

CLASE 3

Web API con Base de Datos

Vamos a construir un web api simple con acceso a datos y a modo de introducción al tema detallaremos los elementos que intervienen al momento de desarrollar un web api con ASP .Net Core.

La idea de este módulo no es profundizar demasiado en conceptos técnicos sino ver las distintas etapas del ciclo de vida del desarrollo de un web api.

Empezaremos creando un web api en Asp .Net Core 2.2 (puede hacerlo en una versión superior), luego configuraremos Entity Framework Core para poder acceder a datos, luego crearemos el primer controlador y aprenderemos a consultar el web utilizando Postman. Y finalmente profundizaremos acerca de datos relacionados.

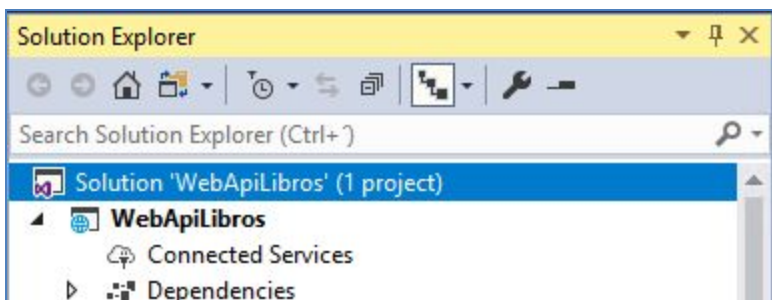
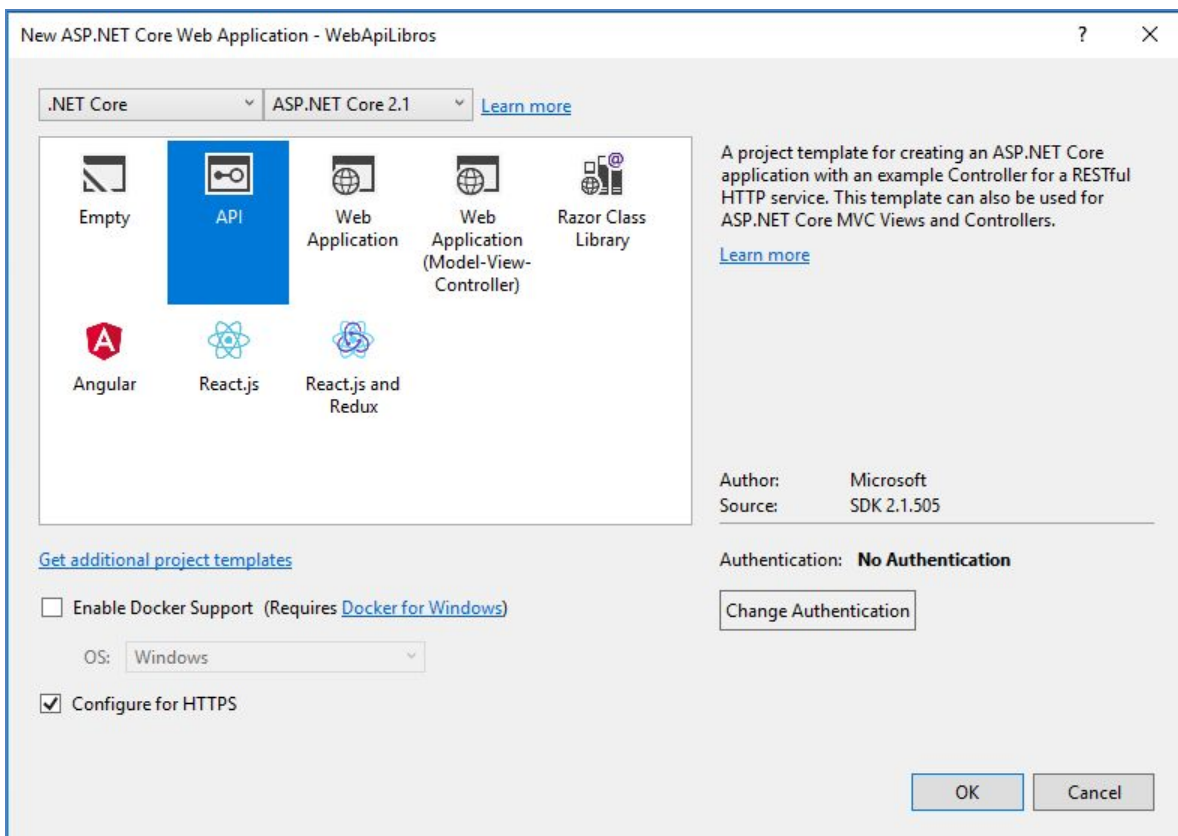
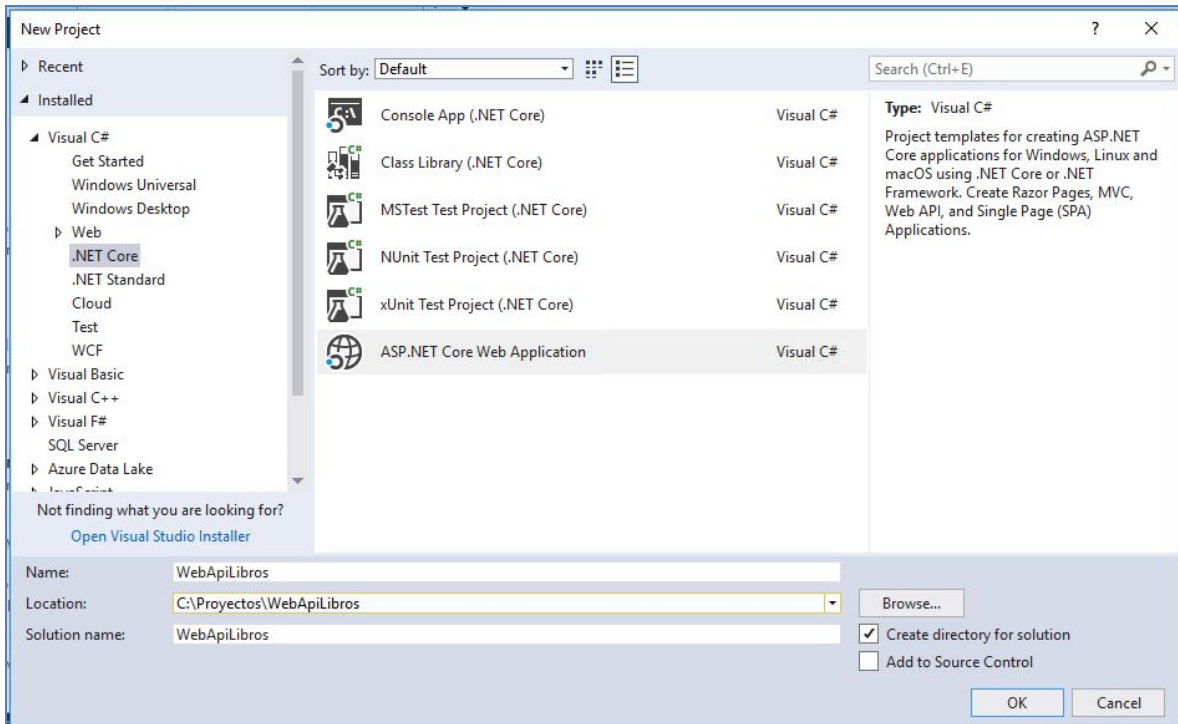
Creación del proyecto Web Api

La idea de este módulo es que sirva como un pequeño vistazo general del flujo de trabajo a seguir a la hora de construir cualquier web API.

Este API no va a ser restful. La intención ahora es aprender a crear un Web API con Asp.Net Core. Veremos acerca de construir un Web API con Asp.Net Core, inyección de dependencias, middleware, controladores, clases, reglas de ruteo, cómo recibir información del Web API, como enviar información al web api, etc.

El web api a realizar se relaciona con un negocio de libros, entre algunas funcionalidades el web API que maneja autores y libros, guardaremos la información relacionada a distintos autores y sus respectivos libros. Podremos ver un listado de todos los autores, ver los libros de un autor determinado. También podremos ingresar nuevos autores y libros hacia la base de datos detrás del Web API.

Vamos a comenzar creando el proyecto desde Visual Studio, desde menú **File – New Project, Asp Net Core Web Application**, indicamos un nombre por ejemplo **WebApiLibros** e indicamos la plantilla de proyecto API:



Esto quiere decir que no se crearán vistas y tendremos un único controlador el cual está optimizado para contener **endpoint de web api**.

Más adelante veremos los aspectos básicos de un proyecto de Asp .Net Core como inyección de dependencias, la clase startup, middleware entre otras cosas.

Nota: Un **endpoint** es un puerto de comunicaciones empleado para realizar llamadas a procedimientos remotos. Contextualizando esta definición a la temática del curso, decimos que endpoints de web api son las URL's que reciben o retornan información de un Web API.

Vamos a ver que tenemos la carpeta de **Controllers** y dentro de ésta tenemos un controlador **ValuesController** con los elementos que vimos en la clase anterior.

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

namespace WebApiLibros.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ValuesController : ControllerBase
    {
        // GET api/values
        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return new string[] { "value1", "value2" };
        }

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

        // POST api/values
        [HttpPost]
        public void Post([FromBody] string value)
        {
        }
    }
}
```

```

        // PUT api/values/5
        [HttpPut("{id}")]
        public void Put(int id, [FromBody] string value)
        {
        }

        // DELETE api/values/5
        [HttpDelete("{id}")]
        public void Delete(int id)
        {
        }
    }
}

```

Como repaso podemos ver que este controlador tiene un método público HTTP GET que significa que este método va a responder a un HTTP GET realizado hacia el **endpoint** indicado como un atributo de la clase.

A través del atributo **Route** estamos diciendo a qué dirección debemos de hacer la petición HTTP para invocar las acciones de este controlador.

Estos métodos públicos que responden a peticiones HTTP son llamados **acciones**.

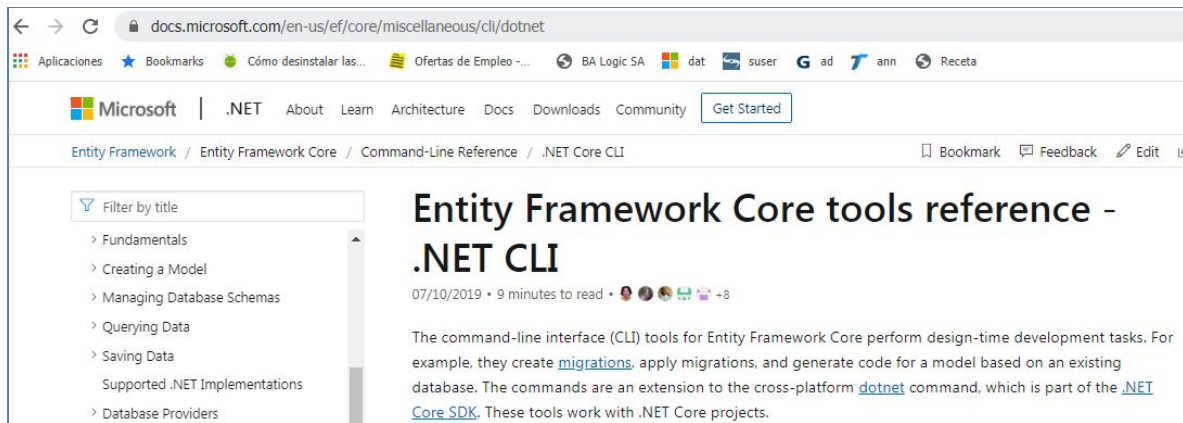
Y finalmente vemos que tenemos `[Route("api/[controller]")]` donde `api/controller` es el prefijo de **Values** Controller es decir **Values** sin la palabra Controller.

.

Instalación de Entity Framework Core

Vamos a instalar Entity Framework Core. Para poder trabajar con **Entity Framework Core 3** en su totalidad necesitamos instalar una herramienta en .Net CLI que nos ayudará a utilizar la interfaz de línea de comandos de Entity Framework por lo cual será muy importante por dos razones: primero para aquellos que utilicen .Net CLI directamente y segundo para cuando vayamos a publicar el web API en Azure vamos a necesitar esta herramienta en **Entity Framework Core 3**. Esto no aplica si utiliza **Entity Framework Core 2.2** (EF Core).

Iremos a esta página <https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet>



Y allí en la sección EF Core 3.x

EF Core 3.x

- `dotnet ef` must be installed as a global or local tool. Most developers will install `dotnet ef` as a global tool with the following command:

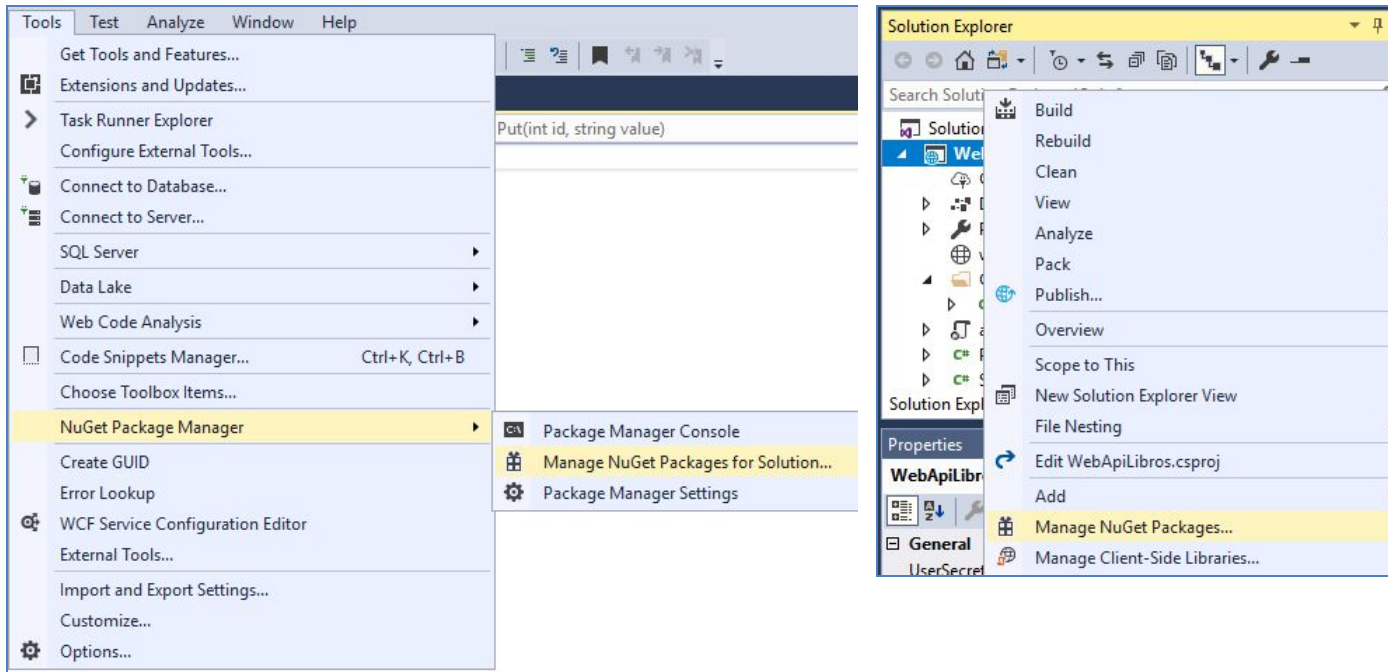
```
.NET Core CLI  
  
