CLASE 4

Fundamentos de ASP.NET Core y Web API

A partir de ahora vamos a profundizar en algunos de los conceptos que vimos en la clase anterior, específicamente los fundamentos de ASP.Net Core que se aplican a todos los tipos de aplicaciones de Asp.Net Core y aparte específicos para aplicaciones WebApi.

Controladores y Acciones

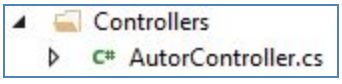
La idea fundamental de tener un web API es que tendremos clientes que harán peticiones HTTP al web API. Estas peticiones HTTP se hacen a un conjunto de URLs de nuestro dominio. A estas URLs las llamamos **endpoints** del Web API es decir que un **endpoint** es donde reside un recurso que representa alguna funcionalidad que brinda el Web API.

Dominio: http://localhost:puerto	**Endpoint**: http://localhost:puerto/api/autor

Cuando hacemos una petición HTTP a un endpoint del web API típicamente queremos que se ejecute ***una función de un controlador***, que hablando en terminología de objetos **es *un método de una clase***, y hablando en terminología de APIs ***es una acción***.

Entonces una acción es una función de un controlador que se ejecuta en respuesta a una petición HTTP realizada al web API, mientras un controlador es una clase que agrupa a un conjunto de acciones que se relacionan con un recurso.

Por convención los controladores se nombran concatenando una palabra como el nombre de un recurso con la palabra controller. Así un controlador de recursos de autor que puede consultar, agregar, modificar, procesar, etc….autores, se puede llamar **AutorControllers**.



Estos controladores por convención están en una carpeta llamada **Controllers**, y esto ya está armado por el template básico de Visual Studio.Net para este tipo de proyecto.

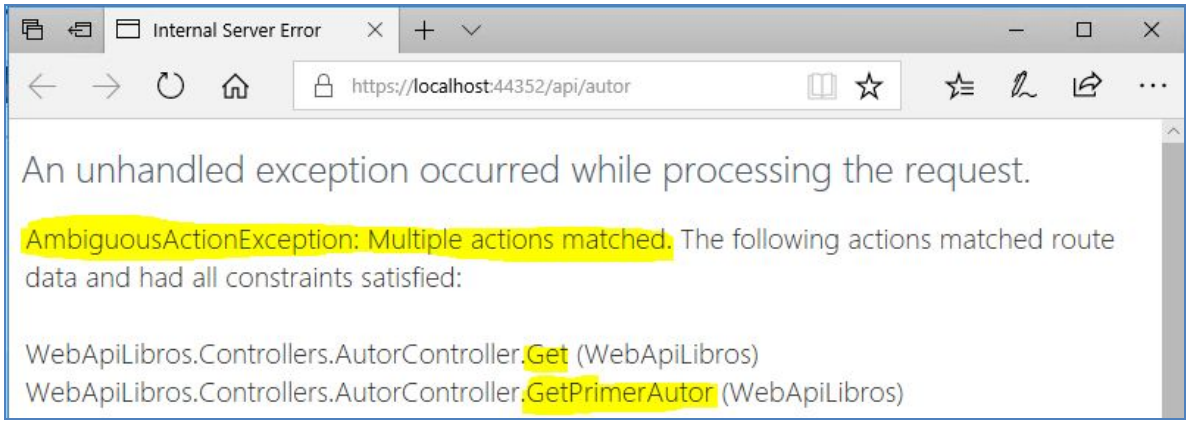
Cada acción del controlador está mapeada a un método HTTP, si quisiéramos otra acción en el controlador Autor, además de las que ya agregamos en la ejercitación

anterior, debemos crear una función e indicarle a qué método HTTP va a responder, es decir ***mapeamos la acción a un método HTTP***.

Por ejemplo si queremos un nuevo endpoint para obtener el primer autor de la base de datos tendríamos esta acción:

```
[HttpGet]
public ActionResult<Autor> GetPrimerAutor()
{
    return context.Autores.FirstOrDefault();
}
```

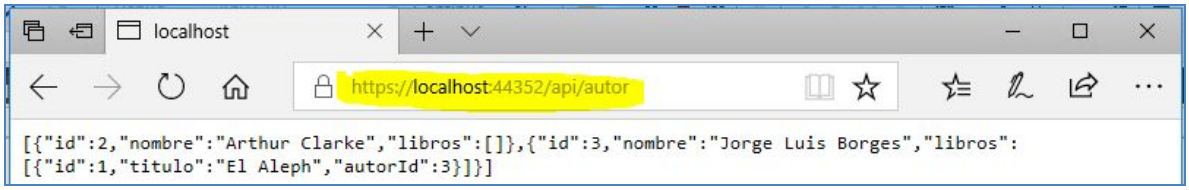
Si compilamos la api obtendrá un error porque el endpoint no es único, dado que **[httpGet]** coincide con varias acciones del controlador, coincide con la acción **Get** o con la acción **GetPrimerAutor**.

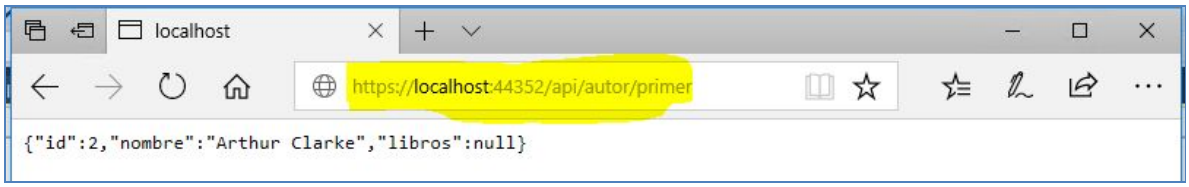


Lo que necesitamos es definir una URL diferente para las dos acciones. Entonces para la acción **GetPrimerAutor** podemos agregar un segmento de URL que se va a corresponder con esta acción.

```
[HttpGet("primer")]
public ActionResult<Autor> GetPrimerAutor()
{
    return context.Autores.FirstOrDefault();
}
```

Si compilamos y ejecutamos nuevamente, veremos el siguiente resultado, la acción Get retorna todos los autores y primer retorna el primero de la lista.





Básicamente lo que acabamos de hacer es indicar es una regla de ruteo.

Ruteo

Las reglas de ruteo son las que nos permiten mapear URLs con una acción es decir que cuando hacemos una petición HTTP a un endpoint, queremos ejecutar una acción.

Las configuraciones de ruteo las podemos realizar en dos lugares del proyecto:

- en los controladores a lo que se le llama **ruteo por atributo**, o,
- en la clase **startup** a lo que se le llama **ruteo convencional**.

