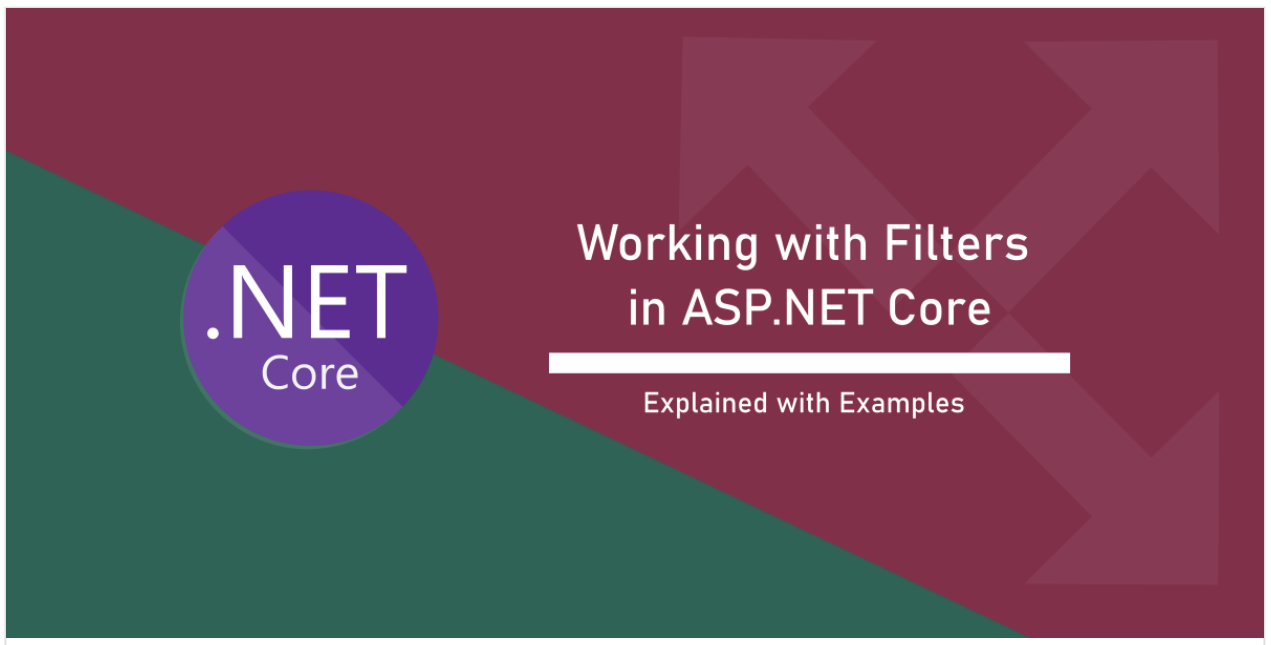

Working with Filters in ASP.NET Core - Explained with Examples

ASP.NET Core Posted Nov 14, 2020

Filters are components built into the ASP.NET Core which can help us in controlling the execution of a request at specific stages of the request pipeline. ***These come into picture post the middleware execution, when the MVC middleware matches a Route and while a specific Action is being invoked.***

SIMILAR ARTICLES

Implementing Cookie Authentication in ASP.NET Core without Identity

Garbage Collection in .NET Core Explained - Concepts and Best Practices

Implementing Pagination in ASP.NET Core Web API and Entity Framework Core

Building Custom Responses for UnAuthorized Requests in ASP.NET Core

Implementing stream based communication with gRPC and ASP.NET Core

Since these executed within an *action invocation pipeline*, these filters can get information about which Action has been selected and the RouteData. Thereby using Filters, we can control the execution at the action level with much more information at hand. We can do things specific and custom to each Action/Controller such as Exception Handling, Caching or passing custom Response Headers.

Types of Filters:

Types of Filters:

ASP.NET Core comes with the below set of predefined Filter types which we can create based on how we want to control our request execution within the context of an Action/Controller.

