Patrón Options (Options pattern) IOptions

Configuración de aplicaciones .NET Core, .NET 5, .NET 6 +

IOptions, IOptionsMonitor, IOptionsSnapshot, IOptionsMonitorCache

NAG 2021.10

Introducción

El llamado 'patrón Options' (*opciones* en castellano) forma parte del sistema de configuración de aplicaciones y componentes .NET Core, .NET 5 +.

Permite el uso de **clases definidas por el usuario para proporcionar objetos de configuración con tipos estrictos**. Es ampliamente empleado en muchos (casi todos) los componentes internos y muchos componentes de terceros existentes en las aplicaciones .NET Core (incluyendo ASP.NET Core) y .NET 5 + por lo que es un patrón bastante maduro y profusamente empleado.

Es recomendable (pero no imprescindible u obligatorio) que las aplicaciones o componentes que diseñemos y construyamos para estas plataformas, y que requieran objetos de configuración, accedan a sus configuraciones mediante el empleo de los patrones Options, según se verá en el artículo.

Existen bastantes maneras de emplear los diferentes aspectos de los patrones Options, por lo que a veces puede resultar un poco complicado saber cuáles debemos emplear o incluso en qué orden.

En este artículo veremos el uso básico de este patrón y se mostrará de una manera lo más sencilla posible cómo se puede emplear o cómo se debería emplear.

También se explica en detalle cómo se deberá **configurar una aplicación de consola o de escritorio (Windows Forms o WPF) .NET 5 o superior** (no una aplicación ASP.NET Core) para el uso del 'patrón Options'.

En una aplicación web ASP.NET Core, la forma de hacerlo puede ser algo diferente, principalmente en cuanto al orden y el lugar de hacer las cosas y quizás más aún en las aplicaciones ASP.NET 6, aunque básicamente el funcionamiento será el mismo. Normalmente en una aplicación web, todas las referencias necesarias para emplear el 'patrón Options' (o casi todas) estarán ya especificadas, por lo que su implementación será aún más sencilla.

Ventajas (y desventajas)

Como muchos de los patrones existentes para cualquier tipo de procedimiento de aplicación, éste tiene sus ventajas y sus desventajas. En la mayor parte de los casos las ventajas son suficientes para adoptar este patrón para gestionar nuestra configuración (o configuraciones) sin mayor problema.

Ventajas

Es un **patrón estándar**, conocido por la mayor parte de la comunidad de programadores de .NET Core y empleado profusamente dentro de las aplicaciones .NET Core, luego es **fácilmente reconocible** y modificable por cualquiera. (Consideraciones de mantenibilidad)

Se emplea **a través de un proveedor de servicios** con inyección de dependencias, lo que lo hace fácilmente empleable en cualquier clase o en cualquier punto / momento de nuestra aplicación.

Trabaja con **tipos estrictos** lo que lo hace mucho más sencillo de emplear que una gestión estándar / básica del sistema de configuración estándar .NET Core.

Permite emplear acciones de configuración y 'post configuración' para ajustar el objeto de configuración a nuestras necesidades concretas de una forma bastante sencilla.

Permite especificar diferentes tipos de **validaciones** que se realizarán antes de suministrar el objeto de configuración al método que lo solicita, por lo que se garantiza que la configuración cumpla con los requisitos necesarios para la aplicación.

Permite el uso de **varias configuraciones con el mismo tipo** (diferente nombre) simultáneamente, lo que puede representar una gran ayuda en algunos casos avanzados.

Como cualquier objeto llenado o completado desde un origen de configuración estándar (IConfiguration) resulta **muy flexible** que las propiedades o sub-elementos se puedan leer desde cualquier proveedor de configuración o desde varios simultáneamente (varios archivos o proveedores del mismo tipo o varios proveedores de configuración de diferente tipo).

