Guia Filter ASP.NET Core

Veamos de implementar dos tipos de filtros que van a ser de mucha ayuda para nuestro sistema. Esos tipos son el de autorizacion y el de excepciones.

- El filtro de autorizacion lo vamos para corroborar si un usuario tiene los permisos suficientes para acceder algunos recursos en particulares. Vamos a ver dos formas de implementar este filtro.
- Con el filtro de excepciones vamos a mejorar el que se mostraba en la guia teorica de fomra de hacer un manejador de excepciones bastante generico y lograr una limpiza en el sistema de try & catch.

Empecemos a programar 🤘

Authorization filter

Primero creemos este filtro. Como ya se menciono este filtro nos va a servir para autorizar a los usuarios el acceso a determinados recursos. Antes de empezar a crear clases como locos, analicemos donde deberia de estar esta clase:

- En un proyecto solamente para filtros
    - Podria ser una opcion valida de forma de seguir la estructura de desacoplamiento, pero podria ser una solucion complicada sin muchas ventajas. Como vimos anteriormente si o si necesitamos saber el tipo del filtro concreto por ende el desacoplamiento no existitira. La mejor justificacion que podemos decir para esta solucion es porque queremos

reusar nuestros filtros para otros sistemas sin la necesidad de llevarme cosas innecesarias.

- En el proyecto WebApi
  - Este podria ser el lugar ideal ya que WebApi es el encargado de procesar, manejar las requests que le llegan de los diferentes clientes. De todas formas tenemos que tener cuidado porque no estaria bien que esten situados en el mismo paquete que los controladores porque estos tienen otra responsabilidad. Lo que nos llevaria a realizar un inner package para los controllers llamado WebApi.Controllers y otro para los filtros llamado WebApi.Filters.

La solucion que vamos a codificar es la que esta resaltada en verde, ambas soluciones tienen justificaciones fuertes, pero realizaremos esta por un tema de no seguir agregando complejidad.

Antes de comenzar crearemos una carpeta en el proyecto WebApi que se llame Filters, y cambiaremos el paquete de los controllers a WebApi.Controllers.

Cambiando el paquete de los controllers:

```
namespace WebApi.Controllers
{
  [ApiController]
  [Route("api/movies")]
  public class MovieController : ControllerBase
  {
  }
}
```

Una vez que se agregue la carpeta filters estamos listo para crear nuestros filtros.

AuthorizationFilter

Empecemos con el filtro de autorizacion. Para realizar este filtro vamos a realizar nuestro propio filtro, eso quiere decir que no vamos a utilizar uno ya creado.

```
namespace WebApi.Filters
{
  public class AuthorizationAttributeFilter : Attribute, IAuthorizationFilter
  {
    public void OnAuthorization(AuthorizationFilterContext context)
    {
      //...SOME CODE
    }
  }
}
```

Analicemos este codigo:

1. El nombre de la clase contiene tres partes: NameAttributeFilter, la primera parte es el nombre, la segunda parte es la indicacion de que es un attribute y la tercera parte es la indicacion de que es un filter.
2. Luego tenemos que extender de la clase Attribute para poder utilizar este filtro de la forma de Attribute
3. Luego implementamos el tipo de filtro que queremos realizar, en este caso como es de autorizacion hay que implementar la interfaz IAuthorizationFilter
4. Al implementar esta interfaz hay que realizar solamente un comportamiento, el cual se ejecutara antes que cualquier otro codigo del controller.
5. Este metodo recibe un contexto, el contexto de la request, con el cual vamos a poder examinar muchas cosas del mismo.

AuthorizationFilterContext

Como se menciono anteriormente, este tipo de parametro que esperamos que se nos pase en la llamada del metodo, nos va a permitir analizar muchas cosas de la request. La que nos interesa en este filtro son los Headers que nos manda el cliente en la request para que la api pueda identificar quien es el usuario que origino la request. Esto es asi porque no se olviden que las WebApi son steatlless.

Veamos como hacemos para ver la informacion de los headers:

```
public class AuthorizationAttributeFilte : Attribute, IAuthorizationFilter
{
  public void OnAuthorization(AuthorizationFilterContext context)
  {
    var token = context.HttpContext.Request.Headers["Authorization"];
  }
}
```

De esta forma podemos sacar el valor del header con key Authorization. En caso de que la request no tenga este valor va a ser null. Para poder agarrar el valor de un header en particular, el cliente tuvo que haber puesto exactamente la misma key en su request que la que se espera en este metodo. Obviamente que esa informacion deberia de estar documentada en la documentacion de la API para que los clientes que la utilicen sepan como es el formato de los headers y para que sirven.

