**Working With Filters In ASP.NET Core MVC**

Introduction

Filters allow us to run custom code before or after executing the action method. They provide ways to do common repetitive tasks on our action method. The filters are invoked on certain stages in the request processing pipeline.

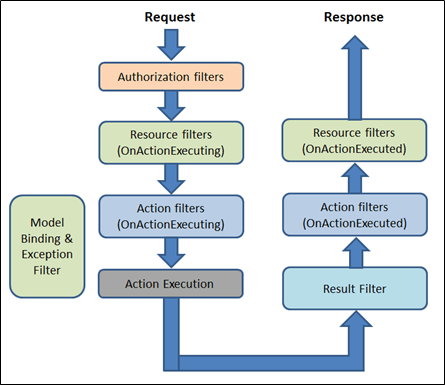
There are many built-in filters available with ASP.NET Core MVC, and we can create custom filters as well. Filters help us to remove duplicate codes in our application.

Filter Types in ASP.NET Core

Every filter type is executed at a different stage in the filter pipeline. Following are the filter types.

* *Authorization filters*  
  The Authorization filters are executed first. This filter helps us to determine whether the user is authorized for the current request. It can short-circuit a pipeline if a user is unauthorized for the current request. We can also create custom authorization filter.
* *Resource filters*  
  The Resource filters handle the request after authorization. It can run the code before and after the rest of the filter is executed. This executes before the model binding happens. It can be used to implement caching.
* *Action filters*  
  The Action filters run the code immediately before and after the controller action method is called. It can be used to perform any action before or after execution of the controller action method. We can also manipulate the arguments passed into an action.
* *Exception filters*  
  The Exception filters are used to handle exception that occurred before anything written to the response body.
* *Result filters*  
  The Result filters are used to run code before or after the execution of controller action results. They are executed only if the controller action method has been executed successfully.

Following diagram shows how these filters interact in filter pipeline during request and response life cycle.



Filter supports two types of implementation: synchronous and asynchronous; Both the implementations use different interface definitions.

The Synchronous filters run the code before and after their pipeline stage defines OnStageExecuting and OnStageExecuted. For example, ActionFilter. The OnActionExecuting method is called before the action method and OnActionExecuted method is called after the action method.

**Synchronous Filter Example**

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class CustomActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

Asynchronous filters are defined with only single method: OnStageExecutionAsync, that takes a FilterTypeExecutingContext and FilterTypeExecutionDelegate as The FilterTypeExecutionDelegate execute the filter’s pipeline stage. For example, ActionFilter ActionExecutionDelegate calls the action method and we can write the code before and after we call action method.

**Asynchronous filter example**

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class CustomAsyncActionFilter : IAsyncActionFilter

{

public async Task OnActionExecutionAsync(ActionExecutingContext context,

ActionExecutionDelegate next)

{

//To do : before the action executes

await next();

//To do : after the action executes

}

}

}

C#

Copy

We can implement interfaces for multiple filter types (stage) in single class. We can either implement synchronous or the async version of a filter interface, not both. The .net framework checks first for async filter interface, if it finds it, it called. If it is not found it calls the synchronous interface's method(s). If we implement both, synchronous interface is never called.

Adding Filter scope and Order of execution

A filter can be added to the pipeline at one of three scopes: by action method, by controller class or globally (which be applied to all the controller and actions). We need to register filters in to the MvcOption.Filters collection within ConfigureServices method.

public void ConfigureServices(IServiceCollection services)

{

// Add framework services.

services.AddMvc(options=> {

//an instant

options.Filters.Add(new CustomActionFilter());

//By the type

options.Filters.Add(typeof(CustomActionFilter));

});

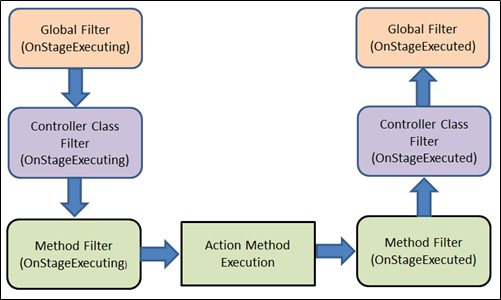
}

C#

Copy

When multiple filters are applied to the particular stage of the pipeline, scope of filter defines the default order of the filter execution. The global filter is applied first, then class level filter is applied and finally method level filter is applied.

Following figure shows the default order of filter execution.



Overriding the default order

We can override the default sequence of filter execution by using implementing interface IOrderedFilter. This interface has property named "Order" that use to determine the order of execution. The filter with lower order value execute before the filter with higher order value. We can setup the order property using the constructor parameter.

**ExampleFilter.cs**

using System;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class ExampleFilterAttribute : Attribute, IActionFilter, IOrderedFilter

{

public int Order { get; set; }

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

**Controller.cs**

using System;

using Microsoft.AspNetCore.Mvc;

using Filters;

namespace Filters.Controllers

{

[ExampleFilter(Order = 1)]

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

}

}

C#

Copy

When filters are run in pipeline, filters are sorted first by order and then scope. All built-in filters are implemented by IOrderFilter and set the default filter order to 0.

Cancellation or short circuiting filters

We can short circuit the filter pipeline at any point of time by setting the "Result" property of the "Context" parameter provided to the filter's methods.

**Filter Example**

using System;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class Example1FilterAttribute : Attribute, IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

context.Result = new ContentResult()

{

Content = "Short circuit filter"

};

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

Filters and DI (Dependency Injection)

As we learned, the filter can be added by the type or by the instance. If we added filter as an instance, this instance will be used for every request and if we add filter as a type, instance of the type will be created for each request. Filter has constructor dependencies that will be provided by the DI.

The filters that are implemented as attributes and added directly to the controller or action methods, cannot have constructor dependencies provided by the DI. In this case, contractor parameter must be supplied when they are applied.

This is a limitation of attribute. There are many way to overcome this limitation. We can apply our filter to the controller class or action method using one of the following,

* ServiceFilterAttribute
* TypeFilterAttribute
* IFilterFactory implemented on attribute

**ServiceFilterAttribute**

A ServiceFilter retrieves an instance of the filter from dependency injection (DI). We need to add this filter to the container in ConfigureService and reference it in a ServiceFilter attribute in the controller class or action method.

One of the dependencies we might require to get from the DI, is a logger. Within filter, we might need to log something happened.

Example is action filter with logger dependency,

using Microsoft.AspNetCore.Mvc.Filters;

using Microsoft.Extensions.Logging;

namespace FiltersSample.Filters

{

public class ExampleFilterWithDI : IActionFilter

{

private ILogger \_logger;

public ExampleFilterWithDI(ILoggerFactory loggerFactory)

{

\_logger = loggerFactory.CreateLogger<ExampleFilterWithDI>();

}

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

\_logger.LogInformation("OnActionExecuting");

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

\_logger.LogInformation("OnActionExecuted");

}

}

}

C#

Copy

Register filter in ConfigureService method

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<ExampleFilterWithDI>();

}

C#

Copy

Use filter for Action method of Controller class

[ServiceFilter(typeof(ExampleFilterWithDI))]

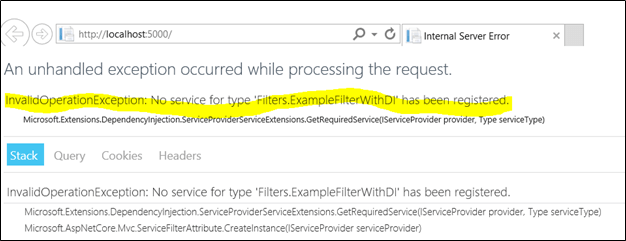
public IActionResult Index()

{

return View();

}

If we are not registering the filter type in ConfigureService method, system will throw an exception – “InvalidOperationException”.



