

**UNIVERSIDAD TECNOLÓGICA NACIONAL**  
**FACULTAD REGIONAL RESISTENCIA**

*Carrera de Ingeniería en Sistemas de Información*

***Materia: Diseño de Aplicaciones Cliente  
Servidor***

*Año: 2022*

**TRABAJO PRÁCTICO INTEGRADOR**

**Primera Entrega**

*Profesores: Teoría: Villaverde, Jorge Eduardo*

*Práctica: Quevedo, Fabricio*

*Integrantes:*

*Bian, Juan Lucas ( [juanlucasbiain@gmail.com](mailto:juanlucasbiain@gmail.com) )*

*Stangaferro, Guido ( [guidostangaferro@gmail.com](mailto:guidostangaferro@gmail.com) )*

*Vera, Gaston ( [vera.gastonn@gmail.com](mailto:vera.gastonn@gmail.com) )*

*Villa, Ricardo ( [tec.villa.ricardo@gmail.com](mailto:tec.villa.ricardo@gmail.com) )*

*Zapata, Rodrigo ( [facultadrodrigozapata@gmail.com](mailto:facultadrodrigozapata@gmail.com) )*

# Índice

1. Que tecnología utilizamos? .....	3
2. Que arquitectura utilizamos? .....	3
a. Programacion por Capas .....	3
b. Capa Repository .....	5
c. Capa Service.....	9
d. Capa Business .....	9
e. Capa Data Access .....	11
3. Tecnología / Herramientas utilizadas .....	15
4. Modulo a Desarrollar .....	17
5. EndPoint .....	17

## ¿Qué tecnologías utilizamos?

Utilizamos Visual Studio con el motor .Net Core (en c#) en un proyecto ASP.NET API y Base de Datos SQL Express.

### Por qué elegimos C#?

Nos parece una tecnología robusta y la mayoría del grupo estaba familiarizado con ella. Es una de las tecnologías más demandadas: es la cuarta tecnología más solicitada actualmente por las empresas, por detrás de Java, JavaScript y SQL. Disminuye el tiempo de desarrollo de los proyectos: al estar basado en el paradigma de Orientación a Objetos, dispone de numerosas funcionalidades ya prediseñadas que permiten adaptar el proyecto en lugar de desarrollarlo desde cero. Cuenta con una amplia comunidad activa debido a que es de código abierto y es de lo más utilizado.

## ¿Qué arquitectura utilizamos?

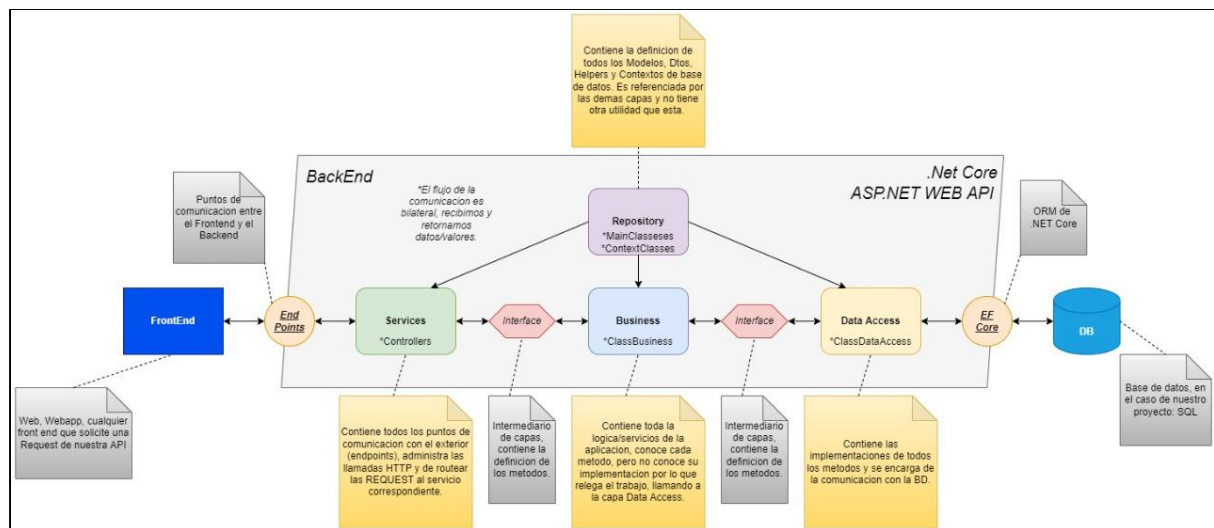
Nuestra Arquitectura es la siguiente

### Programación por CAPAS:

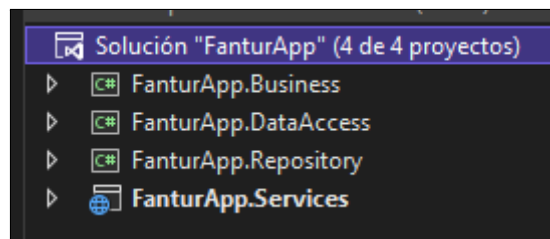
El objetivo de la **programación por capas** es el desacoplamiento de las partes que componen una arquitectura **cliente-servidor**: lógica de negocios, capa de presentación y capa de datos. De esta forma, es fácil y mantenible crear diferentes interfaces sobre un mismo sistema sin tener que cambiar nada en la capa de datos o lógica de negocios.

El motivo principal por el que usamos este tipo de arquitectura es que en caso de que tengamos algún problema y debamos realizar algún cambio, solo afectará al nivel requerido sin tener que revisar entre el código fuente de otros módulos.

También queremos recalcar que en este tipo de arquitectura, a cada nivel se le confía una misión simple, lo que permite el diseño de arquitecturas escalables (que pueden ampliarse con facilidad en caso de que las necesidades aumenten).

