Repository**()**;

USER user = repo.GetUserDetails**(**model.UserName, model.Password**)**;

**if** **(**user != **null)**

**{**

FormsAuthentication.SetAuthCookie**(**user.UserName, model.RememberMe**)**;

FormsAuthentication.SetAuthCookie**(**Convert.ToString**(**user.UserID**)**, model.RememberMe**)**;

var authTicket = new FormsAuthenticationTicket**(**1, user.UserName, DateTime.Now, DateTime.Now.AddMinutes**(**20**)**, **false**, user.Roles**)**;

string encryptedTicket = FormsAuthentication.Encrypt**(**authTicket**)**;

var authCookie = new HttpCookie**(**FormsAuthentication.FormsCookieName, encryptedTicket**)**;

HttpContext.Response.Cookies.Add**(**authCookie**)**;

//Based on the Role we can transfer the user to different page

**return** RedirectToAction**(**"Index", "Home"**)**;

**}**

**else**

**{**

ModelState.AddModelError**(**"", "Invalid login attempt."**)**;

**return** View**(**model**)**;

**}**

**}**

##### ****Modify the LogOff method of Account Controller as shown below****

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult LogOff**()**

**{**

FormsAuthentication.SignOut**()**;

**return** RedirectToAction**(**"Index", "Home"**)**;

**}**

##### ****Modifying Global.asax.cs file****

Open **Global.asax.cs** file and then copy and paste the following code in it.

**using** *System.Web.Security;*

**public** **class** MvcApplication : System.Web.HttpApplication

**{**

**protected** **void** Application\_Start**()**

**{**

AreaRegistration.RegisterAllAreas**()**;

FilterConfig.RegisterGlobalFilters**(**GlobalFilters.Filters**)**;

RouteConfig.RegisterRoutes**(**RouteTable.Routes**)**;

BundleConfig.RegisterBundles**(**BundleTable.Bundles**)**;

**}**

**protected** **void** Application\_PostAuthenticateRequest**(**Object sender, EventArgs e**)**

**{**

var authCookie = HttpContext.Current.Request.Cookies**[**FormsAuthentication.FormsCookieName**]**;

**if** **(**authCookie != **null)**

**{**

FormsAuthenticationTicket authTicket = FormsAuthentication.Decrypt**(**authCookie.Value**)**;

**if** **(**authTicket != **null** && !authTicket.Expired**)**

**{**

var roles = authTicket.UserData.Split**(**','**)**;

HttpContext.Current.User = new System.Security.Principal.GenericPrincipal**(**new FormsIdentity**(**authTicket**)**, roles**)**;

**}**

**}**

**}**

**}**

**Add AuthorizeRole.cs class file in the Models folder and copy and paste the following code.**

**[**AttributeUsage**(**AttributeTargets.Method | AttributeTargets.Class, Inherited = **true**, AllowMultiple = **true)]**

**public** **class** AuthorizeRoleAttribute : AuthorizeAttribute

**{**

**public** AuthorizeRoleAttribute**(params** **object[]** roles**)**

**{**

**if** **(**roles.Any**(**r =**>** r.GetType**()**.BaseType != typeof**(**Enum**)))**

**throw** new ArgumentException**(**"roles"**)**;

this.Roles = string.Join**(**",", roles.Select**(**r =**>** Enum.GetName**(**r.GetType**()**, r**)))**;

**}**

**}**

##### ****Settings required for role-based Authentication.****

Let’s see what have we created and how are we going to use those

1. We have created 3 users ADMIN, SUPERADMIN, and USER
2. Now in Home Controller “**Index**” method we will give access to both Super admin and admin, for the “About” method will give access to only Super Admin and for the Contact method, we give access to only Admin.

Decorate **HomeController**with **[Authorize]** attribute first to restrict unauthorized access, decorate the remaining three methods with respective roles as shown below.