The ServiceFilterAttribute implements IFilterFactory that exposes a method for creating an IFilter instance. This "CreateInstance" method use to load the specific type of DI from the services container.

**TypeFilterAttribute**

It is very similar to ServiceFilterAttribute and also implemented from IFilterFactory interface. Here, type is not resolved directly from the DI container but it instantiates the type using class "Microsoft.Extensions.DependencyInjection.ObjectFactory".

Due to this difference, the types are referenced in TypeFilterAttribute need to be register first in ConfigureService method. The "TypeFilterAttribute" can be optionally accept constructor arguments for the type. Following example demonstrates how to pass arguments to a type using TypeFilterAttribute.

[TypeFilter(typeof(ExampleFilterAttribute), Arguments = new object[] {"Argument if any" })]

public IActionResult About()

{

return View();

}

Summary

The filters allow us to run code before or after certain stages in the request processing pipeline. In this article, we learned type of built-in filter, filter scope and ordering, cancel the request from filter and how to inject the dependency in filters.

Filters In ASP.NET MVC

This article will tell you almost everything about filters used on action methods as well as on controllers in ASP.NET MVC. I am writing this article to tell you the basic to advanced foremost concepts about filters.

Last week one of my friends asked the question “How can I restrict the users to use my resources on the website?” I am dedicating this article to him. I hope he will like this.

The topics to be covered are.

1. What are the Filters?
2. Types of filters
   * Authorization filters
   * Action filters
   * Result filters
   * Exception filters
3. Order of Filters
4. Create a custom Filter in ASP.NET MVC.
5. Filters provided in ASP.NET MVC

The question is how to restrict the user to stop the usage of website resources. So this can be done using Filters in ASP.NET MVC.

Let’s get started with Filters.

Introduction and Background

Filters are used to inject extra processing logic in the MVC request-response pipeline. You have seen in my [previous article](https://www.c-sharpcorner.com/article/mvc-architecture-its-pipeline4/) in the section of “Pipeline in MVC”, that the HTTP request comes to “Routing” then goes to “Controller Initialization” then “Action Execution” and the “Result Execution” comes into action to render the view.

Now, if you want to inject some extra processing logic in between this request-response life cycle you have to use Filters or custom filters.

What are the filters?

Action filters are the attributes that can be applied on action method level, on controller level or application level. When these filters are applied on the controller level then they will apply to all the actions within the controller.

Filters allow us to add pre-processing and post-processing logic to an action or controller. From this filter, the flow of the application can be changed. You can put this filter by decorating the attribute above the action method or controller. This will call the class of that attribute in which the whole logic is written.

Types of filters

There are the following types of action filters.

|  |  |  |
| --- | --- | --- |
| Filter Type | Interface | Description |
| Authentication | IAuthenticationFilter | Always runs, before any action method or any other filter. |
| Authorization | IAuthorizationFilter | These run first, before the action method or any other filters. |
| Action | IActionFilter | These run before and after the action method. |
| Result | IResultFilter | Runs before and after the action result is executed. |
| Exception | IExceptionFilter | Runs only if another filter, the action method, or the action result throws an exception. |

Now let’s understand each filter with examples. (Examples are given in the “Filters provided in ASP.NET MVC” section)

Authorization filters

These filters are useful for implementing the logic of authentication and authorization. This will allow us to execute the action method when we want. This filter implements the IAuthorizationFilter. Examples of authorization filters are AuthorizeAttribute and RequireHttpsAttribute.

Action filters

This filter contains the pre-processing and post-processing logic which can be applied to the action method of the controller or to the entire controller. This filter is executed before and after the action method of the controller executes.

This filter implements the IActionFilter interface. This interface contains two methods, OnActionExecuting and OnActionExecuted which will be called before and after the action method gets executed respectively. This filter allows us to add some additional processing, for example, modifying the view data that an action method of the controller returns or canceling the execution of the action method of the controller.

Result filters

To use this filter, we have to implement the IResultFilter interface, and this interface contains two methods, OnResultExecuting and OnResultExecuted which will run before and after a view result is executed respectively. This filter allows us to modify the HTTP response, which means modifying the view result before the view is rendered to the browser. The OutputCacheAttribute class is an example of a result filter.

Exception Filter

The filter will execute when a controller or action method of the controller throws an unhandled exception. These filters implement the IExceptionFilter interface and can be used to log errors or to display the specific error page. The HandleErrorAttribute class is an example of the exception filter.

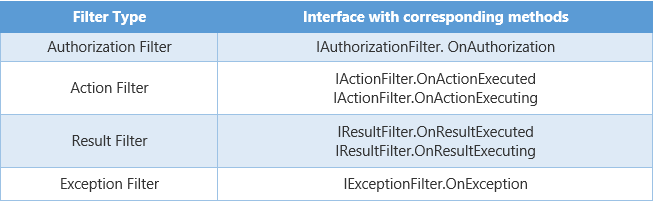
**Note.** The base class for all the action filters is the System.Web.Mvc.FilterAttribute class. Now, if you want to make your own custom type of filter, then you have to create a class that inherits the base filter class and then you have to implement one or more interfaces of the above-explained filters. See the figure below, in which the interfaces of each filter are shown.

|  |  |
| --- | --- |
| Filter Type | Interface |
| Authentication Filter | IAuthenticationFilter |
| Authorization Filter | IAuthorizationFilter |
| Action Filter | IActionFilter |
| Result Filter | IResultFilter |
| Exception Filter | IExceptionFilter |

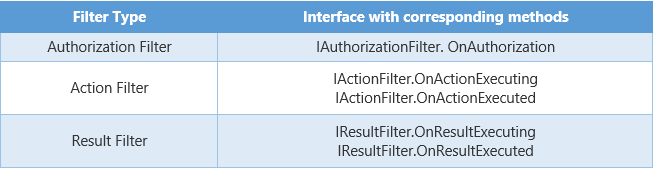
We’ll see how to create the custom filter later in this article. First, look at the order of these filters in which they will execute.

Order of filters

You have read the above five filters and their corresponding interfaces with their methods. The list is as follows.



Now, suppose that we are using all the above filters on one action method, then what will be the order of its execution? So the order will be as follows.

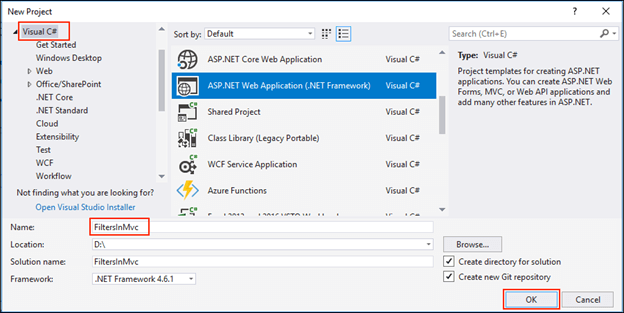


As we know if the exception occurred in the application then the application returns an exception instead of returning the result view. So for the Exception filter, if the exception has occurred then the IExceptionFilter.OnException will be called instead of Result Filters (IResultFilter.OnResultExecuting and IResultFilter.OnResultExecuted).

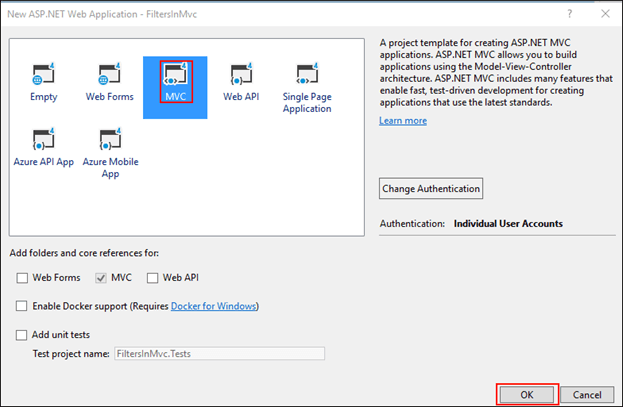
Custom Filter in ASP.NET MVC

Now, let’s create the custom authentication filter which will redirect the user back to the login page if the user is not authenticated.

