



API C#

Documentación



El presente documento ha sido elaborado por Switch Software Solutions (Integralss SRL) conteniendo información confidencial, por lo que está expresamente prohibida la distribución a terceros, salvo se cuente con autorización explícita en forma escrita.

Contenido

[1 Esquema general](#_heading=h.3znysh7) 4

[2 API](#_heading=h.2et92p0) 4

[Program.cs](#_heading=h.tyjcwt) 4

[Properties/launchSettings.json](#_heading=h.3dy6vkm) 4

[appSettings.json](#_heading=h.1t3h5sf) 5

[Startup.cs](#_heading=h.4d34og8) 5

[Extensions](#_heading=h.2s8eyo1) 5

[Endpoints](#_heading=h.17dp8vu) 6

[Controller](#_heading=h.3rdcrjn) 6

[Resources](#_heading=h.26in1rg) 7

[Validators](#_heading=h.lnxbz9) 8

[Related entities](#_heading=h.35nkun2) 8

[Mapping](#_heading=h.1ksv4uv) 9

[3 Core](#_heading=h.44sinio) 9

[Models](#_heading=h.2jxsxqh) 9

[Services](#_heading=h.z337ya) 10

[Repositories](#_heading=h.3j2qqm3) 11

[IUnitOfWork.cs](#_heading=h.1y810tw) 12

[4 Services](#_heading=h.4i7ojhp) 13

[5 Data](#_heading=h.2xcytpi) 13

[Configurations](#_heading=h.3whwml4) 13

[Repositories](#_heading=h.1ci93xb) 13

[UnitOfWork.cs](#_heading=h.2bn6wsx) 13

[DbContext.cs](#_heading=h.qsh70q) 13

[6 Anexo A - Creación/Eliminación de un campo](#_heading=h.3as4poj) 13

[Tarea propuesta](#_heading=h.1pxezwc) 13

[Api.Resource](#_heading=h.49x2ik5) 14

[Api.Validations](#_heading=h.2p2csry) 14

[Core.Models](#_heading=h.147n2zr) 14

[Data.Configurations](#_heading=h.3o7alnk) 14

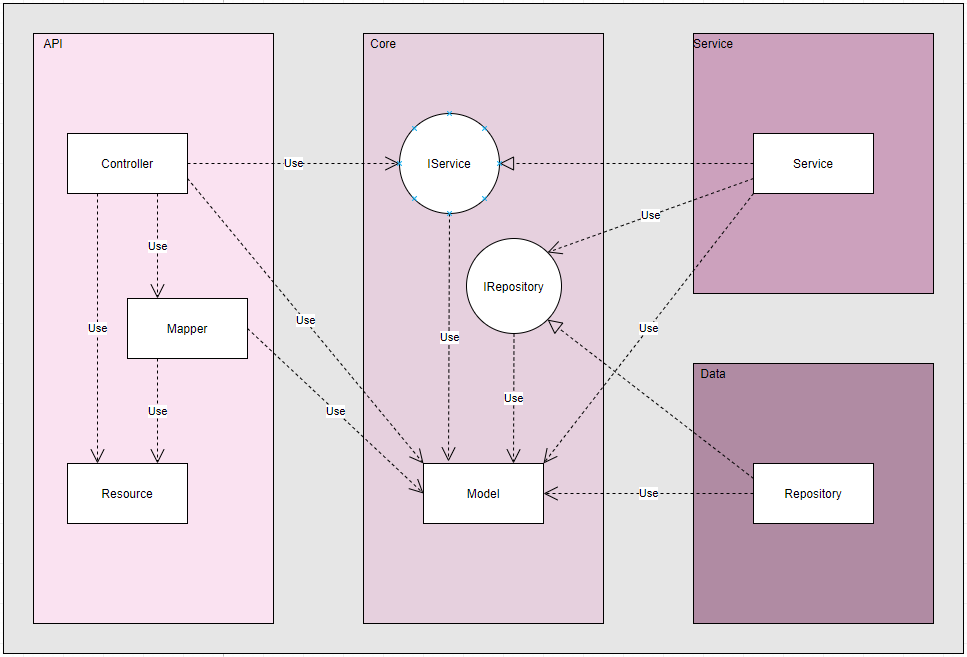
[Services](#_heading=h.23ckvvd) 14

[Migrations](#_heading=h.ihv636) 14

# 

# 

# Esquema general



# API

El proyecto API es la puerta de entrada al backend.

Los clientes realizarán sus peticiones a través de los endpoints expuestos por esta capa.

## Program.cs

Aquí comienza la ejecución del sistema.

Crea el Web host y define nuestra clase startup.

## Properties/launchSettings.json

En este json se configura como y donde se va a levantar el sitio.

