

Programación Web .NET Core

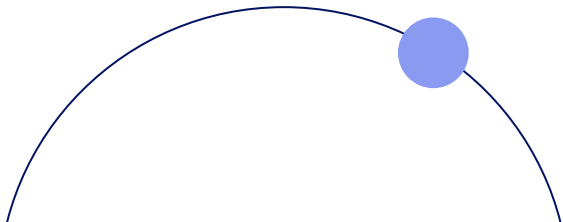
Módulo 5

Escribir filtros de acción

Escribir filtros de acción

- ¿Qué son los filtros de acción?
- Creación y uso de filtros de acción.
- Cómo crear y utilizar filtros de acción (demostración).

En algunas situaciones, es posible que se deba ejecutar un código específico antes o después de que se ejecuten las acciones del controlador. Por ejemplo, antes de que un usuario ejecute cualquier acción que modifique los datos, es posible que desee ejecutar un código que verifique detalles de la cuenta de usuario. Si agregas dicho código a las acciones en sí, tendrás que duplicarlo en todas las acciones en las que desees que se ejecute. Los filtros de acción proporcionan una forma conveniente de evitar esta duplicación.

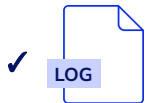


¿Qué son los filtros de acción?

- Los filtros son clases de MVC que ejecutan lógica de filtrado antes o después de llamar al método de acción.



Almacenamiento
en caché



Registro



Autorización

- Son atributos declarativos que se agregan a los métodos de acción.

Los filtros son clases de **MVC** que se pueden utilizar para gestionar diferentes intereses en tu aplicación web, como la autorización a los recursos del sistema. Puedes aplicar un filtro de acción en un controlador anotando el atributo apropiado en el método de acción.

También puedes aplicar un filtro a cada acción anotando el atributo en la clase del controlador.

Los filtros se ejecutan después de que se elige una acción y dentro de una tubería de filtro. La etapa en la tubería de filtrado está determinada por el tipo de filtro:

- **Filtros de autorización:** Se ejecutan antes que cualquier otro filtro y antes del código en el método de acción. Se utilizan para comprobar los derechos de acceso de un usuario a la acción.
- **Filtros de recursos:** Aparecen en la canalización de filtros después de los filtros de autorización y antes de cualquier otro filtro. Se pueden utilizar por motivos de rendimiento. Implementan **InterfazIResourceFilter** o **IAsyncResourceFilter**.
- **Filtros de acción:** Se ejecutan antes y después del código en el método de acción. Puedes usarlos para manipular los parámetros que recibe una acción y para manipular el resultado devuelto por una acción. Implementan la interfaz **IActionFilter** o la interfaz **IAsyncActionFilter**.
- **Filtros de excepción:** Se ejecutan sólo si un método de acción u otro filtro arrojan una excepción. Se utilizan para manejar errores. Los filtros de excepción implementan la **interfazIExceptionFilter** o la **interfazIAsyncExceptionFilter**.

- **Filtros de resultados:** Se ejecutan antes y después de ejecutar el resultado de una acción. Implementan la interfaz **IResultFilter** o la interfaz **IAsyncResultFilter**.

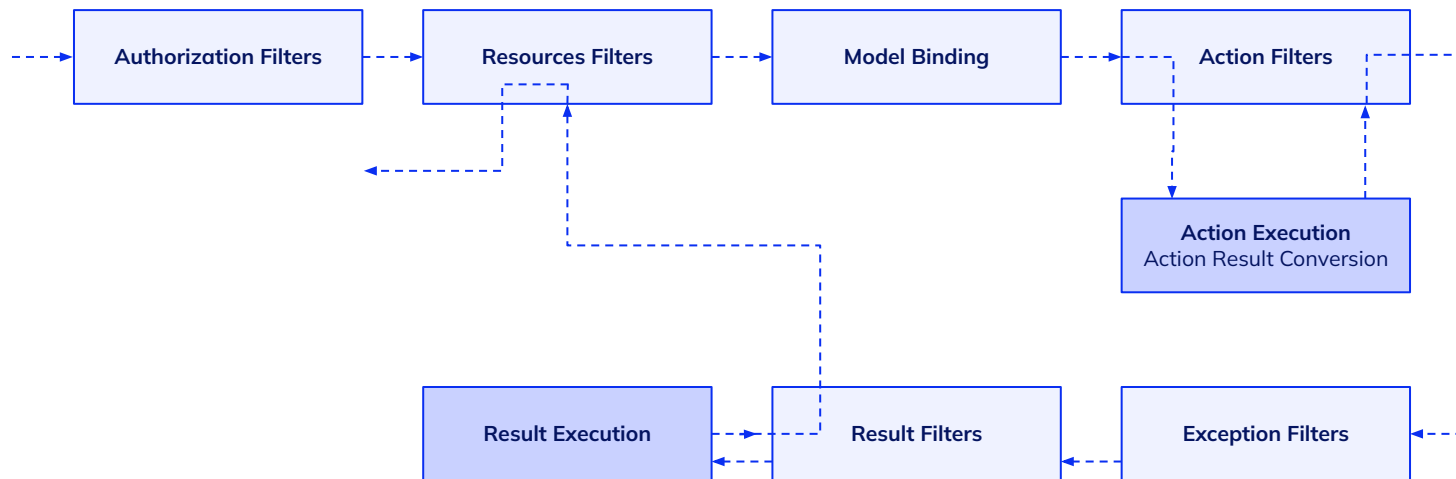
Puedes crear tus propias clases de filtros o utilizar las ya existentes. Por ejemplo:

- **ResponseCacheAttribute.** Anotar una acción con el atributo **ResponseCache** almacenará en caché la respuesta al cliente. El atributo **ResponseCache** contiene varias propiedades. Uno de ellos es la propiedad **Duration**, que obtiene o establece la duración, en segundos, para la que se almacena la respuesta en caché.

- **AllowAnonymousAttribute.** Anotar una acción con el atributo **AllowAnonymous** permitirá a los usuarios acceder a una acción sin iniciar sesión.
- **ValidateAntiForgeryToken y TokenAttribute.** Anotar una acción con estos atributos ayudará a prevenir ataques de falsificación de solicitudes entre sitios.



Tipos de filtros



Creación y uso de filtros de acción

```
public class SimpleActionFilter : ActionFilterAttribute {  
    public override void OnActionExecuting  
(ActionExecutingContextfilterContext)  
    {  
        Debug.WriteLine("This Event Fired: OnActionExecuting");  
    }  
    public override void OnActionExecuted  
(ActionExecutedContextfilterContext)  
    {  
        Debug.WriteLine("This Event Fired: OnActionExecuted");  
    }  
}
```

Puedes crear un filtro de acción personalizado o un filtro de resultados personalizado. También filtros personalizados mediante la implementación de la interfaz **IActionFilter** o la interfaz **IResultFilter**. Sin embargo, la clase base **ActionFilterAttribute** implementa interfaces **IActionFilter** como el **IResultFilter**.

Al derivar su filtro de la clase base **ActionFilterAttribute**, puedes crear un filtro único que puede ejecutar código tanto antes como después de ejecutar una acción, y tanto antes como después se devuelve el resultado.

El siguiente código muestra cómo se usa un filtro de acción para escribir texto en la ventana de salida de Visual Studio:

Filtros de acción personalizado

```
public class SimpleActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContextfilterContext)
    {
        Debug.WriteLine("This Event Fired: OnActionExecuting");
    }
    public override void OnActionExecuted(ActionExecutedContextfilterContext)
    {
        Debug.WriteLine("This Event Fired: OnActionExecuted");
    }
    public override void OnResultExecuting(ResultExecutingContextfilterContext)
    {

```

...

```
Debug.WriteLine("This Event Fired: OnResultExecuting");  
}  
public override void OnResultExecuted(ResultExecutedContextfilterContext)  
{ Debug.WriteLine("This Event Fired: OnResultExecuted");  
}  
}
```

Después de crear un filtro de acción personalizado, puedes aplicarlo a cualquier método de acción o controlador de tu sitio web.



Decorar el método o la clase controlador con el nombre del filtro de acción.

En las siguientes líneas de código, el atributo **SimpleActionFilter** se aplica a la acción Index del **ControladorMyController**:

Usar un filtro de acción personalizado

```
public class MyController : Controller
{
    [SimpleActionFilter]
    public IActionResult Index()
    {
        return Content("some text");
    }
}
```

Usar un filtro de acción personalizado

Los manejadores de eventos:

OnActionExecuting, **OnActionExecuted**, **OnResultExecuting** y **OnResultExecuted** toman un objeto de contexto como parámetro. El objeto de contexto hereda de la clase **FilterContext**.

La clase **FilterContext** contiene propiedades que puede utilizar para obtener información adicional sobre la acción y el resultado. Por ejemplo, puedes recuperar el nombre de la acción utilizando la propiedad **ActionDescriptor.RouteValues**.

