



<Coding[🎵]onata />

6 Types of Filters in ASP.NET Core

ASP.NET Core Filters

In ASP.NET Core, filters are components that let you run code **before or after** certain stages in the request processing pipeline.

These filters apply on **Controllers** within an ASP.NET Core Project, except for 1 which applies only on minimal APIs

In fact, minimal APIs supports **only 1 type of filter**

Keep reading to learn more...

@AramT87

1

Authorization Filter

Authorization Filter

Runs before anything else (even before action and model binding).

Use case: Check if the user is allowed to access the resource.

Example use: Role-based access control (RBAC), custom Auth checks, Multi-tenant restrictions

Interface: **IAuthorizationFilter** or **IAsyncAuthorizationFilter**

Authorization Filter

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

public class RoleAuthorizationFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        var user = context.HttpContext.User;

        if (!user.Identity?.IsAuthenticated ?? true ||
            !user.IsInRole("Admin"))
        {
            context.Result = new ForbidResult();
        }
    }
}
```

```
[ApiController]
[Route("api/[controller]")]
[TypeFilter(typeof(RoleAuthorizationFilter))]
public class AdminController : ControllerBase
{
    [HttpGet("dashboard")]
    public IActionResult GetDashboard()
    {
        return Ok("This is the admin dashboard.");
    }
}
```

2

Resource Filter

Resource Filter

It runs before model binding and after authorization.

Use Cases:

- Response caching.
- Performance metrics.
- Prevent processing based on headers.

Interface: `IResourceFilter` or `IAsyncResourceFilter`

Resource Filter

```
using Microsoft.AspNetCore.Mvc.Filters;
using System.Diagnostics;
public class RequestTimingResourceFilter : IResourceFilter
{
    private Stopwatch? _stopwatch;

    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        _stopwatch = Stopwatch.StartNew();
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        _stopwatch?.Stop();

        Console.WriteLine(
            $"Request took {_stopwatch?.ElapsedMilliseconds} ms"
        );
    }
}
```

```
[ApiController]
[Route("api/[controller]")]
[TypeFilter(typeof(RequestTimingResourceFilter))]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        return Ok(new { Id = id, Name = "Sample Product" });
    }
}
```


3

Action Filter

Action Filter

It runs just before and after the action method is invoked.

Use Cases:

- Logging input/output.
- Timing performance.
- Adding/modifying action results.

Interface: `IActionFilter` or `IAsyncActionFilter`

Action Filter

```
public class LogActionArgumentsFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        foreach (var arg in context.ActionArguments)
        {
            Console.WriteLine($"Param: {arg.Key} = {arg.Value}");
        }
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Optional: log something after action
    }
}
```

```
[ApiController]
[Route("api/[controller]")]
[TypeFilter(typeof(LogActionArgumentsFilter))]
public class UsersController : ControllerBase
{
    [HttpPost("create")]
    public IActionResult CreateUser([FromBody] UserDto user)
    {
        return Ok($"User {user.Name} created");
    }
}

public record UserDto(string Name, int Age);
```

4

Endpoint Filter

Endpoint Filter

This filter was added in .NET 7

It was designed for Minimal APIs

Allows custom logic before/after the endpoint handler.

Similar to middleware, but scoped to a single endpoint.

Endpoint Filter

```
public class SimpleLoggingFilter : IEndpointFilter
{
    public async ValueTask<object?> InvokeAsync(
        EndpointFilterInvocationContext context,
        EndpointFilterDelegate next)
    {
        var endpointName = context.HttpContext.GetEndpoint()?.DisplayName;

        Console.WriteLine($"Calling endpoint: {endpointName}");

        var result = await next(context);

        Console.WriteLine($"Endpoint completed.");

        return result;
    }
}

// Program.cs
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () =>
{
    return Results.Ok("Hello from Minimal API!");
})
.AddEndpointFilter<SimpleLoggingFilter>();

app.Run();
```

5

Exception Filter

Exception Filter

It runs If an **unhandled exception** occurs during the **action execution** or **result rendering**.

Preferred to be **applied globally** rather than on specific action or controller.

Use Cases:

- Logging exceptions.
- Returning friendly error responses.

Interface: **IExceptionFilter** or **IAsyncExceptionFilter**

Exception Filter

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

public class SimpleExceptionHandler(ILogger<SimpleExceptionHandler> logger)
    : IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
        logger.LogError(context.Exception, "Unhandled exception");

        var result = new ObjectResult("An unexpected error occurred.")
        {
            StatusCode = StatusCodes.Status500InternalServerError
        };

        context.Result = result;
        context.ExceptionHandled = true;
    }
}

//Inside Program.cs
builder.Services.AddControllers(options =>
{
    options.Filters.Add<SimpleExceptionHandler>();
});
```



Result Filter

Result Filter

After the action has returned a result but **before and after the result is processed**.

Use Cases:

- Wrapping responses.
- Adding custom headers.
- Formatting content conditionally.

Interface: **IResultFilter** or **IAsyncResultFilter**

Result Filter

```
using Microsoft.AspNetCore.Mvc.Filters;
public class CustomHeaderResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Append(
            "X-Custom-Header", "Filtered"
        );
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Optionally do something after response is sent
    }
}
```

```
[ApiController]
[Route("api/[controller]")]
[TypeFilter(typeof(CustomHeaderResultFilter))]
public class InfoController : ControllerBase
{
    [HttpGet("status")]
    public IActionResult GetStatus()
    {
        return Ok(new { Status = "Running" });
    }
}
```

Found this useful?



Consider Reposting



Thank You

Follow me for more content



Aram Tchekrekjian



Get Free tips and Tutorials in .NET and C#



Join 800+ Readers

CodingSonata.com/newsletters

<CodingSonata />