



Filters

.NET CORE

*Filters in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline. Filters help developers encapsulate cross-cutting concerns, like **exception handling** or **authorization**.*

[HTTPS://DOCS.MICROSOFT.COM/EN-US/ASPNET/CORE/MVC/CONTROLLERS/FILTERS?VIEW=ASPNETCORE-3.1](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1)

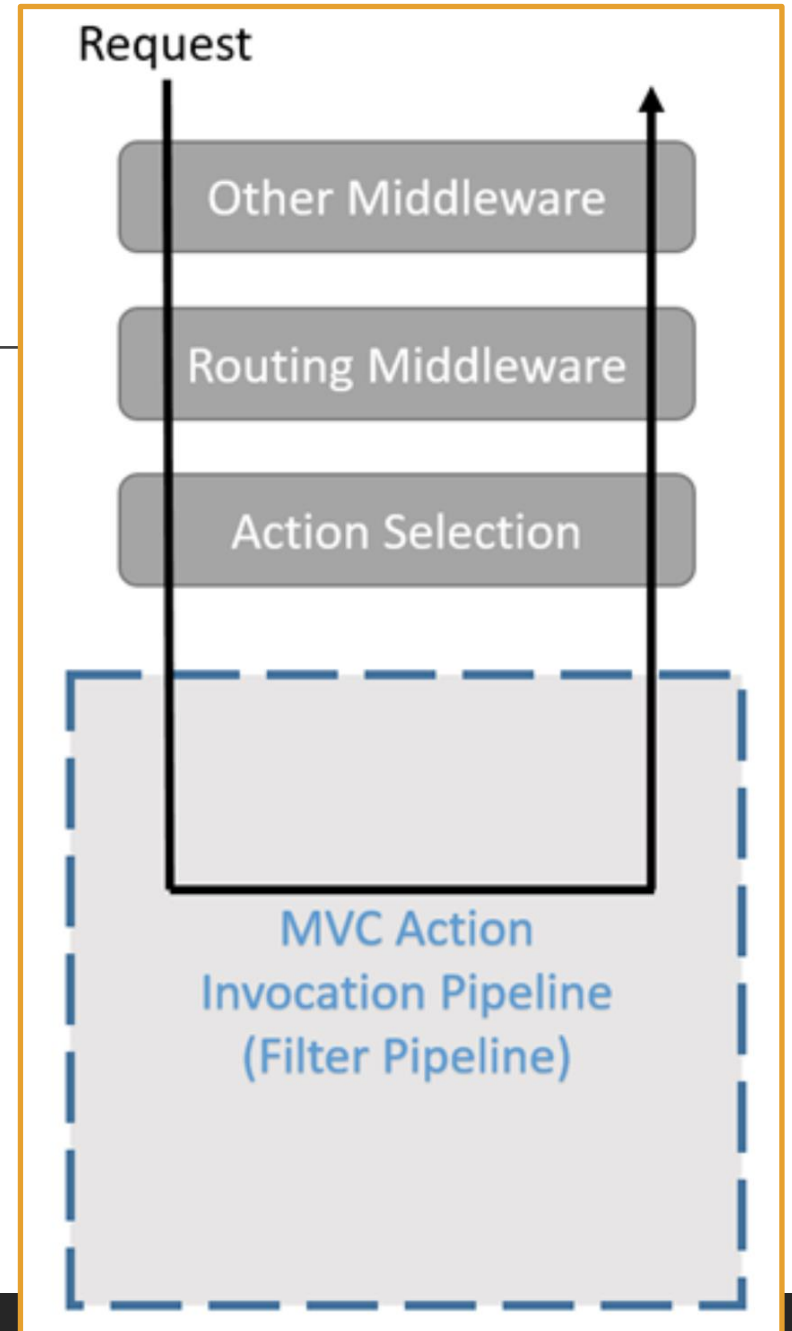
Filters – Overview

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1>

Built-in NET filters handle tasks like ***Authorization*** and ***Response caching***.

Custom filters can be created to handle cross-cutting concerns like ***error handling***, ***caching***, ***configuration***, ***authorization***, and ***logging***.

Filters run within the ASP.NET Core ***Action Invocation Pipeline***. The ***Action Invocation Pipeline*** begins running after ASP.NET Core selects the ***Action*** method to execute.