El siguiente código muestra cómo se puede recuperar el nombre de la acción en **OnActionExecuting** y en **OnActionExecuted**:



Los controladores de eventos

OnActionExecuting y OnActionExecuted

```
public class SimpleActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        string actionName = filterContext.ActionDescriptor.RouteValues["action"];
        Debug.WriteLine(actionName + " started");
    }
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        string actionName = filterContext.ActionDescriptor.RouteValues["action"];
        Debug.WriteLine(actionName + " finished");
    }
}
```

Puedes utilizar el parámetro **ResultExecutedContext** para obtener el contenido que se devolverá al navegador.

El siguiente código muestra cómo el contenido devuelto al navegador se puede recuperar en el controlador de eventos **OnResultExecuted**:

Un filtro de acción personalizado simple

```
public class SimpleActionFilter : ActionFilterAttribute
{
    public override void OnResultExecuted(ResultExecutedContext filterContext)
    {
        ContentResult result = (ContentResult)filterContext.Result;
        Debug.WriteLine("Result: " + result.Content);
    }
}
```

El siguiente código muestra cómo se puede cambiar el contenido de la clase **SimpleActionFilter** para que escriba la salida en un archivo de registro que se encuentra en **c:\logs\log.txt**:



Escribir en un archivo de registro

```
public class SimpleActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        string actionName = filterContext.ActionDescriptor.RouteValues["action"];
        using (FileStream fs = new FileStream("c:\\logs\\log.txt", FileMode.Create))
        {
            using (StreamWriter sw = new StreamWriter(fs))
            {
```



...

```
sw.WriteLine(actionName + " started");
    }
}
}
public override void OnActionExecuted(ActionExecutedContextfilterContext)
{
stringactionName = filterContext.ActionDescriptor.RouteValues["action"];
    using (FileStream fs = new FileStream("c:\\logs\\log.txt", FileMode.Append))
    {
        using (StreamWritersw = new StreamWriter(fs))
        {
sw.WriteLine(actionName + " finished");
        }}}
    public override void OnResultExecuting(ResultExecutingContextfilterContext)
    {
        using (FileStream fs = new FileStream("c:\\logs\\log.txt", FileMode.Append))
        {
```

...

...

```
        using (StreamWritersw = new StreamWriter(fs))
        {
            sw.WriteLine("OnResultExecuting");
        }
    }
    public override void OnResultExecuted(ResultExecutedContextfilterContext)
    {
        ContentResult result = (ContentResult)filterContext.Result;
        using (FileStream fs = new FileStream("c:\\logs\\log.txt", FileMode.Append))
        {
            using (StreamWritersw = new StreamWriter(fs))
            {
                sw.WriteLine("Result: " + result.Content);
            }
        }
    }
}
```


Cuando se realice la navegación en la acción Index en el controlador **MyController**, se creará un archivo llamado **log.txt**. El siguiente código muestra el contenido del archivo de registro **c:\logs\log.txt**:

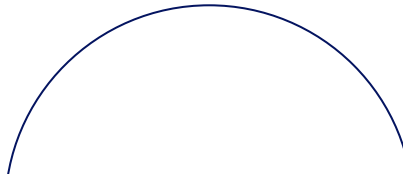
Contenido del archivo de registro:

```
Index started  
Index finished  
OnResultExecuting  
Result: some text
```

Uso de la inyección de dependencia en filtros

Hasta ahora, presentamos una forma de agregar filtros: por instancia. Tales filtros no pueden obtener dependencias utilizando su constructor proporcionado por la inyección de dependencia.

En caso de que necesites tus filtros para obtener dependencias con tu constructor, puedes agregarlos por tipo mediante el atributo **ServiceFilter**.



Por ejemplo, supongamos que tu filtro necesita recibir en tu constructor una instancia de **IHostingEnvironment**:

Utilización de la inyección de dependencia en un filtro

```
public class LogActionFilter : ActionFilterAttribute
{
    private IHostingEnvironment _environment;
    public LogActionFilter(IHostingEnvironment
environment)
    {
    }
}
```

Para aplicar el filtro **LogActionFilter** a una acción denominada Index, puede utilizar el atributo **ServiceFilter**:

Aplicar el filtro LogActionFilter a una acción

```
public class CityController: Controller
{
    [ServiceFilter(typeof(LogActionFilter))]
    public IActionResult Index()
    {
    }
}
```

**¡Sigamos
trabajando!**