Sigamos adelante, ahora deberiamos de ver si efectivamente la request tiene ese header o no.

```
public class AuthorizationAttributeFilte : Attribute, IAuthorizationFilter
{
  public void OnAuthorization(AuthorizationFilterContext context)
  {
    var token = context.HttpContext.Request.Headers["Authorization"];

    if(String.IsNullOrEmpty(token))
    {
      context.Result = new ContentResult()
      {
        StatusCode = 401,
        Content = "Mandame un token papo"
      };
    }
  }
}
```

Ahora codificamos la parte de chequear que exista un token. Para eso vemos si es vacio o null y en caso de serlo tenemos que tomar medidas. Esas medidas que se tienen que tomar son de cortar el transcurso de la request. Para hacer eso se tiene que setear el valor de la property Result que se encuentra en el context. Result es de tipo IActionResult, del mismo que retornamos en nuestros metodos de los controllers 😱. Entonces tenemos que realizar una instancia de un codigo de error, el mejor para este escenario es el 401 el cual indica Unauthorized, que su uso es similar al de 403 pero cuando la autenticacion fallo o no fue provista (falta login). Luego de setear este codigo de error le seteamos la descripcion de la misma. Tambien podriamos setearle el tipo de body que es la response.

Una vez seteada automaticamente cortamos el circuito de ejecucion de la request causando que no ejecute ningun codigo que venga despues en el pipeline. Eso no significa que no va a ejecutar codigo, si existe, despues del seteo. Si queremos evitar que eso ocurra podemos realizar if else o el uso del return, por mas que el tipo del retorno sea void. Esta ultima opcion no es muy clean code.

Sigamos adelante, ahora veamos que debemos de hacer cuando efectivamente la request tiene un header con esa key y un valor

```
public class AuthorizationAttributeFilte : Attribute, IAuthorizationFilter
{
  public void OnAuthorization(AuthorizationFilterContext context)
  {
    var token = context.HttpContext.Request.Headers["Authorization"];

    if(String.IsNullOrEmpty(token))
    {
      context.Result = new ContentResult()
      {
        StatusCode = 401,
        Content = "Mandame un token papo"
```

```
      };
    }
    else
    {
      if(!sessions.IsCorrectToken(token))
      {
        context.Result = new ContentResult()
        {
          StatusCode = 403,
          Content = "No tenes permisos mostro"
        };
      }
    }
  }
}
```

En esta parte del codigo estamos preguntandole a un servicio, que aun no sabemos de donde sale, si el token de la request es valido. La decision de preguntarle a un servicio si el token es valido es porque estamos delegandole esta responsabilidad a otro servicio para no tenerlo en el filtro. De esta forma podemos lograr las siguientes cosas:

- Reuso de la logica de la sesion
- Remplazo de la logica de sesion
- Bajo acoplamiento
- Respeto SRP
- Respeto OCP
- Se modulariza por responsabilidad

Basicamente lo que logramos es que el filtro solamente conozca una interfaz que exponga comportamiento de la sesion, solamente le interesa utilizarla, gracias a esto del otro lado, del lado de la implementacion, puedo tener diferentes implementaciones con respecto a la sesion y nuestro filtro ni se entera cual utiliza porque no le interesa.

Esto quiere decir que podemos tener una implementacion re acoplada a nuestro sistema, que solo tenga sentido con nuestro negocio como tambien podriamos realizar una implementacion sobre sesion mas generica como podria ser con el uso de JWT y esta implementacion reusarla en otro sistema.

Fascinante 💣

Veamos como obtener ese serivioc. Las opciones que tenemos es obtener ese servicio manualmente o automaticamente, la diferencia va a ser en el mecanismo que se utilice. Si se decide de ir a buscar el servicio al contenedor de servicios seria manual y si se decide que se inyecte la dependencia seria automaticamente.