Ruteo por atributo

Atributo Route

Lo primero que vemos en el código de un controlador es el atributo **Route** que permite indicar la base del endpoint de las acciones del controlador.

```
[Route("api/[controller]")]
[ApiController]
public class AutorController : ControllerBase
{
}
```

Esto significa que por omisión todas las acciones del controlador van a tener un endpoint que empiece con lo indicado en el atributo **Route**.

El **string [controller]** se va a reemplazar por el nombre del controlador (sin el sufijo Controller de la definición de la clase controlador), en el código anterior el valor del **string [controller]** sería **Autor**.

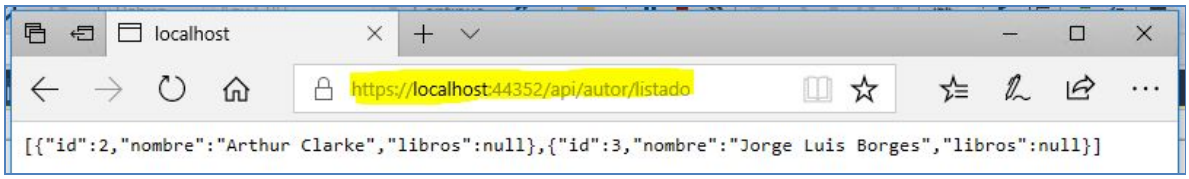
Personalizar el endpoint con el atributo HttpGet

Podemos personalizar el endpoint de cada acción pasándoles reglas de ruteo al atributo HTTP correspondiente, como hicimos en el ejemplo anterior que recupera el primer autor de la lista o como en este código:

```
[HttpGet("listado")]
        public ActionResult<IEnumerable<Autor>> Get()
        {
```

```
        return context.Autores.ToList();
    }
```

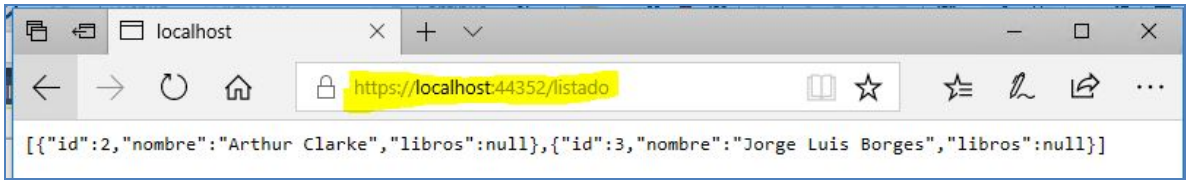
A esto se lo llama ***combinación*** porque combinamos lo indicado en el atributo **Route** que es **api/autor**, con lo indicado en el atributo **HttpGet** que es **listado**, obteniendo el endpoint **api/autor/listado**.



Ignorar el atributo Route

Si no queremos combinar y queremos ignorar el atributo **Route** del controlador, en el atributo HttpGet se puede indicar la ruta completa comenzando con una contrabarra **/**, por ejemplo:

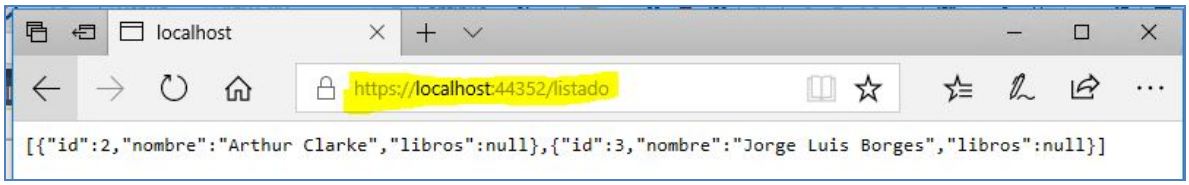
```
[HttpGet("/listado")]
    public ActionResult<IEnumerable<Autor>> Get()
    {
        return context.Autores.ToList();
    }
```

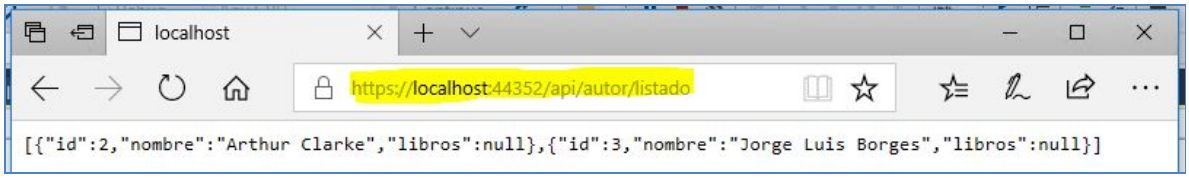


Varios endpoint para una misma acción

También podemos tener una acción que responda a más de un endpoint, agregando todos los atributos HTTP necesarios, por ejemplo, ignorando el atributo **Route** o combinando el atributo **Route**:

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



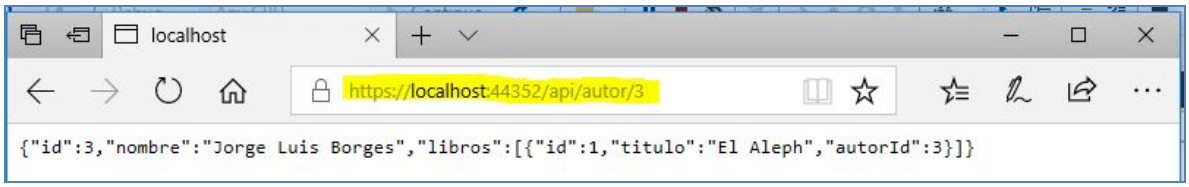


Parámetros en la plantilla de ruteo: obligatorios, opcionales, con valor default

También podemos definir parámetros en la plantilla de ruteo, por ejemplo en esta la acción **Get** espera un parámetro entero llamado **id**:

```
        [HttpGet("{id}")]
         public ActionResult<Autor> Get(int id)
         {
             var resultado = context.Autores.FirstOrDefault(x => x.Id ==
id);

             if (resultado == null)
             { return NotFound(); }

             return resultado;
         }
```

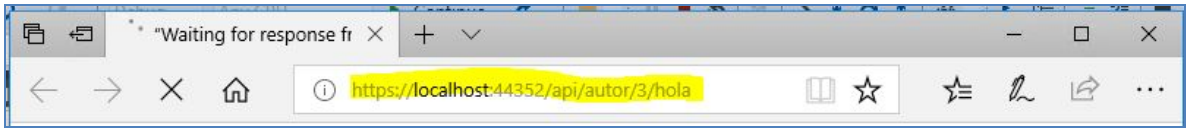
Esta plantilla es ruteo indica el lugar donde se va a encontrar el parámetro en el endpoint, combinando lo indicado en el atributo **Route** con el parámetro indicado en el atributo **HttpGet** que será un número entero. La plantilla es la que define que parámetro de la URL se mapea con que parámetro de la función, en este caso el id.

