Lucas Ambesi - Noviembre del 2023

Ejercicio Desarrollo C# Senior AMV Travel

Instrucciones:

1. Cree un repositorio en Github
2. Desarrolle las soluciones en C# en el repositorio.
3. Realice el commit de sus cambios a su propio repositorio en GitHub.
4. Envíe el enlace del repositorio

Proyecto: Sistema de Reservas de Tours con API.

Parte 1: Modelado de Datos

Diseñe una estructura de datos para representar la información de reservas de tours. Incluya entidades como Cliente, Tour, Reserva y cualquier otra entidad relevante utilizando clases y relaciones.

Parte 2: Implementación del Sistema con API

Implemente el sistema de reservas de tours con una API utilizando ASP.NET. Cree los puntos de conexión necesarios para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades. Asegúrese de que la API sea segura y que se pueda acceder a través de métodos HTTP adecuados.

Parte 3: Documentación de la API – *No excluyente*

Documente la API utilizando Swagger o algún otro framework de documentación de API en C#. Proporcione descripciones claras de los puntos de conexión, los parámetros requeridos y las respuestas esperadas.

Parte 4: Integración con Proveedor Externo

Simule la integración con un proveedor externo de servicios turísticos. Diseñe e implemente un módulo que consulte información externa, como detalles de hoteles o atracciones turísticas, y la integre en su sistema.

Parte 5: Pruebas Unitarias y de Integración – *No excluyente*

Implemente pruebas unitarias para al menos una parte de su código. También, realice pruebas de integración para verificar la interoperabilidad de su sistema con el proveedor externo.

Resolución

1. Estrategia de resolución

Teniendo en cuenta los requisitos planteados se crearon dos soluciones las cuales exponen una APIs Rest respectivamente.

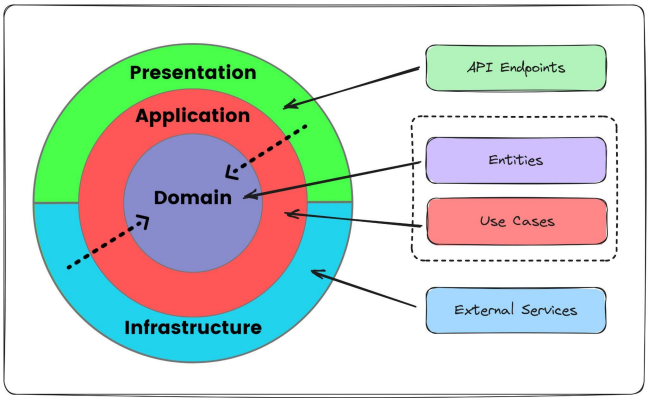
Tenemos una API principal bajo la solucion llamada "*AVMTravel.Tours*", y por otra parte, el rol del "Proveedor Externo" expuesto en la parte 4 de las instrucciones lo ocupa la API desarrollada en la solución *"AVMTravel.Accommodation".*

1.1 AVMTravel.Tours

Esta es la API principal en la cual se basó el desarrollo, en ella se exponen un conjunto de endpoints para dar solución al registro y login de clientes en el sistemas, creación, modificación y consulta de tours y locaciones, finalizando con reserva de un tour a partir de un cliente y tour.

1.1.1 Arquitectura general

Para el desarrollo de la API decidí basarme en los principios de la arquitectura ***"Clean Architecture"*** enfocado en el dominio, lo cual hace énfasis en la lógica de negocio y reduce las dependencias hacia la infraestructura externa. La principal función de este tipo de arquitectura es separar las preocupaciones de los distintos componentes.

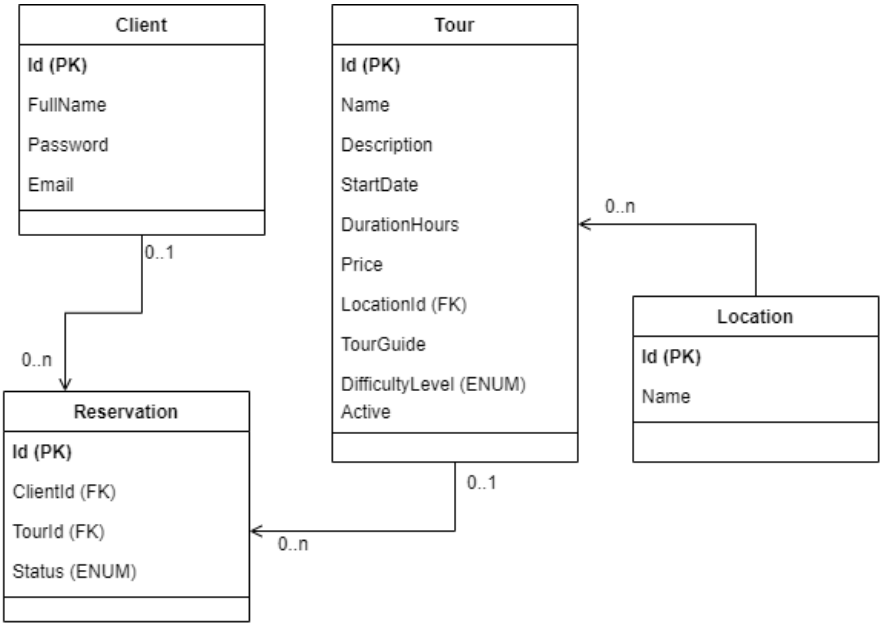


Si bien es un tipo de enfoque que se puede considerar que agrega complejidad para proyectos simples como lo es "AVMTravel.Tours", me pareció interesante desarrollarlo bajo esta modalidad para acercarme lo más posible a un caso real en magnitudes, complejidad y escalabilidad.