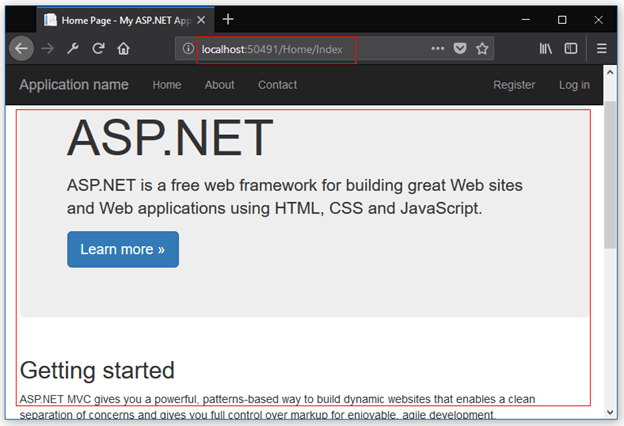
Create the project in the way shown below,



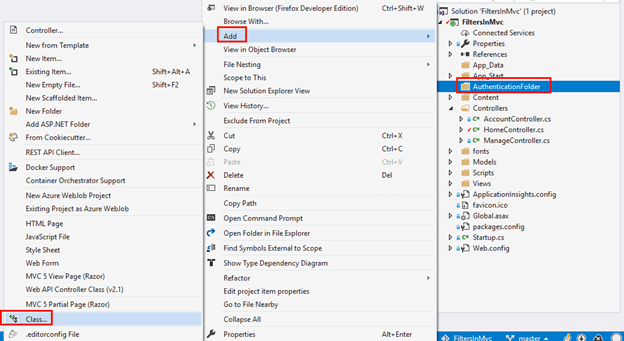
Select MVC then click OK.



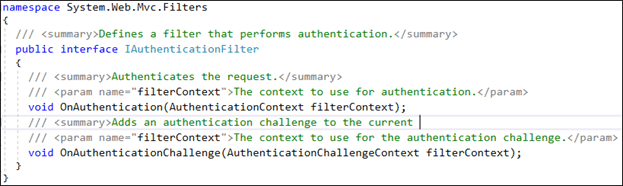
Build and run your application simply and you will prompt the following page on the browser which will call from the Index action method of the HomeController. You can see this page is not prompted to the login page. But when we make an authentication filter and apply it on HomeController then we’ll redirect to the login page.



Now, make a folder with the name of AuthenticationFolder in the project.



Then make a class with the name of AuthAttribute and inherit it from IAuthenticationFilter and IActionFilter. You can see below the look and feel of the IAuthenticationFilter interface in which two methods OnAuthentication and OnAuthenticationChallenge are declared. The purpose of OnAuthentication is to be executed first and to perform the needed authentication logic.



The purpose of OnAuthenticationChallenge is to restrict access of the user if the user is not authenticated. So in the AuthAttribute class, you have to implement these methods. The logic to authenticate a user and to restrict user’s access is written in the following code.

using System.Web.Mvc;

using System.Web.Mvc.Filters;

namespace FiltersInMvc.AuthenticationFolder

{

public class AuthAttribute : ActionFilterAttribute, IAuthenticationFilter

{

private bool \_auth;

public void OnAuthentication(AuthenticationContext filterContext)

{

// Logic for making a user authenticate

\_auth = (filterContext.ActionDescriptor.GetCustomAttributes(

typeof(OverrideAuthenticationAttribute), true).Length == 0);

}

public void OnAuthenticationChallenge(AuthenticationChallengeContext filterContext)

{

var user = filterContext.HttpContext.User;

if (user == null || !user.Identity.IsAuthenticated)

{

filterContext.Result = new HttpUnauthorizedResult();

}

}

}

}

C#

Copy

Now, we’ll apply this filter on the HomeController. So go to HomeController.cs class and decorate the controller with the Auth attribute as shown in the figure below.

using FiltersInMvc.AuthenticationFolder;

using System.Web.Mvc;

namespace FiltersInMvc.Controllers

{

public class HomeController : Controller

{

[Auth]

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();

}

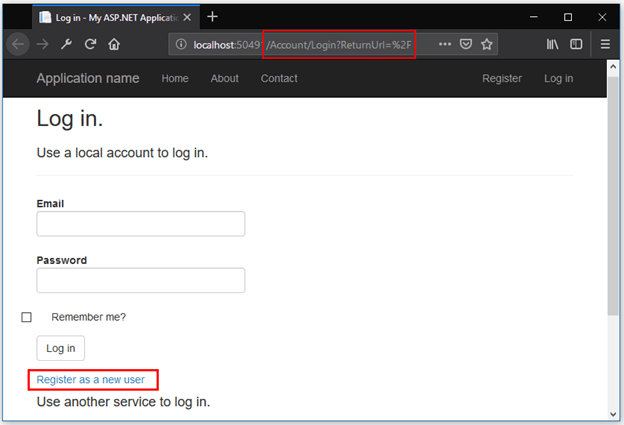
}

}

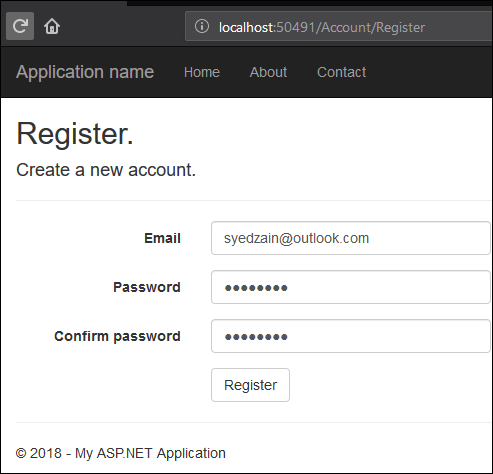
C#

Copy

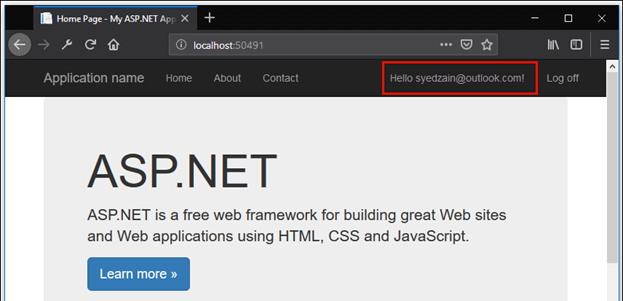
Now, simply build and run the application and you will be prompted to the login page instead of the home page, as shown in the figure below.



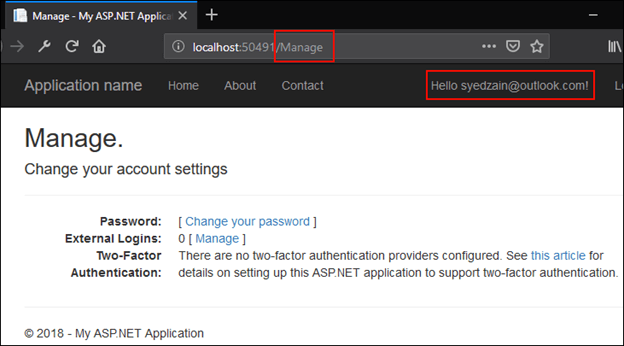
Now I’ll register myself with my supposed credentials.



And then I’ll log in with the same credentials.



When you click on your Name then you will be redirected to Manage Controller in which you can manage your account settings such as change password etc. See the figure below.