1. **AuthorizationFilter** - Occurs at the beginning of the Route execution, determine whether the user is allowed to access the Route

2. **ResourceFilter** - Runs after the Authorization, can be used to bypass execution of the rest of the pipeline. For use cases such as sending out cached response if there is a preprocessed response available for the request we can use this Filter. Other Filters down the pipeline never run, if the request is short-circuited at the ResourceFilter.

3. **ActionFilter** - Runs before and after the action is executed, can be used to surround an action.

4. **ResultFilter** - Runs before and after a Result has been generated from an Action, can be used to surround a result execution.

5. **ExceptionFilter** - Runs whenever an uncaught Exception is thrown within the Action or the Controller. Can be used to design customized error responses specific to Actions or Controllers.

6. **ServiceFilter** - Runs another Filter whose type is passed within itself. Used in cases when the passed ActionFilter type is registered as a service. Resolves any dependencies which are passed to the Filter type via DI.

7. **TypeFilter** - Similar to the ServiceFilter, except that the passed Filter type is not a registered service.

The first five action filters apply at different stages of the execution of the MVC route, while the ServiceFilter and TypeFilter facilitate execution of these filters by providing additional capabilities such as Dependency Injection and so on.

Creating a Simple Filter:

There are two ways in which you can create any Filter of the types we discussed before. For example, to create an ActionFilter which can act upon an action execution we can:

1. create a custom action filter class which implements the IActionFilter or IAsyncActionFilter interfaces (or)
2. create a custom action filter class which extends the ActionFilterAttribute

Similarly, for the five action filters mentioned above, we have the following interfaces:

1. IAuthorizationFilter
2. IResourceFilter
3. IActionFilter
4. IResultFilter
5. IExceptionFilter

and few of these Filters are complemented by their attribute classes which implement the above interfaces and provide us the option to override which ever method is suitable for us.

1. ActionFilterAttribute
2. ExceptionFilterAttribute
3. ResultFilterAttribute
4. FormatFilterAttribute
5. ServiceFilterAttribute
6. TypeFilterAttribute

> *"If we want to decorate our Actions or Controllers with Filters we need to use the Attribute classes to create Filters, if we want to register as Global filters then we can simply go with the interface implementations."*

For example, the ActionFilterAttribute class already implements the IActionFilter and IAsyncActionFilter interfaces for us, so we can extend the ActionFilterAttribute class directly and override the methods we're interested in.

```
public class AddResponseHeaderFilter : ActionFilterAttribute
{
    // async method which can surround the action execution
    // invoked for both before and post execution of action
    public async override Task OnActionExecutionAsync(
        ActionExecutingContext context, ActionExecutionDelegate next)
    {
        // access Request
        context.HttpContext.Response.Headers.Add(
        "X-MyCustomHeader", Guid.NewGuid().ToString());

        var result = await next.Invoke();

        // access Response
        Console.WriteLine(JsonConvert.SerializeObject(result.Result));
    }
}

[Route("api/[controller]")]
[ApiController]
public class HomeController : ControllerBase
{
    [AddResponseHeaderFilter]
    [Route("")]
    [HttpGet]
    public IActionResult Index()
    {
        return Ok(new { Message = "I'm Alive" });
    }
}
```

One advantage of using ActionFilterAttribute over implementing IActionFilter interface is that we can choose the method to override based on our requirement and expectation and we can use the filter to decorate over our action or controller directly.

