Dhruvil Dobariya

dhruvildobariya21@gmail.com

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

.Net

Documentation



Index

[1 Filters 1](#_Toc128491980)

[1.1 Introduction 1](#_Toc128491981)

[1.2 Scope and Order 4](#_Toc128491982)

[1.3 Cancellation or short-circuiting filters 6](#_Toc128491983)

[1.4 Dependency Injection in Filters 7](#_Toc128491984)

1. Filters

# Introduction

* Filters allow us to run custom code before or after executing the action method.
* They provide ways to do common repetitive tasks on our action method.
* The filters are invoked on certain stages in the request processing pipeline.
* There are many built-in filters available with ASP.NET Core MVC, and we can create custom filters as well.
* Filters help us to remove duplicate codes in our application.
  + Authorization filter
  + Resource filter
  + Action filter
  + Exception filter
  + Result filter

**Authorization filter:**

* The Authorization filters are executed first.
* This filter helps us to determine whether the user is authorized for the current request or not.
* It can short-circuit a pipeline if a user is unauthorized for the current request.
* We can also create custom authorization filter.

**Resource filters:**

* The Resource filters handle the request after authorization.
* It can run the code before and after the rest of the filter is executed.
* This executes before the model binding happens.
* It can be used to implement caching.

**Action filters:**

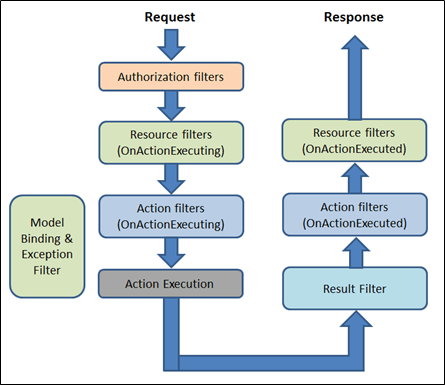
* The Action filters run the code immediately before and after the controller action method is called.
* It can be used to perform any action before or after execution of the controller action method.
* We can also manipulate the arguments passed into an action.

**Exception filters:**

* The Exception filters are used to handle exception that occurred before anything written to the response body.

**Result filters:**

* The Result filters are used to run code before or after the execution of controller action results.
* They are executed only if the controller action method has been executed successfully.



* Filter supports two types of implementation.
  + Synchronous
  + Asynchronous

**Synchronous:**

* The Synchronous filters run the code before and after their pipeline stage defines OnStageExecuting and OnStageExecuted.
* For example: ActionFilter, The OnActionExecuting method is called before the action method and OnActionExecuted method is called after the action method.

**Example:**

*using* Microsoft.AspNetCore.Mvc.Filters;

*namespace* Filters

{

*public* *class* CustomActionFilter : IActionFilter

    {

*public* *void* OnActionExecuting(*ActionExecutingContext* context)

        {

*//To do : before the action executes*

        }

*public* *void* OnActionExecuted(*ActionExecutedContext* context)

        {

*//To do : after the action executes*

        }

    }

}

**Asynchronous:**

* Asynchronous filters are defined with only single method, OnStageExecutionAsync that takes a FilterTypeExecutingContext and FilterTypeExecutionDelegate as The FilterTypeExecutionDelegate execute the filter’s pipeline stage.
* For example: ActionFilter, ActionExecutionDelegate calls the action method and we can write the code before and after we call action method.

**Example:**

*using* System.Threading.Tasks;

*using* Microsoft.AspNetCore.Mvc.Filters;

*namespace* Filters

{

*public* *class* CustomAsyncActionFilter : IAsyncActionFilter

    {

*public* *async* *Task* OnActionExecutionAsync(*ActionExecutingContext* context, *ActionExecutionDelegate* next)

        {

*//To do : before the action executes*

*await* next();

*//To do : after the action executes*

        }

    }

}

* We can implement interfaces for multiple filter types (stage) in single class.
* We can either implement synchronous or the async version of a filter interface, not both.
* The .net framework checks first for async filter interface, if it finds it, it called, If it is not found it calls the synchronous interface's method(s).
* If we implement both, synchronous interface is never called.

# Scope and Order

* A filter can be added to the pipeline at one of three scopes.
  + By action method
  + By controller class
  + Globally (which be applied to all the controller and actions).
* We need to register filters in to the MvcOption.Filters collection within ConfigureServices method.

**Example:**

*public* *void* ConfigureServices(*IServiceCollection* services)

{

*// Add framework services.*

    services.AddMvc(options=> {

*//an instant*

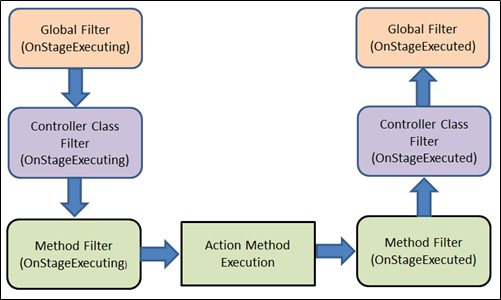
    options.Filters.Add(*new* *CustomActionFilter*());

*//By the type*

    options.Filters.Add(*typeof*(*CustomActionFilter*));

    });

}

* When multiple filters are applied to the particular stage of the pipeline, scope of filter defines the default order of the filter execution.
* The global filter is applied first, then class level filter is applied and finally method level filter is applied.