Authorization Filter

Now, I am going to apply an authorization filter in this sampled application. These filters are used to enforce an authorization policy in which the action method can only be invoked by approved or authenticated users. As you know, these filters are run before the action method is invoked and these implement the IAuthorizationFilter interface. The look and feel of the IAuthorizationFilter interface is as follows.

namespace System.Web.Mvc

{

/// <summary>Defines the methods that are required for an authorization filter.</summary>

public interface IAuthorizationFilter

{

/// <summary>Called when authorization is required.</summary>

/// <param name="filterContext">The filter context.</param>

void OnAuthorization(AuthorizationContext filterContext);

}

}

C#

Copy

Now, let’s make a class with the name of AuthZAttribute to apply the authorization policy. For authorization, you have to inherit the AuthAttribute class to AuthorizeAttribute. In this example, I’ll use the built-in feature of the Authorize attribute. For this, you have to override the AuthorizeCore (HttpContextBase httpContext) method from the AuthorizeAttribute class. The look and feel of the AuthorizeCore method are as follows.

/// <summary>

/// When overridden, provides an entry point for custom authorization checks.

/// </summary>

/// <param name="httpContext">The HTTP context, which encapsulates all HTTP-specific information about an individual HTTP request.</param>

/// <returns>True if the user is authorized; otherwise, false.</returns>

/// <exception cref="System.ArgumentNullException">The <paramref name="httpContext" /> parameter is null.</exception>

protected virtual bool AuthorizeCore(HttpContextBase httpContext);

C#

Copy

As you can see in the comments, the purpose to this method is to return true if the user is authorized; otherwise false. So it will provide us authorization logic and we’ll override it shortly. Let’s write some code in AuthZAttribute. The code is as follows.

using System.Web;

using System.Web.Mvc;

namespace FiltersInMvc.AuthenticationFolder

{

public class AuthZAttribute : AuthorizeAttribute

{

private readonly bool \_localReq;

public AuthZAttribute(bool reqReq)

{

\_localReq = reqReq;

}

protected override bool AuthorizeCore(HttpContextBase httpContext)

{

if (httpContext.Request.IsLocal)

{

return \_localReq;

}

else

{

return true;

}

}

}

}

You can see above we are using the constructor AuthZAttribute (bool allowed), which takes a bool value indicating whether the local requests are permitted by taking advantage of the built-in feature of the authorization attribute base class, I only want to show the basic logic of authorization. Now it’s time to implement this attribute to action methods of the controller. Open the HomeController and decorate the Index action method with this Authorize Attribute. The code is given below.

using System.Web.Mvc;

namespace FiltersInMvc.Controllers

{

public class HomeController : Controller

{

[Authorize]

public ActionResult Index()

{

return View();

}

public ActionResult About()

{

ViewBag.Message = "This is the article by Zain.";

return View();

}

public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

}

}

Now you can simply build and then run the application and you will see the following output in the browser. You will prompt to the login page because you are not an authenticated user. It is because of the Authorize attribute.

Authorization Filter In ASP.NET MVC

* [A person in a grey shirt

  Description automatically generated](https://www.c-sharpcorner.com/members/farhan-ahmed24)

* [Farhan Ahmed](https://www.c-sharpcorner.com/members/farhan-ahmed24)

* Jul 13, 2019

* 157.8k

* 0

* 7

Introduction

In ASP.NET MVC, by default, all the action methods are accessible to both anonymous and authenticated users. But, if you want the action methods to be available only for authenticated and authorized users, then you need to use the AuthorizationFilter in MVC.

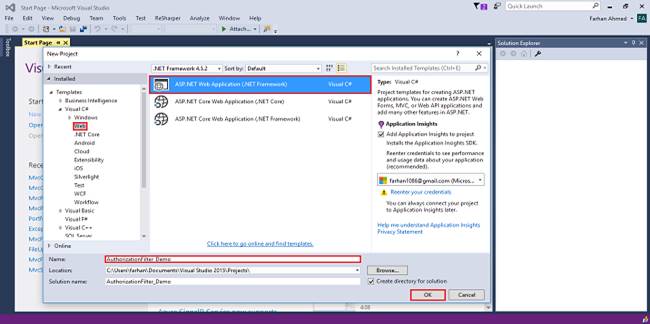
* [Authentication And Authorization In MVC](https://www.c-sharpcorner.com/article/authentication-and-authorization-in-mvc/)

**Step 1**

Open Visual Studio 2015 or your an editor of your choice and create a new project.

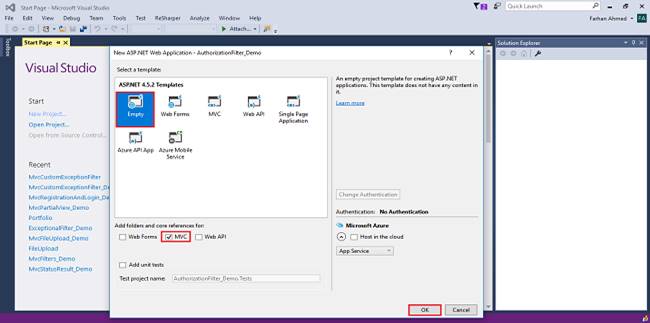
**Step 2**

Choose "web application" project and give an appropriate name to your project.



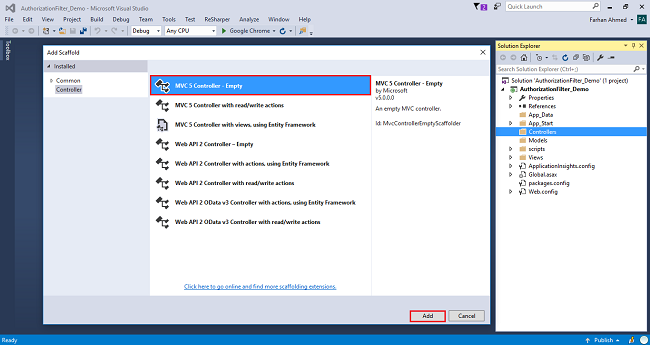
**Step 3**

Select "empty" template, check on MVC checkbox, and click OK.

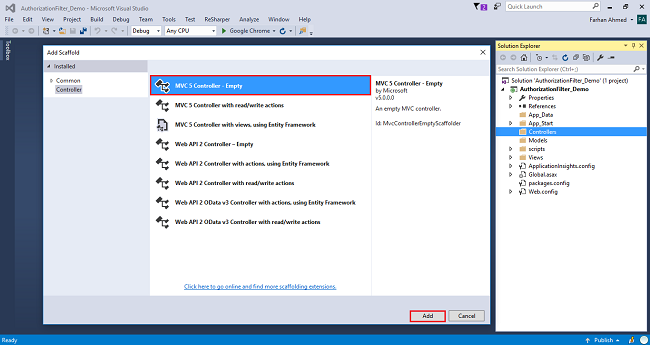


**Step 4**

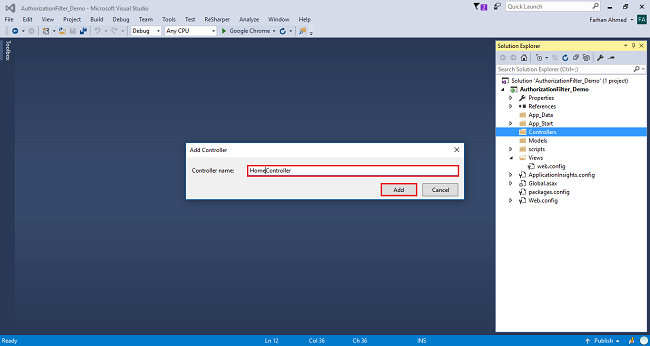
Right-click on the controllers folder and add a new controller.



A window will appear. Choose MVC5 Controller-Empty and click "Add".



After clicking on "Add", another window will appear with DefaultController. Change the name to HomeController and click "Add". The HomeController will be added under the *Controllers* folder. Don’t change the Controller suffix for all controllers, change only the highlight - instead of Default, just change Home.