Contras

Todas las acciones de configuración sirven únicamente para modificar el contenido de un objeto creado internamente por la infraestructura (patrón de factoría). Esto implica que no se podría suplantar (cambiar) el objeto (raíz) creado por la infraestructura por otro que creemos nosotros en una acción (aunque sí podríamos reemplazar completamente algún o algunos sub-elementos de ese objeto en el caso de un grafo de objetos complejo). (Ver uso avanzado para solucionar este problema).

No es fácilmente adaptable – aunque sí se podría hacer con un poco de trabajo – para un objeto de configuración existente y que se deserializa desde un almacenamiento permanente mediante un procedimiento no contemplado por el patrón, como puede ser la creación mediante deserialización desde un archivo en formato Xml o Json.

Para grafos de objetos muy grandes puede resultar relativamente lento en la carga inicial, aunque el empleo de cachés lo hacen perfectamente válido para un uso repetido de un objeto de configuración.

Uso recomendado, más sencillo o más habitual

El tipo de datos de configuración XxxOptions

El tipo de datos (clase) de configuración para nuestra aplicación o componente será una clase con propiedades públicas (lectura y escritura), que pueden ser de tipo simple o de tipo complejo (grafo de objetos), y que contendrá todas las informaciones necesarias u opcionales para el funcionamiento de nuestra aplicación o componente.

El nombre de esta clase puede ser cualquiera, pero para hacerlo más reconocible es habitual que le añadamos el sufijo Options, de esta manera, en los **ejemplos que se emplearán en el artículo** emplearemos una clase ***AppOptions*** que será la que necesitemos para configurar nuestra aplicación.

La clase **deberá tener un constructor público sin parámetros** para que pueda ser construido por la infraestructura del patrón Options. El objeto de configuración se creará (lo creará la infraestructura del patrón options), por defecto, mediante el método Activator.CreateInstance().

Normalmente será un tipo público, pero podría ser un tipo privado si la configuración estuviera en el proyecto en el que se va a emplear y además no se fuera a emplear fuera de este proyecto (por ejemplo, en el proyecto principal de la aplicación).

En la mayor parte de los casos, los valores de las propiedades de la clase los obtendremos, en tiempo de ejecución, desde un origen de configuración (proveedor de configuración), por ejemplo, un archivo json o una variable de entorno o una mezcla de todos ellos.

La clase 'options' puede tener propiedades no serializables (que no se pueden representar en un archivo json) y que igual hay que crear y suministrar programáticamente en tiempo de ejecución porque, por ejemplo, dependen de un servicio que solamente se puede obtener a través del proveedor de servicios, una vez que éste haya sido construido, o porque dependen de otro objeto de configuración Options.

Para ser más realistas (y para no tener que reescribir el mismo código varias veces), se ha definido, en los ejemplos disponibles con este artículo, el tipo de configuración ***AppOptions*** en un ensamblado separado de la aplicación principal, como suele ser habitual en la mayoría de las aplicaciones. De esta manera el tipo AppOptions está definido en el ensamblado (TiposComunes).

Primeros pasos

La forma más sencilla de emplear este patrón sería **registrar cada objeto de configuración** que vayamos a emplear en el sistema de **proveedor de servicios**, normalmente en el arranque de aplicación, mediante el método con un tipo genérico

```
var optionsBuilder = coleccionDescriptores.AddOptions<TipoOpciones>();
```

Este método retorna una referencia a un objeto del tipo OptionsBuilder<TipoOpciones> que nos permitirá registrar todas las acciones de todos los tipos posibles (configuraciones, post configuraciones y validaciones) que necesitemos para nuestro objeto de configuración.

En el caso mostrado arriba, suponemos que solamente habrá un único objeto del tipo TipoOpciones en nuestra aplicación. (Ver múltiples objetos de configuración más abajo).

> **Nota**: Esta no es la única manera de realizar el registro, como veremos más abajo, pero será muy eficaz y probablemente la más clara de hacerlo.

Referencias necesarias

Para emplear el patrón Options necesitaremos en nuestra aplicación o en los ensamblados que deseen emplear el/los objetos de configuración (por lo menos), **referencias a los paquetes NuGet**: '**Microsoft.Extensions.DependencyInjection**' y '**Microsoft.Extensions.Options**' (las versiones adecuadas).