Forma manual

Para realizar la forma manual es necesario ir a buscar el servicio al lugar donde estan los servicios registrados. Esto lo podemos realizar ya que sabemos el contexto en el cual se encuentra la request por ende otra de las cosas que nos provee el contexto es el contenedor de servicios. Veamoslo:

```
public class AuthorizationAttributeFilte : Attribute, IAuthorizationFilter
{
  public void OnAuthorization(AuthorizationFilterContext context)
  {
    var token = context.HttpContext.Request.Headers["Authorization"];

    if(String.IsNullOrEmpty(token))
    {
      context.Result = new ContentResult()
      {
        StatusCode = 401,
        Content = "Mandame un token papo"
      };
    }
    else
    {
      var sessions = this.GetSessionLogic(context);

      if(!sessions.IsCorrectToken(token))
      {
        context.Result = new ContentResult()
        {
          StatusCode = 403,
          Content = "No tenes permisos mostro"
        };
      }
    }
  }

  public ISessionLogic GetSessionLogic(AuthorizationFilterContext context)
  {
    var sessionType = typeof(ISesionLogic);

    return context.HttpContext.RequestServices.GetService(sessionType) as ISessionLogic;
  }
}
```

Este nuevo codigo lo que hace es:

1. Primero llama un metodo local el cual es el encargado de devolver el servicio previamente registrado, para utilizarlo.
2. Lo que hace el metodo local es, a partir del tipo de un servicio, se busca en el contenedor de servicios para retornarnlo. El metodo GetService(Type) retorna un object por lo cual si no hacemos el down cast nos lo marcaria en rojo diciendo que falta una conversion implicita. Esto es porque object es padre de

todos y no podemos retornar un padre en lugar del hijo porque rompemos con Liskov el cual dice que podemos poner a cualquier hijo en el lugar del padre y tratarlo como si fuese el padre. Es por eso que se tiene que hacer el down cast.

Ahora se deben de estar preguntando de donde sale esa interfaz no? La respuesta no es sencilla ni corta. Previamente habiamos dicho las ventajas de haber deducido de tener que delegar la responsabilidad de la sesion a alguien mas, ahora queda saber como implementar esa idea.

Una de las cosas que pensamos fue en el reuso, pensamos en esto porque la sesion es algo que esta en todos los sistemas, o al menos aquellos que permiten un logueo. Para ello una forma es abstraer esa logica en otro proyecto y acoplarnos a el.

Dicho esto deberiamos de pasar a crear un proyecto que se llame SessionInterface y referenciarlo en WebApi.

> *dotnet new classlib -n SessionInterface*
> *dotnet sln add SessionInterface*
> *dotnet add WebApi reference SessionInterface*

Una vez realizado el proyecto le tenemos que agregar la interfaz ISessionLogic con el metodo que se usa en el codigo. Dicha interfaz quedaria de la siguiente manera:

```
public interface ISessionLogic
{
  bool IsCorrectToken(string token);
}
```

Otra cosa que deberiamos de discutir es donde iria la implementacion. Nuevamente la respuesta no es sencilla ni corta. Se puede ubicar en muchos lugares, algunos buenos lugares son:

- BusinessLogic
- SessionLogic

Analicemos estas opciones.

En BusinessLogic lo pondria si la implementacion esta muy arraigada a nuestro negocio lo que no nos permitira reutilizarla en otro sistema. Esto quiere decir que esta muy acoplado a nuestro negocio como tambien a nuestra tecnologica capaz.

Ahora si quisieramos reutilizar esta implementacion y hacerla de una forma muy generica tal vez, lo mejor seria ponerlo en un proyecto aparte que se podria llamar SessionLogic. Lo que logramos con esto es si queremos usar esta dll nos aseguramos que no nos estamos llevando logicas que no tienen sentido con esta funcionalidad. Hay metricas que nos permiten medir que tan relacionado esta nuestro paquete y podemos detectar que pueden haber clases que no tengan nada que ver con las que si estan relacionadas fuertemente y se tendria que estudiar este caso para poder ver a donde es el mejor lugar de migrarlas.

Como nuestro sistema solamente usa la interfaz y no la implemantacion no estariamos rompiendo con ninguna dependencia porque directamente el sistema ni se entera que sacaste la implementacion de la interfaz A y la pusiste en un paquete X, porque solamente mira la interfaz A.

Aca se hara la implementacion en BusinessLogic para no complicar el codigo.

Entonces deberiamos de crear la clase SessionLogic en BusinssLogic que implemente la interfaz ISessionLogic. Para que se vea la interfaz ISessionLogic en BusinessLogic se necesita la referencia:

> *dotnet add BusinessLogic reference SessionInterface*

Y nos quedaria momentaneamente la clase SessionLogic asi:

```
public class SessionLogic : ISessionLogic
{

  public bool IsCorrectToken(string token)
  {
    return true;
  }
}
```

⚠ La implementacion de la interfaz se dejara libre para que la hagan ustedes.