Filter Types

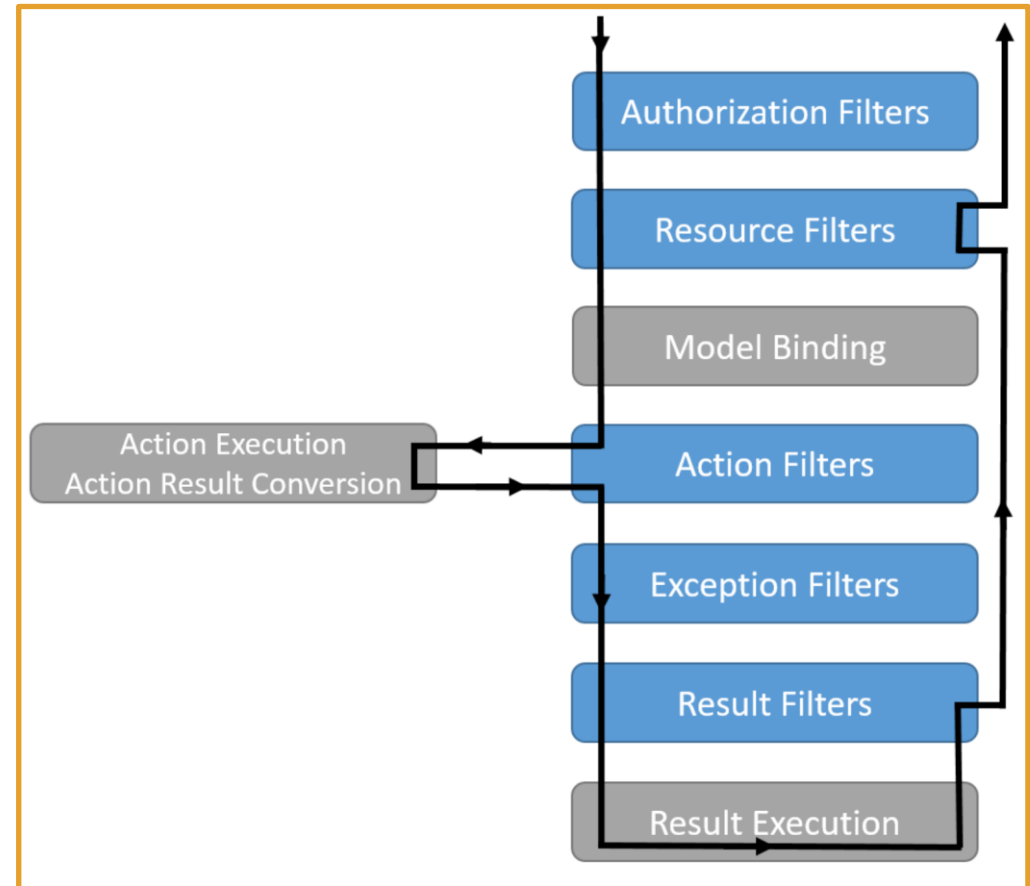
<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#action-filters>

Filter	Description
Authorization Filter	Runs first. Used to determine if the user is authorized for the request. Authorization filters stop the pipeline if the request is <u>not</u> authorized.
Resource Filter	Runs after authorization and encapsulates all other filters. OnResourceExecuting() runs code before <i>model binding</i> and OnResourceExecuted() runs after the pipeline has completed.
Action Filter	Runs OnActionExecuting() immediately before an Action method and OnActionExecuted() immediately after an <i>Action</i> method runs. These filters can change both the arguments passed into an <i>Action</i> and the <i>Result</i> returned from the <i>Action</i> .
Exception Filter	Applies global policies to <i>unhandled</i> exceptions that occur before the response body has been written to.
Result Filter	Runs immediately before and after the execution of <i>Action</i> results. They run only after the <i>Action</i> method has executed successfully.

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#filter-types>

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#using-middleware-in-the-filter-pipeline>

Even if you don't have any filters, middleware, try-catch, etc, ASP.NET Core will itself catch any exceptions within the pipeline of ***Controller, Action*** method, filters, ***View***, etc. An exception in the **Startup** class will probably crash the whole app, but that doesn't happen with exceptions anywhere else. They'll be caught and typically an **HTTP 500** will be sent.



