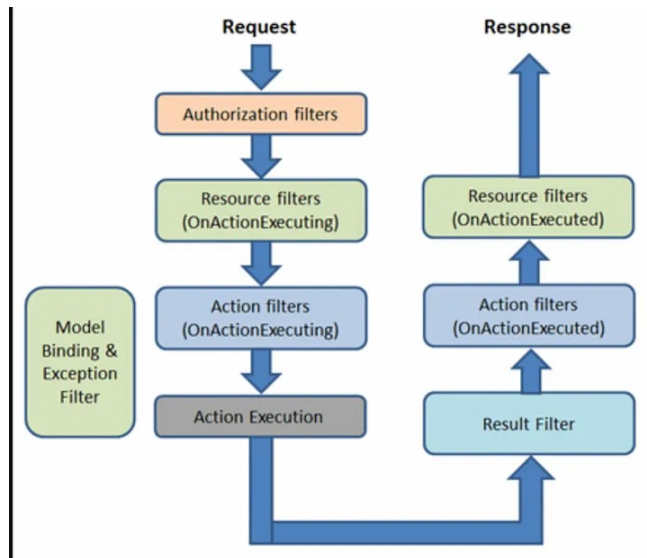


# Filters in .Net Core



Dileep Sreepathi · Follow

6 min read · Jan 15, 2024



Welcome, fellow developers, to a deep dive into the fascinating realm of filters in .NET Core! Filters play a crucial role in shaping the behavior of your applications, providing a powerful mechanism to execute code before or after certain stages in the request-handling process. They can be used for

## Medium

Sign up to discover human stories that deepen your understanding of the world.

### Free

- ✓ Distraction-free reading. No ads.
- ✓ Organize your knowledge with lists and highlights.
- ✓ Tell your story. Find your audience.

Sign up for free

### Membership

- ✓ Read member-only stories
- ✓ Support writers you read most
- ✓ Earn money for your writing
- ✓ Listen to audio narrations
- ✓ Read offline with the Medium app

Try for \$5/month

- **Authorization filters:** run first and determine whether the user is authorized for the request. They can short-circuit the pipeline if the request is not authorized. They are executed early in the pipeline before the action method is invoked. This is the first line of defense for securing your application.

```
public class CustomAuthorizeFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        // Your authorization logic goes here
    }
}
```

- **Resource filters:** run after authorization and can perform tasks before and after the rest of the pipeline. For example, they can implement caching or model binding logic.

```
public class CustomResourceFilter : IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        // Code executed before the action method
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        // Code executed after the action method
    }
}
```

- **Action filters:** run before and after an action method is invoked. They can modify the arguments passed to the action or the result returned from the action. This is where you can manipulate the action's parameters, inspect or modify the result, and perform additional logic.

```
public class CustomActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Code executed before the action method
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Code executed after the action method
    }
}
```

- **Exception filters:** run when an unhandled exception occurs before the response body is written. They can handle the exception and return a custom response. Here, this filter is applied only for action methods, resource filters (for any database file access issues), and model binding. The Exception filter is not application on Result Filter so for this we need to use the Exceptiton middleware concept. Leran about the exception middle [here](#) :

```
public class CustomExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        // Handle and log the exception
    }
}
```

- **Result filters:** run before and after the execution of action results. They can modify the result or perform additional actions such as logging or auditing.

```
public class CustomResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Code executed before the result is executed
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Code executed after the result is executed
    }
}
```

Filters can be applied globally to all actions in the application, or selectively to specific controllers or actions using attributes. For example, the `[Authorize]` attribute is a built-in authorization filter that requires the user to be authenticated. The `[ResponseCache]` attribute is a built-in resource filter that enables response caching for the action.

**Filters can be categorized into two types:** built-in filters and custom filters. Built-in filters are provided by ASP.NET Core and cover common scenarios

such as authorization, caching, exception handling, and more. Custom filters are created by developers and can implement any logic that is needed for the application.

The following table summarizes the built-in filters and their use cases:

Filter Type	Filter Interface	Filter Attribute	Use Case
Authorization	<code>IAuthorizationFilter</code>	<code>[Authorize]</code>	Require the user to be authenticated or have a specific role or policy
Resource	<code>IResourceFilter</code>	<code>[ResponseCache]</code>	Enable response caching for the action
Resource	<code>IResourceFilter</code>	<code>[ValidateAntiForgeryToken]</code>	Validate anti-forgery tokens for POST requests
Action	<code>IActionFilter</code>	<code>[ServiceFilter]</code>	Inject a service from the dependency injection container into the filter
Action	<code>IActionFilter</code>	<code>[TypeFilter]</code>	Inject services or parameters into the filter using constructor injection
Exception	<code>IExceptionHandler</code>	<code>[HandleError]</code>	Handle exceptions and return a custom error view
Result	<code>IResultFilter</code>	N/A	Modify the result or perform additional actions such as logging or auditing

To create a custom filter, you need to implement one or more filter interfaces and override the corresponding methods. For example, the following code shows a simple action filter that logs the execution time of the action:

```
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;
using System.Diagnostics;

public class LogTimeFilter : IActionFilter
{
    private readonly ILogger<LogTimeFilter> _logger;
    private Stopwatch _stopwatch;

    public LogTimeFilter(ILogger<LogTimeFilter> logger)
    {
        _logger = logger;
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Run before the action executes
        _stopwatch = Stopwatch.StartNew();
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Run after the action executes
        _stopwatch.Stop();
        _logger.LogInformation($"Action {context.ActionDescriptor.DisplayName} took {_stopwatch.Elapsed} to execute");
    }
}
```