Como se verá más adelante, una variable de entorno a tener en cuenta es

* "ASPNETCORE\_ENVIRONMENT": "Development"

## appSettings.json

En este json se configura la app, pudiendo personalizar cada configuración por ambiente. Esto se logra mediante el post-fijo que indica justamente en que ambiente estamos corriendo.

* appsettings.**Development**.json
* appsettings.**UAT**.json

Esto se define mediante la variable de entorno **ASPNETCORE\_ENVIRONMENT** antes mencionada, la que permite optar por uno u otra configuración[[1]](#footnote-0).

El string de conexión, la configuración del log y/o de la autorización de usuarios, son ejemplos de configuraciones que se pueden encontrar aquí.

## Startup.cs

Es la clase con la que se define la inicialización de la aplicación y es llamada por el framework al comienzo del proceso.

De manera muy similar a lo que se describió para appsettings, se puede definir la configuración para el ambiente en el que se está ejecutando (definido por la variable **ASPNETCORE\_ENVIRONMENT**). Esto se logra introduciendo el método Configure[enviroment](IApplicationBuilder app, IWebHostEnvironment env), donde [enviroment] es el nombre del ambiente. Esto hace que se ejecute este método y no el método Configure(IApplicationBuilder app, IWebHostEnvironment env).

Lo mismo sucede con el método ConfigureServices.

Con fines prácticos, en esta clase, incluimos todos los servicios que van a ser utilizados (inyectados) en nuestros controllers.

A modo de ejemplo:

* services.AddTransient<IUserService, UserService>();