En nuestro caso tendríamos en nuestra **aplicación de consola de ejemplo** lo siguiente:

```
<ItemGroup>
        <PackageReference Include="Microsoft.Extensions.DependencyInjection" Version="5.0.2" />
        <PackageReference Include="Microsoft.Extensions.Options" Version="5.0.0" />
</ItemGroup>
```

Una vez registrado el 'patrón Options' con todos sus interfaces y objetos necesarios (método AddOptions) en el proveedor de servicios y construido éste, podremos obtener un objeto de *TipoOpciones* , en dos pasos, mediante una llamada explícita al proveedor de servicios (GetService()) o mediante el empleo de un parámetro en el constructor de una clase que se obtenga/construya mediante el proveedor de servicios (inyección de dependencias).

Por ejemplo (caso de obtención explícita de un objeto):

```
var options = sp.GetService<IOptions<AppOptions>>();
var objOptions = options.Value;
```

El ejemplo completo podría ser (en el método Main de nuestra aplicación de consola):

```
        var serviceDescriptors = new ServiceCollection();

        // Línea imprescindible para emplear el patrón Options
        serviceDescriptors.AddOptions<AppOptions>();

        var sp = serviceDescriptors.BuildServiceProvider();

        // ******************
        // Obtenemos el objeto que implementa IOptions de forma explícita (GetService)
        var options = sp.GetService<IOptions<AppOptions>>();
        // Obtenemos el objeto de opciones,
        // mediante la propiedad Value del objeto que implementa el interfaz IOptions
        var objOptions = options.Value;
```

En nuestro ejemplo super simplificado, no estamos empleando el valor devuelto por el método AddOptions<>, por lo que sería exactamente igual emplear la versión no genérica de este método (AddOptions() - que **realmente es el método imprescindible para emplear el patrón Options**), que no retorna un OptionsBuilder.

El método no genérico AddOptions() es el encargado de registrar todos los interfaces (con sus tipos de implementación por defecto) para emplear el 'patrón Options' en sus diferentes versiones. La versión genérica del método AddOptions (AddOptions<TipoOpciones>) internamente llamará al método AddOptions(), que registrará todos los componentes del patrón si es necesario.

Entre otras cosas, el método AddOptions() registrará en el proveedor de servicios los interfaces genéricos IOptions<>, IOptionsMonitor<> y IOptionsSnapshot<> con sus correspondientes tipos de implementación por defecto. De esta manera, aunque no registremos nada más en la aplicación, se podrá obtener una referencia a (por ejemplo) un objeto IOptions<Xyz>, siendo Xyz un nombre de una clase adecuada para el patrón, como se ha comentado anteriormente. Por supuesto, el objeto retornado en este caso estará tal y como lo haya creado el constructor por defecto de la clase.

Avanzando un poco

Viendo lo anterior, parece mucho esfuerzo y mucho código para obtener una instancia de un objeto como si lo hubiéramos construido con un new. Realmente todo el código anterior sería muy parecido a esta única línea:

var objOptions = new AppOptions();

> **Nota**: Es importante hacer notar que el objeto de opciones obtenido (AppOptions) **no ha sido creado por el proveedor de servicios**, como se ha comentado anteriormente, luego no estará controlado por éste y su vida dependerá exclusivamente del uso que haga del objeto la aplicación. No es muy normal que un tipo de configuración requiera implementar el interfaz IDisposable, pero en el caso de necesitarlo, la aplicación debería invocar el método Dispose de forma explícita cuando ya no haga falta el objeto obtenido o empleando un bloque o sentencia ***using***.

Si intentamos obtener dos veces (suponemos que en diferentes partes de la aplicación) una instancia del objeto de configuración AppOptions, podremos ver que **el objeto se ha mantenido en un caché** y no necesita construirse más. (Se pude poner una traza – mostrar un mensaje – en el constructor del tipo de opciones AppOptions para comprobar que solamente se construye una única instancia de este tipo.)

