CLASE 5

Middleware

Hasta ahora hemos mencionado que cuando un webapi recibe una petición HTTP, es una acción de un controlador la que recibe esta petición y la procesa. Profundizaremos más este circuito, en realidad cuando se recibe la petición HTTP, una petición HTTP llega al webapi y pasa por lo que se conoce como una tubería de peticiones HTTP (**HTTP Request Pipeline**).

Una tubería es una cadena de procesos conectados de tal forma que la salida de cada elemento de la cadena es la entrada del próximo elemento de la cadena. Entonces si la tubería de peticiones es el conjunto de procesos conectados que reciben una petición HTTP y la procesan para dar algún tipo de resultado, uno de esos procesos es el **proceso MVC** donde se manejan los controladores y las acciones pero no es el único proceso de la tubería.

Otro proceso importante es el **proceso de autorización** que es el que habilita la funcionalidad de denegar acceso a un recurso dependiendo si el usuario tiene permiso a acceder o no. Es normal configurar los permisos en el proceso MVC donde indicamos que una acción sólo puede ser consumida por determinados usuarios. Por lo tanto para que esta lógica de autorización funcione es importante haber pasado por el proceso de autorización. Esto implica que el orden de los procesos de esa tubería es importante.

La tubería de peticiones HTTP se configura en el método **Configure** de la clase **startup**. Vemos este ejemplo de una tubería de peticiones HTTP:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseAuthentication();
    app.UseMvc();
}
```

La tubería está formada por los métodos que estamos ejecutando sobre el **Application Builder** representado por el parámetro **app** del método **Configure**, y

también sirve para saber en qué ambiente estamos ejecutando la aplicación representado por el parámetro **env** de tipo IHostingEnvironment, por ejemplo, ambientes de producción, o desarrollo, o staging, etc., y así poder utilizar distintos procesos según el ambiente.

Observe también que los procesos indicados en el método **Configure** tienen un orden de ejecución específico, en la imagen vemos que el proceso de autenticación se configura antes de configurar el proceso MVC, al revés no tiene sentido porque MVC requiere para su funcionalidad poder utilizar autenticación que es activada en un paso anterior de la tubería, es decir en un "middleware anterior del pipeline". MVC requiere tener activada las autorizaciones para que las acciones de los controladores se ejecuten según autorizaciones definidas protegiendo los recursos.

Un proceso puede además detener la ejecución de la tubería y devolver una respuesta sin necesidad de pasar por los procesos faltantes. Esto es llamado **shortcircuit** donde un middleware puede detener el proceso de la tubería y dar una respuesta al usuario.

Filtros

Los filtros ayudan a ejecutar código condicional en determinados momentos del ciclo de vida del procesamiento de una petición HTTP. Hasta ahora nos hemos enfocado en las acciones sin embargo **antes y después de ejecutar una acción del controlador** tenemos la opción de ejecutar código que "*responda a eventos*" utilizando filtros.

Los filtros son útiles cuando tenemos la necesidad de ejecutar una lógica en varias acciones de varios controladores y queremos evitar tener que repetir código. Uno de los filtros más comúnmente utilizados es el **filtro de autorización**. Este filtro permite bloquear el acceso a un recurso por ejemplo, cuando un usuario no está logeado, siendo esta lógica una situación común para múltiples acciones de múltiples controladores, o a determinados usuarios dejarles consumir una acción determinada y a otros usuarios no.

Existen varios tipos de filtros:

- **Filtros de autorización** que determinan si un usuario puede consumir una acción específica.
- **Filtros de recursos** que se ejecutan después de la etapa de autorización y se pueden utilizar para validaciones generales o para implementar una capa de caché. Además estos filtros pueden detener la tubería de filtros de tal modo que ni los demás filtros ni acciones del controlador se ejecuten.
- **Filtros de acción** que se ejecutan antes y después de la ejecución de una acción y se pueden utilizar para manejar los argumentos enviados a una acción o la información retornada por los mismos.
- **Filtros de excepción** que se ejecutan cuando hay una excepción no atrapada por un **try-catch** durante la ejecución de una acción, durante la ejecución de

un filtro de acción, durante la creación de un controlador y durante el binding del modelo.

- **Filtros de resultado** que se ejecutan antes y después de la ejecución de un **ActionResult**.

La forma típica de aplicar un filtro es a nivel de una acción, la idea es que la lógica de dicho filtro se va a ejecutar cuando se vaya a procesar una petición HTTP sobre la acción marcada con el filtro.

Los filtros también se pueden aplicar a nivel del controlador de tal modo que aplica todas las acciones de dicho controlador.

Estos filtros se colocan como atributos en las acciones o en los controladores correspondientes.

También podemos aplicar filtros a nivel de todo el Web API de modo que dicho filtro se aplique a todas las acciones de todos los controladores del proyecto.

Filtro de acción

Un ejemplo de un **filtro de nivel de una acción** es **CacheResponse** que sirve para guardar en cache la respuesta de una acción. Para utilizar este filtro se necesita agregarlo en la clase **startup** en el método **ConfigureServices** llamando al método **AddResponseCaching**:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddResponseCaching();
```

Cuando agregamos/configuramos servicios desde el objeto **services**, básicamente lo que estamos haciendo es que internamente esta función **AddResponseCaching** u otra función **Add**…, está habilitando un conjunto de servicios relacionados a una lógica determinada, en este caso estamos habilitando un conjunto de servicios para la funcionalidad de guardar información en caché.