## Extensions

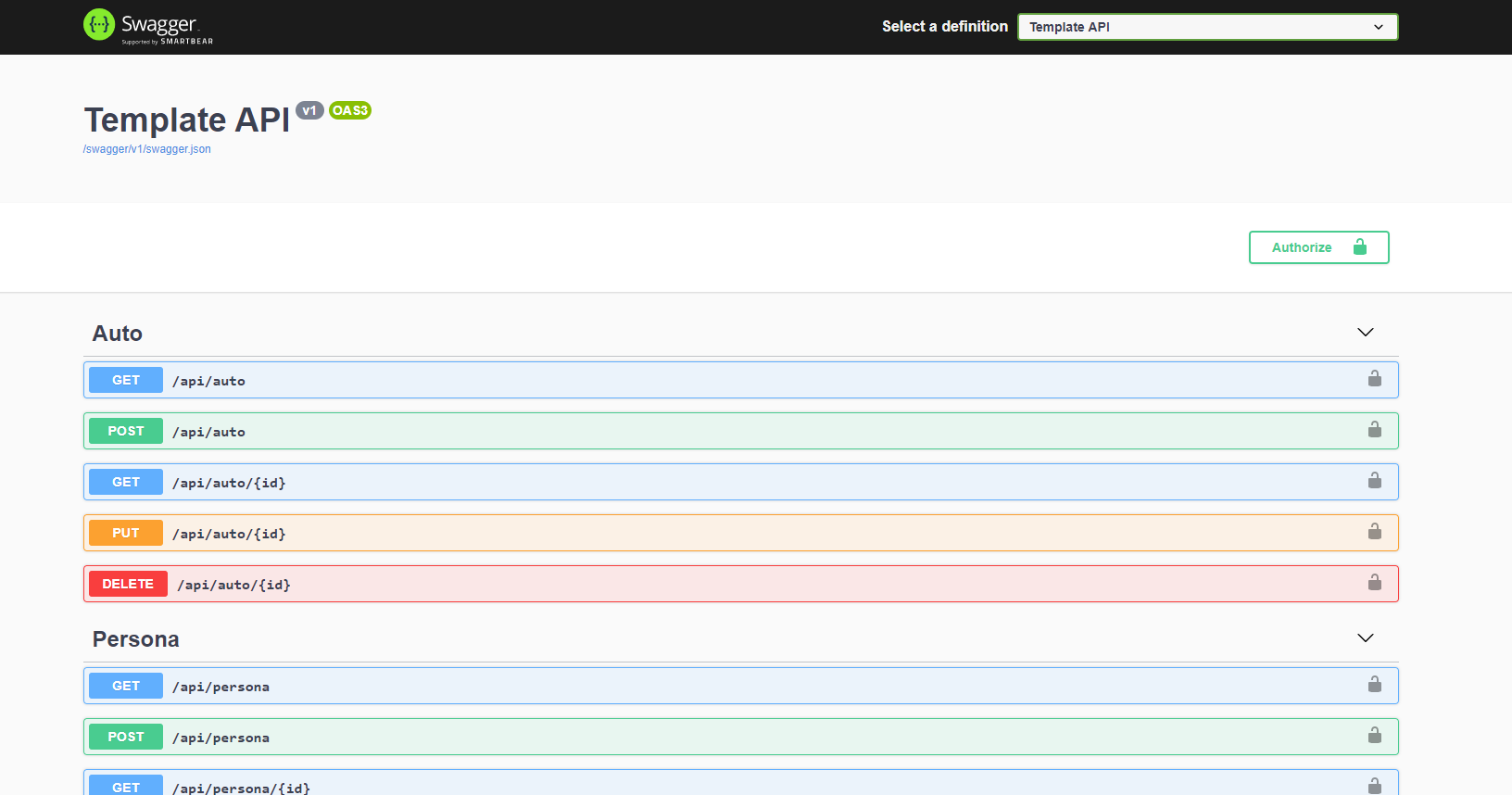
**ServiceExtensions.cs**

La clase tiene como objetivo nuclear las configuraciones de la aplicación, de manera de no sobrecargar de información nuestra clase startup. Desde esta última, se llaman a los métodos incluidos en ServiceExtensions.

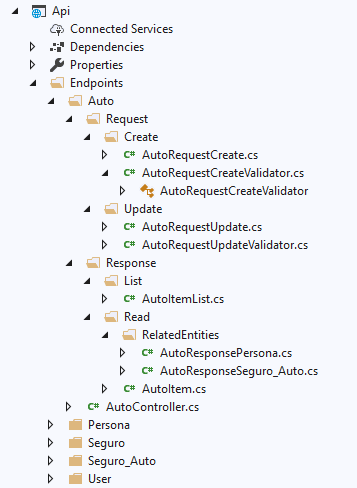
**Swagger**

Un punto a notar es la integración con Swagger incluida en ServiceExtensions.

En el método ConfigureSwagger se configura la generación automática de las vista provista por Swagger, donde se incluye el detalle de cada uno de los endpoints y la posibilidad de ejecutarlos de forma sencilla.



## Endpoints



La carpeta Endpoints contiene cada uno de los servicios expuestos, agrupoados por los controladores responsables de manejar los requests y responses.

En ejemplo desplegado en la imagen, se muestra la carpeta Auto, que incluye AutoController.cs como controlador y recursos (con sus validaciones) para los request y response para las operaciones Create, Update, Read y List.

### Controller

Los controllers son la entrada a la api.

A través de estos, el cliente interactúa con nuestra aplicación.

Aquí se define la firma de los servicios expuestos, lo que define los request, y las formas de respuestas que se darán, lo que define los response.

## Resources

Los objetos resource son responsables de transportar la información en la capa api.

Tanto las peticiones como las respuestas se hacen mediante el uso de estos objetos.

La estructura de cada uno se define de acuerdo a las necesidades de cada endpoint.

|  |
| --- |
|  |
|  |
| En este ejemplo se presenta el request para Create y Update, donde para el Update es requerido el Id del objeto como argumento extra. |

|  |
| --- |
|  |
|  |
| En este ejemplo se presenta el response para List y Read, donde para el response para List es más acotado, no incluyendo entidades relacionadas. |

## Validators

Aquí se realizan las validaciones sobre los objetos resource recibidos en request.

El objetivo es asegurar la correctitud de la información recibida desde el cliente.

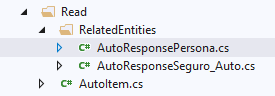


El objeto request para la operación Create de Auto debe de contener la marca del vehículo, con un máximo de 50 caracteres y el identificador de su dueño debe ser incluido.

## Related entities

Estas entidades no son más que entidades del tipo resource, asociadas al resource principal.

A modo de ejemplo, un response de Auto incluye la entidad relacionada Persona (su dueño).



## Mapping

La clase mapping se encarga de traspasar la información contenida en un objeto resource a un objeto model y visceversa.

Las capas siguientes en la solución manejan sus propios objetos de transporte de datos.

El mapping se lleva a cabo mediante la utilización de la librería AutoMapper.

Esta librería nos facilita este mapeado, copiando automáticamente los campos de nombres iguales (y tipos) de un objeto a otro.

De todas maneras, es posible configurar mapeos personalizados.

# Core

El proyecto Core define las interfaces para los servicios y los repositorios a usarse en la Api, así como también los modelos con los que se trabajará.

## Models

Estos objetos se mapean con la tablas en base de datos, aunque puede existir excepciones y/o diferencias.