En cuanto al manejo del acceso y la manipulación de datos decidí utilizar Entity Framework en el cual entraremos más en detalle de las técnicas utilizadas más adelante, como migraciones, integración con LINQ y el mapeo Objeto-Relacional.

Siguiendo con el ecosistema de Microsoft decidí utilizar la base de datos relacional “Microsoft SQL Server”.

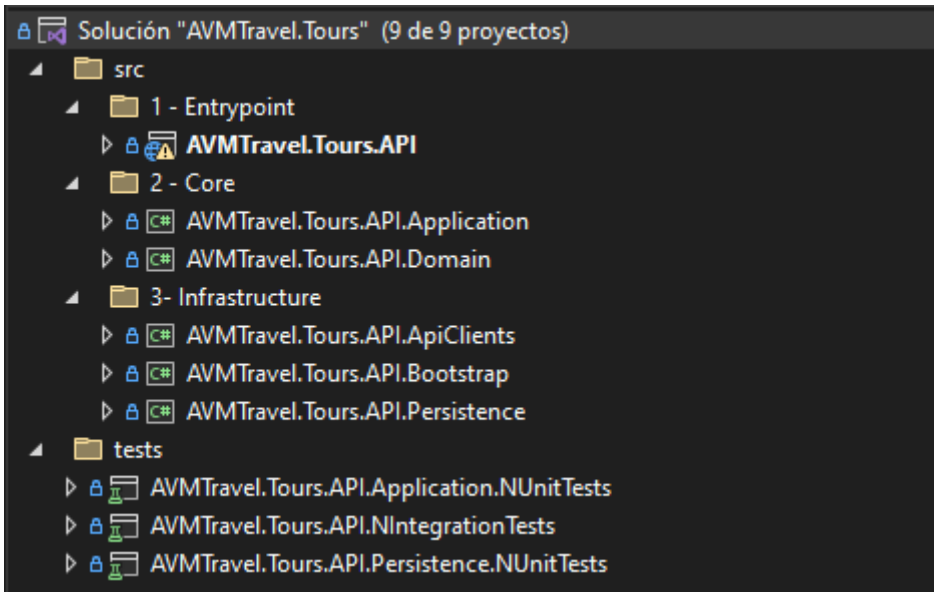
Podemos ver un Diagrama de clases estimado de la base de datos:



Ver DER en Draw.io

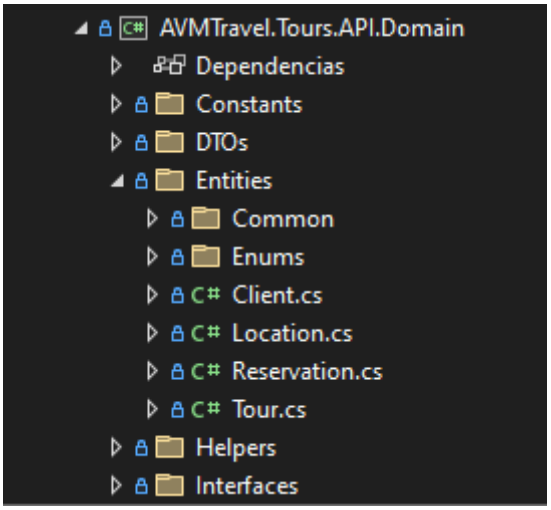
1.1.2. Proyectos

La solución cuenta con 9 proyectos, 1 proyecto ASP.NET Web API, 5 librerias de clases, 2 de pruebas unitarias y 1 de pruebas de integración, los cuales cumplen roles diferentes dentro de la API, en este apartado se explica la participación de cada uno de estos acompañado de técnicas, patrones y librerías utilizadas en el proceso.



1.1.2.1 AVMTravel.Tours.API.Domain

El dominio es el núcleo del sistema, que contiene las entidades y la lógica de negocio, tiene prioridad sobre cualquier otro componente. Esto garantiza que la arquitectura esté centrada en el dominio y no en detalles de implementación o tecnología.



En este proyecto contamos con los siguientes ítems a mencionar:

Entities

Tenemos la entidades definidas en el negocio, las principales son *Client*, *Location*, *Reservation*, *Tour* todas heredan de la clase *AuditableBaseEntity* la cual tiene la responsabilidad de llevar el control de fecha de creación y actualización.

DTOs

Data Transfer Object es un patrón de diseño que se utiliza para transferir datos entre subsistemas de una aplicación. En este caso lo utilizamos para la interacción entre el controlador, servicios y repositorios. Estos contienen datos y no tienen lógica de negocio adicional por lo que podemos resguardar y generar una capa de abstracción a las entidades.

Interfaces

En ella definimos los contratos para los services, querys y repositories que nos permite definir el conjunto de operaciones que las clase que herede de ellas deben implementar, sin especificar cómo se deben realizar esas operaciones.

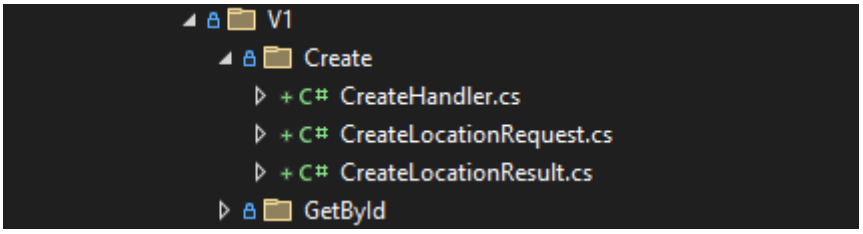
1.1.2.2 AVMTravel.Tours.API.Application

La capa de application representa la lógica de aplicación específica, incluyendo las reglas de negocio donde se definen y se implementan los casos de uso específicos de la aplicación.

