Taller

Como crear API en .NET CORE

Índice

[1 Requerimientos 3](#_Toc498334969)

[2 Creación del primer proyecto 3](#_Toc498334970)

[3 Logger 6](#_Toc498334971)

[4 Versionado 14](#_Toc498334972)

[5 Autenticación 16](#_Toc498334973)

[6 Inyección de Dependencias 19](#_Toc498334974)

[7 Conectar nuestra API a una Base de datos con Entity Framework 23](#_Toc498334975)

[8 Open API con Swagger 32](#_Toc498334976)

# Requerimientos

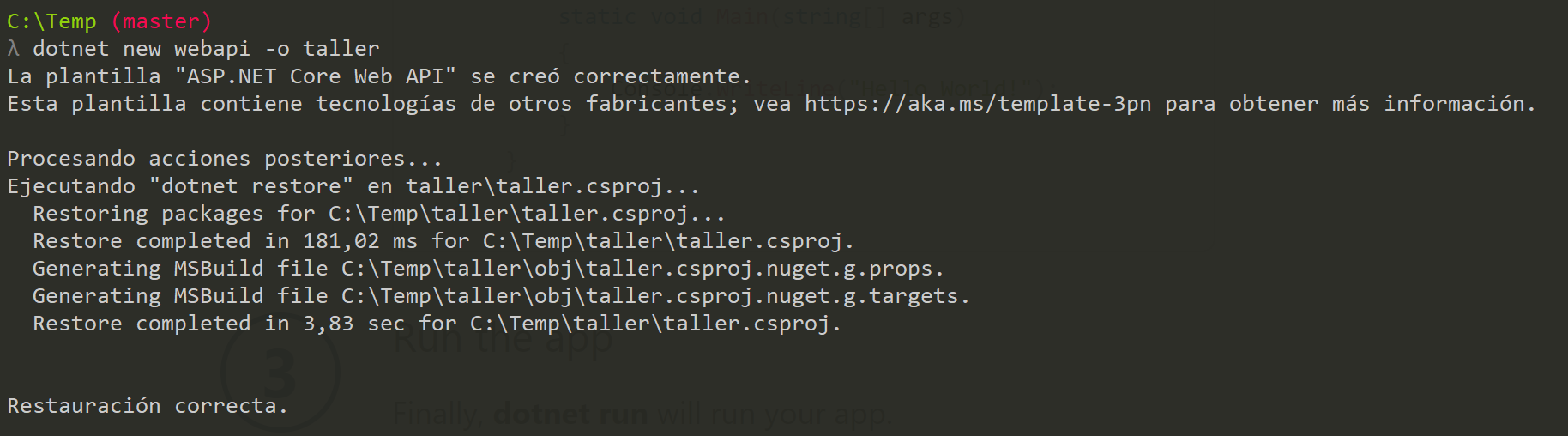
.Net Core se puede ejecutar es una herramienta multiplataforma por lo tanto dependiendo de nuestras necesidades tendremos que instalarnos una versión acorde a nuestras necesidades. Para ello dirigirse a la siguiente url <https://www.microsoft.com/net/core#windowscmd> y seguir los pasos que se indican

En cuanto al IDE a utilizar tanto para Windows como para Mac existen versiones de Visual Studio que nos otorga todas las características del IDE de Microsoft. Para entornos Windows existen distintas versiones (Community, Profesional, Enterprise) <https://www.visualstudio.com/es/vs/compare/>

Además de poder utilizar Visual Studio, el equipo de producto al hacer .NET multiplataforma ha querido que se pueda ejecutar desde cualquier plataforma y cualquier IDE en el que el desarrollador se sienta claro. Podemos utilizar otro IDE más ligero como puede ser Brackets, OmniSharper o Visual Studio Code. Nosotros utilizaremos este último para hacer este laboratorio.

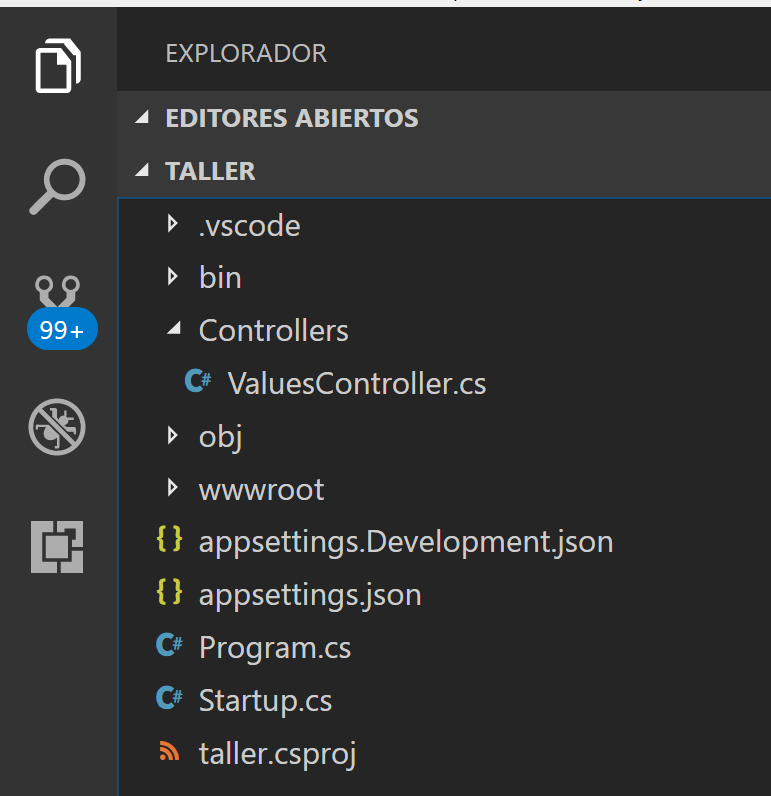
# Creación del primer proyecto

Abrimos un Terminal de Consola y ejecutamos la siguiente instrucción:

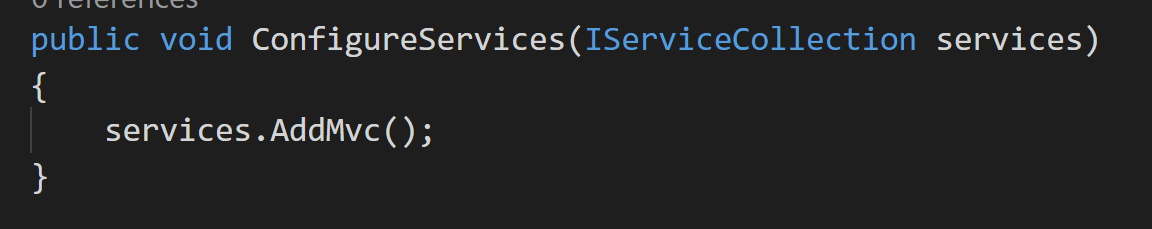


Dotnet es el archivo que hemos instalado en el paso 1. El siguiente paso indica la acción que queremos hacer dentro del SDK ahora mismo tenemos disponibles: New, restore, run, build, test, etc… Para ver todas las opciones que dispone ejecutamos el comando dotnet –-help. En nuestro caso como vamos a empezar a crear un proyecto pondremos la opción new. El siguiente parámetro indica la plantilla con la que se va a crear el proyecto. En nuestro caso “webapi”, la herramienta dispone de multitud de plantillas crear aplicaciones de consola, ASP.NET SPA, SPA con Angular, SPA con React… para ver las plantillas que disponemos donet new --help

Una vez ya tenemos el proyecto creado, vamos a ver que es lo que nos trae la plantilla de serie y que aspectos vamos a modificar con el fin de seguir una mejor organización del proyecto. Para ello en nuestro caso como vamos a utilizar Visual Studio Code lo que vamos a hacer es desde la propia línea de comando pondremos “code . “ y abrirá Code con la solución que acabamos de crear.