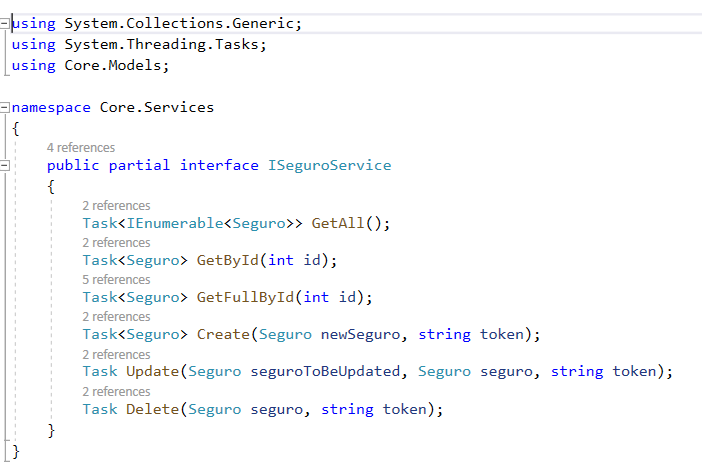
|  |
| --- |
|  |
|  |
| La entidad Persona tiene una colección de entidades Auto. |

## Services

Estas interfaces definen las funcionalidades expuestas para cada entidad.

El comportamiento definido aquí es el comportamiento definido para la lógica del negocio.

Inicialmente, se definen las operaciones más genéricas para cada una de las entidades (CRUD).



## Repositories

Estas interfaces definen las operaciones para el acceso a base de datos.

Inicialmente, se definen las operaciones más genéricas para cada una de las entidades.

Todas las interfaces heredan de IRepository, por lo que se maneja de forma genérica en esta interface, el comportamiento definido para dichas interfaces.

|  |
| --- |
|  |
|  |
| Inicialmente, el repositorio específico par la entidad Persona (y para todas las entidades), está vacío, estando la definición en la clase IRepository de la cual dicha entidad hereda. |

|  |
| --- |
|  |
| En caso de requerir un acceso a datos particular para una entidad, la definición del mismo debería de hacerse en la interfaz propia de la entidad. Esto generalmente se da con consultas a la base de datos particulares. |

## IUnitOfWork.cs

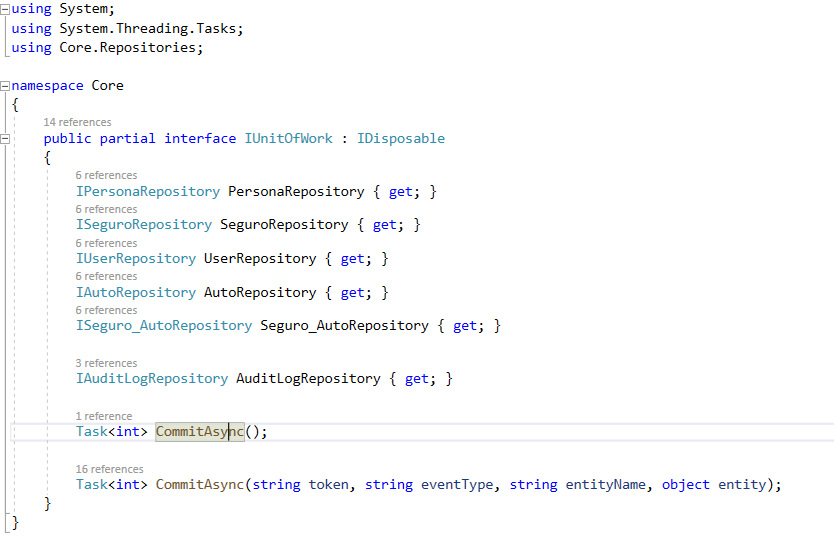
El patrón Unit of Work[[2]](#footnote-1) resuelve el problema de las modificaciones en base de datos de varias entidades en forma atómica, sin la necesidad de trabajar con las transacciones de forma “manual”.

En el caso normal, para efectuar un cambio en base de datos que aplique a más de una entidad, y realizarlo en un solo paso (o no realizarlo), se debe de manejar transacciones, promoviendo esto el agregado de errores si no se realiza de manera correcta.

El patrón mencionado resuelve esto teniendo instancias de todos los repositorios involucrados, trabajando a través de un solo contexto, para finalizar impactando en base con el método SaveChanges (en Entity Framework) que ejecuta todas las operaciones en forma atómica.

De esta forma se logra la ejecución (o no) en un paso sin la necesidad de implementar transacciones.

En esta interfaz se definen los Repositorios que manejará nuestra unidad de trabajo y los métodos commit disponibles.



En general, en nuestro caso, se utilizará el CommitAsync con parámetros, ya que la solución implementa un mecanismo de logueo de datos para las acciones que modifican la base de datos (Create, Update y Delete).

# Services

En este proyecto es donde se implementa la lógica del sistema.

Cada clase implementa su interfaz correspondiente.