Por supuesto, en el caso mostrado anteriormente, las propiedades del objeto AppOptions tendrán los valores por defecto que estén definidos en el propio tipo, ya sea en el constructor o explícitamente en sus propiedades.

Acciones de configuración (post configuración y validación)

El 'patrón Options' permite definir un número indeterminado de acciones de configuración del objeto de opciones, creado por defecto, de varias maneras.

> Ejemplos en el proyecto 'PatronOptions2'
>
> **Nota importante**: Las acciones de configuración (incluyendo las acciones de post configuración y las de validación) **se ejecutarán <u>todas</u> antes de suministrar el objeto** de configuración al código que lo está solicitando. Las acciones de configuración **se ejecutarán cuando el proveedor de servicios esté ya construido** por lo que se podría emplear éste para obtener objetos desde el proveedor de servicios (ver ejemplo un poco más abajo).

La forma más sencilla de registrar estas acciones será empleando el objeto OptionsBuilder retornado por el método AddOptions<TipoOpciones>.

El objeto OptionsBuilder tiene una serie de métodos que permiten definir (mediante un delegado del tipo Action) una serie de acciones de configuración, por ejemplo, con el método **Configure**.

En el ejemplo hemos definido las acciones de configuración con una *expresión Lambda*, pero perfectamente se podrían haber definido en un método externo (método con nombre) con la signatura correcta.

El prototipo de la **versión más básica** del método Configure es la siguiente:

```
public virtual OptionsBuilder<TOptions> Configure(Action<TOptions> configureOptions);
```

y especifica que se requiere pasar al método Configure como argumento un delegado de tipo Action, que perfectamente podría ser un método con nombre y no una expresión Lambda.

Ejemplo de código (proyecto 'PatronOptions2'):

```
        var optionsBuilder = serviceDescriptors.AddOptions<AppOptions>();
        optionsBuilder.Configure(opts =>
        {
                Console.WriteLine($"1. Asignando nuevos valores a propiedades del objeto de
opciones {opts.GetType()}");
                opts.PropiedadInt = 1;
                opts.PropiedadString = "Hola mundo";
        });
        optionsBuilder.Configure<IServiceProvider>((opts, sp) =>
        {
                Console.WriteLine($"2. Asignando otros valores a propiedades del objeto de
opciones {opts.GetType()}");
                opts.PropiedadInt = 2;
                opts.PropiedadString = "Nuevo valor";
        });
```

Las dos acciones de configuración se ejecutarán en el orden en el que hayan sido registradas, en nuestro caso se podrá ver en la salida de la aplicación (consola) que primero se ejecuta la acción 1 y posteriormente la 2.

La segunda acción de configuración registrada, especifica que **se deberá inyectar al método de configuración (además del propio objeto de tipo TipoAcciones) un objeto de un tipo concreto, que se deberá obtener del proveedor de servicios**. En nuestro caso (como opción más sencilla y como ejemplo) se ha inyectado un objeto que implemente IServiceProvider (el propio proveedor de servicios). En este caso el prototipo de la expresión Lambda deberá reflejar que empleará 2 parámetros como se ve en el código, el primer parámetro siempre será el objeto de tipo Options creado y el resto serán los parámetros genéricos especificados en la llamada al método. (Hay versiones del método Configure de hasta 5 parámetros genéricos).

Las acciones de configuración se ejecutarán (depende de la opción del patrón Options que empleemos, pero normalmente) **la primera vez que obtengamos el objeto** de tipo Options solicitado. (por ejemplo con la propiedad options.Value como se ha visto anteriormente) **y no antes**.

Leer contenido de objeto de opciones de la configuración (IConfiguration)

Probablemente asignar valores por defecto a las propiedades de un objeto Options no parezca muy interesante tal como se ha mostrado y sea más adecuado asignar estos valores en el constructor.

La forma más habitual de asignar valores a las propiedades de un objeto Options suele ser leyéndolas desde un origen de configuración (IConfiguration) y más probablemente desde un archivo de configuración de la aplicación, como puede ser el archivo 'appsettings.json' o similares.