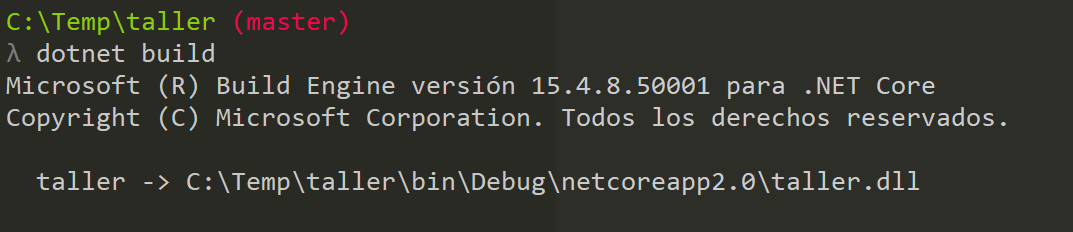


En primer lugar, como vemos es un proyecto con muy pocos ficheros y pocos aspectos de configuración. Abrimos el fichero Startup.cs, este fichero es en que se agregan los middlewares que vamos a utilizar (Ahora en .NET Core han creado un Framework modular en la que solamente vamos a cargar aquellos aspectos que vamos a utilizar, algo muy distinto a lo que se hacía con anterioridad, donde daba igual que es lo que necesitábamos que nuestra solución cargaba todo). Si abrimos este fichero vemos que lo único que tiene de momento es que se le añade el middleware AddMVC

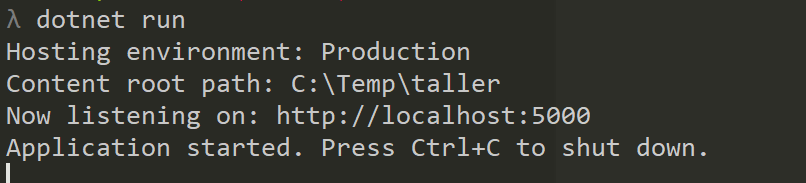


Una vez hemos visto cual es el punto de arranque de nuestra aplicación, vamos a realizar una serie de modificaciones en la misma ya que en nuestra opinión la nomenclatura da lugar a confusión. Esta modificación es cambiar el nombre de la carpeta Controller y ponerle el nombre API. En ASP NET antes de core existen dos tipos de Controller los de WebAPI y los Controller de Acciones. Esta separación era solo a nivel de la devolución de los métodos. En .NET Core el equipo de producto unifico estos dos tipos bajo una única clase (ya que no tenía sentido otra cosa). Ahora bien, de cara a la estructura de un proyecto y su legibildad es mucho mejor que estas estén separadas.

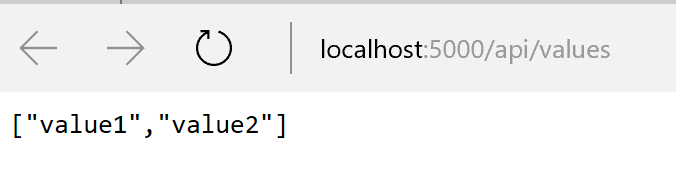
Con estas modificaciones en primer lugar compilaremos la solución con el comando “dotnet build”



SI la compilación ha ido correcta ejecutaremos “dotnet run” que levantará nuestra API

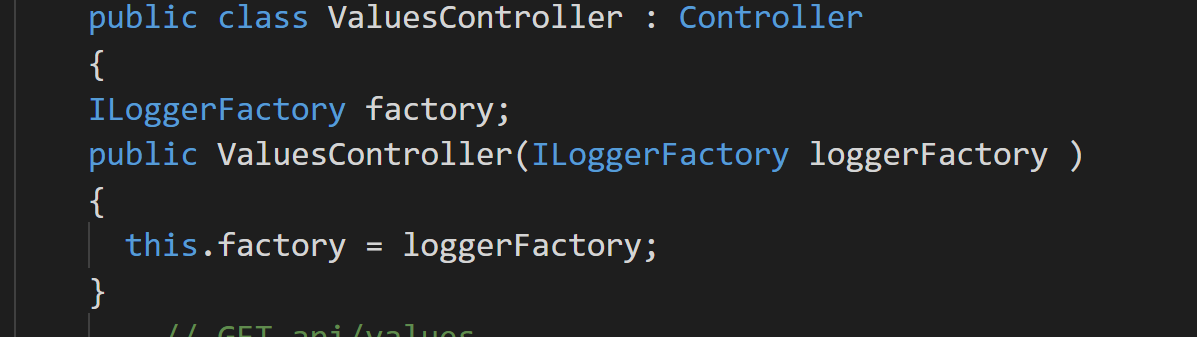


Para probar la WebAPI abrimos un navegador y ponemos la siguiente url <http://localhost:5000/api/values> si todo va bien veremos la siguiente pantalla:

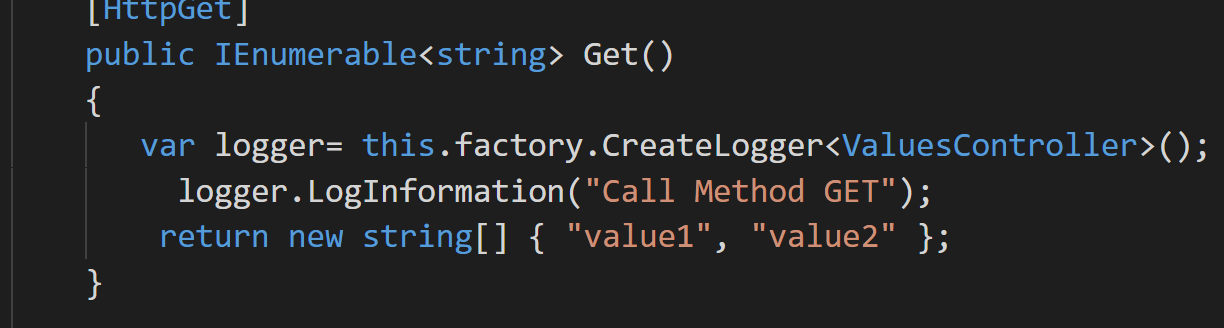


# Logger

Es muy tradicional en casi todo proyecto empezar creando una interfaz ILogger en la que definamos los métodos que va a utilizar nuestra clase de Login. Como casi todo en .NET Core Microsoft ya ha implementado un Log out of the box que puede cumplir con los requisitos en la mayoría de los proyectos que afrontemos. Como empezamos a utilizarlo, dentro del Controlador que hemos creado en el punto 2 nos creamos una propiedad ILoggerFactory y dentro del constructor se le inyecta por dependencia. El código quedaría tal que así:



Ahora para empezar a utilizarlo nos tendríamos que crear una variable que obtenga el log de dicha propiedad, por ejemplo, en el método values:

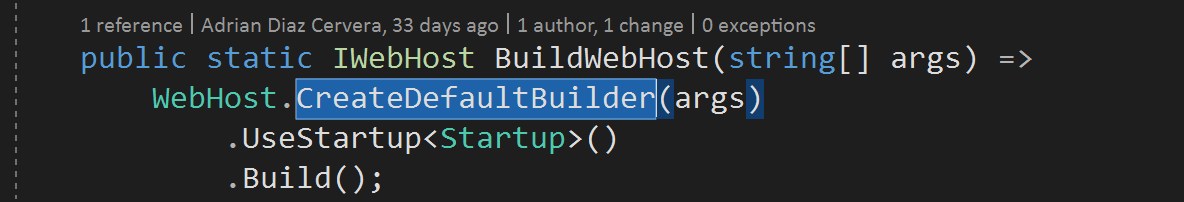


El método CreateLogger tiene varios constructores, una de las opciones que se puede hacer es pasarle la clase en la que se va a hacer uso de él, otra opción de las que se tiene es poder pasarle el nombre de la categoría dentro del constructor (pero un “magic string” no es una buena opción).