En este proyecto tenemos los Casos de uso, los services, los validators y los mappings.

Casos de uso

Un caso de uso es una funcionalidad específica que proporciona valor al usuario. Por cada caso de uso tenemos un Handler, una request y un result o response.



En la **request** tenemos los parámetros de entrada al sistema y el **result** es lo que este retornara o su salida.

En cuanto al **Handler**, es el componente o clase que se encarga de manejar y procesar la solicitud específica.

Y con este último nos surge la necesidad de mencionar el patrón *Mediador* que es el que se utiliza para administrar la comunicación entre los casos de uso y los controllers.

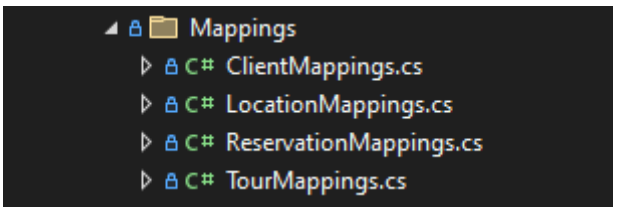
Patrón Mediator

El patrón Mediador (*Mediator*) es un patrón de diseño que define un objeto (llamado mediador) que centraliza la comunicación entre diferentes objetos, evitando que estos se comuniquen directamente entre sí. En lugar de tener conexiones directas entre todos los componentes, estos se comunican a través del mediador lo cual reduce el acoplamiento y hace que los componentes sean más independientes.

En este caso usamos la librería **MediaTr** que proporciona una implementación fácil de usar para la mediación de mensajes y eventos en aplicaciones en C#.

Mappings

Para mapeo de datos utilizo la librería **Automapper** que ofrece convenciones por defecto para mapear propiedades con nombres similares facilitando la conversión sin necesidad de escribir código manual. En la carpeta mappings se hacen las configuraciones de conversión.



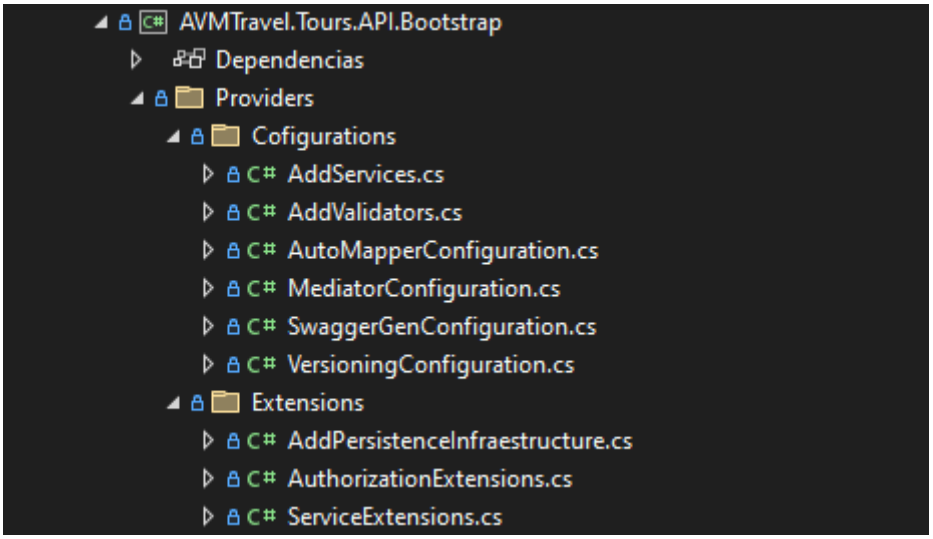
Validators

Utilizó la librería **FluentValidation** para validar los datos de la request (por ejemplo aquellos que son mandatorios) antes de realizar operaciones como la persistencia en la base de datos. Esto me proporciona la prevención de errores y el retorno claro al usuario de la falla o faltante en la request. El código de retorno elegido es ***400*** *- Bad request.*

Estas validaciones se encuentran en la carpeta llamada “Validators”.

1.1.2.3 AVMTravel.Tours.API.Bootstrap

Este proyecto es el encargado de proporcionar al Program de todas las herramientas necesarias para el correcto funcionamiento del sistema y su configuración.



En **Configurations** contamos con la configuración de herramientas y patrones que parten de librerías como mediatr, automapper, swagger y fluentValidation. También tenemos la versión de la API y la inyección de dependencias.

Versión de la API

Tenemos diferentes maneras de versionar una API, tales como la Versión en la URL, en el parámetro de consulta, cuerpo de la solicitud y en los headers.
En este caso opte por la opción del versionado en los headers, en el cual por defecto o caso de ingresar una versión superior se estable la última versión de la API.

```
options.ApiVersionReader = ApiVersionReader.Combine(
    new QueryStringApiVersionReader("version"),
    new HeaderApiVersionReader("x-version")
);
```

Además en los headers de la response nos informa la versión soportada.

