Filters in ASP.NET Core

05/13/2021 • 45 minutes to read •  +25

In this article

How filters work

Implementation

Filter scopes and order of execution

Cancellation and short-circuiting

Dependency injection

Authorization filters

Resource filters

Action filters

Exception filters

Result filters

IFilterFactory

Using middleware in the filter pipeline

Thread safety

Next actions

By Kirk Larkin , Rick Anderson , Tom Dykstra , and Steve Smith

Filters in ASP.NET Core allow code to be run before or after specific stages in the request processing pipeline.

Built-in filters handle tasks such as:

- Authorization, preventing access to resources a user isn't authorized for.
- Response caching, short-circuiting the request pipeline to return a cached response.

Custom filters can be created to handle cross-cutting concerns. Examples of cross-cutting concerns include error handling, caching, configuration, authorization, and logging. Filters avoid duplicating code. For example, an error handling exception filter could consolidate error handling.

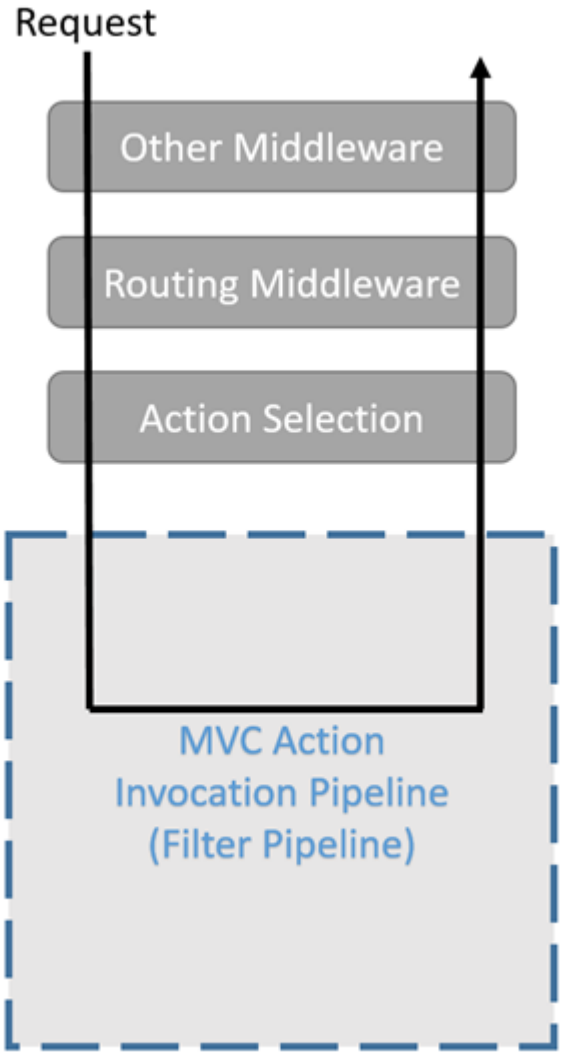
This document applies to Razor Pages, API controllers, and controllers with views. Filters don't work directly with Razor components. A filter can only indirectly affect a component when:

- The component is embedded in a page or view.
- The page or controller and view uses the filter.

View or download sample (how to download).

How filters work

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.



Filter types

Each filter type is executed at a different stage in the filter pipeline:

- Authorization filters run first and are used to determine whether the user is authorized for the request. Authorization filters short-circuit the pipeline if the request is not authorized.