¿Pero como hacemos el caso de que vayamos a utilizar el log dentro de otra librería, que hacemos? En este caso lo que se recomienda como “best practices” es crear una clase en la que se englobe todo esta funcionalidad e invocar esta clase para todo lo relacionado con una misma funcionalidad.

Si lanzamos nuestra aplicación e invocamos a nuestra API veremos como en la salida de la consola podemos ver las trazas que hemos añadido.

Ahora bien, donde está la magia de serie por el que nosotros ya disponemos de un Log. Si visualizamos la clase Programa.cs, vemos que se crea una aplicación por defecto, si vemos el código en GitHub del método CreateDefaultBuild vemos como se inyecta esta clase en el propio pipeline. <https://github.com/aspnet/Hosting>



¿Pero si en lugar de querer utilizar el out of the box puedo utilizar otro Log? Claro para poder hacer uso de ello lo que debemos de hacer es modificar esta clase CreateDefaultBuilder. En nuestro caso vamos a utilizar Serilog. En primer lugar, tendremos que instalarnos los siguientes paquetes de Nuget:

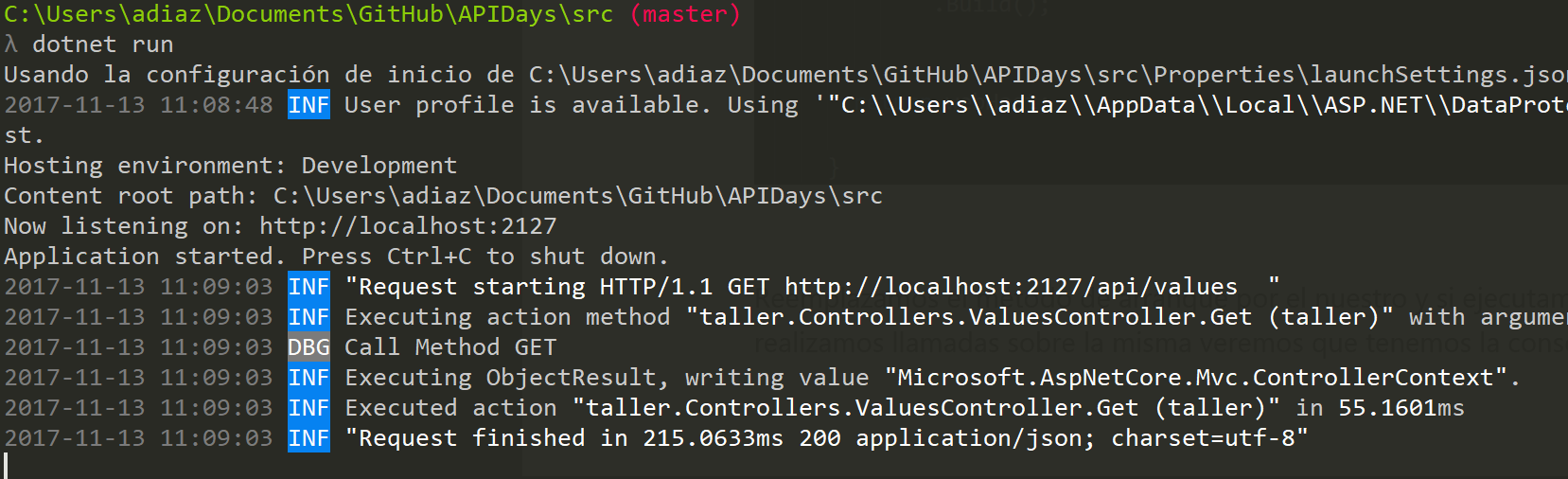
Serilog.AspnetCore

Serilog.Sinks.ColoredConsole

Una vez estos paquetes nos crearemos un BuildWebHostCustom en el que utilizaremos Serilog, para ello tendremos que poner el siguiente código:



Reemplazamos el método de arranque por el nuestro y si ejecutamos la aplicación y realizamos llamadas sobre la misma veremos que tenemos la consola con colores:



Enroutamiento WebAPI

Vamos a crear un nuestra API de StarWars para ello en primer lugar nos crearemos un fichero llamado StarWarsController.cs dentro de la carpeta API en el que pondremos el siguiente código:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

namespace taller.Controllers

{

public class StarWarsController : Controller

{

}

}

Sobre el public class pondremos el atributo “[Route("api/[controller]")] “en el que vamos a indicar cuál es la referencia a dicho controlador. A continuación, nos crearemos los métodos que van a tener nuestra API. En nuestro caso vamos a exponer cuatro métodos:

* Pedir las películas de StarWars
* Buscar las películas de StarWars dado una palabra
* Pedir los personajes de StarWars
* Buscar los personajes de StarWars dada una palabra

Antes de empezar con el propio desarrollo nos crearemos dos clases en los que tendremos el modelo de Peliculas y Personajes. Para ellos nos creamos una carpeta “Model” y añadimos dos ficheros: Films.cs y Characters.cs que tendrán el siguiente código:

Con las clases creadas, vamos al controlador de WebApi StarWars y añadimos el siguiente código:

public IEnumerable<Films> GetFilms() { return null; }

public IEnumerable<Films>GetFilmsByName(string name) { return null; }

public IEnumerable<Characters> GetCharacters() { return null; }

public IEnumerable<Characters> GetCharactersByName(string name)

{ return null;

}

public class Film

{

public int Id { get; set; }

public string Name { get; set; }

public string Year { get; set; }

}

public class Characters

{

public int Id { get; set; }

public string Name { get; set; }

public string SurName { get; set; }

public string Clan { get; set; }

}

Al tratarse de una primera versión, vamos a poner los datos directamente en el controlador (dejamos para más adelante como sacar los datos de una Base de datos o de otro service Web). Para ello dentro de la clase StarWars nos crearemos dos propiedades FilmsCollection y CharactersCollection con los siguientes valores:

IEnumerable<Films> FilmsCollection;

IEnumerable<Characters> CharactersCollection;

public StarWarsController(){

this.FilmsCollection= new List<Films>{

new Films { Id=1, Name="Una Nueva Esperanza"},

new Films {Id=2,Name="El imperio contrataca"},

new Films {Id=3, Name="El retorno del Jedi"},

new Films {Id=4, Name="La amenza fantasma"},

new Films {Id=5, Name="El ataque de los clones"},

new Films {Id=6, Name="La venganza de los Sith"},

};

this.CharactersCollection= new List<Characters>{

new Characters {Id=1, Name="Darth",SurName="Vader"},

new Characters {Id=2, Name="Yoda",SurName=""},

new Characters {Id=3, Name="Luke",SurName="Skywalker"},

new Characters {Id=4, Name="Leia",SurName="Organa"},

new Characters {Id=5, Name="Obi-Wan",SurName="Kenobi"},

};

