



Filter ASP.NET Core

Los filtros o *filters* nos permiten ejecutar código antes o después de determinadas fases en el procesamiento de una solicitud HTTP en un *pipeline*. Es decir puedes interferir esta solicitud antes o después de que llego a nuestro *Controller* o *metodo*.

Podemos encontrar diferentes tipos de filtros para realizar diferentes operaciones, como tambien podemos encontrar filtros ya hechos para determinadas acciones y podemos utilizar. Algunos de esos filtros ya hechos son:

- Authorization (previene el acceso de un usuario no autorizado a un recurso)
- Respuestas hardcodedas (podemos acortar el circuito que a traviesa la request en la pipeline con respuestas cacheadas/hardcodeadas)

Para que usarlos?

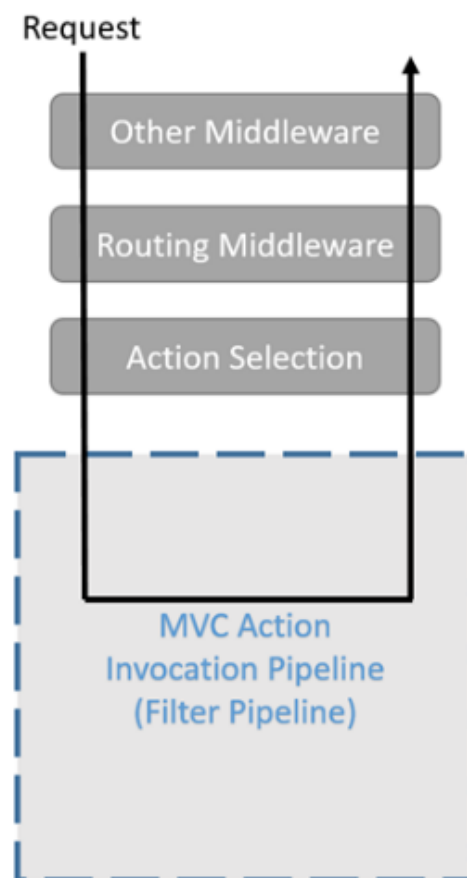
Pero no solamente tenemos tipos ni filtros ya hechos si no que tambien podemos crear los nuestros. Seguramente se esten preguntando, porque crear nuestros filtros si ya existen algunos creados y que podemos usar? Una justificacion de porque queremos crear nuestros propios filtros es para codificar conceptos. Con conceptos nos referimos a manejoamiento de errores, cache, configuracion, autorizacion y logging.

Gracias a esto podemos evitar la duplicacion de codigo en aquellas instancias donde se deben aplicar los mismos procedimientos para muchos metodos del Controller o para diferentes Controller. Por ejemplo la autentificacion, realizar un filtro que se encargue de sacar la informacion correspondiente de la request para validar la autentificacion es una forma de evitar duplicar ese codigo en todos los metodos que necesiten autentificacion. Lo mismo sucede si queremos realizar un manejoamiento de

error (error handling) bastante generico de forma de limpiar nuestro codigo y tener un unico punto de salida para mensajes de error. Estos escenarios seran vistos mas adelante

Como funcionan?

Los filtros corren en ASP.NET Core action invocation pipeline, tambien referido como filter pipeline (la tuberia). Esta tuberia se corre despues de que ASP.NET Core selecciona la accion que se tiene que ejecutar.





Veamos estos pasos por los que pasa la request.

1. La request pasa por un middleware
2. Pasa por el middleware de redireccionamiento, para saber que controller instanciar
3. Pasa por la seleccion de la accion para saber que metodo ejecutar
4. Luego pasa por la pipeline de los filter para realizar ciertas acciones antes o despues de ejecutar la accion