El ejemplo muestra la clase AutoService, que inicialmente incluye los accesos a datos más genéricos.

# Data

En este proyecto se implementa el acceso a datos.

## Repositories

Son las entidades encargadas de manejar el acceso a datos para cada tipo.

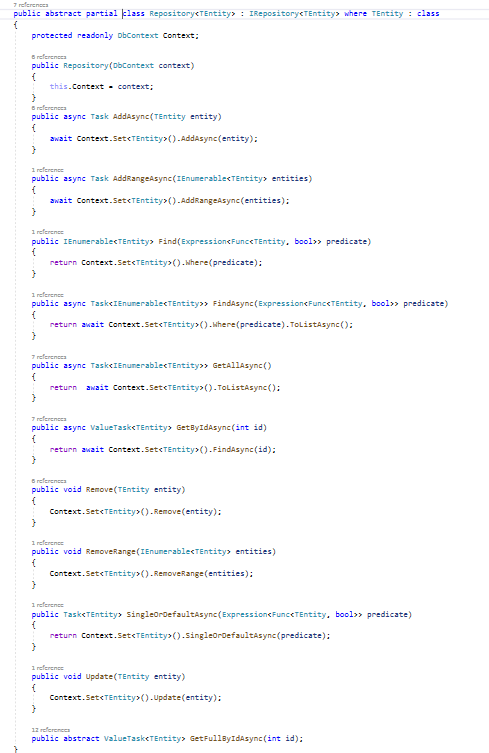
Todos los repositorios heredan análogamente a sus interfaces de Repository.



El único método que implementan es el GetFullIdAsync, ya que es particular a cada tipo de clase.

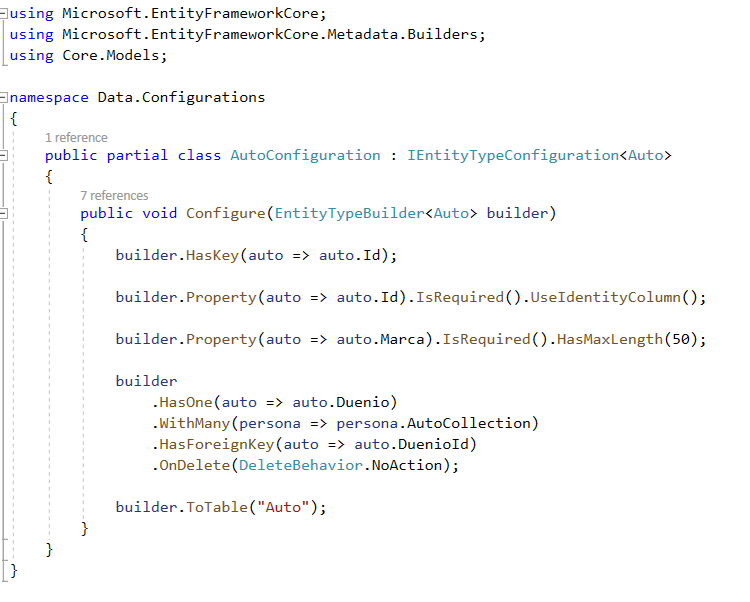
Los métodos restantes, una vez más, análogo a su interfaz +

, son implementados en la clase Repository de la cual heredan.



## Configurations

Estas configuraciones son las responsables de mapear las clases del Model a la base de datos.