Filter Order-of-Execution and Scope

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#default-order-of-execution>

Filters are nested inside each other during different stages of the data lifecycle of an application. How filters are nested determines their scope. *Global filters* surround *class (Controller) filters*, which surround *method (Action) filters*. As a result of filter nesting, the '**On...Executed()**' filter method runs in the reverse order of the '**On...Executing()**' filter method.

- The '**OnActionExecuting**' method of global filters.
 - The '**OnActionExecuting**' code of Controller filters.
 - The '**OnActionExecuting**' code of Action method filters.
 - The '**OnActionExecuted**' code of Action method filters.
 - The '**OnActionExecuted**' code of Controller filters.
- The '**OnActionExecuted**' code of global filters.

Authorization Filter

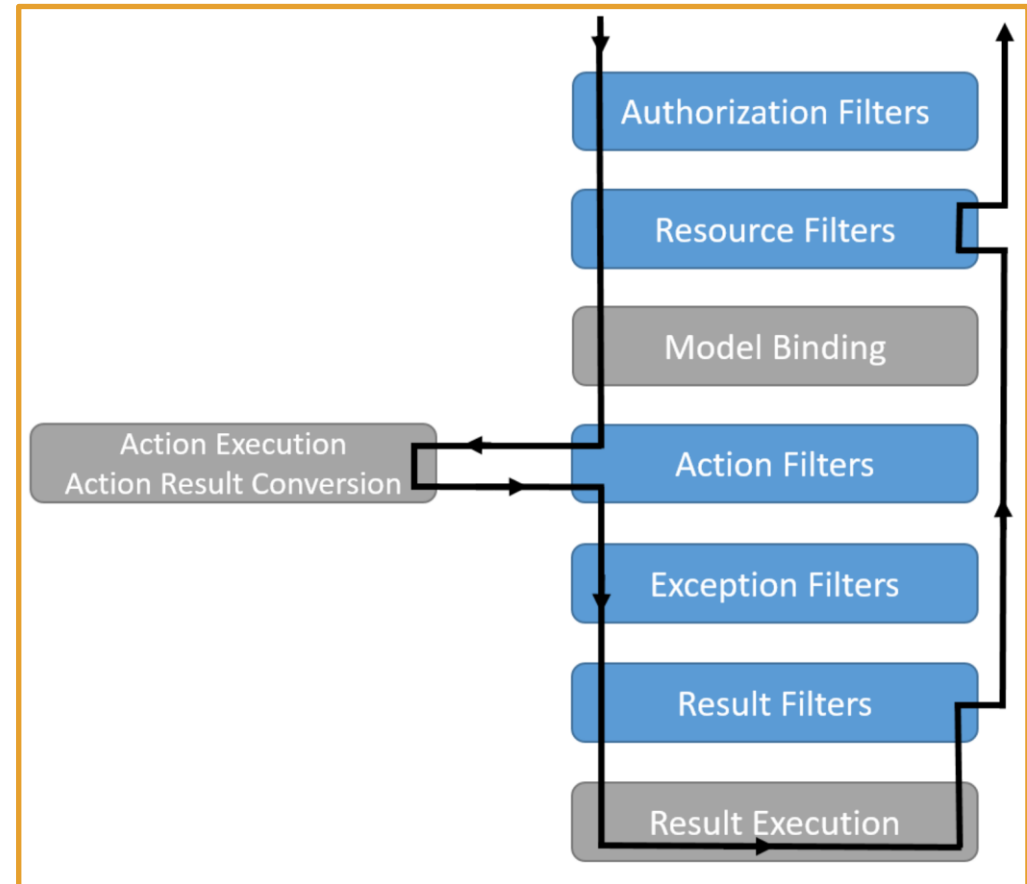
<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#authorization-filters>

Authorization filters:

- Are the first filters to run in the filter pipeline.
- Control access to **Action** methods.
- Have a **OnExecuting()** method, but no **OnExecuted()** method.

The built-in **Authorization** filter:

- Calls the authorization system.
- Does not authorize requests.
- cannot handle thrown exceptions.