Como segundo paso debemos ir al método **Configure** también de la clase **startup** donde están los middleware, para configurar el middleware de caché, que tiene que estar antes del middleware MVC, porque el caché lo vamos a usar en MVC y necesitamos que esté activado antes de llegar a MVC.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change th
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseResponseCaching();
    app.UseMvc();
}
```

Vamos a probar este filtro de acción yendo a la clase **ValueController** y cambiaremos el método **Get** que vez de retornar un **IEnnumerable** de strings retornará sólo un string que retorne segundo actual:

```
public ActionResult<string> Get()
  {
      return DateTime.Now.Second.ToString();
  }
```

Y lo que haremos es guardar en caché esta respuesta así estamos seguros que realmente se retorna la información guardada en caché y no la última información que devuelve el controlador.

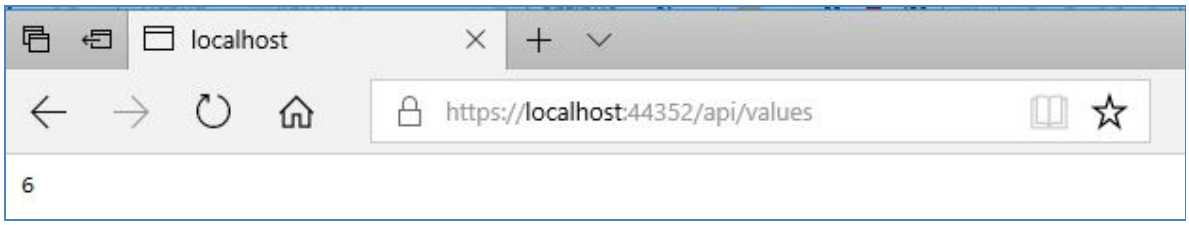
Le agregamos a la acción el atributo ResponseCache para indicar el tiempo que queremos se guarde esta información en caché, 15 segundos:

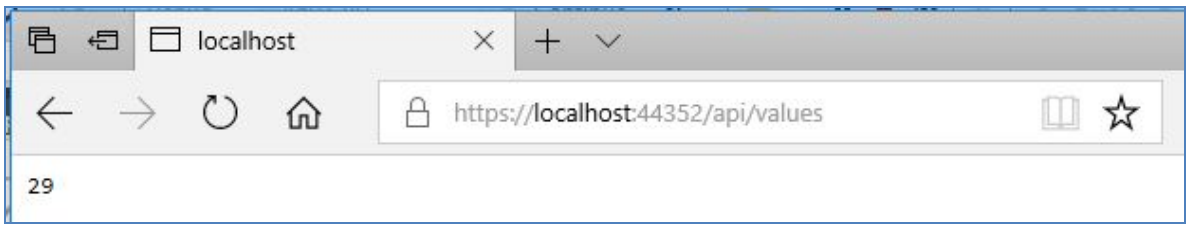
```
// GET api/values
        [HttpGet]
        [ResponseCache(Duration = 15)]
        public ActionResult<string> Get()
        {
            return DateTime.Now.Second.ToString();
        }
```

Es decir si hacemos una petición ahora y esa petición retorna un 10, estamos en el segundo 10 y por los próximos 15 segundos va a seguir retornando la misma respuesta por 15 segundos.

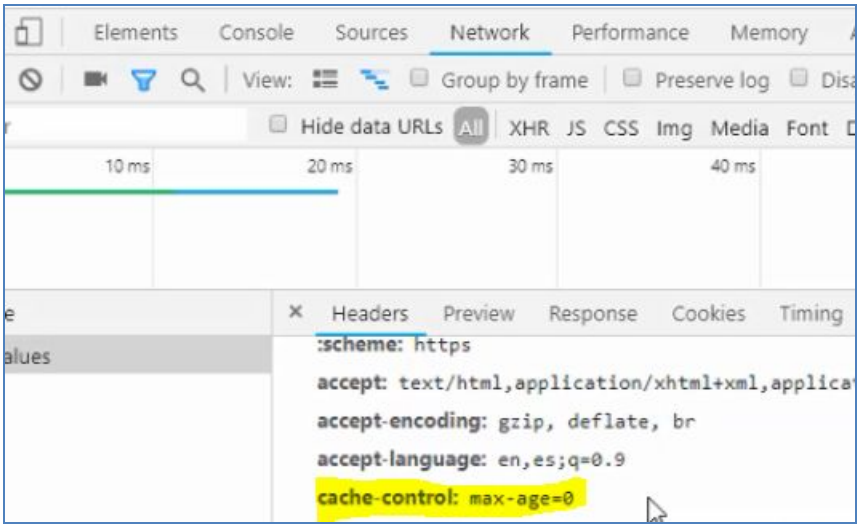
Compilamos y probamos, recuerde que debe tener activado el caché en la clase **startup**:

Observamos que retorna el segundo 6 y si sigue con <F5> refrescando la página seguirá mostrando segundo 6 hasta que se agoten los 15 segundos y el caché expira. Luego mostrará un nuevo valor, el segundo 29.





Nota: este ejercicio fue realizado con Microsoft Edge, también puede probarlo con Firefox, tenga en cuenta que si lo prueba con Chrome es posible que no funcione dado que en la cabecera de la petición HTTP en el atributo **cache-control** se indica que no está usando el caché, puede verlo desde la herramienta del desarrollador <F12>:



Filtro de autorización

Otro filtro bastante utilizado es el **filtro de autorización**, indicando que una acción solamente puede ser accedida por determinados usuarios que cumplan con alguna

condición. El uso más simple es indicar que una acción sólo puede ser utilizada por usuarios que estén autenticados en el webapi. Veremos un ejemplo:

En la clase **startup** indicamos lo siguiente en el método **ConfigureServices**:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddResponseCaching();
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwt
Bearer();
```