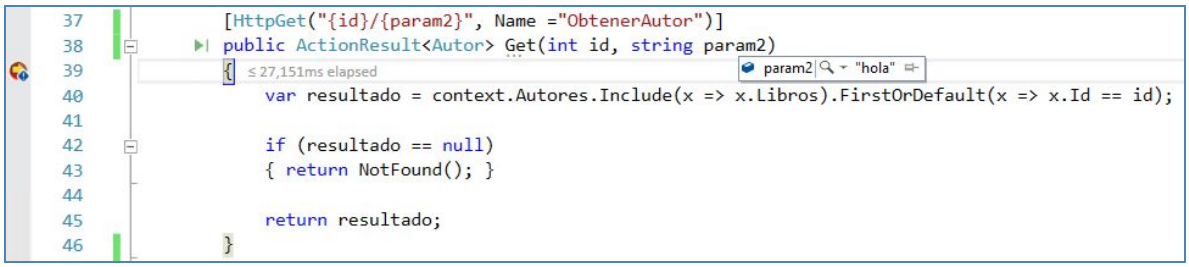


Se puede pasar más de un parámetro a un endpoint como en el siguiente ejemplo, lo probamos con un breakpoint para que vea que el endpoint existe y funciona correctamente cuando se le pasan dos parámetros: el primero tipo entero y el segundo tipo string:

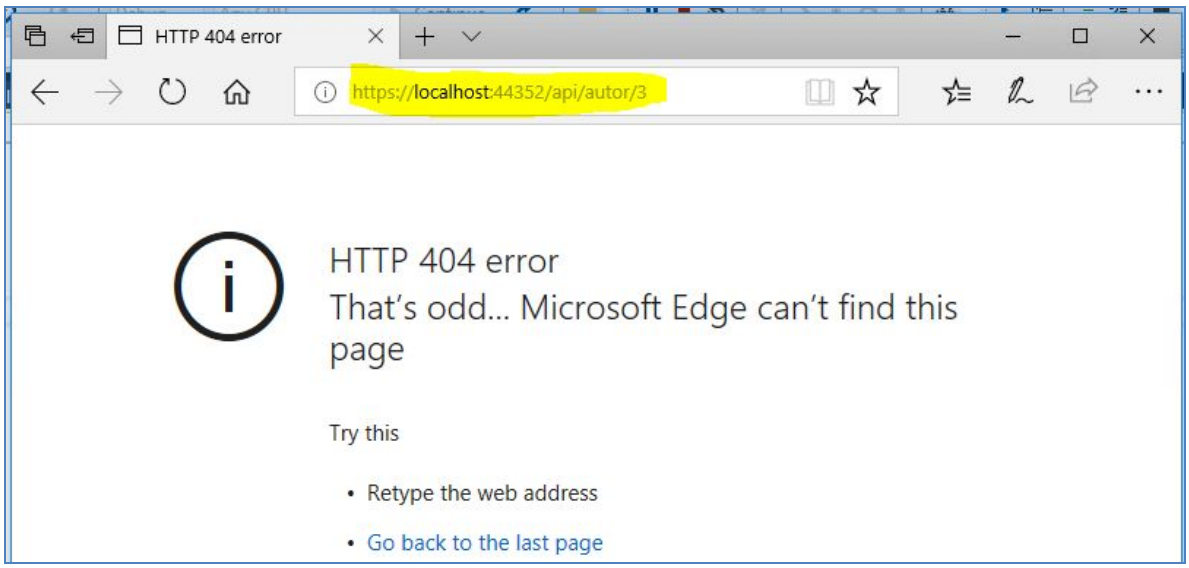
```
        [HttpGet("{id}/{param2}")]
         public ActionResult<Autor> Get(int id, string param2)
```

```
        {
            var resultado = context.Autores.FirstOrDefault(x => x.Id ==
id);

            if (resultado == null)
            { return NotFound(); }

            return resultado;
        }
```



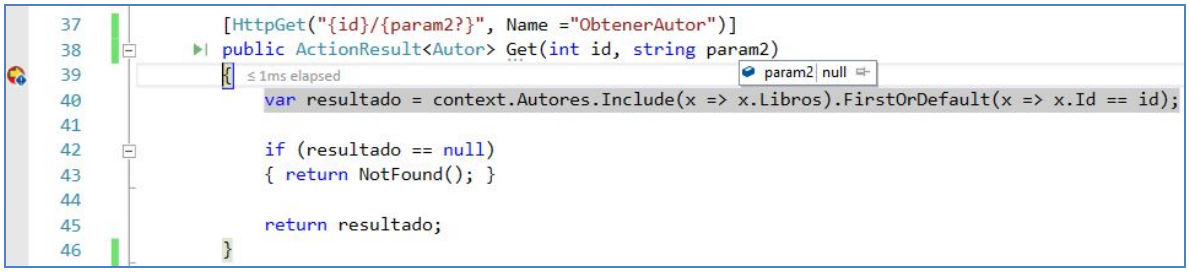


Pero si no le pasamos el segundo parámetro la URL no funciona dado que espera que se lo pasen:



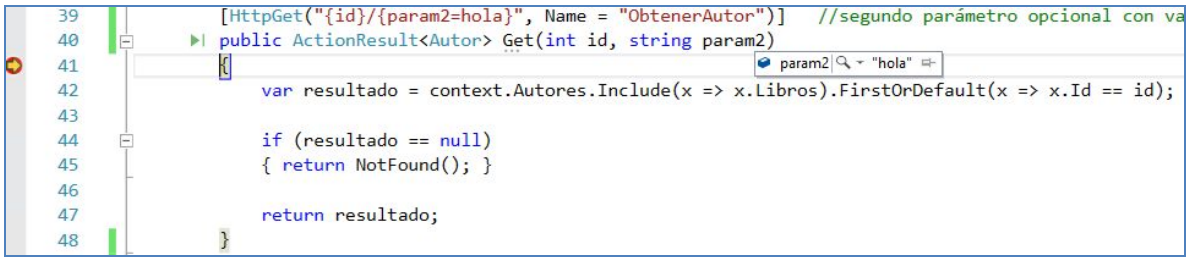
…Pero, podríamos hacer que este parámetro sea opcional agregándole un signo de interrogación con la cual el valor del parámetro opcional si no se pasa, es **null** y si se pasa, tendrá el valor string:

```
        [HttpGet("{id}/{param2?}")]
         public ActionResult<Autor> Get(int id, string param2)
         {
             var resultado = context.Autores.FirstOrDefault(x => x.Id ==
id);

             if (resultado == null)
             { return NotFound(); }

             return resultado;
         }
```



Puedo tener parámetros opcionales con un valor por omisión como en el siguiente código:

```
        [HttpGet("{id}/{param2=hola}")]
         public ActionResult<Autor> Get(int id, string param2)
         {
             var resultado = context.Autores.FirstOrDefault(x => x.Id ==
id);

             if (resultado == null)
             { return NotFound(); }

             return resultado;
         }
```



Tipos de dato que puede retornar una acción

Desde una acción podemos retornar lo siguiente:

- un tipo específico
- un ActionResult<T>
- un IActionResult

Tipo específico y ActionResult

Es una clase que se representa todos los tipos de datos que se pueden retornar desde una acción, es decir el "*resultado de la acción*". Hay clases derivadas de **ActionResult** que se pueden utilizar para indicar que queremos retornar un tipo de dato específico.

