**Filters in ASP.NET Core MVC**

In this article, I will discuss **Filters in ASP.NET Core MVC** Applications with Examples. Filters are one of the most important concepts in ASP.NET Core MVC Applications, and as developers, we should be aware of this concept. So, from this and in a few upcoming articles, I will discuss ASP.NET Core MVC Filters in Detail. As part of this article, we will discuss the following pointers.

1. **What are Filters in ASP.NET Core MVC?**
2. **Why are Filters in ASP.NET Core MVC?**
3. **What are the Differences Between Filters and Middlewares in ASP.NET Core?**
4. **Types of Filters in ASP.NET Core MVC**
5. **Advantages and Disadvantages of Filters in ASP.NET Core MVC**
6. **Where can we configure filters in ASP.NET Core MVC?**
7. **Default Filters Execution Order in ASP.NET Core MVC**

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

Filters are used to add cross-cutting concerns, such as Logging, Authentication, Authorization, Exception Handling, Caching, etc., to our application.

In ASP.NET Core Applications, Filters allow us to execute cross-cutting logic in the following ways:

1. Before an HTTP request is handled by a controller action method.
2. After an HTTP request is handled by a controller action method.
3. After the response is generated but before it is sent to the client.

We have already discussed in [**Routing**](https://dotnettutorials.net/lesson/routing-asp-net-core-mvc/) that when a client makes a request, that request comes to the Routing Engine, and then the Routing Engine navigates that Request to the Controller Action Method. So, the Controller action method will handle the incoming request and send the response back to the client who initially made the request, as shown in the image below.

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

If that is your requirement, then you need to use Filters in your ASP.NET Core application. Filters in ASP.NET Core Applications are the Attributes that allow us to inject logic or code before or after an action method is executed.

**Why are Filters in ASP.NET Core MVC?**

In ASP.NET Core MVC, Filters are used to Perform cross-cutting concerns.

They are as follows:

* Caching
* Logging
* Error Handling
* Modifying the Result
* Authentication and Authorization, etc.

**What are the Differences Between Filters and Middlewares in ASP.NET Core?**

Before proceeding further, we need to understand the differences between Filters and Middleware Components in ASP.NET Core applications. The following are the Key differences between Filters and Middleware Components:

**Scope of Application (Where they Applied):**

* **Filters:** Filters are primarily applied to individual controller actions or controllers. They are used to add specific behaviors or concerns to processing a single action or a group of actions within a controller.
* **Middlewares:** Middlewares are applied to the entire application’s request processing pipeline. They can handle requests and responses globally, regardless of the specific controller or action being invoked.

**Execution Point (When they Execute):**

* **Filters:** Filters execute within the ASP.NET Core Framework’s pipeline and are part of the controller/action execution process. They are triggered before or after the execution of a specific action method.
* **Middlewares:** Middlewares execute earlier in the Request Processing Pipeline, typically before the request reaches the ASP.NET Core MVC Controller action method. They can intercept requests and responses and perform tasks at various stages of the request processing pipeline, such as Routing, Authentication, Response Formatting, etc.

**Purpose and Concerns (Why they are Used):**

* **Filters:** Filters are designed to handle concerns specific to the ASP.NET Core MVC framework, such as Logging, Authentication, Authorization, Exception Handling, Caching, Custom Logic, etc.
* **Middlewares:** Middlewares are more general-purpose and can handle many concerns, including Routing, Authentication, Request/Response Logging, Compression, Security, etc.

**Configuration (How they Configured):**

* **Filters:** Filters are typically configured using attributes (e.g., **[Authorize], [AllowAnonymous], etc.**) on controllers or action methods. You can also register global filters in the Program Class.
* **Middlewares:** Middlewares are configured and ordered in the Program class (e.g., **UseHttpsRedirection(), UseAuthorization(), MapControllers(), etc**).

**Execution Order (What is the Execution Order):**

* **Filters:** The execution order for filters is decided based on the type of filters you are applying to the controllers and action methods. So, the order of Filters is not important.
* **Middlewares:** The execution order for middleware components is determined by the order in which they are added to the IApplicationBuilder pipeline, i.e., to the Request Processing Pipeline. So, the order of Middleware Components is important.

So, Filters and Middleware Components serve different purposes and have different scopes within an ASP.NET Core MVC application. Filters are more tightly integrated with the ASP.NET Core MVC framework and are applied at the controller/action level. On the other hand, Middleware Components are applied globally to the entire application.

**Types of Filters in ASP.NET Core MVC**

In ASP.NET Core MVC, there are several types of filters available that you can use to modify the behavior in the request and response processing pipeline. The following are the commonly used Filters in ASP.NET Core MVC Applications:

**Authorization Filters in ASP.NET Core:**

The Authorization Filter is used to perform Authentication and Authorization checks before an action method is executed. Examples include **AuthorizeAttribute** for role-based or policy-based authorization and **AllowAnonymousAttribute** to allow unauthenticated users to access an action.

* **[Authorize]:** This Built-in Filter restricts access to actions or controllers for UnAuthenticated Users. You can also specify Roles, Policies, or Claims based on which it can also decide who can access specific resources.
* **[AllowAnonymous]:** This Built-in Filter allows unauthenticated users to access actions or controllers.
* **Custom Authentication:** You can also create Custom Authentication. To do so, we need to create a class implementing the **IAuthorizationFilter** interface and provide implementations for the **OnAuthorization** method, where we need to write the custom authentication logic according to our business requirements.

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

The Action Filters in the ASP.NET Core MVC Application are executed before and after an action method is executed. They perform tasks like Logging, Modifying the Action’s Arguments, or Altering the Action’s Result.

In ASP.NET Core, you can create a Custom Action Filter in two ways:

First, by creating a class implementing the **IActionFilter** interface and providing implementations for**[OnActionExecuting] and [OnActionExecuted]** methods.

Second, by creating a class inherited from the **ActionFilterAttribute** class and overriding the**[OnActionExecuting] and [OnActionExecuted]** methods.

**[OnActionExecuting] and [OnActionExecuted]:** These are the two methods within a Custom Action Filter to execute logic before and after an action method is called. The OnActionExecuting method executes before the action method is invoked, and the OnActionExecuted method executes after the action method is invoked.

**How Do We Apply the Custom Filter to Controllers and Action Methods in ASP.NET Core?**

If the custom class is created by overriding the **ActionFilterAttribute** class, then it is an Attribute, and we can directly apply that Custom Filter to the Action methods or Controllers. However, you cannot apply Custom Action Filters (if the custom class is created by implementing the **ActionFilter** interface) directly to the controllers and action methods. For this, we need to use **TypeFilter and ServiceFilter** as built-in attributes.

TypeFilter and ServiceFilter are built-in attributes in ASP.NET Core that apply Custom Action Filters as Attributes to controller actions or controllers. Later, as we progress, we will discuss the differences between TypeFilter and ServiceFilter Attributes and how to use them.

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

The Result Filters in ASP.NET Core MVC Application runs after the action method has been executed but before the result is processed and sent to the client.

This means you can modify the view or the result data before it gets rendered to the output stream. They are used for tasks such as Adding Headers to the response, Modifying the Result, etc.

In ASP.NET Core, you can create a Custom Result Filter in two ways: First, by creating a class implementing the **IResultFilter** interface and providing implementations for **[OnResultExecuting] and [OnResultExecuted]** methods.

Second, by creating a class inherited from the **ResultFilterAttribute** class and overriding the **[OnResultExecuted] and [OnResultExecuted]** methods.

**[OnResultExecuting] and [OnResultExecuted]:**These are the two methods within a

Custom Result Filter to execute logic before and after the Result is generated. The **OnResultExecuting** method executes before the result is generated, and the **OnResultExecuted** method executes after the result is generated.

**Exception Filters in ASP.NET Core:**

The Exception Filters are executed when an unhandled exception occurs during the execution of an action method. They are used for Logging, Error Handling, and Displaying Different Error Pages Based on the Error Status Codes.

In ASP.NET Core, you can create a Custom Exception Filter in two ways: First, you can create a class implementing the **IExceptionFilter** interface and provide implementations for the **[OnException]** method. Second, by creating a class inherited from the **ExceptionFilterAttribute** class and overriding the **[OnException]** method.

**Where Can We Configure Filters in ASP.NET Core MVC?**

In the ASP.NET Core MVC Application, Filters can be configured in the following three places:

**Configuring Filters at Controller Level:**

We can apply filters at the controller level by decorating the controller with the Filter attribute, as shown in the below code. When we apply the Filter at the controller level, it will apply to all the actions of that controller.

**[**Authorize**]** // Authorization filter applied at the controller level

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

**{**

**public** IActionResult Index**()**

**{**

// Action logic

**}**

**}**

**Configuring Filters at Globally:**

We can configure filters globally in the Program class. By adding filters as services, we can ensure they are applied globally to all controllers and actions in our application. Following is an example of configuring a global filter in the Program.cs class:

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new MyGlobalFilter**())**;

**})**;

**Action Method Level:**

We can also apply filters directly to individual action methods within our controller by using the Filter Attribute, as shown in the below code. This allows us to apply specific filters only to specific action methods.

**public** **class** MyController : Controller

**{**

**[**MyCustomFilter**]** // Custom filter applied to this action

**public** IActionResult MyAction**()**

**{**

// Action logic

**}**

**}**

**Default Filters Execution Order in ASP.NET Core MVC**

In ASP.NET Core MVC, filters are executed in a specific order known as the “**Default Execution Order**“. The default execution order ensures filters are applied properly throughout the request processing pipeline.

The default execution order, from the earliest to the latest in the pipeline, is as follows:

* **Authorization Filters:** Authorization filters are executed first. They are responsible for checking whether the current user can access the requested resource or action. If authorization fails, the request will be short-circuited, and the action method will not be executed.
* **Action Filters (Before Action Execution):** Action filters with “**Before Action Execution**” logic are executed before the action method is invoked. These filters can perform tasks like logging, input validation, or pre-processing data.
* **Model Binding:** Model binding occurs at this stage. It binds incoming data to action method parameters and executes model validation.
* **Action Execution:** The action method itself is executed.
* **Action Filters (After Action Execution):** Action filters with “**After Action Execution**” logic are executed after the action method completes its execution. These filters can perform tasks like logging, post-processing data, etc.
* **Result Filters (Before Result Execution):** Result filters with “**Before Result Execution**” logic are executed before the action result is executed. These filters can modify the result or perform additional processing.
* **Action Result Execution:** The action result, which can be a view or any other result type, is executed.
* **Result Filters (After Result Execution):** Result filters with “**After Result Execution**” logic are executed after the action result has been executed. These filters can perform tasks like logging or post-processing of the result.
* **Exception Filters (If an Exception Occurs):** Exception filters are executed if an unhandled exception occurs during the request’s processing. These filters can handle the exception, log it, and return an appropriate error response.

In the next article, I will discuss the [**Exception Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/exception-filter-in-asp-net-core-mvc/) Application**.**In this article, I try to explain Filters in ASP.NET Core MVC Applications with Examples. I hope you understand the need for and use of filters in ASP.NET Core MVC Applications.

Exception Filter in ASP.NET Core MVC

**Exception Filter in ASP.NET Core MVC**

In this article, I will discuss the **Exception Filter in ASP.NET Core MVC** Application with Examples. Please read our previous article discussing the basic concepts of [**Filters in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/filters-in-asp-net-core-mvc/) Applications. As part of this article, we will discuss the following pointers in detail.

1. **What is an Exception Filter in ASP.NET Core MVC?**
2. **Built-in Exception Filter in ASP.NET Core MVC**
3. **Example to understand UseExceptionHandler Middleware**
4. **How Do We Create a Custom Exception Filter in ASP.NET Core MVC?**
5. **Differences Between Exception Handler and Exception Filter in ASP.NET Core.**

**What is an Exception Filter in ASP.NET Core MVC?**

In ASP.NET Core MVC, Exception Filter allows us to handle unhandled exceptions that occur while processing an HTTP request within our application.

Exception Filters in ASP.NET Core Applications are used for tasks such as Logging Exceptions, Returning Custom Error Responses, and Performing any other necessary action when an exception occurs.

Exception Filter also provides a way to centralize the exception-handling logic and keep it separate from our controller or from the business and data access logic, making our code more organized and maintainable.

**Built-In Exception Filter in ASP.NET Core MVC**

The ASP.NET Core Framework does not provide any specialized Built-in Exception Filter for Global Error Handling.

As developers, we generally use the UseExceptionHandler middleware component to handle exceptions globally.

However, we can also create a Custom Exception Filter to handle unhandled exceptions globally.

In ASP.NET Core, we can create the Custom Exception Filter in two ways:

First, we can create a class implementing the **IExceptionFilter** interface and provide implementations for the **[OnException]** method.

Second, by creating a class inherited from the **ExceptionFilterAttribute** class and overriding the**[OnException]** method.

**Example to understand UseExceptionHandler Middleware in ASP.NET Core**

Let us first see an example of how the UseExceptionHandler middleware component handles unhandled exceptions globally. Then, we will see how to create a Custom Exception Filter.

The UseExceptionHandler Middleware allows us to specify an error-handling path that will be executed when an unhandled exception is thrown within our application. We can configure this using the Main method of our Program class.

To configure UseExceptionHandler middleware for global Exception Handling, please add the following code to the program class. With the following code in place, if an exception occurs and if the environment is not Development, then the user will be redirected to the “**/Home/Error**” URL. Here, Home is the Controller name, and Error is the action method within the Home Controller. If the environment is Development, it will display the complete error details using the UseDeveloperExceptionPage middleware component.

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

**else**

**{**

// This will handle exceptions and redirect to the specified error page.

app.UseExceptionHandler**(**"/Home/Error"**)**;

**}**

**Modify the Home Controller:**

Next, modify the Home Controller as follows. As you can see in the code below, within the Index action method, we have written the logic in a way that it will throw runtime exceptions while executing it. Further, if you notice, we have not handled that exception within the Index action method. Again, we have added the Error action method, which should be executed when there is an unhandled exception and if the environment is not set to be Development.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

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

**{**

**int** x = 10;

**int** y = 0;

**int** z = x / y;

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

**}**

**public** ActionResult Error**()**

**{**

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

**}**

**}**

**}**

Next, add the **Error.cshtml** view of the Home Controller. Once you add the Error view, copy and paste the following code. The following view will be executed when your application has an unhandled exception.

@**{**

Layout = **null**;

**}**

**<**!DOCTYPE html**>**

**<**html**>**

**<**head**>**

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

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

**<**/head**>**

**<**body**>**

**<**hgroup**>**

**<**h1 style="color:red"**>**Unknwo Error**<**/h1**>**

**<**h2 style="color:red"**>**An unknown error has occurred. We are working **on** it. Please **try** after some time**<**/h2**>**

**<**/hgroup**>**

**<**/body**>**

**<**/html**>**

Now, set the environment to Production or Staging and run the application, and you should see the following generic error message.

if (!env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

**How Do We Create a Custom Exception Filter in ASP.NET Core MVC?**

As we already discussed, we can create the Custom Exception Filter in two ways: First, we can create a class implementing the **IExceptionFilter** interface and provide implementations for the **[OnException]** method. Second, we can create a class inherited from the **ExceptionFilterAttribute** class and override the**[OnException]** method.

Let us see an example of creating a custom exception filter that logs the exception details. The following are the steps to create and use a Custom Exception filter in ASP.NET Core MVC Application:

**Create a Custom Exception Filter Class**

So, create a class file named **CustomExceptionFilter.cs** and copy and paste the following code. The class inherits from the **ExceptionFilterAttribute** and overrides the **OnException** method, where we write our custom logic. In the code below, we fetch the Controller Name, Action Name, and Exception message from the Context object and then log the details into a text file.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomExceptionFilter : ExceptionFilterAttribute

**{**

**public** **override** **void** OnException**(**ExceptionContext context**)**

**{**

var controllerName = context.RouteData.Values**[**"controller"**]**;

var actionName = context.RouteData.Values**[**"action"**]**;

**string** message = $"\nTime: {DateTime.Now}, Controller: {controllerName}, Action: {actionName}, Exception: {context.Exception.Message}";

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

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

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

**Applying the Custom Exception Filter:**

We can apply the Custom Exception Filter at three levels: Globally, At the Action Method Level, and At the Controller Level. We have created the Custom Exception Filter inheriting from the **ExceptionFilterAttribute**, which means our Custom Exception Filter is also an **Attribute**. As it is an Attribute, we can directly apply it to the Controller or Action Methods. Let us modify the Home Controller as follows to apply the Custom Exception Filter on the Index Action method.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**CustomExceptionFilter**]**

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

**{**

**int** x = 10;

**int** y = 0;

**int** z = x / y;

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

**}**

**}**

**}**

**Note:** If you create the Custom Exception Filter by implementing the **IExceptionFilter** or **IAsyncExceptionFilter** interface, you cannot apply that filter directly to the Controllers or Action Methods. In that case, you need to use the Built-in **TypeFilter vs ServiceFilter** Attribute at the Controller and Action Methods to specify the Custom Filters. These two Built-in Filter Attributes are specifically designed for this purpose. In our upcoming articles, we will discuss the differences [**Between TypeFilter and ServiceFilter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/difference-between-typefilter-and-servicefilter-in-asp-net-core-mvc/).

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1496c14a-b07f-4c33-55c1-c6322466cfc2&d_id=118560&imp_id=256644396905474&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**Registering the Filter Globally**

We can also register the filter globally to apply to all actions in the application. This can be done in the **Program.cs** class file as follows:

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomExceptionFilter**())**;

**})**;

**Creating Custom Exception Filter by Implementing IExceptionFilter Interface:**

Let us see how we can implement the **IExceptionFilter** Interface and provide implementations for the **OnException** method to create a Custom Exception Filter. Let us modify the CustomExceptionFilter class as follows, which inherits from the IExceptionFilter Interface and implements the OnException method. As part of the OnException method, we have written our custom exception logic, which will log the unhandled exception details to a text file:

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomExceptionFilter : IExceptionFilter

**{**

**public** **void** OnException**(**ExceptionContext context**)**

**{**

var controllerName = context.RouteData.Values**[**"controller"**]**;

var actionName = context.RouteData.Values**[**"action"**]**;

**string** message = $"\nTime: {DateTime.Now}, Controller: {controllerName}, Action: {actionName}, Exception: {context.Exception.Message}";

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

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

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

**Note:** You need to remember that the above CustomExceptionFilter class is no longer an Attribute. Hence, we cannot apply this CustomExceptionFilter as an Attribute in our Controller or Action Method.

**Register the Custom Exception Filter Globally**

You can register the custom exception filter globally in the **Program.cs** file as follows. This is for Global registration:

// Global registration

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomExceptionFilter**())**;

**})**;

OR

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**typeof**(**CustomExceptionFilter**))**;

**})**;

You can apply the custom exception filter to specific controllers or actions using **[ServiceFilter or TypeFilter](https://dotnettutorials.net/lesson/difference-between-typefilter-and-servicefilter-in-asp-net-core-mvc/)** attributes. So, modify the Home Controller as follows:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**ServiceFilter**(**typeof**(**CustomExceptionFilter**))]**

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

**{**

**int** x = 10;

**int** y = 0;

**int** z = x / y;

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

**}**

**}**

**}**

So, by following these steps, we can create and apply a Custom Exception Filter in our ASP.NET Core MVC Application.

**Redirecting to an Error View**

Let’s see an example of how the Exception Filter redirects the user to a Custom Error view when an unhandled exception occurs while processing the request. This is similar to the example we created with the UseExceptionHandler middleware component. Let’s create the error view within the **Views/Shared** folder named **Error.cshtml** and then copy and paste the following code.

@**{**

Layout = **null**;

**}**

**<**!DOCTYPE html**>**

**<**html**>**

**<**head**>**

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

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

**<**/head**>**

**<**body**>**

**<**hgroup**>**

**<**h1 style="color:red"**>**Unknwon Error**<**/h1**>**

**<**h2 style="color:red"**>**An unknown error has occurred. We are working **on** it. Please **try** after some time**<**/h2**>**

**<**/hgroup**>**

**<**/body**>**

**<**/html**>**

**Creating the Custom Exception Filter Attribute:**

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** RedirectToErrorViewFilter : ExceptionFilterAttribute

**{**

**public** **override** **void** OnException**(**ExceptionContext context**)**

**{**

context.Result = new ViewResult

**{**

ViewName = "Error"

**}**;

context.ExceptionHandled = **true**;

**}**

**}**

**}**

**Next, modify the Home Controller as follows:**

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**RedirectToErrorViewFilter**]**

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

**{**

// Simulate an authorization exception

**throw** new UnauthorizedAccessException**(**"Access Denied."**)**;

**}**

**}**

**}**

**When Should We Use Exception Filter in ASP.NET Core MVC?**

The Custom Exception filters are useful in the following scenarios:

* **Centralized Exception Handling:** If you want a single, centralized location to handle all the unhandled exceptions that occur throughout your application, exception filters are a good choice.
* **Error Logging:** Exception filters provide a convenient mechanism for logging exceptions or collecting error information. You can log exceptions to various targets (e.g., files, databases, or external services).
* **Custom Error Responses:** Exception filters are helpful when you need to customize the HTTP response sent to the client in case of an exception. You can set specific HTTP status codes, return custom error messages, and format responses according to your application’s requirements.

**Difference Between Exception Handler and Exception Filter in ASP.NET Core**

In ASP.NET Core, Exception Handlers (UseExceptionHandler) and Exception Filters (Custom Exception Filters) handle unhandled exceptions that occur during application execution. Let us proceed and understand the differences between them:

**Exception Handler in ASP.NET Core**

The Exception Handler in ASP.NET Core is used to catch unhandled exceptions that occur during the processing of HTTP requests in the middleware pipeline. It catches exceptions globally for the entire application. The following are the key points of the Exception Handler.

* Typically configured in the Program.cs file.
* Use app.UseExceptionHandler to configure a global exception handler.
* It is useful for catching exceptions that occur in the middleware before the execution reaches the MVC action methods.
* A typical use case is configuring a centralized error handling page or error logging middleware.
* It is often used to return generic error responses in production to hide sensitive error information from end-users.

**Exception Filter in ASP.NET Core**

The Exception Filters in ASP.NET Core apply custom error-handling logic to handle unhandled exceptions thrown by controller action methods. The following are the key points of the Exception Filter.

* It is implemented as part of the MVC pipeline.
* It can be applied globally, per-controller or per-action, using attributes.
* It is useful for handling exceptions specific to controller actions, such as customizing the response based on the type of exception thrown by an action.
* A common use case is to catch domain-specific exceptions in a controller and return a specific HTTP status code or view.
* It is often used to transform the exception into a custom error view or a JSON response.

**Key Differences Between Exception Handler and Exception Filter in ASP.NET Core**

* **Context of Operation:** Exception handlers work at the middleware level and handle global exceptions across the entire application. Exception filters are part of the MVC filter pipeline and handle the exceptions thrown by controller actions.
* **Configuration:** Exception handlers are configured in the middleware pipeline, usually in the Program.cs file. Exception filters are configured in the MVC pipeline and can be applied using attributes.

In the next article, I will discuss [**Handling Non-Success HTTP Status Codes in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/handling-non-success-http-status-codes-in-asp-net-core-mvc/) Applications. In this article, I try to explain the Exception Filter in ASP.NET Core MVC Application with Examples. I hope you understand the need and use of the Exception Filter in ASP.NET Core MVC applications.

Handling Non-Success HTTP Status Codes in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

**Handling Non-Success HTTP Status Codes in ASP.NET Core MVC**

In this article, I will discuss **Handling Non-Success HTTP Status Codes**, i.e., the use of **UseStatusCodePages**, **UseStatusCodePagesWithRedirects**, and **UseStatusCodePagesWithReExecute** middleware components in ASP.NET Core MVC Application with Examples. Please read our previous article discussing [**Exception Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/exception-filter-in-asp-net-core-mvc/) Applications.

Here, we will discuss how to handle 404 Errors, i.e., Page Not Found Errors, in a centralized way with different approaches. At the end of this article, you will understand the following 3 Middleware Components that deal with Status Code Pages in ASP.NET Core MVC Applications.

* **UseStatusCodePages**
* **UseStatusCodePagesWithRedirects**
* **UseStatusCodePagesWithReExecute**

**Types of 404 Errors in ASP.NET Core MVC**

There are 2 types of 404 errors.

**Type 1: Resource With the Specified ID Does Not Exist.**

This type of 404 error occurs when we cannot find the employee, product, customer, etc., with the provided ID. Let us understand this with an example. First, create the following Product model, which will display the Product details to the user.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=85317658-914f-478d-4120-24956f01c776&d_id=118560&imp_id=7780067892840986&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**namespace** *FiltersDemo.Models*

**{**

**public** **class** Product

**{**

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

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

**public** **decimal** Price **{** **get**; **set**; **}**

**}**

**}**

**Modifying the Controller:**

Then modify the HomeController as follows. Please have a look at the Details action method. Here, if the Product details are found, it will render the Details view with the Product details. On the other hand, if the Product is not found, it renders the ProductNotFound view.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

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

**{**

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

**}**

**public** ViewResult Details**(int** id**)**

**{**

//Let us assume the Product with the given Id is does not exists in the database

Product? product = **null**;

**if** **(**product == **null)**

**{**

Response.StatusCode = 404;

**return** View**(**"ProductNotFound", id**)**;

**}**

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

**}**

**}**

**}**

**Creating ProductNotFound View:**

Next, create a view with the name **ProductNotFound**.**cshtml** within the **Views/Home** folder and copy and paste the following code. In this case, we know the user is trying to go to the Product details page, but the provided Product ID value is invalid. So, we are returning a custom error page with the message; the ID Does Not Exist.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=85317658-914f-478d-4120-24956f01c776&d_id=118560&imp_id=2015375166904222&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

@model **int**

@**{**

ViewBag.Title = "404 Error";

**}**

**<**div **class**="alert alert-danger mt-1 mb-1"**>**

**<**h4**>**404 Not Found Error :**<**/h4**>**

**<**hr /**>**

**<**h5**>**

Product with ID = @Model Does Not Exists

**<**/h5**>**

**<**/div**>**

Now, access the **Home/Details/10** URL, and you will see the following error message.

**Type 2: The Provided URL Does Not Match Any Route In Your Application.**

Consider the following URL, which does not exist. This also results in a 404 error.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=85317658-914f-478d-4120-24956f01c776&d_id=118560&imp_id=7074642534858286&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

In this case, we are unaware of the page the user is trying to get, so displaying a custom error page is impossible. Now, let us proceed and understand how to handle this type of 404 error centrally.

**Default 404 Error Page in ASP.NET Core**

We have nothing configured in the HTTP request processing pipeline to handle 404 errors. So, if we navigate to **/Home/Something**, we see the following default 404 error page. This is because the URL **/Home/Something** does not match any routes in our application.

**Handling Non-Success HTTP Status Codes in ASP.NET Core MVC:**

To handle non-success HTTP status codes such as **404 in ASP.NET Core MVC** Application, we can use the following 3 built-in Middleware components.

* **UseStatusCodePages**
* **UseStatusCodePagesWithRedirects**
* **UseStatusCodePagesWithReExecute**

**UseStatusCodePages Middleware in ASP.NET Core MVC**

This is the least useful middleware component among the 3. It always gives a **Generic Error Message**, and we don’t have the option to customize or provide a user-friendly message. For this reason, we rarely use it in a real-time application.

Advertisements

To use it in our ASP.NET Core MVC Application and see what it can do, we need to register it into the Request Processing pipeline. So, modify the **Program** class as follows to register the **UseStatusCodePages** Middleware Components, and this component will come into existence when the environment is not Development:

**namespace** *FiltersDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllersWithViews**()**;

var app = builder.Build**()**;

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

**else**

**{**

app.UseStatusCodePages**()**;

**}**

app.UseHttpsRedirection**()**;

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

app.MapControllerRoute**(**

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}"**)**;

app.Run**()**;

**}**

**}**

**}**

With **UseStatusCodePages** Middleware configured, now, if we navigate to URL **/Home/Something**, it returns the following simple text response.

**UseStatusCodePagesWithRedirects Middleware in ASP.NET Core MVC**

We want to handle these non-success HTTP status codes (other than 200 Status Code) in the production environment and return a Custom Error Page. To achieve this, we can either use **UseStatusCodePagesWithRedirects** or **UseStatusCodePagesWithReExecute** middleware components. Let’s understand these two Middleware Components with Examples, and then we will see the differences.

First, create the **Error Controller** and copy and paste the following code. Here, within the **HttpStatusCodeHandler** action method, we check if the status code is **404**. If it is 404, we set the custom error message into the ErrorMessage ViewBag property and render the NotFound view.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

// If there is 404 status code, the route path will become Error/404

**[**Route**(**"Error/{statusCode}"**)]**

**public** IActionResult HttpStatusCodeHandler**(int** statusCode**)**

**{**

**switch** **(**statusCode**)**

**{**

**case** 404:

ViewBag.ErrorMessage = "Sorry, the resource you requested could not be found";

**break**;

**}**

**return** View**(**"NotFound"**)**;

**}**

**}**

**}**

**Creating NotFound View:**

Next, we need to add the **NotFound.cshtml**view within the **Views/Error** folder. Once you add the view, copy and paste the following code. Here, we are simply displaying the **ViewBag.ErrorMessage** property value, which is set by the **HttpStatusCodeHandler** action method.

@{

ViewBag.Title = "Not Found";

Layout = null; //If you want the Layout, you can set the layout

}

**<h1>**@ViewBag.ErrorMessage**</h1>**

Next, modify the **Program** class as follows. Here, you can see we are registering the **UseStatusCodePagesWithRedirects** middleware component, which will come into existence when the environment is not set to Development.

**namespace** *FiltersDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllersWithViews**()**;

var app = builder.Build**()**;

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

**else**

**{**

app.UseStatusCodePagesWithRedirects**(**"/Error/{0}"**)**;

**}**

app.UseHttpsRedirection**()**;

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

app.MapControllerRoute**(**

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}"**)**;

app.Run**()**;

**}**

**}**

**}**

The following line of code is in place: If there is a 404 error, the user is redirected to **/Error/404**. The placeholder **{0}**, in “**/Error/{0}**” will automatically receive the HTTP status code.

**app.UseStatusCodePagesWithRedirects(“/Error/{0}”);**

Run the application and navigate to the URL **/Home/Something**. You will see the following custom 404 error view **NotFound.cshtml** as expected.

To use **UseStatusCodePagesWithReExecute** middleware instead of **UseStatusCodePagesWithRedirects** middleware

**REPLACE** **app.UseStatusCodePagesWithRedirects(“/Error/{0}”);**

**WITH** **app.UseStatusCodePagesWithReExecute(“/Error/{0}”);**

Re-run the application and navigate to **/Home/Something**, and you should see the same custom 404 error view **NotFound.cshtml**. The obvious question that should come to your mind is, what’s the difference between these 2 middleware components, and which one we should use?

**Difference Between UseStatusCodePagesWithRedirects and UseStatusCodePagesWithReExecute Middleware in ASP.NET Core MVC:**

In an ASP.NET Core MVC Application, UseStatusCodePagesWithRedirects and UseStatusCodePagesWithReExecute are middleware components that handle NON-SUCCESS HTTP Status Codes and provide custom error handling. Before understanding the differences between these two Middleware Components, we first need to understand how they handle the no-success HTTP Status code internally.

**Request Processing with UseStatusCodePagesWithRedirects in ASP.NET Core MVC**

So, when the **UseStatusCodePagesWithRedirects** handles the Non-Success HTTP Status Code, it will actually return a **302 Status** code to the browser. When it returns the **302 Status Code**, it also provides the GET Request Path to be executed by the browser, and the browser will issue a new Request to the requested path. The GET Request path needs to be configured in the UseStatusCodePagesWithRedirects component. In this case, the /**Error/{0}**, i.e., **/Error/404** as the placeholder**{0}**, will be replaced by the Status Code 404.

302 status code means the URI of the requested resource has been temporarily changed; in our case, it changed from **/Home/Something**to **/Error/404**. Hence, the browser will issue another GET Request with the updated URL, **/Error/404.**Because a redirect is issued, the URL in the address bar changes from **/Home/Something** to **/Error/404**.

The new GET request (i.e., **/Error/404**) flows through the pipeline. It is handled by the HttpStatusCodeHandler action method of Error Controller, which will return the NotFound view HTML with status code 200 (which means the request was completed successfully). As far as the browser is concerned, in this entire request flow, there was no 404 error. For a better understanding, please have a look at the following diagram:

**Request Processing with UseStatusCodePagesWithReExecute in ASP.NET Core MVC**

The UseStatusCodePagesWithReExecute middleware also intercepts responses with error status codes. Instead of redirecting the client to a new URL, it re-executes the request pipeline using an internal rewrite to the specified path.

In our example, when a request is issued to **/Home/Something**, a 404 status code is raised. The UseStatusCodePagesWithReExecute middleware intercepts the 404-status code and re-executes the pipeline, pointing it to the URL (**/Error/404**). The request then flows through the pipeline and is handled by the MVC middleware (HttpStatusCodeHandler action method of Error Controller handles the request), which returns NotFound view HTML with status code 200. Then, the UseStatusCodePagesWithReExecute middleware uses the HTML response returned by the NotFound view but replaces the 200 status code with the original 404 status code.

As the name implies, it re-executes the pipeline, keeping the correct (404) status code. It just returns the custom view (NotFound) HTML. As it is just re-executing the pipeline and not issuing a redirect request, it preserves the original URL (/Home/Something) in the address bar. It does not change from /Home/Something to /Error/404. For a better understanding, please have a look at the following diagram:

**Key Differences:**

The following is the key difference between UseStatusCodePagesWithRedirects and UseStatusCodePagesWithReExecute:

* UseStatusCodePagesWithRedirects redirects the browser to a different URL for error handling, resulting in a new request to the error page. So, use UseStatusCodePagesWithRedirects when you need a simple way to redirect the users to an error page without much concern about the original URL or request method.
* UseStatusCodePagesWithReExecute re-executes the original request with a modified route to handle errors within the same request pipeline. So, use UseStatusCodePagesWithReExecute when maintaining the original URL is important, and you might need to handle errors differently based on the original context of the request.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=85317658-914f-478d-4120-24956f01c776&d_id=118560&imp_id=6246131952898381&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

In the next article, I will discuss [**Error Pages Based on Status Codes in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/error-pages-based-on-status-code-in-asp-net-core-mvc/) Applications. In this article, I try to explain Error Pages Based on Status Codes in ASP.NET Core MVC Applications with Examples. I hope you enjoy this Error Pages Based on Status Code in the ASP.NET Core MVC article.

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

In this article, I will discuss **Error Pages Based on Status Codes in ASP.NET Core MVC** Applications with Examples. Please read our previous article discussing [**Handling Non-Success HTTP Status Codes in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/handling-non-success-http-status-codes-in-asp-net-core-mvc/) Application.

**HTTP Status Codes:**

HTTP Status Codes are used to indicate the outcome of an HTTP Request. Non-success HTTP status codes indicate errors or specific conditions that have occurred during the request-response cycle. In ASP.NET Core MVC, you can handle these non-success status codes to provide meaningful responses to the client. The following are some of the standard non-success HTTP status codes in ASP.NET Core MVC:

* **400 Bad Request:** This status code is used when the server cannot understand the request due to invalid syntax or client-side errors.
* **401 Unauthorized:** This status code indicates the request lacks valid authentication credentials.
* **403 Forbidden:** This status code is used when the server understands the request but refuses to fulfill it, often due to insufficient permissions.
* **404 Not Found:** This status code indicates that the requested resource could not be found on the server.
* **500 Internal Server Error:** This status code is a generic error message indicating that something has gone wrong on the server.
* **503 Service Unavailable:** This status code is used when the server is temporarily unavailable to handle the request due to being overloaded or undergoing maintenance.

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

In ASP.NET Core MVC, we can create custom error pages based on Non-Success HTTP status codes to provide a more user-friendly message when errors occur. Let us proceed and see how to create and return Custom Error Pages based on Non-Success HTTP Status Codes like 401, 404, 500, etc., in an ASP.NET Core MVC application.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=e8ad602c-fb60-4281-658f-9b0c1adca224&d_id=118560&imp_id=1000625090849045&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

As we already discussed in our previous article, we can configure the Error Handling Middleware (**UseStatusCodePagesWithReExecute** or **UseStatusCodePagesWithRedirects** middleware in the Program.cs file) to handle the Non-Success HTTP status code and render the appropriate custom error page.

**Create Custom Error Views:**

First, create specific error views for specific status codes you want to handle. These views will be displayed to the user when the corresponding HTTP status code is returned.

For example, you can create views like **PageNotFoundError.cshtml** for handling “Not Found” errors (HTTP status code 404) and **InternalServerError.cshtml** for handling server errors (HTTP status code 500). You must place these views inside the **Views/Shared** directory.

**Views/Shared/PageNotFoundError.cshtml**  
**Views/Shared/InternalServerError.cshtml**  
**Views/Shared/UnauthorizedError.cshtml**  
**Views/Shared/GenericError.cshtml**

**Configuring Error Handling Middleware in ASP.NET Core:**

In the Program.cs file, we need to configure the Error Middleware to return Custom Rrror Mages based on HTTP Status Codes. To use the UseStatusCodePagesWithReExecute middleware, add the following code to the Configure method:

**app.UseStatusCodePagesWithReExecute(“/Error/{0}”);**

This configuration tells ASP.NET Core to re-execute the request with the “Error” Route and pass the status code as a parameter to the action method. To use the UseStatusCodePagesWithRedirects middleware, you can use the following code:

**app.UseStatusCodePagesWithRedirects(“/Error/{0}”);**

This configuration will redirect the user to the Error Route with the corresponding status code in the URL. Let us proceed and implement this example step by step:

**Creating Error Controller:**

Create a controller with the name **ErrorController** within the **Controllers** folders. Once you create the Error Controller, please copy and paste the following code. When an error status code is encountered, the Index action in the ErrorController is executed with the status code as a parameter, and based on the status code, a specific error view is returned.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**Route**(**"Error/{statusCode}"**)]**

**public** IActionResult Index**(int** statusCode**)**

**{**

Response.Clear**()**;

Response.StatusCode = statusCode;

**switch** **(**statusCode**)**

**{**

**case** 401:

**return** View**(**"UnauthorizedError"**)**;

**case** 404:

**return** View**(**"PageNotFoundError"**)**;

**case** 500:

**return** View**(**"InternalServerError"**)**;

**default**:

**return** View**(**"GenericError"**)**;

**}**

**}**

**}**

**}**

**Creating Error Views:**

Next, add the following views to the Views/Shared folder.

**PageNotFoundError.cshtml**

@{

Layout = null;

}

<!DOCTYPE html>

**<html>**

**<head>**

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

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

**</head>**

**<body>**

**<hgroup>**

**<h1** style="color:red"**>**Page Not Found Error**</h1>**

**<h2** style ="color:red"**>**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** style="color:red"**>**Unauthorized Error**</h1>**

**<h2** style="color:red"**>**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** style="color:red"**>**Internal Server Error**</h1>**

**<h2** style="color:red"**>**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** style="color:red"**>**Generic Error**</h1>**

**<h2** style="color:red"**>**An unknown error has occurred. We are working on it. Please try after some time**</h2>**

**</hgroup>**

**</body>**

**</html>**

**Configuring Middleware in Program.cs:**

**namespace** *FiltersDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllersWithViews**()**;

var app = builder.Build**()**;

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseDeveloperExceptionPage**()**;

**}**

**else**

**{**

app.UseStatusCodePagesWithReExecute**(**"/Error/{0}"**)**;

**}**

app.UseHttpsRedirection**()**;

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

app.MapControllerRoute**(**

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}"**)**;

app.Run**()**;

**}**

**}**

**}**

**The following line of code configures the Middleware:**

**app.UseStatusCodePagesWithReExecute(“/Error/{0}”);**

**Modify Home Controller:**

Next, modify the Home Controller as follows. Here, you can see that we have created different action methods to demonstrate the different types of Non-Success HTTP Status Codes. SomeAction1 will throw a 400 Bad Request HTTP Status Code. SomeAction2 will throw a 401 Unauthorized HTTP Status Code. SomeAction3 will throw a 403 Forbidden HTTP Status Code. SomeAction4 will throw a 404 Not Found HTTP Status Code. SomeAction5 will throw a 500 Internal Server Error HTTP Status Code. SomeAction6 will throw a 503 Service Unavailable Error HTTP Status Code.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

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

**{**

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

**}**

//400 Bad Request

**public** IActionResult SomeAction1**()**

**{**

var someConditionIsNotMet = **true**;

**if** **(**someConditionIsNotMet**)**

**{**

**return** new StatusCodeResult**(**400**)**;

**}**

// Other logic

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

**}**

//401 Unauthorized

**public** IActionResult SomeAction2**()**

**{**

var IsAuthenticated = **false**;

**if** **(**!IsAuthenticated**)**

**{**

**return** Unauthorized**()**; // This will return a 401 status code

**}**

// Other logic

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

**}**

//403 Forbidden

**public** IActionResult SomeAction3**()**

**{**

var UserHasPermissionToAccessResource = **false**;

**if** **(**!UserHasPermissionToAccessResource**)**

**{**

**return** new StatusCodeResult**(**403**)**;

**}**

// Other logic

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

**}**

//404 Not Found

**public** IActionResult SomeAction4**()**

**{**

var requestedResourceNotFound = **true**;

**if** **(**requestedResourceNotFound**)**

**{**

**return** NotFound**()**;

**}**

// Other logic

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

**}**

//500 Internal Server Error

**public** IActionResult SomeAction5**()**

**{**

**try**

**{**

// Some code that might throw an exception

// ...

**throw** new Exception**(**"Some Exception Occurred"**)**;

**return** Ok**()**; // If successful

**}**

**catch** **(**Exception ex**)**

**{**

// Log the exception

**return** new StatusCodeResult**(**500**)**;

**}**

**}**

//503 Service Unavailable

**public** IActionResult SomeAction6**()**

**{**

var isServiceUnavailable = **true**;

**if** **(**isServiceUnavailable**)**

**{**

**return** new StatusCodeResult**(**503**)**;

**}**

// Other logic

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

**}**

**}**

**}**

Now, run the application and access the Home Controller Action Methods. You should see the respective error pages based on the Status Code.

In the next article, I will discuss the [**Result Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/result-filters-in-asp-net-core-mvc/) Application. In this article, I try to explain Error Pages Based on Status Codes in ASP.NET Core MVC Applications with Examples. I hope you enjoy this Error Pages Based on Status Codes in the ASP.NET Core MVC article.

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

In this article, I will discuss **Result Filters in ASP.NET Core MVC** Applications with Examples. Please read our previous article discussing [**Error Pages Based on Status Code in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/error-pages-based-on-status-code-in-asp-net-core-mvc/) Application.

**What are Result Filters in ASP.NET Core MVC?**

In ASP.NET Core MVC, Result Filters are a specific type of filter that runs after the action method has been executed but before the result is processed and sent to the client. This means that with Result Filters, we can modify the view or the result data before it is returned to the client.

**How Do We Create a Result Filter in ASP.NET Core MVC?**

In ASP.NET Core MVC, we can create a Custom Result Filter in two ways: First, by creating a class implementing the **IResultFilter**interface and providing implementations for **[OnResultExecuting] and [OnResultExecuted]** methods. Second, a class can be created by inheriting the **ResultFilterAttribute**class and overriding the **[OnResultExecuting] and [OnResultExecuted]** methods.

* **Before Result Execution (OnResultExecuting):** This method is executed just before the action result is executed. You can use this method to modify the action result or insert additional processing before the result is handled.
* **After Result Execution (OnResultExecuted):** This method is called after the action result execution is completed. It allows us to modify the response, log information, handle exceptions, or perform other operations after the result has been processed.

**Note:**If you need to perform asynchronous operations within the Custom Result Filter, you should implement the **IAsyncResultFilter** interface or **AsyncResultFilterAttribute**and need to implement or override the OnResultExecutionAsync method. First, I will show you the example using **ResultFilterAttribute**, and then I will show you another example using the **IResultFilter** interface.

**When Should We Use Result Filters in ASP.NET Core MVC?**

* **Modifying Results:** Result filters can modify or replace the result being executed. For example, you could change the view or data returned by an action based on certain conditions.
* **Logging:** They provide a convenient place to log the use of particular actions or results, such as response size or execution time.
* **Custom Headers:** Custom headers are added to the HTTP response based on certain conditions evaluated before or after the action result.

Advertisements

**Example to Understand Result Filter in ASP.NET Core MVC:**

Let us look at one example of understanding result filters in ASP.NET Core MVC. Imagine you are developing a web application that includes a feature to log the execution time of certain pages and modify the HTTP response based on the user role. For example, we are going to dynamically modify the view based on a query string (admin=true changes the result to “AdminView”).

**Define Result Filter:**

First, create a custom result filter that measures execution time and appends a custom header if the execution time exceeds a predefined threshold. So, create a class file named **CustomResultFilterAttribute.cs** within the Models folder and copy and paste the following code.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Diagnostics;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomResultFilterAttribute : ResultFilterAttribute

**{**

**private** Stopwatch \_timer;

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

**{**

// Initialize and start the timer

\_timer = Stopwatch.StartNew**()**;

// Add a custom header before executing the result

context.HttpContext.Response.Headers.Append**(**"X-Pre-Execute", "Header set before execution"**)**;

// Example: Modify the result based on authorization (dummy condition here)

**if** **(**context.HttpContext.Request.Query.ContainsKey**(**"admin"**)** && context.Result **is** ViewResult viewResult**)**

**{**

context.Result = new ViewResult

**{**

ViewName = "AdminView",

ViewData = viewResult.ViewData,

TempData = viewResult.TempData

**}**;

**}**

**base**.OnResultExecuting**(**context**)**;

**}**

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

**{**

\_timer.Stop**()**;

var actionName = context.ActionDescriptor.DisplayName;

var executionTime = \_timer.ElapsedMilliseconds;

var resultType = context.Result.GetType**()**.Name;

// Log details about the action execution

Debug.WriteLine**(**$"Action '{actionName}' executed in {executionTime} ms, resulting in {resultType}"**)**;

**base**.OnResultExecuted**(**context**)**;

**}**

**}**

**}**

**OnResultExecuting Method**

The OnResultExecuting method is called just before the action result is executed, i.e., before the framework writes the response. Here’s what happens in this method:

* **Timer Initialization:** A Stopwatch instance is created and started to measure the duration of the result’s execution. This is useful for performance monitoring.
* **Setting a Custom Header:** The HTTP response includes a custom header **X-Pre-Execute**. This header is set before the result is actually executed, allowing you to send additional information in the HTTP headers.
* **Modifying the Result Conditionally:** The method checks if the incoming request contains a specific query parameter (admin). If this condition is met, the method changes the view being returned. It sets the view to AdminView instead of the original view, effectively altering the output based on the request parameters.
* **Calling Base Method:** Finally, it calls base.OnResultExecuting(context) to ensure that any logic in the base class is also executed.

**OnResultExecuted Method**

The OnResultExecuted method is called right after the action result is executed, i.e., after the result has been processed and the response has likely been sent to the client. Here’s what happens in this method:

* **Stop Timer:** It stops the Stopwatch instance that was started in OnResultExecuting to get the total execution time.
* **Logging:** It logs details about the action, such as the action name, execution time, and the type of result. This is crucial for debugging and monitoring the behavior of your web application.
* **Calling Base Method:** It calls base.OnResultExecuted(context) to ensure that any logic in the base class is also executed.

**Modifying Home Controller:**

You can apply this filter to a specific action, a controller, or globally across all controllers. Here, we apply it to the Home Controller only. Next, modify the Home Controller as follows:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**CustomResultFilter**]**

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

**{**

**public** IActionResult Index**()**

**{**

// The view name can be dynamically changed based on the filter

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

**}**

**}**

**}**

**Index.cshtml**

Next, modify the **Index.cshtml** view as follows:

@{

ViewData["Title"] = "Home Page";

}

**<h1>**Normal User View**</h1>**

**AdminView.cshtml**

Next, add the **AdminView.cshtml** view within the Views/Home folder and then copy and paste the following code:

@{

ViewData["Title"] = "AdminView";

}

**<h1>**Admin Panel**</h1>**

**Running and Testing the Application**

**Access /Home/Index normally to see the “Index View”. I**

**Advertisements**

**Access /Home/Index?admin=true to trigger the admin view condition.**

In both requests, if you check the response headers, then you will see the custom header, which we set using the Result Filter as shown in the below image:

**Custom Result Filter using IAsyncResultFilter Interface:**

Now, let us rewrite the previous example using IAsyncResultFilter. We need to implement the IAsyncResultFilter interface and needs to provide the necessary asynchronous handling for the result execution and post-execution events. Create a class file named CustomResultFilter.cs and then copy and paste the following code:

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *System.Diagnostics;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomResultFilter : IAsyncResultFilter

**{**

**private** Stopwatch \_timer;

**public** **async** Task OnResultExecutionAsync**(**ResultExecutingContext context, ResultExecutionDelegate next**)**

**{**

// Initialize and start the timer

\_timer = Stopwatch.StartNew**()**;

// Add a custom header before executing the result

context.HttpContext.Response.Headers.Append**(**"X-Pre-Execute", "Header set before execution"**)**;

// Example: Modify the result based on authorization (dummy condition here)

**if** **(**context.HttpContext.Request.Query.ContainsKey**(**"admin"**)** && context.Result **is** ViewResult viewResult**)**

**{**

context.Result = new ViewResult

**{**

ViewName = "AdminView",

ViewData = viewResult.ViewData,

TempData = viewResult.TempData

**}**;

**}**

// Execute the result as planned

var executedContext = **await** next**()**;

// Stop the timer after the result is executed

\_timer.Stop**()**;

var actionName = context.ActionDescriptor.DisplayName;

var executionTime = \_timer.ElapsedMilliseconds;

var resultType = executedContext.Result.GetType**()**.Name;

// Log details about the action execution

Debug.WriteLine**(**$"Action '{actionName}' executed in {executionTime} ms, resulting in {resultType}"**)**;

**}**

**}**

**}**

**Code Explanation:**

* **Initialization of Stopwatch:** A Stopwatch is instantiated and started to measure the time taken to execute the action result. This is useful for performance monitoring.
* **Setting Custom Header:** Before the action result is executed, a custom header (X-Pre-Execute) is added to the HTTP response. This could be used for various purposes, such as providing metadata about the response or for debugging purposes.
* **Conditional Result Modification:** The method checks if the HTTP request contains a specific query parameter (admin). If this condition is met, and the current result is a ViewResult (which typically renders a view), it modifies the result to change the view that will be rendered.
* **Execution of Result:** The await next() call is very important. This line hands over control to the next filter in the pipeline, or if there are no further filters, it executes the action result. This is an asynchronous operation and the method awaits its completion before proceeding. After await next() completes, it returns a ResultExecutedContext, which contains details about the executed action result.
* **Stopwatch Stopping and Logging:** Once the action result has been executed and the control returns to the filter, the stopwatch is stopped. The elapsed time (how long the result took to execute), the name of the executed action, and the type of result that was executed are logged using Debug.WriteLine().

**Modifying the Home Controller:**

Next, modify the Home Controller as follows. As the CustomResultFilter is not an attribute, we cannot directly apply it to the Controller or Action method level. Here, we are using TypeFilter to apply the Custom Result filter at the controller level.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**TypeFilter**(**typeof**(**CustomResultFilter**))]**

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

**{**

**public** IActionResult Index**()**

**{**

// The view name can be dynamically changed based on the filter

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

**}**

**}**

**}**

With these changes, run the application, and it should work as expected.

In the next article, I will discuss [**Custom Result Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/custom-result-filter-in-asp-net-core-mvc/) Application. In this article, I explain Result Filters in ASP.NET Core Applications with Examples. I hope you enjoy this Result Filters in ASP.NET Core article.

Custom Result Filter in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

**Custom Result Filter in ASP.NET Core MVC**

In this article, I will discuss **How to Create a Custom Result Filter in an ASP.NET Core MVC** Application with Real-Time Examples. Please read our previous article discussing [**Result Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/result-filters-in-asp-net-core-mvc/) Application.

**Custom Result Filter in ASP.NET Core MVC**

In ASP.NET Core MVC, Result Filters are a specific type of filter that runs after the action method has been executed but before the result is processed and sent to the client, i.e., before or after executing action results. Action results are what an action method returns to generate a response. This could be a view, a file, a redirect, or a JSON result, among other things. Custom Result Filters in ASP.NET Core MVC can be useful for:

* Modifying or Replacing the Result.
* Adding HTTP Headers to the Response.
* Logging or Auditing Response Data.
* Handling Errors.
* Caching Responses.

**How Do We Create a Custom Result Filter in ASP.NET Core MVC?**

In ASP.NET Core, we can create a Custom Result Filter in two ways: First, we can create a class implementing the **IResultFilter** interface and providing implementations for the **OnResultExecuting** and **OnResultExecuted** methods. Second, we can create a class that inherits from the **ResultFilterAttribute** class and overrides the **OnResultExecuting** and **OnResultExecuted** methods. So, creating and using a Custom Result Filter in ASP.NET Core MVC involves the following steps:

* **Define the Filter:** Create a class that implements the **IResultFilter/IAsyncResultFilter** interface or inherits from the **ResultFilterAttribute** class. For synchronous operations, use IResultFilter, and for asynchronous, use IAsyncResultFilter.
* **Implement Required Methods:** For **IResultFilter**, you need to implement **OnResultExecuting** and **OnResultExecuted**. For **IAsyncResultFilter**, you need to implement **OnResultExecutionAsync**. If you are inheriting the **ResultFilterAttribute** class, you must override the **OnResultExecuting** and **OnResultExecuted** methods.
* **Apply the Filter:** Once the Custom Result Filter is created, apply it to actions or controllers using attributes or add it globally in the Program class.

**Approach 1: Inheriting from ResultFilterAttribute Class**

Create a Custom Result Filter by inheriting from the **ResultFilterAttribute** class and overriding its methods: **OnResultExecuting** and **OnResultExecuted**. So, create a class file named **CustomResultFilterAttribute.cs** and copy and paste the following code. In the example below, we log when the OnResultExecuting and OnResultExecuted methods are executed.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomResultFilterAttribute : ResultFilterAttribute

**{**

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

**{**

// This method is called before the result is executed.

LogMessage**(**"Executing custom result filter attribute before result execution.\n"**)**;

**}**

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

**{**

// This method is called after the result has been executed.

LogMessage**(**"Executing custom result filter attribute after result execution.\n"**)**;

**}**

**private** **void** LogMessage**(string** message**)**

**{**

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

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

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

**Applying the Custom Result Filter to an Action Method:**

Let us modify the Home Controller as follows to apply the Custom Result Filter on the Index Action method:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**CustomResultFilterAttribute**]**

**public** IActionResult Index**()**

**{**

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

**}**

**}**

**}**

With the above code in place, run the application, navigate to the Index action method, and verify the **Log.txt** file, which should be generated within the Log folder.

**Registering the Filter Globally**

We can also register the filter globally to apply to all actions in the application. This can be done in the **Program.cs** class file as follows:

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomResultFilterAttribute**())**;

**})**;

**Approach 2: Implementing IResultFilter (Synchronous Result Filter)**

Let’s say we want to create a result filter that logs the type of ActionResult returned by our action methods. So, create a Custom Result Filter by implementing the **IResultFilter** interface. You need to implement two methods: **OnResultExecuting** and **OnResultExecuted**. These methods are called before and after the result is executed. So, modify the **CustomResultFilter.cs** as follows.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=3be99168-6c24-48c9-69c4-ec894f7aacc2&d_id=118560&imp_id=4508321446838996&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomResultFilter : IResultFilter

**{**

**public** **void** OnResultExecuting**(**ResultExecutingContext context**)**

**{**

// This method is called before the result is executed.

// Log information before the action result starts executing

LogMessage**(**$"Result type is about to be executed: {context.Result.GetType().Name}\n"**)**;

LogMessage**(**"Executing custom result filter attribute before result execution.\n"**)**;

**}**

**public** **void** OnResultExecuted**(**ResultExecutedContext context**)**

**{**

// This method is called after the result has been executed.

// Log information after the action result has finished executing

LogMessage**(**$"Result type has been executed: {context.Result.GetType().Name}\n"**)**;

LogMessage**(**"Executing custom result filter attribute after result execution.\n"**)**;

**}**

**private** **void** LogMessage**(string** message**)**

**{**

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

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

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

Register the CustomResultFilter as a service in your application’s **Program.cs** file. You can specify the filter’s scope (e.g., Transient, Scoped, Singleton) based on your requirements.

**builder.Services.AddScoped<CustomResultFilter>();**

Apply the CustomResultFilter to your controller action methods using the [ServiceFilter] attribute or add it globally in the Program.cs file. For example, apply on to the Action method as follows:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**ServiceFilter**(**typeof**(**CustomResultFilter**))]**

**public** IActionResult Index**()**

**{**

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

**}**

**}**

**}**

**Adding Globally:**

Adding the following code in the Program.cs class file:

//Adding Custom Result Filter Globally

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomResultFilterAttribute**())**;

**})**;

builder.Services.AddScoped**<**CustomResultFilter**>()**;

**Approach 3: Implementing IAsyncResultFilter (Asynchronous Result Filter)**

Assume you want to compress the response’s content asynchronously in certain conditions. You would implement an asynchronous result filter to compress the response stream.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *System.IO.Compression;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CompressResultFilter : Attribute, IAsyncResultFilter

**{**

**public** **async** Task OnResultExecutionAsync**(**ResultExecutingContext context, ResultExecutionDelegate next**)**

**{**

// Check if the response should be compressed based on some logic

var shouldCompress = ShouldCompressResponse**(**context**)**;

**if** **(**shouldCompress**)**

**{**

var originalBodyStream = context.HttpContext.Response.Body;

**using** **(**var compressedStream = new MemoryStream**())**

**{**

**using** **(**var compressionStream = new GZipStream**(**compressedStream, CompressionMode.Compress, leaveOpen: **true))**

**{**

context.HttpContext.Response.Body = compressionStream;

// Execute the result (action filters, action, result filters, result)

**await** next**()**;

// Flush the remaining data

**await** compressionStream.FlushAsync**()**;

**}**

// Copy the compressed data to the original stream

compressedStream.Seek**(**0, SeekOrigin.Begin**)**;

**await** compressedStream.CopyToAsync**(**originalBodyStream**)**;

**}**

context.HttpContext.Response.Body = originalBodyStream;

context.HttpContext.Response.Headers.Add**(**"Content-Encoding", "gzip"**)**;

**}**

**else**

**{**

// If we don't want to compress, just call the next delegate/middleware in the pipeline

**await** next**()**;

**}**

**}**

**private** **bool** ShouldCompressResponse**(**ResultExecutingContext context**)**

**{**

// Your logic to determine if the response should be compressed

**return** **true**; // For this example, we're just compressing every response

**}**

**}**

**}**

In this asynchronous result filter example, the filter checks if the response should be compressed. If so, it sets up a new response body stream that compresses the content. After executing the action result (with await next();), it copies the compressed content to the original response stream and sets the appropriate HTTP header.

**Then, apply this filter:**

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**CompressResultFilter**]**

**public** IActionResult Index**()**

**{**

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

**}**

**}**

**}**

In the next article, I will discuss [**Response Caching in ASP.NET Core**](https://dotnettutorials.net/lesson/response-caching-in-asp-net-core/) Applications. In this article, I try to explain **Custom Result Filter in ASP.NET Core MVC** Application with Examples. I hope you enjoy this Custom Result Filter in ASP.NET Core MVC article.

**Response Caching in ASP.NET Core**

In this article, I will discuss **Response Caching (ResponseCacheAttribute) in ASP.NET Core** Application with Examples. Please read our previous article discussing [**Custom Result Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/custom-result-filter-in-asp-net-core-mvc/) Application. At the end of this article, you will understand the following pointers:

* **What is Caching?**
* **Types of Caching in ASP.NET Core**
* **What is Response Caching in ASP.NET Core?**
* **How Response Caching Works in ASP.NET Core?**
* **Response Caching Examples: Basic, No Caching, Caching with VarByHeader, Location, and VarByQueryKeys.**
* **Custom Cache Profiles in ASP.NET Core Response Caching.**
* **Benefits of Response Caching in ASP.NET Core?**

**What is Caching?**

Caching is a technique for storing frequently accessed data in a temporary storage area that can be quickly retrieved. This will improve the overall performance of the application by reducing the need to fetch the same data repeatedly from the database or other storage mediums.

**Types of Caching in ASP.NET Core:**

ASP.NET Core Supports Several Types of Caching Mechanisms. They are as follows:

* **In-Memory Caching:** This is the simplest form of caching, suitable for a single server. It stores data in the Web Server’s Main Memory. It’s fast and suitable for data that doesn’t consume too much memory and doesn’t need to persist beyond the lifetime of the web server process. It is suitable for storing small amounts of data.
* **Distributed Caching:** This is ideal for applications running in multi-server or load-balancing environments where data needs to be shared across multiple servers. It involves storing data in an external system such as Redis, SQL Server, NCache, etc. It is more complex than in-memory caching but necessary for large-scale applications to ensure consistency across sessions and requests.
* **Response Caching:** Response Caching refers to the process of storing the output of a request-response cycle in the cache so that future requests for the same resource can be served from the cache instead of regenerating the response from scratch. This technique can significantly improve a web application’s performance, especially for resources that are expensive to generate and don’t change often.

**Note:** In this session, we will discuss Response Caching. The rest will be discussed in our upcoming sessions.

**What is Response Caching?**

When a web application receives a request, it often performs several operations, such as database queries, complex calculations, and template rendering, to generate the response. These operations can be time-consuming. If the generated response is the same for each request, it makes sense to store that response once it is generated and then serve the stored response for future requests without regenerating the response again by doing all the processing. This is called Response Caching in ASP.NET Core.

Advertisements

**How Response Caching Works in ASP.NET Core?**

Please look at the following diagram to understand how response caching works in the ASP.NET Core Application.

Here’s how response caching works in a web context, particularly in ASP.NET Core Web Application:

**Initial Request:**

* A client makes a request to the server for a specific resource.
* The server processes the request, generates the response, and returns the response to the client who initially made the request.
* Along with the response, the server includes a **Cache-Control HTTP header** to indicate that the response can be cached and to specify the caching behavior.

**Subsequent Requests:**

* When the client (i.e., browser) needs the same resource again, it first checks the cache.
* Suppose the cached response is available and fresh (i.e., within the period specified by the Cache-Control header). In that case, the client serves it directly from the cache instead of requesting it from the server again.

**What is Proxy Cache in ASP.NET Core?**

In the context of ASP.NET Core, **Proxy Caches** refer to a caching mechanism where a Caching Server (the Proxy Server) is placed between the Client (usually a Web Browser) and the Web Server hosting the ASP.NET Core application. This Proxy Server caches responses from the Web Server, which can then be served quickly to subsequent requests for the same resource. This can significantly improve the performance of web applications by reducing the load on the web server and decreasing response times for end-users.

**Server-Side:**

* In ASP.NET Core, you can use the**[ResponseCache]** attribute to specify caching behavior for controller actions. This attribute allows you to set various parameters, such as duration, location (client, server, or both), and whether the cache should consider query string values.
* ASP.NET Core includes **UseResponseCaching** middleware for caching responses on the server. This middleware needs to be configured in the Program.cs file.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=9dd0f0e9-d869-4999-598a-a3aa2e1ee69e&d_id=118560&imp_id=7770584804858214&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**Client-Side:**

* Browsers respect the **Cache Headers** and store the responses in their cache based on their directives, i.e., no-cache, no-store, public, private, etc.
* Proxy Caches can do the same, helping reduce the load on the server and speeding up response times for users behind the proxy.

**How Do We Implement Response Cacheing in ASP.NET Core?**

To implement Response Caching, we need to use the ResponseCacheAttribute in ASP.NET Core. This attribute allows developers to control how responses from Web Applications are cached. It can be applied to controller actions or entire controllers and specifies the parameters necessary for setting appropriate HTTP headers for caching responses.

The ResponseCache Attribute itself does not cache content. Instead, it sets the appropriate HTTP headers that control the caching behavior. The ResponseCache Attribute has the following properties:

* **Duration(int):** Specifies the time, in seconds, for which the response should be cached. This sets the max-age directive of the Cache-Control header.
* **Location (ResponseCacheLocation):** This setting modifies the Cache-Control header and determines the location where the response can be cached. Options include Any, Client, and None.
* **NoStore (bool)**: When set to true, this property indicates that the response should not be cached (Cache-Control: no-store). It takes precedence over other properties.
* **VaryByHeader (string):** Specifies a header name to vary the cached response by (e.g., User-Agent).
* **VaryByQueryKeys(string):** This option allows you to vary the response cache based on the query string parameters. It’s useful for caching different responses for different query string values.
* **CacheProfileName(string):** This setting refers to a cache profile defined in the application’s configuration, which allows for centralized cache policy settings.

**Examples to Understand Response Caching in ASP.NET Core:**

Let’s look at examples to understand how to use the ResponseCacheAttribute in an ASP.NET Core Web API application to implement Response Caching. We will test the functionality using **Postman** and **ASP.NET Core Web API**. It will not work as expected if you test the same using Browser. This is because each browser manages the Caching in a different way.

**Configure the Response Caching Middleware and Services**

Before using the ResponseCacheAttribute, ensure that the response caching middleware is added to the request pipeline in the Program.cs class file. We need to add the **AddResponseCaching** services and **UseResponseCaching** middleware components to the request processing pipeline. So, modify the **Program.cs** class file as follows:

**namespace** *ResponseCachingDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllers**()**;

//Adding Response Caching Service

builder.Services.AddResponseCaching**()**;

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer**()**;

builder.Services.AddSwaggerGen**()**;

var app = builder.Build**()**;

// Configure the HTTP request pipeline.

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseSwagger**()**;

app.UseSwaggerUI**()**;

**}**

app.UseHttpsRedirection**()**;

//Adding Response Caching Middleware Components

app.UseResponseCaching**()**;

app.UseAuthorization**()**;

app.MapControllers**()**;

app.Run**()**;

**}**

**}**

**}**

**Example to Understand Basic Response Caching**

Next, modify the Home Controller as follows. Here, we have applied the Index action method with the **[ResponseCache(Duration = 60)]** Attribute, which will cache the response for 60 seconds. This means that once a response is generated, it will be reused for subsequent requests for the next 60 seconds.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

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

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

**{**

**return** $"Response Generated at: {DateTime.Now}";

**}**

**}**

**}**

Now, run the application and access the endpoint **/api/home/index using Postman,** and you will see the following output. If you access the same endpoint within 60 seconds, you will also see the same output.

Further, if you check the Response headers, you will see the **Cache-Control** header whose value is set to **public,max-age=60**, as shown in the image below.

**Cache-Control: public, max-age=60:** These settings instruct all caches (like Browser, Postman, Fiddler, Swagger, or Proxy Server) to store and reuse the response for 60 seconds without revalidating it. After 60 seconds, the cached response is considered invalid, and a new copy needs to be fetched from the original server.

Advertisements

**Example: No Caching**

In the following example, we have set the **NoStore** property of the **ResponseCache** Attribute to **true**, which tells the browser not to cache the response.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

**[**ResponseCache**(**Duration = 60, NoStore = **true)]**

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

**{**

**return** $"Response Generated at: {DateTime.Now}";

**}**

**}**

**}**

Now, run the application and access the endpoint **/api/home/index**using Postman, and you will see the caching is not working. Every time you hit the endpoint, you will get a different response even though we have set the cache duration to 60 seconds. This is because we have set the NoStore Attribute value to true, which means caching is disabled. Now, if you check the Response headers, you will see the **Cache-Control** header whose value is set to **no-store**, as shown in the image below.

**Cache-Control: no-store:** This setting instructs all caches (like Browser, Postman, Fiddler, Swagger, or Proxy Server) not to store any part of the response in the cache. The no-store directive ensures that no copy of the response is saved in any cache. Each time the resource is needed, it must be fetched directly from the server. This guarantees that the information is retrieved in its current state without the risk of serving outdated or potentially compromised data from a cache.

**Example to Understand Response Caching with VaryByHeader**

In ASP.NET Core, the VaryByHeader Parameter in Response Caching specifies a set of Request Headers (User-Agent, Accept-Language, or any Custom Header) that should trigger the cache to store multiple responses for the same URL based on their values. This is useful when responses vary not only by URL but also by request headers.

When we configure the Response Cache with the VaryByHeader parameter, the caching mechanism considers the URL and the specified header values to determine the cache key. That means different responses are cached and retrieved based on variations in the specified headers. For a better understanding, please modify the Home Controller as follows. Here, we are using the User-Agent header.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

**[**ResponseCache**(**Duration = 60, VaryByHeader = "User-Agent"**)]**

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

**{**

**return** $"Response Generated at: {DateTime.Now}";

**}**

**}**

**}**

Now, run the application and access the endpoint **/api/home/index using Postman and Swagger**. You will get a different response for each client. This is because the User-Agenet header for Swagger and Postman is going to be different. This is also true in the case of Browsers. Each Web Browser has a different User Agent. When the User-Agent is changed, it will fetch the data from the server. Now, if you check the Response header, then you will see along with the Cache-Control header, the Server is also adding the Vary header in the Response, as shown in the below image:

Advertisements

**Example to Understand Response Caching with Location**

By default, Caching is saved on both the client and server sides. Suppose we want to store the cache response only on the client side; then, we need to use the Location property. Using the Location property of the Response Cache Attribute, we can specify where we want to store the response cache using the following **ResponseCacheLocation** enum.

To store the cache on the browse only, we need to use the Client Option. The **ResponseCacheLocation.Client** specifies that the response should be cached only on the client side, and in this case, it will set the **Cache-Control header to Private**. So, modify the Home Controller as follows:

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

**[**ResponseCache**(**Duration = 60, Location = ResponseCacheLocation.Client, VaryByHeader = "User-Agent"**)]**

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

**{**

**return** $"Response Generated at: {DateTime.Now}";

**}**

**}**

**}**

Now, you can test the functionality using Swagger, and you should see the following output. It will not work on Postman.

Please check the following Key Points of the behavior of Swagger and Postman:

**Swagger Behavior:**

When testing APIs, Swagger UI generally respects the HTTP caching headers sent by the server. This means that if your ASP.NET Core application sends a response with Cache-Control set to private, max-age=300, Swagger UI will cache the response as instructed for the duration specified. Subsequent requests to the same endpoint (within the cache duration) might not hit the server if the cached response is deemed valid.

**Postman Behavior:**

Postman, on the other hand, does not automatically cache responses like a typical web browser would, despite receiving the same caching headers. In Postman, every time you hit the “Send” button, a new request is made to the server, ignoring any previous response cache headers. This is intentional, as Postman is designed for testing and development, where you often need to see live server responses without caching interference.

**Cache-Control: Private:** The Cache-Control header with the value private is used in Response Caching to indicate that the response is specific to a single user and should not be stored by shared caches, such as proxy servers. It can only be stored in the private cache of the user’s browser.

**Example to Understand Response Caching with VaryByQueryKeys**

In ASP.NET Core, the VaryByQueryKeys Property in Response Caching is used to specify that the response cache should vary based on the values of specified query string parameters. This is useful when a URL serves different data based on query string parameters. For example, if you have a URL that can return different results based on a query string like ?page=1 or ?page=2, you can use VaryByQueryKeys to cache these responses separately. For a better understanding, please modify the Home Controller as follows:

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

**[**ResponseCache**(**Duration = 60, VaryByQueryKeys = new**[]** **{** "page" **})]**

**public** **string** Index**(int** page**)**

**{**

**return** $"Response Generated at: {DateTime.Now}, for Page Number: {page}";

**}**

**}**

**}**

In this case, the cached output of the Index action method would differ depending on the value of the page query string parameter. This means if a user requests **Index?page=1** and another user requests **Index?page=2**, they will each get a different cached response specific to the page they requested.

**Example to Understand Custom Cache Profile with Response Caching**

In ASP.NET Core, Custom Cache Profiles allows us to define Reusable Caching Settings that can be applied across multiple actions or controllers. These profiles are defined in the application’s configuration, typically in the **Program.cs** class file, and can then be referenced by its name in controllers. Let us modify the Program class as follows.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

builder.Services.AddControllers**(**options =**>**

**{**

//Creating Custom Cache Profiles

options.CacheProfiles.Add**(**"Default60", new CacheProfile**()**

**{**

Duration = 60,

Location = ResponseCacheLocation.Any

**})**;

options.CacheProfiles.Add**(**"NoCache", new CacheProfile**()**

**{**

Location = ResponseCacheLocation.None,

NoStore = **true**

**})**;

**})**;

//Adding Response Caching Service

builder.Services.AddResponseCaching**()**;

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer**()**;

builder.Services.AddSwaggerGen**()**;

var app = builder.Build**()**;

// Configure the HTTP request pipeline.

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseSwagger**()**;

app.UseSwaggerUI**()**;

**}**

app.UseHttpsRedirection**()**;

//Adding Response Caching Middleware Components

app.UseResponseCaching**()**;

app.UseAuthorization**()**;

app.MapControllers**()**;

app.Run**()**;

**}**

**}**

**}**

**Here, we have configured two cache profiles (Default60 and NoCache), as shown in the below image:**

**Using Cache Profile in Response Cache Attribute:**

Once you define the Cache Profiles, they can be used within controller actions using the **CacheProfileName** Property of the **ResponseCache** attribute by referring to the profile name. So, modify the Home Controller as follows. Here, we have specified the Default60 Cache profile within the Index Action method and NoCache with the Privacy action method.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *ResponseCachingDemo.Controllers*

**{**

**[**Route**(**"api/[controller]/[action]"**)]**

**[**ApiController**]**

**public** **class** HomeController : ControllerBase

**{**

**[**HttpGet**]**

**[**ResponseCache**(**CacheProfileName = "Default60"**)]**

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

**{**

**return** $"Index Response Generated at: {DateTime.Now}";

**}**

**[**HttpGet**]**

**[**ResponseCache**(**CacheProfileName = "NoCache"**)]**

**public** **string** Privacy**()**

**{**

**return** $"Privacy Response Generated at: {DateTime.Now}";

**}**

**}**

**}**

Now, run the application and access the above endpoints using either Swagger or Postman. You will see that Response Caching works with the Index action method while it is disabled with the Privacy Action method.

**Benefits of Response Caching in ASP.NET Core MVC**

The following are some of the key benefits of using Response Caching in ASP.NET Core MVC:

* **Improved Performance:** By storing the output of actions, Response Caching reduces the time and resources needed to generate responses on subsequent requests. This can lead to faster page load times for users, as the server can serve cached content instead of regenerating it each time.
* **Reduced Server Load:** Caching can dramatically reduce the workload on the server. When content is served from the cache, fewer resources are consumed, which means your server can handle more users and requests with the same hardware.
* **Enhanced User Experience:** Faster response times generally lead to a better user experience.
* **Customizable Caching Strategies:** ASP.NET Core offers a variety of caching strategies, such as caching by duration, parameters, headers, etc. This flexibility allows developers to optimize caching based on the application’s needs and behaviors.

**Note:** You need to remember that while Response Caching can significantly improve your application’s performance, it may not be suitable for all scenarios. Caching can lead to incorrect or undesirable behavior, such as serving user-specific data that should not be cached or dealing with rapidly changing content.

In the next article, I will discuss the [**Authorization Filter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/authorization-filters-in-asp-net-core-mvc/) Application. In this article, I try to explain **Response Caching in ASP.NET Core Application** with Examples. I hope you enjoy this Response Caching in ASP.NET Core article.

Authorization Filters in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

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

In this article, I will discuss the **Authorization Filter in ASP.NET Core MVC** Application with Examples. Please read our previous article discussing [**Response Caching in ASP.NET Core**](https://dotnettutorials.net/lesson/response-caching-in-asp-net-core/) Applications. As part of this article, we will discuss the following pointers in detail.

1. **What is the Authorization Filter in ASP.NET Core MVC?**
2. **What is Authentication?**
3. **What is Authorization?**
4. **Examples to Understand Authorization Filter in ASP.NET Core MVC**
5. **Custom Authorization Filter in ASP.NET Core MVC**
6. **When Should We Use Authorization Filter in ASP.NET Core MVC?**

**What is the Authorization Filter in ASP.NET Core MVC?**

In ASP.NET Core MVC, the Authorization Filter allows us to apply authorization rules to controllers and actions within our application. Authorization Filters in ASP.NET Core are responsible for checking whether a user is allowed to perform an action or access a resource. These filters run before the action method is executed, ensuring the user has permission to access the method.

Authorization filters are executed after the routing but before model binding and other action filters. If the authorization fails (e.g., the user does not have the required permissions), the filter short-circuits the request, and the action method does not execute.

**What is Authentication?**

Authentication is a process that ensures and confirms a user’s identity. In other words, we can say that it is a process to validate someone against some data source. Let us understand Authentication from a layman’s point of view. For this, please have a look at the following diagram.

The above image shows the different sections of an IT Company, such as Reception, HR Section, Accounts Section, Server Room, etc. At the gate, we have biometrics to verify the employee. Suppose one employee comes. This biometrics checks the employee credentials against some data source, and if it is found that the employee is valid, it only allows entering into the campus. This is nothing but Authentication.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=a4acee39-ae5d-4d8d-78f0-b81d41e1c69f&d_id=118560&imp_id=3001952790860833&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**What is Authorization?**

Authorization is a mechanism that determines whether users can access a particular resource. The most important point that you need to remember is that authentication happens first and then only authorization. For a better understanding, please have a look at the following image.

As shown in the above image, once the employee is authenticated, he enters the Campus. Then, Authorization comes into the picture. Within the campus, which section he is allowed to enter is determined by the Authorization process. The Role of the Employee does this. If the Employee has list privileges, he may not allow each section. On the other hand, if the Employee has the highest privileges, he may be allowed to enter each section.

**Types of Authorization Filters in ASP.NET Core MVC:**

By default, in the ASP.NET Core MVC applications, all the action methods of all controllers can be accessed by both authenticated and anonymous users. However, if you want the action methods to be available only for authenticated and authorized users, you need to use the Authorization Filter in ASP.NET Core MVC. ASP.NET Core provides two built-in attributes,**[Authorize] and [AllowAnonymous]**, that can be used as filters.

* **Authorize Attribute:** The [Authorize] Attribute specifies that only authenticated users or users with certain roles or policies can access a particular action method or controller. The [Authorize] attribute specifies that the associated action method or controller requires the user to be authenticated.
* **AllowAnonymous Attribute:** The [AllowAnonymous] attribute specifies that an action method or controller should allow anonymous access, even when an [Authorize] attribute is applied at the controller or action level. In other words, it permits unauthenticated users to access the decorated resource, bypassing any authentication and authorization checks that might be in place at higher levels.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=a4acee39-ae5d-4d8d-78f0-b81d41e1c69f&d_id=118560&imp_id=1052947688873102&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**Examples to Understand Authorization Filter in ASP.NET Core MVC:**

Let us understand the Authorize and AllowAnonymous Filters with an example. First, create a new ASP.NET Core Application using the Model-View-Controller Project Template. Once you create the Project, then modify the Home Controller as follows. As you can see, we created the HomeController with three action methods, i.e., **NonSecureMethod**, **SecureMethod**, and **Login**. We want the SecureMethod to be accessed by authenticated users while the NonSecureMethod and Login methods to be accessed by anyone. Whenever an unauthenticated user wants to access the SecureMethod, we must redirect that user to the Login action method.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**public** **string** NonSecureMethod**()**

**{**

**return** "This method can be accessed by everyone as it is non-secure method";

**}**

**public** **string** SecureMethod**()**

**{**

**return** "This method needs to be access by authorized users as it SecureMethod";

**}**

**public** **string** Login**()**

**{**

**return** "This is the Login Page";

**}**

**}**

**}**

At this point, authenticated and anonymous users can access the SecureMethod and the NonSecureMethod methods using the following two URLs.  
**/Home/SecureMethod**  
**/Home/NonSecureMethod**

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

**using** *Microsoft.AspNetCore.Authorization;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**public** **string** NonSecureMethod**()**

**{**

**return** "This method can be accessed by everyone as it is non-secure method";

**}**

**[**Authorize**]** //Requires authentication for the SecureMethod

**public** **string** SecureMethod**()**

**{**

**return** "This method needs to be access by authorized users as it SecureMethod";

**}**

**public** **string** Login**()**

**{**

**return** "This is the Login Page";

**}**

**}**

**}**

Now run the application and navigate to **/Home/SecureMethod.**You will see the following Internal Server Error.

Instead of displaying the above error page, we need to redirect the user to the Login Page. In the later part of this article, I will show you how to create Custom Authentication Filters to achieve the same in ASP.NET Core MVC Applications.

**Can we apply the Authorize Attribute at the controller level?**

Applying the Authorize attribute at the controller level applies to all the action methods present within that controller. For a better understanding, please modify the Home Controller as follows. The Home Controller Action methods are now protected with the Authorize Attribute. So, only the authenticated users can now access the SecureMethod(), NonSecureMethod(), and Login action methods.

**using** *Microsoft.AspNetCore.Authorization;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**Authorize**]** //Requires authentication for the entire controller

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

**{**

**public** **string** NonSecureMethod**()**

**{**

**return** "This method can be accessed by everyone as it is non-secure method";

**}**

**public** **string** SecureMethod**()**

**{**

**return** "This method needs to be access by authorized users as it SecureMethod";

**}**

**public** **string** Login**()**

**{**

**return** "This is the Login Page";

**}**

**}**

**}**

Now, suppose you want to allow anonymous access to the NonSecureMethod and Login method of the Home Controller. In that case, you need to decorate the NonSecureMethod and Login method with the AllowAnonymous attribute, as shown below. The AllowAnonymous attribute in ASP.NET Core MVC is used to skip the authorization enforced by the Authorization Filter.

**using** *Microsoft.AspNetCore.Authorization;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**Authorize**]** //Requires authentication for the entire controller

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

**{**

**public** **string** SecureMethod**()**

**{**

**return** "This method needs to be access by authorized users as it SecureMethod";

**}**

**[**AllowAnonymous**]**

**public** **string** NonSecureMethod**()**

**{**

**return** "This method can be accessed by everyone as it is non-secure method";

**}**

**[**AllowAnonymous**]**

**public** **string** Login**()**

**{**

**return** "This is the Login Page";

**}**

**}**

**}**

**Note:** If both Authorize and AllowAnonymous Attributes are applied to an action method, then the AllowAnonymous attribute will take priority and be accessed by any user.

**Custom Authorization Filter in ASP.NET Core:**

We can also create Custom Authentication filters. To create Custom Authentication in ASP.NET Core MVC, we need to create a class implementing the **IAuthorizationFilter** or **IAsyncAuthorizationFilter** interface and provide implementations for the **OnAuthorization** method, where you need to write the custom authentication logic as per our business requirements.

**Example to Understand Custom Authorization Filter in ASP.NET Core**

In ASP.NET Core, we can create a Custom Authorization Filter to redirect the User to the login page when the user is not authenticated. Let us see how we can create a custom authorization filter to achieve this:

Create a class file named **CustomAuthorizationFilterAttribute.cs,** and then copy and paste the following code. As you can see, the following class is inherited from the **Attribute and IAuthorizationFilter**classes so that we can use this class as an Attribute. This class implements the **IAuthorizationFilter**interface and provides the implementation for the **OnAuthorization** method, where we have written our custom authorization logic.

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *System.Security.Claims;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomAuthorizationFilterAttribute : Attribute, IAuthorizationFilter

**{**

**public** **void** OnAuthorization**(**AuthorizationFilterContext context**)**

**{**

// Your custom authorization logic here

**if** **(**!IsAuthorized**(**context.HttpContext.User**))**

**{**

// If not authenticated, redirect to the login page

context.Result = new RedirectToRouteResult**(**new RouteValueDictionary

**{**

**{** "controller", "Home" **}**, // Change "Home" to your login controller name

**{** "action", "Login" **}** // Change "Login" to your login action method name

**})**;

**}**

**}**

**private** **bool** IsAuthorized**(**ClaimsPrincipal user**)**

**{**

// Check if the user is authenticated

// Implement your custom authorization logic here

// Check roles, claims, policies, or any other criteria

// Return true if authorized, false if not

**return** **false**; // For demonstration purposes

**}**

**}**

**}**

Now, we need to decorate the controller or action method where we need to implement our custom authorization logic. Let us decorate the SecureMethod with our Custom Authorization Filter Attribute. So, modify the Home Controller as follows:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Authorization;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**CustomAuthorizationFilterAttribute**]** // Apply the custom CustomAuthorizationFilterAttribute

**public** **string** SecureMethod**()**

**{**

**return** "This method needs to be access by authorized users as it SecureMethod";

**}**

**[**AllowAnonymous**]**

**public** **string** NonSecureMethod**()**

**{**

**return** "This method can be accessed by everyone as it is non-secure method";

**}**

**[**AllowAnonymous**]**

**public** **string** Login**()**

**{**

**return** "This is the Login Page";

**}**

**}**

**}**

With the above changes in place, visit the SecureMethod, and you will see that it will redirect to the Login Page.

**Creating a Custom Authorization Filter**

In the previous example, the **CustomAuthorizationFilterAttribute** class is inherited from the **Attribute** class, and that’s why we apply the **CustomAuthorizationFilterAttribute** directly to the action methods and controller. Let us remove the inherited **Attribute** class from the CustomAuthorizationFilterAttribute class definition, and let’s inherit from the **IAuthorizationFilter** interface. So, modify the CustomAuthorizationFilterAttribute class as follows:

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *System.Security.Claims;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomAuthorizationFilter : IAuthorizationFilter

**{**

**public** **void** OnAuthorization**(**AuthorizationFilterContext context**)**

**{**

// Your custom authorization logic here

**if** **(**!IsAuthorized**(**context.HttpContext.User**))**

**{**

// If not authenticated, redirect to the login page

context.Result = new RedirectToRouteResult**(**new RouteValueDictionary

**{**

**{** "controller", "Home" **}**, // Change "Account" to your login controller name

**{** "action", "Login" **}** // Change "Login" to your login action method name

**})**;

**}**

**}**

**private** **bool** IsAuthorized**(**ClaimsPrincipal user**)**

**{**

// Check if the user is authenticated

// Implement your custom authorization logic here

// Check roles, claims, policies, or any other criteria

// Return true if authorized, false if not

**return** **false**; // For demonstration purposes

**}**

**}**

**}**

**Register the Filter Globally, on a Controller, or an Action:**

Now, we can register the Custom Authorization Filter at 3 different places. We can register the filter globally, which means it will be applied to all controllers and actions in your application. To do so, you need to modify the MVC Service registration in your Program.cs class file as follows.

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomAuthorizationFilter**())**;

**})**;

**Using Authentication Filter at Controller and Action Method Level:**

You can also apply the Custom Authorization Filter to a specific controller or action method using the built-in ServiceFilter and TypeFilter Attribute. We use the TypeFilter attribute in the code below to specify the CustomAuthorizationFilter.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**TypeFilter**(**typeof**(**CustomAuthorizationFilter**))]**

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

**{**

**return** "String Data from Index Action Method";

**}**

**}**

**}**

**Custom Asynchronous Authorization Filter in ASP.NET Core MVC**

It is also possible to create the Custom Authorization Filter asynchronous. For this, our Custom Authorization Filter class needs to implement the IAsyncAuthorizationFilter interface and provide the implementation for the OnAuthorizationAsync method. This type of filter is suitable for performing asynchronous operations, such as database calls or any I/O-bound work, within your authorization logic.

**Create the Custom Asynchronous Authorization Filter:**

So, create a class file named CustomAsyncAuthorizationFilter.cs and copy and paste the following code. Here, you can see the CustomAsyncAuthorizationFilter class implement the IAsyncAuthorizationFilter interface and provide implementations for the OnAuthorizationAsync method where we have written the custom Authorization Logic:

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomAsyncAuthorizationFilter : IAsyncAuthorizationFilter

**{**

**public** **async** Task OnAuthorizationAsync**(**AuthorizationFilterContext context**)**

**{**

// Your asynchronous custom authorization logic here

**bool** isAuthorized = **await** CheckUserAuthorizationAsync**(**context**)**;

**if** **(**!isAuthorized**)**

**{**

// If not authenticated, redirect to the login page

context.Result = new RedirectToRouteResult**(**new RouteValueDictionary

**{**

**{** "controller", "Home" **}**, // Change "Account" to your login controller name

**{** "action", "Login" **}** // Change "Login" to your login action method name

**})**;

**}**

**}**

**private** **async** Task**<bool>** CheckUserAuthorizationAsync**(**AuthorizationFilterContext context**)**

**{**

// Implement your asynchronous authorization logic here

// For example, you can check user permissions, roles, etc., using async calls

**await** Task.Delay**(**1000**)**; // Simulate async work, like a database call

// Return true if authorized, false otherwise

**return** **false**;

**}**

**}**

**}**

**Register the Filter:**

You can register the filter globally or on specific controllers or actions:

**Globally:**

builder.Services.AddControllersWithViews**(**options =**>**

**{**

options.Filters.Add**(**new CustomAsyncAuthorizationFilter**())**;

**})**;

**On a Controller or Action:**

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**TypeFilter**(**typeof**(**CustomAsyncAuthorizationFilter**))]**

**public** **class** SomeController : Controller

**{**

// All actions in this controller will use the CustomAsyncAuthorizationFilter

**}**

**public** **class** AnotherController : Controller

**{**

**[**TypeFilter**(**typeof**(**CustomAsyncAuthorizationFilter**))]**

**public** **async** Task**<**IActionResult**>** SomeAction**()**

**{**

// This action will use the CustomAsyncAuthorizationFilter

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

**}**

**}**

**}**

**Note:** Once we discuss the ASP.NET Core Identity, you will see the real implementation with proper roles, claims, etc.

**When Should We Use Authorization Filter in ASP.NET Core MVC?**

Authorization filters in ASP.NET Core MVC should be used when you need to control access to specific controllers or actions within your web application based on authentication and authorization rules. The following are some common scenarios in which you should use authorization filters:

* **Securing Sensitive Data or Functionality:** Use authorization filters to protect endpoints that access sensitive data or perform sensitive operations, ensuring that only authorized users can access them.
* **Role-Based Access Control:** When you have different user roles in your application (like admin, user, and moderator) and want to restrict access to certain actions or controllers based on these roles.
* **Error Handling:** Authorization filters can also handle unauthorized access by redirecting users to login pages, displaying access-denied messages, or taking other appropriate actions.

In the next article, I will discuss [**Action Filters in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/action-filters-in-asp-net-core-mvc/) Application. In this article, I try to explain the **Authorization Filter in ASP.NET Core MVC Application** with Examples. I hope you understand the need for and use of Authorization Filters in ASP.NET Core MVC Applications.

Action Filters in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

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

In this article, I will discuss **Action Filters in ASP.NET Core MVC** Applications with Examples. Please read our previous article discussing [**Authorization Filters in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/authorization-filters-in-asp-net-core-mvc/) Applications.

**What are Action Filters in ASP.NET Core MVC?**

In ASP.NET Core MVC, action filters allow us to execute code before or after the execution of action methods in controllers. Action Filters are useful for handling cross-cutting concerns within an application, such as Logging, Authentication, Authorization, Caching, Exception Handling, etc.

Advertisements

**How Do We Create a Custom Action Filter in ASP.NET Core MVC?**

In ASP.NET Core, we can create a Custom Action Filter in two ways: First, by creating a class implementing the **IActionFilter** interface and providing implementations for [OnActionExecuting] and [OnActionExecuted] methods. Second, by creating a class inheriting from the **ActionFilterAttribute** class and overriding the [OnActionExecuting] and [OnActionExecuted] methods.

* **OnActionExecuting:** This method is called before the action method is executed
* **OnActionExecuted:** This method is called after the action method executes but before the result is processed.

**Note:** If you implement the IAsyncActionFilter interface, you need to provide implementations for the OnActionExecutionAsync method.

**Real-Time Example to Understand Action Filters in ASP.NET Core MVC:**

Let us create one Real-Time Application to understand the need and use of Custom Action Filters in ASP.NET Core MVC Application. We are going to develop one application where we will implement the following functionalities:

* **Logging:** Implementing logging of action method calls, parameters, execution times, etc.
* **Data Transformation:** Modifying the data passed to an action or returned from an action.
* **Validation:** Performing custom validation of action parameters or the request itself.
* **Error Handling:** Implementing custom error handling logic for actions.
* **Caching:** Implementing custom caching strategies for action results.

For each functionality, we will create a separate Custom Action Filter. I will also implement the Custom Action filter in different ways. I will also show you how to use Custom Services within the Custom Action filter. Also, I will discuss How to register the Custom Action Filter at different levels, i.e., Globally, Controller level and action method level. Finally, I will create one Controller with action methods showing the use of each Custom Action Filter:

**Creating a Model:**

Create a class file named **MyCustomModel.cs** within the Models folder and then copy and paste the following code into it. This will be our model, which we will use to return the data to the client and the action method parameter.

Advertisements

**namespace** *FiltersDemo.Models*

**{**

**public** **class** MyCustomModel

**{**

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

**public** **string**? Address **{** **get**; **set**; **}**

**public** **void** TransformData**()**

**{**

Name += " - Transformed";

Address += " - Transformed";

**}**

**}**

**}**

We will see how to Validate the above Model using a Custom Action Filter as well as we will see how to modify the above model data using a Custom Action Filter while returning the data to the client.

**Creating a Logger Service:**

Let us define our service interface and implementation for the logging. This Logger service is going to be used by our Custom Action Filters. So, create an interface named **ILoggerService.cs** within the **Models** folder and then copy and paste the following code:

**namespace** *FiltersDemo.Models*

**{**

**public** **interface** ILoggerService

**{**

**public** **void** Log**(string** message**)**;

**}**

**}**

Next, create another class file named **LoggerService.cs** within the **Models** folder and copy and paste the following code. This class implements the ILoggerService interface and implements the Log method, where we have written the logic to store the log message in a text file.

**namespace** *FiltersDemo.Models*

**{**

**public** **class** LoggerService : ILoggerService

**{**

**public** **void** Log**(string** message**)**

**{**

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

//saving the data in a text file called Log.txt within the Log folder which must be

//created at the Project root directory

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

**Creating the Log Folder:**

Next, create a folder called Log within the Project root directory where the Log.txt file is going to be generated by the application.

**Registering the Logger Service:**

Next, we need to register the Logger Service into the built-in dependency injection container. This is because we want to use the Logger service through our application, including the Custom Action Filter, and we want the Framework to inject the logger service through the constructor. So, add the following code to the Program.cs class file:

**builder.Services.AddSingleton<ILoggerService, LoggerService>();**

**Creating a Custom Action Filter for Logging in ASP.NET Core MVC:**

Create a class file named **LoggingFilterAttribute.cs** within the **Models** folder, and then copy and paste the following code. This filter logs details about the action method calls, including parameters, execution times, etc.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Newtonsoft.Json;*

**using** *System.Diagnostics;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** LoggingFilterAttribute : ActionFilterAttribute

**{**

**private** Stopwatch? \_timer = **null**;

**private** **readonly** ILoggerService \_LoggerService;

**public** LoggingFilterAttribute**(**ILoggerService LoggerService**)**

**{**

\_LoggerService = LoggerService;

**}**

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

**{**

\_timer = Stopwatch.StartNew**()**;

var actionName = context.ActionDescriptor.RouteValues**[**"action"**]**;

var controllerName = context.ActionDescriptor.RouteValues**[**"controller"**]**;

var parameters = JsonConvert.SerializeObject**(**context.ActionArguments**)**;

**string** message = $"Starting {controllerName}.{actionName} with parameters {parameters}";

\_LoggerService.Log**(**message**)**;

**base**.OnActionExecuting**(**context**)**;

**}**

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

**{**

\_timer?.Stop**()**;

var actionName = context.ActionDescriptor.RouteValues**[**"action"**]**;

var controllerName = context.ActionDescriptor.RouteValues**[**"controller"**]**;

**string** message = $"Finished {controllerName}.{actionName} in {\_timer.ElapsedMilliseconds}ms";

\_LoggerService.Log**(**message**)**;

**base**.OnActionExecuted**(**context**)**;

**}**

**}**

**}**

The above LoggingFilterAttribute is a Custom Filter that inherits from ActionFilterAttribute. This class is used to log information about MVC actions, specifically when they start and when they finish. It uses two methods, OnActionExecuting and OnActionExecuted, to achieve this.

**OnActionExecuting Method**

The OnActionExecuting method is called before the action method is executed. This method is used for the following purposes:

* **Start a Timer:** It initializes and starts the Stopwatch to measure the duration of the action’s execution. This is important for logging the action’s execution time.
* **Preparing the Log Message**: It constructs a message that includes the controller’s name, the action’s name, and the action’s serialized arguments.
* **Log the Start of the Action:** The constructed message (indicating the start of action execution) is logged using the \_LoggerService. This helps in tracking when an action begins its execution.

**OnActionExecuted Method**

The OnActionExecuted method is invoked after the action method has been executed. This method is used for the following purposes:

* **Stop the Timer:** It stops the Stopwatch that was started in OnActionExecuting.
* **Preparing the Log Message**: Similar to OnActionExecuting, it constructs a message that includes the name of the controller, the action, and the time taken (in milliseconds) to execute the action.
* **Logging the End of the Action:** This message (indicating the completion of the action and its duration) is then logged using the same logging service.

Advertisements

**Note:** Both methods call base.OnActionExecuting(context) and base.OnActionExecuted(context) at the end of their implementations. These calls ensure that any additional processing defined in the base class (ActionFilterAttribute) is also executed.

**Creating a Custom Action Filter for Data Transformation in ASP.NET Core MVC:**

Create a class file named **DataTransformationFilterAttribute.cs** within the **Models** folder, and then copy and paste the following code. This filter modifies the data returned from an action method. It assumes you are returning a specific model that could be transformed. Here, the following class is inherited from the ActionFilterAttribute and overrides the OnActionExecuted method.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** DataTransformationFilterAttribute : ActionFilterAttribute

**{**

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

**{**

**if** **(**context.Result **is** ViewResult viewResult**)**

**{**

**if** **(**viewResult.Model **is** MyCustomModel model**)**

**{**

// Directly modify the model data

model.TransformData**()**;

**}**

**}**

**base**.OnActionExecuted**(**context**)**;

**}**

**}**

**}**

**Understanding the OnActionExecuted method:**

* **Check the Result Type:** The method first checks if the action method’s result (context.Result) is of type ViewResult. ViewResult is a type of action result that renders a view as the response to the request.
* **Access and Modify the Model:** If the result type is ViewResult, the method then checks if the model associated with this view result is of type MyCustomModel. If it is, it accesses this model.
* **Transform the Model Data:** Once it has access to the model, the method calls the TransformData() function on the model. This function is assumed to manipulate or transform the data within the model. Depending on the implementation of TransformData(), this could involve changing values, formatting data, calculating fields, etc.
* **Call Base Method:** Finally, the method calls base.OnActionExecuted(context), which ensures that any logic implemented in the base class’s OnActionExecuted method is also executed.

**Creating a Custom Action Filter for Validation in ASP.NET Core MVC:**

Create a class file named **CustomValidationFilter.cs** within the Models folder, and then copy and paste the following code. This filter performs custom validation of action parameters. Here, we created the Custom Action Filter, implementing the **IActionFilter** interface and providing implementations for the **OnActionExecuting**and**OnActionExecuted** methods. Actually, we are not doing anything on the OnActionExecuted methods.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.ModelBinding;*

**using** *Microsoft.AspNetCore.Mvc.ViewFeatures;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomValidationFilter : IActionFilter

**{**

**public** **void** OnActionExecuting**(**ActionExecutingContext context**)**

**{**

**if** **(**context.ActionArguments.TryGetValue**(**"model", out var **value)** && **value** **is** MyCustomModel model**)**

**{**

// Validate Name

**if** **(string**.IsNullOrWhiteSpace**(**model.Name**))**

**{**

context.ModelState.AddModelError**(nameof(**model.Name**)**, "Name cannot be empty or whitespace."**)**;

**}**

// Validate Address

**if** **(string**.IsNullOrWhiteSpace**(**model.Address**))**

**{**

context.ModelState.AddModelError**(nameof(**model.Address**)**, "Address cannot be empty or whitespace."**)**;

**}**

**}**

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

**{**

// Assuming the controller action expects a return of the same view with the model

context.Result = new ViewResult

**{**

ViewName = context.RouteData.Values**[**"action"**]**.ToString**()**, // Gets the action name

ViewData = new ViewDataDictionary**(**new EmptyModelMetadataProvider**()**, context.ModelState**)**

**{**

Model = context.ActionArguments.FirstOrDefault**(**arg =**>** arg.Value **is** MyCustomModel**)**.Value

**}**

**}**;

**}**

**}**

**public** **void** OnActionExecuted**(**ActionExecutedContext context**)**

**{**

// Custom logic after the action executes

// For this filter, we do nothing here, but you could add post-processing logic if needed.

**}**

**}**

**}**

**Understanding OnActionExecuting Method**

* **Argument Validation:** The method starts by attempting to retrieve an argument named “model” from the context’s ActionArguments. It then checks if this argument is of type MyCustomModel.
* **Name Validation:** If the model’s Name property is null or whitespace, a model error stating that “Name cannot be empty or whitespace” is added.
* **Address Validation:** Similarly, it checks the Address property and adds a model error if it’s null or whitespace.
* **Invalid Model Handling:** If the model state is invalid, it prevents the execution of the action method by setting the context’s Result property.
* **Creating and Returning ViewResult:** A new ViewResult is created to return the same view with the invalid model data. This ViewResult includes the name of the action (retrieved from context.RouteData) to indicate which view to return. A new ViewDataDictionary is initialized with the model state and model, allowing the view to display validation errors.

**Creating a Custom Action Filter for Error Handling in ASP.NET Core MVC:**

Create a class file named **ErrorHandlerFilterAttribute.cs** within the Models folder, and then copy and paste the following code. This filter implements custom error-handling logic for actions. The Custom Action Filter is inherited from the ExceptionFilterAttribute class and overrides the OnException method.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.ModelBinding;*

**using** *Microsoft.AspNetCore.Mvc.ViewFeatures;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** ErrorHandlerFilterAttribute : ExceptionFilterAttribute

**{**

**private** **readonly** ILoggerService \_LoggerService;

**public** ErrorHandlerFilterAttribute**(**ILoggerService LoggerService**)**

**{**

\_LoggerService = LoggerService;

**}**

**public** **override** **void** OnException**(**ExceptionContext context**)**

**{**

**string** message = $"An error occurred in {context.ActionDescriptor.DisplayName}: {context.Exception}";

\_LoggerService.Log**(**message**)**;

// Set the result to redirect to the generic error view

context.Result = new ViewResult

**{**

ViewName = "~/Views/Shared/Error.cshtml", // Explicit path to the view

ViewData = new ViewDataDictionary**(**new EmptyModelMetadataProvider**()**, context.ModelState**)**

**{**

**{**"Exception", context.Exception**}** // Optionally pass exception data to the view

**}**

**}**;

context.ExceptionHandled = **true**; // Mark exception as handled

**}**

**}**

**}**

**Understanding OnException Method:**

* **Logging the Exception:** When an exception occurs in any action method to which this filter is applied, the OnException method captures the exception and logs a detailed message. The message includes the name of the action where the exception occurred and the exception details itself. This is done by the Logger Service, which is injected through the constructor.
* **Setting the Response:** After logging the exception, the method proceeds to alter the user’s experience by redirecting them to a generic error page. This is done by setting context.Result to a new ViewResult:
* **View Name:** It specifies the path to the error view (**~/Views/Shared/Error.cshtml**). This ensures that whenever an exception is handled by this filter, the user is redirected to a standard error page, maintaining a consistent error-handling strategy across the application.
* **ViewData:** The ViewDataDictionary is populated with the exception data using the EmptyModelMetadataProvider and the current ModelState. This allows the error view to access details of the exception, if necessary, which can be useful for displaying error messages or diagnostic information on the error page.
* **Marking the Exception as Handled:** By setting context.ExceptionHandled to true, the filter communicates to the ASP.NET Core framework that the exception has been handled. This prevents the exception from propagating further, which means it won’t trigger other exception handlers or result in the framework’s default error-handling mechanisms taking over (such as showing the developer exception page).

**Creating the Error View:**

Next, create a view named **Error.cshtml** within the **Views/Shared** folder and then copy and paste the following code. This is the view which is going to be rendered by the above Custom Error Handler:

@**{**

Layout = "~/Views/Shared/\_Layout.cshtml";

ViewData**[**"Title"**]** = "Error";

**}**

**<**h1 **class**="text-danger"**>**Oops! Something went wrong.**<**/h1**>**

**<**p**>**We're having trouble processing your request. Please **try** again later.**<**/p**>**

**Creating a Custom Action Filter for Caching in ASP.NET Core MVC:**

Create a class file named **AsyncCachingFilter.cs** within the **Models** folder, and then copy and paste the following code. The following Custom Action Filter implements the IAsyncActionFilter and provides an implementation for the OnActionExecutionAsync method.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.Extensions.Caching.Memory;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** AsyncCachingFilter : IAsyncActionFilter

**{**

**private** **readonly** IMemoryCache \_cache;

**private** **readonly** TimeSpan \_expirationTimeSpan;

**public** AsyncCachingFilter**(**IMemoryCache cache, **double** secondsToCache = 60**)**

**{**

\_cache = cache;

\_expirationTimeSpan = TimeSpan.FromSeconds**(**secondsToCache**)**;

**}**

**public** **async** Task OnActionExecutionAsync**(**ActionExecutingContext context, ActionExecutionDelegate next**)**

**{**

var key = $"{context.HttpContext.Request.Path}";

**if** **(**\_cache.TryGetValue**(**key, out IActionResult cachedResult**))**

**{**

context.Result = cachedResult; // Return cached result

**}**

**else**

**{**

// Proceed with the action execution

var executedContext = **await** next**()**;

// Cache any IActionResult that is successfully returned

**if** **(**executedContext.Result **is** ActionResult actionResult**)**

**{**

\_cache.Set**(**key, actionResult, \_expirationTimeSpan**)**;

**}**

**}**

**}**

**}**

**}**

**Understanding OnActionExecutionAsync Method:**

* **Initialization:** The Constructor uses an injected IMemoryCache instance (\_cache) to store and retrieve cached data. The cache’s expiration is set via a TimeSpan (\_expirationTimeSpan), which is determined by the number of seconds passed into the constructor.
* **Caching Logic:** The cache key is generated from the request path (context.HttpContext.Request.Path). This ensures that each unique URL has its own cache entry.
* **Cache Lookup:** The method first checks if there is a cached result available for the generated key using \_cache.TryGetValue(key, out IActionResult cachedResult). If a cached result exists (cachedResult), it is immediately assigned to context.Result. This tells the framework to skip executing the action method and return the cached result directly to the client.
* **Action Execution:** If no cached result is found, the method proceeds to execute the action by calling await next(). This ActionExecutionDelegate (next) represents the next delegate in the action filter pipeline, which typically leads to the execution of the action method itself. After the action executes, the resulting context (executedContext) is returned, which includes the result of the action method.
* **Caching the Result:** The result of the action method (executedContext.Result) is checked to see if it’s an instance of ActionResult. This check is important because only action results should be cached. If it is an ActionResult, it is added to the cache with the earlier generated key. The cache entry is set to expire based on \_expirationTimeSpan.

**Modifying the Home Controller:**

Let’s modify the Home Controller class code as follows to demonstrate the use of all created Custom Action Filters.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

**[**TypeFilter**(**typeof**(**LoggingFilterAttribute**))]**

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

**{**

**[**TypeFilter**(**typeof**(**AsyncCachingFilter**))]**

**public** IActionResult Index**()**

**{**

// Storing the current time in ViewBag

ViewBag.CurrentTime = DateTime.Now;

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

**}**

**[**DataTransformationFilter**]**

**public** IActionResult Details**()**

**{**

var model = new MyCustomModel

**{**

Name = "Initial Name",

Address = "Initial Address"

**}**;

//return Ok(model);

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

**}**

**[**HttpGet**]**

**public** IActionResult Create**()**

**{**

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

**}**

**[**TypeFilter**(**typeof**(**CustomValidationFilter**))]**

**[**HttpPost**]**

**public** IActionResult Create**(**MyCustomModel model**)**

**{**

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

**{**

// Process the valid model here

**return** RedirectToAction**(nameof(**Index**))**;

**}**

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

**}**

**[**TypeFilter**(**typeof**(**ErrorHandlerFilterAttribute**))]**

**public** IActionResult Error**()**

**{**

**throw** new Exception**(**"This is a forced error!"**)**;

**}**

**}**

**}**

**Creating and Modifying the Views:**

Next, we need to modify and create the Views as per our requirements:

**Index.cshtml View:**

Modify the **Index.cshtml** View as follows:

@{

ViewData["Title"] = "Index Page";

}

**<h2>**Current Server Time**</h2>**

**<p>**The current server time is: @ViewBag.CurrentTime.ToString("F")**</p>**

**Details.cshtml View**

Add **Details.cshtml** View within the **Views/Home** folder and then copy and paste the following code:

@model FiltersDemo.Models.MyCustomModel

@{

ViewData["Title"] = "Details Page";

}

**<div** class="container mt-5"**>**

**<h1** class="mb-3"**>**Details**</h1>**

**<div** class="card"**>**

**<div** class="card-body"**>**

**<h5** class="card-title"**>**Name**</h5>**

**<p** class="card-text"**>**@Model.Name**</p>**

**<h5** class="card-title"**>**Address**</h5>**

**<p** class="card-text"**>**@Model.Address**</p>**

**</div>**

**</div>**

**</div>**

**Create.cshtml View**

Add **Create.cshtml** View within the **Views/Home** folder and then copy and paste the following code:

@model FiltersDemo.Models.MyCustomModel

@{

ViewData["Title"] = "Create Model";

}

**<div** class="container mt-5"**>**

**<h1>**Create Model**</h1>**

**<form** asp-action="Create" method="post" class="needs-validation" novalidate**>**

**<div** asp-validation-summary="ModelOnly" class="alert alert-danger"**></div>**

**<div** class="form-group"**>**

**<label** for="Name"**>**Name**</label>**

**<input** asp-for="Name" class="form-control" **/>**

**<span** asp-validation-for="Name" class="text-danger"**></span>**

**</div>**

**<div** class="form-group"**>**

**<label** for="Address"**>**Address**</label>**

**<input** asp-for="Address" class="form-control" **/>**

**<span** asp-validation-for="Address" class="text-danger"**></span>**

**</div>**

**<button** type="submit" class="btn btn-primary"**>**Submit**</button>**

**</form>**

**</div>**

**Testing the Application:**

Now, run the application and test each action method. We have applied the Logging Custom Filter at the Controller level, meaning it will be applied to all Action methods of the Home Controller.

**Index Action Method:**

Now, access the **Home/Index** URL, and you should see the following. Now, within 60 seconds, if you access the same page, then you will see that the Date is not going to be changed. This is because we have applied the Custom Cache filter on the Index Action method:

**Details Action Method:**

Now, access the **Home/Details** URL, and you should see the following. This is because we have applied the Custom Data Modification filter on the Details Action method, which modifies the Model:

**Create Action Method:**

Now, access the **Home/Create** URL without providing any data, click on the Submit button, and you should see the following. This is because we have applied the Custom Validator filter on the Create Post Action method, which is doing the Model validation before executing the Create Post action method:

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=8f38bc20-b61e-498b-60a5-0b3a1bb2b619&d_id=118560&imp_id=384999776852725&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**Error Action Method:**

Now, access the **Home/Error** URL, and you should see the following. This is because we have applied the Custom Exception filter to the Error Action method. The Error action method throws an unhandled exception that is going to be handled by the Custom Exception Filter, and then it returns a generic error page to the client.

In the next article, I will discuss the [**Difference Between TypeFilter and ServiceFilter in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/difference-between-typefilter-and-servicefilter-in-asp-net-core-mvc/) Applications. In this article, I try to explain the **Action Filters in ASP.NET Core MVC** Applications with Examples. I hope you understand the need and use of Action Filters in the ASP.NET Core MVC Applications.

TypeFilter vs ServiceFilter in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

**TypeFilter vs ServiceFilter in ASP.NET Core MVC**

In this article, I will discuss **TypeFilter vs ServiceFilter, i.e., the Difference Between TypeFilter and ServiceFilter in ASP.NET Core** MVC Applications with Examples. Please read our previous article discussing [**Action Filters in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/action-filters-in-asp-net-core-mvc/) Applications.

**TypeFilter vs ServiceFilter in ASP.NET Core MVC**

In ASP.NET Core, Filters execute custom pre- and post-processing logic before and after the action method execution. They can be used for various purposes, such as Authentication, Authorization, Caching, Exception Handling, Result Modification, etc. There are different kinds of filters (like Authorization, Action, Exception, and Result filters). We can also apply the filters to controllers and actions in different ways, such as applying at the action method level, at the controller level, or globally.

In ASP.NET Core, TypeFilter and ServiceFilter are built-in attributes that apply custom filters to controller actions or controllers. However, they have different purposes and are used in different scenarios. Now, let us proceed and try to understand the TypeFilter and ServiceFilter, what are the differences between them, and when to use which one:

**TypeFilter Attribute in ASP.NET Core MVC:**

TypeFilter is a Built-in Attribute in ASP.NET Core that allows us to apply a Custom Filter to a single action or controller. We must provide the filter type (Custom Filter Class name) as a parameter to the TypeFilter attribute. It is useful when we want to apply a Custom Filter Class to a specific action or controller and don’t need to configure it globally or inject additional services.

**Example to Understand TypeFilter Attribute in ASP.NET Core MVC:**

Let us see an example of understanding the TypeFilter Attribute in an ASP.NET Core MVC Application. Remember that you need to use the TypeFilter or Service Filter Attribute only when your Custom Filter class requires some services and you want to inject such services using the Dependency Injection Container.

So, let us first create a service that is going to be consumed by the Custom Filter Classes. So, create a class file named **ILoggerService.cs** and copy and paste the following code. Here, you can see that the ILoggerService interface defines one method called Log, and the LoggerService class implements the ILoggerService interface and provides the implementation for the Log method.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ea413a1a-5a0a-43d6-6b1f-882dc9d12ec3&d_id=118560&imp_id=6433625956902453&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**namespace** *FiltersDemo.Models*

**{**

**public** **interface** ILoggerService

**{**

**public** **void** Log**(string** methodName, **string** message**)**;

**}**

**public** **class** LoggerService : ILoggerService

**{**

**public** **void** Log**(string** methodName, **string** message**)**

**{**

**string** filePath = Path.Combine**(**Directory.GetCurrentDirectory**()**, "Log", "Log.txt"**)**;

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

File.AppendAllText**(**filePath, message**)**;

**}**

**}**

**}**

**Creating Custom Action Filter:**

Next, create a class file named **CustomLoggingFilter.cs** and then copy and paste the following code. As you can see, this class implements the **IActionFilter** interface and provides implementations for the **OnActionExecuting** and **OnActionExecuted** methods. Further, we have injected the LoggerService instance through constructor dependency injection.

**using** *Microsoft.AspNetCore.Mvc.Filters;*

**namespace** *FiltersDemo.Models*

**{**

**public** **class** CustomLoggingFilter : IActionFilter

**{**

**private** **readonly** ILoggerService \_LoggerService;

**public** CustomLoggingFilter**(**ILoggerService LoggerService**)**

**{**

\_LoggerService = LoggerService;

**}**

**public** **void** OnActionExecuting**(**ActionExecutingContext context**)**

**{**

var controllerName = context.RouteData.Values**[**"controller"**]**;

var actionName = context.RouteData.Values**[**"action"**]**;

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

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

// Log the information before the action executes.

\_LoggerService.Log**(**"OnActionExecuting", message**)**;

**}**

**public** **void** OnActionExecuted**(**ActionExecutedContext context**)**

**{**

var controllerName = context.RouteData.Values**[**"controller"**]**;

var actionName = context.RouteData.Values**[**"action"**]**;

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

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

// Log the information after the action executes.

\_LoggerService.Log**(**"OnActionExecuted", message**)**;

**}**

**}**

**}**

**Registering the Services to the Dependency Injection Container:**

Next, we must register the Services to the Dependency Injection Container within the Program class so that the MVC Framework injects the service. So, add the following line to the Program class. Here, it is not required to register the Custom Filter into the dependency injection container.

**builder.Services.AddScoped<ILoggerService, LoggerService>();**

**Apply Filter as a Service:**

To apply the filter as a service, we can use TypeFilter Attribute. A **TypeFilterAttribute** is used to create an instance of a Custom Filter Class using dependency injection. You need to pass the Custom Filter Type as an argument to its constructor. The Type Filter instantiates the filter type each time the filter is applied. So, modify the Home Controller as follows:

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**TypeFilter**(**typeof**(**CustomLoggingFilter**))]**

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

**{**

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

**}**

**[**TypeFilter**(**typeof**(**CustomLoggingFilter**))]**

**public** ActionResult Details**()**

**{**

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

**}**

**}**

**}**

**Modify Index.cshtml File**

@{

ViewData["Title"] = "Index Page";

}

**<h2>**Index Page**</h2>**

**Add Details.cshtml File**

@{

ViewData["Title"] = "Details Page";

}

**<h2>**Details Page**</h2>**

**Testing the Application:**

Each time a request is made to either the Index or Details action methods, a new instance of CustomLoggingFilter is created. This is because the [TypeFilter] attribute creates an instance of the specified filter type per request for each action method it decorates.

Therefore, if we make a request to the Index method and another request to the Details method, two separate instances of CustomLoggingFilter will be created, i.e., one for each request. Each subsequent request to these methods will create new instances accordingly. Now, run the application, access both Index and Details action methods, and then check the**Log.txt** file; you should see that the Instance of the CustomLoggingFilter was created twice, as shown in the image below.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ea413a1a-5a0a-43d6-6b1f-882dc9d12ec3&d_id=118560&imp_id=2650924980910387&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**ServiceFilter Attribute in ASP.NET Core MVC:**

ServiceFilter is also a Built-in Attribute in ASP.NET Core that applies a Custom Filter as an Attribute in the Controller or Controller Action Method. However, the Custom Filter must be registered as a service in the dependency injection container. It is useful when we want to reuse the same Custom Filter Instance across multiple actions or controllers. That means, unlike the TypeFilter, it doesn’t create a new instance but retrieves it from the DI container.

We are going to work with the same example. So, modify the Home Controller as follows to use the ServiceFilter Attribute. In both action methods, you can see that we have applied the CustomLoggingFilter using the ServiceFilter attribute.

**using** *FiltersDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *FiltersDemo.Controllers*

**{**

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

**{**

**[**ServiceFilter**(**typeof**(**CustomLoggingFilter**))]**

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

**{**

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

**}**

**[**ServiceFilter**(**typeof**(**CustomLoggingFilter**))]**

**public** ActionResult Details**()**

**{**

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

**}**

**}**

**}**

At this point, if you run the application, then you will get the following 500 HTTP Error. This is because we are applying the CustomLoggingFilter using the ServiceFilter attribute, but we have not yet registered the CustomLoggingFilter as a service into the built-in dependency injection container.

**Registering the Custom Filter as a Service:**

With TypeFilter, we specify the filter type directly within the TypeFilter constructor and don’t need to register it as a service into the Dependency Injection Container. But, with ServiceFilter, we must register the Custom Filter as a service into the Dependency Injection Container before using it with the ServiceFilter attribute. So, add the following code to the Program class.

**builder.Services.AddSingleton<CustomLoggingFilter>();**

With the above changes, run the application and access both the Index and Details action methods. Then verify the Log.txt file, and this time, you will see that only one instance of the CustomLoggingFilter is created, as shown in the image below. This is because we registered the CustomLoggingFilter using the AddSingleton method, which, as we already discussed, will create only one instance of the registered service throughout the application’s life and rescue that instance.

**Differences Between TypeFilter and ServiceFilter in ASP.NET Core:**

The following are the Differences Between TypeFilter and ServiceFilter in ASP.NET Core MVC:

* **TypeFilter**creates a new instance of the filter type on each request. On the other hand, the **ServiceFilter** retrieves the instance from the DI container, which could be a singleton, scoped, or transient instance, depending on how it’s registered.
* The **TypeFilter** is more flexible because it can instantiate types that aren’t registered in the DI container, while the **ServiceFilter**requires the type to be registered in the DI container.

So, we need to use TypeFilter if we need to inject dependencies that aren’t registered in the container. We need to use ServiceFilter if the filter is already registered as a service in the container, and we want the Dependency Injection Container to manage the filter’s lifetime.

In the next article, I will discuss [**Cross-Site Request Forgery and Antiforgery Tokens in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/antiforgerytoken-in-asp-net-core-mvc/). In this article, I explain the difference between TypeFilter and ServiceFilter in ASP.NET Core Applications with examples. I hope you enjoy this Difference Between TypeFilter and ServiceFilter in the ASP.NET Core article.

AntiForgery Token in ASP.NET Core MVC

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](https://dotnettutorials.net/course/asp-net-core-tutorials/)

**Cross-Site Request Forgery and AntiForgeryToken in ASP.NET Core MVC**

In this article, I will discuss **Cross-Site Request Forgery and AntiForgery Token in ASP.NET Core MVC** Applications with Examples. Please read our previous article discussing the Difference Between **[TypeFilter and ServiceFilter in ASP.NET Core MVC](https://dotnettutorials.net/lesson/difference-between-typefilter-and-servicefilter-in-asp-net-core-mvc/)**. In this article, first, I will discuss What is Cross-Site Request Forgery (CSRF or XSRF) with an example, and then I will show you how we can prevent the CSRF or XSRF attack using AntiForgery Token in ASP.NET Core MVC Applications.

**What is a Cross-Site Request Forgery Attack?**

Cross-Site Request Forgery (CSRF), also known as XSRF or Sea Surf, is a type of security attack that exploits the trust that a web application has in an authenticated user’s browser. This allows an attacker to perform actions on behalf of the authenticated user without their knowledge or consent. In a CSRF attack, the attacker tricks the victim into submitting a malicious request to a web application where they are authenticated, exploiting the user’s trust. If this is not clear at the moment, don’t worry; we will try to understand it with an example.

**Example to Understand Cross-Site Request Forgery (CSRF or XSRF) Attack**

Let us create a new ASP.NET Core Application using the Model-View-Controller Project template and give the project name **BankingApplication**. Once you create the project, modify the **Home Controller** as follows. So, our requirement is to create a view for updating the PIN of an Account Number of a user. In the below code, the Get version of the ChangePin action method will create a form where the user will enter his Account Number and Updated PIN and click on the PIN Change button. Once the User clicks on the PIN Change button, the form data will be submitted to the Post version of the ChangePin action method, which will process the data, update the PIN number, and then redirect to the PinChangeSuccess page.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *BankingApplication.Controllers*

**{**

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

**{**

**[**HttpGet**]**

**public** IActionResult ChangePin**()**

**{**

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

**}**

**[**HttpPost**]**

**public** ActionResult ChangePin**(string** AccountNumber, **string** Pin**)**

**{**

// Process the data

TempData**[**"Message"**]**= $"AccountNumber: {AccountNumber} Pin Changed to: {Pin}";

**return** RedirectToAction**(**"PinChangeSuccess"**)**;

**}**

**public** ActionResult PinChangeSuccess**()**

**{**

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

**}**

**}**

**}**

**Creating the ChangePin View:**

Next, create a view named **ChangePin.cshtml** within the **Views/Home** folder and then copy and paste the following code. Here, you can see we have one form with two text boxes and one submit button.

@{

ViewData["Title"] = "Pin Change Page";

}

**<div>**

**<form** asp-controller="Home" asp-action="ChangePin" method="post" class="mt-3"**>**

**<div** style="margin-top:7px" class="form-group row"**>**

**<label** for="AccountNumber" class="col-sm-2 col-form-label"**>**Account Number**</label>**

**<div** class="col-sm-10"**>**

**<input** type="text" name="AccountNumber" id="AccountNumber" class="form-control" **/>**

**</div>**

**</div>**

**<div** style="margin-top:7px" class="form-group row"**>**

**<label** for="Pin" class="col-sm-2 col-form-label"**>**Pin**</label>**

**<div** class="col-sm-10"**>**

**<input** type="text" name="Pin" id="Pin" class="form-control" **/>**

**</div>**

**</div>**

**<div** style="margin-top:10px" class="form-group row"**>**

**<div** class="col-sm-10"**>**

**<button** type="submit" class="btn btn-primary"**>**PIN Change**</button>**

**</div>**

**</div>**

**</form>**

**</div>**

**Creating the PinChangeSuccess View:**

Next, create a view named **PinChangeSuccess.cshtml** within the **Views/Home** folder and then copy and paste the following code. We are simply displaying the PIN Change Success message.

@{

ViewData["Title"] = "PIN Change Success";

}

**<div** class="container"**>**

**<div** class="row justify-content-center"**>**

**<div** class="col-md-8"**>**

**<div** class="alert alert-success mt-5" role="alert"**>**

**<h4** class="alert-heading"**>**Success!**</h4>**

**<p>**@TempData["Message"]**</p>**

**<hr>**

**<p** class="mb-0"**>**If you need to make any further changes, please return to your account settings.**</p>**

**</div>**

**</div>**

**</div>**

**</div>**

Now, run the application, visit the **Home/ChangePin**, enter the Account Number and PIN, and then click on the PIN Change button as shown in the below image:

Once you click on the PIN Change button, the PIN has been changed for the account number, and you should see the following success message:

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2202a7f3-71ae-4f5a-62e3-7f9b0337e24e&d_id=118560&imp_id=8029651916856884&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

As you can see, the application is working as expected. Please observe the URL **https://localhost:7199/Home/ChangePin**. Now, let’s see how a hacker can use the above URL to launch a Cross-Site Request Forgery (CSRF or XSRF) attack.

**Hackers Application:**

Let us create a new ASP.NET Core Application using the Model View Controller Project template and give the project name **HackerApplication**. Once you create the HackerApplication, then modify the Home Controller as follows:

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *HackerApplication.Controllers*

**{**

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

**{**

**public** IActionResult Index**()**

**{**

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

**}**

**}**

**}**

Next, modify the **Index.cshtml** view of the Home Controller of HackerApplication as follows. Here, you can see we are posting the form data to the Banking Application. We have a text field to store the Account Number and a hidden field to store some hard-coded PIN. Once the user clicks on the **Claim My Gift Card** button, the request will be submitted to the Banking Application, where it will update the Account Number and PIN.

@{

ViewData["Title"] = "Claim Gift Card Page";

}

**<div** class="container"**>**

**<div** class="row justify-content-center"**>**

**<div** class="col-md-6"**>**

**<div** class="text-center"**>**

**<h3>**Enter your Account Number to Claim Your Gift**</h3>**

**<form** action="https://localhost:7199/Home/ChangePin" method="post" class="form-group"**>**

**<div** class="form-group mt-3"**>**

**<input** type="text" name="AccountNumber" id="AccountNumber" class="form-control" placeholder="Account Number" **/>**

**</div>**

**<div** class="form-group mt-3"**>**

**<input** type="hidden" name="Pin" id="Pin" value="1111" **/>**

**<button** type="submit" class="btn btn-primary"**>**Claim Your Gift**</button>**

**</div>**

**</form>**

**</div>**

**</div>**

**</div>**

**</div>**

Now, run the application, enter your account number, and click the **Claim Your Gift button**, as shown in the image below.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2202a7f3-71ae-4f5a-62e3-7f9b0337e24e&d_id=118560&imp_id=8328888008896869&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Once you click on the **Claim Your Gift** button, your account number pin is updated in the Banking Application as shown in the below image:

This is nothing but a Cross-Site Request Forgery (CSRF or XSRF) attack on a website.

**How can we prevent Cross-Site Request Forgery (CSRF or XSRF) Attack in ASP.NET Core MVC?**

To prevent Cross-Site Request Forgery (CSRF or XSRF) in ASP.NET Core MVC Web Applications, we need to use AntiForgery Tokens. ASP.NET Core MVC uses AntiForgery Tokens, also known as request verification tokens, to prevent CSRF attacks. These tokens are unique to each user session and confirm that the user submitting a request is the one who originally requested the page.

**How Does It Work in ASP.NET Core?**

* When a form is rendered, ASP.NET Core MVC injects a hidden form field with the anti-forgery token.
* The user’s browser submits this token along with the form data.
* Upon receiving the request, the server validates the token to ensure that the request is legitimate.

Advertisements

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2202a7f3-71ae-4f5a-62e3-7f9b0337e24e&d_id=118560&imp_id=181178272911141&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**How to Use AntiForgery Tokens in ASP.NET Core MVC?**

To generate the anti-forgery token, we need to use the **@Html.AntiForgeryToken()** helper method within the <form> tag. Then, we need to decorate the action method that handles the posted form data with the [ValidateAntiForgeryToken] attribute. The [ValidateAntiForgeryToken] attribute will ensure that the action method processes the request only if it comes with a valid anti-forgery token.

From ASP.NET Core 2.0 onwards, the framework automatically generates anti-forgery tokens for all forms by default, so we don’t need to use the **@Html.AntiForgeryToken()** helper method. The framework validates the token automatically when the action method is decorated with the **[ValidateAntiForgeryToken]** attribute. If the token is missing or invalid, it rejects the request. So, let us proceed and implement this in our Banking Application:

Modify the **ChangePin.cshtml** view of the Home Controller of our banking application as follows. Here, we are adding the **@Html.AntiForgeryToken()** helper method within the form body. The @Html.AntiForgeryToken() method generates a hidden input field containing the anti-forgery token.

@{

ViewData["Title"] = "Pin Change Page";

}

**<div>**

**<form** asp-controller="Home" asp-action="ChangePin" method="post" class="mt-3"**>**

@Html.AntiForgeryToken()

**<div** style="margin-top:7px" class="form-group row"**>**

**<label** for="AccountNumber" class="col-sm-2 col-form-label"**>**Account Number**</label>**

**<div** class="col-sm-10"**>**

**<input** type="text" name="AccountNumber" id="AccountNumber" class="form-control" **/>**

**</div>**

**</div>**

**<div** style="margin-top:7px" class="form-group row"**>**

**<label** for="Pin" class="col-sm-2 col-form-label"**>**Pin**</label>**

**<div** class="col-sm-10"**>**

**<input** type="text" name="Pin" id="Pin" class="form-control" **/>**

**</div>**

**</div>**

**<div** style="margin-top:10px" class="form-group row"**>**

**<div** class="col-sm-10"**>**

**<button** type="submit" class="btn btn-primary"**>**PIN Change**</button>**

**</div>**

**</div>**

**</form>**

**</div>**

Next, modify the **Home Controller** of our **Banking Application** as follows. Here, we are decorating the Post ChangePin action method with the ValidateAntiForgeryToken attribute. The **[ValidateAntiForgeryToken]** attribute ensures the request includes a valid anti-forgery or request verification token.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *BankingApplication.Controllers*

**{**

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

**{**

**[**HttpGet**]**

**public** IActionResult ChangePin**()**

**{**

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

**}**

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult ChangePin**(string** AccountNumber, **string** Pin**)**

**{**

// Process the data

TempData**[**"Message"**]**= $"AccountNumber: {AccountNumber} Pin Changed to: {Pin}";

**return** RedirectToAction**(**"PinChangeSuccess"**)**;

**}**

**public** ActionResult PinChangeSuccess**()**

**{**

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

**}**

**}**

**}**

With the above changes, run the Banking Application; you will see that the account number pin updation functionality is working as expected. Now, run the Hacker Application and click on the **Claim My Gift Card** button by entering the Account, and you should see the following error page:

The anti-forgery token filed name is **\_\_RequestVerificationToken,** and if you want, you can also capture this field value in your application code. For example, modify the Home Controller of Banking Application as follows:

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *BankingApplication.Controllers*

**{**

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

**{**

**[**HttpGet**]**

**public** IActionResult ChangePin**()**

**{**

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

**}**

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** ActionResult ChangePin**(string** AccountNumber, **string** Pin, **string** \_\_RequestVerificationToken**)**

**{**

// Process the data

TempData**[**"Message"**]**= $"AccountNumber: {AccountNumber} Pin Changed to: {Pin}";

**return** RedirectToAction**(**"PinChangeSuccess"**)**;

**}**

**public** ActionResult PinChangeSuccess**()**

**{**

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

**}**

**}**

**}**

**How AntiForgeryToken Generated and Validated in ASP.NET Core MVC?**

ASP.NET Core generates a unique token called the AntiForgeryToken when a form is rendered. This token is created using the **@Html.AntiForgeryToken()** helper method in Razor views. The generated token consists of two parts: a hidden field in the form and a cookie. The hidden field ensures the token is sent along with the form data during a POST request while the cookie is stored in the user’s browser. The token is encrypted and securely signed to ensure that it cannot be tampered with by an attacker. So, when we send the request to the server, the token is sent using the hidden form field as well as through the cookie header.

In ASP.NET Core MVC, when a form is submitted, the framework automatically validates the AntiForgeryToken if the **[ValidateAntiForgeryToken]** attribute is applied to an action method. During validation, the framework compares the token from the hidden field (sent in the POST request) with the token coming from the request header cookie. If the tokens do not match or are missing, ASP.NET Core will throw an exception, preventing the action method from executing, thereby stopping the potential CSRF attack.

In the next article, I will discuss **Action Name Attributes in ASP.NET Core MVC**. In this article, I explain **Cross-Site Request Forgery and AntiForgeryToken in ASP.NET Core MVC** Applications with Examples. I hope you enjoy this Cross-Site Request Forgery and AntiForgeryToken in the ASP.NET Core MVC article.