Deberá agregar el namespace que posibilita usar el tipo de autenticación **JSON Web Token:**

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

Nota: JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma para transmitir información de forma segura entre las partes y a través de un objeto JSON. Esta información se puede verificar y es confiable porque está firmada digitalmente. Los JWT pueden firmarse usando un "secret" (con el algoritmo HMAC) o un par de claves pública/privada usando RSA o ECDSA. Más información en este link https://jwt.io/introduction/ y un ejemplo simple en https://www.c-sharpcorner.com/article/jwt-json-web-token-authentication-in-asp-net-core/

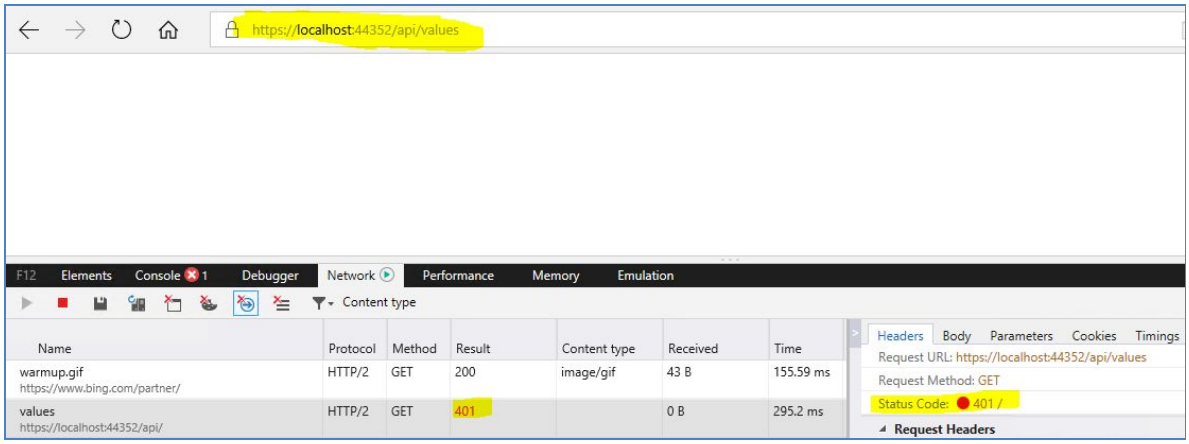
Luego en el método **Configure** debemos activar la autenticación antes del middleware MVC porque en MVC es donde ubicaremos los atributos de autorización.

```
       public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseResponseCaching();
            app.UseAuthentication();
            app.UseMvc();
        }
```

En el ValuesController y a nivel de acción agregamos el atributo [Authorize] con el namespace correspondiente `using Microsoft.AspNetCore.Authorization;`:

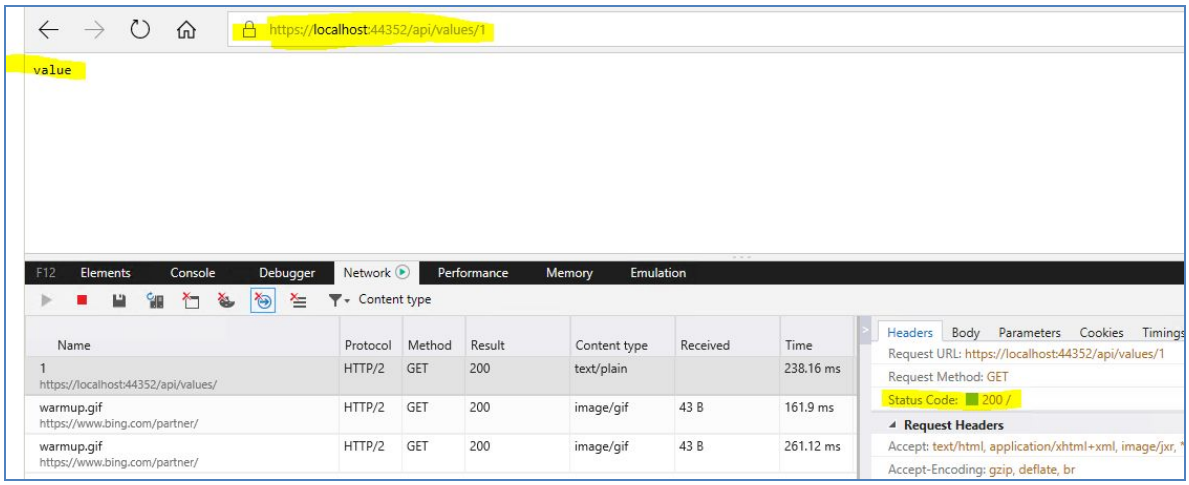
```
// GET api/values
[HttpGet]
[ResponseCache(Duration = 15)]
[Authorize]
public ActionResult<string> Get()
{
    return DateTime.Now.Second.ToString();
}
```