Por ejemplo si tenemos una acción que siempre queremos que retorne un JSON podríamos utilizar la clase **JsonResult, s**in embargo en vez de utilizar ActionResult como tal, solemos utilizar otras opciones y las más comunes son: **ActionResult<T>** y **IActionResult**.

Vamos a empezar viendo el tipo de retorno más simple que es un tipo específico y puede ser cualquier tipo de dato, ya sea un string u objeto complejo creado por nosotros, sin incluir un **ActionResult**.

Por ejemplo podemos tener este código donde la acción Get retorna un valor tipo Autor:

```
public Autor Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

    if (resultado == null)
    { return NotFound(); }

    return resultad
}
```

NotFoundResult ControllerBase.NotFound() (+ 1 overload)
Creates an NotFoundResult that produces a StatusCodes.Status404NotFound response.

Cannot implicitly convert type 'Microsoft.AspNetCore.Mvc.NotFoundResult' to 'WebApiLibros.Entidades.Autor'

El problema con esto es que las acciones pierden versatilidad, observe que el error que marca el retorno de **NotFound()**, indica que no se puede retornar un status 404 si no existe el autor, dado que estamos hablando de otro tipo de dato distinto a un objeto Autor.

Para ganar flexibilidad nuevamente podemos utilizar una **ActionResult** de algún tipo de dato, esto es posible por la cualidad de Generics propia del lenguaje C# (`using System.Collections.Generic;`), podemos retornar tanto **un tipo Autor** como un **ActionResult**, solo esos dos tipos de valores de retorno.

```
public ActionResult<Autor> Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

    if (resultado == null)
    { return NotFound(); }

    return resultado;
}
```

.

IActionResult

El **IActionResult** es parecido al ActionResult<T>, solo que IActionResult solamente permite retornar un ActionResult, pero cualquier tipo de acción.

Por ejemplo analizamos el siguiente ejemplo donde retornamos un IActionResult, pero la función reconoce el ActionResult pero no reconoce el tipo Autor, recordamos que NotFound() hereda de ActionResult:

```
public IActionResult Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

    if (resultado == null)
    { return NotFound(); }

    return resultado;
}
```

(local variable) Autor resultado

Cannot implicitly convert type 'WebApiLibros.Entidades.Autor' to 'Microsoft.AspNetCore.Mvc.IActionResult'. An explicit

Para resolver esto podemos usar la función **Ok(),** como una alternativa produciendo un código de estado 200 OK, pero con la desventaja que puede retornar además cualquier tipo de dato, no solo un Autor, sino cualquier otro tipo como un string, una fecha, etc., porque retorna un **ObjectResult** que es un tipo de dato muy ampllio (recuerde que en C# todo hereda de System.Object):

```
public IActionResult Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

    if (resultado == null)
    { return NotFound(); }

    return Ok(resultado);
}
```

OkObjectResult ControllerBase.Ok(object value) (+ 1 overload)

Creates an OkObjectResult object that produces an StatusCodes.Status200OK response.

```
public IActionResult Get(int id, string param2)
{
    var resultado = context.Autores.Include(x => x.Libros).FirstOrDefault(x => x.Id == id);

    if (resultado == null)
    { return NotFound(); }

    //return Ok(resultado);
    return Ok("hola");
}
```

Esto es darle demasiada responsabilidad y puede traer problemas de consistencia de información porque no dará un error cuando retorne algún valor que no sea de tipo Autor. Por eso la buena práctica recomendada es utilizar ActionResult de un tipo específico: en este ejemplo sería **ActionResult<Autor>**.

Programación Asincrónica

Una acción puede ser sincrónica o asincrónica.

Todo lo visto hasta ahora han sido de acciones sincrónicas, es decir que hasta que no se terminan por completo de ejecutar todas las instrucciones de la acción, ésta no termina. Veremos ahora acciones asincrónicas.

En una **acción asíncrona** permite que mientras dicha función se ejecute nuestro servidor web en vez de quedarse esperando a que la acción termine por completo, realice otras tareas así puede aprovechar y optimizar su tiempo de procesamiento, minimizando las esperas.

Sin embargo es importante destacar que no siempre queremos hacer todas las acciones asincrónicas, y en algunos casos y según la lógica de negocio a procesar, no se puede ejecutar una acción en forma asincrónica.

Hacer una acción asíncrona tiene un pequeño costo de rendimiento. En general si se harán operaciones con bases de datos o comunicación con otros servicios web, vale la pena hacer acciones asincrónicas.

Por ejemplo si una acción se comunica con la base de datos nos conviene utilizar programación asincrónica en esta acción porque la operación en la base de datos es una operación independiente de nuestra aplicación así que nos conviene esperar a que esa operación termine y con programación asincrónica haremos que los recursos del servidor web se aprovechen en forma más eficiente.

En el ejemplo de código, los nombres de los métodos de EF finalizarán en **Async,** la función deberá ser marcada como **async** y instrucción que se ejecuta asincrónicamente debe tener un **await** para que espere hasta terminar. Una acción asincrónica retorna un **Task** de **ActionResult**.

```
        [HttpGet("primer")]
        public async Task<ActionResult<Autor>> GetPrimerAutor()
//asincrónico
        {
            return await context.Autores.FirstOrDefaultAsync();
        }
```

Esta es la buena práctica a seguir a la hora de construir web APIs.

Validación de modelos con atributos predefinidos

La información que es recibida en cada acción, debe ser válida según las necesidades y reglas del negocio. Es decir, debemos “validar”, y para validar los modelos podemos utilizar atributos.

Profundizando el tema “validación”, en una aplicación siempre hay validaciones comunes y procesos de validación complejos. Respecto de las validaciones comunes, un ejemplo es si hay datos de carga o envío obligatorios, un rango permitido de un valor numérico, etc.