dotnet tool install --global dotnet-ef
```

You can also use `dotnet ef` as local tool. To use it as a local tool, restore the dependencies of a project that declares it as a tooling dependency using a [tool manifest file](#).

- Install the [.NET Core SDK 3.0](#). The SDK has to be installed even if you have the latest version of Visual Studio.
- Install the latest `Microsoft.EntityFrameworkCore.Design` package.

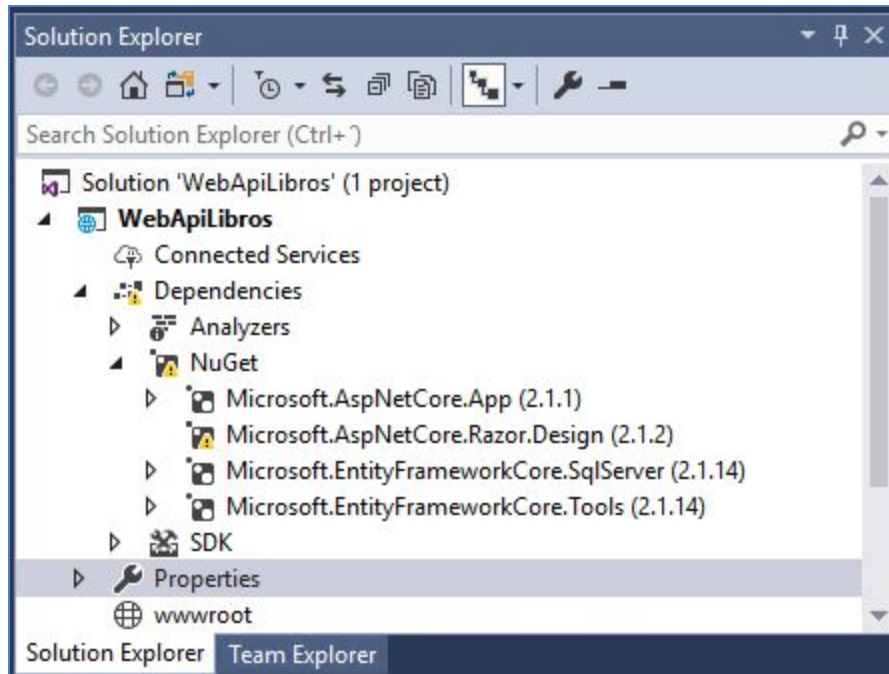
```
.NET Core CLI  
  
dotnet add package Microsoft.EntityFrameworkCore.Design
```

Y copiar el comando **dotnet tool install --global dotnet-ef** para instalar las herramientas del CLI de Entity Framework core Esta instrucción la debe ejecutar como un comando en Power Shell.



Según el motor de bases de datos que usará, deberá instalar el paquete Nuget de EF Core correspondiente (por ejemplo, SqlServer, Oracle, MySql, PostgreSql, etc.). Y según la versión de **Microsoft.AspNetCore.App** y **Microsoft.AspNetCore.Razor.Design** deberá bajar la versión de **Microsoft.EntityFrameworkCore.SqlServer** respectiva.

Y también debemos instalar el paquete **Microsoft.EntityFrameworkCore.Tools** para tener acceso a comandos necesarios entre Asp Net Core y EF Core, por ejemplo, relacionados con las migraciones de EF.



Configuración de Entity Framework Core

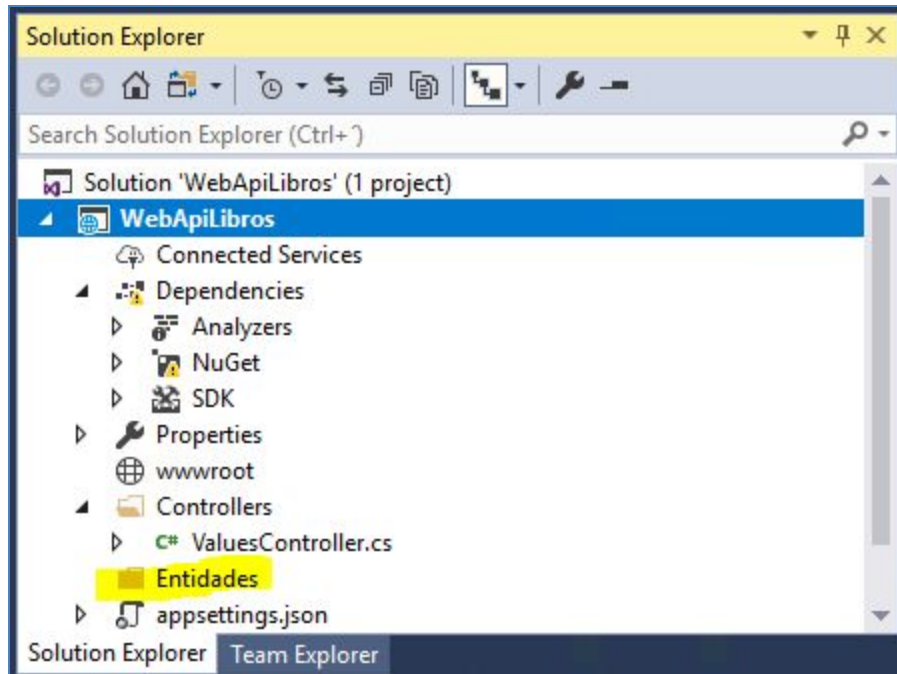
La tarea ahora es crear un controlador que nos va a permitir leer crear, actualizar y buscar información de autores de la base de datos que en breve tendremos.

Creación de una clase del modelo

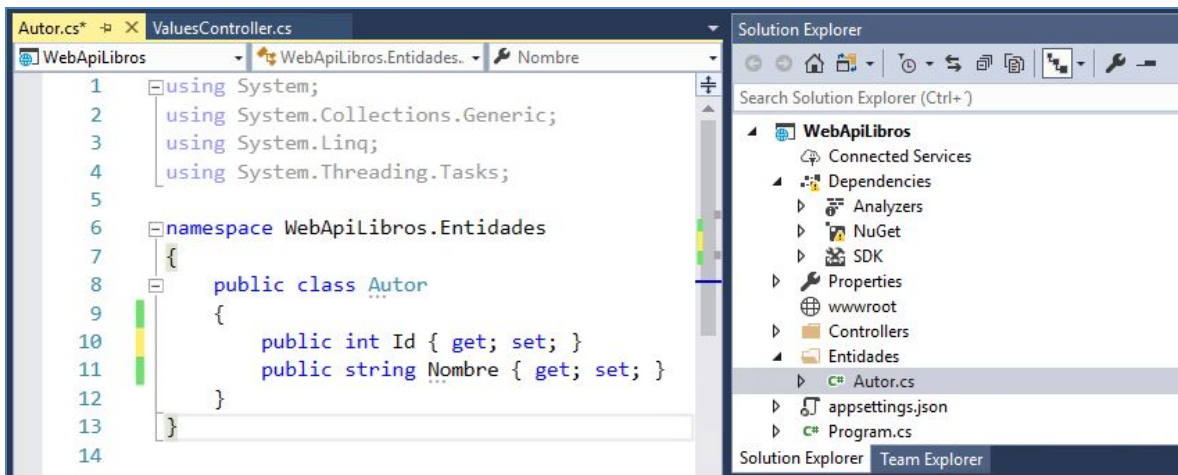
Vamos a empezar creando un nuevo modelo en una clase llamada autor, y organizaremos las clases en carpetas (esto no es obligatorio pero en aplicaciones con gran cantidad de clases de negocio y complejidad, suele hacerse una separación/organización en carpetas en base a algún criterio).

Crearemos una carpeta llamada **Entidades** y allí ubicaremos las entidades que se van a corresponder con las tablas de la base de datos.

Hacemos click con el botón derecho en el proyecto e indicamos **Add New Folder** con el nombre **Entidades**.



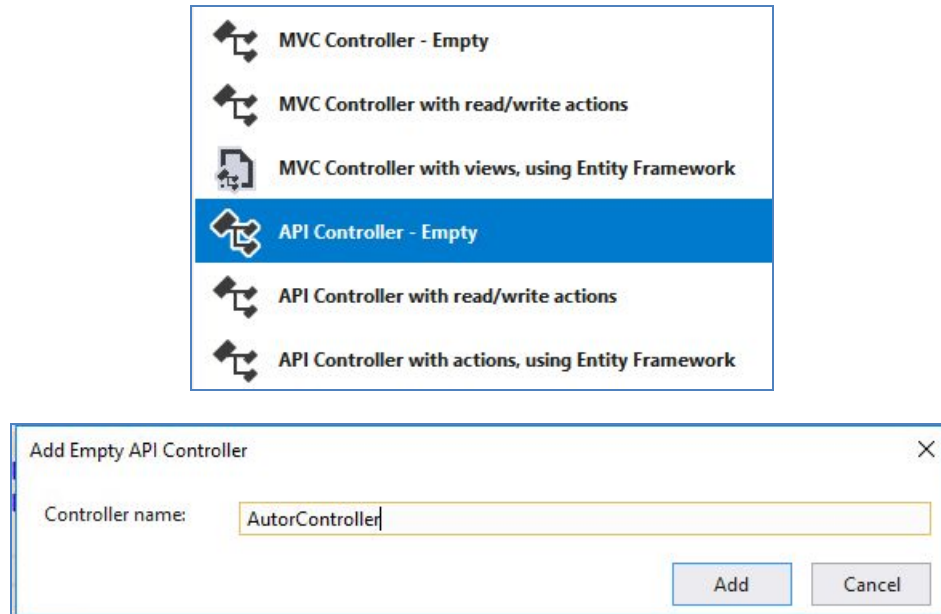
Y dentro de la carpeta Entidades, vamos a crear una clase llamada autor, con dos propiedades Id y Nombre que serán la identificación y el nombre del autor. Para agregar la clase click con el botón derecho en el proyecto e indicamos **Add Class** con el nombre **Autor**.



Creación de una clase controlador

Ahora lo que vamos a hacer es que vamos a crear un controlador en la carpeta controllers. Y lo haremos desde la expresión mínima de un controlador para comprender mejor su función y objetivo.

Click con el botón derecho en la carpeta **Controllers** e indicamos **Add Controllers** y en el cuadro de diálogo seleccionamos **Api Controller Empty** con el nombre **AutorController**.



Se ha creado un controlador para el modelo autor con el siguiente código generado:

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

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

¿Cómo sabemos que la clase es un controlador de un web api?

1. Porque su nombre termina con la palabra **Controller** y porque la clase debe de heredar de una clase base llamada **ControllerBase**.

```
public class AutorController : ControllerBase
```

2. También por el atributo **ApiController** que nos da ciertas facilidades y que fue introducido en Asp .Net Core 2.1 y que trae un conjunto de convenciones que podemos utilizar para simplificar el código de nuestras acciones.

```
[ApiController]
```

3. También tiene definidas reglas de ruteo que se corresponde con este controlador. La idea de colocar controller entre corchetes es que automáticamente el **endpoint** de este controlador va a ser **api/autor**, y es autor porque es el nombre del controlador o por lo menos el prefijo de su nombre.

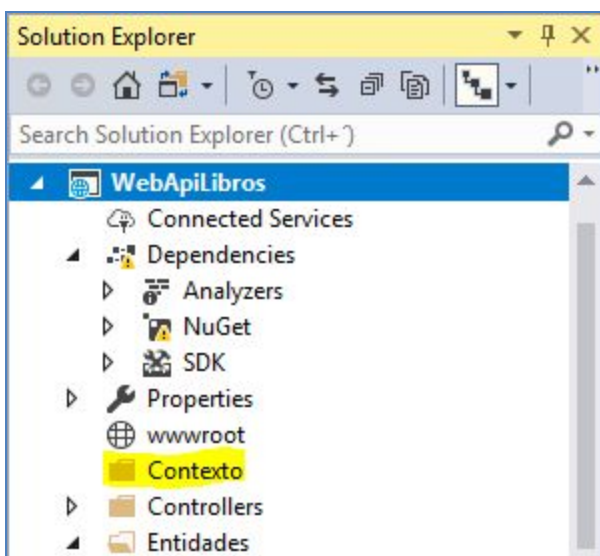
La razón por la cual empezamos todos los endpoints con api/ es más por un tema de convención, no es obligatorio hacerlo de esta forma pero es genérico y universal.

```
[Route("api/[controller]")]
```

Desde este controlador podremos crear, actualizar, borrar y leer información de autores de la base de datos, pero para eso primero necesitamos una base de datos!

Creación de una clase para el contexto de datos

Utilizaremos EF Core 2.2 y **localdb** porque viene por omisión con Visual Studio (puede usar una versión superior de EF según su instalación). Puede usar también otra instancia del motor de base de datos Sql Server que tenga acceso u otro motor de base de datos relacional como mencionamos en la sección anterior donde instalamos EF Core.

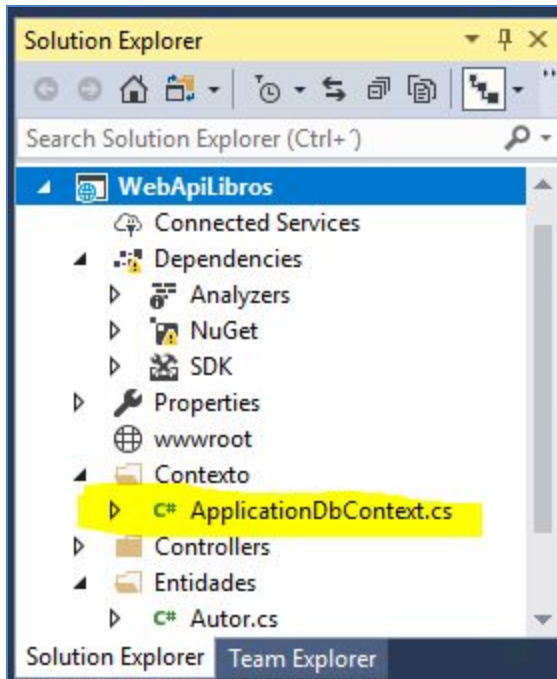


Ahora configuraremos EF Core, y en la aplicación crearemos una nueva carpeta también por un tema de organización, y se llamará **Contexto** representando a las clases nuestro contexto de datos.

Dentro de esta carpeta vamos a crear una clase llamada **ApplicationDbContext** que será

nuestro contexto de datos y heredará de la clase base **DbContext**.

Aquí es donde vamos a configurar las distintas tablas de nuestra base de datos y también podemos configurar Fluent Api, una característica que agrega funcionalidad y simplicidad a EF Core, configuraciones de las relaciones entre los modelos y configuraciones a nivel de campos de las tablas.



```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApiLibros.Contexto
{
    public class
    ApplicationDbContext:DbContext
    {
    }
}
```

Crearemos un constructor al que le vamos a pasar una instancia de `DbContextOptions<ApplicationDbContext>` y a la clase base le vamos a pasar las siguientes opciones. Básicamente esta es la forma estándar de configurar EF en un proyecto de Asp .Net Core.

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApiLibros.Contexto
{
    public class ApplicationDbContext : DbContext
    {
        public
        ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
        base(options)
        {
        }
    }
}
```

```

    }
}

```

Vamos a indicar también que la clase de autor se va a corresponder con una tabla de la base de datos. Para eso agregamos una propiedad del tipo de **dbSet<Autor>** con el nombre **Autores** que será el nombre de la tabla en la base de datos, es decir estamos indicando un “mapeo” entre un conjunto de objetos autor hacia la tabla relacional Autores.

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using WebApiLibros.Entidades;

namespace WebApiLibros.Contexto
{
    public class ApplicationDbContext : DbContext
    {
        public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
base(options)
        {
        }

        public DbSet<Autor> Autores { get; set; }
    }
}

```

Con esto hemos configurado que en nuestra base de datos va a haber una tabla llamada **Autores** cuyo esquema va a ser copiado a partir de las propiedades de la clase **autor** es decir que la tabla autores va a tener dos campos: **Id y Nombre**.

Configuración de la cadena de conexión

Ahora vamos a ir a la clase **startup** para configurar este contexto de datos, y en el método **ConfigureServices** usaremos un método especial que sirve para configurar un contexto de datos.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;

```

```

using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.HttpsPolicy;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using WebApiLibros.Contexto;
using Microsoft.EntityFrameworkCore;

namespace WebApiLibros
{
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add
        // services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
String")));

            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_
1);
        }
    }

```

Es decir agregamos un servicio con el contexto de la aplicación que es la clase ApplicationDbContext que acabamos de crear, y en sus opciones configuramos usar SQL Server. Nosotros usaremos **localdb** pero también es SQL Server pero en la cadena de conexión indicamos que nos conectaremos a una instancia de local con **DefaultConnectionString**

Estamos obteniendo de uno de los proveedores de configuración la cadena de conexión, la idea es no tener la cadena de conexión hardcodeada aquí porque es mala práctica, lo que tenemos en una fuente externa de nuestra aplicación. En el módulo de configuraciones vamos a ver distintos lugares en donde podemos guardar cadenas de conexión o cualquier otro tipo de información que queramos se mantenga segura.

Por ahora utilizaremos el archivo de configuración **appsettings.json**, y aquí indicaremos específicamente la cadena de conexión que mencionamos en el método **ConfigureServices** de la clase **startup**. Indicamos lo siguiente:

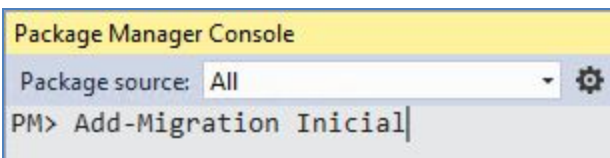
```
{
  "connectionStrings": {
    "DefaultConnectionString": "Data
Source=(localdb)\\mssqllocaldb;Initial Catalog=WebApiLibrosDB;Integrated
Security=True"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

En este archivo json tenemos dos niveles el nivel de **connectionStrings** y el nivel del campo con la conexión **DefaultConnection** donde está la cadena de conexión que apunta a una instancia localdb de MS SQL, el nombre de la base de datos **WebApiLibrosDB**, y respecto de la autenticación con seguridad integrada al sistema operativo Windows.

Creación de una clase para migraciones de EF

Vamos entonces a crear una migración en EF. Una migración es una clase de C# que contiene los cambios que van a ser realizados sobre la base de datos.

La idea es que cuando hacemos cambios a nuestros modelos o a nuestro contexto de datos podemos crear una nueva migración que va a indicar los cambios correspondientes que se harán en la base de datos, para que reflejen los cambios que hicimos en el código C#.

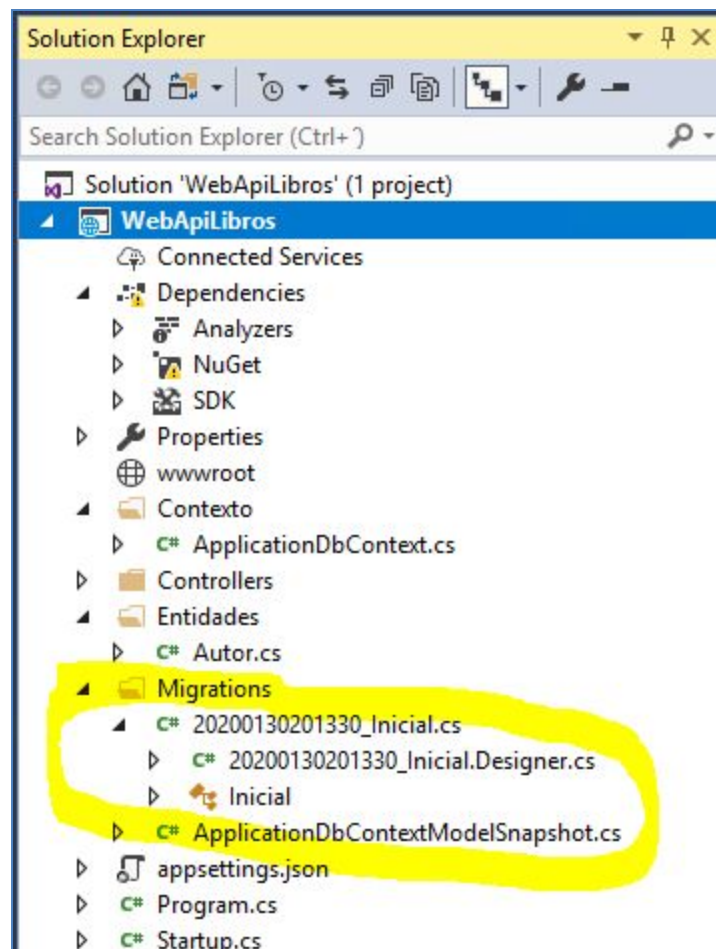


En Visual Studio podemos ir al **Package Manager Console** (lo puede acceder desde **menú Tools – NuGet Package Manager - Package Manager Console**), y para crear una

migración utilizamos el comando **Add-Migration** y le podemos poner un nombre, por ejemplo **Inicial**.

Se ha creado la migración, vemos el resultado de la ejecución del comando y también en el Solution Explorer la carpeta Migrations con las clases de migración:

```
Package Manager Console
Package source: All Default project: WebApiLibros
PM> Add-Migration Inicial
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized 'ApplicationDbContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
To undo this action, use Remove-Migration.
PM>
```



```
using Microsoft.EntityFrameworkCore.Metadata;
using Microsoft.EntityFrameworkCore.Migrations;

namespace WebApiLibros.Migrations
{
    public partial class Inicial : Migration
    {
```

```

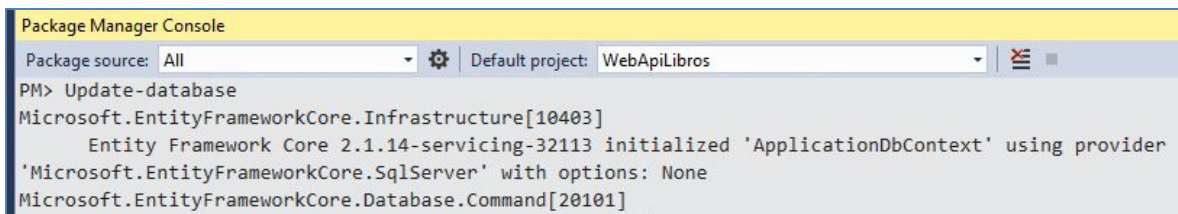
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Autores",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy",
SqlServerValueGenerationStrategy.IdentityColumn),
            Nombre = table.Column<string>(nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Autores", x => x.Id);
        });
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Autores");
}
}
}

```

Se ha incluido una clase llamada **Inicial** que es el nombre de la migración tenemos el método Up que indica que se va a crear una tabla **Autores** con las columnas **Id** y **Nombre** donde Id es un entero el nombre va a ser un string, y también se va a crear una clave primaria **PK_Autores** sobre la columna **Id**.

Vamos ahora a llevar esos cambios que se indican en la migración hacia nuestra base de datos. Dado que no tenemos una base de datos EF Core la va a crear por nosotros, entonces volvemos al Package Manager Console e ingresamos el comando **Update-database**:



```

Package Manager Console
Package source: All | Default project: WebApiLibros
PM> Update-database
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized 'ApplicationDbContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20101]

```

Mostramos el detalle completo de la ejecución de la instrucción **Update-database**:

```

PM> Update-database
Microsoft.EntityFrameworkCore.Infrastructure[10403]

```

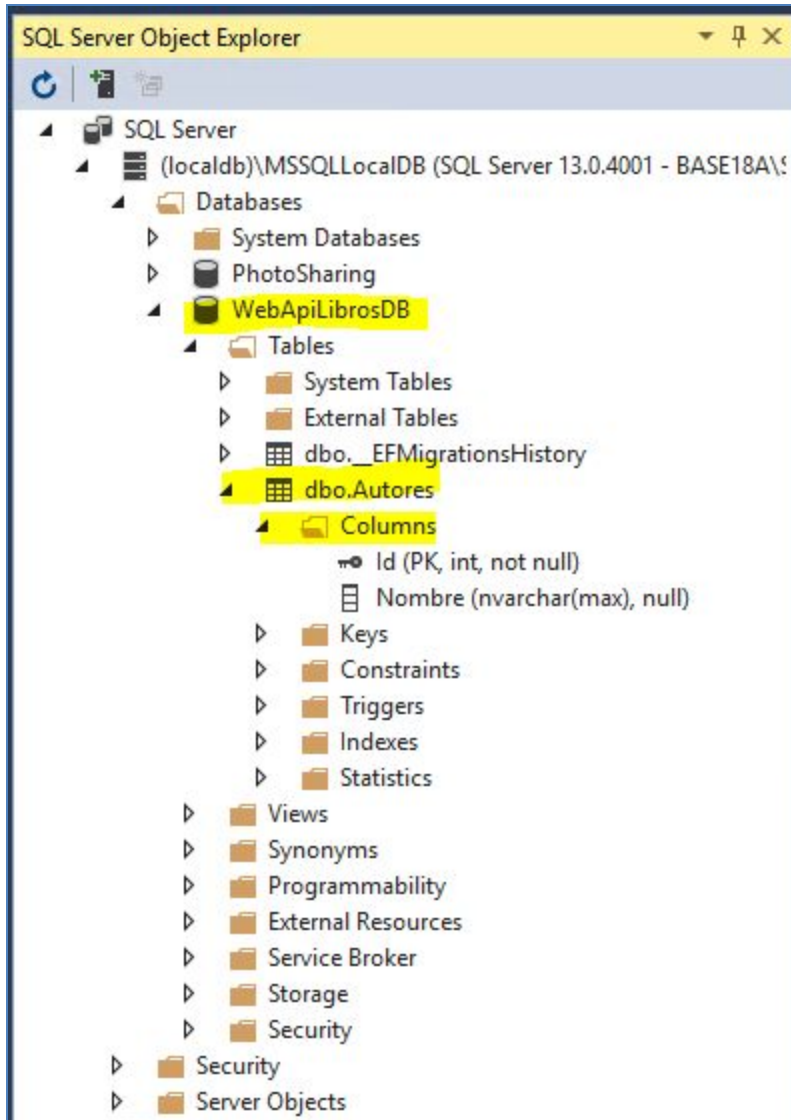
```

Entity Framework Core 2.1.14-servicing-32113 initialized
'ApplicationDbContext' using provider
'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (513ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
    CREATE DATABASE [WebApiLibrosDB];
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (147ms) [Parameters=[], CommandType='Text',
CommandTimeout='60']
    IF SERVERPROPERTY('EngineEdition') <> 5
    BEGIN
        ALTER DATABASE [WebApiLibrosDB] SET READ_COMMITTED_SNAPSHOT ON;
    END;
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (11ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
    CREATE TABLE [__EFMigrationsHistory] (
        [MigrationId] nvarchar(150) NOT NULL,
        [ProductVersion] nvarchar(32) NOT NULL,
        CONSTRAINT [PK__EFMigrationsHistory] PRIMARY KEY
([MigrationId])
    );
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (12ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
    SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (7ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
    SELECT [MigrationId], [ProductVersion]
    FROM [__EFMigrationsHistory]
    ORDER BY [MigrationId];
Applying migration '20200130201330_Inicial'.
Microsoft.EntityFrameworkCore.Migrations[20402]
    Applying migration '20200130201330_Inicial'.
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']
    CREATE TABLE [Autores] (
        [Id] int NOT NULL IDENTITY,
        [Nombre] nvarchar(max) NULL,
        CONSTRAINT [PK_Autores] PRIMARY KEY ([Id])
    );
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (3ms) [Parameters=[], CommandType='Text',
CommandTimeout='30']

```

```
INSERT INTO [__EFMigrationsHistory] ([MigrationId],  
[ProductVersion])  
VALUES (N'20200130201330_Inicial', N'2.1.14-servicing-32113');  
Done.
```

A continuación podemos verificar luego que la base de datos se ha creado desde la opción **menú View – Sql Server Object Explorer – instancia MSSQLLocalDB** y la base de datos **WebApiLibrosDB** con la tabla **Autores**:



Creación de un CRUD para Autores: inserción y selección

Vamos a continuar el desarrollo del controlador **Autor** siguiendo estos pasos:

1. En el controlador, inyectaremos una instancia del **ApplicationDbContext** a través del constructor de la clase **AutorController**. La idea aquí es que podemos utilizar inyección de dependencias para servirle a una clase en todas sus dependencias. Esto es una buena práctica de ingeniería de software dado que eso indica claramente las dependencias de una clase y además permite cambiar las implementaciones de estas dependencias en tiempo de ejecución sin tener que cambiar el código de nuestra clase. Más adelante explicaremos más a fondo la inyección de dependencias, por ahora sólo indicamos este link de introducción al tema <https://desarrolloweb.com/articulos/patron-diseno-contenedor-dependencias.html>

Por ahora vamos a inyectar el **ApplicationDbContext** en la clase **AutorController** y vamos a utilizar el constructor de la clase pasándole como parámetro formal un valor del tipo **ApplicationDbContext** e inicializando el campo contexto definido localmente y de solo lectura en la clase, con el valor del parámetro pasado, este es el código resultante:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using WebApiLibros.Contexto;

namespace WebApiLibros.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class AutorController : ControllerBase
    {
        private readonly ApplicationDbContext context;

        public AutorController(ApplicationDbContext context)
        {
            this.context = context;
        }
    }
}
```

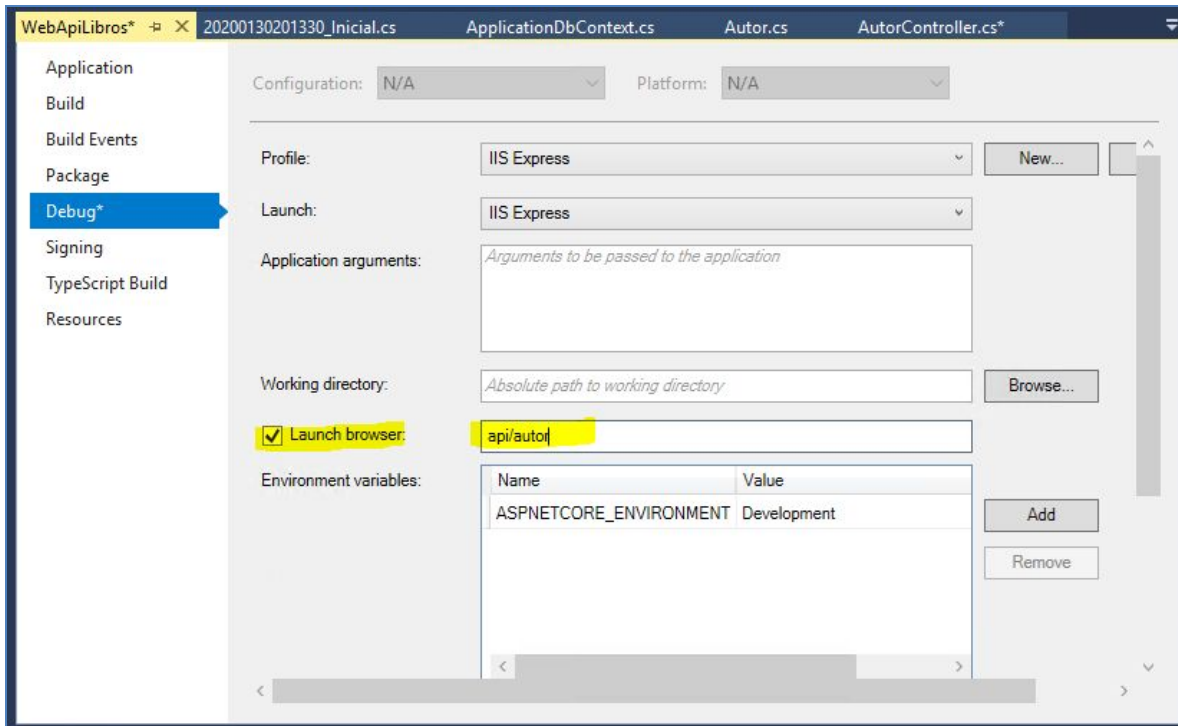
Ya con esto estamos inyectando una instancia de `ApplicationDbContext` en el controlador y podemos utilizar una instancia de EF.

2. Vamos a implementar ahora la primer acción que responderá a un HTTP GET. Se llamará **Get**, retornará un **IEnumerable** de objetos tipo **Autor**, y en el cuerpo de la acción o de la función indicamos que desde el campo **context** que tiene el **dbSet Autores** retorne su lista de objetos **Autor**. Esta función se va ejecutar cuando hagamos un **Get** en el endpoint **api/Autor** y traerá un listado de autores de la base de datos. No olvide agregar la referencia al namespace **WebApiLibros.Entidades**.

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

Sin embargo debemos destacar que no es correcto utilizar funciones sincrónicas para comunicarse con la base de datos desde un web API, la buena práctica es utilizar programación asincrónica, pero por ahora vamos a mantener el ejemplo simple sólo con programación sincrónica.

3. Podemos ejecutar el web api para ver hasta aquí como funciona, pero antes modificaremos en las propiedades del proyecto, la propiedad **Launch browser** con el valor **api/autor**. Es decir cuando ejecutemos la aplicación por omisión solicitará la api autor, así evitamos cambiar manualmente la URL en el navegador.



Pulsamos <CTRL+F5> para ejecutar la aplicación y veremos que carga el navegador en la URL deseada, y vemos que retorna una lista vacía porque todavía no agregamos datos en la base de datos en la tabla de autores.

El tipo de dato de retorno de la acción **Get**, básicamente indica que existen dos cosas que podemos retornar de este método, ya sea un **ActionResult** o un **IEnumerable** de la clase **Autor**, es decir una colección de autores como una lista o un arreglo de autores. Cuando hablamos de **ActionResult** nos referimos a las posibles respuestas que una acción puede retornar; un ejemplo típico son los códigos de estado por ejemplo 404 no encontrado. Pero cuando los recursos se han retornado vamos a querer determinar el recurso por eso indicamos como tipo de datos de la acción un **ActionResult entidad** donde tenemos dos posibles resultados: "ActionResult y la entidad que queremos retornar".

4. Agregaremos otro **Get** que reciba como parámetro el Id de un autor específico que buscaremos en la base de datos a través de EF y para retornarlo:

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

    if (resultado == null)
```



```

        { return NotFound(); }

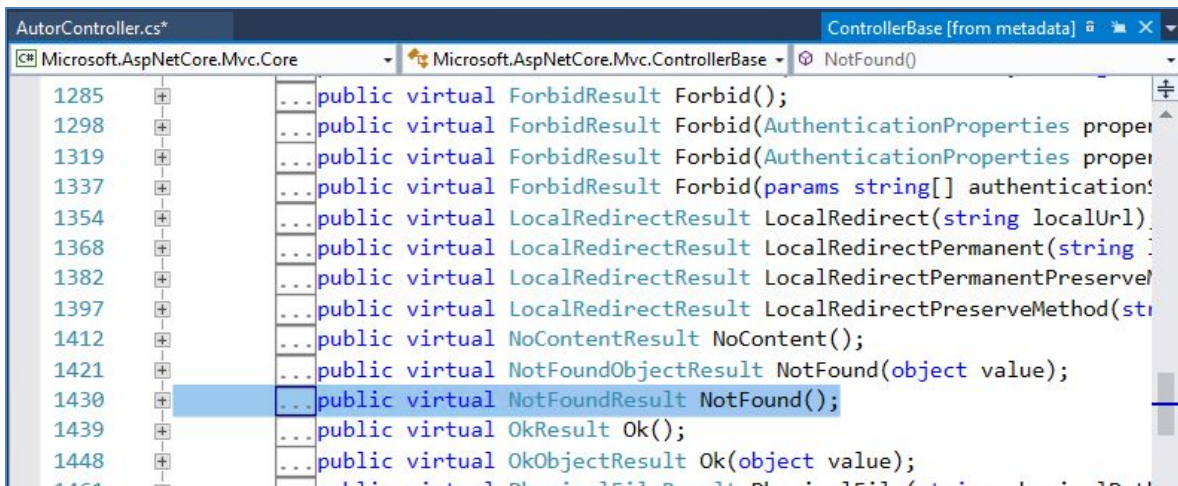
        return resultado;
    }

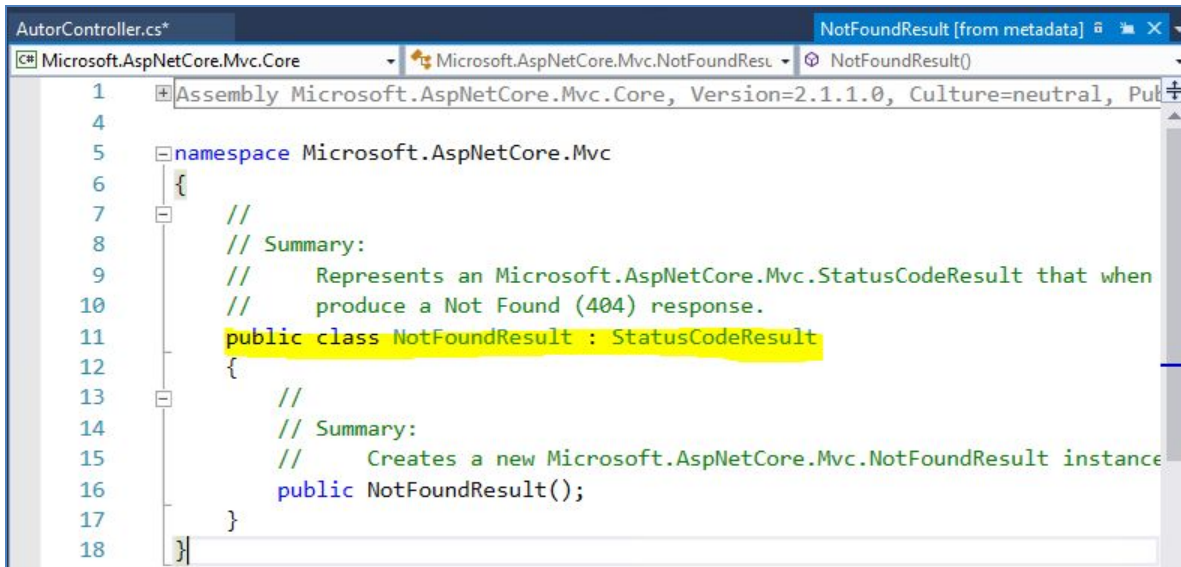
```

Entonces decoramos la acción **Get** con el atributo `[HttpGet("{id}")]` dado que recibirá como parámetro el **Id** y en la URL se pasará su valor; el método **Get** espera ese parámetro **Id** de tipo entero; en el cuerpo del método definimos una variable con tipado implícito llamada **resultado** que recibirá del contexto de EF del dbset **Autores**, la búsqueda de un autor cuyo **Id** sea igual al valor del parámetro recibido (observar la expresión lambda `x => x.Id == id`, y si ese autor es encontrado **ActionResult** será una instancia del objeto **Autor** y en caso contrario retornará un estado 404 not found a través del método `NotFound()`.

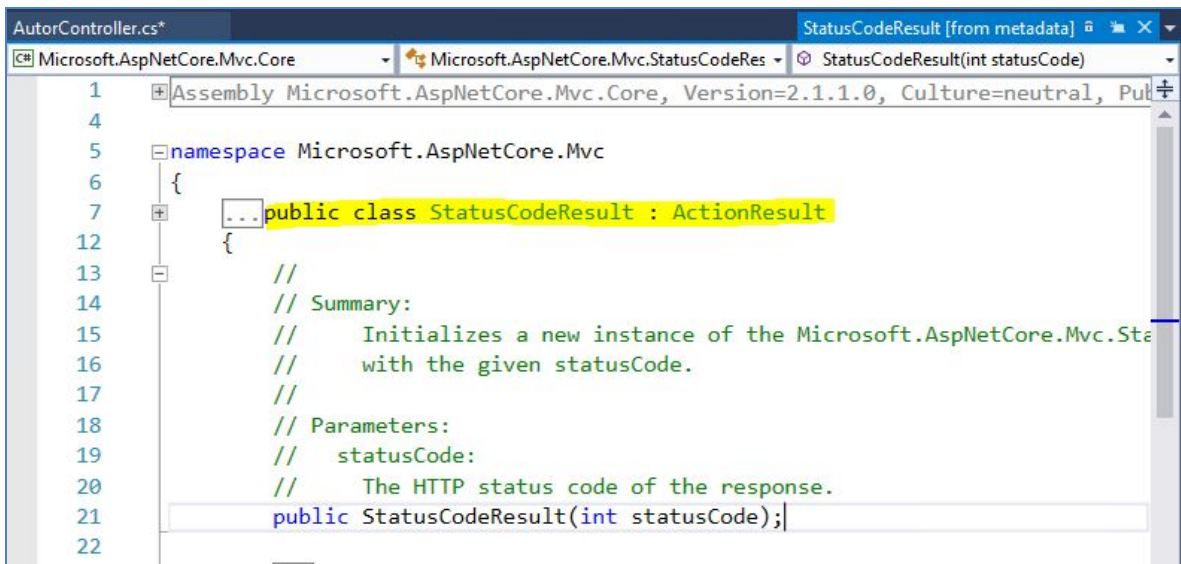
¿Cómo sabemos que el tipo de dato de retorno del método **NotFound()** es **ActionResult**? Si no fuera de ese tipo estaría ocurriendo un error en la escritura del código.

Si ubicados con el cursor en el método **NotFound()** pulsamos la tecla <F12> vamos a la definición del método y vemos que retorna el tipo **NotFoundResult**, y si pulsamos <F12> nuevamente sobre **NotFoundResult**, vemos que es una clase derivado que hereda de la clase base **StatusCodeResult**, y si pulsamos <F12> nuevamente sobre la clase derivada **StatusCodeResult** vemos que hereda de la clase base **ActionResult** que es el tipo de dato que indicamos en la acción del controlador.





```
1  Assembly Microsoft.AspNetCore.Mvc.Core, Version=2.1.1.0, Culture=neutral, Pub
4
5  namespace Microsoft.AspNetCore.Mvc
6  {
7      //
8      // Summary:
9      //     Represents an Microsoft.AspNetCore.Mvc.StatusCodeResult that when
10     //     produce a Not Found (404) response.
11     public class NotFoundResult : StatusCodeResult
12     {
13         //
14         // Summary:
15         //     Creates a new Microsoft.AspNetCore.Mvc.NotFoundResult instance
16         public NotFoundResult();
17     }
18 }
```

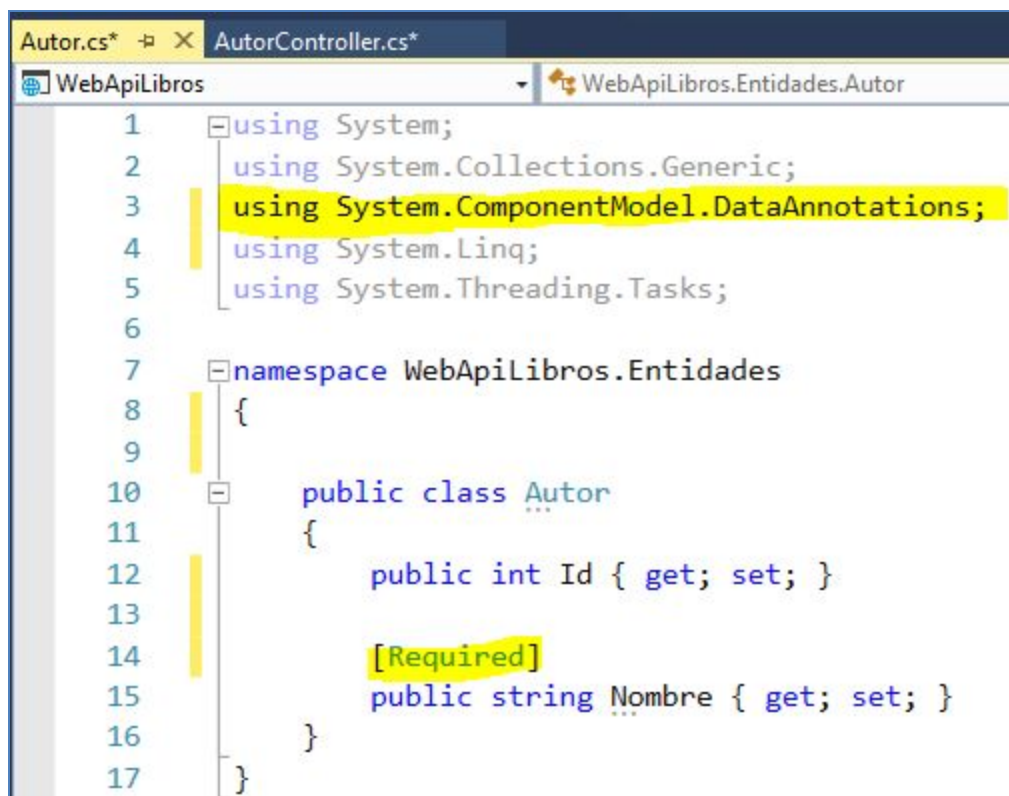


```
1  Assembly Microsoft.AspNetCore.Mvc.Core, Version=2.1.1.0, Culture=neutral, Pub
4
5  namespace Microsoft.AspNetCore.Mvc
6  {
7      ... public class StatusCodeResult : ActionResult
12     {
13         //
14         // Summary:
15         //     Initializes a new instance of the Microsoft.AspNetCore.Mvc.Sta
16         //     with the given statusCode.
17         //
18         // Parameters:
19         //     statusCode:
20         //         The HTTP status code of the response.
21         public StatusCodeResult(int statusCode);
22     }
```

5. Además de los GET podemos implementar las acciones que se ejecutarán cuando con otros métodos HTTP como POST, agregaremos uno que reciba un autor para insertarlo en la base de datos. Ese autor cuando se manda por un post va a venir en el cuerpo de la petición HTTP tal cual vimos, por eso vamos a utilizar el atributo **FromBody** para indicarle a Asp.Net Core que debe de buscar la información del autor en el cuerpo de la petición HTTP.

Si ya antes ha utilizado ASP.Net Framework o ASP.Net Core, por ejemplo en proyectos MVC tradicionales, es posible que habitualmente haya usado una llamada a **ModelState.IsValid** que es un código repetitivo para asegurarnos que el modelo es válido, y si no es válido retornar un bad request. Recordamos que el atributo **[ApiController]** que tenemos a nivel de clase ayuda a eliminar el código repetitivo.

La idea es que los modelos en general tienen *reglas de validación* que forman la lógica de negocio de la aplicación. Por ejemplo si vamos a la clase **Autor** podemos usar el atributo **[Required]** sobre la propiedad **Nombre** para que sea requerida, es decir si un cliente envía un autor sin nombre ese modelo será inválido porque no ha respetado todas las reglas de validación, y deberá avisar que el modelo es inválido.



```
Autor.cs* X AutorController.cs*
WebApiLibros
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.DataAnnotations;
4 using System.Linq;
5 using System.Threading.Tasks;
6
7 namespace WebApiLibros.Entidades
8 {
9
10     public class Autor
11     {
12         public int Id { get; set; }
13
14         [Required]
15         public string Nombre { get; set; }
16     }
17 }
```

```
[HttpPost]
public ActionResult Post([FromBody] Autor autor)
{
    if (!ModelState.IsValid)
    { return BadRequest(ModelState); }
}
```

Reiteramos, desde ASP.Net Core versión 2.1 en adelante esto no hace falta porque el api controller se encarga de verificar si el modelo es válido y si no es válido retornará un bad request con el model state.

Continuaremos entonces con el objetivo de agregar un autor a la base de datos. Detallamos el código completo de la acción **Post**, con el agregado de una ruta en la acción **Get**:

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

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

        return resultado;
}

[HttpPost]
public ActionResult Post([FromBody] Autor autor)
{
    context.Autores.Add(autor);
    context.SaveChanges();
    return new CreatedAtRouteResult("ObtenerAutor", new { id =
    autor.Id }, autor);
}
```

En la acción **Post** utilizamos la función **Add** de **Autores**, agregarnos un **context.SaveChanges()** y retornamos un 201 created a través de la instanciación del objeto **CreatedAtRouteResult** con una ruta en donde el cliente va a poder localizar el recurso, es decir que en la acción **Get** agregaremos el nombre de la ruta pasándole valor al parámetro **Name="ObtenerAutor"** que será la ruta, y luego la mencionamos en la acción **Post**.

Recordamos que una de las convenciones de HTTP POST es que si un recurso es creado debemos retornar una cabecera **location** en donde se coloque la ubicación del recurso. Asp.Net Core facilita esto con el objeto **CreatedAtRouteResult** pasándole el nombre de la ruta donde se va a encontrar el recurso y también los parámetros que la acción **Get** espera **id=autor.Id** y finalmente en el cuerpo de la respuesta HTTP pasar el autor, y con eso estamos respetando el estándar del HTTP POST.

En breve probaremos este método utilizando **Postman**.

Herramienta Postman ¿Qué es?

Vamos a utilizar Postman pero antes introducimos el objetivo de la herramienta.

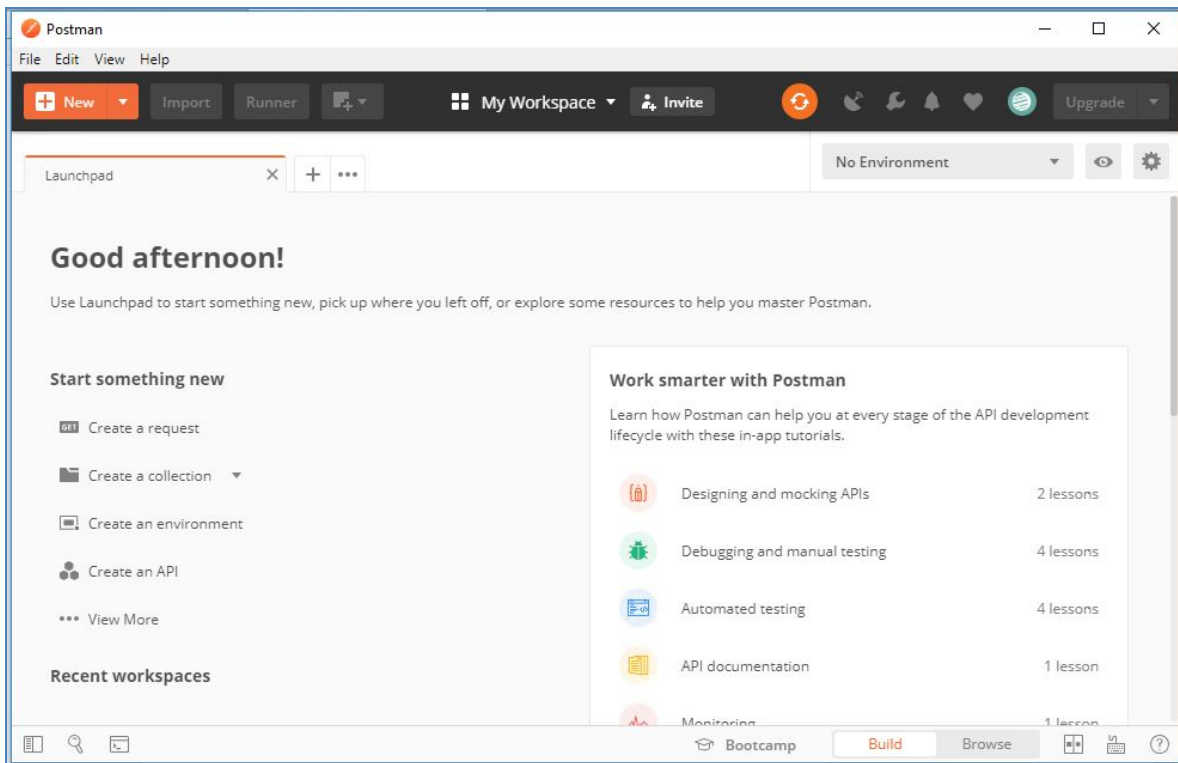
Postman es una herramienta que se utiliza, sobre todo, para el testing de API REST, aunque también admite otras funcionalidades que se salen de lo que engloba el testing de este tipo de aplicaciones.

Además de testear, consumir y depurar API REST, podremos monitorizarlas, escribir pruebas automatizadas para ellas, documentarlas, mockearlas, simularlas, etc.

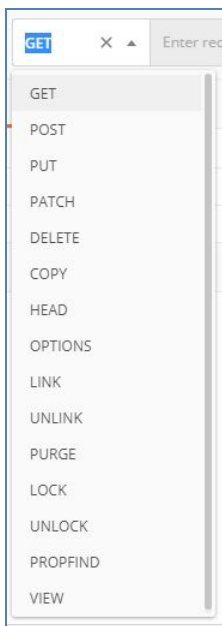
Es una de las herramientas más utilizadas para hacer testing exploratorio de este tipo de aplicaciones. Es importante destacar también que, aunque no sea una de las herramientas más famosas para documentar API REST, genera una documentación bastante buena e interesante, con ejemplos y snippets de código, de forma que hace que sea muy fácil de entender cómo funciona una API determinada.

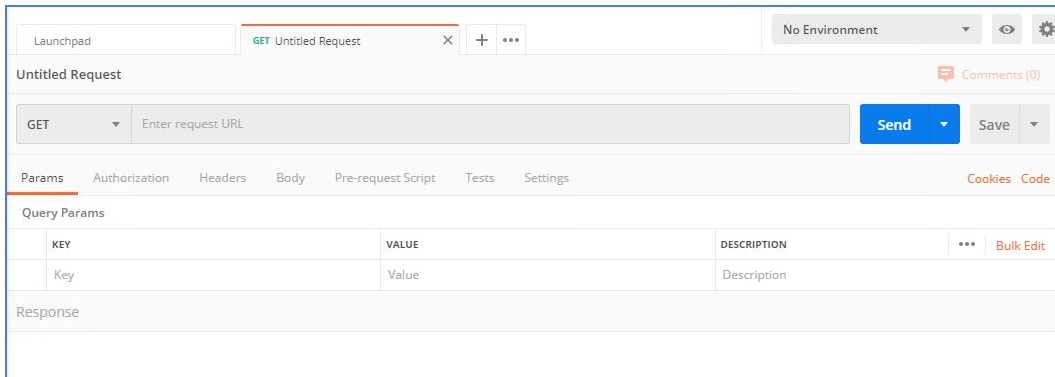
Con Postman podemos hacer peticiones HTTP hacia una URL, en el ejemplo que armamos podemos hacer un post al endpoint **/api/autor** para por ejemplo, crear un registro en la base de datos.

Debemos descargar e instalar Postman desde este link <https://www.postman.com/>, la aplicación es gratuita. Asegúrese de instalar la versión que corresponda al sistema operativo que usted tiene.

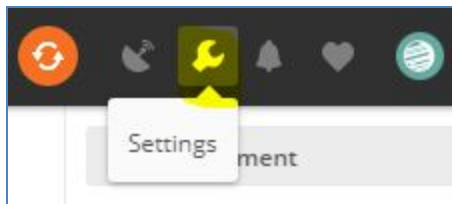


En Postman y desde una ventana de Request, podemos probar la api.



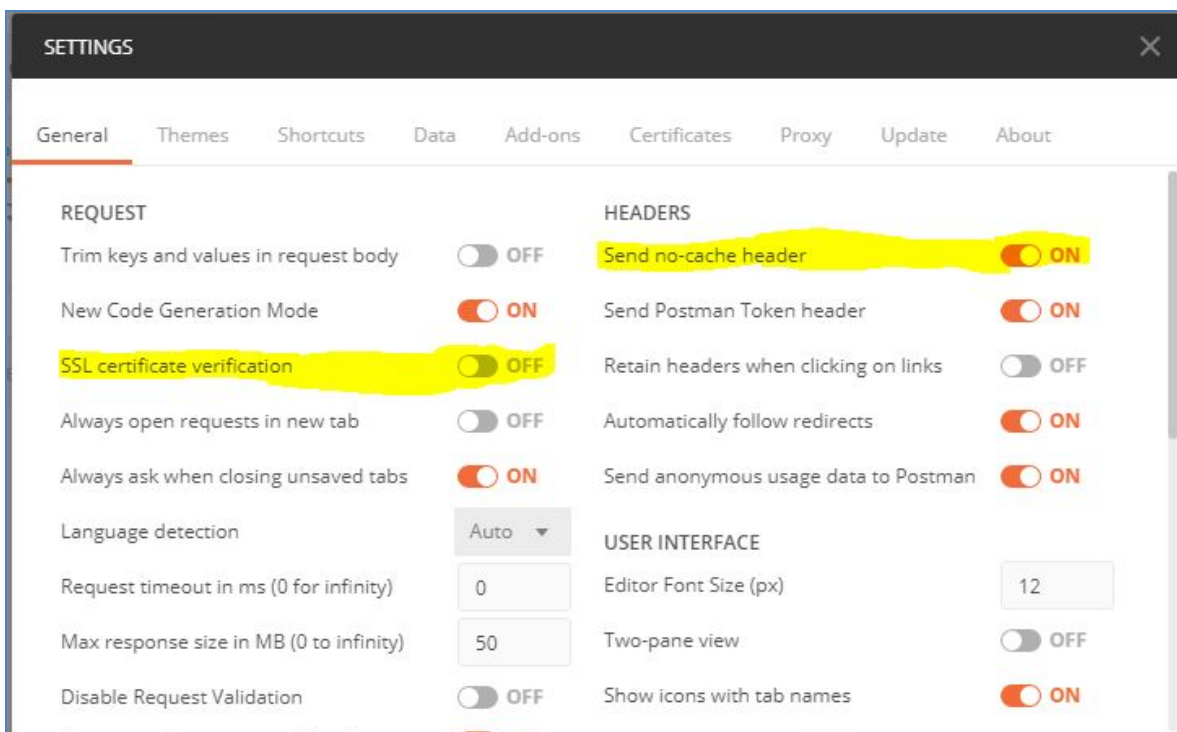


Podemos escribir la URL a probar, indicar distintos métodos HTTP a usar y entre todas las solapas distintos usos de la api que se pueden tener.

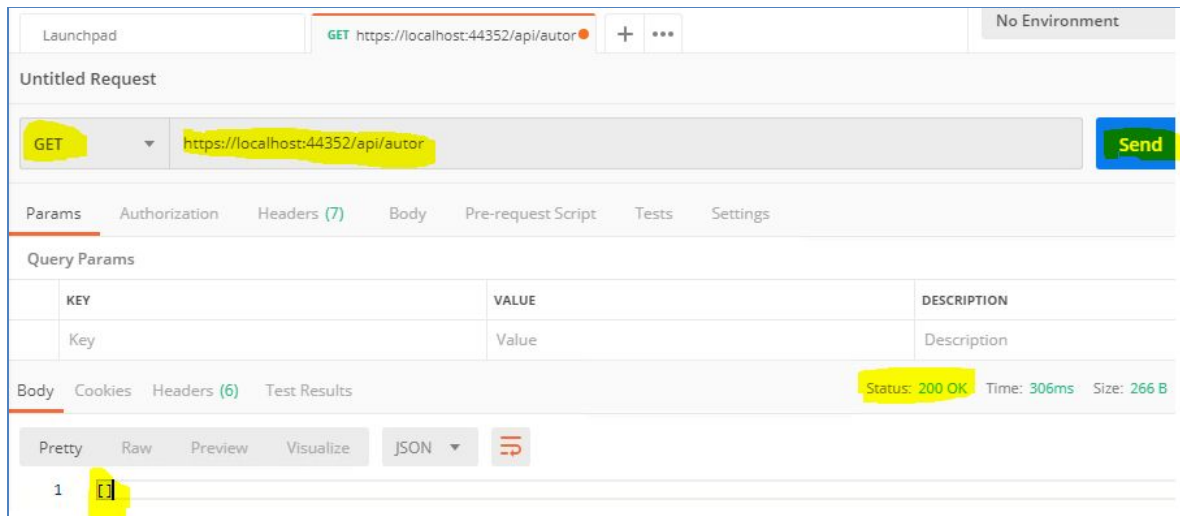


Antes de utilizar Postman necesitamos hacer una configuración desde el icono **Settings** y en la opción **Settings**, en la sección **Headers** la opción **Set no-cache header** activado y en la sección **Request** la opción **SSL certificate verification** desactivado porque los certificados de HTTPS que ahora

estamos usando para los ejemplos no son válidos y no queremos que Postman de errores de certificado.

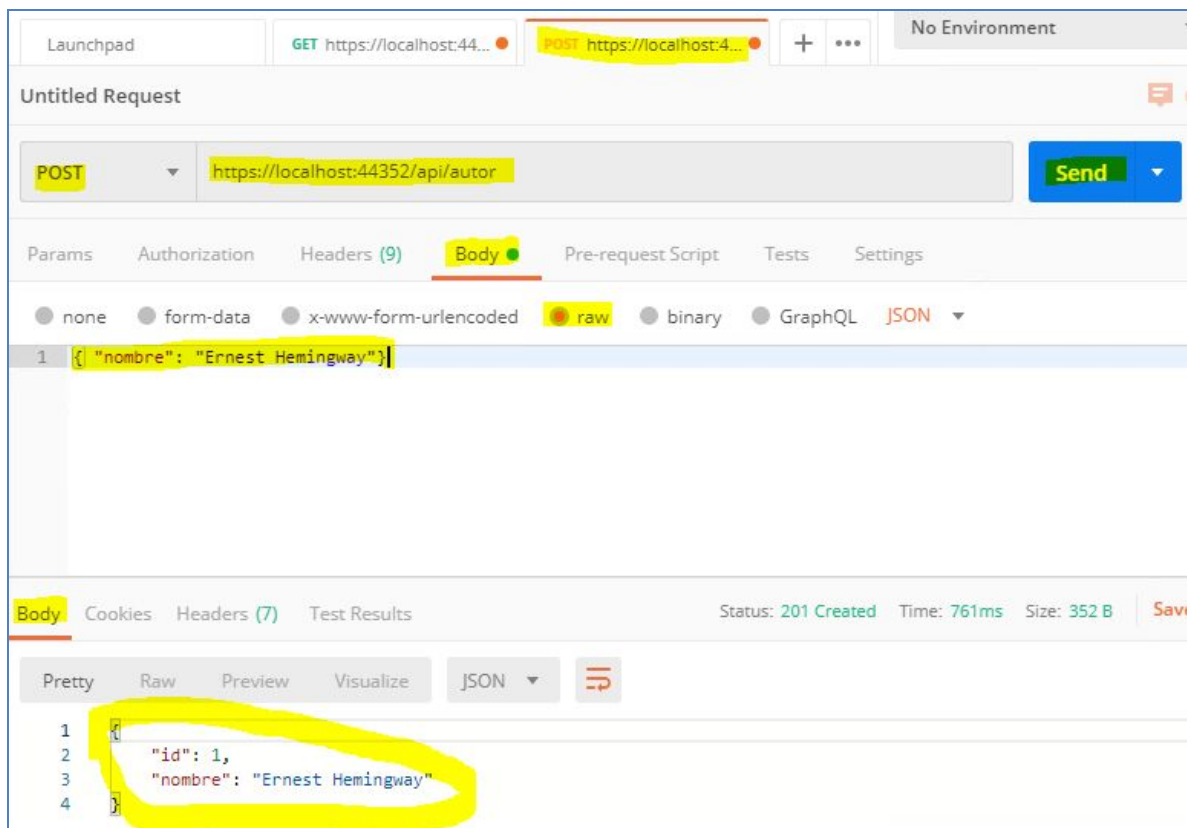


Una vez hecha la configuración, volvemos para ingresar la URL del endpoint a probar **https://localhost:44352/api/autor** con el método **GET** y luego click en **Send**, recuerde que la acción Get por ahora no retorna autores, solo un arreglo vacíos con **status 200 OK**.



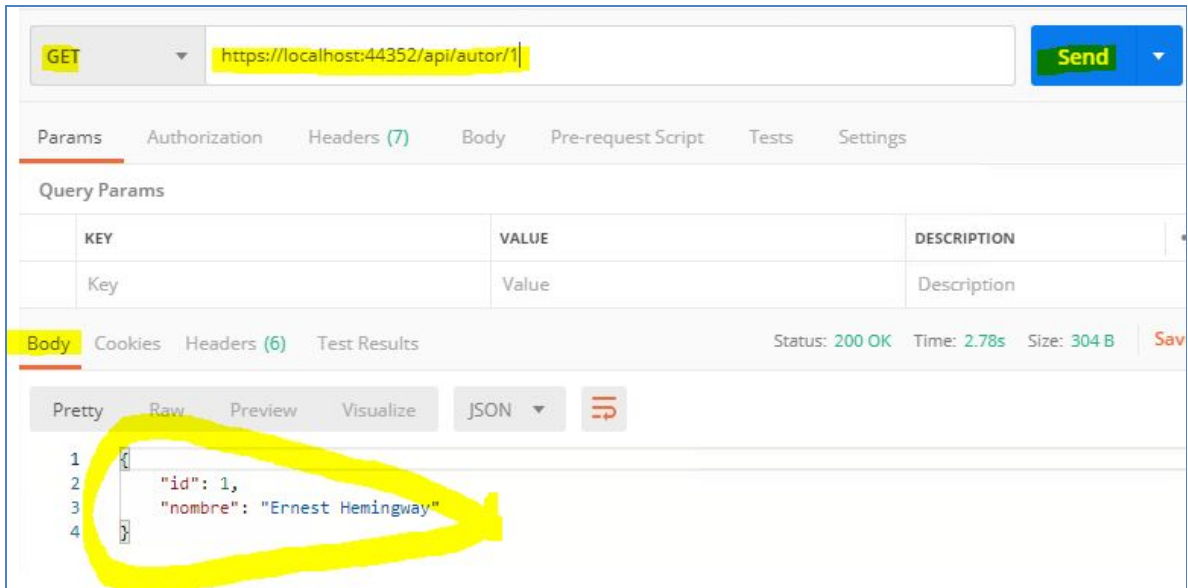
Ahora estando funcionando Postman probaremos la acción **POST** con un nuevo endpoint en una nueva pestaña, así tenemos distintas pestañas para probar distintos endpoints y la prueba es más ágil.

Deberá indicar que el método a probar es **POST**, y en la sección **Body – raw**, indicar que los datos se enviarán en formato **JSON**, e indicar el valor de la propiedad nombre de la clase **Autor** del modelo que definimos y hacer click en **Send**. Si todo funciona correctamente debería obtener un status **201 Created**, y en el cuerpo de la petición el autor agregado con el valor de la propiedad **Id**.

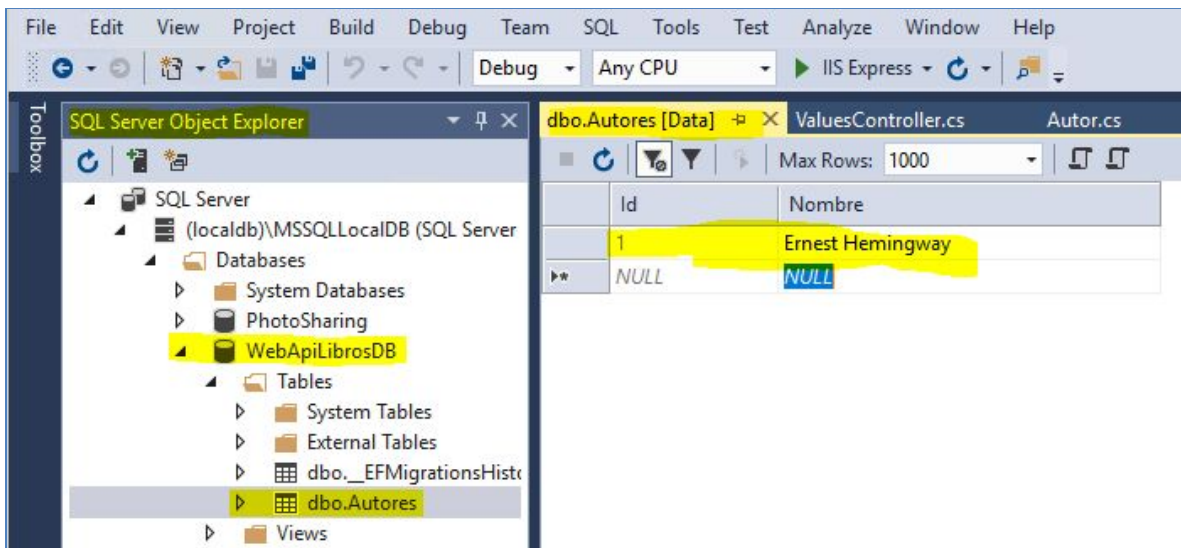


Y si vemos la cabeceras tenemos **Location** que apunta a la URL <https://localhost:44352/api/Autor/1>

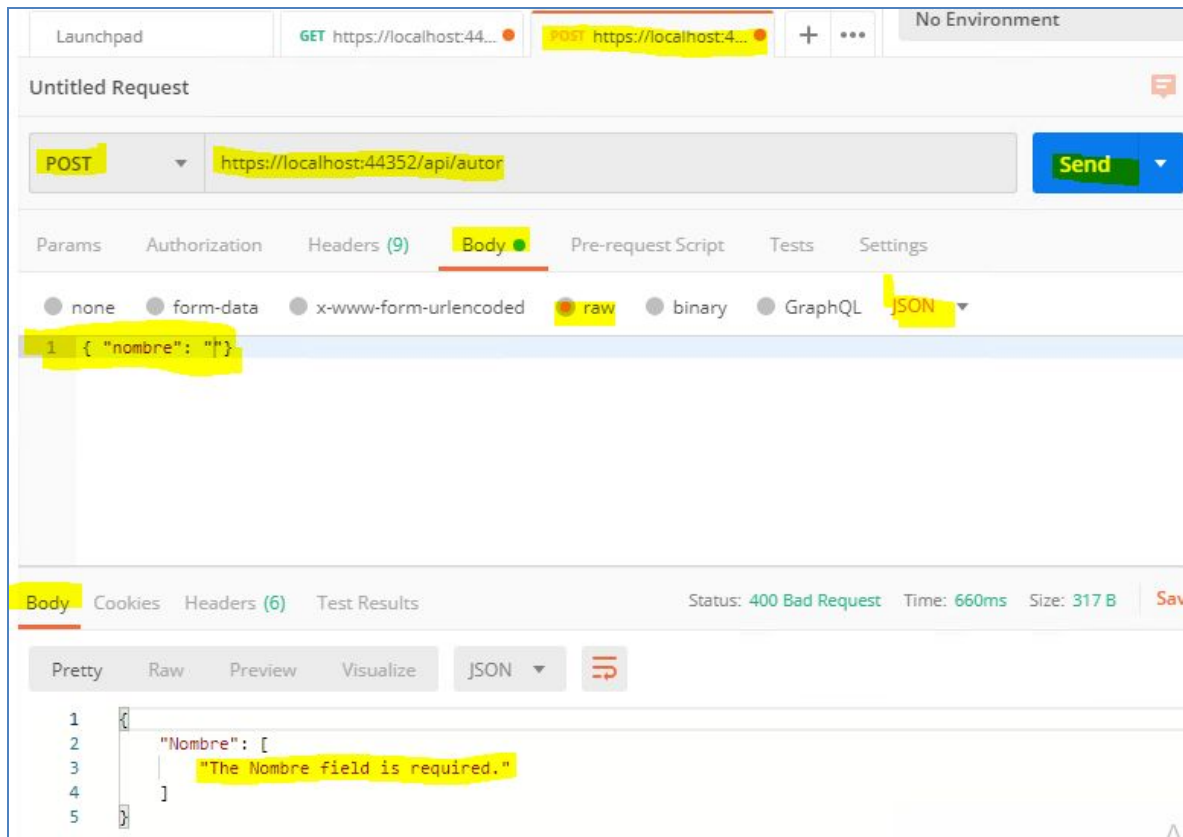
Body		Cookies	Headers (7)	Test Results	Status: 201 Created	Time: 761ms	Size: 352 B	Save Res
	KEY			VALUE				
	Transfer-Encoding			chunked				
	Content-Type			application/json; charset=utf-8				
	Location			https://localhost:44352/api/Autor/1				
	Server			Kestrel				
	X-SourceFiles			=?UTF-8?B?QzpcUHJveWVjdG9zXFdlYkFwaUxpYnJvc1xXZWJBcGIMaWJyb3NcYXBpXGF1dG9y?=<tr><td>X-Powered-By</td><td>ASP.NET</td></tr><tr><td>Date</td><td>Mon, 10 Feb 2020 18:56:23 GMT</td></tr></table>				



Y en la base de datos figura el registro:



Para cerrar este tema, habíamos mencionado que tenemos reglas de validación sobre el modelo **Autor**, habíamos agregado un atributo **Required** para la propiedad **Nombre**, es decir que si tratamos de hacer un **POST** con un nombre sin ningún valor, retornará un error de campo requerido.



Agregando funcionalidad al CRUD: actualización y eliminación

Analizaremos el siguiente código para las acciones **PUT** y **DELETE**:

```
[HttpPut("{id}")]
public ActionResult Put(int id, [FromBody] Autor value)
{
    if (id != value.Id)
    {
        BadRequest();
    }

    context.Entry(value).State = EntityState.Modified;
    context.SaveChanges();
    return Ok();
}
```

En la implementación del **PUT** es donde vamos a actualizar un autor. Los parámetros son **id** que viene de la URL y el **autor** que viene del cuerpo. Lo primero que hacemos es validar y aseguramos que el **id** de la URL y el **id** del cuerpo del autor sean iguales. Si no lo son entonces retornamos un **BadRequest()**. Si los valores son iguales,

utilizamos EF para actualizar el registro, lo marcamos como modificado y guardamos los cambios y finalmente retornamos un **200 Ok**.

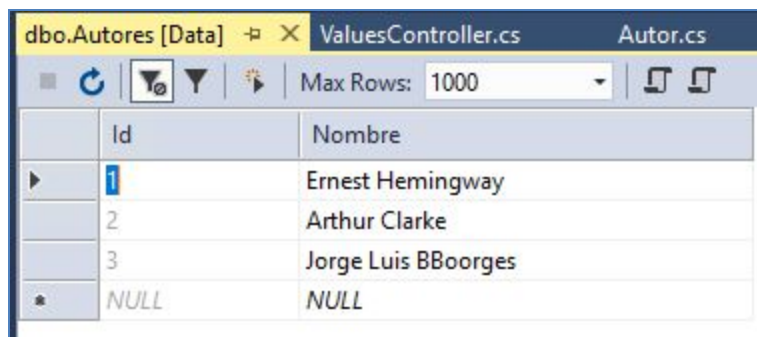
```
[HttpDelete("{id}")]
public ActionResult<Autor> Delete(int id)
{
    var resultado = context.Autores.FirstOrDefault(x => x.Id ==
id);

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

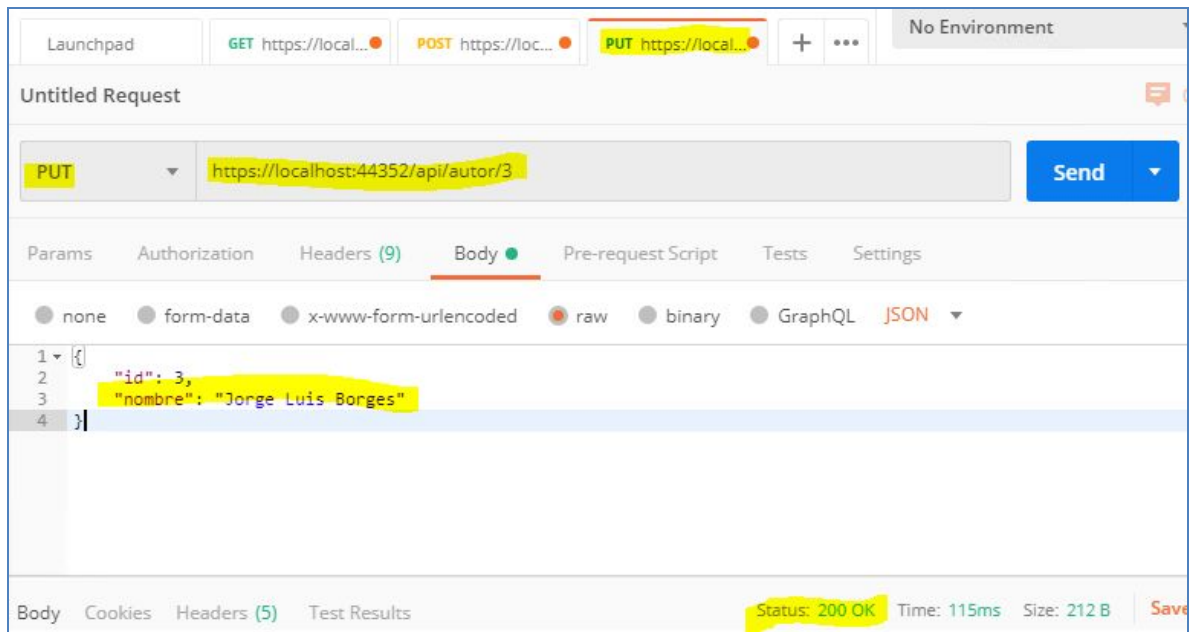
    context.Autores.Remove(resultado);
    context.SaveChanges();
    return resultado;
}
```

Y con la acción DELETE, recibe un id como parámetro del autor a eliminar que viene de la URL, y lo primero que hacemos es buscar el autor en la base de datos y si el autor es nulo retornamos un 404, es decir que lo que deseamos eliminar no existe en la base de datos, caso contrario usamos el método Remove para eliminarlo y guardamos los cambios. Finalmente retornamos el autor eliminado para que la interfaz visual lo use a nivel informativo.

Podemos probar ambos métodos con Postman agregando una solapa para cada acción. Observe primero la modificación que haremos con la acción PUT en el apellido del autor que está mal escrito:



Id	Nombre
1	Ernest Hemingway
2	Arthur Clarke
3	Jorge Luis BBoorges
NULL	NULL



dbo.Autores [Data] X ValuesController.cs		
Max Rows: 1000		
	Id	Nombre
▶	1	Ernest Hemingway
	2	Arthur Clarke
	3	Jorge Luis Borges
★	NULL	NULL

Y por último eliminaremos un autor con DELETE:

Launchpad GET https://localh... POST https://localh... PUT https://localh... DEL https://localh...

Untitled Request

DELETE https://localhost:44352/api/autor/1

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

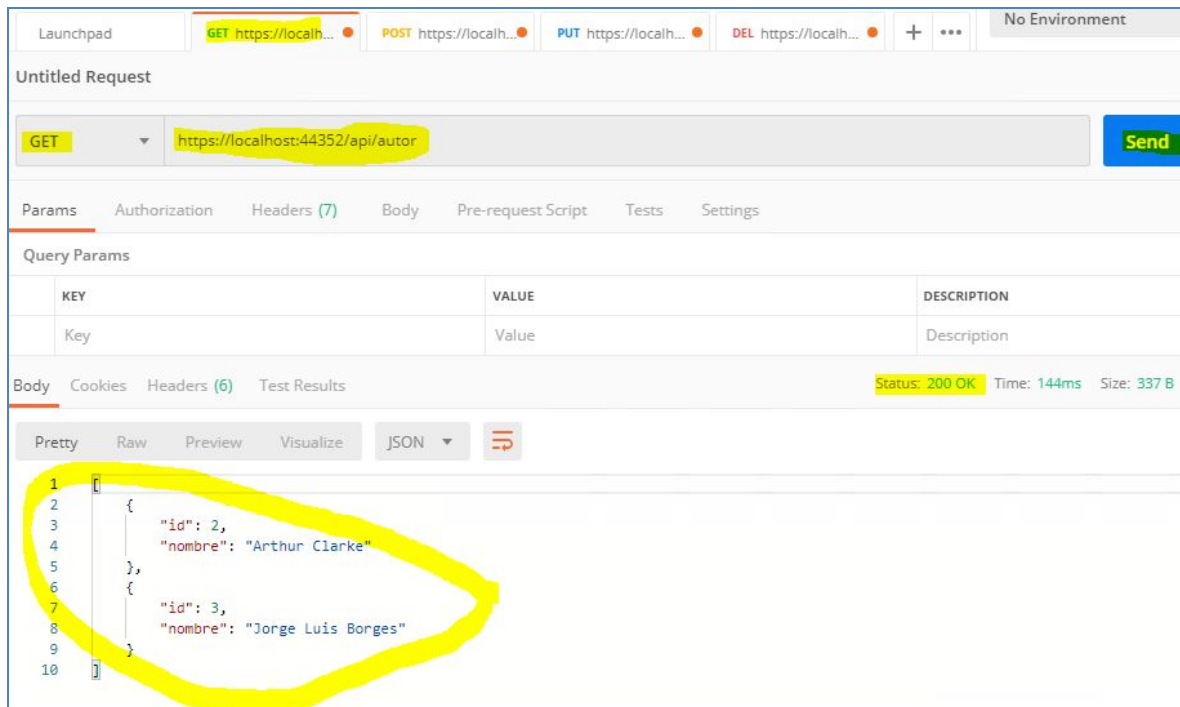
Body Cookies Headers (6) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "nombre": "Ernest Hemingway"
4 }
```

dbo.Autores [Data] ValuesController.cs Autor.cs		
Max Rows: 1000		
	Id	Nombre
▶	2	Arthur Clarke
	3	Jorge Luis Borges
✱	NULL	NULL

Si finalmente volvemos a hacer un GET de todos los autores, veremos reflejado lo que nos quedó en la base de datos, en PostMan.



Relacionando entidades del modelo

Hasta ahora nos hemos enfocado casi exclusivamente en el controlador **Autor controller**. Recordamos que en la aplicación también queremos modelar libros, por tanto haremos una relación de uno o muchos entre autores y libros donde a cada autor le corresponde un listado de libros que ha escrito.

Siguiendo la teoría del álgebra relacional en bases de datos, esta relación en realidad es una relación muchos a muchos pues un libro puede estar escrito por más de un autor. Sin embargo para no complicar demasiado el ejemplo lo haremos con una relación uno a uno, es decir un libro está escrito por un solo autor.

Crearemos una clase **Libro** en la carpeta de entidades con sus propiedades, entre ellas una propiedad de navegación **Autor** y atributos de validación **[Required]**:

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

```
namespace WebApiLibros.Entidades
{
```

```

public class Libro
{
    public int Id{ get; set; }
    [Required]
    public string Titulo{ get; set; }
    [Required]
    public int AutorId { get; set; }
    public Autor Autor{ get; set; }
}
}

```

Y luego editamos la clase **Autor** para agregar un listado de libros es decir una propiedad de navegación que sea un listado de libros y esta propiedad va a recibir el nombre **Libros**.

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;

namespace WebApiLibros.Entidades
{
    public class Autor
    {
        public int Id { get; set; }
        [Required]
        public string Nombre { get; set; }
        public List<Libro> Libros { get; set; }
    }
}

```

Finalmente iremos hacia el contexto de datos de la aplicación para agregar el DbSet correspondiente a Libro para poder crear la tabla de libros.

```

using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using WebApiLibros.Entidades;

namespace WebApiLibros.Contexto
{
    public class ApplicationDbContext : DbContext

```

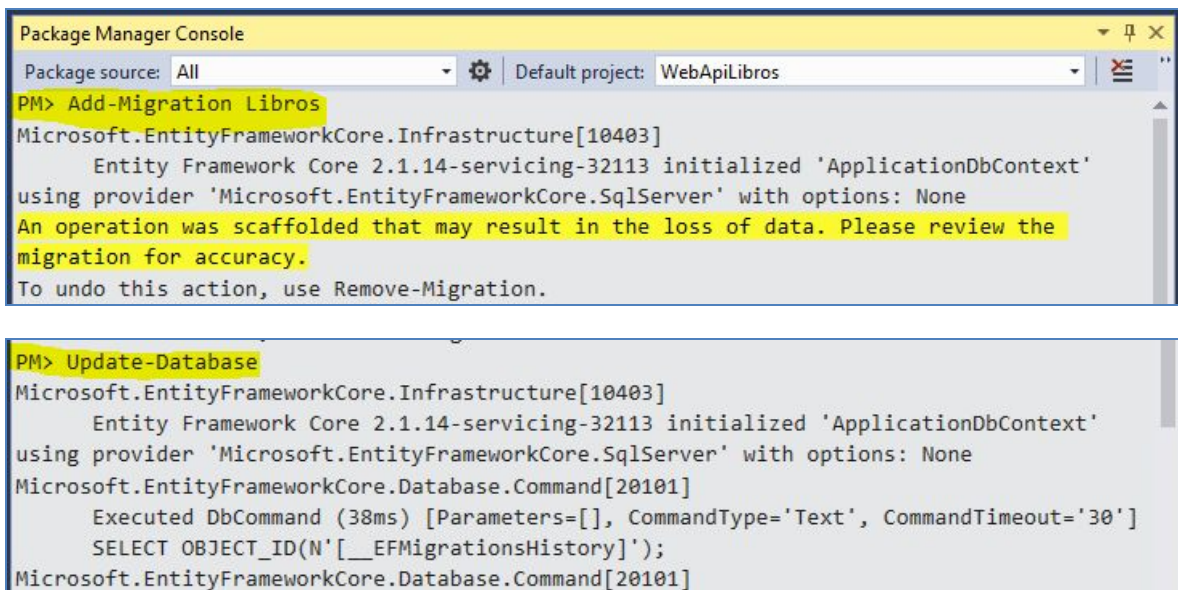
```

    {
        public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
base(options)
    {
    }

    public DbSet<Autor> Autores { get; set; }
    public DbSet<Libro> Libros { get; set; }
    }
}

```

Ahora agregaremos una migración de EF para que el nuevo modelo se refleje en la base de datos, con las instrucciones **Add-Migration Libros** y **Update-Database**:



The screenshot shows the Package Manager Console window with the following content:

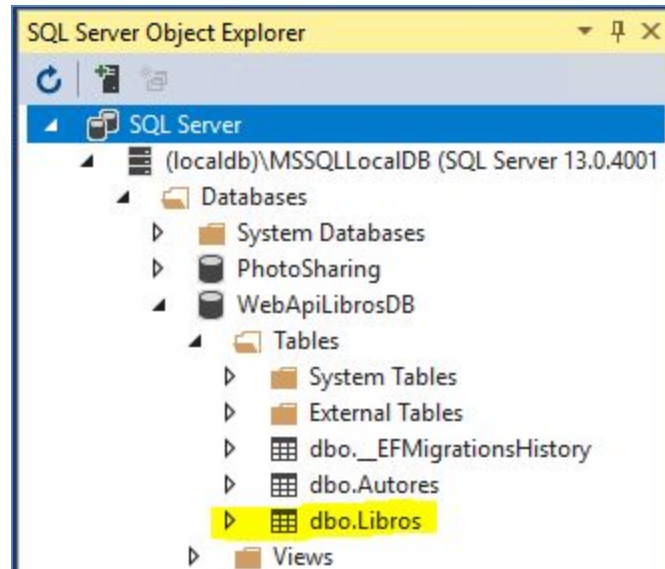
```

Package Manager Console
Package source: All | Default project: WebApiLibros
PM> Add-Migration Libros
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized 'ApplicationDbContext'
using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
An operation was scaffolded that may result in the loss of data. Please review the
migration for accuracy.
To undo this action, use Remove-Migration.

PM> Update-Database
Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 2.1.14-servicing-32113 initialized 'ApplicationDbContext'
using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: None
Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (38ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT OBJECT_ID(N'[__EFMigrationsHistory]');
Microsoft.EntityFrameworkCore.Database.Command[20101]

```

Podrá verificarlos en la base de datos, está la tabla **Libros**:



Ahora vamos a crear el controlador de libros desde la carpeta Controllers:

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

namespace WebApiLibros.Controllers
{
    [Route("api/[controller]")]
    [Microsoft.AspNetCore.Mvc.ApiController]
    public class LibroController : ControllerBase
    {
        private readonly ApplicationDbContext context;

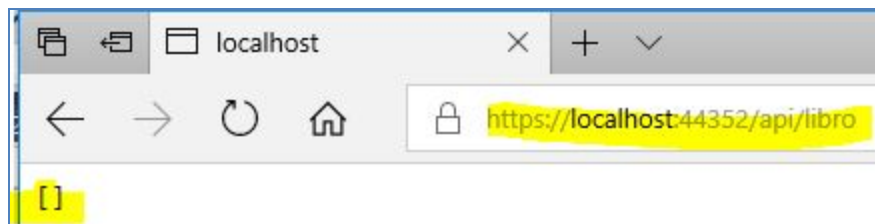
        public LibroController(ApplicationDbContext context)
        {
            this.context = context;
        }
    }
}
```

Y a partir de aquí crear un método Get para recuperar los libros, incluyendo la información del autor del libro, esto lo logramos con el método **Include** para incluir entidades relacionadas a la entidad que estamos leyendo de la base de datos, es decir, leemos **Libros** y obtenemos su relación con **Autores**.

En este caso además de traer los **libros** queremos traer la información del autor como su nombre, a través de la propiedad de navegación que se corresponde con la entidad relacionada que queremos traer, en este caso la propiedad de navegación es Autor.

```
[HttpGet]
public ActionResult<IEnumerable<Libro>> Get()
{
    return context.Libros.Include(x => x.Autor).ToList();
}
```

Vamos a probarlo solicitando el listado de libros que aún no tenemos por lo que no retornará ninguno:



Entonces debemos en el controlador Libro agregar el código necesario para poder insertar libros y recuperar un libro (el código es análogo al del controller Autor):

```
[HttpGet("{id}", Name = "ObtenerLibro")]
public ActionResult<Libro> Get(int id)
{
    var resultado = context.Libros.FirstOrDefault(x => x.Id ==
id);

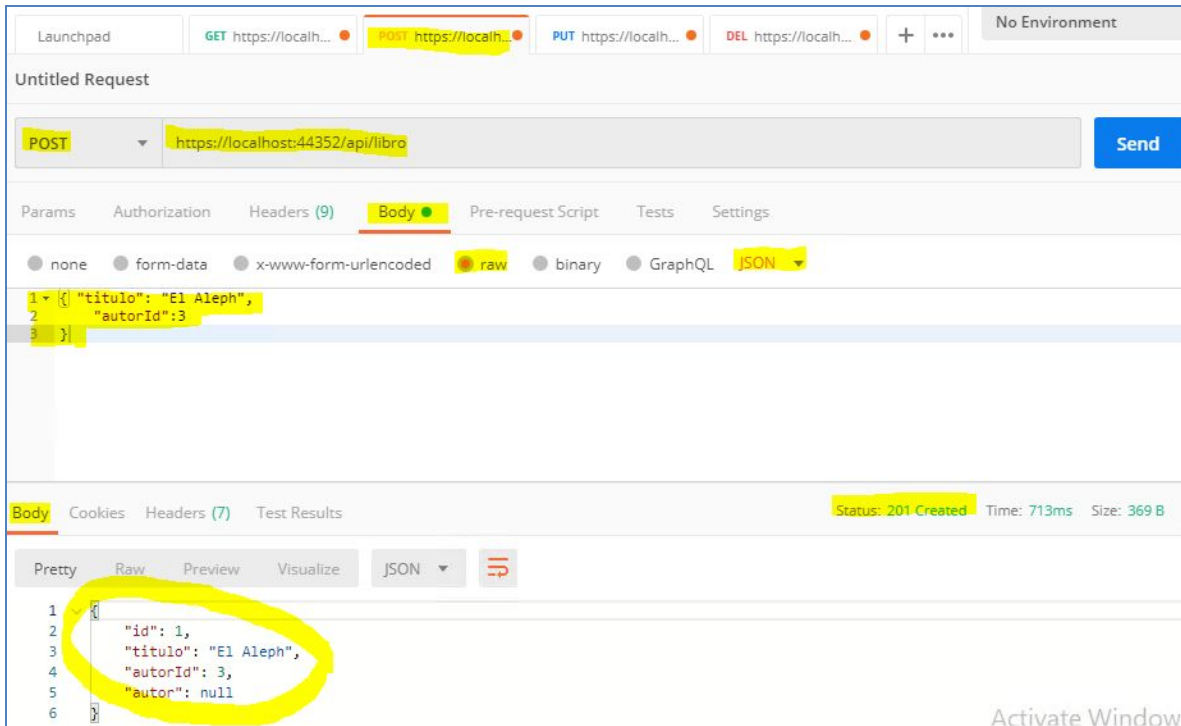
    if (resultado == null)
    { return NotFound(); }

    return resultado;
}

[HttpPost]
public ActionResult Post([FromBody] Libro libro)
{
    context.Libros.Add(libro);
    context.SaveChanges();
    return new CreatedAtRouteResult("ObtenerLibro", new { id =
libro.Id }, libro);
}
```

}

Desde Postman podemos agregar un nuevo libro:



Al tratar de hacer ahora un GET de los libros del autor, es posible que dé un error de referencia cíclica dado que en la definición del modelo Autor se hace referencia a los libros del autor, y en la definición del modelo Libro y para cada libro, se hace referencia al autor del libro. Esto se soluciona agregando una directiva de MVC en el método **ConfigureServices** de la clase **Startup**, simplemente indicando que ignore la referencia cíclica:

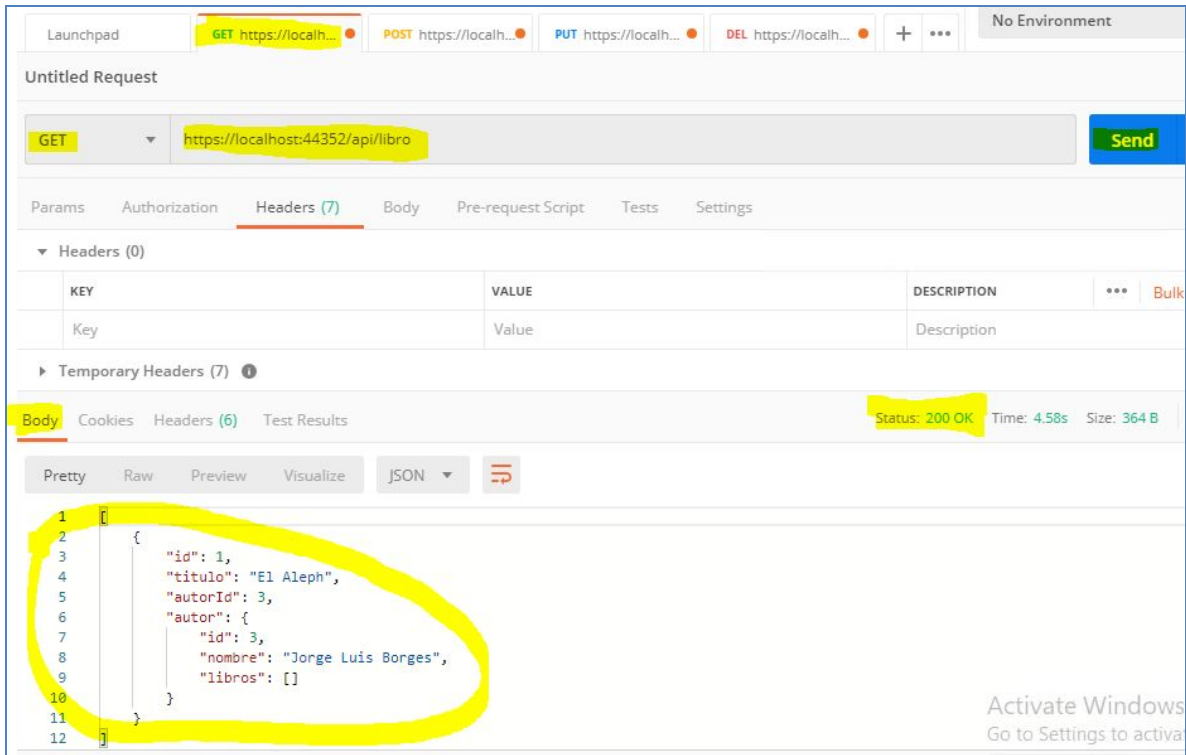
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection
String"))));

    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_
1)
```

```

        .AddJsonOptions(options =>
options.SerializerSettings.ReferenceLoopHandling=Newtonsoft.Json.Reference
eLoopHandling.Ignore);
    }

```



Nota: tenga en cuenta que si usa Asp.NetCore3, para ignorar las referencias circulares, debe tener la referencia a **Microsoft.AspNetCore.Mvc.NewtonsoftJson versión 3 o superior**, e ignorar la referencia circular con esta instrucción.

```

services.AddControllers()
    .AddNewtonsoftJson(x =>
    {
        x.SerializerSettings.ReferenceLoopHandling = ReferenceLoopHandling.Ignore;
    });

```

<https://stackoverflow.com/questions/55246715/reference-loop-handling-ignore-not-working-on-asp-net-core-3-0-preview-3>

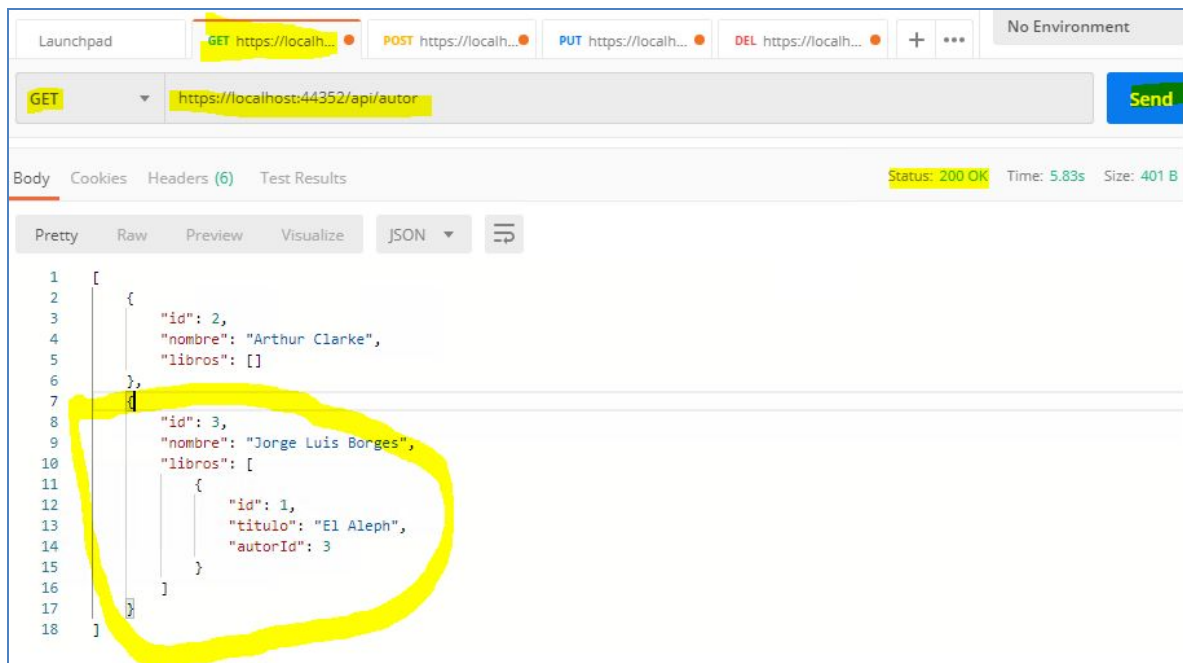
Si observamos la respuesta del GET anterior, vemos que está la información del libro, y del autor pero su lista de libros viene vacía porque indicamos ignorar la referencia cíclica. Para corregir esto, debemos hacer en la acción Get del controlador **Autor**, un **Include** de los libros:


```
[HttpGet]
public ActionResult<IEnumerable<Autor>> Get()
{
    return context.Autores.Include(x => x.Libros).ToList();
}
```

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

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

    return resultado;
}
```



Launchpad GET https://localhost:44352/api/autor POST https://localhost:44352/api/autor PUT https://localhost:44352/api/autor DEL https://localhost:44352/api/autor + ... No Environment

GET https://localhost:44352/api/autor Send

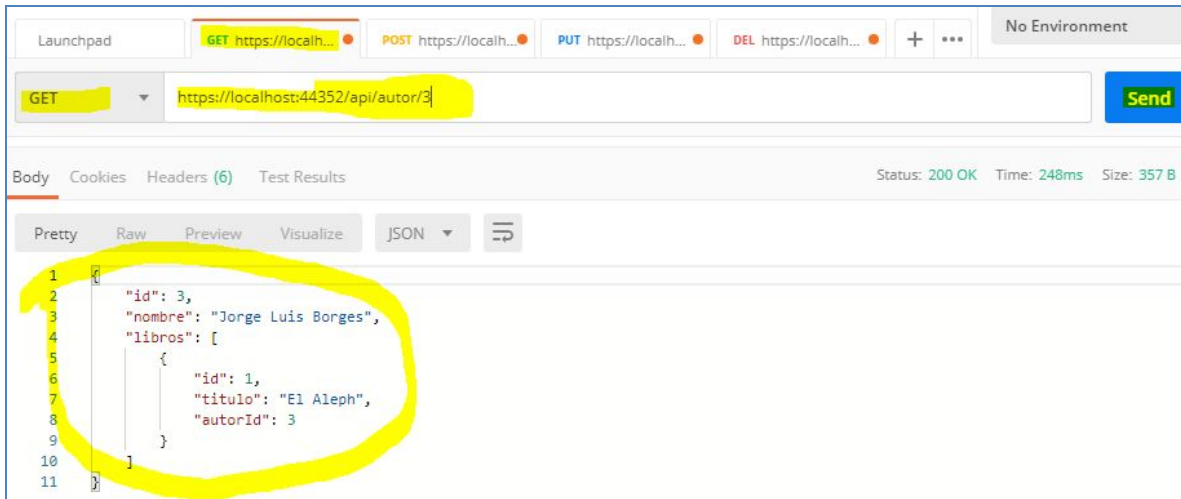
Body Cookies Headers (6) Test Results Status: 200 OK Time: 5.83s Size: 401 B

Pretty Raw Preview Visualize JSON

```

1  [
2  |
3  |   {
4  |     "id": 2,
5  |     "nombre": "Arthur Clarke",
6  |     "libros": []
7  |   },
8  |   {
9  |     "id": 3,
10 |     "nombre": "Jorge Luis Borges",
11 |     "libros": [
12 |       {
13 |         "id": 1,
14 |         "titulo": "El Aleph",
15 |         "autorId": 3
16 |       }
17 |     ]
18 |   }
19 ]

```



Y en el controlador **Libro**:

```
[HttpGet]
public ActionResult<IEnumerable<Libro>> Get()
{
    return context.Libros.Include(x => x.Autor).ToList();
}

[HttpGet("{id}", Name = "ObtenerLibro")]
public ActionResult<Libro> Get(int id)
{
    var resultado = context.Libros.Include(x =>
x.Autor).FirstOrDefault(x => x.Id == id);

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

    return resultado;
}
```

Por último dejamos aquí las acciones PUT y DELETE que son análogas a las del controlador **Libro**.

```
[HttpDelete("{id}")]
public ActionResult<Libro> Delete(int id)
{
    var resultado = context.Libros.FirstOrDefault(x => x.Id ==
id);

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

    context.Libros.Remove(resultado);
}
```

```
        context.SaveChanges();
        return resultado;
    }

    [HttpPut("{id}")]
    public ActionResult Put(int id, [FromBody] Libro value)
    {
        if (id != value.Id)
        {
            BadRequest();
        }

        context.Entry(value).State = EntityState.Modified;
        context.SaveChanges();
        return Ok();
    }
}
```