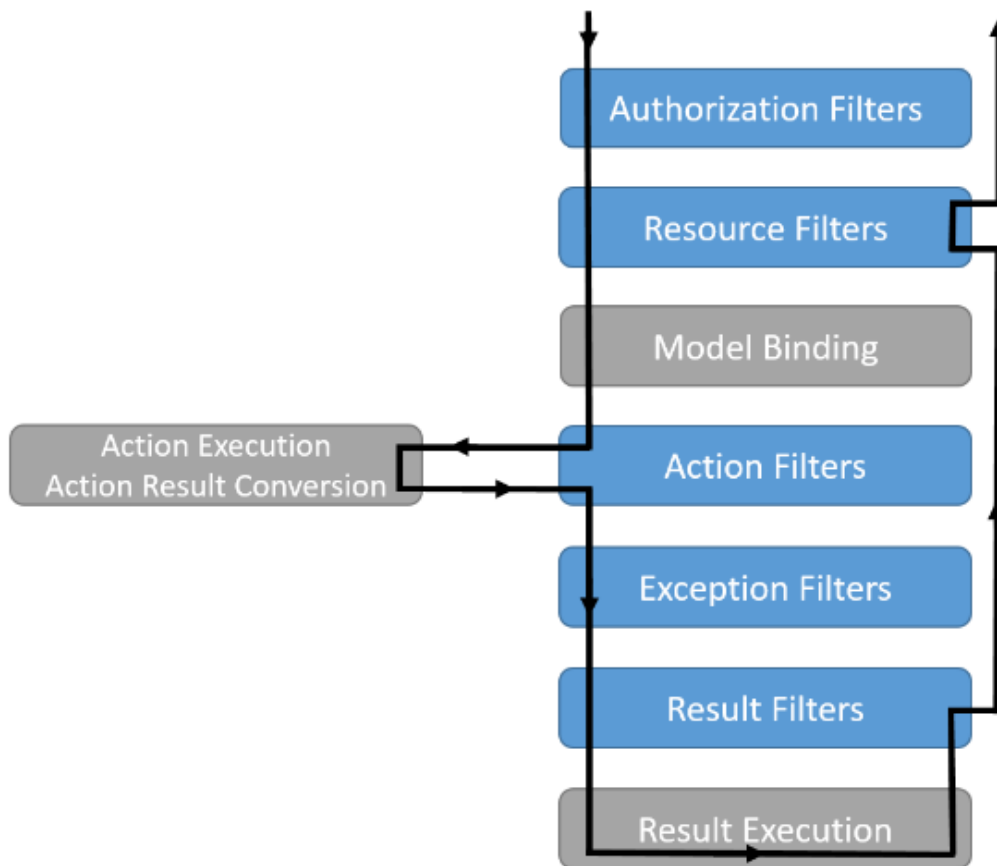
Tipos de filtros

Cada tipo de filtro es ejecutado en una fase diferente de la solicitud, es decir tienen un orden de ejecución según su responsabilidad.

Tipos

 Tipo	 Interface	 Descripción
<u>Autorización</u>	IAuthorizationFilter	Es el primer filtro que se corre y se utiliza para determinar si el usuario esta autorizado para realizar dicha accion que solicito con la request. Estos filtros acortan el recorrido de la request en la pipeline si no estas autorizado
<u>Recurso</u>	IResourceFilter	Corre despues del filtro de autorizacion. Tiene un metodo que corre antes de seguir (OnResourceExecuting) y otro para despues de que la pipeline se completo (OnResourceExecuting)
<u>Acción</u>	IActionFilter	Corre codigo inmediatamente antes y despues de la llamada del metodo. Puede cambiar los argumentos que se le pasan al metodo Puede cambiar modificar la response que armo el metodo
<u>Excepción</u>	IExceptionFilter	Aplican politicas globales para excepciones no manejadas que ocurren antes de que el body de la response se haya creado
<u>Resultado</u>	IResultFilter	Corren codigo inmediatamente antes y despues del retorno de un metodo. Se corren unicamente cuando el metodo se ejecuto exitosamente. Son utiles para aplicar cierta logica a un formato

Veamos un diagrama de como los filtros interactuan entre ellos en la pipeline.



Filtros ya creados (built-in filters)

ASP.NET Core nos provee atributos que implementan algun tipo de filtro que los podemos tomar como clase padre y hacer override de ciertos metodos.

Por ejemplo, el siguiente filtro nuestro le agrega un header a la response:

```
public class AddHeaderFilter : ResultFilterAttribute
{
    private readonly string name;
    private readonly string value;

    public AddHeaderFilter(string name, string value)
    {
        this.name = name;
        this.value = value;
    }

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

Basicamente lo que hace este filtro es hacerle un override a un metodo de la clase padre para agregarle el header a la response y luego llamar al metodo padre para

que retorne la response.

Veamos como se usaria este filtro en un controller.

```
[ApiController]
[Route("api/movies")]
[AddHeader("Author", "Rick Anderson")]
public class MovieController : ControllerBase
{
    //... SOME CODE
    [HttpGet]
    public IActionResult Get()
    {
        return Ok(/*... SOME CODE*/);
    }
}
```

Aca podemos ver como se puede llamar el filtro recién creado para este controlador. Estamos viendo un attribute que acepta parametros.

Muchos de los diferentes tipos de filtros tienen una clase base que se nos provee para poder utilizar y podamos modificar su implementacion. Estos son:

1. [ActionFilterAttribute](#)
2. [ExceptionFilterAttribute](#)
3. [ResultFilterAttribute](#)
4. [FormatFilterAttribute](#)
5. [ServiceFilterAttribute](#)
6. [TypeFilterAttribute](#)

Alcance de un filtro

Un filtro se puede agregar a la pipeline de las siguientes maneras:

- Como un attribute en un metodo de un controller.