Resource Filters

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#resource-filters>

Resource filters:

- Runs after the Authorization filter.
- Implement either the ***IResourceFilter*** or ***IAsyncResourceFilter*** interface.
- It wraps most of the rest of the filter pipeline.

Resource filter example:

- ***DisableFormValueModelBindingAttribute***
- Prevents model binding from accessing the form data.
- Used for large file uploads to prevent the form data from being read into memory.

```
public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult()
        {
            Content = "Resource unavailable - header not set."
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```


Action Filters

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#action-filters>

Action filters:

- Implement either the ***IActionFilter*** or ***IAsyncActionFilter*** interface.
- Their execution surrounds the execution of ***Action*** methods.

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

Action Filter - ActionExecutingContext

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#action-filters>

The **ActionExecutingContext** provides the following properties:

- **ActionArguments** - enables reading the inputs to the **Action** method.
- Access to various **Controller** Properties like **HttpContext**, **ModelState**, etc. These enable direct manipulation of the **Controller** instance and properties inherited from Controller class.
- **Result** - This gets or sets the **ActionResult** returned by the **Action** method. Setting **Result** to anything not-null skips execution of the **Action** method and subsequent **Action** filters.

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

Action Filter

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#action-filters>

After the **Action** method executes, the **ActionExecutedContext** provides **Controller** access, **Result**, and the properties **Canceled** and **Exception**.

Canceled is set to **True** if **Action** execution was short-circuited by another filter.

Exception is Non-null if the **Action** or a previously run **Action** filter threw an exception.

Setting **Exception** to null:

- Effectively handles the exception.
- **Result** is executed as if it was returned from the **Action** method.

```
public class MySampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), context.HttpContext.Request.Path);
    }
}
```

Exception Filter

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#exception-filters>

Exception filters:

- handle exceptions thrown at any previous step
- Are an alternative to error-handling middleware (e.g. *UseExceptionHandler*), which is put in *Startup.cs* and is global
- Implement *IExceptionHandler* or *IAsyncExceptionHandler*.
- Can be used to implement common error handling policies.

```
[TypeFilter(typeof(CustomExceptionHandler))]
public class FailingController : Controller
{
    [AddHeader("Failing Controller",
        "Won't appear when exception is handled")]
    public IActionResult Index()
    {
        throw new Exception("Testing custom exception filter.");
    }
}
```

This code tests the custom exception filter to the right.

```
public class CustomExceptionHandler : IExceptionHandler
{
    private readonly IWebHostEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionHandler(
        IWebHostEnvironment hostingEnvironment,
        IModelMetadataProvider modelMetadataProvider)
    {
        _hostingEnvironment = hostingEnvironment;
        _modelMetadataProvider = modelMetadataProvider;
    }

    public void OnException(ExceptionContext context)
    {
        if (!_hostingEnvironment.IsDevelopment())
        {
            return;
        }

        var result = new ViewResult { ViewName = "CustomError" };
        result.ViewData = new ViewDataDictionary(_modelMetadataProvider,
            context.ModelState);

        result.ViewData.Add("Exception", context.Exception);
        // TODO: Pass additional detailed data via ViewData
        context.Result = result;
    }
}
```

This custom exception filter uses a custom exception handler to handle exceptions that occur when the app is in development.

Result Filter

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-3.1#result-filters>

Result filters:

- Only execute when an Action produces an Action result.
- Implement an interface:
 - `IResultFilter` or `IAsyncResultFilter`
 - `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter`
- Run before and then after preparing the result to be sent and sending it a `[HttpPost]` attribute.
- Provides access to the `Canceled` and `Exception` properties

```
public class AddHeaderResultServiceFilter : IResultFilter
{
    private ILogger _logger;
    public AddHeaderResultServiceFilter(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<AddHeaderResultServiceFilter>();
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        var headerName = "OnResultExecuting";
        context.HttpContext.Response.Headers.Add(
            headerName, new string[] { "ResultExecutingSuccessfully" });
        _logger.LogInformation("Header added: {HeaderName}", headerName);
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Can't add to headers here because response has started.
        _logger.LogInformation("AddHeaderResultServiceFilter.OnResultExecuted");
    }
}
```

This code shows a result filter that adds an HTTP header: