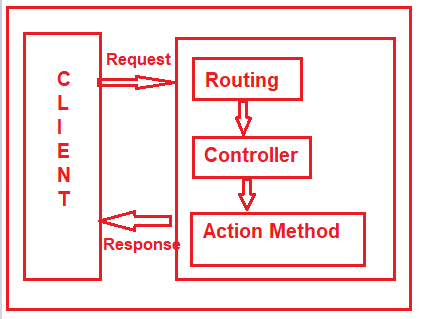
**Action filters in mvc**

What are action filters in asp.net mvc?

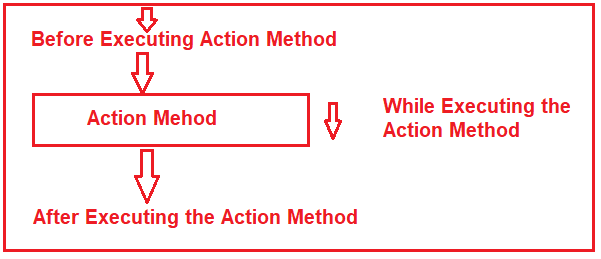
Action filters are attributes that can be applied either on a controller action method or on a controller. When applied at the controller level, they are applicable for all actions within that controller. Action filters allow us to add, pre and post processing logic to an action method. This means, they allow us to modify the way in which an action is executed.

##### **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

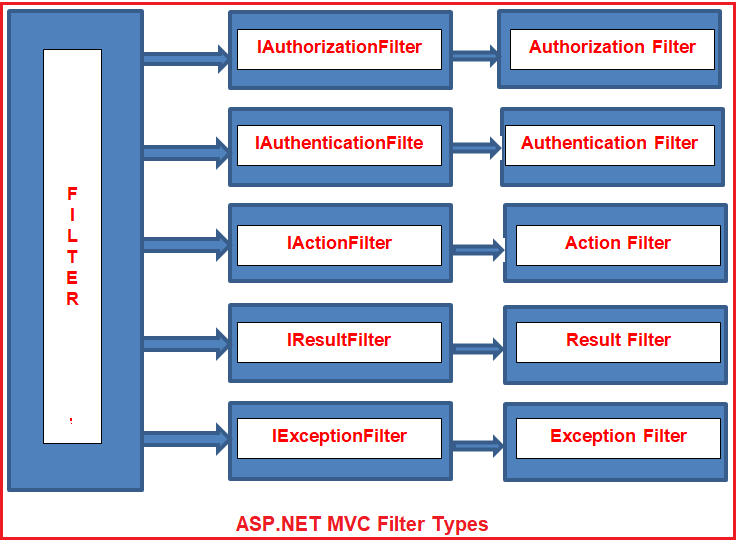
**return** View**()**;

**}**

**}**

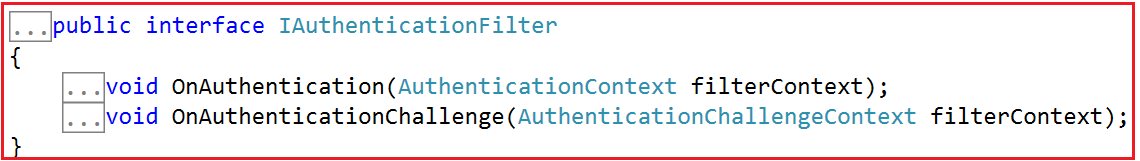
##### **Role and Responsibility of MVC Filters:**

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-



##### **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.

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

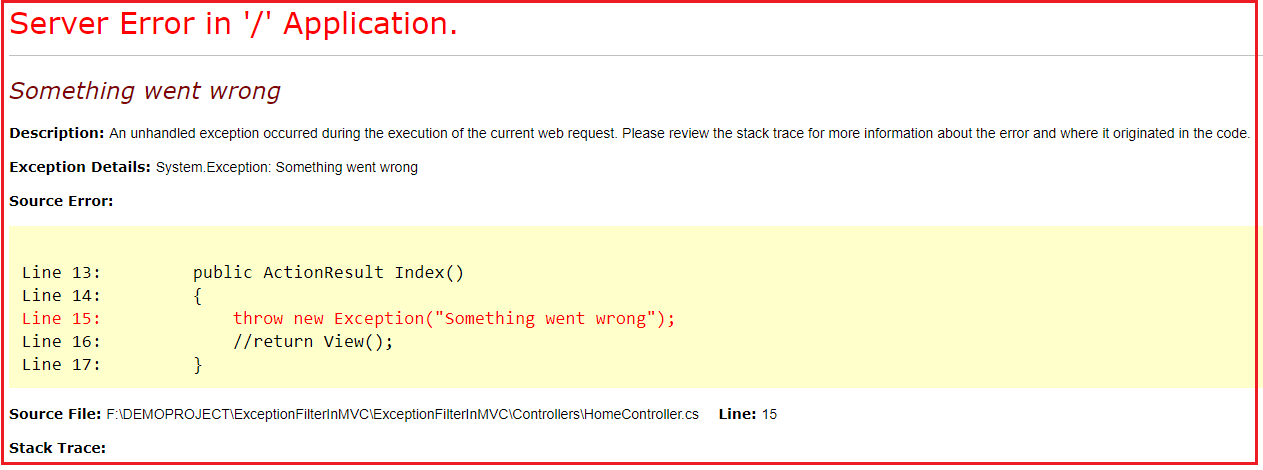
//return View();

**}**

**}**

**}**

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>**Unknown 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"**)**;

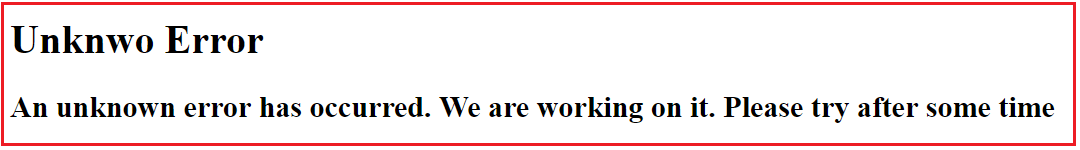
//return View();

**}**

**}**

**}**

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>**

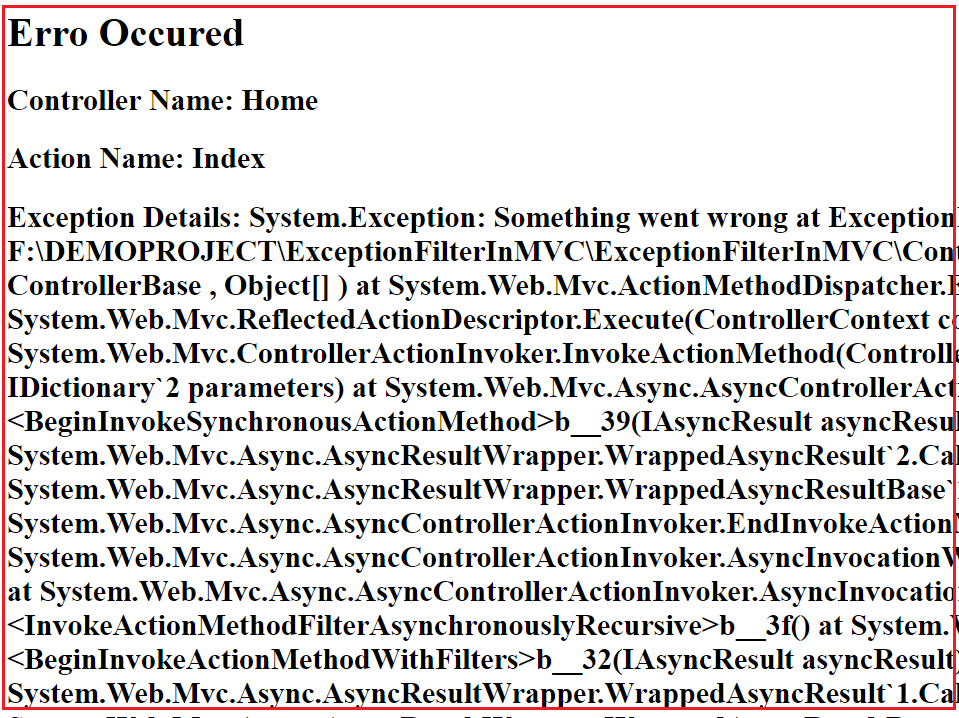
**<h2>**Exception Details: @Model.Exception**</h2>**

**</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.

Name a few action filters in mvc?