**Overriding the default order:**

* We can override the default sequence of filter execution by using implementing interface IOrderedFilter.
* This interface has property named "Order" that use to determine the order of execution.
* The filter with lower order value execute before the filter with higher order value.
* We can setup the order property using the constructor parameter.

**Example:**

*// ExampleFilter.cs*

*using* System;

*using* Microsoft.AspNetCore.Mvc.Filters;

*namespace* Filters

{

*public* *class* ExampleFilterAttribute : Attribute, IActionFilter, IOrderedFilter

    {

*public* *int* Order { *get*; *set*; }

*public* *void* OnActionExecuting(*ActionExecutingContext* context)

        {

*//To do : before the action executes*

        }

*public* *void* OnActionExecuted(*ActionExecutedContext* context)

        {

*//To do : after the action executes*

        }

    }

}

*// HomeController.cs*

*using* System;

*using* Microsoft.AspNetCore.Mvc;

*using* Filters;

*namespace* Filters.Controllers

{

    [ExampleFilter(Order = 1)]

*public* *class* HomeController : Controller

    {

*public* *IActionResult* Index()

        {

*return* View();

        }

    }

}

* When filters are run in pipeline, filters are sorted first by order and then scope.
* All built-in filters are implemented by IOrderFilter and set the default filter order to 0.

# Cancellation or short-circuiting filters

* We can short circuit the filter pipeline at any point of time by setting the "Result" property of the "Context" parameter provided to the filter's methods.

*using* System;

*using* Microsoft.AspNetCore.Mvc;

*using* Microsoft.AspNetCore.Mvc.Filters;

*namespace* Filters

{

*public* *class* Example1FilterAttribute : Attribute, IActionFilter

    {

*public* *void* OnActionExecuting(*ActionExecutingContext* context)

        {

*//To do : before the action executes*

            context.Result = *new* *ContentResult*()

            {

                Content = "Short circuit filter"

            };

        }

*public* *void* OnActionExecuted(*ActionExecutedContext* context)

        {

*//To do : after the action executes*

        }

    }

}

# Dependency Injection in Filters

* As we learned, the filter can be added by the type or by the instance.
* If we added filter as an instance, this instance will be used for every request and if we add filter as a type, instance of the type will be created for each request.
* Filter has constructor dependencies that will be provided by the DI.
* The filters that are implemented as attributes and added directly to the controller or action methods, cannot have constructor dependencies provided by the DI.
* In this case, contractor parameter must be supplied when they are applied.
* This is a limitation of attribute.
* There are many way to overcome this limitation.
* We can apply our filter to the controller class or action method using one of the following,
  + ServiceFilterAttribute
  + TypeFilterAttribute
  + IFilterFactory implemented on attribute

**ServiceFilterAttribute:**

* A ServiceFilter retrieves an instance of the filter from dependency injection (DI).
* We need to add this filter to the container in ConfigureService and reference it in a ServiceFilter attribute in the controller class or action method.
* One of the dependencies we might require to get from the DI, is a logger. Within filter, we might need to log something happened.

**Example:**

*// Filter*

*using* Microsoft.AspNetCore.Mvc.Filters;

*using* Microsoft.Extensions.Logging;

*namespace* FiltersSample.Filters

{

*public* *class* ExampleFilterWithDI : IActionFilter

    {

*private* *ILogger* \_logger;

*public* ExampleFilterWithDI(*ILoggerFactory* loggerFactory)

        {

            \_logger = loggerFactory.CreateLogger<*ExampleFilterWithDI*>();

        }

*public* *void* OnActionExecuting(*ActionExecutingContext* context)

        {

*//To do : before the action executes*

            \_logger.LogInformation("OnActionExecuting");

        }

*public* *void* OnActionExecuted(*ActionExecutedContext* context)

        {

*//To do : after the action executes*

            \_logger.LogInformation("OnActionExecuted");

        }

    }

}

*// Register filter in ConfigureService method*

*public* *void* ConfigureServices(*IServiceCollection* services)

{

    services.AddScoped<*ExampleFilterWithDI*>();

}

*// Use filter for Action method of Controller class*

[ServiceFilter(*typeof*(*ExampleFilterWithDI*))]

*public* *IActionResult* Index()

{

*return* View();

}

**TypeFilterAttribute:**

* It is very similar to ServiceFilterAttribute and also implemented from IFilterFactory interface.
* Here, type is not resolved directly from the DI container but it instantiates the type using class "Microsoft.Extensions.DependencyInjection.ObjectFactory".
* Due to this difference, the types are referenced in TypeFilterAttribute need to be register first in ConfigureService method.
* The "TypeFilterAttribute" can be optionally accept constructor arguments for the type.

**Example:**

[TypeFilter(*typeof*(*ExampleFilterAttribute*), Arguments = *new* *object*[]

{"Argument if any" })]

*public* *IActionResult* About()

{

*return* View();

}