Para esto, nuestra aplicación deberá estar empleando el modelo de configuración estándar de las aplicaciones .NET Core.

Ejemplos en el proyecto 'PatronOptions3'

Para ello, añadiremos a la aplicación, si no lo tenemos ya, una referencia a los paquetes NuGet '**Microsoft.Extensions.Configuration**', '**Microsoft.Extensions.Configuration.Json**' y '**Microsoft.Extensions.Configuration.Binder**' (si necesitáramos más orígenes de configuración que los especificados, se deberían añadir sus referencias, por ejemplo, si necesitamos leer la configuración desde las variables de entorno o de los parámetros de entrada a la aplicación).

En el arranque de la aplicación construiremos un objeto ConfigurationBuilder, añadiremos los proveedores necesarios y construiremos el sistema de configuración.

Por ejemplo:

```
var cfgBuilder = new ConfigurationBuilder();
cfgBuilder.AddJsonFile("appsettings.json", false);
var cfgRoot = cfgBuilder.Build();
```

Obtendremos la **sección de configuración** correspondiente a la configuración de nuestro tipo de Options

```
var section = cfgRoot.GetSection(nameof(AppOptions));
```

El nombre de la sección de configuración puede ser cualquiera, pero lo más habitual es que coincida con el nombre del tipo de opciones, de ahí emplear el operador *nameof* para obtener el nombre del tipo a partir del nodo raíz de la configuración (objeto que implementa el interfaz IConfigurationRoot).

Una vez obtenida la sección de configuración (hay que tener en cuenta que, aunque la sección no exista en ningún proveedor, el método GetSection(…) no producirá un error en ningún caso), en una acción de configuración podremos obtener la configuración mediante el método de extensión Bind (). (El método Bind() está definido en el componente '**Microsoft.Extensions.Configuration.Binder**')

```
var optionsBuilder = serviceDescriptors.AddOptions<AppOptions>();
optionsBuilder.Configure(opts =>
{
        Console.WriteLine($"1. Leyendo datos del origen de configuración");
        section.Bind(opts);
});
```

El método anónimo definido mediante una *expresión Lambda* **captura la variable** que contendrá la sección de configuración requerida e invocará al método Bind() para **completar** / leer el objeto de opciones suministrado, desde la sección de configuración. Como se puede deducir por la signatura del método Bind(), éste no crea el objeto de configuración, sino que **asigna valores** leídos desde la configuración a las propiedades del objeto suministrado. Si en la configuración no existen todas las propiedades del objeto, solamente se asignarán aquellas que tengan valores en la configuración y el resto mantendrá los valores por defecto o los que se hayan asignado en acciones anteriores.

Si la sección de configuración no estuviera definida en ningún proveedor de configuración, este método no tendría ningún efecto.

Si hemos añadido a la aplicación un archivo 'appsettings.json' con el siguiente contenido

```
/*
        appsettings.json
*/
{
        "AppOptions": {"PropiedadInt": 345, "PropiedadString": "Un valor cualquiera"}
}
```

y ejecutamos esta versión modificada de la aplicación, el objeto de opciones AppOptions que obtendremos tendrá los valores especificados en el archivo en las correspondientes variables.

Versión mejorada (y más adecuada en casi todas las circunstancias)

Esta forma de leer los valores de un objeto de configuración es muy básica (aunque eficaz y válida en muchos casos) pero existe una versión muy mejorada empleando un método de extensión del objeto OptionsBuilder.

Ejemplos en el proyecto 'PatronOptions4'

Tendremos que añadir a nuestra aplicación una referencia al paquete NuGet '**Microsoft.Extensions.Options.ConfigurationExtensions**'.