Response headers

```
access-control-allow-methods: PUT,GET,HEAD,POST,DELETE,OPTIONS
access-control-allow-origin: https://localhost:7235
api-supported-versions: 1.0
content-type: application/json; charset=utf-8
date: Fri,24 Nov 2023 23:53:35 GMT
server: Kestrel
```

Inyección de dependencias

Es un patrón de diseño en el que una clase recibe las dependencias necesarias desde el exterior en lugar de crearlas internamente. Esto mejora la modularidad y la reutilización del código.

En él se establecen las dependencias en la configuración del proyecto.

```
public static IServiceCollection ConfigureServices(this IServiceCollection services)
{
    //Services
    services.AddScoped<ILocationService, LocationService>();
    services.AddScoped<ITourService, TourService>();
    services.AddScoped<IClientService, ClientService>();
    services.AddScoped<IReservationService, ReservationService>();
```

Para luego utilizar en las clases según sus necesidades como tal es el siguiente caso de LocationService en donde inyectamos IMapper, ILocationRepository y ILocationQuery.

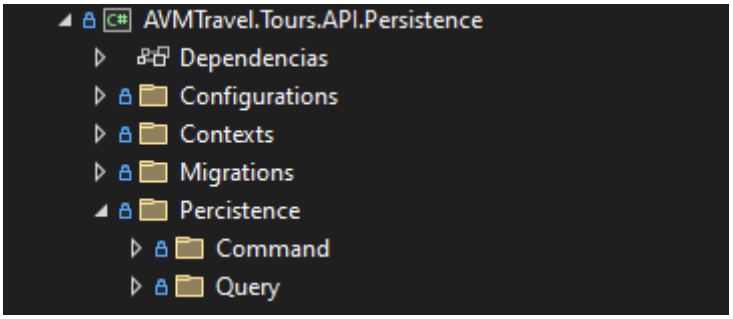
```
2 referencias
public class LocationService : ILocationService
{
    private readonly ILocationQuery _locationQuery;
    private readonly ILocationRepository _locationRepository;
    private readonly IMapper _mapper;

    0 referencias
    public LocationService(
        ILocationQuery locationQuery,
        ILocationRepository locationRepository,
        IMapper mapper)
    {
        _locationQuery = locationQuery;
        _locationRepository = locationRepository;
        _mapper = mapper;
    }
```

Por otro lado, en **Extensions** contamos con el agregado del contexto de DB creado en la capa de persistencia, los servicios ya mencionados y seguridad de autenticación, de la cual entramos en más detalle en el apartado de seguridad de la API.

1.1.2.4 AVMTravel.Tours.API.Persistence

El proyecto denominado “Persistence” es el encargado de la definición de un contexto y conexión con la base de datos.



Dándonos así la creación, manejo del acceso y la manipulación de datos.

Configurations y seeds

Las configuraciones de Entity Framework son la forma en que se especifican las reglas y configuraciones específicas de la base de datos para las entidades en el modelo.
Para ello se creó una Configuración para cada entidad que iba a ser representada en la base de datos como una tabla.

En cuanto a las seeds o semillas me refiero a los datos iniciales que se insertan en la base de datos al crearla por primera vez. En este caso se creó una seed para Location y Tour.

Migrations

Utilice migraciones para evolucionar el esquema de la base de datos a lo largo del tiempo, permitiendo cambios en la estructura de la base de datos sin perder los datos existentes.

Querys y Commands (LINQ)

En cuanto al acceso a los datos se hace una distinción entre operaciones de consulta (queries) y operaciones de comando (commands).

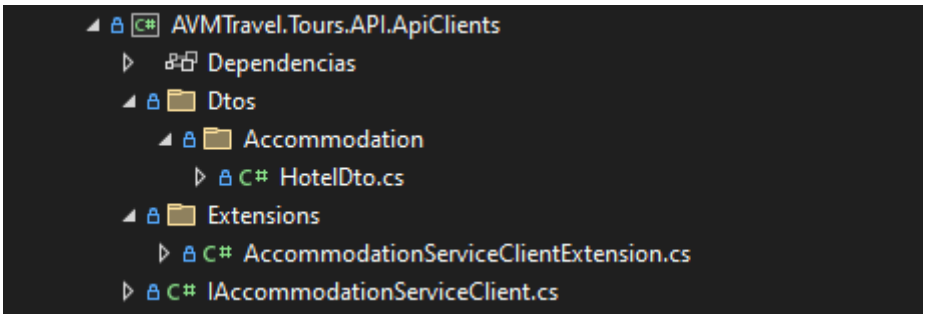
En donde las queries se utilizan para leer información sin modificar el estado de los datos y los commands se utilizan para modificar el estado de los datos, como insertar, actualizar o eliminar registros.

Utilizó Language-Integrated Query (LINQ) para consultar y manipular colecciones de datos de manera declarativa.

1.1.2.5 AVMTravel.Tours.API.ApiClients

En este proyecto se gestiona la conexión con proveedores y servicios externos, en él tenemos los clients y la inyección de dependencias del mismo junto con los dtos para transformar los datos recibidos.
Por cada servicio integrado se debe crear un cliente independiente.

En este caso nos comunicamos con la [API de AVMTravel.Accommodation] la cual las utilizamos para obtener hoteles en base a un id de locación.



Para realizar esta prueba de funcionamiento en el Controller de GetById Location, una vez recuperada la Locación se utiliza el id de la misma para consultar a la API de *Accommodation* y obtener los hoteles vinculados a la locación, esto lo podemos ver en el Handler del caso de uso.
Los hoteles se agregan al resultado final que da la API retornarlos en el result.

```
try
{
    var hotels = await _accommodationServiceClient.GetHotelByLocationIdAsync(location.Id);

    if (hotels != null)
    {
        result.Hotels = hotels.ToList();
    }
}
catch(Exception ex)
{
    result.Hotels = null;
}
```

1.1.2.6 AVMTravel.Tours.API

Este proyecto es el punto de entrada al sistema, es un proyecto ASP.NET Core Web API para servicios RESTful HTTP.