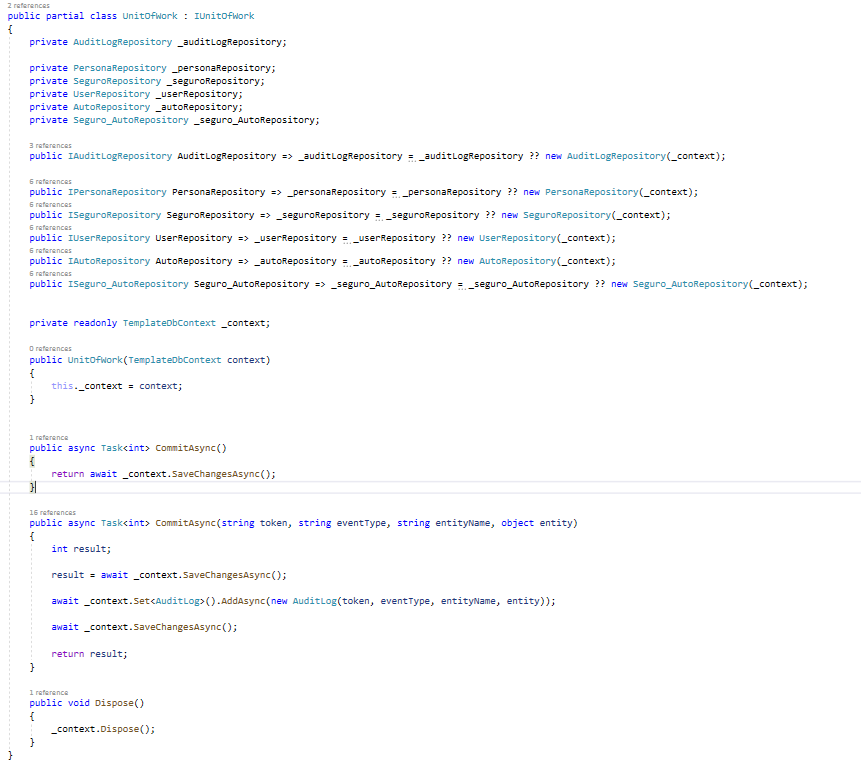


En el ejmplo se puede ver el mapeo de los atributos a los campos, así como las relaciones con otras tablas.[[3]](#footnote-2)

## UnitOfWork.cs

En esta clase se implementa la interface IUnitOfWork.

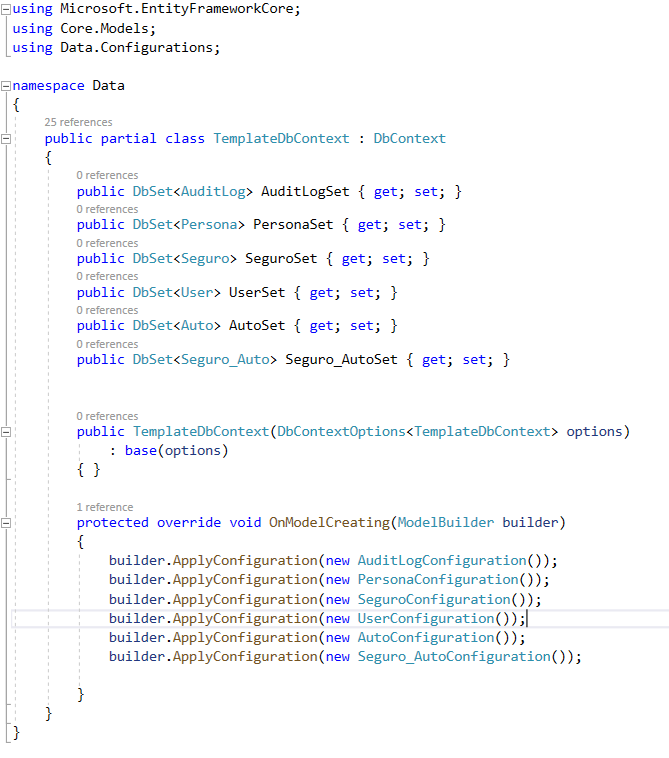
Aquí se definen los repositorios concretos que se incluirán, se define el contexto y se implementan los commit.



## DbContext.cs

Aquí se implementa el contexto a ser usado.