Detallamos los atributos más usados:

- **Required:** sirve para indicar una propiedad requerida es decir que su valor debe estar presente.
- **StringLenght**: que indica la longitud máxima de un string.
- **Range**: que iindica el rango en que se debe encontrar un valor.
- **CreditCard**: que valida el formato del número de una tarjeta de crédito.
- **Compare**: que valida que dos propiedades sean iguales.
- **Phone**: que valida el formato de un número telefónico.
- **RegularExpression**: valida que el valor de la propiedad coincida con una expresión regular.
- **URL**: valida el formato de una URL.
- **BindRequired**: requiere que se haga binding a la propiedad.

Nota: en este link se encuentran todas las clases de atributos de validación del namespace **System.ComponentModel.DataAnnotations**, con su definición y ejemplos de código:
https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netframework-4.8

Para utilizar estos atributos debemos colocarlos encima de la propiedad o parámetro que deseamos validar.

Por ejemplo en el caso del parámetro de una acción podríamos utilizar **Required** a nivel del parámetro de la acción y si no enviamos un valor para el parámetro nombre, se retornará un 400 BadResult.

```
[HttpGet("{id}")]
public ActionResult<string> Get(int id, [Required] string nombre)
{
    return "value";
}
```

Lo más habitual, consistente y centralizado en relación al código es tener estas validaciones a nivel del modelo y no a nivel de parámetros de una acción.

```
[HttpPost]
public void Post([FromBody] Autor value)
{…}
public class Autor{
    public int Id { get; set; }
    [Required]
    public string Nombre { get; set; }
}
```

Por ejemplo en la acción **POST FromBody Autor**, y con el modelo planteado de la clase **Autor** que tiene el atributo **Required** en la propiedad Nombre indicando que esta propiedad es requerida.

Otros ejemplos, clases película y estudiante:

```
public class Movie
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    [ClassicMovie(1960)]
    [DataType(DataType.Date)]
    [Display(Name = "Release Date")]
    public DateTime ReleaseDate { get; set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; set; }

    [Range(0, 999.99)]
    public decimal Price { get; set; }

    public Genre Genre { get; set; }

    public bool Preorder { get; set; }
}
```

https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-3.1

```
public class Student
{
      public int StudentID { get; set; }

      [Required(ErrorMessage = "Enter Student Name")]
      [StringLength(50, ErrorMessage = "Only 50 character are
      allowed")]
      public string StudentName { get; set; }

      [Required(ErrorMessage = "Please enter Date of Birth")]
      public DateTime StudentDOB { get; set; }

      [RegularExpression(@"[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}",
      ErrorMessage = "Please enter Valid Email ID")]
      [Required(ErrorMessage = "Please enter Student EmailID")]
      public string StudentEmailID { get; set; }

      [Range(5000, 15000, ErrorMessage = "Please enter valid range")]
      [Required(ErrorMessage = "Please enter Student Fees")]
      public decimal StudentFees { get; set; }
```

```
        [Required(ErrorMessage = "Please enter Student Address")]
        [StringLength(50, ErrorMessage = "Only 50 character are
        allowed")]
        public string StudentAddress { get; set; }

        [DataType(DataType.Password)]
        [Required(ErrorMessage = "Please enter password")]
        public string Password { get; set; }

        [Required(ErrorMessage = "Please enter ConfirmPassword")]
        [DataType(DataType.Password)]
        [Compare("Password", ErrorMessage = "Password not matching")]
        public string ConfirmPassword { get; set; }
}
```

Estas son reglas de validación predeterminadas pero también podemos crear reglas de validación propias y personalizadas de dos formas distintas, por atributo y por modelo.

Validación personalizada de modelos

Las validaciones predefinidas son bastante amplias y ayudan en escenarios bastante estándar, pero hay casos con validaciones más complejas y muy ligadas a los procesos de negocio, lo que llevará a que necesitemos crear validaciones personalizadas.

Los tipos de validación personalizada son:

- por atributo
- por modelo

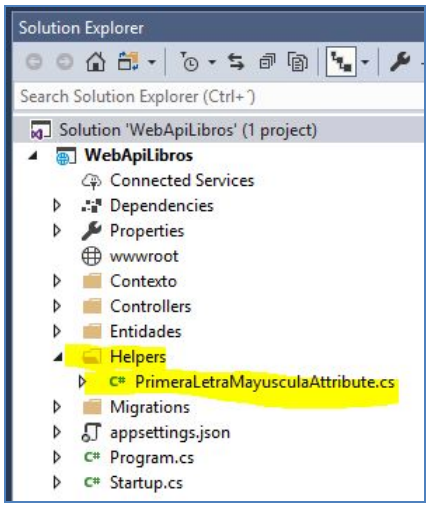
Validación personalizada por atributo

Esta validación por atributo tiene la ventaja de poder ser reutilizada en distintos modelos y propiedades pero su enfoque es el de validar una sola propiedad y no un modelo completo.

Si quisiéramos validar un conjunto de propiedades de un modelo deberíamos usar las validaciones personalizadas por modelo, que tiene la ventaja de poder acceder a todas las propiedades del modelo para hacer reglas de validación complejas, pero tienen la desventaja que es una validación a nivel de modelo y no se puede reutilizar con otros modelos.

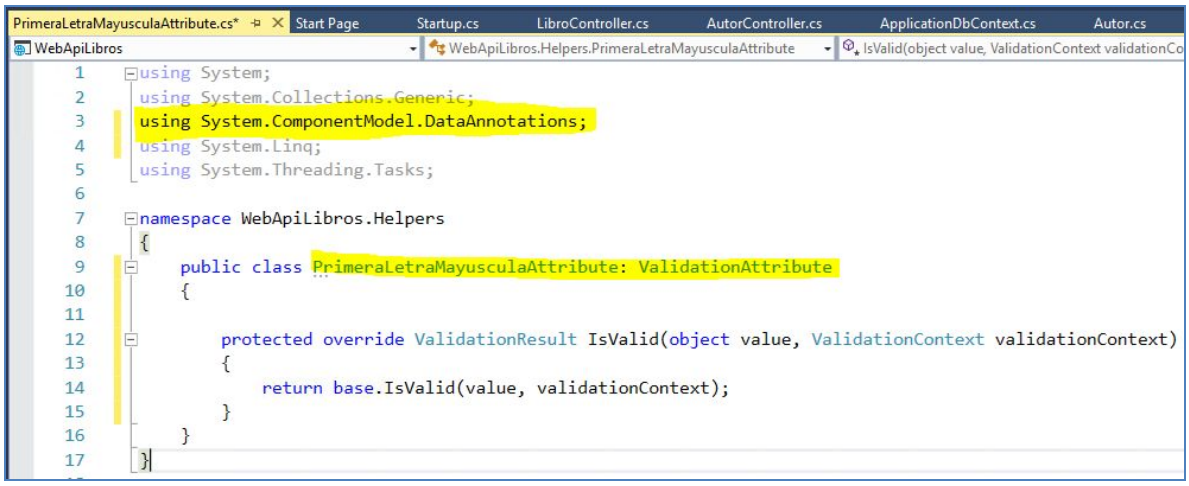
Supongamos que necesitamos que la propiedad nombre de la clase **Autor** tenga la primer letra en mayúsculas, dado que esta validación no involucra varias propiedades y es una regla de negocio que tal vez a futuro la quisiéramos aplicar en distintas

propiedades, es conveniente utilizar una regla de validación personalizada por atributo.