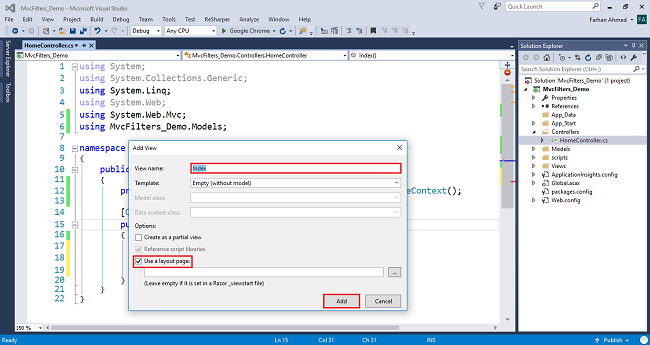


**Complete code for HomeController**

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Web;
5. **using** System.Web.Mvc;
7. **namespace** AuthorizationFilter\_Demo.Controllers
8. {
9. **public** **class** HomeController : Controller
10. {
12. **public** ActionResult Index()
13. {
14. **return** View();
15. }
17. [AllowAnonymous]
18. **public** ActionResult NonSecured()
19. {
20. **return** View();
21. }
22. [Authorize]
23. **public** ActionResult Secured()
24. {
25. **return** View();
26. }
28. **public** ActionResult Login()
29. {
30. **return** View();
31. }
32. }
33. }

**Step 5**

Right-click on Index method in HomeController. The "Add View" window will appear with default index name checked (use a Layout page), and click on "Add. Add three other views - NonSecured, Secured, and Login.



**NonSecured View**

Right-click on the NonSecured() and then, add a view with name NonSecured. Write the following code in NonSecure.cshtml view.

1. @{
2. ViewBag.Title = "NonSecured";
3. }
5. <h2>This method can be accessed by everyone as it is non-secure method</h2>

**Secured View**

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

1. @{
2. ViewBag.Title = "Secured";
3. }
5. <h2>This method needs to be accessed by authorized users as it Secure Method</h2>

**Login View**

Similarly, right-click on the Login() method and add the view with name Login.cshtml. Then, write the following code in Login.cshtml view.

1. @{
2. ViewBag.Title = "Login";
3. }

6. @using (Html.BeginForm())
7. {
8. <div **class**="login-form">
9. <h2 **class**="text-center">Log **in**</h2>
10. <div **class**="form-group">
11. <input type="text" **class**="form-control" placeholder="Username" required="required">
12. </div>
13. <div **class**="form-group">
14. <input type="password" **class**="form-control" placeholder="Password" required="required">
15. </div>
16. <div **class**="form-group">
17. <button type="submit" **class**="btn btn-primary btn-block">Log **in**</button>
18. </div>
19. <div **class**="clearfix">
20. <label **class**="pull-left checkbox-inline"><input type="checkbox"> Remember me</label>
21. <a href="#" **class**="pull-right">Forgot Password?</a>
22. </div>
23. <p **class**="text-center"><a href="#">Create an Account</a></p>
24. </div>
25. }

**Step 6**

Open web.config file and write the following code under the system.web section.

1. <authentication mode="Forms">
2. <forms loginUrl="/Home/Login"></forms>
3. </authentication>

**Step 7**

Build and run your project by pressing ctrl + F5.

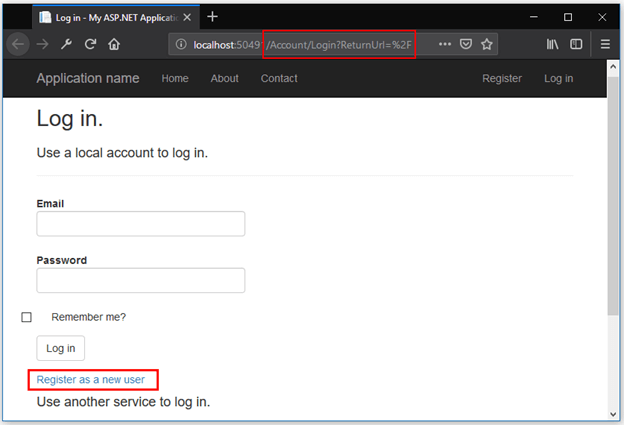
That’s it. Now, run the application and navigate to /Home/Secured. Then, you will see that it will redirect you to the Login method.

How to use AllowAnonymous in MVC?

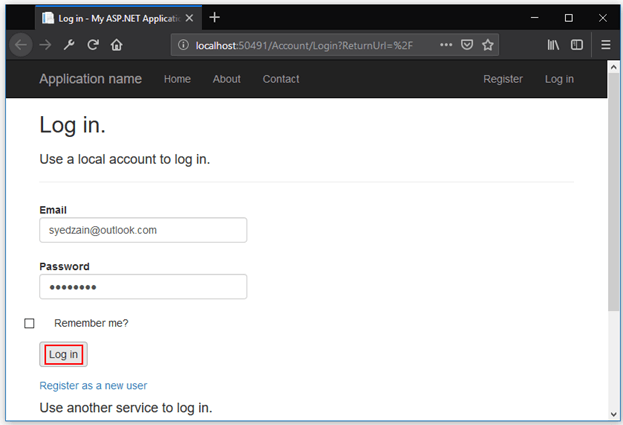
If you want to allow anonymous access to the NonSecured of Home controller, then you need to decorate the AllowAnonymous attribute to that NonSecuredmethod as shown below. The AllowAnonymous attribute in MVC is used to skip the authorization which is enforced by Authorization Filter in MVC.

1. [AllowAnonymous]
2. **public** ActionResult NonSecured()
3. {
4. **return** View();
5. }

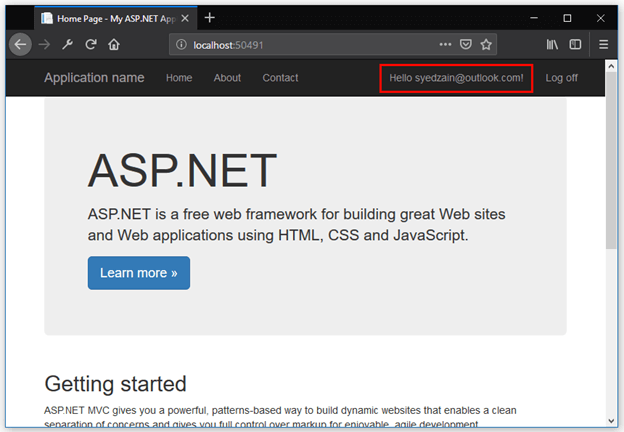
Now, run the application and navigate to /Home/NonSecured and you will see that it displays the page as expected and when you navigate to /Home/Secured, then it will redirect you to the Login page.



As I have registered myself with my credentials so there is no need to register again. I will log in with the credentials.

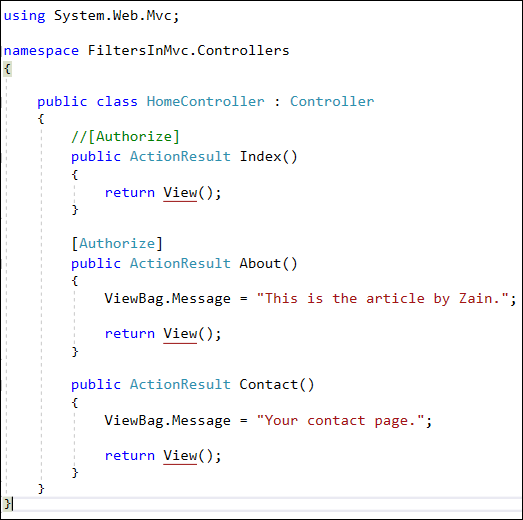


Now, you can see in the figure below, I am logged in.