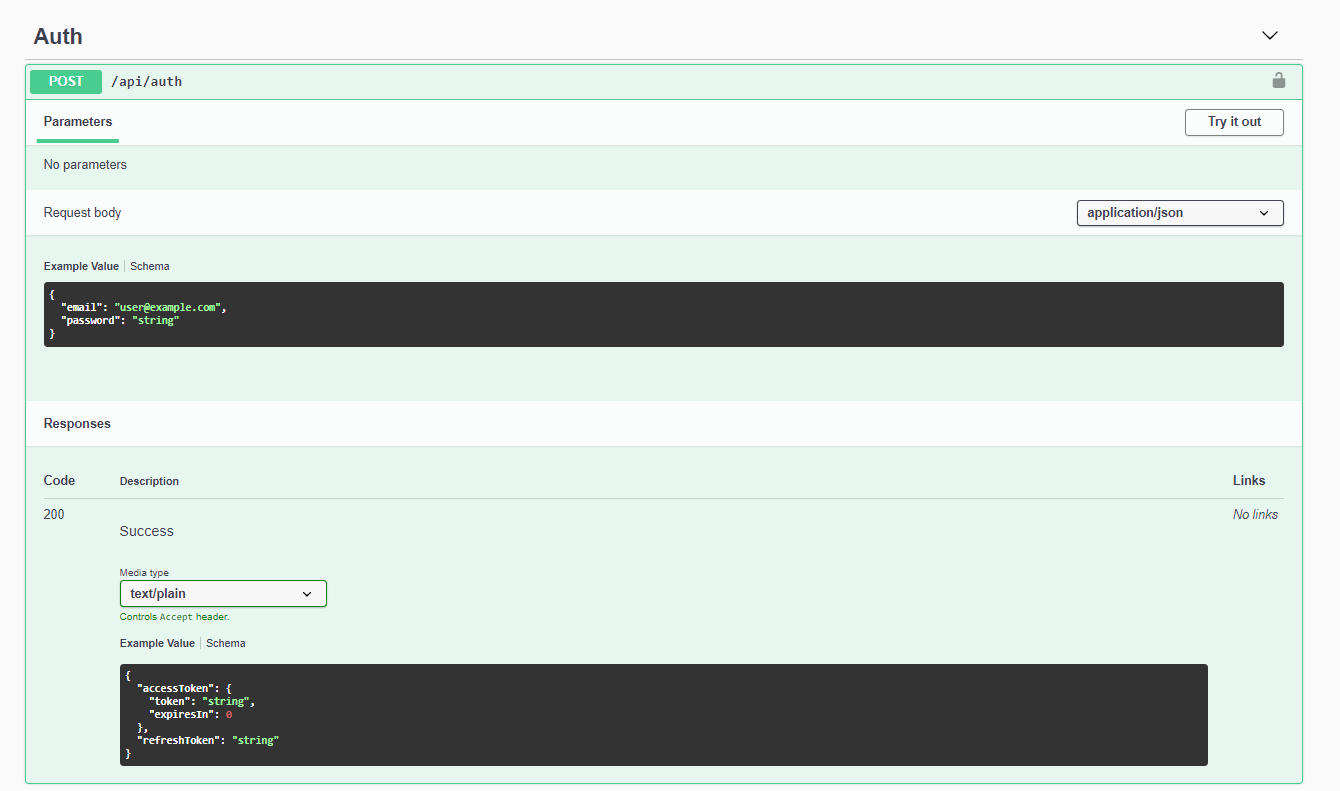
Básicamente se definen los set de entidades/tablas y se los instancias aplicandoles las configuraciones mencionadas en un capítulo anterior.



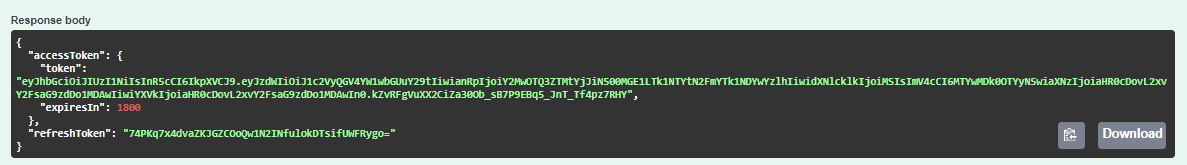
# Anexo A - Authorize con Swagger

La Api utiliza JWT para resolver la authorization en el sistema.

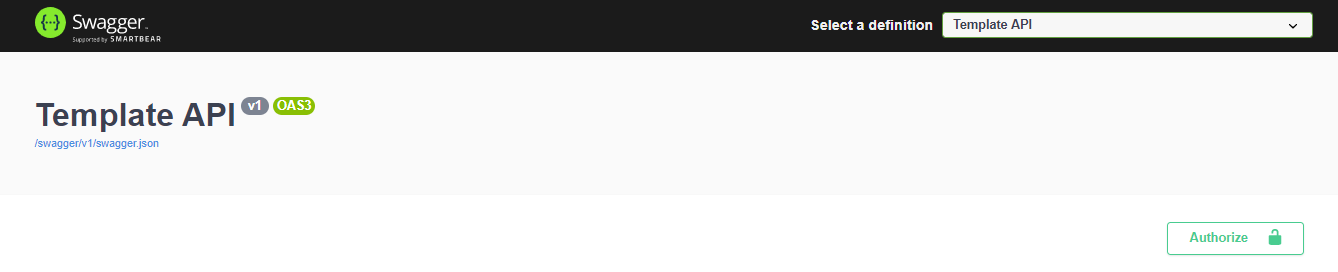
De esta manera, realizando un request a través del endpoin Api/Auth, obtendremos el token que nos habilita a acceder a endpoints no accesibles de forma anonima.