}

Con estos valores vamos implementar los métodos que va a tener nuestros métodos:

public IEnumerable<Films> GetFilms()

{

return this.FilmsCollection;

}

public IEnumerable<Films>GetFilmsByName(string name)

{

return this.FilmsCollection.Where(x=>x.Name.Contains(name));

}

public IEnumerable<Characters> GetCharacters()

{

return this.CharactersCollection;

}

public IEnumerable<Characters> GetCharactersByName(string name)

{

return this. CharactersCollection.Where(x=>x.Name.Contains(name));

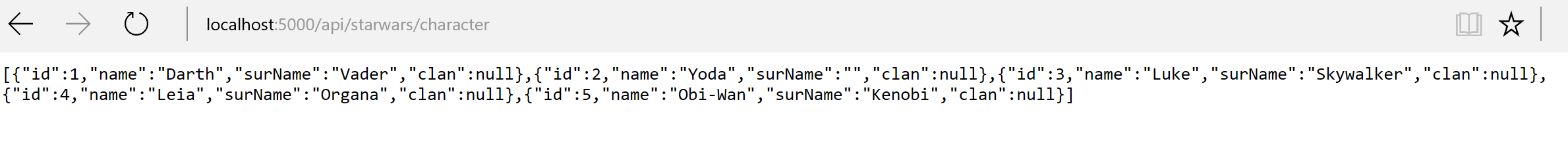
}

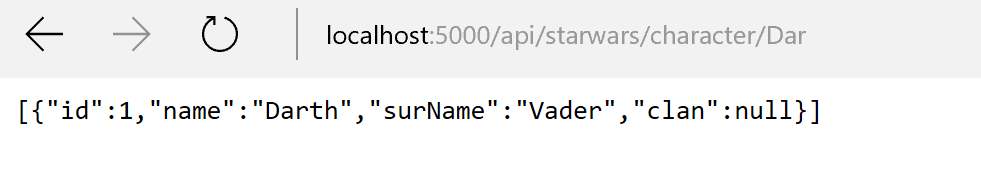
Ahora bien, para que nuestra API funcione lo que debemos de hacer es establecer los atributos por los que va a responder nuestros métodos. Para ello en cada uno de los métodos tenemos que añadirle el “verbo” por el que va a contestar en nuestro caso al ser operaciones solo de consulta añadiremos el atributo [HttpGet]. En caso de que queramos pasarle algún parámetro como por ejemplo en este caso la cadena del nombre lo que tenemos que poner es el verbo y dentro de dicho verbo el nombre de la variable que estamos esperando. Por ejemplo. [HttpGet(“films/{name}”]

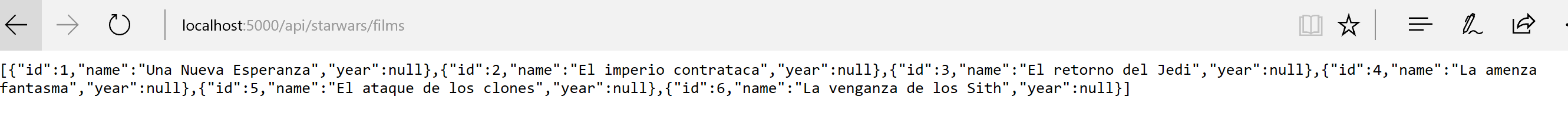
Una vez añadido estos métodos vamos a la consola de ejecución. Compilamos nuestra aplicación y posteriormente arrancamos la WebApi.

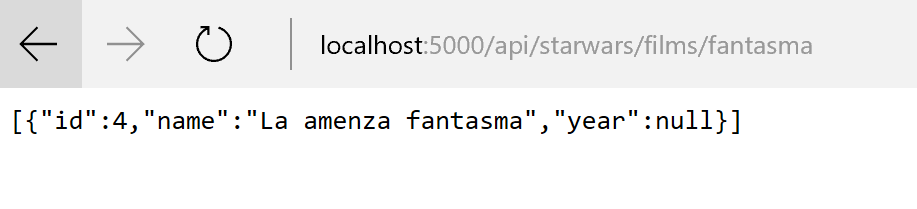
* Dotnet build
* Dotnet run

Y lanzamos diversas pruebas contra los métodos de la API.









# Versionado

Uno de los aspectos esenciales a la hora del desarrollo de una API es como establecemos su versionado de forma que podamos estar evolucionando la API y a los clientes de nuestra API no le vean afectados nuestros cambios. Con el fin de facilitar este versionado al desarrollador el equipo de ingenieros que ha desarrollado WebAPI, publico un paquete de Nuget que nos proporciona out of the box esta característica. Para ello vamos a la consola de ejecución y añadimos el paquete de Nuget donde disponemos un middleware para utilizar en nuestro proyecto:

* Dotnet add package Microsoft.AspNetCore.Mvc.Versioning

Con el paquete de Nuget añadido el siguiente paso es añadir este middleware dentro del punto de arranque. Para ello en el Startup.cs en el método ConfigureServices añadiremos las siguientes líneas:

services.AddApiVersioning(options =>

{

options.ReportApiVersions=true;

options.AssumeDefaultVersionWhenUnspecified = true;

var multiVersionReader = new HeaderApiVersionReader("x-version");

options.ApiVersionReader= multiVersionReader;

options.DefaultApiVersion = new ApiVersion(1, 0);

});

Dentro de estas opciones:

* ReportApiVersion 🡪 Indica que en la petición indicamos que versión de la API soporta la petición que hemos realizado.
* AssumeDefaultVersiónWhenUnspecified 🡪 En caso de que no se notifique la versión en la petición como tratamos esta petición, si se envía un error o bien si asume la versión por defecto.
* ApiVersiónReader 🡪 Ubicación donde indicamos la versión bien puede ser por QueryString o bien por HeaderAPIVersión
* DefaultApiVersión 🡪 Versión por defecto la API

Una vez definida como va a funcionar nuestro versionados vamos a versionar nuestra API. Para ello dentro de API nos crearemos una carpeta StarWars, con dos carpertas V1 y V2. Dentro de la carpeta V1 copiamos el fichero StarWarsController. Entramos en el mismo y modificamos el Namespace añadiendo v1 en su ruta quedando así:

namespace taller.Controllers.StarWars.v1

Ahora dentro de dicho fichero añadiremos una cabecera en el que indicamos la versión de la API:

[ApiVersion("1.0")]

Una vez ya tenemos versionado nuestra primera versión de la API, copiamos el fichero y lo ponemos en la carpeta V2 y modificamos el constructor del mismo de la siguiente forma:

[ApiVersion("2.0")]

public class StarWarsController : Controller

{

IEnumerable<Films> FilmsCollection;