Con esto ya estamos protegiendo este recurso de tal modo que solamente quienes estén autenticados en el web API van a poder acceder al mismo y si tratamos de acceder al recurso desde el navegador, observe que da un **HTTP Error 401** que significa **unauthorized**, es decir que no estamos autorizados para visualizar o utilizar ese recurso.



Tenga en cuenta que también podemos utilizar este filtro a nivel del controlador para que afecte a todas las acciones del controlador, de hecho si accedemos a este endpoint https://localhost:44352/api/values/1 retorna un valor dado que la acción **Get** con el parámetro entero no tiene el filtro de autorización:

```
// GET api/values/5
[HttpGet("{id}")]
public ActionResult<string> Get(int id)
{
    return "value";
}
```



Entonces si quisiéramos que la autorización valga para todas las acciones agregamos el filtro a nivel de controlador como en este ejemplo:

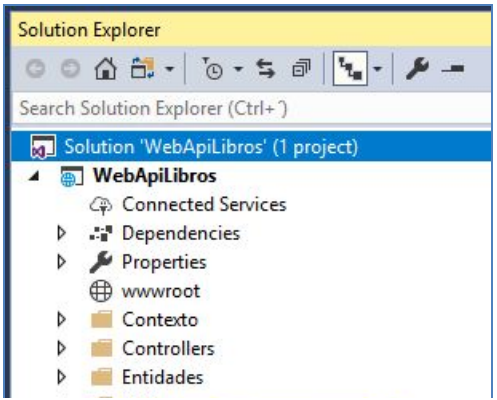
```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;

namespace WebApiLibros.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [Authorize]
    public class ValuesController : ControllerBase
    { …
```

Filtros personalizados

No estamos limitados solamente a utilizar los filtros que nos provee el framework, podemos también crear nuestros propios filtros.

Recordamos que existen distintos tipos de filtros que se ejecutan en distintos momentos del ciclo de vida del procesamiento de una petición HTTP.

Filtro de acción personalizado



Vamos a crear un filtro de acción, y lo hacemos creando una clase que implemente de una de las siguientes interfaces: **IActionFilter** o **IAsyncActionFilter** para filtro asincrónico.

Para que estos agregados personalizados queden bien organizados en el proyecto, usaremos la carpeta **Helpers** que ya teníamos en el proyecto (si no la tiene cree la carpeta), y dentro de ella la nueva clase llamada **FiltroAccionPersonalizado**.

En la clase implementamos la interfaz **IActionFilter** y tendrá este código que analizaremos:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.Extensions.Logging;

namespace WebApiLibros.Helpers
{
    public class FiltroAccionPersonalizado : IActionFilter
    {
        private readonly ILogger<FiltroAccionPersonalizado> logger;

            // Inyección de dependencias para inyectar un logger así
      logear información
        public
FiltroAccionPersonalizado(ILogger<FiltroAccionPersonalizado> logger)
        {
            this.logger = logger;
        }
        public void OnActionExecuting(ActionExecutingContext context)
        {
            logger.LogError("OnActionExecuting");
        }
        public void OnActionExecuted(ActionExecutedContext context)
        {
            logger.LogError("OnActionExecuted");
        }
    }
}
```

Esta clase implementa la interfaz hay **IActionFilter** y en esta interfaz nos pide que implementemos los métodos **OnActionExecuting** y **OnActionExecuted**.

OnActionExecuting se ejecuta antes de la acción y **OnActionExecuted** después de la acción. También se usa inyección de dependencias para inyectar el logger así escribimos en el log el orden de ejecución de estas funciones según los eventos que se van sucediendo.

Dado el hecho de que estamos utilizando inyección de dependencias en el filtro debemos utilizar una notación especial para aplicarlo a una acción. En el controlador

de autores y para hacer más simple el ejemplo agregaremos un nuevo endpoint **[HttpGet]** y ahora con respecto al filtro agregamos el atributo **ServiceFilter**:

```
        [HttpGet("/listado")]
        [HttpGet("listado")]
        [HttpGet]
        [ServiceFilter(typeof(FiltroAccionPersonalizado))]
        public ActionResult<IEnumerable<Autor>> Get()
        {
            return context.Autores.ToList();
        }
```

Reiteramos la razón por la que tenemos que utilizar **ServiceFilter** es porque estamos utilizando inyección de dependencias, si no tuviéramos inyección de dependencias no sería necesario utilizarlo..

Finalmente debemos configurar el servicio del filtro de acción en la clase **startup** en el método **ConfigureServices:**

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<FiltroAccionPersonalizado>();
            services.AddResponseCaching();

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwt
Bearer();
            services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
String")));


services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_
1)
                .AddJsonOptions(options =>
options.SerializerSettings.ReferenceLoopHandling=Newtonsoft.Json.Referenc
eLoopHandling.Ignore);

        }
```