To apply a custom filter to a controller or action, you can use the `[ServiceFilter]` or `[TypeFilter]` attributes, or register the filter globally in the `Startup` class. For example, the following code shows how to register the `LogTimeFilter` globally:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
        services.AddScoped<LogTimeFilter>(); // Register the filter as a scoped service
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller}/{action}/{id?}");
        });
    }
}
```

```

        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });

    app.UseMvc(config =>
    {
        config.Filters.AddService<LogTimeFilter>(); // Add the filter global
    });
}
}

```

## Execution Flow of Filters

The order of execution of filters is important because it can affect the behavior and outcome of the request. For example, if an authorization filter short-circuits the pipeline, the subsequent filters will not run. Similarly, if an exception filter handles an exception, the result filters will not run.

By default, ASP.NET Core executes filters in the following order:

- **Global** filters, in the order they are registered in the `Startup` class
- **Controller** filters, in the order they are defined on the controller class
- **Action** filters, in the order they are defined on the action method

Within each filter type, the order of execution is determined by the `Order` property of the filter attribute or the filter interface. The lower the value of the `Order` property, the earlier the filter runs. The default value of the `Order` property is zero, which means the filter runs before any other filter with the same filter type and a higher order value.

For example, the following code shows how to specify the order of execution for two authorization filters on the same action:

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet]
    [Authorize(Roles = "Admin", Order = 2)] // Run second
    [Authorize(Policy = "Premium", Order = 1)] // Run first
    public IActionResult Get()
    {
        // Return the list of products
    }
}

```

In this case, the `[Authorize(Policy = "Premium")]` filter will run before the `[Authorize(Roles = "Admin")]` filter, because it has a lower order value. If both filters pass, the action will execute. If either filter fails, the action will not execute and the pipeline will be short-circuited.

Filters are game-changers for your .NET Core applications. They offer a plethora of benefits:

- **Code Reusability:** Write common logic once and apply it across multiple actions, reducing code duplication and enhancing maintainability.
- **Improved Performance:** Cache responses, short-circuit unnecessary processing, and handle errors effectively, leading to a smoother, faster application.
- **Simplified Development:** Focus on core business logic while leaving cross-cutting concerns like authorization and error handling to filters.
- **Enhanced Security:** Implement robust authorization and input validation checks with filters, adding an extra layer of protection to your application.

Mastering the art of filters in .NET Core empowers you to build more

modular, maintainable, and secure applications. By understanding the different types of filters available and their execution flow, you can harness their full potential to shape the behavior of your application.

Experiment with these filters, apply them strategically and witness the transformation in your application's architecture. Filters are not just code; they are the architects of a robust and flexible system. So go ahead, dive into the world of filters, and elevate your .NET Core development to new heights! Happy coding!

Stackademic


Thank you for reading until the end. Before you go:


- Please consider **clapping** and **following** the writer! 🙌
- Follow us [X](#) | [LinkedIn](#) | [YouTube](#) | [Discord](#)
- Visit our other platforms: [In Plain English](#) | [CoFeed](#) | [Venture](#)


Dot Net Core


Filters


Web Api Development

 7










Written by Dileep Sreepathi

126 Followers · 29 Following

Full Stack Developer | Tech Enthusiastic in the field of Computer science and technology , <https://www.linkedin.com/in/dileep-sreepathi>

Follow




No responses yet



What are your thoughts?

Respond

More from Dileep Sreepathi

 Dileep Sreepathi

OOPs Concepts C# — Object Oriented Programming

The better way to explain Object Oriented Programming is to think of the OOPs in real-...

Jul 25, 2023

 102

 3




 In Cloud Native Daily by Dileep Sreepathi

Factory Design Pattern in C# for .NET Core Projects

In the world of software development, design patterns play a pivotal role in creating well-...

Aug 2, 2023

 29



 Dileep Sreepathi

Middleware in .NET Core

 Dileep Sreepathi

Angular CRUD App

In the world of web development, middleware plays a crucial role in handling requests and...

Aug 14, 2023 5



Angular is a framework for building web applications using TypeScript, a superset of...

Feb 4 7



See all from Dileep Sreepathi

## Recommended from Medium

Ravi Patel

### ASP.NET Core MVC CRUD Operations with Product Example...

In this detailed blog, we will walk through how to implement CRUD operations (Create,...

Sep 25 2



Yohan Malshika

### Rate Limiting in ASP.NET Core API

How to config Rate Limiting in ASP.NET Core Web API

Oct 21 60



## Lists

### Staff picks

791 stories · 1533 saves

### Stories to Help You Level-Up at Work

19 stories · 901 saves

### Self-Improvement 101

20 stories · 3168 saves

### Productivity 101

20 stories · 2682 saves

CodeWithHonor

### Authentication and Authorization in .NET

1. Introduction

Oct 24 71



Kamlesh Singh

### Run, Use, and Map Methods in .NET Core Pipeline

In .NET Core, "Run", "Map", and "Use" are all methods that help you configure the...

Oct 5 17



In CodeX by Justin Muench

### LINQ 8 Ways: How to Get the Most Out of LINQ (and What's New...

LINQ (Language Integrated Query) has been a developer's best friend for years, making...

Dec 22 5



Richard Nwonah

### Creating and Using Custom Attributes in C# for ASP.NET Core

Introduction

Aug 24 18



[See more recommendations](#)

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)