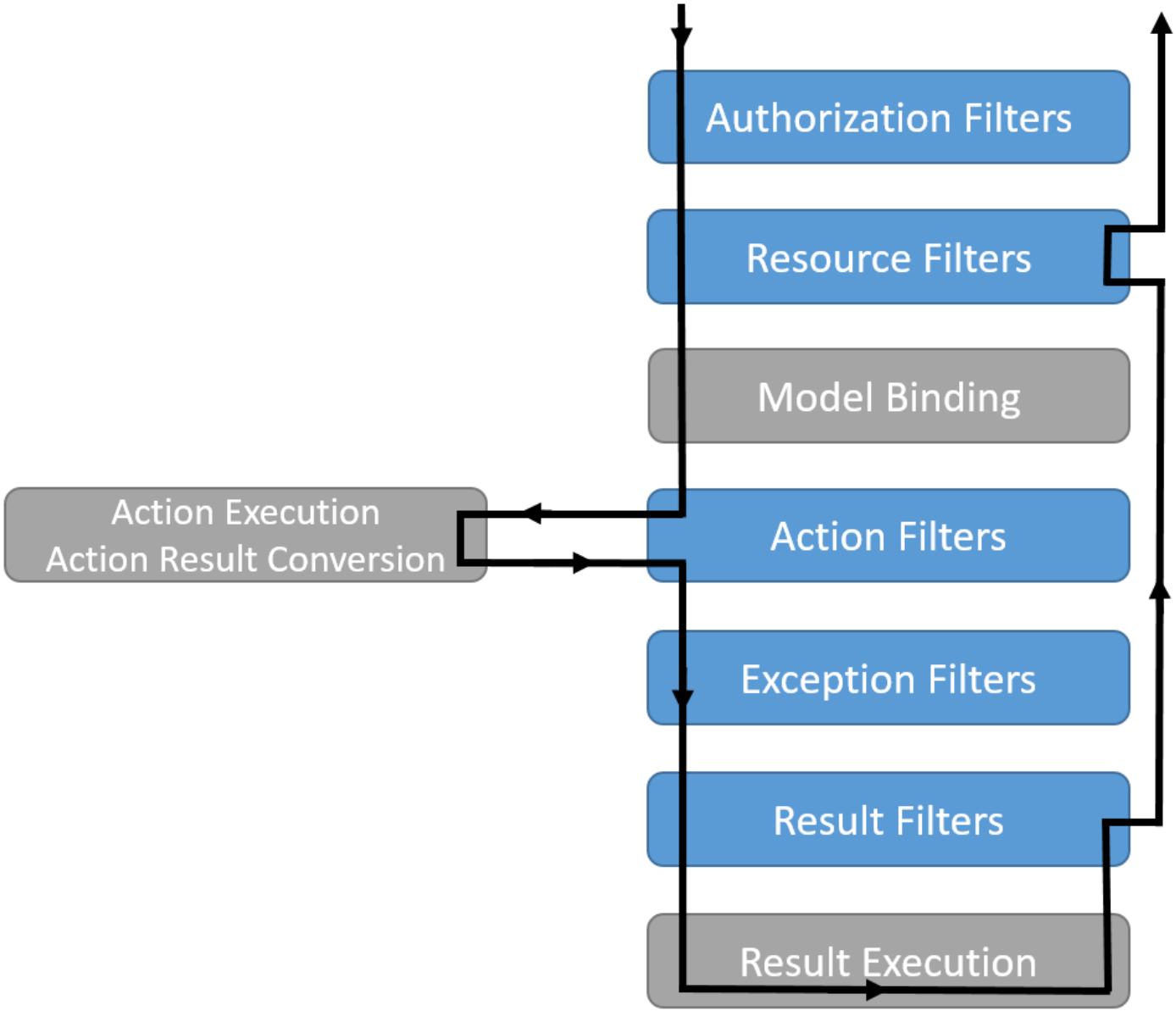
- Resource filters:
  - Run after authorization.
  - OnResourceExecuting runs code before the rest of the filter pipeline. For example, `OnResourceExecuting` runs code before model binding.
  - OnResourceExecuted runs code after the rest of the pipeline has completed.

- Action filters:
  - Run code immediately before and after an action method is called.
  - Can change the arguments passed into an action.
  - Can change the result returned from the action.
  - Are **not** supported in Razor Pages.

- Exception filters apply global policies to unhandled exceptions that occur before the response body has been written to.

- Result filters run code immediately before and after the execution of action results. They run only when the action method has executed successfully. They are useful for logic that must surround view or formatter execution.

The following diagram shows how filter types interact in the filter pipeline.



Implementation

Filters support both synchronous and asynchronous implementations through different interface definitions.

Synchronous filters run code before and after their pipeline stage. For example, OnActionExecuting is called before the action method is called. OnActionExecuted is called after the action method returns.

C# Copy

```
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);
    }
}
```

In the preceding code, MyDebug is a utility function in the sample download .

Asynchronous filters define an `On-Stage-ExecutionAsync` method. For example, OnActionExecutionAsync:

C# Copy

```
public class SampleAsyncActionFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context,
        ActionExecutionDelegate next)
    {
        // Do something before the action executes.

        // next() calls the action method.
```

```
        var resultContext = await next();
        // resultContext.Result is set.
        // Do something after the action executes.
    }
}
```

In the preceding code, the `SampleAsyncActionFilter` has an ActionExecutionDelegate (`next`) that executes the action method.

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.

