Working With Filters In ASP.NET Core MVC

Introduction

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

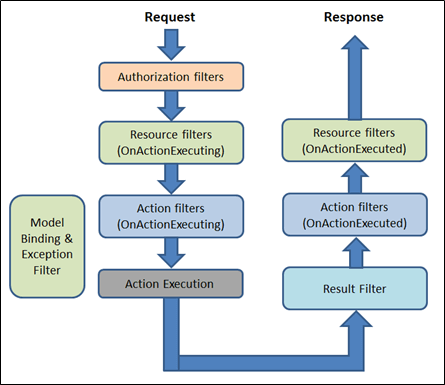
There are many built-in filters available with ASP.NET Core MVC, and we can create custom filters as well. Filters help us to remove duplicate codes in our application.

Filter Types in ASP.NET Core

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

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

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



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

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

**Synchronous Filter Example**

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class CustomActionFilter : IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

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

**Asynchronous filter example**

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class CustomAsyncActionFilter : IAsyncActionFilter

{

public async Task OnActionExecutionAsync(ActionExecutingContext context,

ActionExecutionDelegate next)

{

//To do : before the action executes

await next();

//To do : after the action executes

}

}

}

C#

Copy

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

Adding Filter scope and Order of execution

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

public void ConfigureServices(IServiceCollection services)

{

// Add framework services.

services.AddMvc(options=> {

//an instant

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

//By the type

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

});

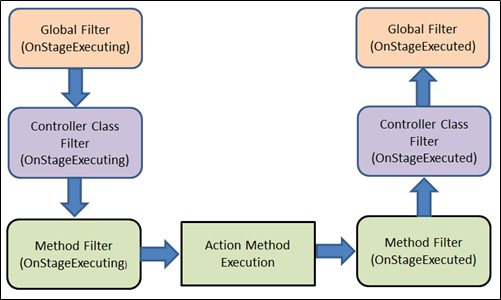
}

C#

Copy

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

Following figure shows the default order of filter execution.



Overriding the default order

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

**ExampleFilter.cs**

using System;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class ExampleFilterAttribute : Attribute, IActionFilter, IOrderedFilter

{

public int Order { get; set; }

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

**Controller.cs**

using System;

using Microsoft.AspNetCore.Mvc;

using Filters;

namespace Filters.Controllers

{

[ExampleFilter(Order = 1)]

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

}

}

C#

Copy

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

Cancellation or short circuiting filters

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

**Filter Example**

using System;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Filters;

namespace Filters

{

public class Example1FilterAttribute : Attribute, IActionFilter

{

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

context.Result = new ContentResult()

{

Content = "Short circuit filter"

};

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

}

}

}

C#

Copy

Filters and DI (Dependency Injection)

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

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

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

* ServiceFilterAttribute
* TypeFilterAttribute
* IFilterFactory implemented on attribute

**ServiceFilterAttribute**

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

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

Example is action filter with logger dependency,

using Microsoft.AspNetCore.Mvc.Filters;

using Microsoft.Extensions.Logging;

namespace FiltersSample.Filters

{

public class ExampleFilterWithDI : IActionFilter

{

private ILogger \_logger;

public ExampleFilterWithDI(ILoggerFactory loggerFactory)

{

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

}

public void OnActionExecuting(ActionExecutingContext context)

{

//To do : before the action executes

\_logger.LogInformation("OnActionExecuting");

}

public void OnActionExecuted(ActionExecutedContext context)

{

//To do : after the action executes

\_logger.LogInformation("OnActionExecuted");

}

}

}

C#

Copy

Register filter in ConfigureService method

public void ConfigureServices(IServiceCollection services)

{

services.AddScoped<ExampleFilterWithDI>();

}

C#

Copy

Use filter for Action method of Controller class

[ServiceFilter(typeof(ExampleFilterWithDI))]

public IActionResult Index()

{

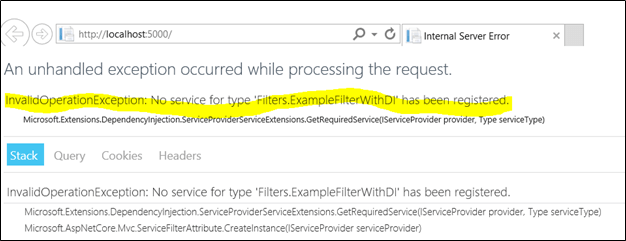
return View();

}

C#

Copy

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



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

**TypeFilterAttribute**

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

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

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

public IActionResult About()

{

return View();

}

C#

Copy

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.