Controllers

En él se exponen los controladores, estos son los siguiente:

CLIENT CONTROLLER
- GET - /api/clients/{id}

LOGIN CONTROLLER
- POST - /api/login

REGISTER CONTROLLER
- POST - /api/register

TOUR CONTROLLER
- POST - /api/tours
- PUT - /api/tours
- DELETE - /api/tours/{id}
- GET - /api/tours/{id}

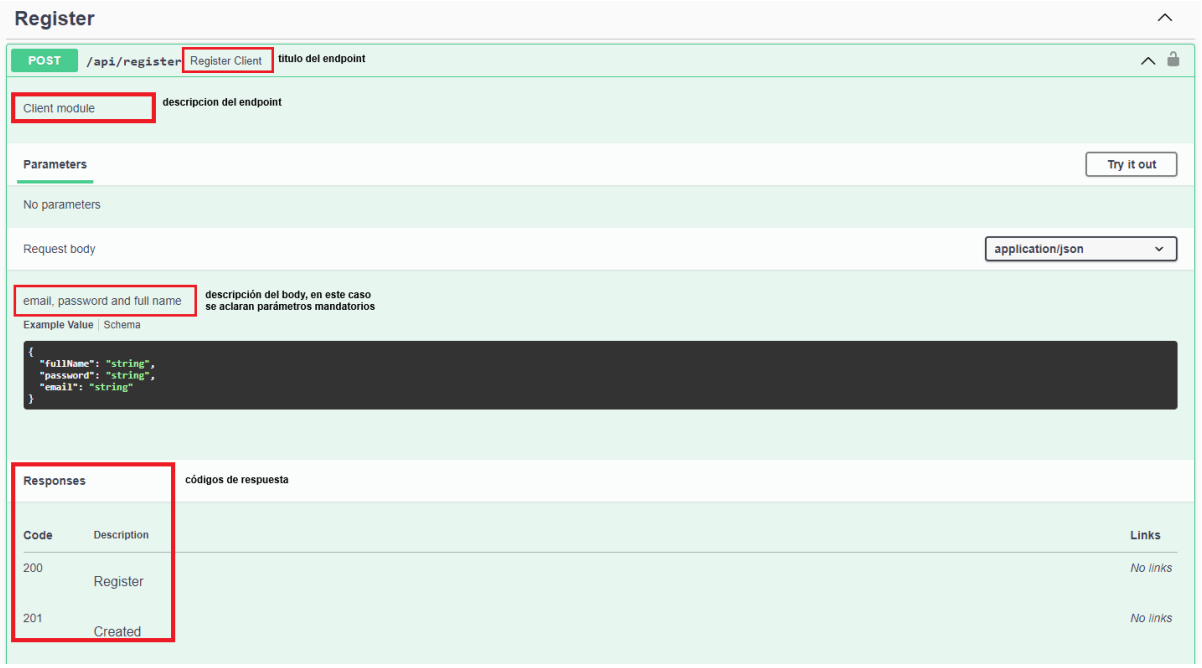
RESERVATION CONTROLLER
- GET - /api/reservations/{id}
- POST - /api/reservations
- PATCH - /api/reservations/status

LOCATION CONTROLLER
- GET - /api/locations/{id}
- POST - /api/locations

Documentación con Swagger

Para documentar la API se eligió swagger que es el recomendado por .NET y ya esta incluido cuando se crear un nuevo proyecto ASP.NET Web API, además se le incorporó información relevante en cada endpoint, esta información se especifica en cada controller y luego se inyecta en el swagger mediante un archivo autogenerado (configurado desde Program).

Un ejemplo es el siguiente:



1.1.3 Seguridad

En términos de seguridad se implementó un Token de Acceso para autenticar solicitudes.

JSON Web Tokens

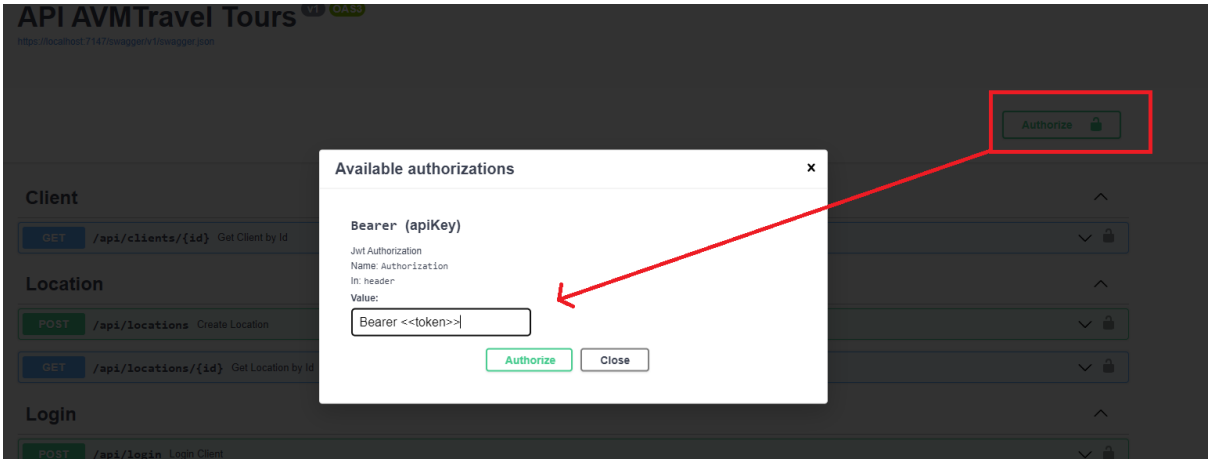
JSON Web Tokens (JWT) es un estándar abierto que define un formato compacto y autocontenible para representar información de manera segura entre dos partes. En él se genera un token separado en 3 partes. El **Header**, en el que especificó el tipo de algoritmo de encriptación (HS256), el **payload** donde se hace la carga de información relevante, llamados *claims*, en este caso incorpore el id de cliente y email, y por último la **firma** donde se garantizar que el mensaje no haya cambiado en el camino.