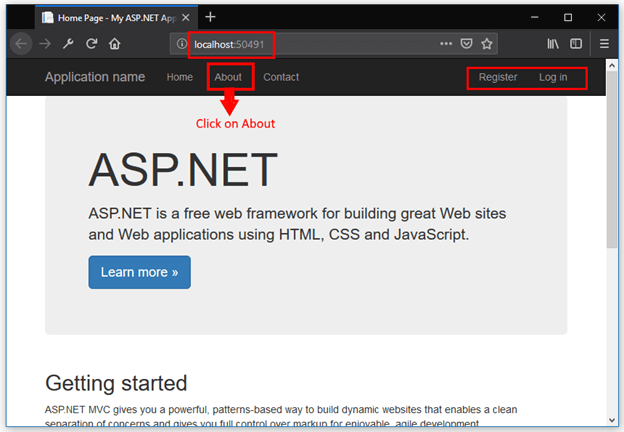


As you can see, I am logged in and can access the About and Contact page without any need to give login credentials.

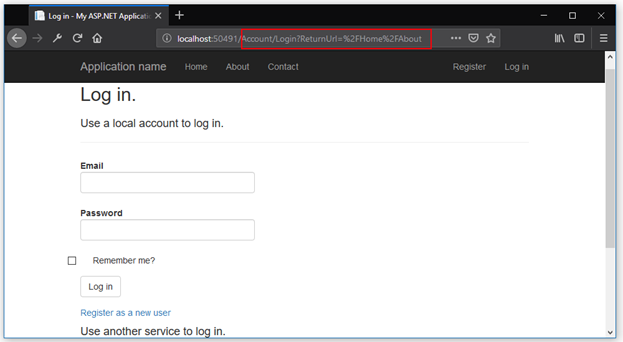
Now, put the Authorize attribute to the About action method, as shown below.



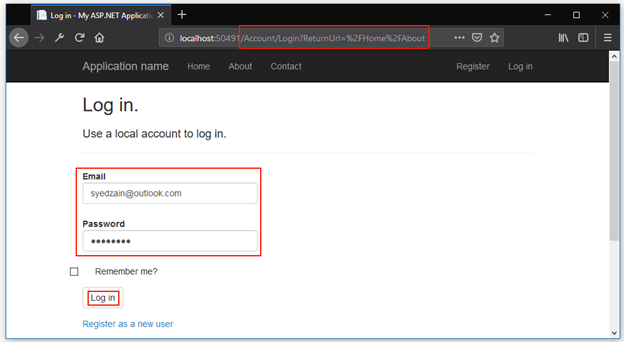
Now, build and run the application, and you’ll see the following output in the browser.



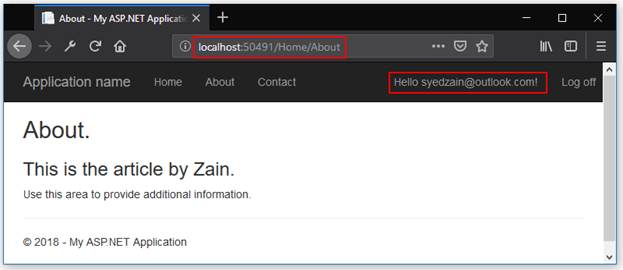
But when you click on About link, then the application will redirect your browser to log-in page. Because the About page can only be used by authenticated users. See the picture below.



Now, I’ll write my email and password to access the About page, as shown below.



You can see below, I am redirected to the About page successfully.



Hence you have seen the basic functionality of Authentication and Authorization. One more feature of authorization is, that you can specify specific users to access any action method, decorated with the Authorize attribute.

I am decorating the Contact action method with the Authorize attribute and specifying the user’s email as “zainshah@outlook.com” so my previous email would not work at all. Let’s see the code.

using System.Web.Mvc;

namespace FiltersInMvc.Controllers

{

public class HomeController : Controller

{

//[Authorize]

public ActionResult Index()

{

return View();

}

[Authorize(Users = "[syedzain@outlook.com](mailto:syedzain@outlook.com)")]

public ActionResult About()

{

ViewBag.Message = "This is the article by Zain.";

return View();

}

[Authorize(Users = "[zainshah@outlook.com](mailto:zainshah@outlook.com)")]

public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

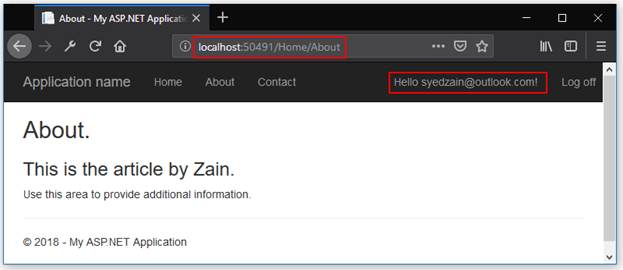
}

}

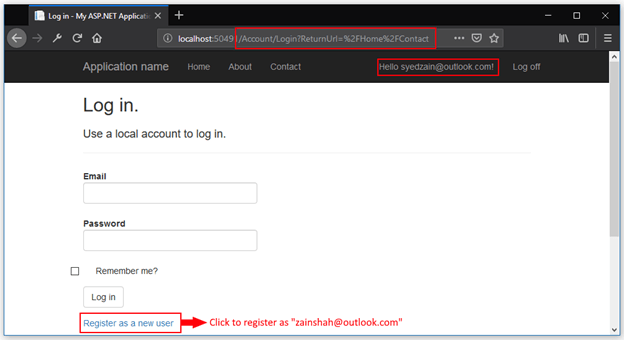
C#

Copy

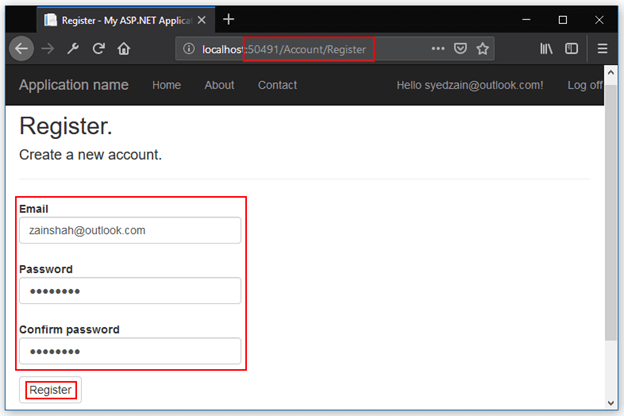
Now, build and run the application. You can see below, that my old credentials are logged in. I am able to access the About page, as shown below.



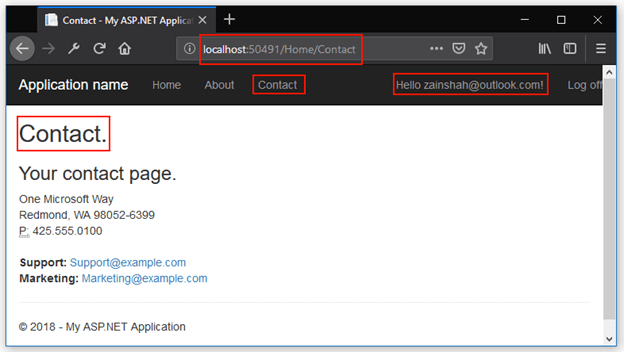
But when I click on the Contact link, I can’t access the Contact page, because I am not an authenticated user with the email “zainshah@outlook.com”. I will redirect to the login page, as shown below.