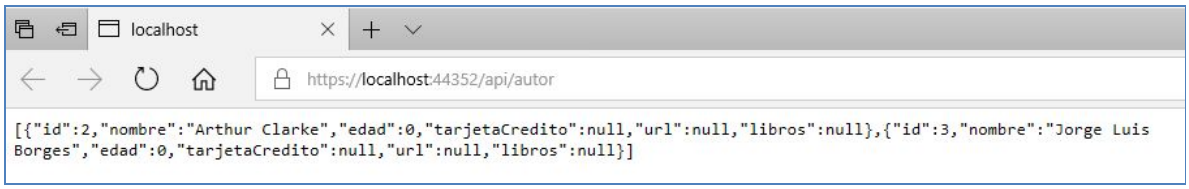
Para visualizar mejor los eventos que se ejecutan para la acción, ubicamos breakpoints en las dos funciones y en la acción **Get** de **AutorController**. Y si presionamos F5 para ejecutar el proyecto en modo debugging vemos que primero se ejecuta la función **OnActionExecuting**, luego la acción **Get** y por último la función **OnActionExecuted**.

```
FiltroAccionPersonalizado.cs | AutorController.cs | ApplicationDbContext.cs | Autor.cs | LibroController.cs | Startup.cs
WebApiLibros | WebApiLibros.Helpers.FiltroAccionPersonalizado | OnActionExecuting(ActionExecuting
10  public class FiltroAccionPersonalizado : IActionFilter
11  {
12      private readonly ILogger<FiltroAccionPersonalizado> logger;
13
14      // Inyección de dependencias para inyectar un logger así logear información
15      public FiltroAccionPersonalizado(ILogger<FiltroAccionPersonalizado> logger)
16      {
17          this.logger = logger;
18      }
19      public void OnActionExecuting(ActionExecutingContext context)
20      {
21          logger.LogError("OnActionExecuting");  ≤ 1ms elapsed
22      }
23      public void OnActionExecuted(ActionExecutedContext context)
24      {
25          logger.LogError("OnActionExecuted");
26      }
```

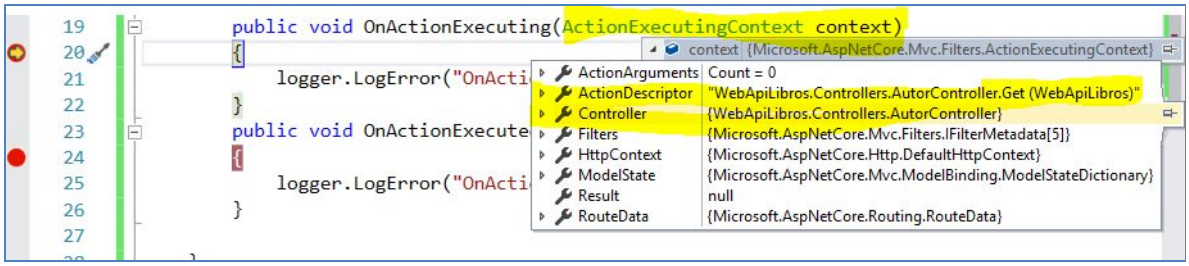
```
31          [HttpGet("/listado")]
32          [HttpGet("listado")]
33          [HttpGet]
34          [ServiceFilter(typeof(FiltroAccionPersonalizado))]
35          public ActionResult<IEnumerable<Autor>> Get()
36          {
37              return context.Autores.ToList();  ≤ 1ms elapsed
38          }
```

```
23      public void OnActionExecuted(ActionExecutedContext context)
24      {
25          logger.LogError("OnActionExecuted");  ≤ 1ms elapsed
26      }
```

Y por último el resultado de la acción **Get** de **Autor**: El filtro de acción está funcionando.

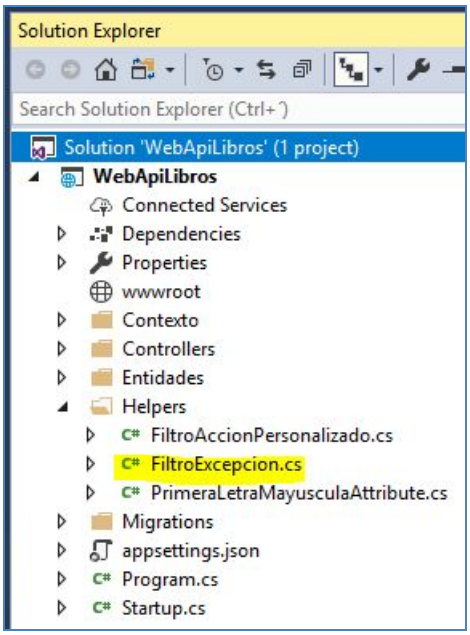


Es importante destacar que debuggeando los filtros podemos acceder al **contexto de ejecución** del mismo. Si vemos la siguiente imagen, nos muestra desde el filtro que acción y qué controlador se está ejecutando:



Filtro de excepción personalizado (filtro global)

Además de poder aplicar filtros a nivel de acciones y a nivel de controladores, podemos crear **filtros globales**; estos se aplicarán en todas las acciones del webapi.



