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

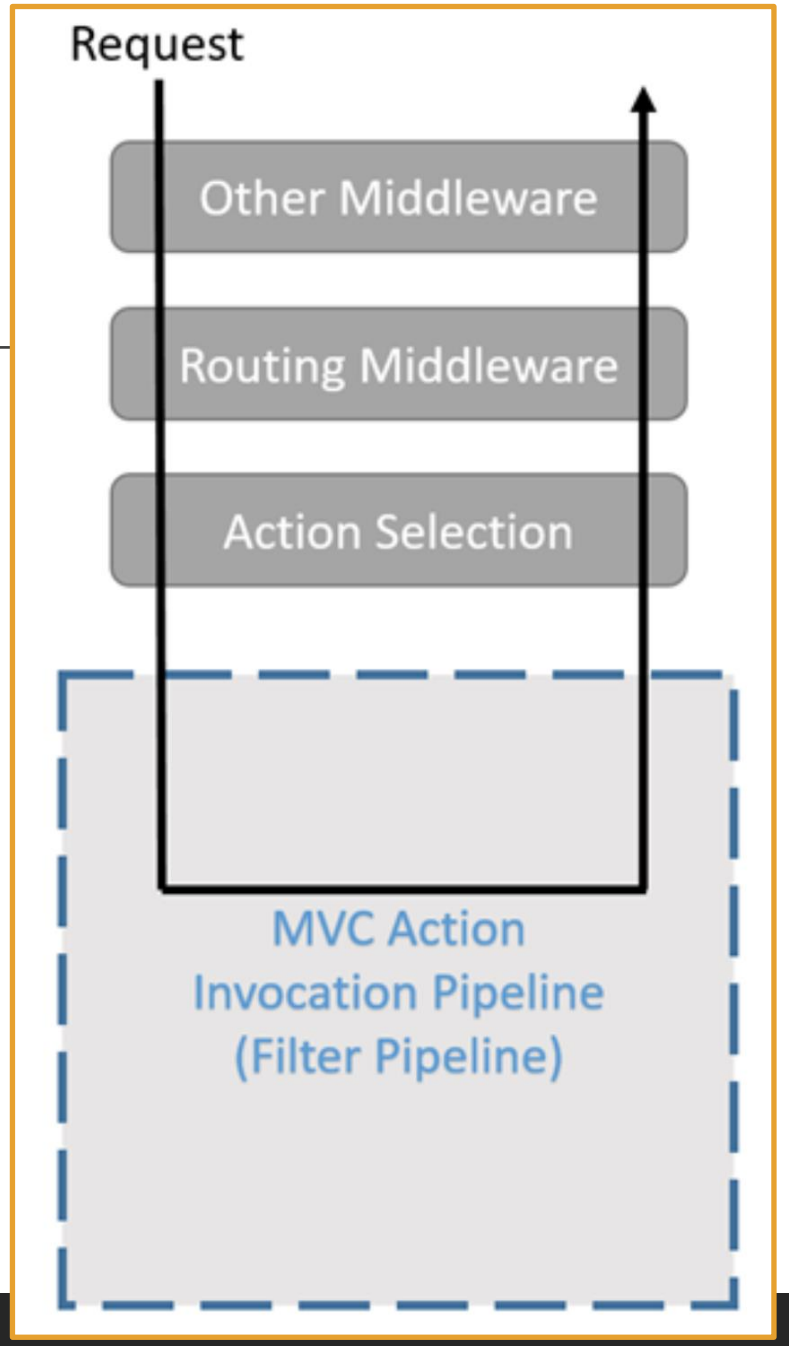
Filters – Overview

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

.NET built-in filters handle tasks such as ***Authorization*** and ***Response caching***.

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

Filters don't work directly with Razor components. Filters run within the ASP.NET Core action invocation pipeline. The action invocation pipeline runs after ASP.NET Core selects the action to execute.