Emplearemos ahora el método de extensión Bind() del tipo OptionsBuilder para hacer más o menos lo que hemos hecho en la versión anterior, pero con una sintaxis simplificada y añadiendo algunas sofisticaciones necesarias en algunos casos y que veremos posteriormente.

```
optionsBuilder.Bind(section);
```

El método Bind() del tipo OptionsBuilder añade una acción de configuración (como en el caso anterior) para leer los datos desde el origen de configuración, más una serie de tareas complementarias. La acción se ejecutará en el orden en el que esté registrada con respecto a otras acciones de configuración.

Qué más se puede hacer en una acción de configuración de Options

Como hemos visto anteriormente, una acción de configuración definida mediante una expresión Lambda puede capturar fácilmente una variable externa y emplearla dentro del método para diferentes labores. (Esto mismo se podría hacer con un método de un objeto creado exprofeso y que contuviera el valor de la variable, pero es mucho más sencillo – en principio – realizarlo mediante una expresión Lambda).

Si por ejemplo quisiéramos modificar el valor de una propiedad de nuestro objeto de configuración, si se ha especificado un valor diferente de otra manera que no sea de la forma habitual – caso habitual de la ruta de un archivo que podría venir en la sección de configuración estándar o en otro origen – podremos asignar este nuevo valor, en una acción de configuración **posterior** a la acción de lectura de las propiedades del objeto desde una sección de configuración (hay que tener en cuenta que, como se ha comentado anteriormente el orden es importante).

Ejemplo (extracto de código del proyecto 'PatronOptions5'):

```
var optionsBuilder = serviceDescriptors.AddOptions<AppOptions>();
optionsBuilder.Bind(section);
if (!string.IsNullOrEmpty(strValor))
{
            optionsBuilder.Configure(opts =>
            {

            Console.WriteLine($"Asignando valor capturado a la propiedad de tipo String '{strValor}'");
                        opts.PropiedadString = strValor;
            });
}
```

En el ejemplo anterior, suponemos que hemos obtenido un valor alternativo (quizás no especificado en el archivo de configuración o que queremos reemplazar) y lo tenemos en la variable ***strValor***, si este valor no es nulo o vacío, registramos una acción de configuración para asignar este valor cuando se obtenga el objeto. La acción de configuración en este caso, debe ser posterior a la llamada al método Bind(), como se ve en el ejemplo.

Post configuraciones

Una acción de post configuración es exactamente igual a una acción de configuración, con la diferencia de que **independientemente de dónde se haya registrado** – con respecto a las acciones de configuración – , se ejecutará posteriormente a todas las acciones de configuración.

Si hay varias acciones de post configuración registradas para ese tipo de configuración, las acciones de post configuración se ejecutarán en el orden en el que hayan sido registradas, pero **siempre después de todas las acciones de configuración**.

Normalmente una acción de post configuración se definirá para ser ejecutada asegurándonos de que todas las acciones de configuración hayan sido ejecutadas previamente.

Proyecto de ejemplo 'PatronOptions6'

Las acciones de post configuración se pueden registrar y definir exactamente igual que las de configuración, pero empleando en lugar del método Configure el método **PostConfigure**.

```
optionsBuilder.PostConfigure(opts =>
{
            Console.WriteLine("Ejecutando acción de post configuración");
});
```

En el código de ejemplo que se muestra a continuación, la acción de post configuración se ha registrado y definido como la primera acción para nuestro tipo de opciones (normalmente, si puede ser, se añadirá, para mayor claridad, al final de la lista de registro de acciones), para que se vea que se ejecutará – a pesar de su posición – posteriormente a todas las acciones de configuración.

```
var optionsBuilder = serviceDescriptors.AddOptions<AppOptions>();
optionsBuilder.PostConfigure(opts =>
{
            Console.WriteLine("Ejecutando acción de post configuración");
});
optionsBuilder.Bind(section);
if (!string.IsNullOrEmpty(strValor))
{
            optionsBuilder.Configure(opts =>
            {

            Console.WriteLine($"Asignando valor capturado a la propiedad de tipo String '{strValor}'");
                        opts.PropiedadString = strValor;
            });
}
optionsBuilder.Configure(opts =>
{
            Console.WriteLine("Ejecutando acción de configuración");
});
```

Validaciones

Al igual que las acciones de configuración y post configuración se pueden registrar para un tipo de opciones una o varias acciones de validación.