Exploring ASP.NET Core Filters

 07 Aug 2024

 Advanced

 49.5K Views

 20 min read

A blue circle with white text

Description automatically generated

Learn via Video Course & by Doing Hands-on Labs

Self-Paced ASP.NET Core Course

[Free Course](https://www.scholarhat.com/course/aspnet-core-course?utm_source=course&utm_medium=start-learning&utm_campaign=https://www.scholarhat.com/tutorial/aspnet/mvc-core-filters-real-world-exmaple)[Interview Q&A](https://www.scholarhat.com/interview/aspnetcore-interview-questions-answers-course-book?utm_source=hands-on-lab&utm_medium=article&utm_campaign=https://www.scholarhat.com/tutorial/aspnet/mvc-core-filters-real-world-exmaple)[Hands-On Labs](https://www.scholarhat.com/labs/aspnetcore?utm_source=interview&utm_medium=article&utm_campaign=https://www.scholarhat.com/tutorial/aspnet/mvc-core-filters-real-world-exmaple)[Projects](https://www.scholarhat.com/projects/aspnetcore/pizza-delivery-website-project-using-aspnet-core)

ASP.NET Core Filters: An Overview

In [**Asp.Net Core**](https://www.scholarhat.com/tutorial/aspnet) applications, Controllers have action strategies and methodologies that operate once any user interacts with the interface. If the user clicks on any button the corresponding action methodology is dead. However, this action methodology does not die directly. It's to be passed in many steps. That means, it follows the route - once the user clicks on the button, the request is routed to seek out the Controller and also the corresponding action methodology is named. In this tutorial, we'll get to know **What are Filters in .NET Core?, Types of Filters with examples**and much more.

Join our [**ASP.NET Core Certification Training**](https://www.scholarhat.com/training/aspnet-core-certification-training), where your coding journey transforms into a remarkable tech adventure. Embrace the power of problem-solving and enroll today.

**Read More:**[**Top 50 ASP.NET Interview Questions and Answers**](https://www.scholarhat.com/tutorial/aspnet/aspnet-webforms-interview-questions-and-answers)

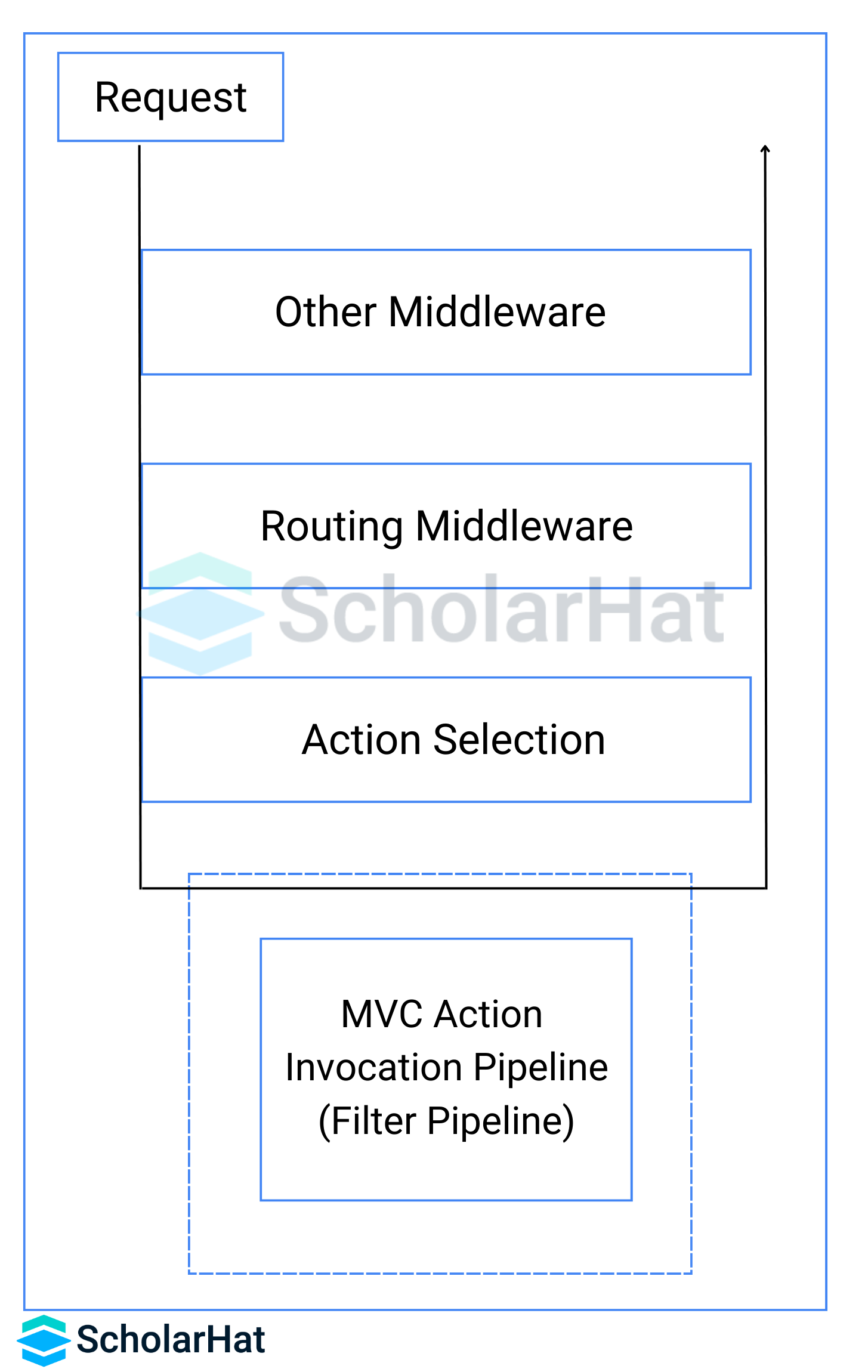
What are Filters in .NET Core?

If we would like to perform any operation before or once the action methodology is named, we've to have faith in filters. As a result, filters are used for performing arts pre- and post-logic before and once the action methodology gets dead.[ASP.NET Core](https://www.scholarhat.com/tutorial/aspnet/introduction-to-aspnet-core)comes with a thought of filters. Filters intercept the stages of the [MVC](https://www.scholarhat.com/tutorial/aspnet/what-is-aspnet-mvc-developer-skills-to-become-aspnet-mvc-developer)pipeline and permit us to run code before/after their execution.

Process Flow Of Filters

Filters run among the [**ASP.Net Core**](https://www.scholarhat.com/tutorial/aspnet) action invocation pipeline also referred to as the filter pipeline. The filter pipeline runs when ASP.Net Core selects the action to execute. So, when a filter is executed within the pipeline, there are always different scenarios for every execution. So, before creating a filter, we first analyze our requirements so that we can decide which filters we require exactly and in which position of the filter pipeline for executions. In Asp.Net Core, the filter always executes from the MVC Action method which is known as the Filter Pipeline and it will be executed when the action method is executed.

Different filter kinds run at all completely different points along the pipeline. In the filter pipeline, some filters are executed before the execution of the next level like Authorization filters. Also, some filters are executed before and after the state of execution in the filter pipeline. Action filters are one of the examples of these types of filters.