IEnumerable<Characters> CharactersCollection;

public StarWarsController(){

this.FilmsCollection= new List<Films>{

new Films { Id=1, Name="A new hope"},

new Films {Id=2,Name="The Empire Strikes Back"},

new Films {Id=3, Name="The return of the Jedi"},

new Films {Id=4, Name="The Phantom Menace"},

new Films {Id=5, Name="Attack of the clones"},

new Films {Id=6, Name="Revenge of the Sith"},

};

this.CharactersCollection= new List<Characters>{

new Characters {Id=1, Name="Darth",SurName="Vader"},

new Characters {Id=2, Name="Yoda",SurName=""},

new Characters {Id=3, Name="Luke",SurName="Skywalker"},

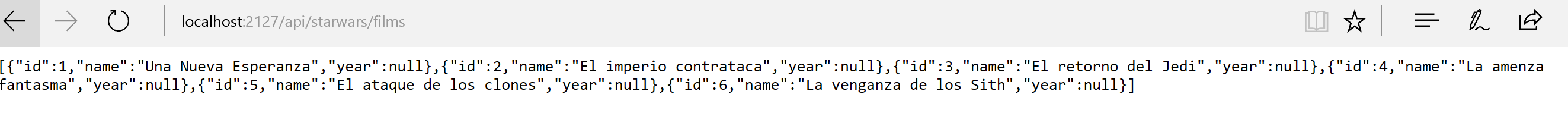
new Characters {Id=4, Name="Leia",SurName="Organa"},

new Characters {Id=5, Name="Obi-Wan",SurName="Kenobi"},

};

}

Si ahora compilamos la solución, y volvemos a arrancar la API, si hacemos la petición normal nos devolverá la versión 1



Si hacemos la petición utilizando por ejemplo PostMan y añadimos el parámetro x-version en el header. Comprobamos que nos devuelve la versión 2:



# Autenticación

Una vez tenemos nuestra WebAPI funcionando ahora vamos a establecerle un sistema de seguridad para eso vamos a utilizar el servicio de Azure Active Directory. Para hacer este paso es necesario una cuenta con un Live ID. Para ello ir a la siguiente dirección y crearemos nuestra Aplicación <https://docs.microsoft.com/es-es/azure/active-directory/develop/active-directory-v2-app-registration>

De la misma forma en la que se configura el resto de aspectos en WebAPI, vamos a implementarnos un Middleware que nos implemente la autenticación contra el AzureAD. Para ello en primer lugar nos crearemos una carpeta “Infraestructure” dentro de esta nos creamos una carpeta “AzureAD” y nos crearemos dos ficheros: AzureAdAuthenticationBuilderExtensions.cs y AzureAdOptions.cs

Dentro del AzureADOptions es una clase donde consultaremos la configuración de la misma y dentro de la otra clase estará todos los aspectos que necesitará nuestra WebAPI para poder autenticarse. Quedará como la siguiente:

namespace Microsoft.AspNetCore.Authentication.Extensions

{

public class AzureAdOptions

{

public string ClientId { get; set; }

public string ClientSecret { get; set; }

public string Instance { get; set; }

public string Domain { get; set; }

public string TenantId { get; set; }

public string CallbackPath { get; set; }

}

}

Y la otra Clase quedará como la siguiente:

using System;

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Options;

using Microsoft.AspNetCore.Authentication.Extensions;

using Microsoft.AspNetCore.Authentication.OpenIdConnect;

namespace Microsoft.AspNetCore.Authentication

{

public static class AzureAdServiceCollectionExtensions

{

public static AuthenticationBuilder AddAzureAdBearer(this AuthenticationBuilder builder)

=> builder.AddAzureAdBearer(\_ => { });

public static AuthenticationBuilder AddAzureAdBearer(this AuthenticationBuilder builder, Action<AzureAdOptions> configureOptions)

{

builder.Services.Configure(configureOptions);

builder.Services.AddSingleton<IConfigureOptions<JwtBearerOptions>, ConfigureAzureOptions>();

builder.AddJwtBearer();

return builder;

}

private class ConfigureAzureOptions: IConfigureNamedOptions<JwtBearerOptions>

{

private readonly AzureAdOptions \_azureOptions;

public ConfigureAzureOptions(IOptions<AzureAdOptions> azureOptions)

{

\_azureOptions = azureOptions.Value;

}

public void Configure(string name, JwtBearerOptions options)

{

options.Audience = \_azureOptions.ClientId;

options.Authority = $"{\_azureOptions.Instance}{\_azureOptions.TenantId}";

}

public void Configure(JwtBearerOptions options)

{

Configure(Options.DefaultName, options);

}

}

}

}

Una vez tenemos la clase lista el siguiente paso es configurar nuestro arranque para indicarle que vamos a utilizar la autenticación sobre Azure AD y utilizando JWT. Para ello dentro del fichero StartUp, en el método Configure Services añadimos las siguientes líneas:

services.AddAuthentication(sharedOptions =>

{

sharedOptions.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddAzureAdBearer(options => Configuration.Bind("AzureAd", options));

Con esto, ahora lo único que nos queda seria añadir en los controladores que deseemos seguridad añadir el atributo [Authorize]

ASP.NET permite cualquier tipo de autenticación bien utilizando Google, Facebook, Twitter o una personalizada, para ello hay que crearse el Middleware apropiado.

**NOTA:** Para obtener un token sobre el Azure AD podemos seguir los siguientes pasos <https://gist.github.com/nzpcmad/687d3f7e2f0d9f1592b55fac66cd82ec>

# Inyección de Dependencias

Como hemos visto hasta la fecha en ASP NET Core todo es modular y podemos utilizar cualquier elemento del Framework o bien reemplazarlo por otro. Esto es una de las grandes ventajas que tenemos. En este aspecto como en el anterior el framework trae inyección de dependencias de Serie. Vamos manos a la obra. En primer lugar nos crearemos una carpeta Services, dentro de hay crearemos una Interface IStarWarsService.cs y una clase StarWarsMockService.cs donde implementaremos dicha interface. El código quedaría de la siguiente forma:

IStarwarsService.cs

using System.Collections.Generic;

using taller.Model;

namespace taller.Services

{

public interface IStarWarsService

{

IEnumerable<Films> GetFilms();

IEnumerable<Films> GetFilmsByName(string name);

IEnumerable<Characters> GetCharacters();

IEnumerable<Characters> GetCharactersByName(string name);

}

}

StarWarsMockService.cs

using System.Collections.Generic;

using System.Linq;

using taller.Model;

namespace taller.Services

{

public class StarWarsMockService : IStarWarsService

{

IEnumerable<Films> FilmsCollection;

IEnumerable<Characters> CharactersCollection;

public StarWarsMockService()

{

this.FilmsCollection= new List<Films>{

new Films { Id=1, Name="Una Nueva Esperanza"},

new Films {Id=2,Name="El imperio contrataca"},

new Films {Id=3, Name="El retorno del Jedi"},

new Films {Id=4, Name="La amenza fantasma"},

new Films {Id=5, Name="El ataque de los clones"},

new Films {Id=6, Name="La venganza de los Sith"},

};

this.CharactersCollection= new List<Characters>{

new Characters {Id=1, Name="Darth",SurName="Vader"},

new Characters {Id=2, Name="Yoda",SurName=""},

new Characters {Id=3, Name="Luke",SurName="Skywalker"},

new Characters {Id=4, Name="Leia",SurName="Organa"},

new Characters {Id=5, Name="Obi-Wan",SurName="Kenobi"},

};

}

IEnumerable<Characters> IStarWarsService.GetCharacters()

{

return this.CharactersCollection;