Para organizar mejor los elementos del proyecto, agregaremos una carpeta **Helpers** al proyecto y allí crearemos una clase llamada **PrimeraLetraMayusculaAttribute**, la razón de la palabra Attribute en el nombre de la clase es por convención en Net.

La nueva clase debe heredar de la clase **ValidationAttribute** del namespace **System.ComponentModel.DataAnnotatios**. Y es necesario sobrescribir el método **IsValid** y dentro de este método irá el código de la regla de validación.



El parámetro **value** de la función **IsValid** trae el valor de la propiedad en donde se ha ubicado el atributo, es decir si el atributo personalizado se ubicó en la propiedad nombre, en **value** vendrá el valor de nombre. Y el parámetro **validationContext** trae información acerca del contexto en el cual se está ejecutando la validación, por ejemplo a través de **validationContext** podemos ver el objeto que se está validando.

La desventaja de utilizar el objeto como tal a través de **validationContext** es que limita el atributo de validación a una clase específica, y la validación por atributo pierde flexibilidad.

Implementaremos las reglas de validación con el siguiente código:

```
protected override ValidationResult IsValid(object value,
ValidationContext validationContext)
        {
```

```
            if (value == null || string.IsNullOrEmpty(value.ToString()))
            {
                return ValidationResult.Success;
            }
            var primeraLetra = value.ToString()[0].ToString();

            if (primeraLetra != primeraLetra.ToUpper())
            {
                return new ValidationResult("La primer letra debe ser en
mayúscula!");
            }
            return ValidationResult.Success;
        }
```

Lo primero a hacer es verificar si el valor es nulo o vacío, si esta condición es verdadera retornamos **Success** indicando que el atributo ha pasado la validación. Esto lo hacemos porque ya tenemos un atributo que se encarga de verificar la presencia de un valor y es **Required**, como este control ya lo hace **Required** no lo hacemos aquí porque tendríamos múltiples validaciones solapadas, y lo correcto en POO es tener bajo acoplamiento y alta cohesión, con el concepto que cada validación debe asegurarse de validar una sola cosa y no solapar validaciones entre sí.

A continuación buscamos la primer letra y verificamos que sea mayúscula, si no lo es retornamos un new **ValidationResult** con el mensaje de error correspondiente.

Finalmente si la primera letra es mayúscula retornamos **Success**.

Ahora probaremos la regla de validación personalizada en la clase Autor ubicando la nueva validación en la propiedad nombre.

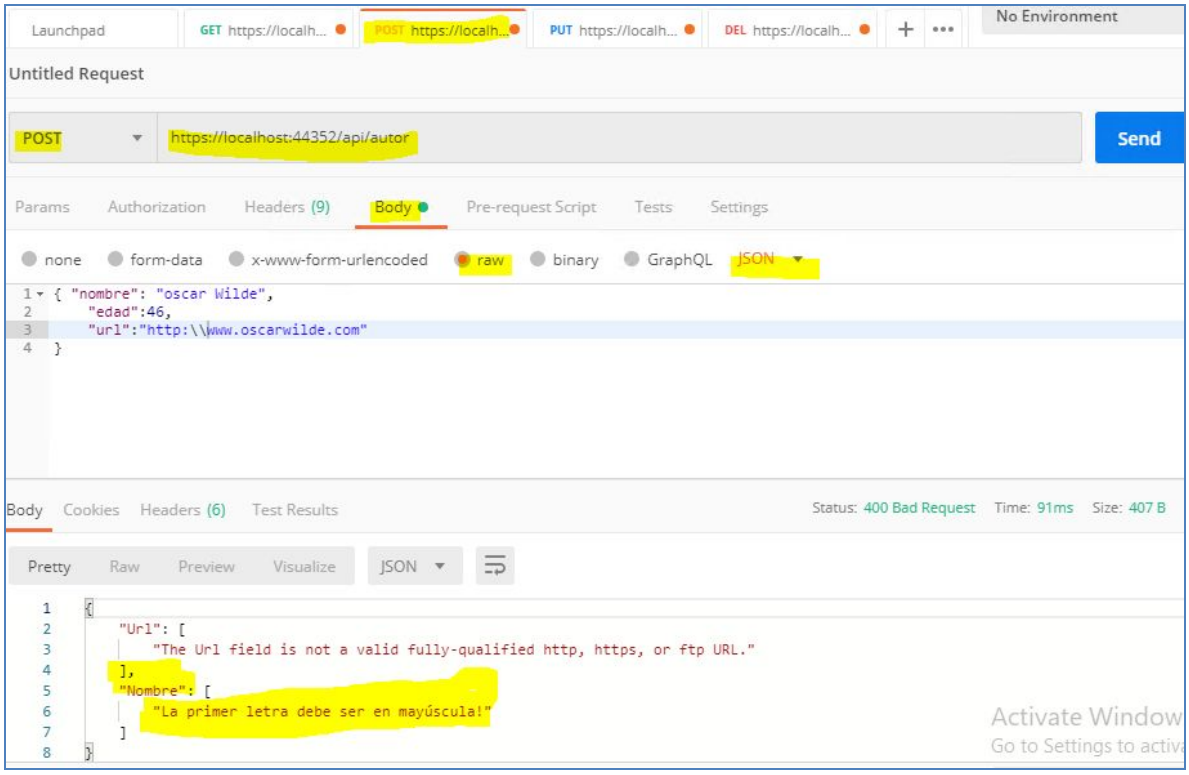
```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using WebApiLibros.Helpers;

namespace WebApiLibros.Entidades
{
    public class Autor
    {
        public int Id { get; set; }

        [Required]
        [PrimeraLetraMayuscula]
        [StringLength(30, ErrorMessage = "El nombre del autor puede tener
hasta {1} caracteres")]
```

```
        public string Nombre { get; set; }

        [Range(18,120)]
        public int Edad { get; set; }

        [CreditCard]
        public string TarjetaCredito{ get; set; }

        [Url]
        public string Url{ get; set; }

        public List<Libro> Libros { get; set; }
    }
}
```

Nota: Link interesante que ayuda a comprender los parámetros pasados al atributo **StringLenght**:

https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.stringlengthattribute?redirectedfrom=MSDN&view=netframework-4.8

Podemos probar su funcionamiento con Postman, previo compile el proyecto:



Validación personalizada por modelo

En la validación personalizada por modelo podemos realizar validaciones para las mismas reglas de negocio anterior. Para ello necesitamos que el modelo implemente la interfaz **IValidatableObject**, y vemos que esta interfaz retorna un conjunto de objetos **ValidationResult** que quiere decir que dentro de esta función podemos hacer varias validaciones y retornar distintos errores de validación.