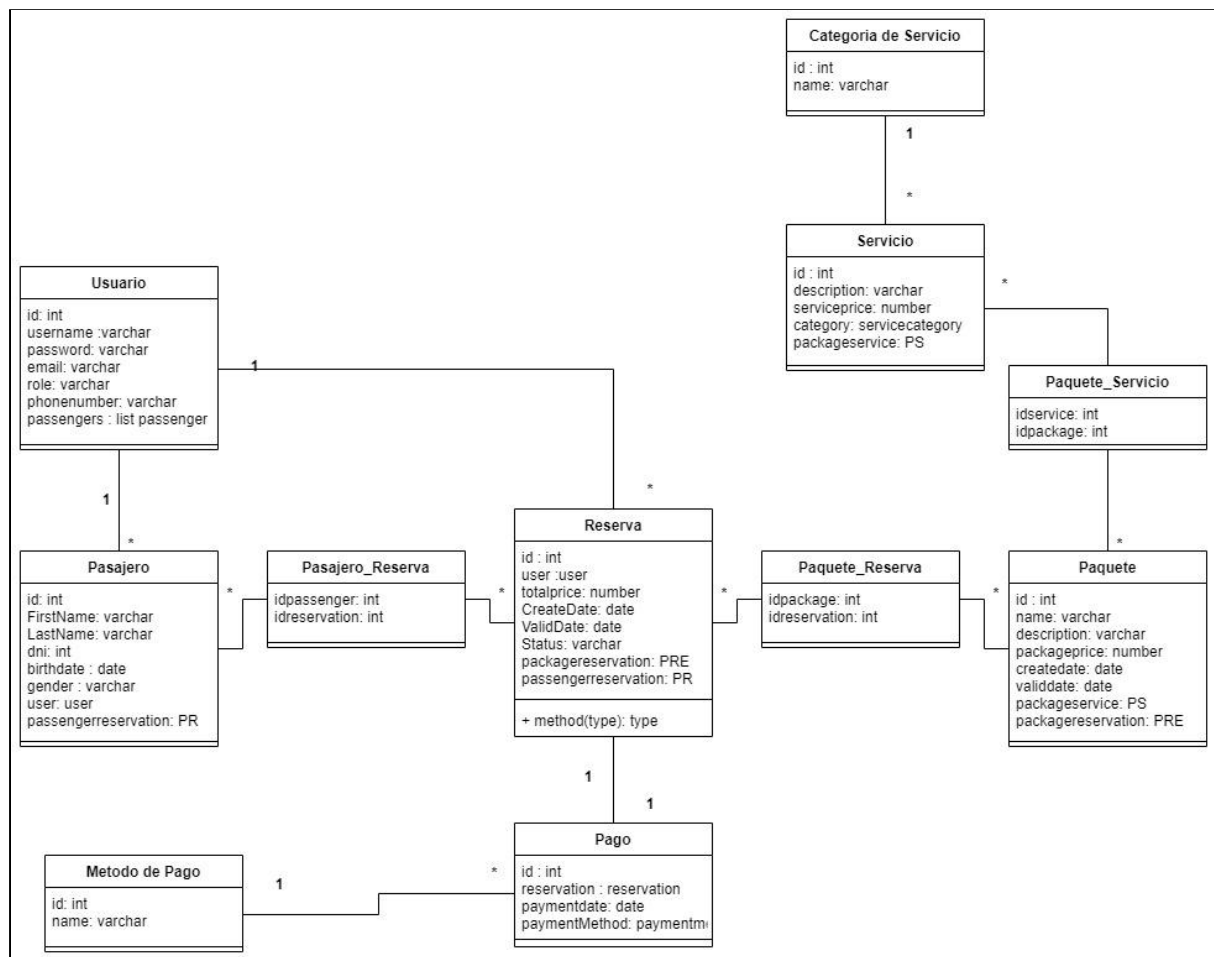


En visual basic cuando creamos capas, las definimos como proyectos, por lo que contamos con 4 proyectos



## CAPA REPOSITORY

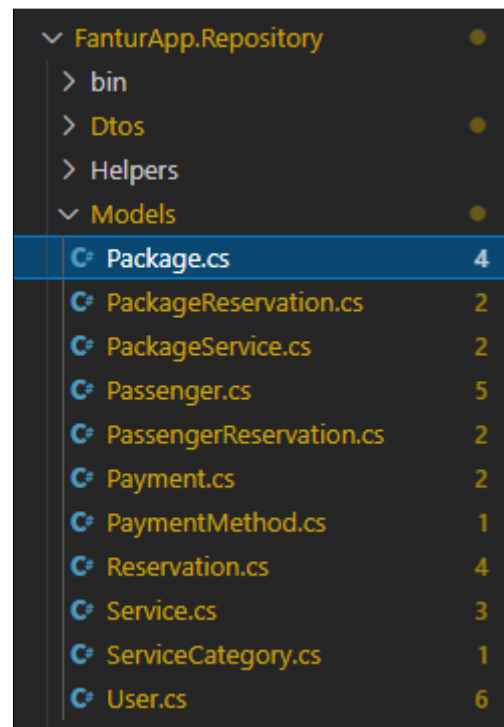
La siguiente imagen es un diagrama UML el cual lo realizamos de manera preliminar para poder tener una base sobre la cual empezar a realizar el código del trabajo. Como podemos ver tenemos diferentes clases con sus atributos y métodos.



una vez que ya tengamos diagramada la base de datos, nos basamos en el enfoque de CODE FIRST:

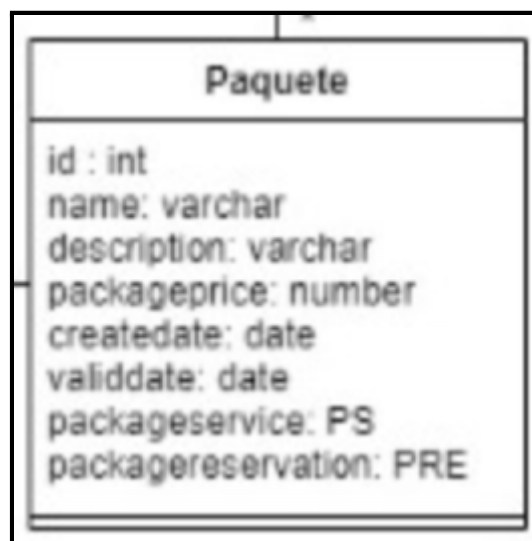
- El flujo de trabajo de modelado de Code First se dirige a una base de datos que no existe y Code First la creará.
- También se puede usar si tiene una base de datos vacía y luego Code First le agrega las tablas.
- Code First le permite definir su modelo usando clases de C# o VB.Net.
- La configuración adicional se puede realizar opcionalmente usando atributos en sus clases y propiedades o usando una API fluida.

entonces procedemos a realizar el modelado en código. Esto se modela en la capa Repositorio.



▼ FanturApp.Repository	
> bin	
> Dtos	
> Helpers	
▼ Models	
Package.cs	4
PackageReservation.cs	2
PackageService.cs	2
Passenger.cs	5
PassengerReservation.cs	2
Payment.cs	2
PaymentMethod.cs	1
Reservation.cs	4
Service.cs	3
ServiceCategory.cs	1
User.cs	6

#### Clase Package



Como podemos observar la clase Package posee los siguientes atributos:

Un Id el cual es un entero, también posee un nombre, una descripción, el precio del paquete, la fecha de creación del mismo, así como también la fecha de validación.

Además, posee un atributo llamado PackageServices el cual es una Colección que hace referencia a la tabla PackageService ya que es una relación de Muchos a muchos, es decir, un paquete puede tener muchos servicios y un servicio puede estar en distintos paquetes. Lo mismo sucede con el atributo con el atributo PackageReservations.

en código queda de la siguiente manera:

```
public class Package
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public double? PackagePrice { get; set; }
    public DateTime? CreateDate { get; set; }
    public DateTime? ValidDate { get; set; }
    public ICollection<PackageService> PackageServices { get; set; }
    public ICollection<PackageReservation> PackageReservations { get; set; }
}
```

- Clase User

La clase User posee un Identificador (Id), un nombre de usuario, contraseña, correo electrónico, también posee un rol y un número de teléfono. Además posee el atributo Passengers el cual es una colección que hace referencia a la tabla passenger.

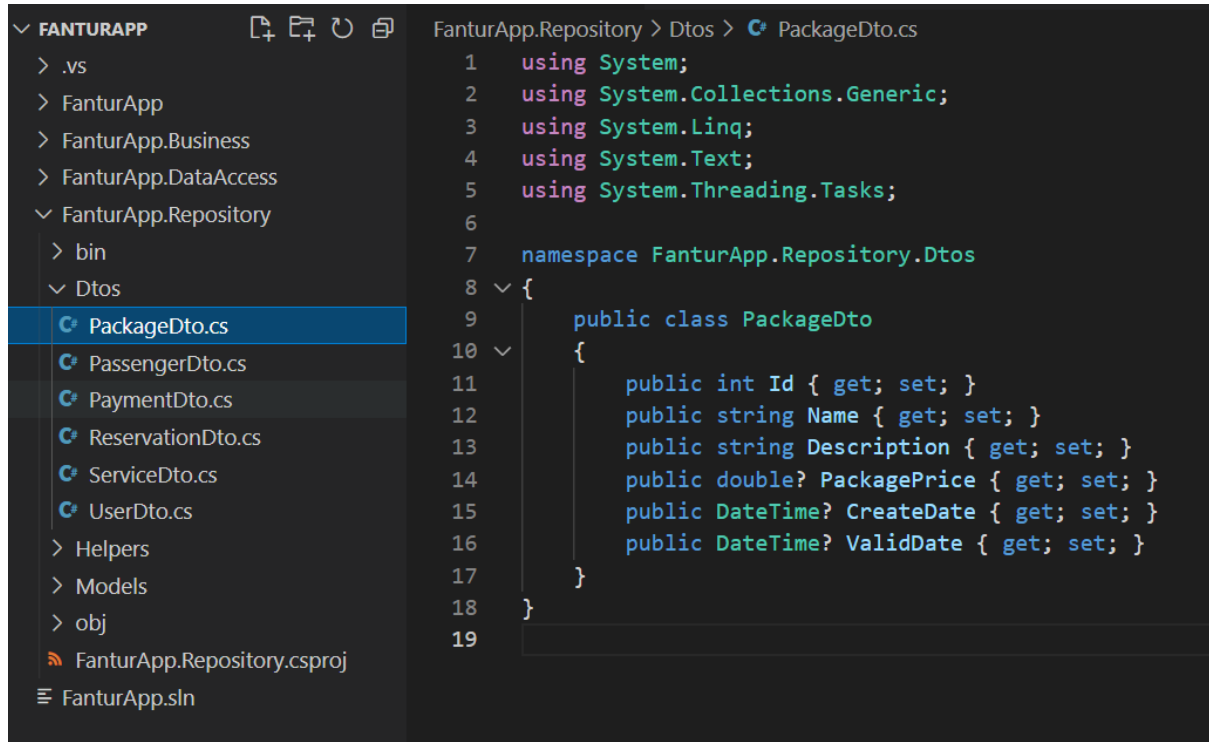
```
public class User
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public string PassWord { get; set; }
    public string Email { get; set; }
    public string Role { get; set; }
    public string PhoneNumber { get; set; }
    public ICollection<Passenger> Passengers { get; set; }
}
```

también en esta capa contamos con los DTO

### ¿Qué son los DTO?

DTO(Data Transfer Object): es básicamente un objeto que nos ayuda a mover datos entre un cliente y un servidor, pues permite crear estructuras de datos independientes de nuestro modelo de datos, podríamos decir que son una suerte de “vistas” y podemos crear cuantas sean necesarias de un conjunto de tablas u orígenes de datos. Además, nos permite controlar el formato, nombre y tipos de datos con los que transmitimos los datos para

ajustarnos a un determinado requerimiento. Finalmente, si por alguna razón, el modelo de datos cambia (y con ello las entidades) el cliente no se afectará, pues seguirá recibiendo el mismo DTO. A continuación vemos que nuestros Dtos se encuentran en la capa Repository y podemos ver un ejemplo el PackageDto.



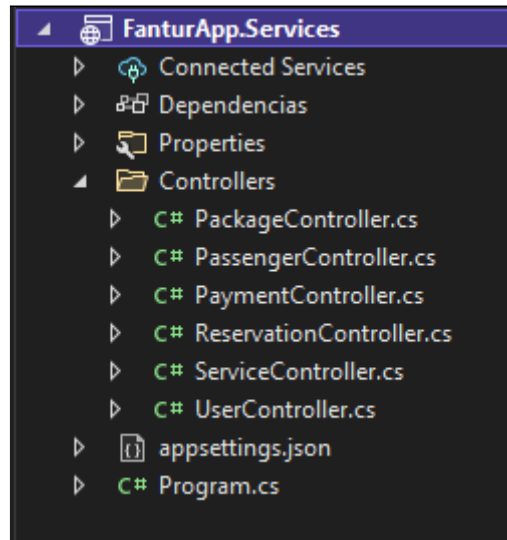
The screenshot shows the Visual Studio IDE with the FanturApp.Repository.Dtos namespace selected in the Solution Explorer. The PackageDto.cs file is open in the editor, displaying the following C# code:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace FanturApp.Repository.Dtos
8  {
9      public class PackageDto
10     {
11         public int Id { get; set; }
12         public string Name { get; set; }
13         public string Description { get; set; }
14         public double? PackagePrice { get; set; }
15         public DateTime? CreateDate { get; set; }
16         public DateTime? ValidDate { get; set; }
17     }
18 }
19
```



## Capa Service

Contiene todos los puntos de comunicación con el exterior (endpoints), administra las llamadas HTTP y de routear las REQUEST al servicio correspondiente.

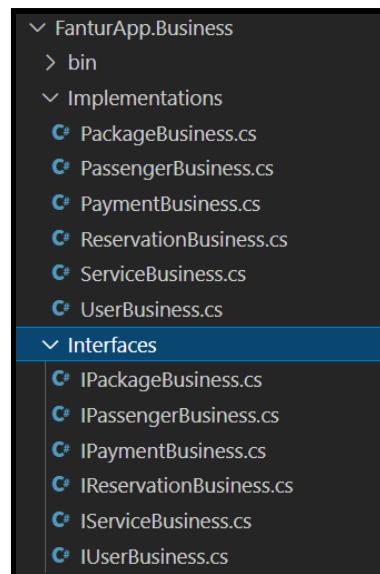


La capa service tiene controladores, tantos como entidades tengamos. Además es un “proyecto especial” porque esta capa la creamos con un template de API, esto lo que hace es:

- nos instala ya el SWAGGER, (interfaz grafica web para consumir la api, asi no usamos postman)
- tambien nos crea un archivo json llamado “appsetings.json” (contiene configuración útil, como por ejemplo, nos conecta con la base de datos)
- nos crea program.cs (es por donde inicia el programa)

## Capa Business

Esta capa contiene toda la lógica de dominio de nuestra aplicación, es decir contiene cada método pero no conoce la implementación de los mismos por lo que delega ese trabajo a la capa DataAccess. En esta capa procesamos toda la información que no requiera acceso a datos. Por ejemplo si necesitamos establecer promociones por un “Hot Sale” la lógica para definirlo se desarrollaría en esta capa.



En esta capa definimos las entidades con los cuales vamos a trabajar. Esto lo dividimos en dos partes:

Interfaces: Es una primera vista de los métodos utilizados en esta capa, pero sin su correspondiente implementación.

Implementacion: Cuenta con el desarrollo de los métodos definidos en la sección Interfaces.

### **Librerías utilizadas en esta capa:**

Linq: Utilizada para el desarrollo de las queries.

Collection.Generic: Lo utilizamos para Listas genéricas.

Text: Nos permite utilizar caracteres ASCII y UNICODE.

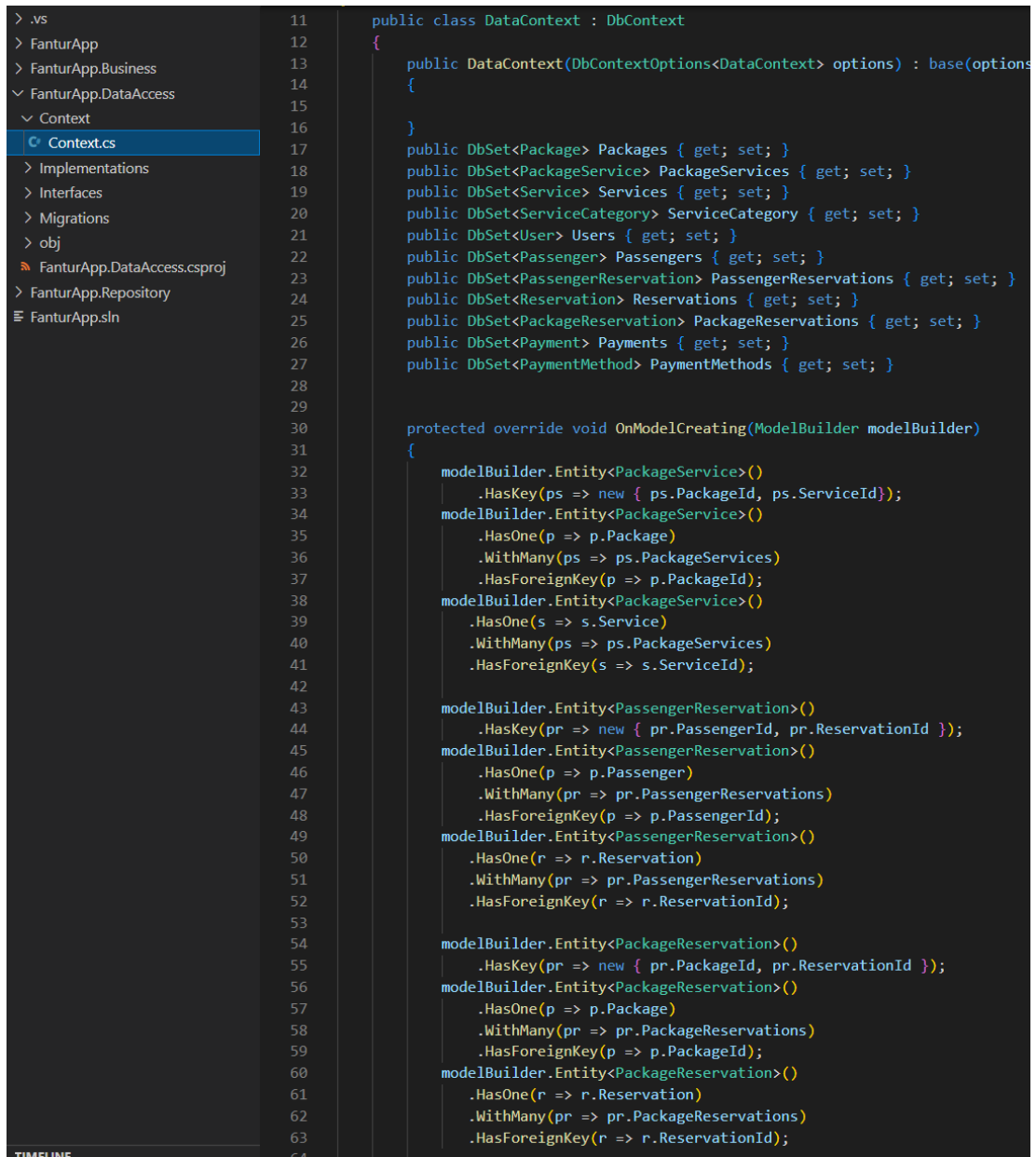
Threading.Tasks: Nos permite utilizar poder hacer operaciones asincrónicas.

## Capa Data Access

En esta capa se encuentran las implementación de cada uno de los métodos que se encuentran en la capa Business. Esta capa es la encargada de comunicarse con la Base de Datos.

En esta capa tenemos 3 carpetas:

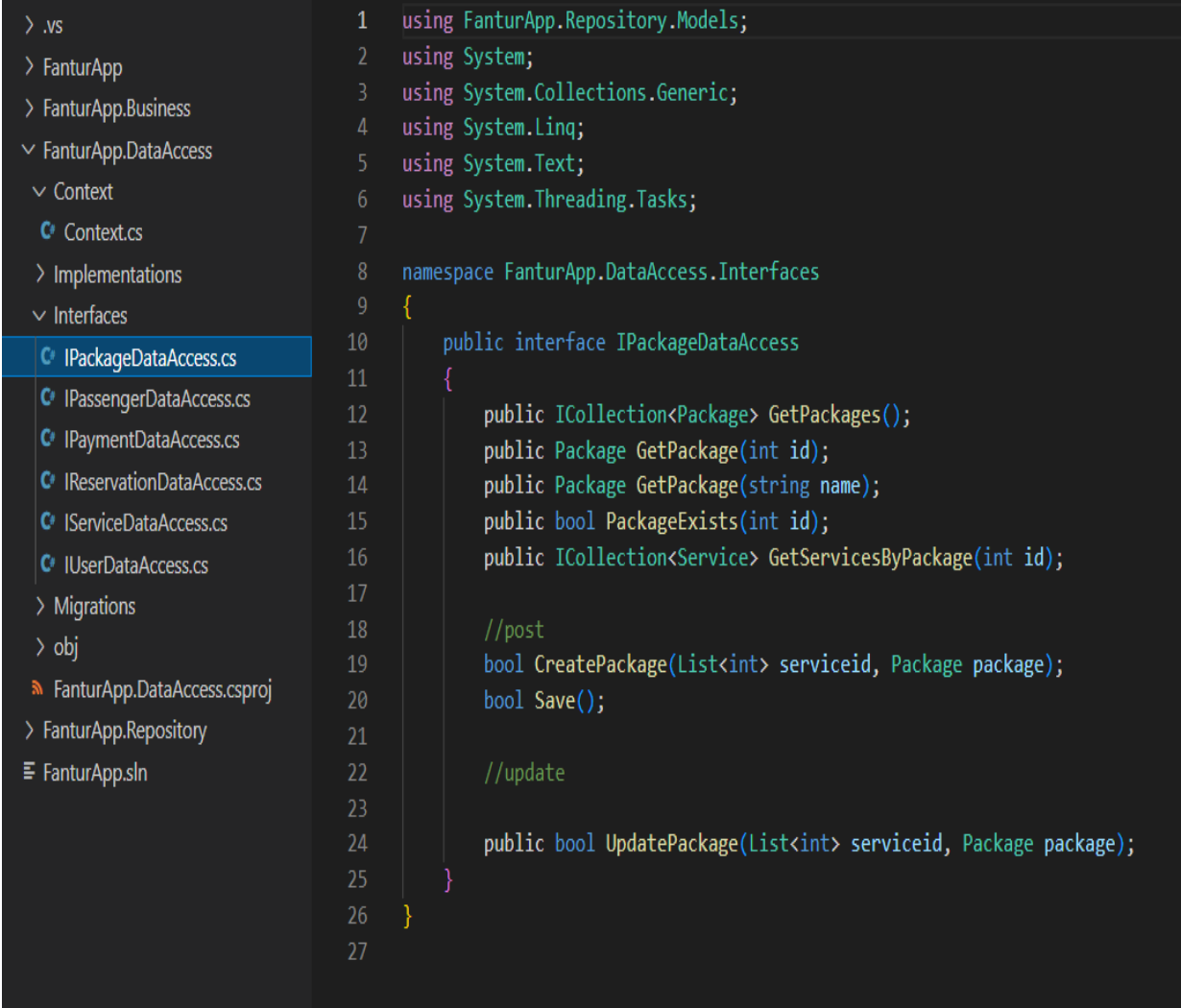
1. Context: aquí es donde relacionamos las clases del programa con la base de datos, es decir, decimos que una clase va a ser una tabla que se llama de la manera definida.



```
11 public class DataContext : DbContext
12 {
13     public DataContext(DbContextOptions<DataContext> options) : base(options)
14     {
15     }
16     public DbSet<Package> Packages { get; set; }
17     public DbSet<PackageService> PackageServices { get; set; }
18     public DbSet<Service> Services { get; set; }
19     public DbSet<ServiceCategory> ServiceCategory { get; set; }
20     public DbSet<User> Users { get; set; }
21     public DbSet<Passenger> Passengers { get; set; }
22     public DbSet<PassengerReservation> PassengerReservations { get; set; }
23     public DbSet<Reservation> Reservations { get; set; }
24     public DbSet<PackageReservation> PackageReservations { get; set; }
25     public DbSet<Payment> Payments { get; set; }
26     public DbSet<PaymentMethod> PaymentMethods { get; set; }
27
28
29
30     protected override void OnModelCreating(ModelBuilder modelBuilder)
31     {
32         modelBuilder.Entity<PackageService>()
33             .HasKey(ps => new { ps.PackageId, ps.ServiceId });
34         modelBuilder.Entity<PackageService>()
35             .HasOne(p => p.Package)
36             .WithMany(ps => ps.PackageServices)
37             .HasForeignKey(p => p.PackageId);
38         modelBuilder.Entity<PackageService>()
39             .HasOne(s => s.Service)
40             .WithMany(ps => ps.PackageServices)
41             .HasForeignKey(s => s.ServiceId);
42
43         modelBuilder.Entity<PassengerReservation>()
44             .HasKey(pr => new { pr.PassengerId, pr.ReservationId });
45         modelBuilder.Entity<PassengerReservation>()
46             .HasOne(p => p.Passenger)
47             .WithMany(pr => pr.PassengerReservations)
48             .HasForeignKey(p => p.PassengerId);
49         modelBuilder.Entity<PassengerReservation>()
50             .HasOne(r => r.Reservation)
51             .WithMany(pr => pr.PassengerReservations)
52             .HasForeignKey(r => r.ReservationId);
53
54         modelBuilder.Entity<PackageReservation>()
55             .HasKey(pr => new { pr.PackageId, pr.ReservationId });
56         modelBuilder.Entity<PackageReservation>()
57             .HasOne(p => p.Package)
58             .WithMany(pr => pr.PackageReservations)
59             .HasForeignKey(p => p.PackageId);
60         modelBuilder.Entity<PackageReservation>()
61             .HasOne(r => r.Reservation)
62             .WithMany(pr => pr.PackageReservations)
63             .HasForeignKey(r => r.ReservationId);
64     }
65 }
```

2. Interfaces: En esta carpeta encontramos las diferentes interfaces de cada una de las entidades como por ejemplo: Paquetes, pasajeros, usuario, etc. En cada una de las interfaces encontraremos definidos los metodos con los cuales esta capa se va a comunicar con la Base de Datos pero sin su implementacion.

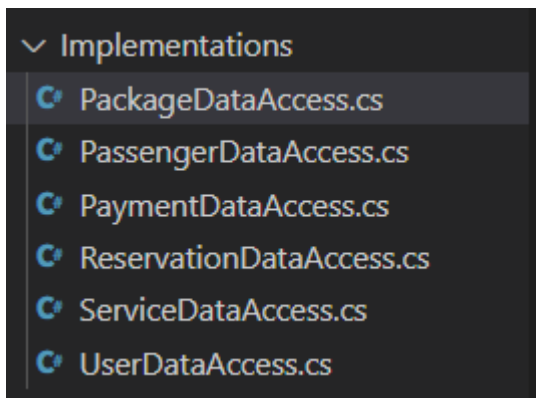
En la siguiente imagen vemos el ejemplo de la interfaz de Package.



The image shows a Visual Studio interface with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like .vs, FanturApp, FanturApp.Business, FanturApp.DataAccess, Context, Implementations, and Interfaces. The file IPackageDataAccess.cs is selected under the Interfaces folder. The code editor shows the following C# code:

```
1 using FanturApp.Repository.Models;
2 using System;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Text;
6 using System.Threading.Tasks;
7
8 namespace FanturApp.DataAccess.Interfaces
9 {
10     public interface IPackageDataAccess
11     {
12         public ICollection<Package> GetPackages();
13         public Package GetPackage(int id);
14         public Package GetPackage(string name);
15         public bool PackageExists(int id);
16         public ICollection<Service> GetServicesByPackage(int id);
17
18         //post
19         bool CreatePackage(List<int> serviceid, Package package);
20         bool Save();
21
22         //update
23
24         public bool UpdatePackage(List<int> serviceid, Package package);
25     }
26 }
27
```

3. Implementations: En esta carpeta vamos a encontrar la implementación de los métodos de cada una de las interfaces de la carpeta mencionada anteriormente, es decir, vemos como se va a comunicar cada método con la base de datos.



En la siguiente imagen vemos la implementacion de la interfaz Package y sus metodos, por ejemplo: GetPackage, GetPackages, PackageExists, etc.

```
public class PackageDataAccess : IPackageDataAccess
{
    private readonly DataContext _context;
    public PackageDataAccess(DataContext context)
    {
        _context = context;
    }

    public Package GetPackage(int id)
    {
        return _context.Packages.SingleOrDefault(c => c.Id == id);
    }

    public ICollection<Package> GetPackages()
    {
        return _context.Packages.OrderBy(p => p.Id).ToList();
    }

    public Package GetPackage(string name)
    {
        return _context.Packages.SingleOrDefault(c => c.Name == name);
    }

    public bool PackageExists(int id)
    {
        return _context.Packages.Any(c => c.Id == id);
    }
}
```

Como podemos observar, a pesar de que nuestra base de datos es SQL, la sintaxis que utilizamos en cada método puede resultar extraña. Esto se debe a que nosotros utilizamos **LINQ**

## ¿Qué es LINQ?

Language Integrated Query o Consulta Integrada en el Lenguaje es un componente de la plataforma Microsoft .NET que agrega capacidades de consulta a datos de manera nativa

LINQ extiende el lenguaje a través de las llamadas expresiones de consulta, que son parecidas a las sentencias SQL y pueden ser usadas para extraer y procesar convenientemente datos

En la siguientes imagenes se puede observar mas de cerca como utilizamos la sintaxis de LINQ

En esta primera imagen vemos que traemos de la base de datos las reservaciones por usuario y las devolvemos en forma de lista

```
public ICollection<Reservation> GetReservationsByUser(int id)
{
    return _context.Reservations.Where(r => r.User.Id == id).ToList();
}
```

En esta segunda ocacion, traemos el servicio que coincida con el id que le pasamos a traves del metodo incluyendo la categoria.

SingleOrDefault: esto nos trae un solo resultado o el resultado por defecto.

```
public Service GetService(int id)
{
    return _context.Services.Include(s => s.Category).SingleOrDefault(c => c.Id == id);
}
```

## tecnologías/herramientas utilizadas en el proyecto

### **¿Qué es una inyección de dependencias ?**

Una inyección de dependencia es: dicho de mala manera, una forma de tercerizar el trabajo dentro de nuestro código. Viene de la mano con el principio de Single Responsibility de SOLID.

Cada una de nuestras capas y clases tiene un solo trabajo y responsabilidad, si en algún momento requerimos que por ejemplo, un objeto de business acceda a la base de datos esto claramente implicaría incumplir su responsabilidad, lo que podemos hacer es realizar la inyección de dependencia, la cual consiste en instanciar un objeto de la capa que sí tenga esa responsabilidad definida y dejar que esta haga el trabajo sucio, para luego devolver el control al objeto principal.

### **¿Cuáles son los elementos en nuestra carpeta helper del proyecto?**

Los Helpers son clases especializadas que nos ayudan en diversos procesos del funcionamiento de la app, en nuestro caso tenemos definida una clase mapping la cual es necesaria por el paquete AutoMapper, este mapping establece una relación entre los modelos con los Dtos (data transfer objects) y viceversa. El automapper nos ayuda a “convertir” esos modelos en un Dto mapeando los campos de cada uno de manera automática.

También definimos una clase Seed, la cual será utilizada para poblar la BD, pero en esta primera entrega aunque WIP.

### **¿Qué son las interfaces? ¿Para qué las usamos?**

Las interfaces son un medio común para que los objetos no relacionados se comuniquen entre sí, en las interfaces se definen todos los métodos pero no se implementan.

Las usamos justamente para que actúen de intermediarios entre objeto y objeto. En lugar de permitir que un objeto se comunique directamente con otro, lo que hacemos es definir interfaces para cada clase y que los objetos heredan estas, de esta forma toda comunicación debe pasar primero por la interfaz y esto a su vez es de gran ayuda ya que disminuye el acoplamiento.

## ¿Qué es el ORM?

Un *ORM (Object-Relational Mapper)* es un modelo de programación que permite mapear las estructuras de una base de datos relacional (*SQL Server, Oracle, MySQL, etc.*), sobre una estructura lógica de entidades con el objeto de simplificar y acelerar el desarrollo de nuestras aplicaciones.

Con la ayuda de Entity Framework Core, podemos completar nuestros métodos realizando consultas a nuestra base SQL utilizando cómodos métodos predefinidos (ej. `FirstOrDefault(id)` buscar el primer registro con id coincidente o por defecto) que nos facilitan la codificación y legibilidad del código. También podemos acceder a nuestra DB realizando queries LINQ, cuyo lenguaje es muy similar al SQL convencional, solo que enfocado a utilizarse desde .Net.

## ¿Qué es el Entity Framework core?

*Entity Framework Core* es una tecnología de acceso a datos para .NET Core y .NET Framework. Es multiplataforma y de código abierto desarrollado por Microsoft con aportes de la comunidad. Propiamente dicho es un asignador objeto relacional o ORM por sus siglas en inglés. Su función principal es servir como interprete entre dos tecnologías fundamentadas en distintos principios por un lado la programación orientada a objetos y por el otro las bases de datos relacionales y no relacionales.

Permite al programador controlar una base de datos relacional usando un lenguaje de programación en lugar de SQL estándar o uno de sus dialectos. Libera al programador de escribir gran cantidad de código repetitivo para acceder a los datos.



# Módulo a desarrollar

Se realizó el desarrollo tanto del módulo de clientes, como del módulo de administradores. Esto significa que nuestro api contiene servicios tanto para la reserva/compra de los paquetes de mano de los clientes, como para la creación y administración de estos paquetes por los administradores.

## Endpoints

En esta primera entrega, nuestra api presenta los siguientes EndPoints.

Payment		^
GET	/api/Payment SIN PARAMETROS, BUSCA Y DEVUELVE UNA LISTA DE TODOS LOS PAGOS	▼
POST	/api/Payment RESERVA ID - METODO PAGO ID, CREA UNA NUEVA RESERVA CON LOS DATOS DEL BODY Y LOS PARAMETROS DADOS	▼
GET	/api/Payment/{paymentId} ID PAGO, BUSCA Y MUESTRA EL PAGO	▼
PUT	/api/Payment/{paymentId} ID PAGO, BUSCA Y MODIFICA EL PAGO CON LOS VALORES DEL BODY	▼
DELETE	/api/Payment/{paymentId} ID PAGO, BUSCA Y BORRA EL PAGO	▼
GET	/api/Payment/{paymentId}/reservation ID PAGO, MUESTRA LA RESERVA VINCULADA AL PAGO	▼
Passenger		^
GET	/api/Passenger SIN PARAMETROS, BUSCA Y RETORNA TODOS LOS PASAJEROS COMO LISTA	▼
POST	/api/Passenger ID USUARIO, CREA UN NUEVO PASAJERO CON LOS DATOS DEL BODY Y LO VINCULA AL ID USUARIO DADO	▼
GET	/api/Passenger/{passengerId} ID PASAJERO, BUSCA AL PASAJERO Y LO MUESTRA	▼
PUT	/api/Passenger/{passengerId} ID PASAJERO, BUSCA AL PASAJERO Y LO MODIFICA CON LOS DATOS DEL BODY	▼
DELETE	/api/Passenger/{passengerId} ID PASAJERO, BUSCA AL PASAJERO Y LO BORRA	▼
GET	/api/Passenger/{passengerId}/user ID USUARIO, BUSCA TODOS LOS PASAJERO VINCULADOS CON EL ID USUARIO DADO	▼
Reservation		^
GET	/api/Reservation SIN PARAMETROS, BUSCA Y LISTA TODOS LAS RESERVACIONES	▼
POST	/api/Reservation USUARIO ID - LISTA PAQUETE ID - LISTA PASAJERO ID, CREA UNA NUEVA RESERVA CON LOS DATOS DEL BODY Y LOS PARAMETROS DADOS	▼
GET	/api/Reservation/{reservationId} RESERVA ID, BUSCA LA RESERVA Y LA MUESTRA	▼
PUT	/api/Reservation/{reservationId} RESERVA ID - PASAJERO ID - PAQUETE ID, BUSCA Y MODIFICA LA RESERVA CON LOS DATOS DEL BODY Y LOS PARAMETROS DADOS	▼
GET	/api/Reservation/By/{status} STATUS RESERVA, BUSCA Y TRAE LOS PAQUETES CUYO ESTADO SEA EL SOLICITADO	▼
GET	/api/Reservation/{reservationId}/user ID RESERVA, BUSCA LOS USUARIOS VINCULADOS A LA RESERVA Y LOS MUESTRA	▼
GET	/api/Reservation/{reservationId}/packages ID RESERVA, BUSCA LOS PAQUETES VINCULADOS A LA RESERVA Y LOS MUESTRA	▼
GET	/api/Reservation/{reservationId}/passengers ID RESERVA, MUESTRA LOS PASAJEROS VINCULADOS A LA RESERVA Y LOS MUESTRA	▼

Service			^
GET	/api/Service	SIN PARAMETROS, BUSCA Y TRAE UNA LISTA DE TODOS LOS SERVICIOS	✓
POST	/api/Service	CATEGORIA ID, CREA UN SERVICION CON LOS DATOS DEL BODY Y EL PARAMETRO DADO	✓
GET	/api/Service/{serviceId}	SERVICIO ID, BUSCA EL SERVICIO Y LO MUESTRA	✓
PUT	/api/Service/{serviceId}	SERVICIO ID, BUSCA EL SERVICIO Y LO MODIFICA CON LOS DATOS DEL BODY	✓
DELETE	/api/Service/{serviceId}	SERVICIO ID, BUSCA Y BORRA EL SERVICIO	✓
GET	/api/Service/{serviceId}/packages	SERVICIO ID, BUSCA Y MUESTRA LOS PAQUETES VINCULADOS AL SERVICIO	✓
Package			^
GET	/api/Package	SIN PARAMETROS, BUSCA Y RETORNA UNA LISTA DE TODOS LOS PACKAGES	✓
POST	/api/Package	ID DE SERVICIO COMO PARAMETRO, TOMA LOS VALORES PARA LA CREACION DEL OBJETO DESDE EL BODY, RETORNA OK	✓
GET	/api/Package/{packageId}	ID PAQUETE, BUSCA EL PAQUETE QUE COINCIDA CON EL ID DADO Y LO MUESTRA	✓
PUT	/api/Package/{packageId}	ID PAQUETE, BUSCA EL PAQUETE QUE COINCIDA CON EL ID DADO Y LO ACTUALIZA CON LOS DATOS DEL BODY, RETORNA OK	✓
GET	/api/Package/{packageId}/services	ID PAQUETE, BUSCA EL PAQUETE QUE COINCIDA CON EL ID DADO Y MUESTRA LOS SERVICIOS VINCULADOS A ESTE	✓
User			^
GET	/api/User	SIN PARAMETROS, BUSCA Y LISTA TODOS LOS USUARIOS	✓
POST	/api/User	SIN PARAMETROS, CREA UN NUEVO USUARIO CON LOS DATOS DE BODY	✓
GET	/api/User/{userId}	ID USUARIO, BUSCA AL USUARIO DEL ID Y LO MUESTRA	✓
PUT	/api/User/{userId}	ID USUARIO, BUSCA LA USUARIO Y LO MODIFICA CON LOS DATOS DEL BODY	✓
DELETE	/api/User/{userId}	ID USUARIO, BUSCA AL USUARIO Y LO BORRA	✓
GET	/api/User/{userId}/passengers	ID USUARIO, BUSCA Y MUESTRA LOS PASAJEROS VINCULADOS AL USUARIO ID	✓
GET	/api/User/{userId}/reservations	ID USUARIO, BUSCA Y MUESTRA LAS RESERVAS VINCULADAS AL USUARIO ID	✓