> *"A filter which implements IActionFilter or IAsyncActionFilter can't be used as an attribute directly. Instead we need to use a TypeFilter or a ServiceFilter which can invoke our filter during runtime."*

If we create our AddResponseHeaderFilter by implementing the IAsyncActionFilter interface instead of the ActionFilterAttribute, we need to go by the TypeFilter as below:

```
public class AddResponseHeaderFilter : IAsyncActionFilter
{
    public async Task OnActionExecutionAsync(
        ActionExecutingContext context, ActionExecutionDelegate next)
    {
        // access Request
        context.HttpContext.Response.Headers.Add(
        "X-MyCustomHeader", Guid.NewGuid().ToString());

        var result = await next.Invoke();

        // access Response
        Console.WriteLine(JsonConvert.SerializeObject(result.Result));
    }
}

[Route("api/[controller]")]
[ApiController]
public class HomeController : ControllerBase
{
    [TypeFilter(typeof(AddResponseHeaderFilter))]
    [Route("")]
    [HttpGet]
    public IActionResult Index()
    {
        return Ok(new { Message = "I'm Alive" });
    }
}
```

Filters Scope:

Scope of the Filters is defined by how they're attached to the MVC pipeline. For example, we can have a Filter be executed only when a specific Action is being executed, or when any action of a specific Controller is being executed or whenever a Route is matched and picked for execution. Hence we have three scopes here:

1. **Action specific** - by applying the Filter attribute to a particular Action method inside the controller. Decorate the Action with the Filter Attribute and the Filter executes only when the Action is picked for the incoming request to execute.
2. **Controller specific** - by applying the Filter attribute to a particular Controller class and the filter executes for all the actions under that Controller.

3. **Global Filters** - applied to every Route that is matched and picked up by the Routing middleware. If there is no Endpoint selected for the request, no Filter executes.

A Filter is registered as a Global Filter by adding to the Filters array inside the ConfigureServices() method within the Startup class, as below:

```
services.AddControllers(options => {
    options.Filters.Add(typeof(ConsoledGlobalActionFilter));
});
```

Dependency Injection in Filters:

There can be cases, where we might need to inject dependencies for use in our action filters. Say for example, if we want to log events that happen in our ActionFilter, we can't directly do a constructor injection like we do for any other service. Instead, we use either of the TypeFilter or ServiceFilter attributes, which resolve the dependencies required for the ActionFilter to execute if available in the DI and invoke the filter instance.

```
public class AddResponseHeaderFilter : IAsyncActionFilter
{
    private readonly ILogger<AddResponseHeaderFilter> _logger;

    public AddResponseHeaderFilter(ILogger<AddResponseHeaderFilter>
logger)
    {
        _logger = logger;
    }

    public async Task OnActionExecutionAsync(
      ActionExecutingContext context, ActionExecutionDelegate next)
    {
        _logger.LogInformation("Inside the AddResponseHeaderFilter");
        // access Request
        context.HttpContext.Response.Headers.Add(
          "X-MyCustomHeader", Guid.NewGuid().ToString());
        var result = await next.Invoke();
        // access Response
        Console.WriteLine(JsonConvert.SerializeObject(result.Result));
    }
}

[Route("api/[controller]")]
[ApiController]
public class HomeController : ControllerBase
{
    [TypeFilter(typeof(AddResponseHeaderFilter))]
    [Route("")]
    [HttpGet]
    public IActionResult Index()
    {
        return Ok(new { Message = "I'm Alive" });
    }
}
```

> *"When to use TypeFilter and ServiceFilter? When your ActionFilter implementation is itself registered as a Service, then we need to use the ServiceFilter attribute to invoke the Filter. Otherwise TypeFilter attribute would do the job for us."*

Order of Filters Execution:

Although all the filters are executed only after a route is selected by the Routing middleware and within the MVC scope, there is a specific order in which all the above action filters are executed. To figure out how they are executed, let's create simple implementations for all these filter types and decorate over a single action. On execution of the action, we can understand how the actions are called.

```
public class ConsoledResourceFilter : IResourceFilter
{
    public void OnResourceExecuted(ResourceExecutedContext context)
    {
        Console.WriteLine("ConsoledResourceFilter -
OnResourceExecuted");
    }

    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        Console.WriteLine("ConsoledResourceFilter -
OnResourceExecuting");
    }
}

public class ConsoledActionFilter : IActionFilter
{
    public void OnActionExecuted(ActionExecutedContext context)
    {
        Console.WriteLine("ConsoledActionFilter - OnActionExecuted");
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("ConsoledActionFilter - OnActionExecuting");
    }
}

public class ConsoledAuthorizationFilter : IAuthorizationFilter
{
    public void OnAuthorization(AuthorizationFilterContext context)
    {
        Console.WriteLine("ConsoledAuthorizationFilter -
OnAuthorization");
    }
}

public class ConsoledResultFilter : IResultFilter
{
    public void OnResultExecuted(ResultExecutedContext context)
    {
        Console.WriteLine("ConsoledResultFilter - OnResultExecuted");
    }

    public void OnResultExecuting(ResultExecutingContext context)
    {
        Console.WriteLine("ConsoledResultFilter - OnResultExecuting");
    }
}

// Global ActionFilter
// Registered within the Startup Class
// Executes for all Actions within the MVC
public class ConsoledGlobalActionFilter : IActionFilter
{
    public void OnActionExecuted(ActionExecutedContext context)
```

```
    {
        Console.WriteLine("ConsoledGlobalActionFilter -
OnActionExecuted");
    }

    public void OnActionExecuting(ActionExecutingContext context)
    {
        Console.WriteLine("ConsoledGlobalActionFilter -
OnActionExecuting");
    }
}

public class HomeController : ControllerBase
{
    [TypeFilter(typeof(ConsoledAuthorizationFilter))]
    [TypeFilter(typeof(ConsoledResourceFilter))]
    [TypeFilter(typeof(ConsoledActionFilter))]
    [TypeFilter(typeof(ConsoledResultFilter))]
    [HttpGet]
    [Route("consoled")]
    public IActionResult Consoled()
    {
        return Ok();
    }
}
```

The output we get for the filters execution is:

```
ConsoledAuthorizationFilter - OnAuthorization

ConsoledResourceFilter - OnResourceExecuting

ConsoledGlobalActionFilter - OnActionExecuting

ConsoledActionFilter - OnActionExecuting
ConsoledActionFilter - OnActionExecuted

ConsoledGlobalActionFilter - OnActionExecuted

ConsoledResultFilter - OnResultExecuting
ConsoledResultFilter - OnResultExecuted

ConsoledResourceFilter - OnResourceExecuted
```

From the output, we can understand that:

1. Authorization filter is executed first and only once within a request lifetime.
2. ResourceFilter follows the AuthorizationFilter, and executes first and last of the request.
3. ActionFilters are invoked next, within the ActionFilters the Global filters are invoked first followed by the Route level ActionFilters.
4. Once the ActionFilters are done with ActionExecution, the ResultFilters are executed before the Result is passed down the pipeline as Response.

Final Thoughts:

Filters are one of the fundamental features of the MVC framework which is even more polished and sophisticated in ASP.NET Core.

Filters help us in further customizing or tweaking how the Actions or Controllers work once a route is selected.

Since these are invoked within MVC piepline, they have complete information about which Action is being invoked along with the Route as well as RouteData, giving us more control over the action execution than the case with a generic Middleware which only has access to the HttpContext.

We have also seen ActionFilters in action when we looked at Generating ETag for Response Caching. With the Dependency Injection built into the .NET Core framework, we can also levarage the use of services and dependencies within our Filters by means of ServiceFilter or TypeFilter attributes.

asp.net core mvc authorization filter *asp.net core mvc custom filter*
asp.net core mvc custom authorization filter *asp.net core mvc action filters* *asp .net core filters*
filters in asp.net core mvc *asp.net core filter attribute* *asp.net mvc filters*
filters in asp.net mvc core *filters in .net core mvc* *asp.net core mvc global filter*
asp.net mvc 5 global filters *microsoft.aspnetcore.mvc.filters .net core 3.1*
asp.net mvc filters and attributes *asp.net core mvc filter order*
asp.net mvc 5 authentication filters using example *asp .net mvc filters*

Ram

I'm a full-stack developer and a software enthusiast who likes to play around with cloud and tech stack out of curiosity.

You can now show your support. 😊

Buy me a coffee