**Read More:**[**Salary Offered to ASP.NET Developers**](https://www.scholarhat.com/tutorial/aspnet/salary-offered-to-asp-net-developers)

Types Of Filters

In Asp.Net Core, filter execution depends on the MVC Action pipeline. Normally, the filter pipeline is executed when any action of the MVC Controller needs to be executed or execution is already done. There are different types of filters in the Asp.Net Core. Below are the different filter types and their importance in the process flow.

**Authorization Filters**

[A screenshot of a computer program

Description automatically generated](https://www.scholarhat.com/training/aspnet-core-certification-training?utm_source=scholarhat&utm_medium=banner&utm_campaign=https://www.scholarhat.com/tutorial/aspnet/mvc-core-filters-real-world-exmaple)

The Authorization filters performance measure dead initial. This filter helps us to work out whether or not the user is allowed for the present request. If a user is not authorized for the request, then this filter will break the pipeline process flow. We can additionally produce a custom authorization filter.

[Authorize]

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

{

// Action logic

}

**Resource Filters**

The Resource filters handle the authorized request in the process flow. This filter can execute before and after the process flow in the filter execution. Normally, this type of filter activates before the model data binding at the controller level. For example, OnResourceExecuting runs code before model binding. This filter can be used for caching implementations. IResourceFilter or IAsyncResourceFilter interfaces are implemented by this filter.

**public** **class** **MyResourceFilter** : **IResourceFilter**

{

**public** **void** **OnResourceExecuting**(ResourceExecutingContext context)

{

// Before action execution

}

**public** **void** **OnResourceExecuted**(ResourceExecutedContext context)

{

// After action execution

}

}

**Action Filters**

If we want to execute the filter code instantly with the controller action method, we need to use the Action filters. We can use the action filter before or after the execution of any controller action method. We can additionally manipulate the arguments passed into Associate in Nursing action. Action filters normally implement the IActionFilter interface or IAsyncActionFilter interface.

**public** **class** **MyActionFilter** : **IActionFilter**

{

**public** **void** **OnActionExecuting**(ActionExecutingContext context)

{

// Before action execution

}

**public** **void** **OnActionExecuted**(ActionExecutedContext context)

{

// After action execution

}

}

**Exception Filters**

If we want to track any type of exception at the time of code execution and then return that exception message to the process from where a request has been raised, we need to use Exception filters. These types of filters can be implemented with the help of the IExceptionFilter or IAsyncExceptionFilter interface. This type of filter is normally used to handle common error-trapping messages or [logging](https://www.scholarhat.com/tutorial/aspnet/logging-in-net-core-a-comprehensive-guide) in any application.

**public** **class** **MyExceptionFilter** : **IExceptionFilter**

{

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

{

// Exception handling logic

}

}

**Result Filters**

If we want to track the result of any controller action method then we need to use the Result filter. They're dead given that the controller action technique has been dead with success. With the help of the IResultFilter or IAsyncResultFilter interface, we can define the Result filters.