Forma automatica

Ahora que se desarrollo la forma manual que simplemente era la parte de ir a buscar el servicio al RequestServices. Veamos como seria la forma automatica con DI.

```
public class AuthorizationAttributeFilte : IAuthorizationFilter
{
  private readonly ISessionLogic sessions;

  public AuthorizationAttributeFilter(ISessionLogic sessionsLogic)
  {
    this.sessions = sessionsLogic;
  }

  public void OnAuthorization(AuthorizationFilterContext context)
  {
    var token = context.HttpContext.Request.Headers["Authorization"];

    if(String.IsNullOrEmpty(token))
    {
      context.Result = new ContentResult()
      {
        StatusCode = 401,
        Content = "Mandame un token papo"
      };
    }
    else
    {
      if(!sessions.IsCorrectToken(token))
      {
        context.Result = new ContentResult()
        {
          StatusCode = 403,
          Content = "No tenes permisos mostro"
        };
      }
    }
  }
}
```

Como se ve, el codigo queda un poco mas simple y corto ya que nos encargamos de usar la dependencia y no de ir a buscarla.

De igual manera falta configurarla en el Startup. Seria de la siguiente manera:

```
public class Startup
{
  //...SOME CODE
  public void ConfigureServices(IServiceCollection services)
  {
      services.AddControllers();
      ServiceFactory serviceFactory = new ServiceFactory(services);
      serviceFactory.AddDbContextService();
      serviceFactory.AddCustomServices();
      services.AddScoped<AuthorizationAttributeFilter>();
  }
}
```

La inyeccion de dependencia del filtro tiene que ser despues de que se inyectaron los servicios nuestros porque para inyectar el filtro previamente tiene que estar registrado la sesion.

Por supuesto que ahora tenemos que registrar la ISessionLogic en factory.

En Factory:

```
public class ServiceFactory
{
  //...SOME CODE
  public void AddCustomServices()
  {
    services.AddScoped<IMovieRepository, MovieRepository>();
    services.AddScoped<IMovieLogic, MovieLogic>();
    services.AddScoped<ISessionLogic, SessionLogic>();
  }
}
```

Para poder realizar esta inyeccion nos falta agregar la referencia de SessionInterface a Factory.

> *dotnet add Factory reference SessionInterface*

Listo, ya vimos dos formas de implementar los filtros, veamos como se usarlos.

Usemos el filtro de forma automatica

Este filtro lo vamos a usar en nuestro MovieController para que las peliculas sean accedidas solamente por usuarios que esten autenticados y tengan permisos.

```
[ApiController]
[Route("api/movies")]
[ServiceFilter(typeof(AuthorizationAttributeFilter))]
public class MovieController : ControllerBase
{
  //...SOME CODE
}
```

Pronto con el filtro de la autorizacion. Ahora veamos el de excepciones.

ExceptionFilter

Con este filtro vamos a poder limpiera el sistema de try & catch y generar un manejamiento de errores de forma global. Para esto necesitamos crear la clase ExceptionFilter que estara en la misma carpeta que AuthorizationAttributeFilter.

```
namespace WebApi.Filters
{
  public class ExceptionFilter : IExceptionFilter
  {
    public void OnException(ExceptionContext context)
    {
      try
      {
        throw context.Exception;
      }
      catch(MyConcreteException ex)
      {
        context.Result = new ContentResult()
        {
          StatusCode = 400,
          Content = ex.Message;
        };
      }
      catch(Exception)
      {
        context.Result = new ContentResult()
        {
          StatusCode = 500,
          Content = "Problemas del servidor, que se le va hacer?"
        }
      }
    }
  }
}
```

Lo que hace el codigo es lanzar la excepcion que se detecto y tener diferentes catch para retornar el correcto codigo de error con su mensaje.

Esto no es una mala practica porque yo no se que excepcion estoy lanzando en el try por eso la idea es tener diferentes catch y en el la forma correcta de cortar el circuito de la request en la pipeline.

Seguramente esten pensando que ese metodo va a crecer un monton si hay muchas excepciones y la estan en lo cierto, si el sistema maneja 15 excepciones, todas ellas se manejarian aca.

Se deja como desafio el diseñar la mejora para el caso que se menciono mas arriba utilizando IExceptionFilter.

Una vez creada este filtro lo tenemos que aplicar de forma global, para esto lo tenemos que configuar en el Startup de la siguiente manera:

```
public class Startup
{
```

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

      ServiceFactory serviceFactory = new ServiceFactory(services);

      serviceFactory.AddDbContextService();
      serviceFactory.AddCustomServices();

      services.AddScoped<AuthorizationDIFilter>();
  }
}
```

Pronto, ahora nuestro sistema usa dos filtros de forma global.