Built-in filter attributes

ASP.NET Core includes built-in attribute-based filters that can be subclassed and customized. For example, the following result filter adds a header to the response:

C# Copy

```
public class AddHeaderAttribute : ResultFilterAttribute
{
    private readonly string _name;
    private readonly string _value;

    public AddHeaderAttribute(string name, string value)
    {
        _name = name;
        _value = value;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add( _name, new string[] { _value });
        base.OnResultExecuting(context);
    }
}
```

Attributes allow filters to accept arguments, as shown in the preceding example. Apply the `AddHeaderAttribute` to a controller or action method and specify the name and value of the HTTP header:

C# Copy

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
```

Use a tool such as the browser developer tools to examine the headers. Under **Response Headers**, `author: Rick Anderson` is displayed.

The following code implements an `ActionFilterAttribute` that:

- Reads the title and name from the configuration system. Unlike the previous sample, the following code doesn't require filter parameters to be added to the code.
- Adds the title and name to the response header.

C# Copy

```
public class MyActionFilterAttribute : ActionFilterAttribute
{
    private readonly PositionOptions _settings;

    public MyActionFilterAttribute(IOptions<PositionOptions> options)
    {
        _settings = options.Value;
        Order = 1;
    }

    public override void OnResultExecuting(ResultExecutingContext context)
    {
        context.HttpContext.Response.Headers.Add(_settings.Title,
                                                 new string[] { _settings.Name });
        base.OnResultExecuting(context);
    }
}
```

The configuration options are provided from the configuration system using the options pattern. For example, from the `appsettings.json` file:

JSON Copy

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

In the `StartUp.ConfigureServices`:

- The `PositionOptions` class is added to the service container with the `"Position"` configuration area.
- The `MyActionFilterAttribute` is added to the service container.

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<PositionOptions>(
             Configuration.GetSection("Position"));
    services.AddScoped<MyActionFilterAttribute>();

    services.AddControllersWithViews();
}
```

The following code shows the `PositionOptions` class:

C# Copy

```
public class PositionOptions
{
    public string Title { get; set; }
    public string Name { get; set; }
}
```

The following code applies the `MyActionFilterAttribute` to the `Index2` method:

C# Copy

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ServiceFilter(typeof(MyActionFilterAttribute))]
    public IActionResult Index2()
    {
        return Content("Header values by configuration.");
    }
```

Under **Response Headers**, `author: Rick Anderson`, and `Editor: Joe Smith` is displayed when the `Sample/Index2` endpoint is called.

The following code applies the `MyActionFilterAttribute` and the `AddHeaderAttribute` to the Razor Page:

C# Copy

```
[AddHeader("Author", "Rick Anderson")]
[ServiceFilter(typeof(MyActionFilterAttribute))]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
}
```

Filters cannot be applied to Razor Page handler methods. They can be applied either to the Razor Page model or globally.

Several of the filter interfaces have corresponding attributes that can be used as base classes for custom implementations.

Filter attributes:

- ActionFilterAttribute
- ExceptionFilterAttribute
- ResultFilterAttribute
- FormatFilterAttribute
- ServiceFilterAttribute
- TypeFilterAttribute

Filter scopes and order of execution

A filter can be added to the pipeline at one of three *scopes*:

- Using an attribute on a controller action. Filter attributes cannot be applied to Razor Pages handler methods.
- Using an attribute on a controller or Razor Page.
- Globally for all controllers, actions, and Razor Pages as shown in the following code:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter));
    });
}
```

Default order of execution

When there are multiple filters for a particular stage of the pipeline, scope determines the default order of filter execution. Global filters surround class filters, which in turn surround method filters.

As a result of filter nesting, the *after* code of filters runs in the reverse order of the *before* code. The filter sequence:

- The *before* code of global filters.
  - The *before* code of controller and Razor Page filters.
    - The *before* code of action method filters.
    - The *after* code of action method filters.
  - The *after* code of controller and Razor Page filters.
- The *after* code of global filters.

The following example that illustrates the order in which filter methods are called for synchronous action filters.

Sequence	Filter scope	Filter method
1	Global	`OnActionExecuting`
2	Controller or Razor Page	`OnActionExecuting`
3	Method	`OnActionExecuting`
4	Method	`OnActionExecuted`
5	Controller or Razor Page	`OnActionExecuted`
6	Global	`OnActionExecuted`

Controller level filters

Every controller that inherits from the Controller base class includes Controller.OnActionExecuting, Controller.OnActionExecutionAsync, and Controller.OnActionExecuted `OnActionExecuted` methods. These methods:

- Wrap the filters that run for a given action.