}

IEnumerable<Characters> IStarWarsService.GetCharactersByName(string name)

{

return this.CharactersCollection.Where(x=>x.Name.Contains(name));

}

IEnumerable<Films> IStarWarsService.GetFilms()

{

return this.FilmsCollection;

}

IEnumerable<Films> IStarWarsService.GetFilmsByName(string name)

{

return this.FilmsCollection.Where(x=>x.Name.Contains(name));

}

}

}

Una vez tenemos esto implementado vamos al controlador de la versión 1 añadimos este servicio y lo quedaría de la siguiente forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using taller.Model;

using taller.Services;

namespace taller.Controllers.StarWars.v1

{

[Route("api/StarWars/")]

[ApiVersion("1.0")]

public class StarWarsController : Controller

{

private readonly IStarWarsService service;

public StarWarsController(IStarWarsService service){

this.service=service;

}

[HttpGet("films")]

public IEnumerable<Films> GetFilms()

{

return this.service.GetFilms();

}

[HttpGet("films/{name}")]

public IEnumerable<Films>GetFilmsByName(string name)

{

return this.service.GetFilmsByName(name);

}

[HttpGet("character")]

public IEnumerable<Characters> GetCharacters()

{

return this.service.GetCharacters();

}

[HttpGet("character/{name}")]

public IEnumerable<Characters> GetCharactersByName(string name)

{

return this.service.GetCharactersByName(name);

}

}

}

Una vez modificado el controlador tenemos que añadir en nuestro StartUpc.s la resolución de esta interfaz para ello tendremos que poner el siguiente código dentro del método ConfigureServices:

services.AddScoped<IStarWarsService,StarWarsMockService>();

Podemos registrar los servicos de tres formas:

AddTransient -> Interfaz a objeto volátil

AddSingleton 🡪 En memoria solo existe una copia de esta interfaz y se utiliza el resto de peticiones

AddScope 🡪 “Per-Request”

En caso de que no querramos utilizar el contenedor de dependencias de serie podemos utilizar cualquiera. En nuestro caso vamos a utilizar Autofac. Para ello lo que debemos de hacer es:

- En primer lugar lo que debemos de hacer es instalar los siguientes paquetes de Nuget:

🡪 dotnet add package autofac

🡪 dotnet add package autofac.extensions.dependencyinjection

A continuación, dentro de infraestructure nos crearemos una carpeta AutofacModules y dentro de la misma un fichero ServicesModules.cs

using Autofac;

using taller.Services;

namespace taller.Infraestructure.AutofacModules

{

public class ServicesModules : Module

{

protected override void Load(ContainerBuilder builder)

{

builder.RegisterType<StarWarsMockService>().As<IStarWarsService>();

}

}

}

Vamos otra vez al método ConfigureServices, modificamos el método para que ahora nos devuelve una interfaz IServiceProvider y modificamos el pipeline para indicarle que el contenedor de dependencias es Autofac. El método quedaría así:

public IServiceProvider ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(sharedOptions =>

{

sharedOptions.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddAzureAdBearer(options => Configuration.Bind("AzureAd", options));

services.AddApiVersioning(options =>

{

options.ReportApiVersions=true;

options.AssumeDefaultVersionWhenUnspecified = true;

var multiVersionReader = new HeaderApiVersionReader("x-version");

options.ApiVersionReader= multiVersionReader;

options.DefaultApiVersion = new ApiVersion(1, 0);

});

services.AddMvc();

// services.AddScoped<IStarWarsService,StarWarsMockService>();

var container = new ContainerBuilder();

container.Populate(services);

container.RegisterModule<taller.Infraestructure.AutofacModules.ServicesModules>();

return new AutofacServiceProvider(container.Build());

}

# Conectar nuestra API a una Base de datos con Entity Framework

Es el momento de empezar a extraer datos desde una base de datos real y para ello tenemos que seguir los siguientes pasos:

1. **Instalación de Paquetes para utilizar EntityFramework**

Debemos de añadir los siguientes paquetes a la solución, ejecutando las siguientes instrucciones por consola:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package --version 1.1.0-msbuild3-final Microsoft.EntityFrameworkCore.Tools

dotnet restore

Una vez añadido los paquetes debemos añadir al .csproj las siguientes líneas de código:

<ItemGroup>

<DotNetCliToolReference

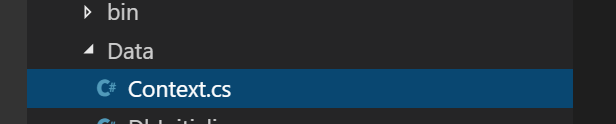
Include="Microsoft.EntityFrameworkCore.Tools.DotNet"

Version="1.0.0-msbuild3-final" />

</ItemGroup>

1. **Crear los contextos de bases de datos**

Para empezar a trabajar con EF contra nuestra base de datos debemos preparar los contextos, y para ello vamos a crear una carpeta Data en nuestra solución, y dentro de ella un fichero **Context.cs**



Esta clase Context se va a encargar de mapear nuestros modelos implementados de

**Films y Characters** con el modelo de base de datos.

Para ello deberemos pegar en la clase creada el siguiente código:

using Microsoft.EntityFrameworkCore;

using taller.Model;

using Microsoft.Extensions.Configuration;

namespace taller.Data

{

public class EFCoreWebDemoContext : DbContext

{

public IConfiguration config {get;set;}

public DbSet<Characters> Characters { get; set; }

public DbSet<Films> Films { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=ApiDayDemoDB;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False");

}

}

}

Con este código estamos vinculando dos tablas con los “DbSet” a nuestros modelos.

Además estamos configurando el acceso a datos con el método OnConfiguring, que como vemos tiene una cadena de conexión a una base de datos local llamada **ApiDayDemoDB**

1. **Crear inicializador de Base de datos**

Hasta ahora hemos configurado nuestra clase de contexto, pero ahora vamos a crear una clase **DbInitializer.cs** en la carpeta Data que nos va a rellenar en la primera conexión de datos las dos tablas.

Para ello debemos copiar el siguiente código en la nueva clase:

using System.Collections.Generic;

using System.Linq;

using taller.Model;

using taller.Data;

namespace taller.Data

{

public sealed class DbInitializer

{

private EFCoreWebDemoContext \_context;

public DbInitializer(EFCoreWebDemoContext context)

{

\_context = context;

}

public void SeedData()

{

if (!\_context.Characters.Any())

{

\_context.Characters.Add(new Characters{Name = "Maestro",SurName ="Yoda",Clan ="Jedi"});

\_context.Characters.Add(new Characters{Name = "Obi",SurName ="One",Clan ="Jedi"});

\_context.Characters.Add(new Characters{Name = "Luk",SurName ="SkyWalker",Clan ="Jedi"});

\_context.Characters.Add(new Characters{Name = "Anakin",SurName ="SkyWalker",Clan ="Jedi"});

\_context.Characters.Add(new Characters{Name = "Darth",SurName ="Vader",Clan ="Lord Sith"});

\_context.Characters.Add(new Characters{Name = "Darth",SurName ="Pleguis",Clan ="Lord Sith"});

\_context.Characters.Add(new Characters{Name = "Darth",SurName ="Sidious",Clan ="Lord Sith"});

\_context.SaveChanges();

}

if (!\_context.Films.Any())

{

\_context.Films.Add(new Films(){Name ="Star Wars: Episode IV - A New Hope",Year="1977"});

\_context.Films.Add(new Films(){Name ="Star Wars: Episode V - The Empire Strikes Back",Year="1980"});

\_context.Films.Add(new Films(){Name ="Star Wars: Episode VI - Return of the Jedi",Year="1983"});

\_context.Films.Add(new Films(){Name ="Star Wars: Episode I - The Phantom Menace",Year="1999"});

\_context.Films.Add(new Films(){Name ="Star Wars: Episode II - Attack of the Clones",Year="2002"});

\_context.Films.Add(new Films(){Name ="Star Wars: Episode III - Revenge of the Sith",Year="2005"});

\_context.SaveChanges();

}

}

}

}

1. **Crear la base de datos: Migraciones**

Para crear la base de datos en local, y rellenarla por primera vez vamos a generar una migración en base a nuestros modelos iniciales de Film y Characters, aplicando un modelo CodeFirst, generamos las tablas en base a las clases.

