

Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 2

Diseño de aplicaciones 1

Fecha: 10/6/2024

Nicolás Russo 227286
Victoria Chappuis 278311

Docentes: Gastón Mosqués, Martín Radovitzky, Facundo
Arancet

Grupo: M5A

[Repositorio](#)

1- Nuevas funcionalidades implementadas	3
2 - Justificación de Diseño	4
A - Diagrama de paquetes	4
B - Diagramas de Clases	5
Paquete DepoQuick.DataAccess.Repos	5
Paquete DepoQuick.Backend.Models	6
Paquete DepoQuick.Backend.Services	8
Paquete DepoQuick.Backend.Dtos	9
C - Diagramas de Interacción	9
1. Exportación de reservas en CSV.	9
2. Registro de un usuario (no admin)	10
3. Reserva exitosa de un warehouse	10
D - Modelo de Tablas de la base de datos	11
E - Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas	12
Actualización de la primera entrega	12
Segunda entrega	13
F - Análisis de los criterios seguidos para asignar las responsabilidades	14
4 - Mantenibilidad y Extensibilidad	15
5 - Análisis de Dependencias	15
Actualización de la primera entrega	15
Segunda entrega	16
6 - Cobertura de Pruebas Unitarias	16
Primera Entrega	16
Test Implementado	18
Código de función testeada	19
Cambios en función en el commit GREEN	19
Image	19
Cambios en función en el commit REFACTOR	19
Segunda entrega	19
Instructivo para acceder a la aplicación	19
Anexo	20

1- Nuevas funcionalidades implementadas

En esta segunda etapa del proyecto, hemos logrado un avance significativo en la persistencia de datos. Implementamos una solución de base de datos utilizando Azure SQL Edge.

Esta base de datos se ejecuta en un contenedor Docker, lo que nos proporciona una gran flexibilidad y portabilidad, ya que podemos desplegar nuestra base de datos en cualquier sistema que soporte Docker. Esto también facilita la escalabilidad y la gestión de la base de datos, ya que podemos fácilmente escalarla añadiendo más contenedores según sea necesario.

Para interactuar con la base de datos, hemos utilizado Entity Framework Core (code first). Esta es una tecnología ORM (Object-Relational Mapping) de Microsoft que nos permite trabajar con la base de datos utilizando objetos de .NET.

El enfoque “code first” fue nuestra elección ya que nuestro proyecto ya contaba con un conjunto de código preexistente. Este enfoque de Entity Framework Core nos permite generar y manejar la base de datos directamente desde nuestro código. De esta manera, no tuvimos que invertir tiempo adicional en diseñar la base de datos manualmente, ya que Entity Framework Core puede crearla automáticamente a partir de las clases y relaciones que ya habíamos definido en nuestro código.

Con esta nueva funcionalidad, nuestro proyecto ahora tiene una solución de persistencia de datos robusta y escalable que puede manejar grandes volúmenes de datos y proporcionar análisis en tiempo real.

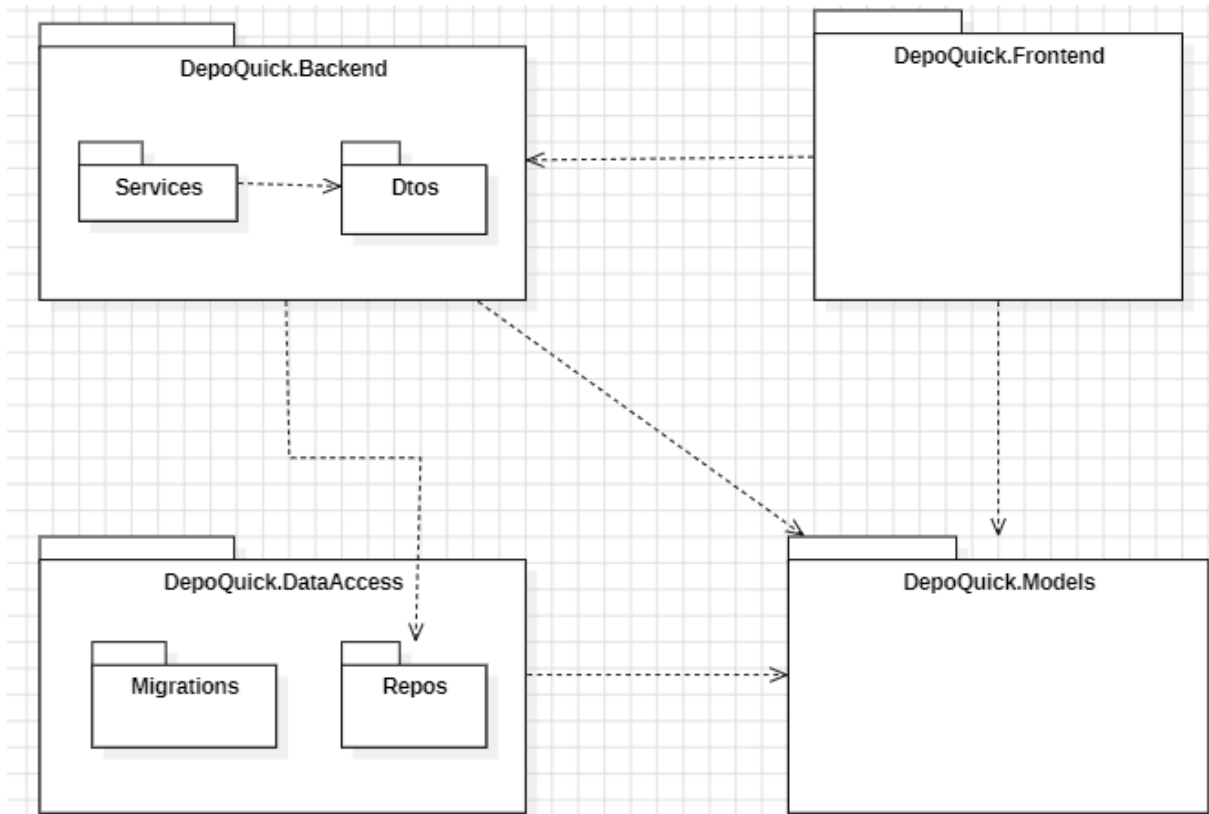
Por otro lado, también implementamos nuevas features como la gestión de disponibilidad de reservas, donde no pueden haber dos reservas de un mismo depósito solapadas. Además al reservar solamente se muestran los depósitos con disponibilidad en las fechas seleccionadas. También incorporamos la gestión de pagos. Ahora, al realizar una reserva, los usuarios pueden efectuar el pago correspondiente, que inicialmente se marca como “reservado”. Una vez que la reserva es confirmada por el administrador, el estado del pago se actualiza a “capturado”. Tanto los administradores como los clientes pueden visualizar el estado del pago para cada reserva. Todo esto se maneja de manera sencilla a través de un botón “PAGAR”.