**public** **class** **MyResultFilter** : **IResultFilter**

{

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

{

// Before result execution

}

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

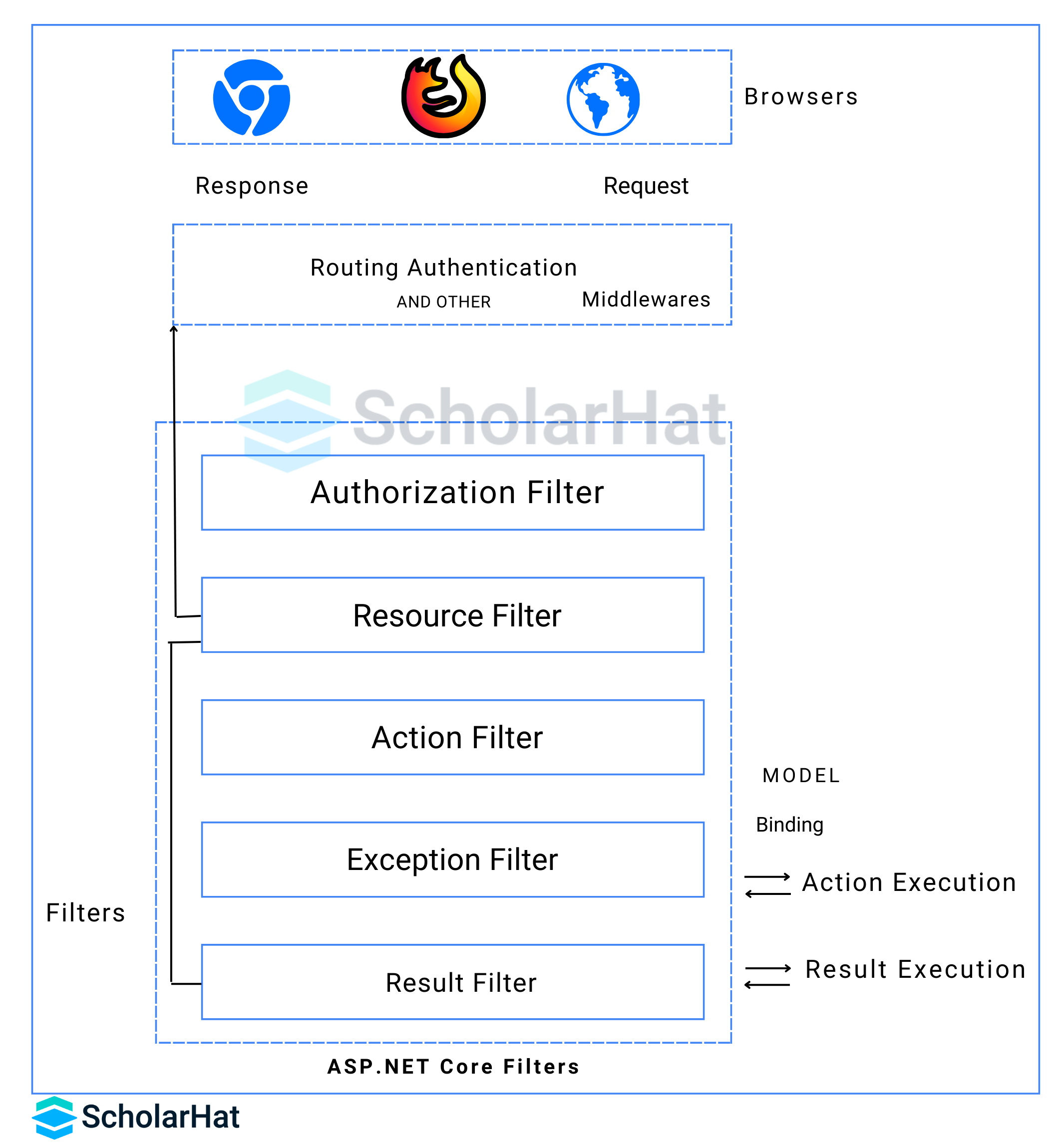
{

// After result execution

}

}

The below image demonstrates the process flow including execution flow in Asp.Net Core –



Implementation of Filters in ASP.NET Core

We can implement filters in two different ways.

1. **Synchronous filters:** They normally execute the code before the execution of the action methods OnStageExecuting and OnStageExecuted. An example of the Synchronous filter is ActionFilter. The OnActionExecuting is called before the action method is called and OnActionExecuted after the action method returns.
2. **namespace** **CoreDemoSamples.Filters**
3. {
4. **public** **class** **MyActionFilter** : **IActionFilter**
5. {
6. **public** **void** **OnActionExecuting**(ActionExecutingContext objContext)
7. {
8. }
9. **public** **void** **OnActionExecuted**(ActionExecutedContext objContext)
10. {
11. }
12. }
13. }
14. **Asynchronous filters:** It defines an OnStageExecutionAsync methodology. This type of filter method execution always depends on the FilterTypeExecutionDelegate delegate. As demonstrated in the below code, the ActionExecutionDelegate parameter is responsible for the execution call of the next filter action. Also, we can execute the action method code before the filter executions.
15. **namespace** **CoreDemoSamples.Filters**
16. {
17. **public** **class** **CustomAsyncActionFilter** : **IAsyncActionFilter**
18. {
19. **public** **async** Task **OnActionExecutionAsync**(
20. ActionExecutingContext objContext,
21. ActionExecutionDelegate objNext)
22. {
23. }
24. }
25. }

Multiple Filter Stages

Interfaces for multiple filter stages can be implemented in a single class. For example, the ActionFilterAttribute class implements:

* **Synchronous:** IActionFilter and IResultFilter
* **Asynchronous:** IAsyncActionFilter and IAsyncResultFilter
* IOrderedFilter

Implement either the synchronous or the async version of a filter interface, not both. The runtime checks first to see if the filter implements the async interface, and if so, it calls that. If not, it calls the synchronous interface's method(s). If both asynchronous and synchronous interfaces are implemented in one class, only the async method is called. When using abstract classes like ActionFilterAttribute, override only the synchronous methods or the asynchronous methods for each filter type.

Filter Scopes and Order of Execution

In Asp.Net Core, the filters are often side to the pipeline at one in every of three different scopes

1. by action methodology
2. by controller category
3. By global declaration means we can use or apply those filters either at the controller level, action method level, or both.

If we want to register the filters, then we can define those within the ConfigureServices method with the help MvcOption.Filters.

**public** **void** **ConfigureServices**(IServiceCollection objServices)

{

objServices.AddMvc(options=> {

//an instant

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

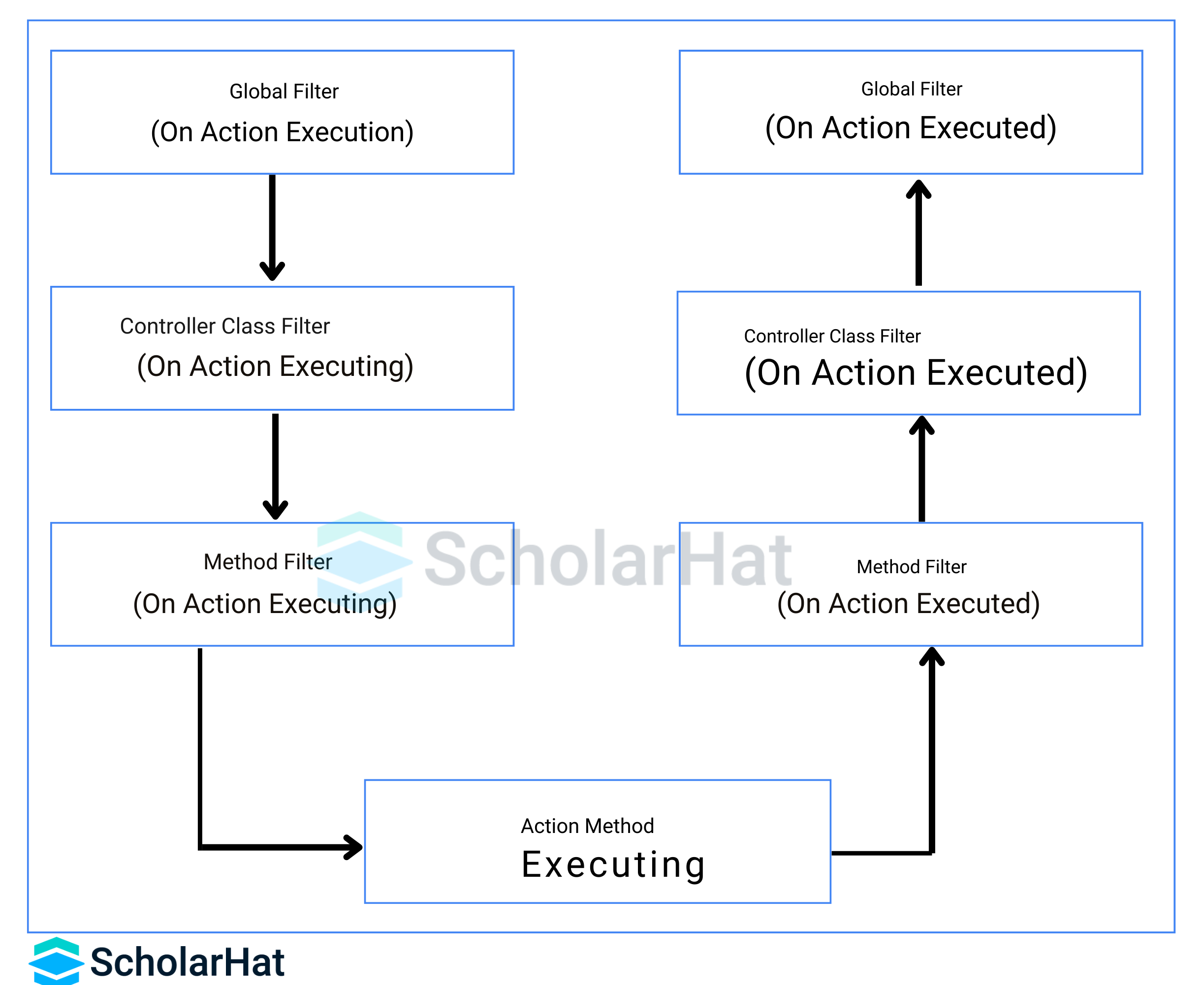
//By the type

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

});

}

When multiple filter area units are applied to the actual stage of the pipeline, the scope of the filter defines the default order of the filter execution. In the filter process flow, the first global level filter is executed first, then it is executed the controller level filters and at last, it will execute the action method level filters. The below image displays the filter process execution order.



If we want to override the process execution order of the filter, then we can perform that with the help IOrderedFilter interface. This interface has the property named Orderwhich is used to work out the order of execution. In this case, the process flow normally executes the filters in ascending order means from lower order to higher order. We will set up the order property exploitation of the creator parameter.

**public** **class** **DemoFilterAttribute** : **Attribute**, **IActionFilter**, **IOrderedFilter**

{

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

**public** **void** **OnActionExecuting**(ActionExecutingContext objContext)

{

//before the action executes

}

**public** **void** **OnActionExecuted**(ActionExecutedContext objContext)

{

//after the action executes

}

}

Now, this DemoFilterAttribute can be applied in the controller action method as below –

[DemoFilter(SequenceNo = 1)]

**public** **class** **UserController** : **BaseController**

{

**public** IActionResult **HomeIndex**()

{

**return** View();

}

}

In the normal scenario, the order sequence of all the inbuilt filters is 0. If we want to create any custom filter, then the order sequence of that filter must start from 1. Because, at the execution time, it will soften the filter list based on Order and then start execution as per the sorted filter list.

Filter Dependency Injection

Filters will be added by sort or by instance. In the case of the instance, each request is used for this instance. If you add a sort, it'll be type-activated, which means an instance is created for every request and any constructor dependencies are inhabited by [dependency injection](https://www.scholarhat.com/tutorial/aspnet/dependency-injection-implementation-asp-net-core-mvc) (DI). Adding a filter by sort is equivalent to the constructor of a class. Filters that area unit enforced as attributes and add on to controller categories or action ways cannot have builder dependencies provided by dependency injection (DI).

If we develop some filters that contain dependency, then we need to use that dependency with the help of dependency injection. You can apply your filter to a category or action methodology victimization one in every of the following

1. ServiceFilterAttribute
2. TypeFilterAttribute
3. IFilterFactory

In the case of ServiceFilter, we can filter instances with the help of [Dependency Injection](https://www.scholarhat.com/tutorial/aspnet/dependency-injection-implementation-asp-net-core-mvc). To activate this filter, we first need to add this filter with the help of ConfigureService and then, we can use the reference of this filter either in the controller class or action method as a ServiceFilter. One of the dependencies we would need to urge from the DI could be a faller among filters, we would log one thing happened. An example is an action filter with faller dependency.