- `OnActionExecuting` is called before any of the action's filters.
- `OnActionExecuted` is called after all of the action filters.
- `OnActionExecutionAsync` is called before any of the action's filters. Code in the filter after `next` runs after the action method.

For example, in the download sample, `MySampleActionFilter` is applied globally in startup.

The `TestController`:

- Applies the `SampleActionFilterAttribute` (`[SampleActionFilter]`) to the `FilterTest2` action.
- Overrides `OnActionExecuting` and `OnActionExecuted`.

C# Copy

```
public class TestController : Controller
{
    [SampleActionFilter(Order = int.MinValue)]
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuting(context);
    }

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

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

Navigating to `https://localhost:5001/Test/FilterTest2` runs the following code:

- `TestController.OnActionExecuting`
  - `MySampleActionFilter.OnActionExecuting`
    - `SampleActionFilterAttribute.OnActionExecuting`
      - `TestController.FilterTest2`
    - `SampleActionFilterAttribute.OnActionExecuted`
  - `MySampleActionFilter.OnActionExecuted`
- `TestController.OnActionExecuted`

Controller level filters set the Order property to `int.MinValue`. Controller level filters can **not** be set to run after filters applied to methods. Order is explained in the next section.

For Razor Pages, see Implement Razor Page filters by overriding filter methods.

Overriding the default order

The default sequence of execution can be overridden by implementing IOrderedFilter. `IOrderedFilter` exposes the Order property that takes precedence over scope to determine the order of execution. A filter with a lower `Order` value:

- Runs the *before* code before that of a filter with a higher value of `Order`.
- Runs the *after* code after that of a filter with a higher `Order` value.

The `Order` property is set with a constructor parameter:

C# Copy

```
[SampleActionFilter(Order = int.MinValue)]
```

Consider the two action filters in the following controller:

C# Copy

```
[MyAction2Filter]
public class Test2Controller : Controller
{
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
        MyDebug.Write(MethodBase.GetCurrentMethod(), HttpContext.Request.Path);
        base.OnActionExecuting(context);
    }

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

A global filter is added in `StartUp.ConfigureServices`:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter));
    });
}
```

The 3 filters run in the following order:

- `Test2Controller.OnActionExecuting`
  - `MySampleActionFilter.OnActionExecuting`
    - `MyAction2FilterAttribute.OnActionExecuting`
      - `Test2Controller.FilterTest2`
    - `MyAction2FilterAttribute.OnResultExecuting`
  - `MySampleActionFilter.OnActionExecuted`
- `Test2Controller.OnActionExecuted`

The `Order` property overrides scope when determining the order in which filters run. Filters are sorted first by order, then scope is used to break ties. All of the built-in filters implement `IOrderedFilter` and set the default `Order` value to 0. As mentioned previously, controller level filters set the Order property to `int.MinValue` For built-in filters, scope determines order unless `Order` is set to a non-zero value.

In the preceding code, `MySampleActionFilter` has global scope so it runs before `MyAction2FilterAttribute`, which has controller scope. To make `MyAction2FilterAttribute` run first, set the order to `int.MinValue`:

C# Copy

```
[MyAction2Filter(int.MinValue)]
public class Test2Controller : Controller
{
    public IActionResult FilterTest2()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

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

```
        base.OnActionExecuting(context);
    }

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

To make the global filter `MySampleActionFilter` run first, set `Order` to `int.MinValue`:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
   {
        options.Filters.Add(typeof(MySampleActionFilter),
                            int.MinValue);
    });
}
```

Cancellation and short-circuiting

The filter pipeline can be short-circuited by setting the Result property on the ResourceExecutingContext parameter provided to the filter method. For instance, the following Resource filter prevents the rest of the pipeline from executing:

C# Copy

```
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)
    {
    }
}
```

In the following code, both the `ShortCircuitingResourceFilter` and the `AddHeader` filter target the `SomeResource` action method. The `ShortCircuitingResourceFilter`:

- Runs first, because it's a Resource Filter and `AddHeader` is an Action Filter.
- Short-circuits the rest of the pipeline.

Therefore the `AddHeader` filter never runs for the `SomeResource` action. This behavior would be the same if both filters were applied at the action method level, provided the `ShortCircuitingResourceFilter` ran first. The `ShortCircuitingResourceFilter` runs first because of its filter type, or by explicit use of `Order` property.