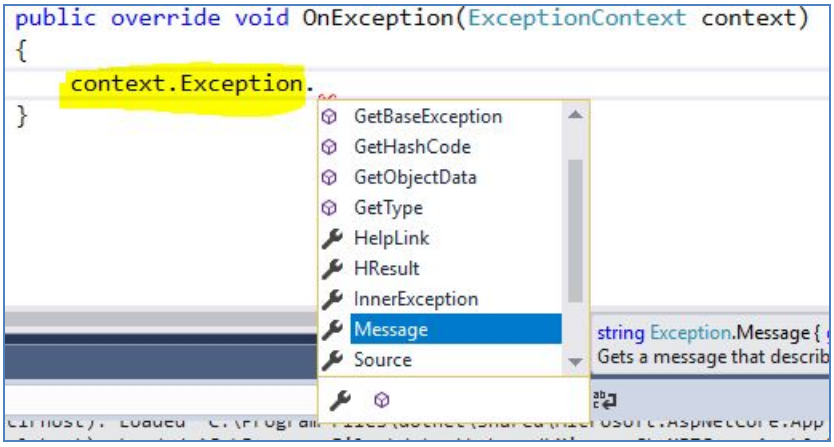
Haremos un filtro de excepción que sólo se ejecutará cuando ocurra un error en la aplicación. Con este filtro podemos personalizar el manejo de excepciones según distintos parámetros cómo por ejemplo, el endpoint utilizado así saben en qué endpoint ocurrió el error.

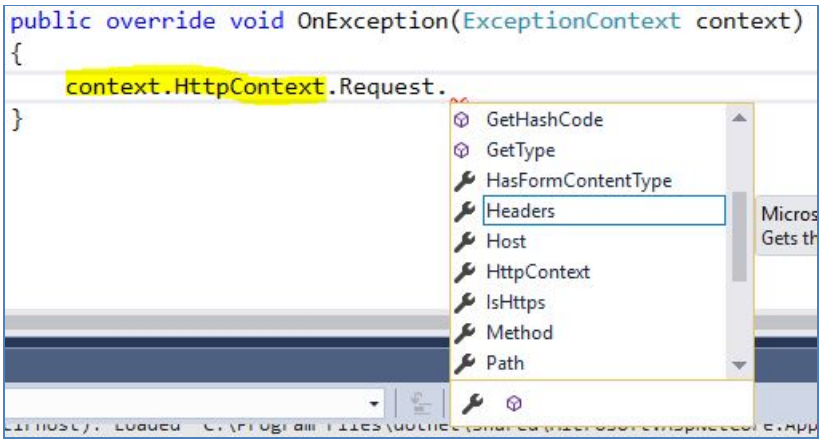
Creamos un nuevo filtro llamado **FiltroExcepcion** en la carpeta **Helpers**. Este filtro heredará de **ExceptionFilter** y tendrá el siguiente código:

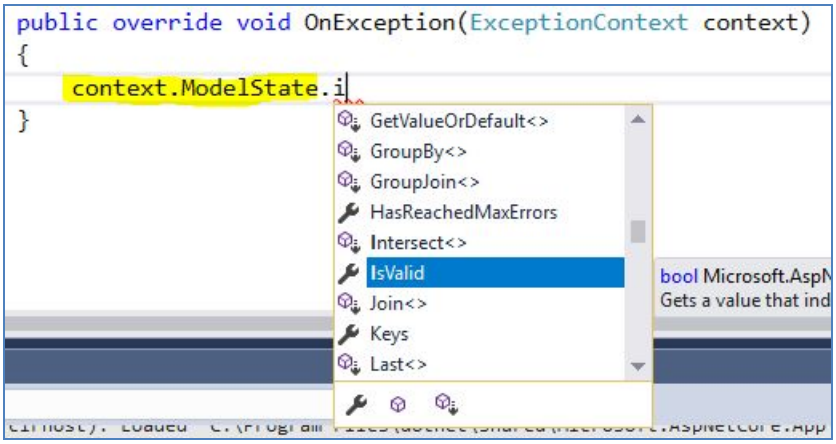
```
using Microsoft.AspNetCore.Mvc.Filters;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApiLibros.Helpers
{
    public class FiltroExcepcion : ExceptionFilterAttribute
    {
        public override void OnException(ExceptionContext context)
        {

        }
    }
}
```

Hacemos un override al método **OnException**, aquí podemos hacer lo que queramos por ejemplo acceder al contexto y analizar la excepción que ocurrió, y otros

informaciones como el contexto HTTP, el ModelState, el resultado, los datos de ruta, etc.