Para ello vamos a ejecutar los siguientes comandos:

* **dotnet ef migrations add localDB**

Este primer comando si se genera de forma correcta nos creará una carpeta Migrations en la solución, y tres ficheros con los cambios recogidos de nuestros modelos. Este paso además de servirnos para generar un primer schema de nuestra base de datos lo deberemos ejecutar siempre que añadamos un cambio a nuestras clases modelo.

* **dotnet ef database update**

Esta instrucción ejecuta todas las migraciones pendientes de ejecutarse, y nuestro caso nos crea las dos tablas correspondientes

1. **Cadenas de conexión para nuestro contexto y registro del contexto y del inicializador**

Como hemos visto en el paso 2 hemos dejado la cadena de conexión “harcodeada” a la inicialización del contexto, por lo que vamos a mejorar un poco el código.

Para ello debemos en un primer paso añadir al fichero appsettings.json de nuestra solución el siguiente código:

"ConnectionStrings": {

"LocalDB": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=ApiDayDemoDB;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=True;ApplicationIntent=ReadWrite;MultiSubnetFailover=False",

"AzureDB": "Server=tcp:apidaydemoserver.database.windows.net,1433;Initial Catalog=ApiDayDemoDB;Persist Security Info=False;User ID=admindb;Password=12345.aa;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"

}

Esto nos servirá para indicar tanto la cadena de conexión a nuestra base de datos local, como la cadena de conexión del ejemplo en Azure SQL que os hemos proporcionado.

Para poder usar estas cadenas volveremos al código de nuestro Context.cs y lo dejaremos de la siguiente forma:

using Microsoft.EntityFrameworkCore;

using taller.Model;

using Microsoft.Extensions.Configuration;

namespace taller.Data

{

public class EFCoreWebDemoContext : DbContext

{

public IConfiguration config {get;set;}

public DbSet<Characters> Characters { get; set; }

public DbSet<Films> Films { get; set; }

public EFCoreWebDemoContext(IConfiguration configuration)

{

config = configuration;

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(ConfigurationExtensions.GetConnectionString(config,"LocalDB"));

}

}

}

Como novedad ahora instanciamos la clase IConfiguration que se configura en la clase Startup.cs de nuestra solución, y que nos permitirá acceder al archivo json de configuración donde dejamos nuestras cadenas.

Por último, para que tanto Dbinitializer.cs como Context.cs funcionen correctamente deberemos añadir el siguiente código en Startup.cs:

* **Al método ConfiguresServices**

services.AddDbContext<EFCoreWebDemoContext>();

services.AddTransient<DbInitializer>();

* **Al método Configure justo delante de “app.UseMVC”**

init.SeedData();

Con estos cambios inyectamos el contexto en nuestra aplicación ya que nos será necesario para nuestros servicios, y además registramos el DbInitializer para poder instanciarlo al inicio de la ejecución del api.

Si ejecutáramos ahora la solución, se creará una base de datos con dos tablas, además de insertarse todos los datos necesarios para la demo.

1. **Añadiendo un servicio contra la base de datos**

Para terminar vamos a sustituir nuestro servicio de Mock, por uno que consulte de la base de datos. Para ello vamos a crear la clase “StarWarsEFService” en la carpeta Services de nuestra solución.

Esta clase contendrá el siguiente código:

using System.Collections.Generic;

using System.Linq;

using taller.Model;

using taller.Data;

namespace taller.Services

{

public class StarWarsEFService : IStarWarsService

{

IEnumerable<Films> FilmsCollection;

IEnumerable<Characters> CharactersCollection;

private EFCoreWebDemoContext \_context;

public StarWarsEFService(EFCoreWebDemoContext context)

{

\_context = context;

}

IEnumerable<Characters> IStarWarsService.GetCharacters()

{

return \_context.Characters.ToList();

}

IEnumerable<Characters> IStarWarsService.GetCharactersByName(string name)

{

return \_context.Characters.Where(c=> c.Name == name).ToList();

}

IEnumerable<Films> IStarWarsService.GetFilms()

{

return \_context.Films.ToList();

}

IEnumerable<Films> IStarWarsService.GetFilmsByName(string name)

{

return \_context.Films.Where(c=> c.Name == name).ToList();

}

}

}

Como veis en la clase, si no hubiésemos registrado nuestro contexto de base de datos no podríamos recibirlo en el constructor de esta clase.

Para que el contenedor de dependencias nos registre el servicio correcto debemos cambiar la clase **ServiceModules,** dejándola de la siguiente forma:

using Autofac;

using taller.Services;

namespace taller.Infraestructure.AutofacModules

{

public class ServicesModules : Module

{

protected override void Load(ContainerBuilder builder)

{

builder.RegisterType<StarWarsEFService>().As<IStarWarsService>();