```
//...SOME CODE
public class MyController : ControllerBase
{
    //...SOME CODE
    [HttpGet]
    [MyFilter]
    public IActionResult Get()
    {
        //...SOME CODE
    }
}
```

- Como un attribute en un controller
 - Visto en el ejemplo de arriba
- De forma global para todos los controllers y metodos.

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

- Este codigo se encuentra en la clase Startup. Vease que no se puede ubicar en Factory porque para hacer eso tendríamos que depender de WebApi y estaríamos haciendo una dependencia ciclica. Lo que se podria hacer es sacar los filtros para otro proyecto, pero no tiene mucho sentido hacer esto porque los filtros estan bien ubicados en WebApi, hacer eso seria agregar complejidad innecesaria.

Orden de ejecucion

Cuando hay multiples filtros para una etapa particular en la pipeline, el alcance determina el orden por defecto de la ejecucion de los filtros. Los filtros globales envuelven los filtros concretos que en consecuencia envuelven los metodos de estos.

Esto se llama filter nesting, veamos como seria la ejecucion:

- El codigo antes de un filtro global
 - El codigo antes de un filtro concreto aplicado a un controller
 - El codigo antes de un filtro concreto aplicado a un metodo
 - El codigo despues de un filtro concreto aplciado a un metodo
 - El codigo despues de un filtro concreto aplicado a un controller
- El codiog despues de un filtro global

Veamoslo con los metodos correspondientes:

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

Pero este orden se puede modificar, al implementar la interfaz `IOrderedFilter`. Esta interfaz tiene una propiedad `Order` que toma la precedencia sobre el alcance del filtro para determinar el orden de ejecución. Un filtro con un valor de `Order` bajo implica que:

- El código antes corre antes que el de un filtro con el valor de `Order` mayor
- El código después corre después que el de un filtro con el valor `Order` mayor

Para indicar este valor se hace de la siguiente manera:

```
[ApiController]
[Route("api/my_endpoint")]
[MyAttribute(Order = int.MinValue)]
public class MyController : ControllerBase
{
}
```

Cortando el camino de la request en la pipeline

La pipeline por la cual atraviesa la request se puede acortar, esto puede suceder porque el filtro no permite que avance más la request porque no cumple con alguna validación necesaria. Esto se logra al setear un valor en la propiedad `Result` (el setear un valor no-null a esta propiedad dentro de un filtro va a cortar la ejecución de filtros posteriores y la acción misma) que se encuentra en el contexto que se encuentra la request. Veamos como setear esta propiedad:

```
public class CuttingPipelineFilter : Attribute, IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        context.Result = new ContentResult()
        {
            StatusCode = 400,
        }
    }
}
```

```

        Content = "Metodo o controller no disponible"
    };
}

public void OnActionExecuted(ActionExecutedContext context)
{
}
}

```

Este código ilustra la creación de un filtro del tipo `IActionFilter` (la forma de cortar el camino de la request es independiente al tipo de filtro) y simplemente le setea el valor a la property `Result` que se encuentra en el contexto, y este le pone un código de error junto con un mensaje. Esta clase tiene más estados, la metadata está disponible para ver que se puede hacer con ella.

Inyección de dependencia

Los filtros pueden ser agregados por tipo o por instancia. Si es agregado por instancia, esa instancia es usada por cada request. Si es agregado por tipo, se dice que ese tipo está activado, esto quiere decir que:

- Una instancia es creada para cada request
- Cualquier dependencia en el constructor es llenado con inyección de dependencia.



Un filtro que es implementado como attribute y agregado directamente a un controller o método, no puede tener dependencias en el constructor. Esas dependencias no podrán ser llenadas por inyección de dependencia porque:

- Los attribute que tienen parámetros en sus constructores, se les tiene que brindar el valor cuando son utilizados
- Es una limitación de como los attribute funcionan

Los filtros que necesitan que se les inyecte una dependencia serían de la siguiente manera:

Código de mi filtro:

```

public class MyFilter : IActionFilter
{
    private readonly MyDependency myDependency;

    public MyFilter(MyDependency myDependency)
    {
        this.myDependency = myDependency;
    }
}

```



```
//...SOME CODE  
}
```

Codigo de mi controlador:

```
[ApiController]  
[Route("api/my_endpoints")]  
[ServiceFilter(typeof(MyFilter))]  
public class MyController : ControllerBase  
{  
}
```

Codigo de mi Startup:

```
public class Startup  
{  
    //...SOME CODE  
    public void ConfigureServices(IServiceCollection services)  
    {  
        //...SOME CODE  
        services.AddScoped<MyDependency, MyDependencyImplementation>();  
        services.AddScoped<MyFilter>();  
    }  
}
```

Estos tipos de filtros se les llama `ServiceFilterAttribute`. Estos filtros tienen que ser registrados en `ConfigureServices` como se muestra. Es el mismo proceso de cuando inyectamos dependencias. Estos filtros se utiliza una instancia del filtro por DI.

Luego tenemos los filtros que son por tipo. Estos son llamados `TypeFilterAttribute` que son similares a los `ServiceFilterAttribute`, pero su tipo no es resuelto directamente por el contenedor de servicios. Se realiza una instancia del tipo al utilizar una libreria particular, la cual es

[Microsoft.Extensions.DependencyInjection.ObjectFactory](#). Como estos filtros no son resueltos directamente por el contenedor de servicios estos pueden:

- Ser referenciados usando el tipo y no se necesita que sean registrados en el contenedor de servicios. Igualmente pueden acceder a servicios en el contenedor.
- Estos filtros pueden aceptar parametros en su constructor y se tienen que pasar el valor al referenciarlo.

Tipos de filtros

Filtros de autorización

- Se ejecutan antes que cualquier otro filtro y permiten evitar llegar al controller en caso de no cumplir con las políticas de seguridad.
- Estos se utilizan con el fin de autenticar y crear políticas de seguridad para nuestra aplicación web.
- Tiene un método antes pero no uno después

No solamente existe el tipo si no que existe un filtro ya creado para este tipo. Este filtro nos permite:

- Llamadas al sistema de autorización
- No autoriza requests

No se deberían de lanzar excepciones en este filtro, porque:

- La excepción no va a ser cacheada
- Los filtros de excepciones no van a poder agarrar esta excepción

Se puede tomar como desafío el manejo de errores cuando ocurre uno en este tipo de filtro 🐱.

Estos están comprendidos en la interface `IAuthorizationFilter`

Definición de la *interface* `IAuthorizationFilter`

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IAuthorizationFilter : IFilterMetadata {
        void OnAuthorization(AuthorizationFilterContext context);
    }
}
```

Analicemos este tipo:

El método `OnAuthorization` se utiliza para escribir el código para que el filtro pueda autorizar la solicitud. El parámetro `AuthorizationFilterContext` context, recibe los datos del contexto que describen la solicitud.

Filtros de acción

Se llaman justo antes de que se llame un método del *Controller* y justo después de que se termina un método del *Controller*. Se derivan de la interface de `IActionFilter`.

Definición de la *interface* `IActionFilter`.

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IActionFilter : IFilterMetadata {
        void OnActionExecuting(ActionExecutingContext context);
        void OnActionExecuted(ActionExecutedContext context);
    }
}
```

Al aplicar un filtro de acción a un método del *controller*, se llama al método `OnActionExecuting` justo antes de que se invoque el método del *controller*, y se llama al método `OnActionExecuted` justo después de que el método del *controller* haya terminado de ejecutarse.

El método `OnActionExecuting` tiene un parámetro del tipo `ActionExecutingContext`. Que destacaremos las siguientes propiedades de este:

Aa Nombre	Descripción
<u>Controller</u>	El nombre del controlador cuyo método está a punto de ser invocado.
<u>Result</u>	Esta propiedad es de tipo <code>ActionResult</code> . Si esta propiedad establece un valor de este tipo, se sobrescribe el resultado del método del Controller.

El método `OnActionExecuted` tiene un parámetro del tipo `ActionExecutedContext`. Que destacaremos las siguientes propiedades:

Aa Nombre	Descripción
<u>Controller</u>	El nombre del controlador cuyo método fue invocado.
<u>Exception</u>	Esta propiedad contiene la excepción que ocurrió en el método del Controller.
<u>ExceptionHandled</u>	Cuando establece su propiedad en true, las excepciones no se propagarán más.
<u>Result</u>	Esta propiedad devuelve el <code>ActionResult</code> devuelto por el método del Controller, y puede cambiarlo o reemplazarlo si lo necesita.

Ejemplo

En este ejemplo vamos a crear un filtro de acción que simplemente controle el tiempo de ejecución del método del *Controller*. Para esto vamos a iniciar un temporizador antes de la ejecución y lo vamos a parar luego de la ejecución para devolver el tiempo que duró esta ejecución.