Ubicamos un break en el método **OnException**, y configuraremos previamente el filtro de excepción en la clase **startup**, en el método **ConfigureServices** y dentro de **AddMvc** configuramos las siguientes opciones para el filtro global:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<FiltroAccionPersonalizado>();
            services.AddResponseCaching();

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwt
Bearer();
            services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
String")));

            services.AddMvc(Options =>
            {
                Options.Filters.Add(new FiltroExcepcion());
            }).SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
                .AddJsonOptions(options =>
options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore);
        }
```

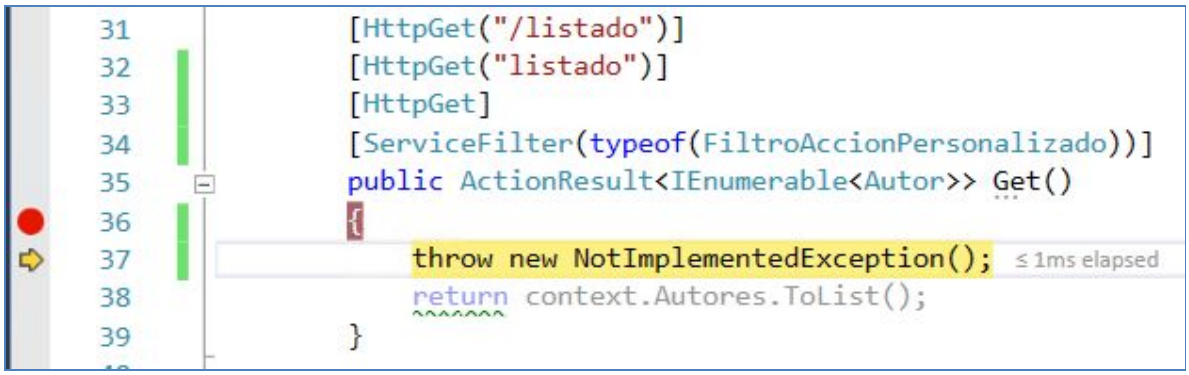
.

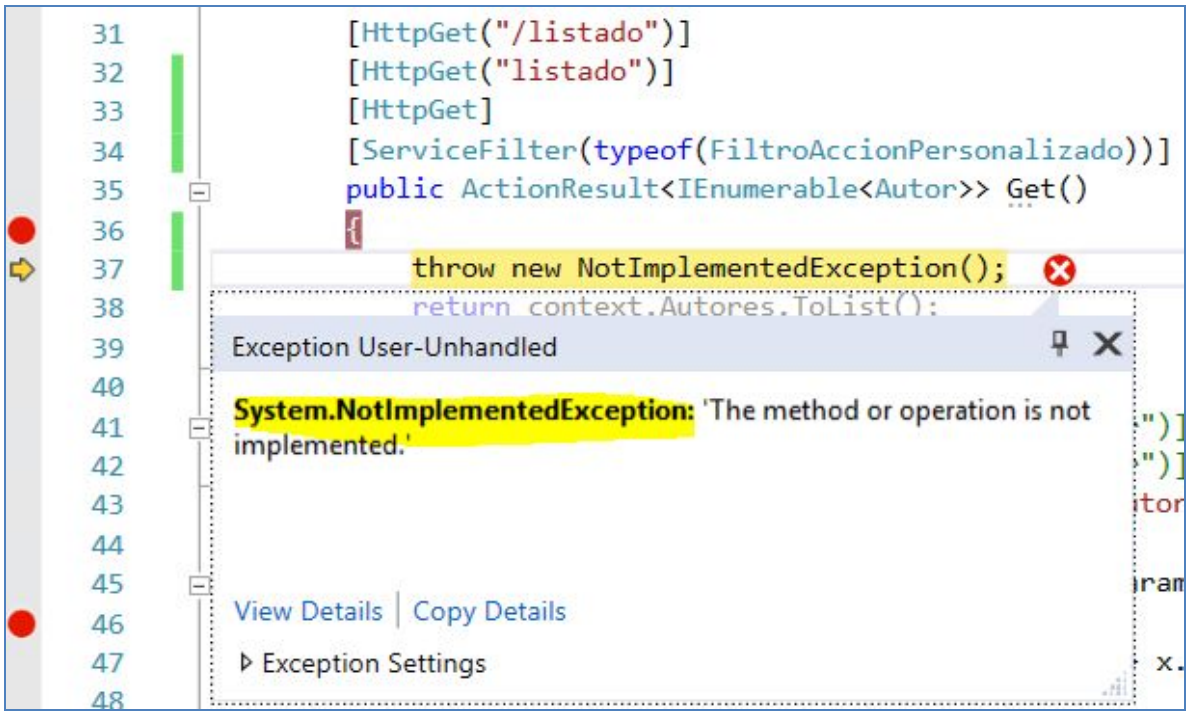
Directamente en el método **Add** hacemos instanciamos la clase **FiltroException** directamente porque en este filtro no estamos utilizando inyección de dependencias. Si estuviéramos utilizando inyección de dependencias no podríamos instanciarlo aquí, sino que lo que haríamos es `Options.Filters.Add(typeof(FiltroExcepcion);`

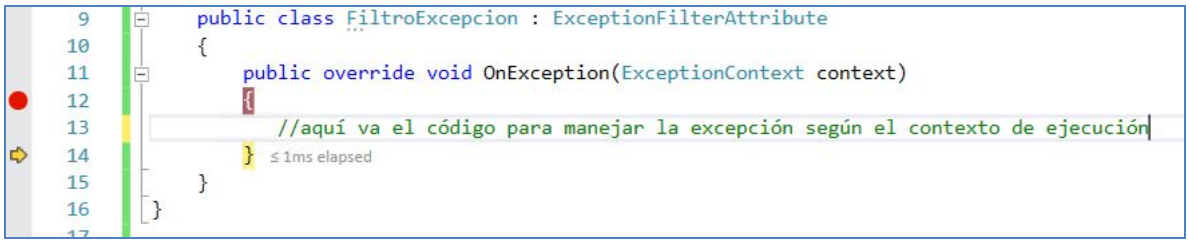
Lo probamos yendo a **AutorController**, y lanzar una excepción en la acción **Get** y vamos a ver que cuando querramos utilizar la acción Get se va a ejecutar el filtro de excepción:

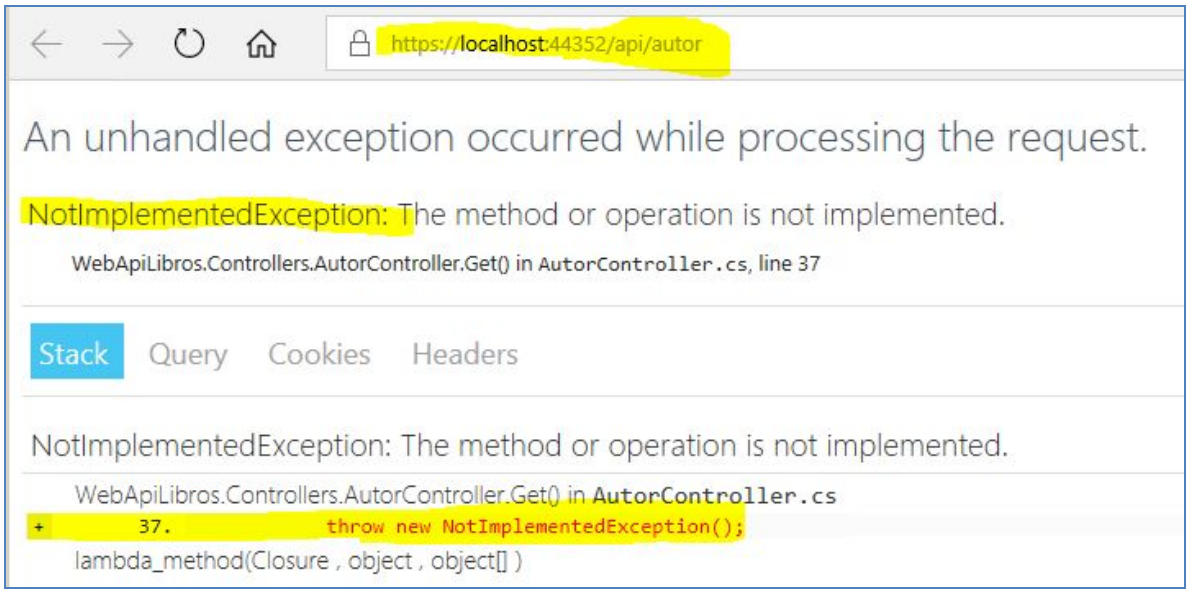
```
        public ActionResult<IEnumerable<Autor>> Get()
        {