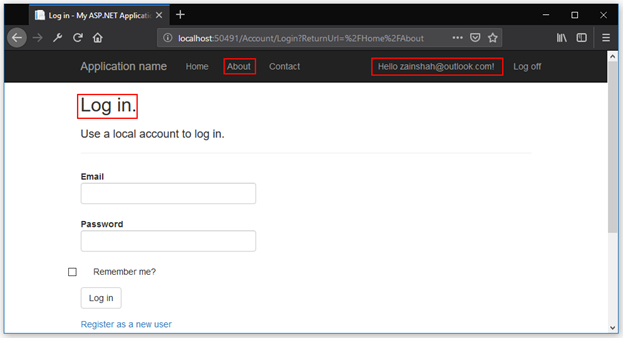
Now, I’ll register myself with a new email and password. The email should be “zainshah@outlook.com” to access the Contact page.



Now click on Register, and you will automatically be prompted to the index page. So click on the Contact page, and you’ll successfully access it, as shown below.



But now if you click on the About page, you should be prompted to log in page, because you are not authenticated with the email as “syedzain@outlook.com”. About can only be accessed with this email Id. You can see the login page below while logged in as “zainshah@outlook.com”.



Hence this is all about authentication and authorization. Now we’ll look at some filters provided in ASP.NET MVC, each with an example. So let’s move towards it.

Filters provided in ASP.NET MVC

ASP.NET MVC provides us with some filters that can be implemented as attributes. These are applicable at the level of action method, the controller, or the whole application.

1. AuthorizeAttribute
2. HandleErrorAttribute
3. OutputCacheAttribute
4. RequireHttpsAttribute

AuthorizeAttribute

This filter is used to restrict access by authentication and authorization. In other words, this is an attribute that can be used to restrict access to the action methods of the controller by callers.

**Syntax**

namespace System.Web.Mvc

{

/// <summary>Specifies that access to a controller or action method is restricted to users who meet the authorization requirement.</summary>

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

public class AuthorizeAttribute : FilterAttribute, IAuthorizationFilter

{

// Some code here.

}

}

C#

Copy

As you know, almost every website requires users to log in before using the restricted content of the website. But in some web applications, even when the user is logged in we want to restrict the users to view or modify some specific content. To accomplish this task, you have to use the AuthorizeAttribute class. So when you mark an action method with the AuthorizeAttribute then the access to that action method is restricted to users who are both authenticated and authorized. When the controller is decorated with this attribute then all the action methods of the controller are restricted.

When an unauthorized user tries to access the action method of the controller which is decorated with AuthorizeAttribute, then the user gets a 401 HTTP status code. But if ASP.NET forms authentication is set in the website then the 401 status code causes the browser to redirect the user to the login page.

**Example**

We have already seen the example of AuthorizeAttribute used in the authorization section above. Here you will just see the sampled example and its result (without proper images of outputs).

Below you can see the example in which there is a controller (HomeController) who has a total of 5 action methods in which three methods are decorated with an Authorize attribute and two methods are not decorated with an attribute.

public class HomeController : Controller

{

public ActionResult Index()

{

ViewData["Message"] = "This is Zain’s article!";

return View();

}

public ActionResult About()

{

return View();

}

[Authorize]

public ActionResult AuthenticatedUsers()

{

return View();

}

[Authorize(Roles = "Admin, Super User")]

public ActionResult AdministratorsOnly()

{

return View();

}

[Authorize(Users = "[Zain@outlook.com](mailto:Zain@outlook.com), [syed@outlook.com](mailto:syed@outlook.com)")]

public ActionResult SpecificUserOnly()

{

return View();

}

}

C#

Copy

The first two action methods, Index and About, can be accessed by anyone, even anonymous users. The AuthenticatedUsers action method can only be accessed by the users who are properly logged in. The AdministratorsOnly action method can only be accessed to those users who have either the role Admin or Super User. And the SpecificUser action method can only be accessed by those users who have either the name “zain@outlook.com” or “syed@outlook.com”.

HandleErrorAttribute

This filter is used to handle an exception thrown by an action method. Let’s move to see its syntax.

**Syntax**

namespace System.Web.Mvc

{

/// <summary>Represents an attribute that is used to handle an exception that is thrown by an action method.</summary>

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

public class HandleErrorAttribute : FilterAttribute, IExceptionFilter

{

// Some code here

}

}

C#

Copy

After decorating this attribute to an action method, when the action method throws an exception, MVC displays the error view, which is located in the ~/View/Shared folder. There are the following properties of HandleErrorAttribute, which are given below.

* **ExceptionType:**Specifies the type of exception that is handled by the filter. The return type of this property is Type, which means it returns the type of exception that occurred. We can modify its default behavior.
* **Master:**This property gets or sets the master view for displaying exception information. The return type of this property is a string so that it can display exception information.
* **TypeId:**This property is used to get the unique identifier for the HandleErrorAttribute. We can’t modify or set the value for this property.
* **View:**This property is used to get or set the page view for displaying exception information. We can set the view page name for this attribute. As we know when the action method throws an exception, MVC displays an Error view in a Shared folder located in the View folder. MVC framework sends information on exceptions in the ViewDictionary object. The Model property of this view is set to an instance of the ExceptionContext class. The ViewData dictionary contains the values for the keys.
  1. **ActionName:** The intended action method that throws an exception.
  2. **ControllerName:**The intended controller in which the action method throws an exception.
  3. **Exception:**The exception object was thrown by the action method of the controller.

**Example**

To use HandleErrorAttribute, let’s some changes in the project FiltersInMvc. Follow the steps given below.

**Step 1.**Add an action method with the name of CustomException in the HomeController.

public ActionResult CustomException()

{

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

}

C#

Copy

**Step 2.**Go to the Shared folder, then open \_Layout. cshtml and add the following line of code for the navigation bar.

<li>@Html.ActionLink("Custom Exception", "CustomException", "Home")</li>

Markup

Copy

You can see the code below.

The code of HomeController is as follows.

using System;

using System.Web.Mvc;

namespace FiltersInMvc.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

return View();

}

[Authorize(Users="[syedzain@outlook.com](mailto:syedzain@outlook.com)")]

public ActionResult About()

{

ViewBag.Message = "This is the article by Zain.";

return View();

}

[Authorize(Users="[zainshah@outlook.com](mailto:zainshah@outlook.com)")]

public ActionResult Contact()

{

ViewBag.Message = "Your contact page.";

return View();

}

public ActionResult CustomException()

{

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

}

}

}

C#

Copy

And you should add this piece of code into \_Layout.cshtml file in the Shared folder is as follows.

<li>@Html.ActionLink("Custom Exception", "CustomException", "Home")</li>

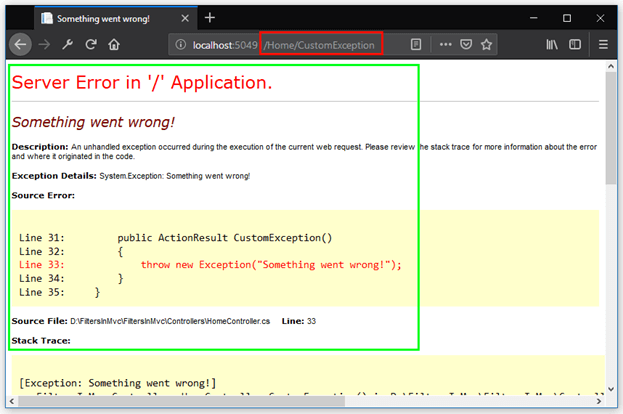
Markup

Copy

The output of the above line shows the Customer Exception in the navigation bar, as shown below.

Customer Exception

Now let’s build and run the application, and go to the /Home/CustomException URL, as shown below.



So when you call the CustomException action method, you can see the above output has an exception.