C# Copy

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }

    [ServiceFilter(typeof(MyActionFilterAttribute))]
    public IActionResult Index2()
    {
        return Content("Header values by configuration.");
```

```
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header is set.");
    }

    [AddHeaderWithFactory]
    public IActionResult HeaderWithFactory()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
}
```

Dependency injection

Filters can be added by type or by instance. If an instance is added, that instance is used for every request. If a type is added, it's type-activated. A type-activated filter means:

- An instance is created for each request.
- Any constructor dependencies are populated by dependency injection (DI).

Filters that are implemented as attributes and added directly to controller classes or action methods cannot have constructor dependencies provided by dependency injection (DI). Constructor dependencies cannot be provided by DI because:

- Attributes must have their constructor parameters supplied where they're applied.
- This is a limitation of how attributes work.

The following filters support constructor dependencies provided from DI:

- ServiceFilterAttribute
- TypeFilterAttribute
- IFilterFactory implemented on the attribute.

The preceding filters can be applied to a controller or action method:

Loggers are available from DI. However, avoid creating and using filters purely for logging purposes. The built-in framework logging typically provides what's needed for logging. Logging added to filters:

- Should focus on business domain concerns or behavior specific to the filter.
- Should **not** log actions or other framework events. The built-in filters log actions and framework events.

ServiceFilterAttribute

Service filter implementation types are registered in `ConfigureServices`. A ServiceFilterAttribute retrieves an instance of the filter from DI.

The following code shows the `AddHeaderResultServiceFilter`:

C# Copy

```
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");
    }
}
```

In the following code, `AddHeaderResultServiceFilter` is added to the DI container:

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    // Add service filters.
    services.AddScoped<AddHeaderResultServiceFilter>();
    services.AddScoped<SampleActionFilterAttribute>();

    services.AddControllersWithViews(options =>
   {
       options.Filters.Add(new AddHeaderAttribute("GlobalAddHeader",
           "Result filter added to MvcOptions.Filters"));         // An instance
        options.Filters.Add(typeof(MySampleActionFilter));         // By type
        options.Filters.Add(new SampleGlobalActionFilter());       // An instance
    });
}
```

In the following code, the `ServiceFilter` attribute retrieves an instance of the `AddHeaderResultServiceFilter` filter from DI:

C# Copy

```
[ServiceFilter(typeof(AddHeaderResultServiceFilter))]
public IActionResult Index()
{
    return View();
}
```

When using `ServiceFilterAttribute`, setting ServiceFilterAttribute.IsReusable:

- Provides a hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime doesn't guarantee:
  - That a single instance of the filter will be created.
  - The filter will not be re-requested from the DI container at some later point.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

ServiceFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from DI.

TypeFilterAttribute

TypeFilterAttribute is similar to ServiceFilterAttribute, but its type isn't resolved directly from the DI container. It instantiates the type by using Microsoft.Extensions.DependencyInjection.ObjectFactory.

Because `TypeFilterAttribute` types aren't resolved directly from the DI container:

- Types that are referenced using the `TypeFilterAttribute` don't need to be registered with the DI container. They do have their dependencies fulfilled by the DI container.
- `TypeFilterAttribute` can optionally accept constructor arguments for the type.

When using `TypeFilterAttribute`, setting TypeFilterAttribute.IsReusable:

- Provides hint that the filter instance *may* be reused outside of the request scope it was created within. The ASP.NET Core runtime provides no guarantees that a single instance of the filter will be created.
- Should not be used with a filter that depends on services with a lifetime other than singleton.

The following example shows how to pass arguments to a type using `TypeFilterAttribute`:

C# Copy

```
[TypeFilter(typeof(LogConstantFilter),
    Arguments = new object[] { "Method 'Hi' called" })]
public IActionResult Hi(string name)
{
    return Content($"Hi {name}");
}
```

Authorization filters

Authorization filters:

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

Custom authorization filters require a custom authorization framework. Prefer configuring the authorization policies or writing a custom authorization policy over writing a custom filter. 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.

Consider issuing a challenge when an exception occurs in an authorization filter.

Learn more about Authorization.

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 filters are useful to short-circuit most of the pipeline. For example, a caching filter can avoid the rest of the pipeline on a cache hit.

Resource filter examples:

- The short-circuiting resource filter shown previously.
- DisableFormValueModelBindingAttribute :
  - Prevents model binding from accessing the form data.
  - Used for large file uploads to prevent the form data from being read into memory.

Action filters

Action filters do **not** apply to Razor Pages. Razor Pages supports IPageFilter and IAsyncPageFilter . For more information, see Filter methods for Razor Pages.