**public** **class** **DemoFilterForDI** : **IActionFilter**

{

**private** ILogger \_objLogger;

**public** **DemoFilterForDI**(ILoggerFactory objFactory)

{

\_objLogger = objFactory.CreateLogger<DemoFilterForDI>();

}

**public** **void** **OnActionExecuting**(ActionExecutingContext objContext)

{

//before the action executes

\_objLogger.LogInformation("OnActionExecuting");

}

**public** **void** **OnActionExecuted**(ActionExecutedContext objContext)

{

//after the action executes

\_objLogger.LogInformation("OnActionExecuted");

}

}

Now, register this filter in the configuration service method as below

**public** **void** **ConfigureServices**(IServiceCollection objService)

{

objService.AddScoped<DemoFilterForDI>();

}

Now, use the above service filter in the action method of the controller

[ServiceFilter(typeof(DemoFilterForDI))]

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

{

**return** View();

}

It is a terrible kind of ServiceFilterAttribute and conjointly enforced from the IFilterFactory interface. In this case, it does resolve directly from the dependency injection, but it initializes the type with the help of "Microsoft.Extensions.DependencyInjection.ObjectFactory". Due to this difference, we need to register types within the ConfigureService method which are already referenced in TypeFilterAttribute.

The TypeFilterAttribute will optionally settle for builder arguments for the sort. The following example demonstrates the way to pass arguments to a sort victimization TypeFilterAttribute.

[TypeFilter(typeof(DemoFilterAttribute), Arguments = new object)]

**public** IActionResult **AboutIndex**()

{

**return** View();

}

Security Considerations with Filters

Security is always a major concern even while you're working with filters in ASP.NET Core so as to make sure that all your sensitive data stays protected and there is proper authorization and [authentication](https://www.scholarhat.com/tutorial/aspnet/authentication-authentication-aspnet-identity-example).

A. Preventing Overuse of Filters

* Overuse of filters may increase the chances of decrease in performance and can also lead to attack surface.
* To avoid unnecessary processing, try to apply filters only where they are really required, if not then avoid using them.
* If we put multiple filters for the same action, they can lead to redundancy resulting in unpredictable behavior and unnecessary security risks.

B. Ensuring Proper Authorization and Authentication Checks

* It is advised to always make sure that there is proper authorization and authentication done.
* You can authorization filters to make sure that all the sensitive data is protected to prevent it from unauthorized access.
* Implementing authentication logic along with filters will help in verifying the identities of the users that are trying to access the protected resources.

C. Handling Sensitive Data in Filters

* You should always perform data validation and sanitization so as to prevent vulnerabilities such as injection attacks whenever filters interact with user input.
* Using encryption techniques in both transit and at rest will make sure that all the sensitive data stays protected and confidential.
* You should hardcoding sensitive information like API keys, they can be stored securely in environment variables or encrypted configuration stores.

Summary

The filters enable us to run code before or once bound stages within the request process pipeline. In this article, we discuss the process flow of filters, different types of filters, how it is executed, etc. Also, discussed the ordering of filters, filter scope, how to cancel the request from the filter, and how to implement dependency injection using filters. Enroll yourself In the [**Best Online ASP.NET Core Course**](https://www.scholarhat.com/training/aspnet-core-certification-training)from **[ScholarHat](https://www.scholarhat.com/" \t "_blank)** and Start Your Tech Career the Right Way!

FAQs

Q1. How filters work in ASP.NET Core?

Filters run within the ASP.NET Core action invocation pipeline sometimes referred to as the filter pipeline. The filter pipeline runs after ASP.NET Core selects the action to execute

Q2. What is the other name of invocation pipeline?

It is also referred to as the filter pipeline.

Q3. When Action filters execute?

It runs immediately before and after an action method is called.

Q4. How many types of filters are there in asp .net core?

There are 5 types of filters in ASP.NET Core:

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

Q5. How can I control the order of execution for multiple filters in ASP.NET Core?

To control the order of execution for multiple filters in ASP.NET Core, you can specify it with the help of 'Order' property of the filter attribute or 'IOrderedFilter' interface.