}

}

}

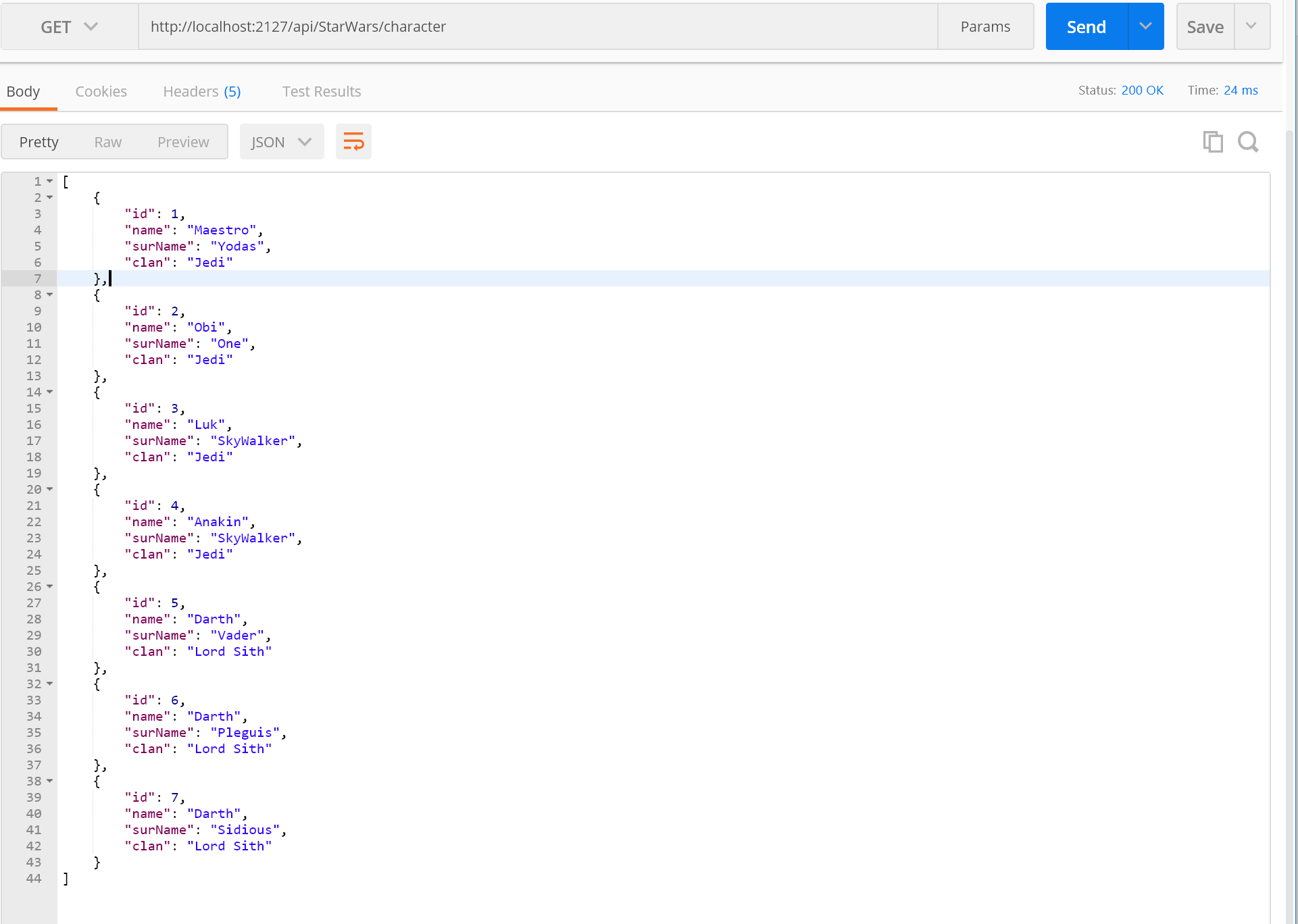
1. **Un poco de testing**

Deberemos compilar la solución con la instrucción **“dotnet build”** y si todo es correcto lanzaremos la app con **“dotnet run”**.

Una vez lanzada la api de forma correcta podemos hacer uso de nuestro postman o bien vía url y realizar las siguientes peticiones:

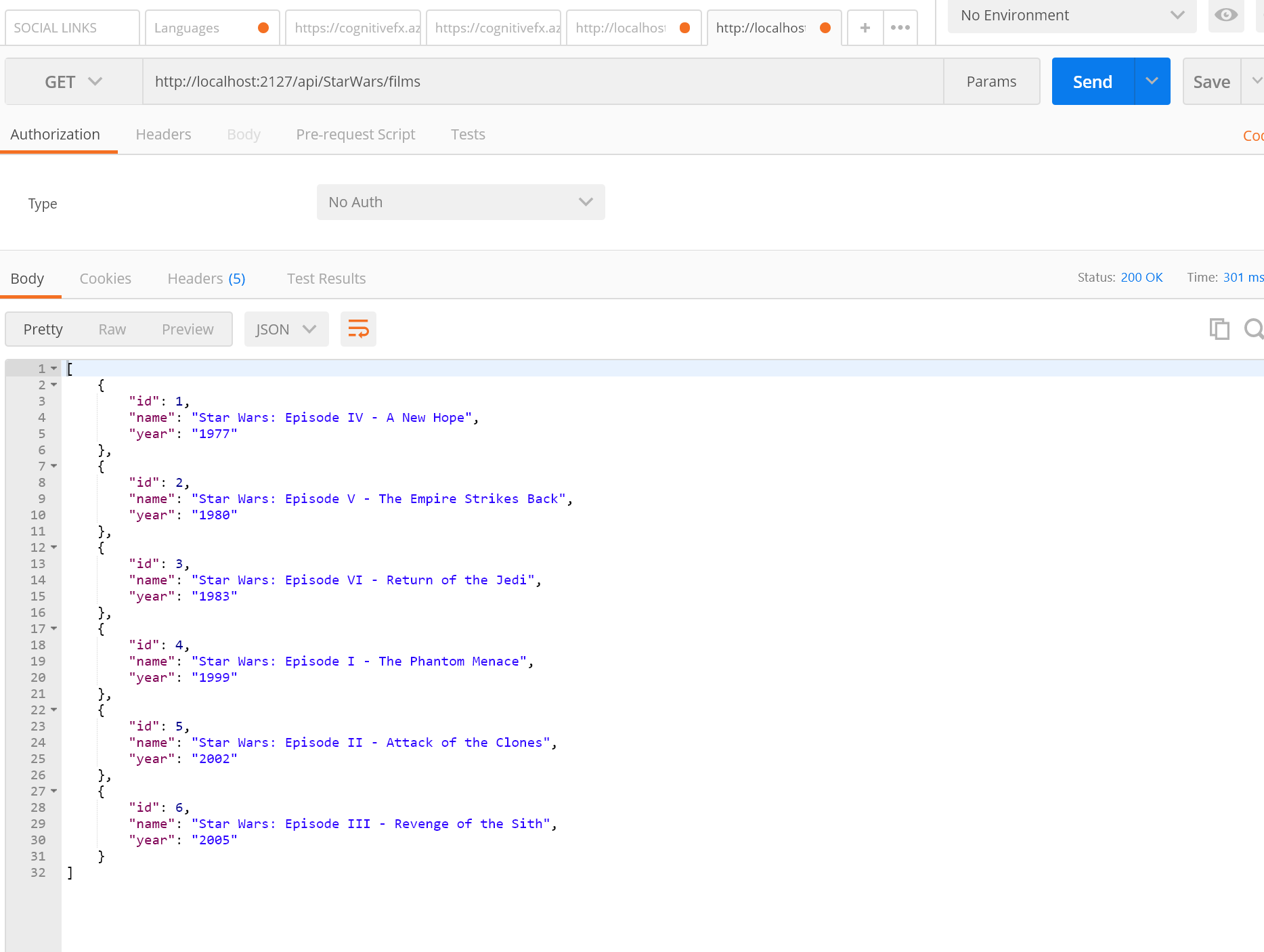
**[Get]**

[Http://localhost:<port>/api/StarWars/Character](Http://localhost:%3cport%3e/api/StarWars/Character)



**[Get]**

[Http://localhost:<port>/api/StarWars/films](Http://localhost:%3cport%3e/api/StarWars/films)



# Open API con Swagger

Para añadir Swagger a nuestra solución deberemos ejecutar en primer lugar la siguiente instrucción en la consola:

**dotnet add package swashbuckle.AspNetCore**

Esta instrucción instala los paquetes necesarios para poder configurar la herramienta Swagger en nuestra solución.

El siguiente paso es configurar la clase **StarttUp.cs** para que podamos iniciar Swagger. Para ello vamos añadir los siguientes fragmentos de código en la clase:

* Using para referencias los paquetes de Swagger

using Swashbuckle.AspNetCore.Swagger;

using Swashbuckle.AspNetCore;

* Añadimos al método ConfiguresServices el siguiente código:

services.AddMvc();

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });

});

* Añadimos al método Configure el siguiente código:

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");

});