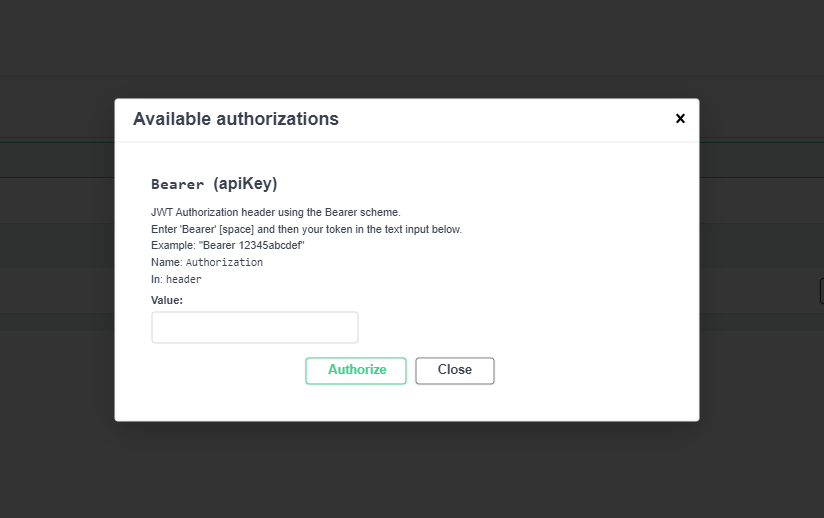
Seleccionando Try it out, ingresando los datos requeridos en formato json y luego dando click en el botón execute, obtendremos (si las credenciales son correctas) el token buscado.



Nos dirigimos ahora a Authorize, en la parte superior derecha de la pantalla



Al seleccionar el botón, se despliega un formulario con instrucciones a seguir para cumplir con la autorización.



En el campo de text Value debemos de tipear BEARER y a continuación pegar el token obtenido.

Ejemplo:

BEARER yJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VyQGV4YW1wbGUuY29tIiwian

Luego selecionamos Authorize y procedemos a cerrar el formulario.

Ya estamos en condiciones de acceder a los endpoints de la Api

# Anexo B - Creación/Eliminación de un campo

En esta sección se detalla el proceso de eliminación de un campo.

El escenario que se presenta, es que un campo agregado anteriormente, se entiende ahora que ya no es necesario y debemos eliminarlo.

Con eliminarlo, nos referimos a quitarlo de todas las capas de la aplicación y de la base de datos.

El ejemplo es análogo para la creación de un campo.

## Tarea propuesta

*THRCA-314 - Remove old unused columns from Services*

***Description***

*Column content1 on services are no longer valid. They have to be removed*

## Api.Resource

De los objetos resources relacionados con el objeto Service del Model, de la capa Core, se elimina el atributo Content1.

En las clases

* ServiceResourceCreateUpdateRequest
* ServiceResourceResponse

Se elimina el atributo

* public virtual String Content1 { get; set; }

## Api.Validations

Se elimina la validación relacionada con el campo Content1 en la clase ServiceCreateUpdateRequestValidator

* RuleFor(m => m.Content1).MaximumLength(250);

## Core.Models

En la clase Service, eliminamos el atributo de Content1

* public virtual String Content1 { get; set; }

## Data.Configurations

En la clase ServiceConfiguration eliminamos la configuración relacionada con la el campo Content1.

* builder.Property(service => service.Content1).HasMaxLength(250);

## Services

En la clase ServiceService, eliminamos del método Update, la línea relacionada con el campo Content1

* serviceToBeUpdated.Content1 = service.Content1;

## Migrations

Se generan las migrations que definen los cambios en la base.

En la carpeta de la solución (archivo .sln)

* dotnet ef --startup-project Api/ migrations add RemoveServiceContent1 -p Data/

Una vez completado el proceso, en la carpeta migrations del proyecto Data, deberíamos de encontrar un nuevo archivo

* 20200822185535\_RemoveServiceContent1

El contenido del archivo, debería contener las instrucciones necesarias para eliminar la columna referida

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropColumn(

name: "Content1",

table: "Service");

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<string>(

name: "Content1",

table: "Service",

type: "nvarchar(250)",

maxLength: 250,

nullable: true);

}

El método Up es ejecutado cuando se corre la migration y el método Down es ejecutado cuando se vuelve atrás esta.

Para actualizar la base de datos, a través de la migration recientemente creada ,desde la misma ubicación que ejecutamos la sentencia para crearla ejecutamos

* dotnet ef --startup-project Api/ database update

Una vez finalizado el proceso la base de datos debería haber sido afectada con la eliminación de la columna Content1 en la tabla Service.

# Versionado

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Responsable | Comentario | Versión |
| 19/08/2020 | Neri Custodio | Creación del documento | 1.0 |
| 23/09/2020 | Neri Custodio | Completado del total de puntos definidos | 1.1 |
| 23/09/2020 | Marcelo Bendahan | Revisión | 1.1 |
| 24/09/2020 | Neri Custodio | Agregado de integración con Swagger | 1.2 |

1. <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-3.1> [↑](#footnote-ref-0)
2. https://www.c-sharpcorner.com/UploadFile/b1df45/unit-of-work-in-repository-pattern/#:~:text=Unit%20of%20Work%20is%20referred,implementation%20of%20the%20Repository%20Pattern. [↑](#footnote-ref-1)
3. https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx [↑](#footnote-ref-2)