```
public class ExampleActionFilter : Attribute, IActionFilter
{
    private Stopwatch timer;

    public void OnActionExecuting(ActionExecutingContext context)
    {
        timer = Stopwatch.StartNew();
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {

```

```

        timer.Stop();
        string result = " Tiempo de ejecucion: " + $"{timer.Elapsed.TotalMilliseconds} ms";
        ((ObjectResult)context.Result).Value = result;
    }
}

```

Para esto inicializamos este cronómetro en el método `OnActionExecuting` y lo detenemos en el método `OnActionExecuted`, cambiando en este último el valor de la respuesta por el tiempo de ejecución.

Filtros de resultados

Los filtros de resultados se ejecutan antes y después de que se procese el resultado del método del *Controller*. Se ejecutan después de los filtros de acción. Se derivan de la *interface* `IResultFilter`. Cabe destacar que los filtros de resultados tienen el mismo patrón que los filtros de acción.

Definición de la *interface* `IResultFilter`


```

namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IResultFilter : IFilterMetadata {
        void OnResultExecuting(ResultExecutingContext context);
        void OnResultExecuted(ResultExecutedContext context);
    }
}

```

La *interface* `IResultFilter` tiene 2 métodos. El método `OnResultExecuting` se llama justo antes de que se procese el resultado del método del *Controller*, mientras que el método `OnResultExecuted` se llama justo después de que se procesa el resultado del método del *Controller*.

El método `OnResultExecuting` tiene un parámetro del tipo `ResultExecutingContext`. Que destacaremos las siguientes propiedades:

<u>Aa</u> Nombre	 Descripción
<u>Controller</u>	El nombre del controlador cuyo método se invoca.
<u>Result</u>	Esta propiedad es de tipo <code>IActionResult</code> y contiene el objeto <code>IActionResult</code> devuelto por el método del <i>Controller</i> .
<u>Cancel</u>	Establecer esta propiedad en verdadero detendrá el procesamiento del resultado de la acción y dará una respuesta 404.

El método `OnResultExecuted` tiene un parámetro del tipo `ResultExecutedContext`. Que destacaremos las siguientes propiedades:

Aa Nombre	Descripción
<u>Controller</u>	El nombre del controlador cuyo método se invoca.
<u>Canceled</u>	Una propiedad de solo lectura que indica si la solicitud fue cancelada.
<u>Exception</u>	Contiene las excepciones lanzadas en el método del Controller.
<u>ExceptionHandled</u>	Cuando esta propiedad se establece en true, las excepciones no se propagan más.
<u>Result</u>	Una propiedad de solo lectura que contiene el IActionResult generado por el método del Controller.

Filtros de excepciones

Los filtros de excepciones permiten capturar excepciones sin tener que escribir el bloque try & catch. Pueden ser usados para implementar un mecanismo en comun para el manejo de errores.

Esto nos va a proveer un sistema mas limpio y leible ya que no seria necesario ensuciarlo como se dice con varios try & catch. Para esto implementaremos la interface `ExceptionHandler`.

Definición de la interface `ExceptionHandler`

```
namespace Microsoft.AspNetCore.Mvc.Filters {
    public interface IExceptionHandler : IFilterMetadata {
        void OnException(ExceptionContext context);
    }
}
```

Para esta interface, los datos de contexto se proporcionan a través de la clase `ExceptionContext`, que es un parámetro del método `OnException`. Destacaremos las siguientes propiedades:

Aa Nombre	Descripción
<u>Exception</u>	La propiedad contiene las excepciones que se lanzan
<u>ExceptionDispatchInfo</u>	Contiene los detalles de seguimiento de la pila de la excepción.
<u>ExceptionHandled</u>	Una propiedad de solo lectura que indica si se maneja la excepción
<u>Result</u>	Esta propiedad establece el IActionResult que se usará para generar la respuesta

Ejemplo

En este ejemplo vamos a capturar las excepciones que salten y vamos a devolver con una respuesta personalizada la solicitud.

```
public class ExampleExceptionHandler : Attribute, IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
        context.Result = new ContentResult()
        {
            StatusCode = 500,
            Content = "Se lanzo una excepcion con el siguiente mensaje: "
                    + context.Exception.Message
        };
    }
}
```



Obtenemos la excepción que se ejecutó y anexamos el mensaje a la respuesta. Es de gran utilidad para cuando repetimos muchos try & catch en los métodos de los Controllers.

Este es un ejemplo en donde devolvemos 500 ante cualquier excepcion no agarrada en el sistema. En la guia practica veremos una forma de agarrar todas las excepciones del sistema evitando try & catch en diferentes capas.

Referencias

- [Filters in ASP.NET Core](#)