Filter Types

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

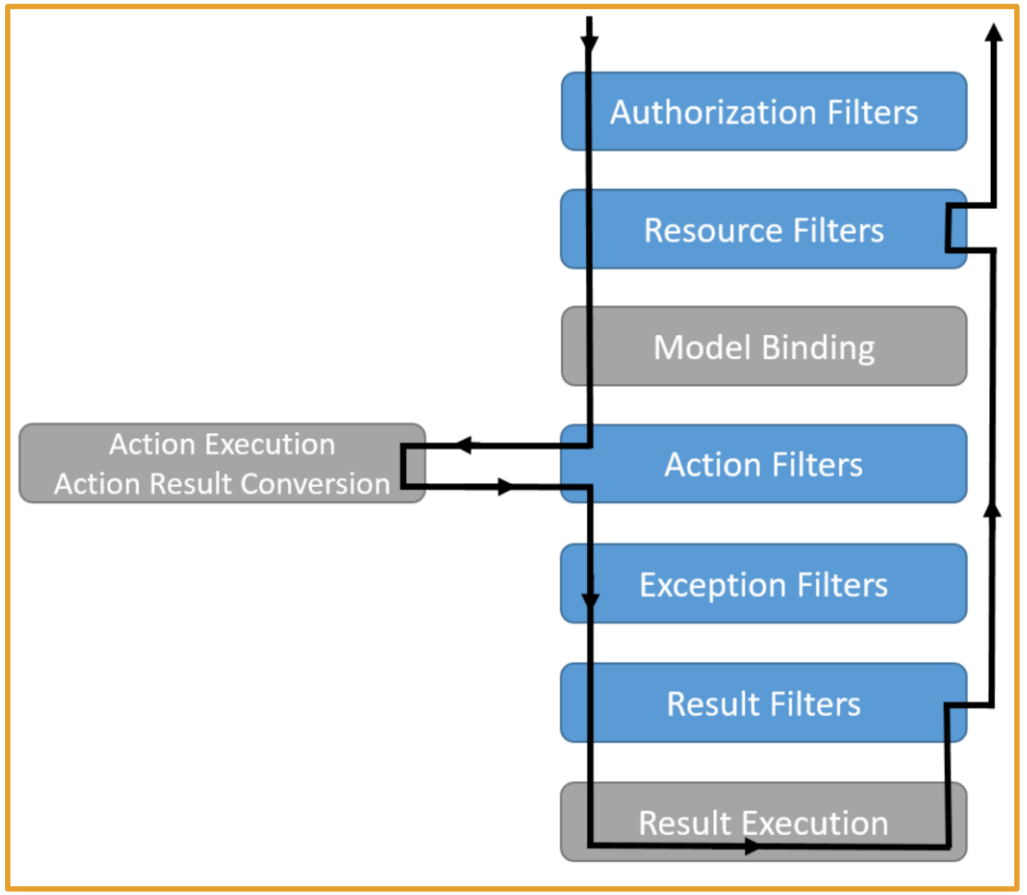
Filter	Description
Authorization	Run first. Used to determine if the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is not authorized.
Resource	Run after authorization. ***OnResourceExecuting*** runs code before model binding and after the rest of the pipeline has completed.
Action	Run code immediately before and after an action method is called. Can change the arguments passed into an action and the result returned from the action. Are not supported in Razor Pages.
Exception	Apply global policies to ***unhandled*** exceptions that occur before the response body has been written to.
Result	Run code immediately before and after the execution of action results. They run only when the action method has executed successfully.

Filter Interaction in the Filter Pipeline

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

Filters work like middleware in that they surround the execution of everything that comes later in the 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 not exceptions anywhere else. They'll be caught and typically an HTTP 500 will be sent.



Authorization Filter

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

Authorization filters:

