Proyecto NewShore AIR

El proyecto tiene la siguiente estructura:

NewShore
- src
  - Common
  - Core
    - NewShore.Application
    - NewShore.Domain
  - Infrastructure
    - NewShore.Infrastructure
    - NewShore.Persistence
  - Presentation
    - NewShore.Api
    - NewShore.Api.Client
- tests
  - NewShore.Api.FunctionalTests
  - NewShore.Application.IntegrationTests
  - NewShore.Domain.UnitTests

NewShore.Api.Client
- openapi

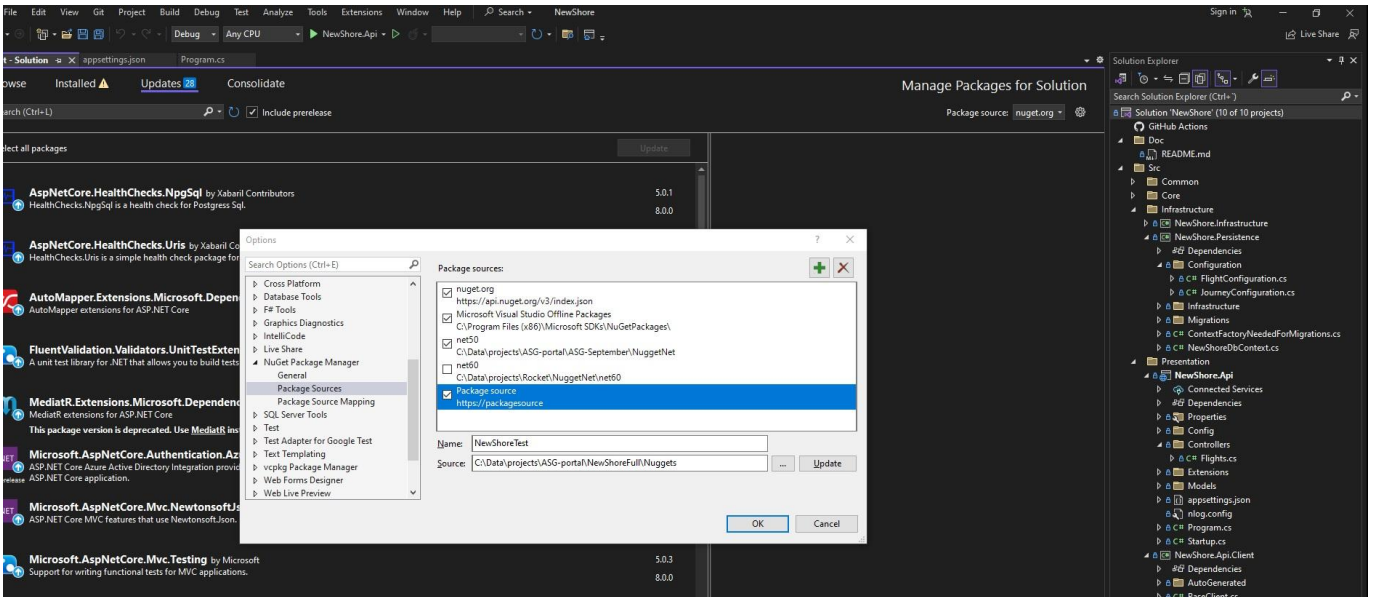
Nuggets

La carpeta **NewShore** Contiene el proyecto con el Servicio solicitado

La carpeta **NewShore.Api**.Client tiene un proyecto para generar el client para el consumo de la REST API. En este caso se genera un client.API a partir de la especificación que hay en el fichero: **newshore.apiclient/openapi/NewShoreAIR.json.**
Una vez generado el clienteAPI, se pone el nugget generado en el directorio Nuggets, de esta manera en caso de necesitar modificar el contrato del RESTAPI, solo hay que cambiar el JSON y generar un nuevo clientAPI.

En la carpeta **Nuggets** están las diferentes librerías utilizadas, como el clientAPI para acceder al servicio REST API , y así como un framework interno utilizado (y e esta manera el código es solo de un servicio y no tiene código duplicado de otros posibles servicios)

Para poder compilar el proyecto se debe configurar los nuggets necesarios



El primer punto se puede ver en las siguientes carpetas que está el modelado de clases:

NewShore\src\Core\NewShore.Domain\Entities
NewShore\src\Core\NewShore.Domain\\ValueObjects

El segundo punto, se ha creado un cliente RESTAPI en el proyecto **NewShore.Api.Client,** se consume con las siguientes características:
Se define en el appsettings:

```
 "HttpClientServices": {
        "Timeout": 30, // In seconds
        "NewShoreAIR": {
        "BaseUrl": "https://recruiting-api.newshore.es/api/"
        }
 },
```

De manera que se puede cambiar la URL base en cualquier momento sin afectar en nada.

Se ha creado un servicio que utiliza el clientAPI y consume el servicio por REST API.
NewShoreFull\NewShore\src\Infrastructure\NewShore.Infrastructure\Services\NewShoreAIR Service.cs

En el caso de querer optimizar el número de peticiones a la API y teniendo en cuenta que se indica que siempre devolverá los mismos valores, la solución que se ha realizado es con un Singleton, el cual solo hará una petición y guardará en memoria el resultado de dicha petición, el resto de veces que sea llamado, en lugar de consumir de servicio, devolverá lo que tiene en memoria.

El tercer punto, se ha creado un endpoint el cual utilizando Mediator llama a NewShore\src\Core\NewShore.Application\Flights\Queries\GetFlights\GetFlightQueryHandlerValidation.cs, de esta manera valida los datos entrados, y en caso de estar todo correcto se llama a GetFlightQueryHandler.cs que realiza las operaciones oportunas.

En el caso actual se ha optado por realizar las operaciones de vuelos múltiples en la Version1. En el caso de querer solucionar con vuelos de ida y vuelta, no tenía claro si los vuelos indican si son de ida o vuelta o se considera que se tenía que buscar en la vuelta el origen como destino y el destino como origen. En cualquier caso la solución se ha creado como Version2.

Comentar que el guardar a BD la consulta de vuelo se hace con un evento de Dominio, esto es por dos motivos: Primero GetFlightQueryHandler está encargado de consultar vuelos, si tiene que guardar en BD, ya está realizando dos acciones y eso no es del todo correcto. Segundo al utilizar un evento de dominio, liberamos al usuario de tener que esperar que se guarde en BD algo que a él no le interesa y que puede provocar un retraso a la hora de devolver su consulta.

El cuarto punto, pues he utilizado Entity framework Core, indicando la relación entre los diferentes objetos, NewShore\src\Infrastructure\NewShore.Persistence\Configuration\FlightConfiguration.cs y JourneyConfiguration.cs. Para ver como quedan las tablas se puede ver en: NewShore\src\Infrastructure\NewShore.Persistence\Migrations*

Para finalizar indicar que se ha utilizado Mapping NewShoreFull\NewShore\src\Infrastructure\NewShore.Infrastructure\AutoMapper\InfrastructureProfile.cs, para así tener las copias más fáciles y controladas.

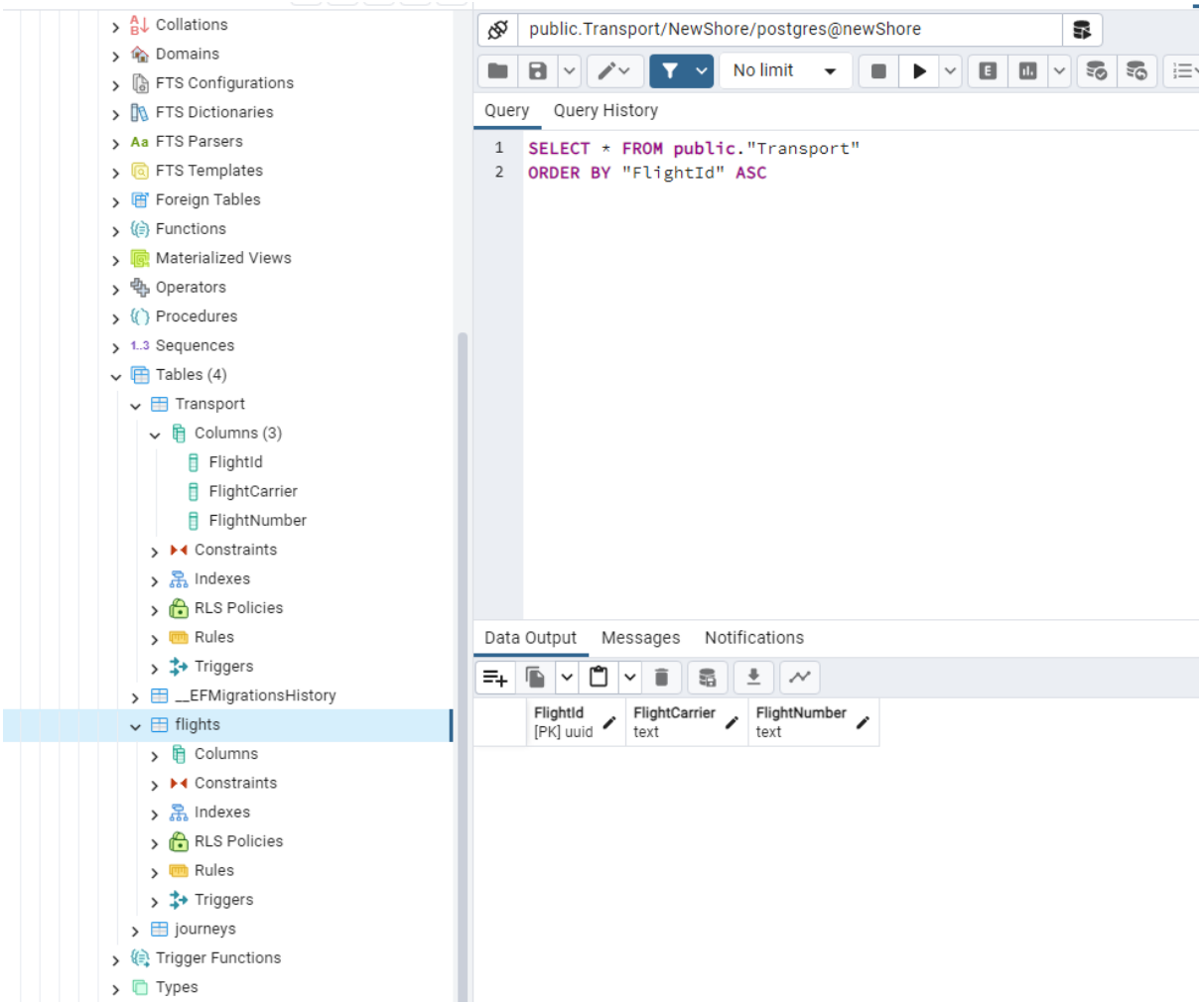
Lo último a comentar es: La BD se crea al encender el servicio gracias a: startup.cs

```
var host = CreateHostBuilder(args).Build()
        .UseIf(configuration.GetValue<bool>("Database:AutomaticMigrations"), host
=>
        {
        return host.MigrateDataBase<NewShoreDbContext>();
        })
```

Se configura la BD por appsettings:

```
"ConnectionStrings": {
        "NewShoreConnection":
"Host=localhost;Port=6000;Username=postgres;Password=postgres;Database=NewShore;K
eepalive=10;"
 },
```

Se ha comprobado que se crea con una imagen de Docker:
>docker run -d -e POSTGRES_PASSWORD=postgres --name postgres -p 6000:5432 postgres



Los UnitTests:

En este caso quería mencionar que todas las pruebas (al no tener un servicio que me devolviera posibles vuelos) se han realizado a partir de los uniTests. Podría haber creado mi propio servicio, e incluso podría haber utilizado un “mock” hardCode en el proyecto, pero al tener que realizar los UnitTests, pues lo he aprovechado “para todo”.

El nivel de cobertura de los tests he intentado que sea del 100% aunque eso no quiere decir que todos los casos están cubiertos pues eso provocaría que hubiera muchos más tests y no creo sea el objetivo primordial de la prueba, aun así me gustaría mencionar que un UniTest solo debe probar una sola cosa, y que los unittests siempre tienen que controlar el caso positivo y el caso negativo.