Las acciones de validación **se ejecutarán posteriormente a todas las acciones de configuración y post configuración** y (al igual que éstas) en el orden en el que se hayan registrado.

Para registrar una acción de validación emplearemos el método del tipo OptionsBuilder Validate(), que en su versión más sencilla requiere un (único) parámetro de tipo (**Func<TOptions, bool> validation**, esto es, una función (un delegado de tipo Func) que toma como parámetro de entrada una referencia a un objeto de tipo TOptions y retornará un valor bool) para realizar la validación de un objeto de tipo TipoOptions.

Si la validación del objeto es correcta, el método deberá retornar ***true***.

Si la validación es incorrecta, el método retornará ***false*** o **podría producir una excepción** con un texto / mensaje para indicar qué es lo que ha fallado en la validación. En cualquiera de los dos casos de validación negativa, el llamador **obtendrá una excepción** cuando intente acceder al objeto de opciones (por ejemplo, con la propiedad Value del objeto que implementa IOptions). En el primer caso, la excepción tendrá un texto genérico indicando que la validación no se ha superado ('Ha ocurrido un error de validación') y en el segundo caso, el mensaje (y el tipo) de la excepción será el que hayamos generado nosotros en la acción de validación.

Ejemplo:

```
// post configuraciones
optionsBuilder.PostConfigure(opts =>
{
            Console.WriteLine("Ejecutando acción de post configuración");
});
// validaciones
optionsBuilder.Validate((opts) =>
{
            Console.WriteLine("Ejecutando acción de validación 1");
            return true;
});
optionsBuilder.Validate((opts) =>
{
            Console.WriteLine("Ejecutando acción de validación 2");
            if (string.IsNullOrEmpty( opts.PropiedadString))
            {
                        throw new Exception("No se ha superado la validación por este motivo ...");
            }
            if (opts.PropiedadInt > 1000)
            {
                        return false;
            }
            return true;
});
```

En el ejemplo anterior, hemos añadido en el registro del tipo de opciones dos acciones de validación. La primera de ellas no comprueba nada y retorna ***true*** para indicar que la validación se ha superado (simplemente como ejemplo).

La segunda acción comprueba que la propiedad String tenga un valor que no sea nulo o vacío, en cuyo caso producirá una excepción con un texto definido por nosotros.

También se comprueba en esta acción que el valor de la propiedad Int no sea superior a 1000. En caso contrario se retornará ***false*** para indicar que la validación no se ha superado.

En el ejemplo se pueden probar estos casos modificando los valores en el archivo de configuración 'appsettings.json'.

Al igual que en las acciones de configuración y post configuración (como se ha mostrado anteriormente), podemos emplear las variantes del método Validate que toman más parámetros y obtener éstos del proveedor de servicios. Para ello emplearemos las versiones que toman argumentos genéricos para indicar el tipo (o los tipos) de parámetro que necesitamos y modificaremos la expresión Lambda (o suministraremos una referencia al método adecuado) para contemplar el número necesarios de argumentos.

Por ejemplo:

```
// validaciones
optionsBuilder.Validate<IConfiguration>((opts, cfg) =>
{
          Console.WriteLine("Ejecutando acción de validación 1");
          return true;
});
```

En el ejemplo anterior se inyectará en el método de acción de validación (como segundo parámetro) un parámetro de tipo IConfiguration.

En una aplicación ASP.NET Core el objeto raíz de configuración de la aplicación se registra automáticamente en el proveedor de servicios, no así en una aplicación básica de consola o de escritorio. En estos casos deberemos **registrar expresamente el objeto de configuración raíz** obtenido al comienzo de la aplicación con el proveedor de servicios, por ejemplo, de la siguiente manera:

```
var cfgRoot = cfgBuilder.Build();

. . .

var serviceDescriptors = new ServiceCollection();

serviceDescriptors.AddSingleton<IConfiguration>(cfgRoot);
```

Validaciones mediante atributos

Se pueden especificar, en nuestro tipo de opciones empleado en el patrón Options, atributos de validación en las propiedades del tipo.