Dentro del proyecto Bootstrap en la carpeta Extensión está el archivo "*AuthorizationExtensions*" donde se realiza la configuración del token, además en la carpeta Configurations dentro del archivo "*SwaggerGenConfiguration*" se incorpora el botón de autenticación al Swagger para realizar la utilización desde la interfaz.

Proceso de autenticación

Una vez realizado el registro, el cliente se debe logear, en este caso se puede realizar desde el Swagger con el endpoint "*/api/login*", este login retorna el token de acceso que debe ser ingresado el botón Authorize.

Aclaración: recordar agregar la palabra "Bearer" al comienzo del token obtenido el endpoint de login separada con un espacio.



Esto nos permitirá utilizar los endpoints protegidos, a continuación se detallan los mismos:

Controllers con autenticación

- GET - /api/clients/{id}
- GET - /api/reservations/{id}
- POST - /api/reservations
- PATCH - /api/reservations/status
- POST - /api/tours
- PUT - /api/tours
- DELETE - /api/tours/{id}
- GET - /api/tours/{id}

Controllers sin autenticación

- POST - /api/login
- POST - /api/register
- GET - /api/locations/{id}
- POST - /api/locations

Encriptación de contraseñas

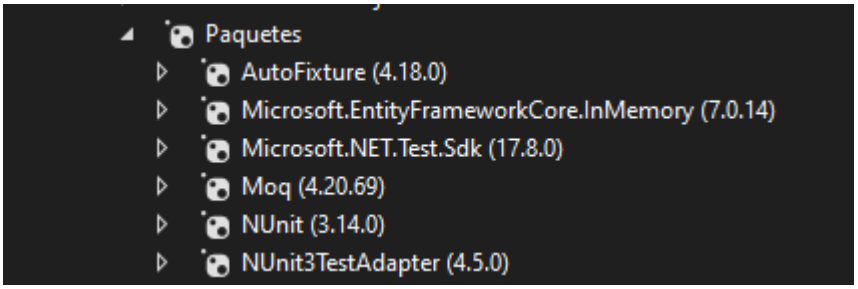
Con respecto a la administración de contraseñas de clientes se implementó un proceso de encriptación de contraseña antes de la persistencia en la base de datos, lo cual previene la

filtración de información en la base de datos, ya que nadie que no posea la key de desencriptado puede leerlo.
Dentro del proyecto “Domain” en la carpeta helpers se encuentra el archivo **Encrypt** con el proceso de encriptado.

1.1.3 Tests

Para realizar las pruebas se hicieron test unitarios y de integración. Se eligió el módulo de Locations para cubrir todo el proceso con pruebas.

Para esto se usaron las siguientes librerías en todos los proyectos



Tenemos tres proyectos de pruebas.

AVMTravel.Tours.API.Persistence.NUnitTests

En él se realizan las pruebas unitarias de Location referentes a la persistencia. Para esto se crea un nuevo contexto de base de datos pero esta se encuentra en memoria y se crea al momento de ejecutar las pruebas, esta configuración se encuentra en el archivo “*ContextDbMock*”. Además se realizó el mocking de la clase IMapper.

En este proyecto se realizaron las pruebas de Query y Commands con acceso real a la base de datos generada para realizar las mismas.

AVMTravel.Tours.API.Application.NUnitTests

En él se realizan las pruebas unitarias de Location referentes a los servicios. Ya que los Query y Command se testean en el proyecto de Percistence en este caso podemos realizar un Mock de ambos, además del IMapper.
Además se hacen las pruebas unitarias referentes a los casos de uso. Ya que los servicios se probaron ahora podemos hacer un Mock de este y probar el funcionamiento del Handler.

AVMTravel.Tours.API.NIntegrationTests

En él se hace la prueba completa del Controller de Location. Lo que primero crea una nueva Locación y luego se recupera a partir del Id asignado en la creación.

Para esto es necesario realizar las inyecciones de dependencias de todos los servicios utilizados en el proceso, definir un nuevo contexto con una base de datos en memoria como

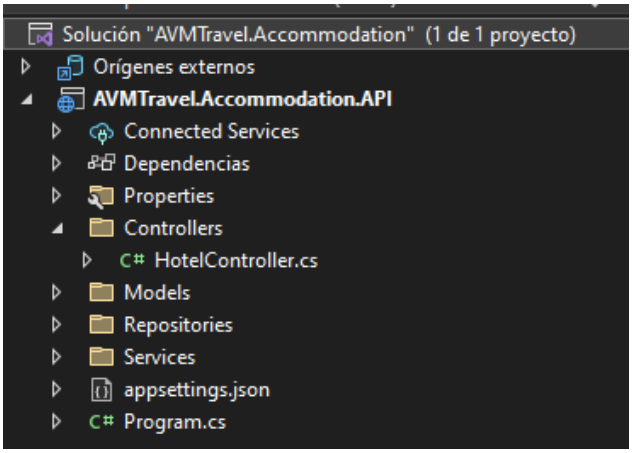
se realizó en el proyecto de test de Persistence y un Mock de IMapper. Todas estas configuraciones se encuentran en la carpeta “Common”.