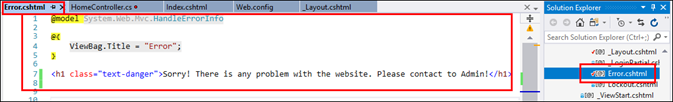
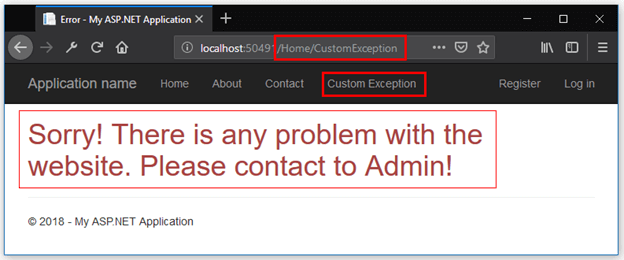
But the problem with the output here is, it has the code of the application, which can be helpful for a hacker. So we can display any friendly error page. This can be done in two ways,

1. Go to Web. config file of the root directory, then go to the system. web section and place the following element in that section.

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

XML

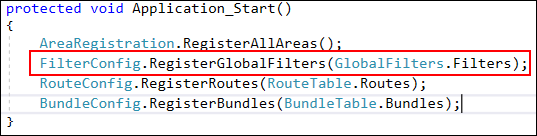
Copy

1. Place user-friendly text in the Error.cshtml file in Shared You can see below the screenshot of the code.  
   
2. Now simply build and run the application, go to /Home/CustomException URL then you should see the output given below.  
   

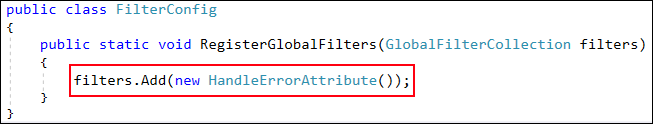
Question. I didn’t use the HandleError attribute yet, then how did all this work?

**Answer**

Look, this is all due to HandleError attribute which is already added to the GlobalFilters collection in global.asax file. When a filter is added to the GlobalFilters collection, then it is applicable to all the controllers and their action methods in the entire application. You can see the code below.

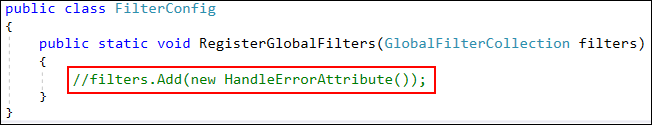


If we put the cursor on FilterConfig and select “Go to definition” option, then you can see the below code.

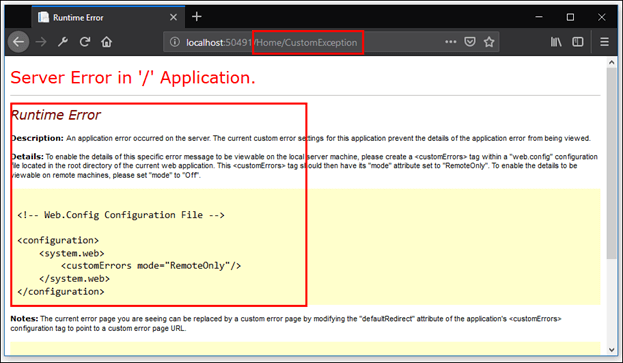


As you can see, HandleErrorAttribute is already added to the filters collection. So it is already applied to all the controllers and their action methods.

Now if we comment on this line of code, then what happened?



Now, if you build and run the application, and go to /Home/CustomException URL, you should see the following runtime error.



Now, if you want to again handle the error, then you have to decorate the action method with HandleErrorAttribute.

[HandleError]

public ActionResult CustomException()

{

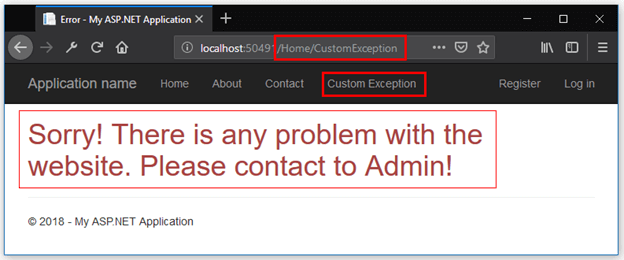
throw new Exception("Something went wrong!");

}

C#

Copy

Now you should see the following output in the browser.



Hence, you have seen the use of the HandleError attribute. You can apply it to the controller level if more than one action method of your controller has thrown an exception.

OutputCacheAttribute

This filter is used to decorate an action method of a controller whose output will be cached.

**Syntax**

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

public class OutputCacheAttribute : ActionFilterAttribute, IExceptionFilter

{

// Some code here

}

C#

Copy

OutputCacheAttribute is used to store the output of an action method in memory on the web server. For example, if the action method renders a view, that view page will be cached. This cached page will be available to the HTTP requests for some specified time. This can be helpful in saving the time and resources it would take to re-create the result of that action method. Now we are going to take a simple example in which the output of the Test action method of the HomeController will be cached for 10 seconds. There will be a tab for the Test action method in the navigation bar, so when you repeatedly click on the tab, you can see that the page stays cached for 10 seconds.

To use OutputCacheAttribute, let’s some changes in the project FiltersInMvc. Follow the steps given below.

**Step 1.**Add an action method with the name of Test in the HomeController.

[OutputCache(Duration = 10)]

public ActionResult Test()

{

ViewData["Message"] = "This page was cached at " + DateTime.Now;

return View();

}

C#

Copy

**Step 2.**Go to the Shared folder, then open \_Layout. cshtml and add the following line of code for the navigation bar.

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

C#

Copy

**Step 3.**Add the View with the name of Test. The code of the Test view is as follows.

@{

ViewBag.Title = "Output";

}

<h2>Output</h2>

<p>

@Html.Encode(ViewData["Message"])

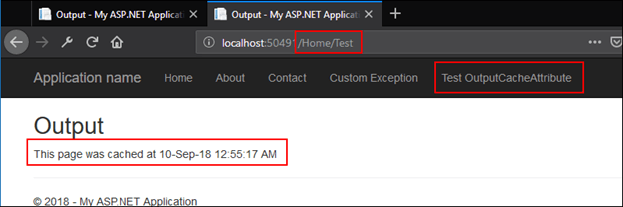
</p>

Markup

Copy

Output

Simply build and run the application. You should see the below output.



When you repeatedly click within 10 seconds, then you will not see any change in the time. But after 10 seconds you can see the change in time. Hence in his fashion, you can store the output of an action method for the specified time.

RequireHttpsAttribute

RequireHttpsAttribute is used to force an unsecured HTTP request to be re-sent over HTTPS.

To use RequireHttpsAttribute, let’s some changes in the project FiltersInMvc. Follow the steps given below:

**Step 1.**Add an action method with the name of Force in the HomeController.

public string Force()

{

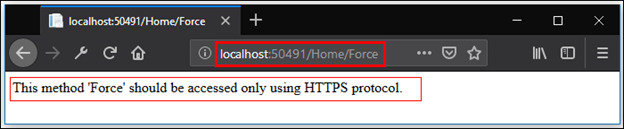
return @"This method 'Force' should be accessed only using HTTPS protocol.";

}

C#

Copy

When you build and run the application through this URL http://localhost:50491/Home/Force, you should see the following output.



But when we applied the RequireHttps attribute to the Force action method, as shown below in the code, then we couldn’t browse it with HTTP protocol. If we try to browse it with HTTP then it will be redirected to HTTPS URL.

[RequireHttps]

public string Force()

{

return @"This method 'Force' should be accessed only using HTTPS protocol.";

}

C#

Copy

Now, build and reload the browser, you can see the change in URL, and the protocol is changed from HTTP to HTTPS. So, the [RequireHttps] attribute, forces an HTTP request to be re-sent over HTTPS.

A screen shot of a computer

Description automatically generated

RequireHttps attribute can be applied on the controller level, in this case, it will apply to all the action methods in that controller.

Conclusion

I hope this article has helped you in understanding all the concepts about filters. If you have any query then feel free to contact me in the comments section. Also, giving feedback, either positive or negative, it will help me to make my articles better and increase my enthusiasm to share my knowledge.