Los atributos de validación que se pueden emplear (aparte de los que podamos implementar nosotros) están definidos en el componente (paquete NuGet) '**System.ComponentModel.Annotations**' que deberemos añadir al proyecto que contenga el tipo de opciones.

Existen una serie de atributos para controlar que el contenido de una propiedad sea válido según nuestras reglas de negocio. Se puede especificar más de un atributo de validación a una propiedad de nuestro tipo. El atributo de validación puede especificar un mensaje de error que se empleará para producir la excepción cuando no se cumpla la condición especificada en el atributo.

Algunos ejemplos podrían ser:

```
public class AppOptions
{
        /// <summary>
        /// Constructor sin parámetros necesario para el patrón Options
        /// </summary>
        public AppOptions()
        {
                Console.WriteLine($"Construyendo objeto de tipo {this.GetType ()}");
        }

        [System.ComponentModel.DataAnnotations.Range (1, 1000, ErrorMessage = "El valor debe e
star en el rango de 1 a 1000")]
        public int PropiedadInt { get; set; }

        [System.ComponentModel.DataAnnotations.Required]
        [System.ComponentModel.DataAnnotations.MinLength(1)]
        public string PropiedadString { get; set; }
}
```

Para poder validar los atributos de validación en la aplicación deberemos añadir una referencia al componente '**Microsoft.Extensions.Options.DataAnnotations**' y añadir el método de extensión **ValidateDataAnnotations** en el registro del tipo de opciones, junto con el resto de validaciones, si las hay. (Puede haber otras validaciones adicionales.)

Las validaciones de los atributos **se ejecutarán posteriormente** al resto de las validaciones que hayamos especificado.

Por ejemplo:

```
// Validaciones
optionsBuilder.Validate<IConfiguration>((opts, cfg) =>
{
        Console.WriteLine("Ejecutando acción de validación 1");
        return true;
});
optionsBuilder.ValidateDataAnnotations();
optionsBuilder.Validate((opts) =>
{
        Console.WriteLine("Ejecutando acción de validación 2");
        if (string.IsNullOrEmpty(opts.PropiedadString))
        {
                throw new Exception("No se ha superado la validación por este motivo ...");
        }
        if (opts.PropiedadInt > 1000)
        {
                return false;
        }
        return true;
});
```

Más detalles de este patrón

El patrón Options permite especificar variantes o versiones diferentes para diferentes usos. El interfaz **IOptions<TipoOpciones>** es probablemente el más empleado, pero existen otras versiones que son: **IOptionsMonitor<TipoOpciones> y IOptionsSnapshot<TipoOpciones>** y algunos interfaces relacionados con éstos para tareas más específicas.

En casos más avanzados se pueden necesitar **varios objetos de opciones del mismo tipo con contenidos diferentes**. Cada uno de estos empleará un nombre (identificador único) para diferenciarse de los demás. Los métodos de registro, configuración, post configuración y validación permiten adaptarnos a estos casos (opciones con nombre) sin mayor problema.

Se puede especificar, para casos avanzados igualmente y para tener un mayor control sobre las acciones o validaciones, un tipo / una clase que realice las labores de configuración, post configuración y validación (o solamente alguno de éstos). En la mayor parte de los casos las acciones sencillas de configuración, post configuración y validación serán suficientes para casi cualquier caso, pero en casos extremos este enfoque puede ser necesario.

Todos estos detalles están especificados en detalle en el siguiente documento de esta serie (PatronOptionsAvanzado).

El funcionamiento interno de los elementos que forman el 'patrón options' es un buen ejemplo de cómo se suele emplear (o cómo se debe emplear) un proveedor de servicios con inyección de dependencias para relacionar unos objetos con otros de los que depende y de otros detalles importantes de implementación. Este funcionamiento interno se explica en detalle en el documento (PatronOptionsInternals).