Es decir, dentro de la implementación del método **Validate** podríamos verificar todas las validaciones de **Autor**, primera letra en mayúsculas, el rango de edad entre 18 y 120, el formato de la tarjeta de crédito, validaciones cruzadas entre campos y otros procesos de negocio, todo porque la validación personalizada por modelo toma en cuenta todo el modelo.

La desventaja de esta validación es que se limita al modelo y en general no podemos reutilizarla en otros modelos, pero la lógica de negocio del modelo queda centralizada en una sola función.

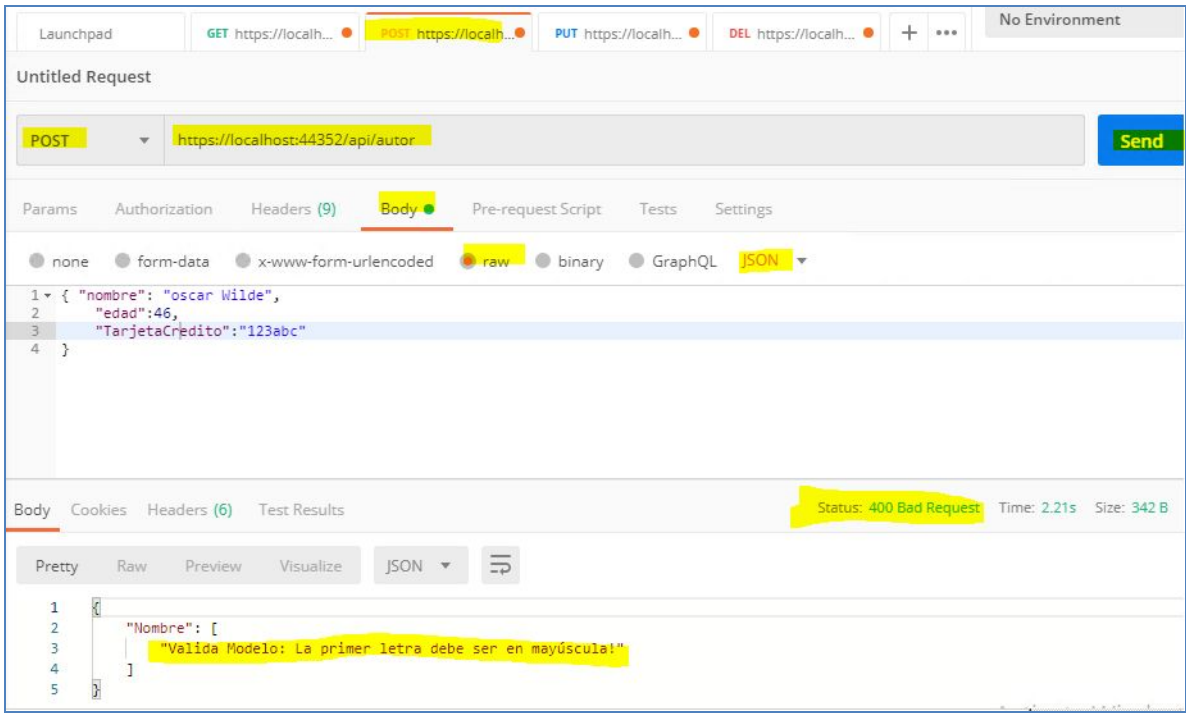
El código de esta validación sería este:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;
using WebApiLibros.Helpers;

namespace WebApiLibros.Entidades
{
    public class Autor : IValidatableObject
    {
        public int Id { get; set; }

        [Required]
        [PrimeraLetraMayuscula]
        [StringLength(30, ErrorMessage = "El nombre del autor puede tener
hasta {1} caracteres")]
        public string Nombre { get; set; }

        [Range(18, 120)]
        public int Edad { get; set; }

        [CreditCard]
        public string TarjetaCredito { get; set; }

        [Url]
        public string Url { get; set; }

        public List<Libro> Libros { get; set; }
```

```
        public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
        {
            if (!string.IsNullOrEmpty(Nombre))
            {
                var primeraLetra = Nombre.ToString()[0].ToString();

                if (primeraLetra != primeraLetra.ToUpper())
                {
                    yield return new ValidationResult("Valida Modelo: La
primer letra debe ser en mayúscula!", new string[] { nameof(Nombre)} );
                }
            }
        }
    }
}
```

Nota: un buen ejemplo de yield para quienes no conozcan esta instrucción http://www.thatcsharpguy.com/posts/la-palabra-yield-en-c/

Volvemos a probar con Postman y tenga presente que en tiempo de ejecución primero se realizan las validaciones por atributo y luego por modelo, se sugiere comentar la validación por atributo **[PrimeraLetraMayuscula]** y la validación **[CreditCard]** así puede ver la validación por modelo.



Inyección de dependencias

Las clases de la aplicación no son completamente autosuficientes y en POO es normal separar responsabilidades entre distintas clases. Lo mismo ocurre con los controladores, dado que son clases.

Un controlador contiene un conjunto de acciones y la responsabilidad del controlador es recibir peticiones HTTP realizadas hacia los endpoints para procesarlas. Pero por ejemplo, un controlador no tiene la responsabilidad de guardar registros en una base de datos, ni describir mensajes en la consola, ni de calcular el cierre financiero contable de un balance de la empresa. Las tareas se delegan en otras clases.

Cuando una clase A utiliza una clase B decimos que la clase B es una dependencia de la clase A. Las dependencias son inevitables y han sido estudiadas en arquitectura de software y utilizamos el término acoplamiento cuando hablamos de la interacción entre clases.

Tenemos dos tipos de acoplamiento:

- **Alto acoplamiento**: se caracteriza por una dependencia poco flexible en otras clases y no es bueno en POO y modelado de objetos.

  Supongamos que tenemos una clase A y una clase B y un método de la clase hacemos lo siguiente:

```
public class ClaseA{
        public void RealizarAccion(){
            ClaseB claseB = new ClaseB();
            claseB.HacerAlgo();
        }
}
```

Tenemos un método llamado **RealizarAccion()** que instancia un objeto de la clase B para llamar a su método **HacerAlgo()**, estamos usando la clase B dentro de la clase A y esto es una dependencia.

El problema está en la falta de flexibilidad que existe en esta dependencia, no hay manera de poder usar el método de **RealizarAcción()** de la clase A sin usar la clase B y esto no se recomienda. Lo ideal es tener bajo acoplamiento.

- **Bajo acoplamiento**: se da cuando existe facilidad de intercambiar dependencias. Una formar de lograr esta flexibilidad es utilizando ***inyección de dependencias***.

La **inyección de dependencias** es una técnica en donde las dependencias de un objeto son suplidas por otro objeto. Una manera forma de utilizar inyección de dependencias es inyectando las dependencias de una clase en su constructor.