1.2 AVMTravel.Accommodation

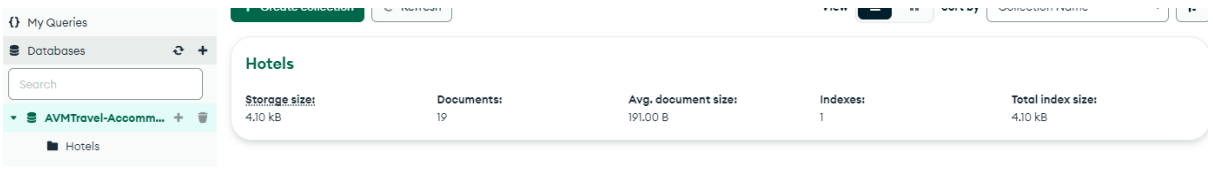
La API nace a partir de la necesidad de tener una api externa para la resolución de la parte 4 de los requisitos. Esta es una API pequeña que tiene como objetivo la consulta de hoteles a partir de un id de Locación.

1.2.1 Arquitectura general

En ella contamos con un proyecto ASP.NET Core Web API para servicios RESTful HTTP. Dentro del proyecto tenemos una arquitectura de 3 capas (Controllers - Services - Repositories).



Y una base de datos no relacional MongoDB, la cual contiene una sola colección llamada “*Hotels*”. Para la comunicación con la misma desde la API se utiliza el contexto proporcionado por la librería “*MongoDB.Driver*”.



1.2.2 Implementación

La API contiene 4 endpoints, dos de creación de hoteles (individual y masivo) y dos de consulta de hoteles, por id y locación.

La API gira en torno a la entidad **Hotel** definida en Models. Esta cuenta con propiedades consideradas relevantes como Name, Description, Address, Rating y un referencia al id de Locación.

2. Instalacion y configuracion

2.1 Tecnología y herramientas necesarias

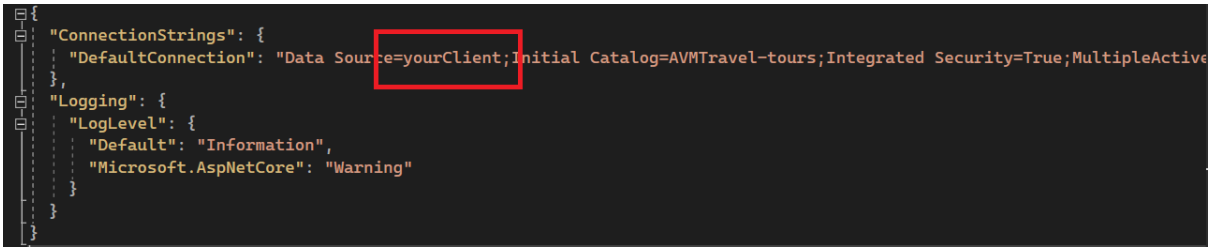
En este apartado se mencionan todas las tecnologías y herramientas necesarias para el funcionamiento de ambas APIs