            throw new NotImplementedException();
            return context.Autores.ToList();
        }
```

Asegúrese de tener los breakpoints necesarios para probar con **<F5>** y **<F11>** (si aún tiene los breakpoints en los eventos del filtro de acción personalizado, también pasará por allí):


[HttpGet("/listado")]
[HttpGet("listado")]
[HttpGet]
[ServiceFilter(typeof(FiltroAccionPersonalizado))]
public ActionResult<IEnumerable<Autor>> Get()
{
throw new NotImplementedException();
≤ 1ms elapsed
return context.Autores.ToList();
}


[HttpGet("/listado")]
[HttpGet("listado")]
[HttpGet]
[ServiceFilter(typeof(FiltroAccionPersonalizado))]
public ActionResult<IEnumerable<Autor>> Get()
{
throw new NotImplementedException();
Exception User-Unhandled
System.NotImplementedException: 'The method or operation is not implemented.'
View Details
Copy Details
Exception Settings


public class FiltroExcepcion : ExceptionFilterAttribute
{
public override void OnException(ExceptionContext context)
{
//aquí va el código para manejar la excepción según el contexto de ejecución
}
≤ 1ms elapsed
}
}


https://localhost:44352/api/autor
An unhandled exception occurred while processing the request.
NotImplementedException: The method or operation is not implemented.
WebApiLibros.Controllers.AutorController.Get() in AutorController.cs, line 37
Stack
Query
Cookies
Headers
NotImplementedException: The method or operation is not implemented.
WebApiLibros.Controllers.AutorController.Get() in AutorController.cs
+ 37. throw new NotImplementedException();
lambda_method(Closure , object , object[])