```
public class ClaseA {
    private readonly ClaseB claseB;
    public ClaseA (ClaseB claseB){
      this.claseB = claseB;
    }

    public void RealizarAccion (){
      claseB.HacerAlgo();
    }
}
```

Entonces en el caso de la clase A podemos inyectar la clase B a través del constructor de la clase A. La clase B será un parámetro del constructor de la clase A, de tal modo que para utilizar la clase A necesitamos pasar clase B como parámetro al instanciar un objeto de la clase A.

Esto es un poco mejor que el alto acoplamiento porque ahora la dependencia de la clase A con la clase B es explícita pero… la clase A sigue teniendo una dependencia poco flexible con la clase B y en algunas ocasiones seguro querremos cambiar la implementación de la clase B por otra y le vamos a querer pasar esta nueva implementación a la clase A. Una forma de lograr esto es con **interfaces.**

Evolucionando el ejemplo creamos una interfaz llamada **IClaseB** que tiene la definición del método **HacerAlgo()**, y la clase B deberá implementar esta interfaz; también nutrir el ejemplo creamos otra clase llamada **ClaseB2** que va a implementar la misma interfaz:

```
public class ClaseB : IClaseB
{
    public void HacerAlgo()
    {
    }
}

public class ClaseB2 : IClaseB
{
    public void HacerAlgo()
    {
    }
}
```

Finalmente en la clase A cambiamos clase B por **IClaseB**:

```
public class ClaseA
{
    private readonly IClaseB claseB;
    public ClaseA(IClaseB claseB)
    {
        this.claseB = claseB;
    }
    public void RealizarAccion()
    {
        claseB.HacerAlgo();
    }
}
```

Para instanciar la clase A podemos hacer:

```
ClaseA claseA = new ClaseA(new ClaseB());
ClaseA claseA2 = new ClaseA(new ClaseB2());
```

Con este cambio la clase A ya no depende de la clase B sino que depende de la interfaz **IClaseB**, la diferencia está en que ahora cualquier clase que implemente la interfaz **IClaseB** puede ser inyectada en la clase A y esta flexibilidad logra bajo acoplamiento entre las clases A y B porque se pueden cambiar las dependencias de la clase A siempre y cuando se respete la interfaz **IClaseB**.

Servicios

Los conceptos presentados aquí para manejo de servicios, son avanzados pero se introducen brevemente para que a futuro los tenga en cuenta al momento de trabajar con APIs.

Retomando lo hablado anteriormente, para lograr un buen modelado de objetos es imprescindible mantener las clases enfocadas en sus tareas específicas según su responsabilidad y delegar a otras clases otro tipo de funcionalidades.

Aparte mantener bajo acoplamiento de las dependencias con otras clases. Esto se puede hacer con ASP.Net Core para facilitar la resolución de dependencias de las clases, configurando servicios en la clase **startup** de un proyecto.

En este contexto un **servicio** es una clase que puede ser inyectada a otras clases por el framework. Podemos configurar los servicios de nuestra aplicación y cuando estos servicios aparezcan en constructores o métodos de clases el framework, se va a encargar de inyectar la clase correspondiente.

Si tomamos el ejemplo de las clases A y B del ejemplo anterior, podemos realizar esta configuración dentro del método de **ConfigureServices** de la clase **startup** con el método **AddTransient** y la interfaz que queremos usar cuando la clase sea solicitada, en este caso la clase que queremos servir es **ClaseB** cuando se solicita algo que cumpla con la interfaz **IClaseB**. Entonces esta dependencia va a ser satisfecha con la clase suministrada en el segundo parámetro en cualquier lugar de la aplicación.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<MiFiltroDeAccion>();
    services.AddResponseCaching();
    services.AddTransient<IClaseB, ClaseB>();
```

Vemos aquí podemos la importancia de utilizar **inversión de controles** que es que podemos centralizar la resolución de dependencias en un solo lugar la nuestra aplicación. Más detalles en este link https://danielggarcia.wordpress.com/2014/01/15/inversion-de-control-e-inyeccion-de-dependencias/

Tiempo de Vida

Los servicios pueden tener distintos **tiempos de vida** esto es el tiempo que se sirve la misma instancia de una clase o una instancia distinta para que otros la usen.

Tenemos tres opciones disponibles para manejar el tiempo de vida de un servicio:

- la opción **Transient**: donde cada vez que un servicio se ha solicitado se va a servir una nueva instancia de la clase que representa al servicio. Para registrar un servicio como **Transient**, el código es el siguiente:

```
services.AddTransient<IClaseB, ClaseB>();
```

- la opción **Scoped**: los servicios tipo **Scoped** son creados uno por cada petición HTTP, es decir que si distintas clases piden el mismo servicio durante una petición HTTP se les va a entregar la misma instancia del servicio. Para registrar un servicio como **Scoped**, el código es el siguiente:

```
services.AddScoped<IClaseB, ClaseB>();
```

- la opción **Singleton**, donde siempre se nos dará la misma instancia del servicio siendo enviada una y otra vez a distintos lugares de la aplicación aún si es para servir a distintas peticiones HTTP de distintos usuarios. Para registrar un servicio como **Singleton**, el código es el siguiente:

```
services.AddSingleton<IClaseB, ClaseB>();
```

Para más detalles sugerimos estos links:
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1

https://aspnetcore.readthedocs.io/en/stable/fundamentals/dependency-injection.html

En los tres ejemplos anteriores se indican en los parámetros pasados, una dependencia de la clase con una interfaz, en los ejemplos **IClaseB** es la interfaz y **ClaseB** es la clase que implementa dicha interfaz.

Si tuviéramos una clase que no implementa una interfaz, podríamos pasar directamente la clase y servir esa clase como un servicio, manejando su tiempo de vida con los métodos antes descriptos, es decir servir clases en vez de interfaces.