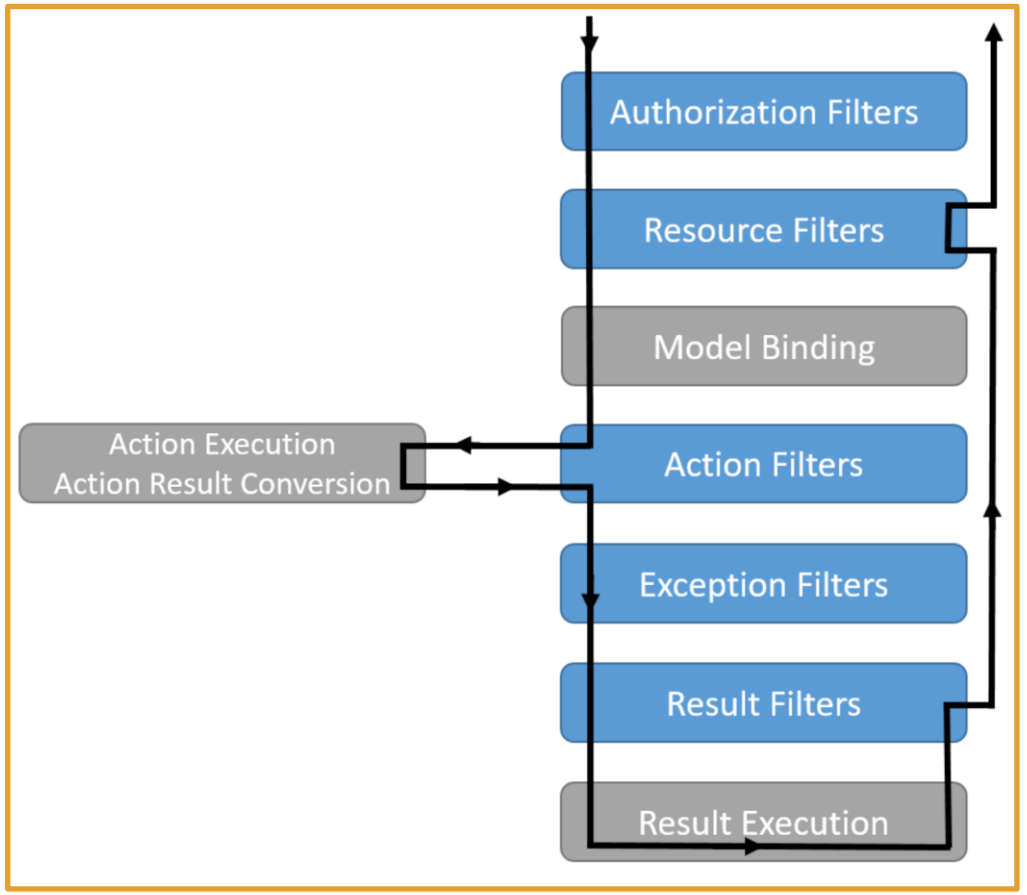
- Are the first filters to run in the filter pipeline.
- Control access to action methods.
- Have a before method, but no after method.

The built-in authorization filter:

- Calls the authorization system.
- Does not authorize requests.

Do not throw exceptions within authorization filters:

- The exception will not be handled.
- Exception filters will not handle the exception.



Resource Filters

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

Resource filters:

- Implement either the *IResourceFilter* or *IAsyncResourceFilter* interface.
- Execution wraps most of the filter pipeline.
- Only Authorization filters run before resource filters.

Resource filter examples:

- The short-circuiting resource filter.
- *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 Filter

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 an action method.
- ***Controller*** - enables manipulating the controller instance.
- ***Result*** - setting Result short-circuits execution of the action method and subsequent action filters.

Action filters:

- Implement either the ***IActionFilter*** or ***IAsyncActionFilter*** interface.
- 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

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

The ***ActionExecutedContext*** provides ***Controller*** and Result plus the following properties:
Canceled - True if the action execution was short-circuited by another filter.
Exception - Non-null if the action or a previously run ***action*** filter threw an exception. Setting this property to null:

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

Action filters:

- Implement either the ***IActionFilter*** or ***IAsyncActionFilter*** interface.
- run just before or just after the ***action*** method to manipulate inputs
- result can do server-side validation here in just one place

```
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 *IExceptionFilter* or *IAsyncExceptionFilter*.
- Can be used to implement common error handling policies.

```
[TypeFilter(typeof(CustomExceptionFilter))]
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 exception filter

```
public class CustomExceptionFilter : IExceptionFilter
{
    private readonly IWebHostEnvironment _hostingEnvironment;
    private readonly IModelMetadataProvider _modelMetadataProvider;

    public CustomExceptionFilter(
        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 sample exception filter uses a custom error view to display details about 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:

- Implement an interface:
  - IResultFilter or IAsyncResultFilter
  - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
- Run just before & just after preparing the result to be sent and sending it a [HttpPost] attribute.

```
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: