Working With Filters In ASP.NET Core MVC

Jignesh Trivedi Updated date Jul 18, 2017

Introduction

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

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

Filter Types

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

- *Authorization filters*
  The Authorization filters are executed first. This filter helps us to determine whether the user is authorized for the current request. It can short-circuit a pipeline if a user is unauthorized for the current request. We can also create custom authorization filter.

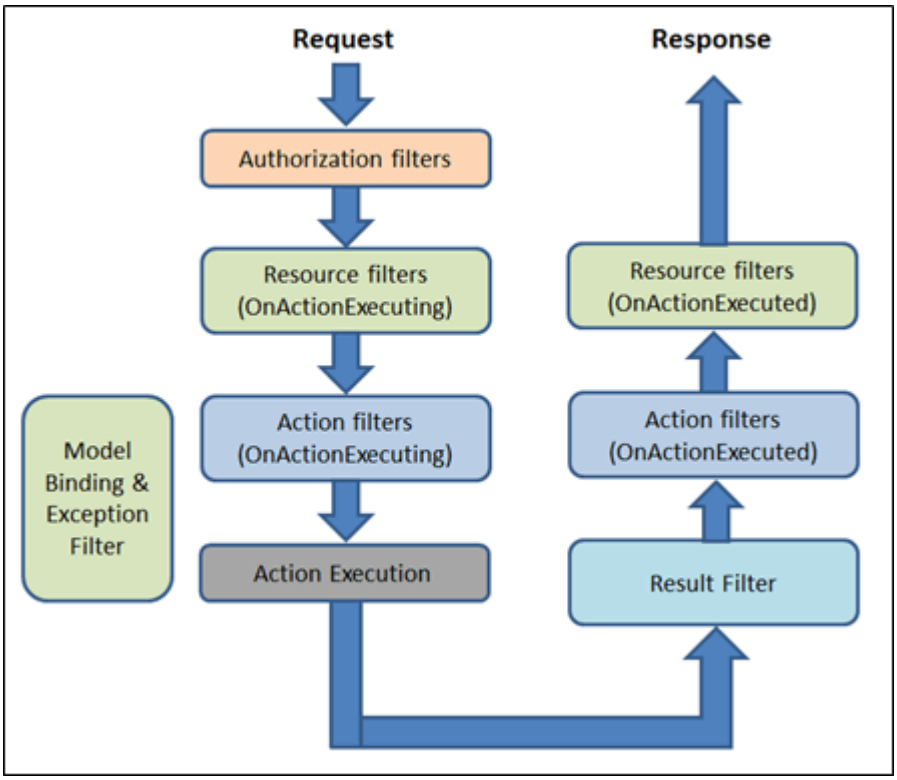
- *Resource filters*
  The Resource filters handle the request after authorization. It can run the code before and after the rest of the filter is executed. This executes before the model binding happens. It can be used to implement caching.

is called. It can be used to perform any action before or after execution of the controller action method. We can also manipulate the arguments passed into an action.

- *Exception filters*
  The Exception filters are used to handle exception that occurred before anything written to the response body.

- *Result filters*
  The Result filters are used to run code before or after the execution of controller action results. They are executed only if the controller action method has been executed successfully.

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



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

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

Synchronous Filter Example

```
{
    public class CustomActionFilter : IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
        }
    }
}
```

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

Asynchronous filter example

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters
{
    public class CustomAsyncActionFilter : IAsyncActionFilter
    {
        public async Task OnActionExecutionAsync(ActionExecutingContext cc
            ActionExecutionDelegate next)
        {
            //To do : before the action executes
            await next();
            //To do : after the action executes
        }
    }
}
```

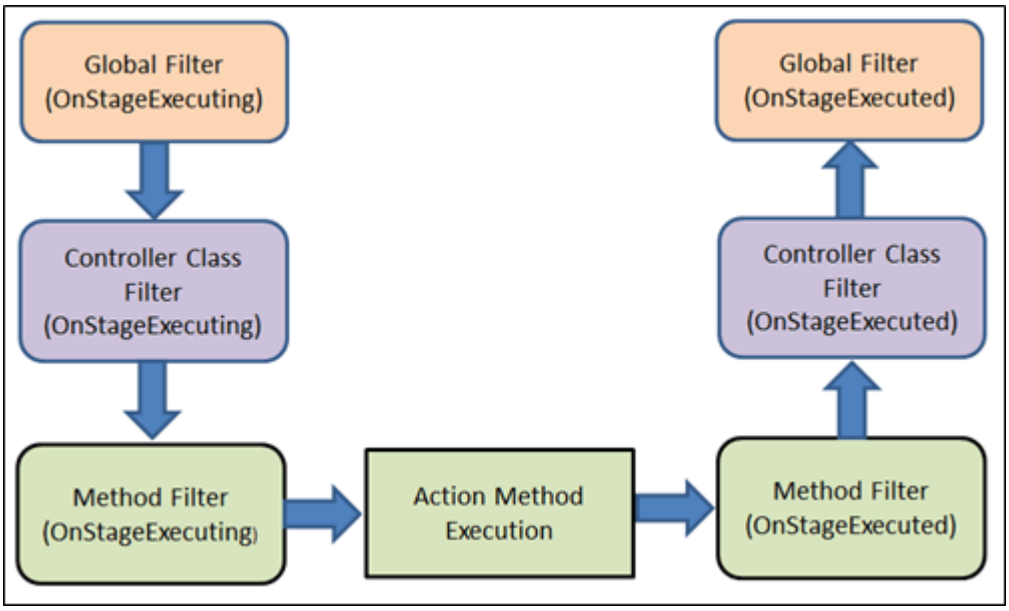
We can implement interfaces for multiple filter types (stage) in single class. We can either implement synchronous or the async version of a filter interface, not both. The .net framework checks first for async filter interface, if it finds it, it called. If it is not found it calls the synchronous interface's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

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

```
01. public void ConfigureServices(IServiceCollection services)
02. {
03.     // Add framework services.
04.     services.AddMvc(options=> {
05.     //an instant
06.     options.Filters.Add(new CustomActionFilter());
07.     //By the type
08.     options.Filters.Add(typeof(CustomActionFilter));
09.     });
10. }
```

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

Following figure shows the default order of filter execution.



Overriding the default order

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

ExampleFilter.cs

```
namespace Filters
{
    public class ExampleFilterAttribute : Attribute, IActionFilter, IOrder
    {
        public int Order { get; set; }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
        }
    }
}
```

Controller.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
using Filters;

namespace Filters.Controllers
{
    [ExampleFilter(Order = 1)]
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

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

Cancellation or short circuiting filters

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

Filter Example

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
```

```
06.     public class ExampleFilterAttribute : Attribute, IActionFilter
07.     {
08.         public void OnActionExecuting(ActionExecutingContext context)
09.         {
10.             //To do : before the action executes
11.             context.Result = new ContentResult()
12.             {
13.                 Content = "Short circuit filter"
14.             };
15.         }
16.         public void OnActionExecuted(ActionExecutedContext context)
17.         {
18.             //To do : after the action executes
19.         }
20.     }
21. }
```

Filters and DI (Dependency Injection)

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

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

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

- ServiceFilterAttribute
- TypeFilterAttribute
- IFilterFactory implemented on attribute

ServiceFilterAttribute

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

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

```
01. using Microsoft.AspNetCore.Mvc.Filters;
02. using Microsoft.Extensions.Logging;
```

```
    public class ExampleFilterWithDI : IActionFilter
    {
        private ILogger _logger;
        public ExampleFilterWithDI(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<ExampleFilterWithDI>
();
        }
        public void OnActionExecuting(ActionExecutingContext context)
        {
            //To do : before the action executes
            _logger.LogInformation("OnActionExecuting");
        }
        public void OnActionExecuted(ActionExecutedContext context)
        {
            //To do : after the action executes
            _logger.LogInformation("OnActionExecuted");
        }
    }
}
```

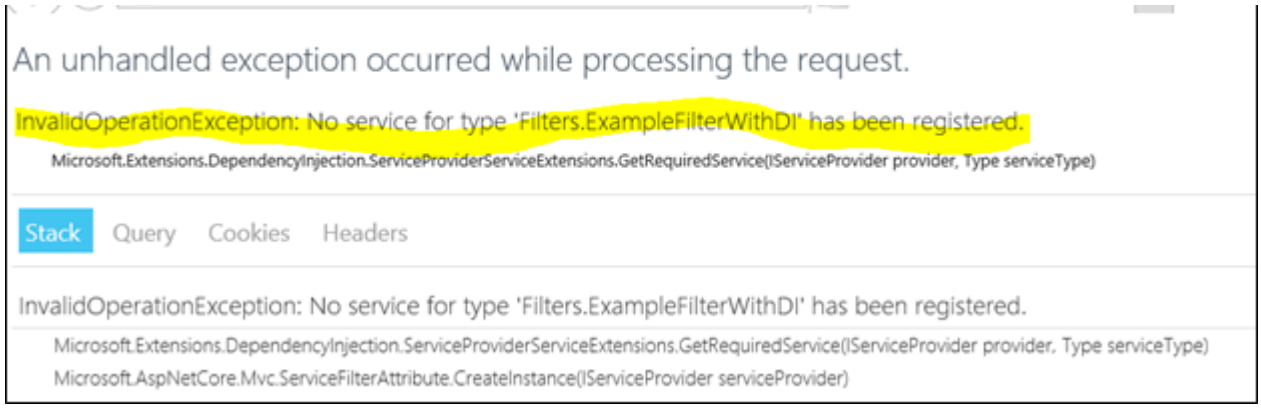
Register filter in ConfigureService method

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<ExampleFilterWithDI>();
}
```

Use filter for Action method of Controller class

```
[ServiceFilter(typeof(ExampleFilterWithDI))]
public IActionResult Index()
{
    return View();
}
```

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



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

TypeFilterAttribute

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

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

```
01. [TypeFilter(typeof(ExampleFilterAttribute), Arguments = new object[] {"Arg
02. public IActionResult About()
03. {
04.     return View();
05. }
```

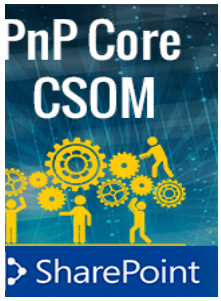
Summary

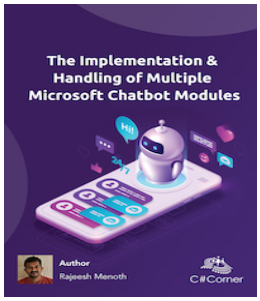
The filters allow us to run code before or after certain stages in the request processing pipeline. In this article, we learned type of built-in filter, filter scope and ordering, cancel the request from filter and how to inject the dependency in filters.

ASP.NET Core | Filters In ASP.NET Core | MVC

How To Create ASP.NET Core MVC Application

OUR BOOKS





Jignesh Trivedi TOP 50

Jignesh Trivedi is a Developer, C# Corner MVP, Microsoft MVP, Author, Blogger,eager to learn new technologies

https://www.c-sharpcorner.com/members/jignesh-trivedi

3 **35.4m** **9** **2**

14 **2**

Type your comment here and press Enter Key (Minimum 10 characters)

This is quite well written post

Bohdan Stupak

630 **3.1k** **159k**

Jul 16, 2020

1 0 Reply

Nice Article Jignesh Trivedi. Thanks for sharing us.

Rajeesh Menoth

73 **25.3k** **1.9m**

Jul 19, 2017

1 0 Reply

FEATURED ARTICLES

Understanding Synchronization Context Task.ConfigureAwait In Action

What is Non Fungible Tokens (NFT)? Why NFTs are So Popular Today?

Unit Testing With xUnit And Moq In ASP.NET Core

Using Certificates For API Authentication In .NET 5

View All

TRENDING UP

01 Use Dynamic Data Masking To Protect Sensitive Data In Azure SQL Database

02 Dynamics 365 Solution Export & Import as Managed Using AzureDevOps Build & Release Pipeline

03 <⚡> Time Triggered Azure Functions - A Guide To Background Tasks Using C#

04 Agile Methodology In Nutshell

05 Some Cool Features In C# 10

06 Caching Mechanism In ASP.NET Core

07 Implementing Unit Of Work And Repository Pattern With Dependency Injection In .Net 5

08 Difference Between HAVING And WHERE Clause In SQL Server

09 Migrating From salesforce To Microsoft Dynamics 365

10 Introduction To .NET Multi-Platform App UI (MAUI) - An Overview

View All