Authorize

ChildActionOnly

HandleError

OutputCache

RequireHttps

ValidateInput

ValidateAntiForgeryToken

### Authorize and AllowAnonymous action filters in mvc

In ASP.NET MVC, by default, all the controller action methods are accessible to both **anonymous**and **authenticated** users. If you want action methods, to be available only for authenticated and authorised users, then use Authorize attribute. Let us understand **"Authorize"** and **"AllowAnonymous"** action filters with an example.  
  
  
  
**1.** Create a blank asp.net mvc4 application. Name your application MVCDemo.  
  
**2.** Right click on the **"Controllers"** folder and add HomeController. Copy and paste the following code.   
public class HomeController : Controller  
{  
    public ActionResult NonSecureMethod()  
    {  
        return View();  
    }  
  
    public ActionResult SecureMethod()  
    {  
        return View();  
    }  
}  
  
Right click on **NonSecureMethod**() and add a view with name = **NonSecureMethod**. Similarly add a view with name = **SecureMethod**.

Now remove the [Authorize] attribute from SecureMethod(), and apply it on the HomeController.  
[Authorize]  
public class HomeController : Controller  
{  
    public ActionResult NonSecureMethod()  
    {  
        return View();  
    }  
  
    public ActionResult SecureMethod()  
    {  
        return View();  
    }  
}  
  
At this point, **"Authorize"** attribute is applicable for all action methods in the HomeController. So, only authenticated users will be able to access SecureMethod() and NonSecureMethod().  
  
To allow anonymous access to NonSecureMethod(), apply [AllowAnonymous] attribute. AllowAnonymous attribute is used to skip authorization enforced by Authorize attribute.   
[AllowAnonymous]  
public ActionResult NonSecureMethod()  
{  
    return View();  
}

### childactiononly attribute in mvc

Add HomeController. Copy and paste the following code.  
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);  
    }  
}  
  
  
  
**Step 3:** Right click on the **"Countries()"** action method and add **"Countries"** view. This view will render the given list of strings as an un-ordered list.  
  
@model List<string>  
@foreach (string country in Model)  
{  
    <ul>  
        <li>  
            <b>  
                @country  
            </b>  
        </li>  
    </ul>  
}  
  
**Step 4:** Right click on the **"Index()"** action method and add **"Index"** view.  Copy and paste the following code. Notice that, to invoke childaction, we are using Action() HTML Helper.  
  
<h2>Countries List</h2>  
@Html.Action("Countries", new { countryData = new List<string>() { "US", "UK", "India" } })  
  
**Please Note:** Child actions can also be invoked using **"RenderAction()"** HTMl helper as shown below.  
@{  
    Html.RenderAction("Countries", new { countryData = new List<string>() { "US", "UK", "India" } });  
}  
  
**Points to remember about "ChildActionOnly" attribute**  
**1.** Any action method that is decorated with [ChildActionOnly] attribute is a child action method.  
  
**2.** Child action methods will not respond to URL requests. If an attempt is made, a runtime error will be thrown stating - Child action is accessible only by a child request.  
  
**3.** Child action methods can be invoked by making child request from a view using "Action()" and "RenderAction()" html helpers.  
  
**4.** An action method doesn’t need to have [ChildActionOnly] attribute to be used as a child action, but use this attribute to prevent if you want to prevent the action method from being invoked as a result of a user request.  
  
**5.** Child actions are typically associated with partial views, although this is not compulsory.  
  
**6.** Child action methods are different from NonAction methods, in that NonAction methods cannot be invoked using Action() or RenderAction() helpers.