Action filters:

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

The following code shows a sample action filter:

C# Copy

```
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);
    }
}
```

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.

Throwing an exception in an action method:

- Prevents running of subsequent filters.
- Unlike setting `Result`, is treated as a failure instead of a successful result.

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.

For an `IAsyncActionFilter`, a call to the ActionExecutionDelegate:

- Executes any subsequent action filters and the action method.
- Returns `ActionExecutedContext`.

To short-circuit, assign Microsoft.AspNetCore.Mvc.Filters.ActionExecutingContext.Result to a result instance and don't call `next` (the `ActionExecutionDelegate`).

The framework provides an abstract ActionFilterAttribute that can be subclassed.

The `OnActionExecuting` action filter can be used to:

- Validate model state.
- Return an error if the state is invalid.

C# Copy

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
                                           context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(
                                                context.ModelState);
        }
    }
```

Note

Controllers annotated with the `[ApiController]` attribute automatically validate model state and return a 400 response. For more information, see **Automatic HTTP 400 responses**.

The `OnActionExecuted` method runs after the action method:

- And can see and manipulate the results of the action through the Result property.
- Canceled is set to true if the action execution was short-circuited by another filter.
- Exception is set to a non-null value if the action or a subsequent action filter threw an exception. Setting `Exception` to null:
  - Effectively handles an exception.
  - `ActionExecutedContext.Result` is executed as if it were returned normally from the action method.

C# Copy

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
                                           context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(
                                                context.ModelState);
        }
    }


    public override void OnActionExecuted(ActionExecutedContext
                                          context)
    {
        var result = context.Result;
        // Do something with Result.
        if (context.Canceled == true)
        {
            // Action execution was short-circuited by another filter.
        }

        if(context.Exception != null)
```

```
        {
            // Exception thrown by action or action filter.
            // Set to null to handle the exception.
            context.Exception = null;
        }
        base.OnActionExecuted(context);
    }
}
```

Exception filters

Exception filters:

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

The following sample exception filter uses a custom error view to display details about exceptions that occur when the app is in development:

C# Copy

```
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;
    }
}
```

The following code tests the exception filter:

C# Copy

```
[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.");
    }
}
```

Exception filters:

- Don't have before and after events.
- Implement OnException or OnExceptionAsync.
- Handle unhandled exceptions that occur in Razor Page or controller creation, model binding, action filters, or action methods.
- Do **not** catch exceptions that occur in resource filters, result filters, or MVC result execution.

To handle an exception, set the ExceptionHandled property to `true` or write a response. This stops propagation of the exception. An exception filter can't turn an exception into a "success". Only an action filter can do that.

Exception filters:

- Are good for trapping exceptions that occur within actions.
- Are not as flexible as error handling middleware.

Prefer middleware for exception handling. Use exception filters only where error handling *differs* based on which action method is called. For example, an app might have action methods for both API endpoints and for views/HTML. The API endpoints could return error information as JSON, while the view-based actions could return an error page as HTML.

Result filters

Result filters:

- Implement an interface:
  - IResultFilter or IAsyncResultFilter
  - IAlwaysRunResultFilter or IAsyncAlwaysRunResultFilter
- Their execution surrounds the execution of action results.

IResultFilter and IAsyncResultFilter

The following code shows a result filter that adds an HTTP header:

C# Copy

```
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");
    }
}
```

The kind of result being executed depends on the action. An action returning a view includes all razor processing as part of the ViewResult being executed. An API method might perform some serialization as part of the execution of the result. Learn more about action results.

Result filters are only executed when an action or action filter produces an action result. Result filters are not executed when:

- An authorization filter or resource filter short-circuits the pipeline.
- An exception filter handles an exception by producing an action result.

The Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuting method can short-circuit execution of the action result and subsequent result filters by setting Microsoft.AspNetCore.Mvc.Filters.ResultExecutingContext.Cancel to `true`. Write to the response object when short-circuiting to avoid generating an empty response. Throwing an exception in `IResultFilter.OnResultExecuting`:

- Prevents execution of the action result and subsequent filters.
- Is treated as a failure instead of a successful result.

When the Microsoft.AspNetCore.Mvc.Filters.IResultFilter.OnResultExecuted method runs, the response has probably already been sent to the client. If the response has already been sent to the client, it cannot be changed.

`ResultExecutedContext.Canceled` is set to `true` if the action result execution was short-circuited by another filter.

`ResultExecutedContext.Exception` is set to a non-null value if the action result or a subsequent result filter threw an exception. Setting `Exception` to null effectively handles an exception and prevents the exception from being thrown again later in the pipeline. There is no reliable way to write data to a response when handling an exception in a result filter. If the headers have been flushed to the client when an action result throws an exception, there's no reliable mechanism to send a failure code.

For an IAsyncResultFilter, a call to `await next` on the ResultExecutionDelegate executes any subsequent result filters and the action result. To short-circuit, set ResultExecutingContext.Cancel to `true` and don't call the `ResultExecutionDelegate`:

C# Copy

```
public class MyAsyncResponseFilter : IAsyncResultFilter
{
    public async Task OnResultExecutionAsync(ResultExecutingContext context,
                                             ResultExecutionDelegate next)
    {
        if (!(context.Result is EmptyResult))
        {
            await next();
        }
        else
        {
            context.Cancel = true;
        }

    }
}
```

The framework provides an abstract `ResultFilterAttribute` that can be subclassed. The AddHeaderAttribute class shown previously is an example of a result filter attribute.

IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter

The IAlwaysRunResultFilter and IAsyncAlwaysRunResultFilter interfaces declare an IResultFilter implementation that runs for all action results. This includes action results produced by:

- Authorization filters and resource filters that short-circuit.
- Exception filters.

For example, the following filter always runs and sets an action result (ObjectResult) with a *422 Unprocessable Entity* status code when content negotiation fails:

C# Copy

```
public class UnprocessableResultFilter : Attribute, IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult &&
            statusCodeResult.StatusCode == (int) HttpStatusCode.UnsupportedMediaType)
        {
            context.Result = new ObjectResult("Can't process this!")
            {
                StatusCode = (int) HttpStatusCode.UnsupportedMediaType,
            };
        }
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
    }
}
```

IFilterFactory

IFilterFactory implements IFilterMetadata. Therefore, an `IFilterFactory` instance can be used as an `IFilterMetadata` instance anywhere in the filter pipeline. When the runtime prepares to invoke the filter, it attempts to cast it to an `IFilterFactory`. If that cast succeeds, the CreateInstance method is called to create the `IFilterMetadata` instance that is invoked. This provides a flexible design, since the precise filter pipeline doesn't need to be set explicitly when the app starts.

`IFilterFactory.IsReusable`:

- Is a hint by the factory that the filter instance created by the factory may be reused outside of the request scope it was created within.
- Should ***not*** be used with a filter that depends on services with a lifetime other than singleton.

The ASP.NET Core runtime doesn't guarantee:

- That a single instance of the filter will be created.
- The filter will not be re-requested from the DI container at some later point.

> **⚠ Warning**
>
> Only configure **IFilterFactory.IsReusable** to return `true` if the source of the filters is unambiguous, the filters are stateless, and the filters are safe to use across multiple HTTP requests. For instance, don't return filters from DI that are registered as scoped or transient if `IFilterFactory.IsReusable` returns `true`.

`IFilterFactory` can be implemented using custom attribute implementations as another approach to creating filters:

C# Copy

```
public class AddHeaderWithFactoryAttribute : Attribute, IFilterFactory
{
    // Implement IFilterFactory
```

```
    public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
    {
        return new InternalAddHeaderFilter();
    }

    private class InternalAddHeaderFilter : IResultFilter
    {
        public void OnResultExecuting(ResultExecutingContext context)
        {
            context.HttpContext.Response.Headers.Add(
                "Internal", new string[] { "My header" });
        }

        public void OnResultExecuted(ResultExecutedContext context)
        {
        }
    }

    public bool IsReusable
    {
        get
        {
            return false;
        }
    }
}
```

The filter is applied in the following code:

C# Copy

```
[AddHeader("Author", "Rick Anderson")]
public class SampleController : Controller
{
    public IActionResult Index()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
```

```
    [ServiceFilter(typeof(MyActionFilterAttribute))]
    public IActionResult Index2()
    {
        return Content("Header values by configuration.");
    }

    [ShortCircuitingResourceFilter]
    public IActionResult SomeResource()
    {
        return Content("Successful access to resource - header is set.");
    }

    [AddHeaderWithFactory]
    public IActionResult HeaderWithFactory()
    {
        return Content("Examine the headers using the F12 developer tools.");
    }
}
```

Test the preceding code by running the download sample :

- Invoke the F12 developer tools.
- Navigate to `https://localhost:5001/Sample/HeaderWithFactory`.

The F12 developer tools display the following response headers added by the sample code:

- **author:** `Rick Anderson`
- **globaladdheader:** `Result filter added to MvcOptions.Filters`
- **internal:** `My header`

The preceding code creates the **internal:** `My header` response header.

IFilterFactory implemented on an attribute

Filters that implement `IFilterFactory` are useful for filters that:

- Don't require passing parameters.
- Have constructor dependencies that need to be filled by DI.

TypeFilterAttribute implements IFilterFactory. `IFilterFactory` exposes the CreateInstance method for creating an IFilterMetadata instance. `CreateInstance` loads the specified type from the services container (DI).

C# Copy

```
public class SampleActionFilterAttribute : TypeFilterAttribute
{
    public SampleActionFilterAttribute()
                        :base(typeof(SampleActionFilterImpl))
    {
    }

    private class SampleActionFilterImpl : IActionFilter
    {
        private readonly ILogger _logger;
        public SampleActionFilterImpl(ILoggerFactory loggerFactory)
        {
            _logger = loggerFactory.CreateLogger<SampleActionFilterAttribute>();
        }

        public void OnActionExecuting(ActionExecutingContext context)
        {
           _logger.LogInformation("SampleActionFilterAttribute.OnActionExecuting");
        }

        public void OnActionExecuted(ActionExecutedContext context)
        {
            _logger.LogInformation("SampleActionFilterAttribute.OnActionExecuted");
        }
    }
}
```

The following code shows three approaches to applying the `[SampleActionFilter]`:

C# Copy

```
[SampleActionFilter]
public IActionResult FilterTest()
{
    return Content("From FilterTest");
}

[TypeFilter(typeof(SampleActionFilterAttribute))]
public IActionResult TypeFilterTest()
{
    return Content("From TypeFilterTest");
}

// ServiceFilter must be registered in ConfigureServices or
// System.InvalidOperationException: No service for type '<filter>'
// has been registered. Is thrown.
[ServiceFilter(typeof(SampleActionFilterAttribute))]
public IActionResult ServiceFilterTest()
{
    return Content("From ServiceFilterTest");
}
```

In the preceding code, decorating the method with `[SampleActionFilter]` is the preferred approach to applying the `SampleActionFilter`.

Using middleware in the filter pipeline

Resource filters work like middleware in that they surround the execution of everything that comes later in the pipeline. But filters differ from middleware in that they're part of the runtime, which means that they have access to context and constructs.

To use middleware as a filter, create a type with a `Configure` method that specifies the middleware to inject into the filter pipeline. The following example uses the localization middleware to establish the current culture for a request:

C# Copy

```
public class LocalizationPipeline
{
    public void Configure(IApplicationBuilder applicationBuilder)
    {
        var supportedCultures = new[]
        {
            new CultureInfo("en-US"),
            new CultureInfo("fr")
        };

        var options = new RequestLocalizationOptions
        {
            DefaultRequestCulture = new RequestCulture(
                                        culture: "en-US",
                                        uiCulture: "en-US"),
            SupportedCultures = supportedCultures,
            SupportedUICultures = supportedCultures
        };
        options.RequestCultureProviders = new[]
            { new RouteDataRequestCultureProvider() {
                Options = options } };

        applicationBuilder.UseRequestLocalization(options);
    }
}
```

Use the MiddlewareFilterAttribute to run the middleware:

C# Copy

```
[Route("{culture}/[controller]/[action]")]
[MiddlewareFilter(typeof(LocalizationPipeline))]
```

```
public IActionResult CultureFromRouteData()
{
    return Content(
          $"CurrentCulture:{CultureInfo.CurrentCulture.Name},"
        + $"CurrentUICulture:{CultureInfo.CurrentUICulture.Name}");
}
```

Middleware filters run at the same stage of the filter pipeline as Resource filters, before model binding and after the rest of the pipeline.

Thread safety

When passing an *instance* of a filter into `Add`, instead of its `Type`, the filter is a singleton and is **not** thread-safe.

Next actions

- See Filter methods for Razor Pages.
- To experiment with filters, download, test, and modify the GitHub sample .

Is this page helpful?

Yes No

Recommended content

Handle errors in ASP.NET Core

Discover how to handle errors in ASP.NET Core apps.

ASP.NET Core Middleware

Learn about ASP.NET Core middleware and the request pipeline.

Handle errors in ASP.NET Core web APIs

Learn about error handling with ASP.NET Core web APIs.

Routing in ASP.NET Core

Discover how ASP.NET Core routing is responsible for matching HTTP requests and dispatching to executable endpoints.

Show more