**[**Authorize**]**

**public** **class** HomeController : Controller

**{**

**[**AuthorizeRole**(**Role.SUPERADMIN, Role.ADMIN**)]**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**AuthorizeRole**(**Role.SUPERADMIN**)]**

**public** ActionResult About**()**

**{**

ViewBag.Message = "Your application description page.";

**return** View**()**;

**}**

**[**AuthorizeRole**(**Role.ADMIN**)]**

**public** ActionResult Contact**()**

**{**

ViewBag.Message = "Your contact page.";

**return** View**()**;

**}**

**[**AuthorizeRole**(**Role.USER**)]**

**public** ActionResult UserPage**()**

**{**

ViewBag.Message = "User Page";

**return** View**()**;

**}**

**}**

**Note:** we need to Create a UserPage action method as this method is not created manually. Along with creating the UserPage View and copy-paste the following code in UserPage.cshtml file

@{

ViewBag.Title = "User Page";

}

**<h2>**@ViewBag.Title.**</h2>**

**<h3>**@ViewBag.Message**</h3>**

Now we need to do a small change in our \_Layout.cshtml file which is inside Shared Folder. Add the following line below **<li>@Html.ActionLink(“Contact”, “Contact”, “Home”)</li>**

**<li>@Html.ActionLink(“User Page”, “UserPage”, “Home”)</li>**

That’s it we are ready with our application with custom authentication and authorization. Now let’s run the application, as we have decorated HomeController with the [Authorize] attribute, we will get the Login page first instead of the Default HomeController Index method. If we see the URL it is not directly called **Account/Login** method, there is extra ReturnUrl

**http://localhost:58040/Account/Login?ReturnUrl=%2F**

##### ****Let’s see the route config****

**public** **class** RouteConfig

**{**

**public** **static** **void** RegisterRoutes**(**RouteCollection routes**)**

**{**

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

routes.MapRoute**(**

name: "Default",

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

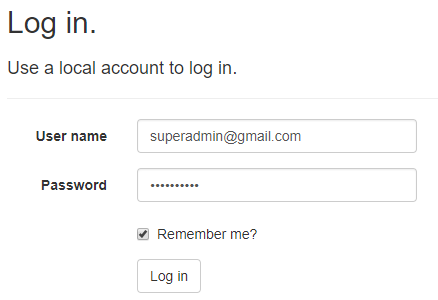
defaults: new **{** controller = "Home", action = "Index", id = UrlParameter.Optional **}**

**)**;

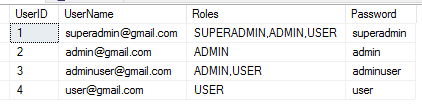
**}**

**}**

So when it goes to HomeController directly it doesn’t get authenticated so it redirects to the Login page in AccountController. Now enter the required credentials created by us in the database. I am entering Super admin details and submit the page as shown below



Then it will navigate us to the index page as the index page is accessible to Super Admin. Now click on every page and all pages are accessible to Super admin as we set in the database as shown below.



Now Login with admin@gmail.com and try to navigate to UserPage link and it will navigate to the Login page as UserPage does not have access to Admin.

##### ****There is a problem:****

When the user is authenticated and if the user does not have access to a particular page then instead of Navigating to the Login page we need to navigate to Access denied page.

**Let’s modify the Requirement.**

If the user is not authenticated navigate to the Login Page. If the user is authenticated but Access is not given for a particular page then navigate to the Access Denied page.

**Add the AccessDenied action method with AllowAnonymous attribute as shown below**

**[**AllowAnonymous**]**

**public** ActionResult AccessDenied**()**

**{**

**return** View**()**;

**}**

##### ****Add AccessDenied View and Copy and Paste the following code as shown below****

@**{**

ViewBag.Title = "Unauthorized Access";

**}**

**<**h1 **class**="text-danger"**>**Unauthorized Access**<**/h1**>**

**<**h2 **class**="text-danger"**>**You don’t have permission to view this page.**<**/h2**>**

**Then open AuthorizeRole.cs file which is in Models folder and then copy and paste the following code**

**[**AttributeUsage**(**AttributeTargets.Method | AttributeTargets.Class, Inherited = **true**, AllowMultiple = **true)]**

**public** **class** AuthorizeRoleAttribute : AuthorizeAttribute

**{**

**public** AuthorizeRoleAttribute**(params** **object[]** roles**)**

**{**

**if** **(**roles.Any**(**r =**>** r.GetType**()**.BaseType != typeof**(**Enum**)))**

**throw** new ArgumentException**(**"roles"**)**;

this.Roles = string.Join**(**",", roles.Select**(**r =**>** Enum.GetName**(**r.GetType**()**, r**)))**;

**}**

**public** **override** **void** OnAuthorization**(**AuthorizationContext filterContext**)**

**{**

**base**.OnAuthorization**(**filterContext**)**;

**if** **(**!filterContext.HttpContext.User.Identity.IsAuthenticated**)**

**{**

filterContext.Result = new RedirectResult**(**"~/Account/Login"**)**;

**return**;

**}**

**if** **(**filterContext.Result **is** HttpUnauthorizedResult**)**

**{**

filterContext.Result = new RedirectResult**(**"~/Account/AccessDenied"**)**;

**return**;

**}**

**}**

**}**

Now build the application and run. Check everything is working as expected. In the next article, I am going to discuss the [**Custom Authentication Filter in MVC**](https://dotnettutorials.net/lesson/customizing-authentication-filter-mvc/) Application. Here, in this article, I try to explain the **Custom Authorization Filter in MVC** application step by step with a real-time example.

**Custom Authentication Filter in MVC**

**Custom Authentication Filter in ASP.NET MVC Application**

In this article, I am going to discuss how to create a **Custom Authentication Filter in MVC** application. The Authentication Filter was introduced with MVC 5 and provides a great improvement for authenticating a user. As of now, there is no in-built Authentication Filer in MVC. So, if you want to use, then the one and the only way to create a custom authentication filter and use that filter in your application.

As of now, we have used two built-in filters i.e. [**Authorize**](https://dotnettutorials.net/lesson/authorization-filter-mvc/)and [**AllowAnonymous**](https://dotnettutorials.net/lesson/authorization-filter-mvc/). The Authorize filter performs the authorization tasks for an authenticated user. A good example is Role-based authorization. The AllowAnonymous filter allows anonymous users to access certain Controllers/Actions. In this way, we can protect the entire application by using the Authorize and AllowAnonymous attribute.

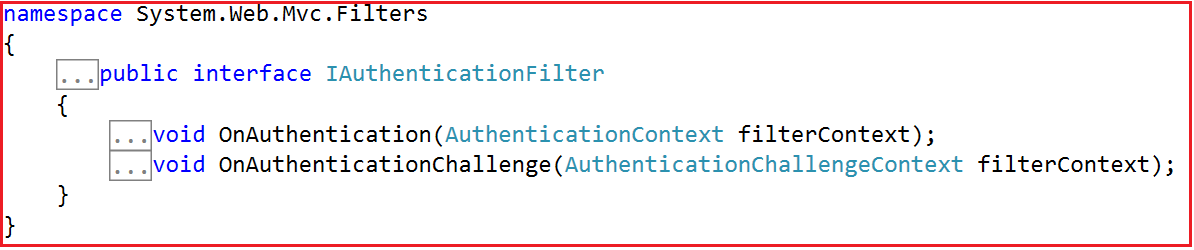
**What is the real reason behind Authentication filters?**

Before Authentication Filter, as a developer, we use the Authorization filters for two purposes i.e. Authentication and Authorization. It was convenient because the Authorization filters were executed before executing any other action filters.  For example, before executing the actual action method, we can use an Authorization filter to redirect an unauthenticated user to a login page or some error page.

But now, you can separate the Authentication related tasks to a new custom authentication filter and performs the authorization related tasks using the authorization filters only. So, in simple words, we can say it is basically about separating of concerns which will provide the developers to focus on one aspect using one filter only.

**How to create a Custom Authentication Filter in MVC?**

In order to create a Custom Authentication filter in MVC, you need to create a class by implementing the **IAuthenticationFilter** Interface. This **IAuthenticationFilter** interface has 2 methods. Following is the class definition of the IAuthenticationFilter interface.



**OnAuthentication:**

This method is used to authenticate the request. The AuthenticationContext provides us the necessary information which is required for performing authentication. We can use this information to make authentication decisions based on the current context. For example, we may decide to modify the ActionResult to different result types based on the authentication context, or we may decide to change the current principal based on the authentication context, etc.

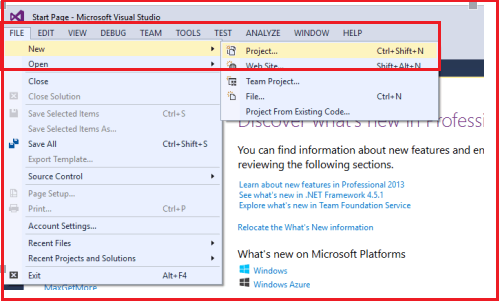
**OnAuthenticationChallenge:**

This method gets called when Authentication or Authorization is failed or after the execution of the action method but before rendering the view. That means the OnAuthenticationChallenge method can run at various stages.

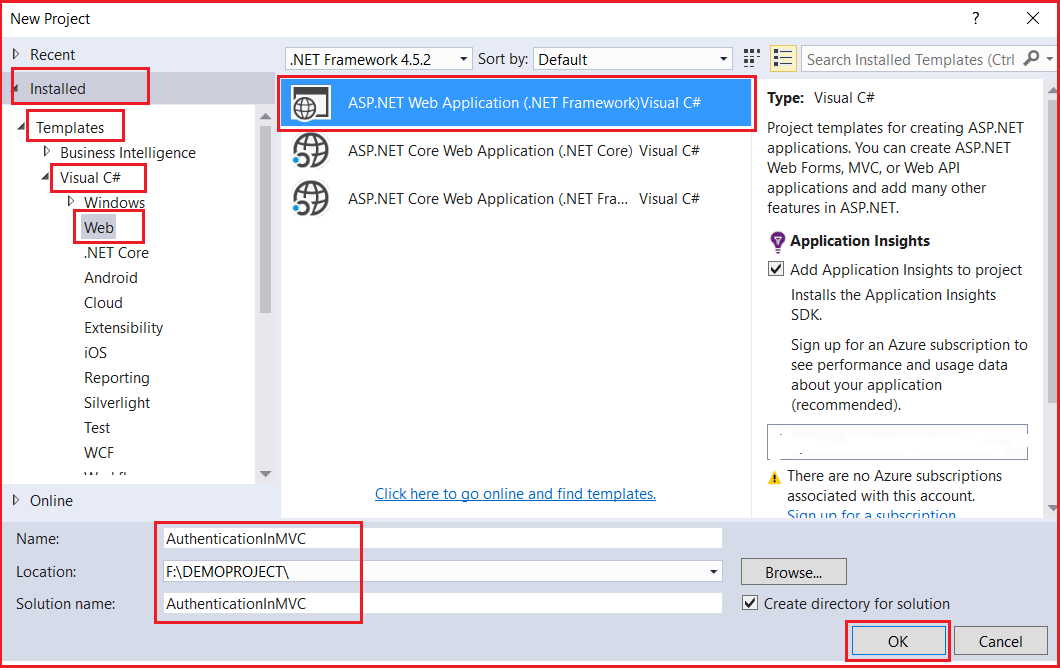
For example, it can run after the AuthorizationFilters or it can run after Action Method execution completed and so on. Since this method runs at various stages, you now have the ability to change the action result based on the authentication.

**Create a new Empty MVC Application**

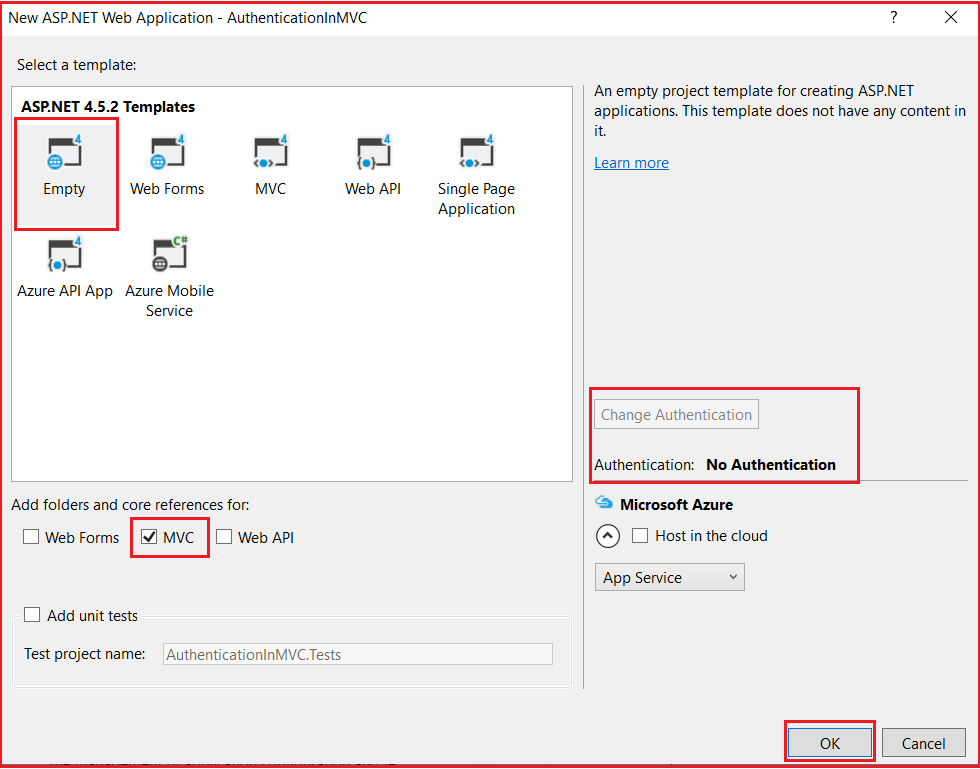
Open visual studio and select **File => New => Project**from the context menu as shown in the below image.



After clicking on the “**Project**” link, a new dialog will pop up. In that, we are going to select “**Web**” templates from the left pane. From the middle pane, select “**ASP.NET Web Application**“. Provide a meaningful name to the project here I am giving my project name as “**AuthenticationInMVC**”. Finally, click on the “**OK**” button as shown in the below image



Once you click on the “**OK”** button a new dialog will pop up with the name “**New ASP.NET Project**” for selecting project Templates as shown in the below image.



In this dialog, we are going to choose the “**Empty”** and “**MVC”** project template with the Authentication type as “**No Authentication”** and then click on the “**OK**” button. Once you click on the OK button it will take some time to create the project for us.

**Creating Login Model:**

Create a class file with the name LoginModel within the Models folder and then copy and paste the following code in it.

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *AuthenticationInMVC.Models*

**{**

**public** **class** LoginModel

**{**

**[**Required**(**ErrorMessage = "Please Enter User Name"**)]**

**[**Display**(**Name = "User Name"**)]**

**public** string UserName **{** **get**; **set**; **}**

**[**Required**(**ErrorMessage = "Please Enter Password"**)]**

**[**DataType**(**DataType.Password**)]**

**public** string Password **{** **get**; **set**; **}**

**}**

**}**

We created the above LoginModel class two properties i.e. Username and Password.

**Creating Accounnt Controller:**

Create a Controller with the name AccountController (Empty MVC5 Controller) within the Controllers Folder and then copy and paste the following code in it.

**using** *System.Web.Mvc;*

**using** *AuthenticationInMVC.Models;*

**using** *System.Web.Security;*

**namespace** *AuthenticationInMVC.Controllers*

**{**

**public** **class** AccountController : Controller

**{**

**[**HttpGet**]**

**public** ActionResult Login**()**

**{**

**return** View**()**;

**}**

**[**HttpPost**]**

**public** ActionResult Login**(**LoginModel model**)**

**{**

**if(**ModelState.IsValid**)**

**{**

//Here we are checking the values with hardcoded admin and admin

//You can check these values from a database

**if** **(**model.UserName.ToLower**()** == "admin" && model.Password == "admin"**)**

**{**

//Store the Username in session

Session**[**"UserName"**]** = model.UserName;

//Then redirect to the Index Action method of Home Controller

**return** RedirectToAction**(**"Index", "Home"**)**;

**}**

**else**

**{**

ModelState.AddModelError**(**"", "Invalid User Name or Password"**)**;

**return** View**(**model**)**;

**}**

**}**

**else**

**{**

**return** View**(**model**)**;

**}**

**}**

**}**

**}**

As you can see, we created the Account Controller with two Action Methods, one of which handles the HttpGet Request while the other one handles the HttpPost Request.

In Post Request, first, we are checking the ModelState is valid or not. If the model state is not valid then we are simply returning the model to the View which will display the model error and allows the user to submit the credentials again.

If it is valid then we are going to check the Username and Password is valid or not. Here to make things simple we are checking the credentials against the hardcoded values but in real-time, you need to check these credentials from a database or any persistent storage.

If the user name and password are not valid then we will stay on the same login page by providing the error message that Invalid User Name or Password. On the other hand, if the credentials are valid then we set Session[“UserName”] value from the UserName property of the Model object and then redirect the user to the Index View of Home Controller.

**Creating Login View:**

Create the Login View of Account Controller and then copy and paste the following code in it.

@model AuthenticationInMVC.Models.LoginModel

@{

ViewBag.Title = "Login";

}

**<h2>**Login**</h2>**

@using (Html.BeginForm())

{

@Html.AntiForgeryToken()

**<div** class="form-horizontal"**>**

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

**<div** class="form-group"**>**

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

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

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

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

**</div>**

**</div>**

**<div** class="form-group"**>**

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

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

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

@Html.ValidationMessageFor(model => model.Password, "", 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-default" **/>**

**</div>**

**</div>**

**</div>**

}

**<div>**

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

**</div>**

**<script** src="~/Scripts/jquery-1.10.2.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.min.js"**></script>**

**<script** src="~/Scripts/jquery.validate.unobtrusive.min.js"**></script>**

**Creating a Custom Authentication Filter in MVC:**

Create a class file with the name **CustomAuthenticationFilter.cs** within the Models folder and then copy and paste the following code in it. The class is inherited from **ActionFilterAttribute and IAuthenticationFilter** interface. We want to use this **CustomAuthenticationFilter** class as an attribute, so we inherit this class from **ActionFilterAttribute** class. Along the same line, we want this class an Authentication Filter so inherit from the **IAuthenticationFilter** interface.

**using** *System;*

**using** *System.Web.Mvc;*

**using** *System.Web.Mvc.Filters;*

**using** *System.Web.Routing;*

**namespace** *AuthenticationInMVC.Models*

**{**

**public** **class** CustomAuthenticationFilter : ActionFilterAttribute, IAuthenticationFilter

**{**

**public** **void** OnAuthentication**(**AuthenticationContext filterContext**)**

**{**

**if** **(**string.IsNullOrEmpty**(**Convert.ToString**(**filterContext.HttpContext.Session**[**"UserName"**])))**

**{**

filterContext.Result = new HttpUnauthorizedResult**()**;

**}**

**}**

**public** **void** OnAuthenticationChallenge**(**AuthenticationChallengeContext filterContext**)**

**{**

**if** **(**filterContext.Result == **null** || filterContext.Result **is** HttpUnauthorizedResult**)**

**{**

//Redirecting the user to the Login View of Account Controller

filterContext.Result = new RedirectToRouteResult**(**

new RouteValueDictionary

**{**

**{** "controller", "Account" **}**,

**{** "action", "Login" **}**

**})**;

//If you want to redirect to some error view, use below code

//filterContext.Result = new ViewResult()

//{

// ViewName = "Login"

//};

**}**

**}**

**}**

**}**

The **OnAuthentication** method is going to run before the **OnAuthenticationChallenge** method. Within the **OnAuthentication** method, we are just checking the **Session[“UserName”]** value is null or empty. If the **Session[“UserName”]** value is NULL or Empty then we are going to set the Result of the filterContext object to **HttpUnauthorizedResult**.

The **OnAuthenticationChallenge** method gets called when Authentication or Authorization is failed and also this method is called after the Execution of Action Method but before rendering the View. Here, within the **OnAuthenticationChallenge** method, we are just checking the Result value of the filterContext object. If the Result of the filterContext object is either null or HttpUnauthorizedResult then we just redirecting the request to the Login view of Account Controller.

**Using Custom Authentication Filter in MVC:**

In order to test the Custom Authentication Filter in MVC, Let’s add a Controller with the name Home within the Controllers folder. Once you create the Home Controller then copy and paste the following code in it.

**using** *System.Web.Mvc;*

**using** *AuthenticationInMVC.Models;*

**namespace** *AuthenticationInMVC.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

**[**CustomAuthenticationFilter**]**

**public** ActionResult Index**()**

**{**

**return** View**()**;

**}**

**[**CustomAuthenticationFilter**]**

**public** ActionResult Contact**()**

**{**

**return** View**()**;

**}**

**public** ActionResult About**()**

**{**

**return** View**()**;

**}**

**}**

**}**

As you can see, we created the above Home Controller class with three action methods. Two action methods i.e. Index and Contact are decorated with the custom Authentication Filter attribute. Now let’s create the respective views.

**Index.cshtml**

@{

ViewBag.Title = "Index";

}

**<h2>**Index View From Home Controller**</h2>**

**Contact.cshtml**

@{

ViewBag.Title = "Contact";

}

**<h2>**Contact View From Home Controller**</h2>**

**About.cshtml**

@{

ViewBag.Title = "About";

}

**<h2>**About View From Home Controller**</h2>**

That’s it. We are done with our implementation. Now run the application and you will see when you want to access the Index and Contact page, it will navigate to the Login page of Account Controller. But you can access the About page as we have not applied our custom authentication filter on this action method. Now login with the proper credentials and once you login then you can access each page of the application.