- Git
- Visual Studio
- Microsoft SQL Server
- MongoDB y MongoDB Compass

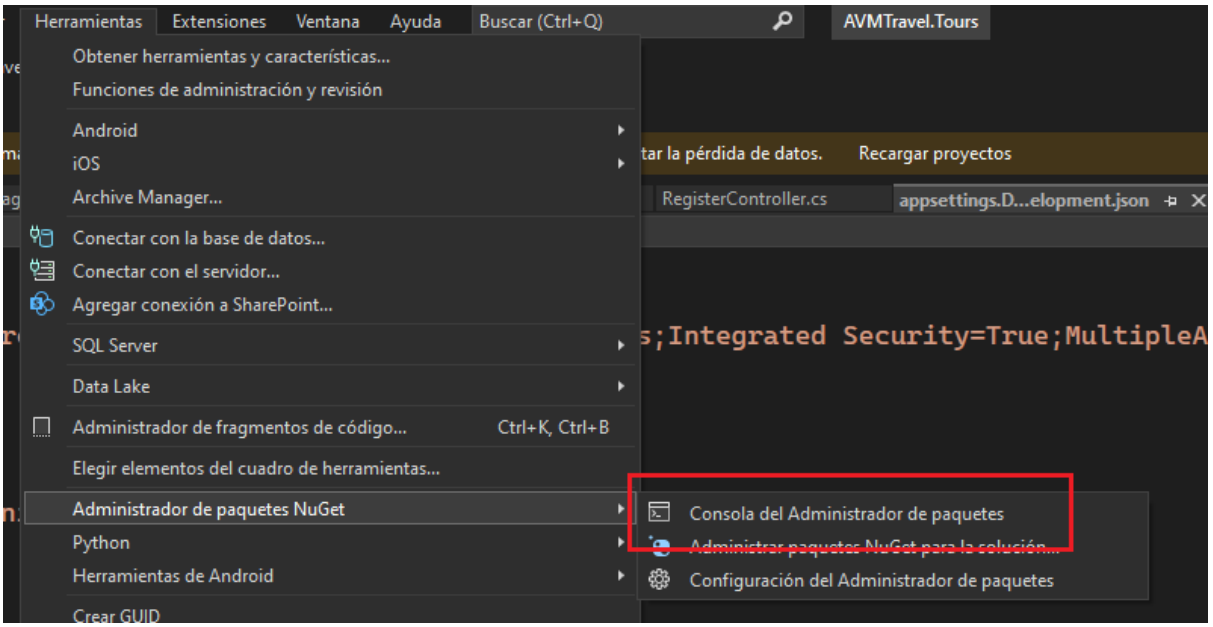
2.2 AVMTravel.Tours

Paso 1: Una vez clonado el proyecto se deben restaurar las dependencias, esto se administra de manera automática por visual studio pero en caso de que no pase se puede ejecutar de manera manual.

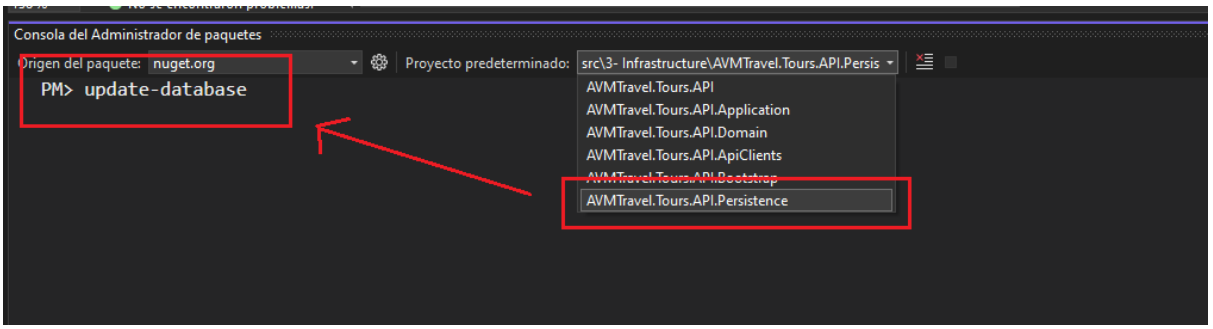
Paso 2: Cambiar connection strings, donde se debe cambiar "*yourClient*" por el nombre de usuario de su motor de base de datos en el DataSource. Este se encuentra en "appsettings.Development" del proyecto "AVMTravel.Tours.API"



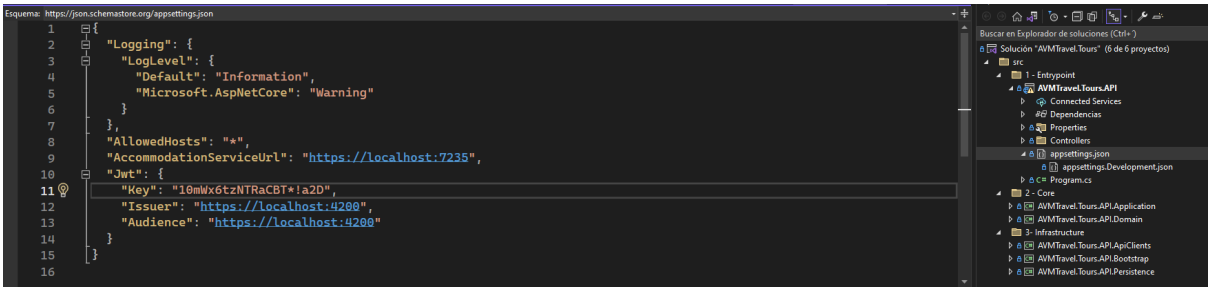
Paso 3: Si las migraciones no se ejecutan al levantar el proyecto (solo lo hacen si está en modo release) estas se pueden ejecutar de manera manual. Para esto se debe abrir la consola de Administrador de paquetes.



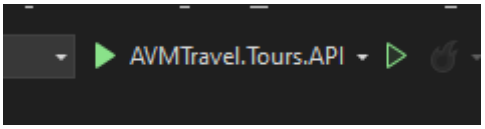
Esto abrirá una consola en la parte inferior, en donde se debe seleccionar el proyecto “AVMTravel.Tours.API.Persistence” y ejecutar el comando “update-database”



Paso 4: Revisar que el puerto de la ruta “AccommodationServiceUrl” coincida con el que expone la api **AVMTravel.Accommodations**. Esto se encuentra en appsettings dentro de AVMTravel.Tours.API



Paso 5: Ejecutar el proyecto **“**AVMTravel.Tours.API**”**



2.3 AVMTravel.Accommodations

Paso 1: Una vez clonado el proyecto se deben restaurar las dependencias, esto se administra de manera automática por visual studio pero en caso de que no pase se puede ejecutar de manera manual.

Paso 2: Revisar el puerto en donde se expondrá la api, esto nos sirve simplemente para la conexión con la api principal (paso 4 de AVMTravel.Tours)

Paso 3: La base de datos y sus colecciones se creará cuando se levante el proyecto pero esta no se mostrará en el MongoDB compass hasta que haya una inserción en las colecciones. Es recomendable revisar que el puerto de conexión definido en el archivo “*MongoDbRepository*” coincida con el de su configuración.

Paso 4: Para la inserción de datos hay un archivo en la ruta base del proyecto llamado “**entry-data-hotels.json**” el cual contiene un array de hoteles, este se puede insertar mediante el endpoint “*POST - api/hotels/many*”