Por último, ahora el administrador puede descargar un reporte de reservas, tanto en CSV como en TSV. Esto a través de tocar tan solo un botón que lo indica.

Todas las demás funcionalidades implementadas en la etapa anterior, continúan vigentes.

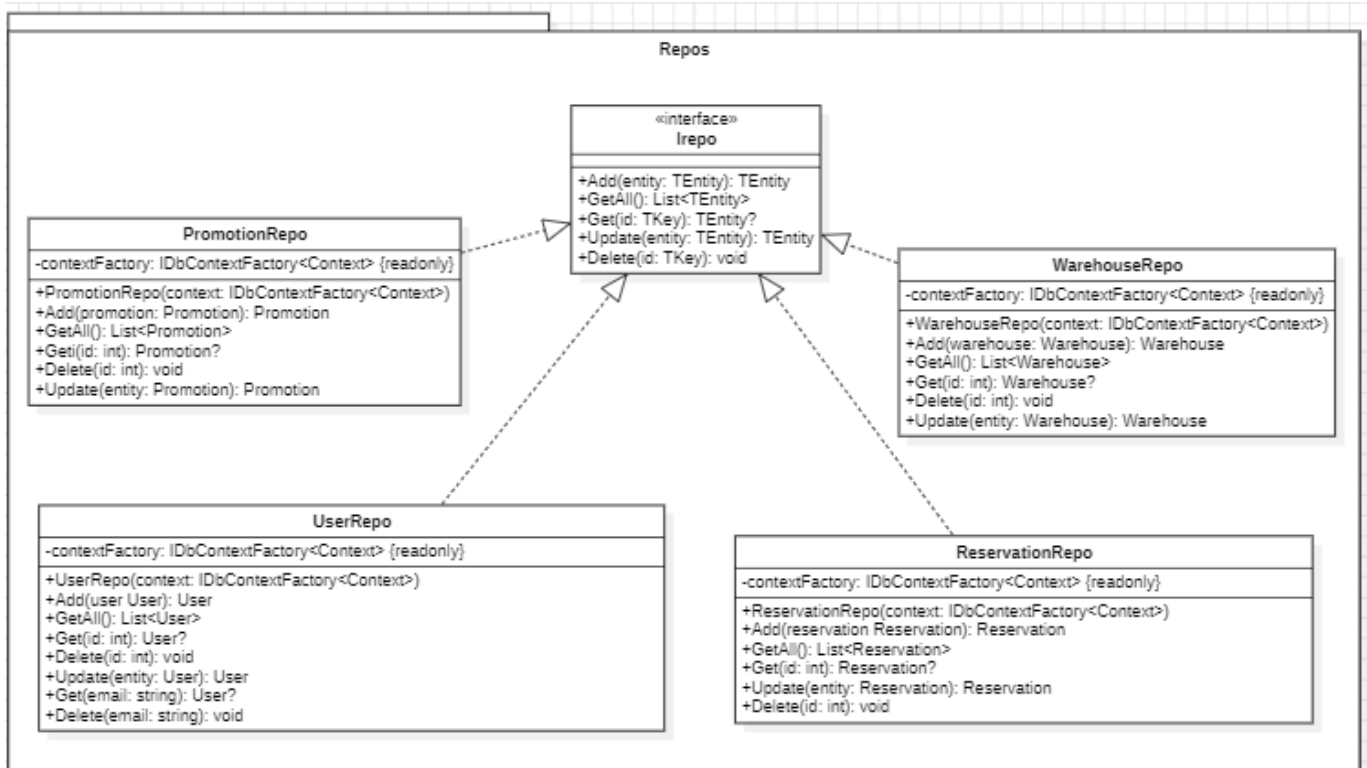
2 - Justificación de Diseño

A - Diagrama de paquetes



B - Diagramas de Clases

Paquete DepoQuick.DataAccess.Repos



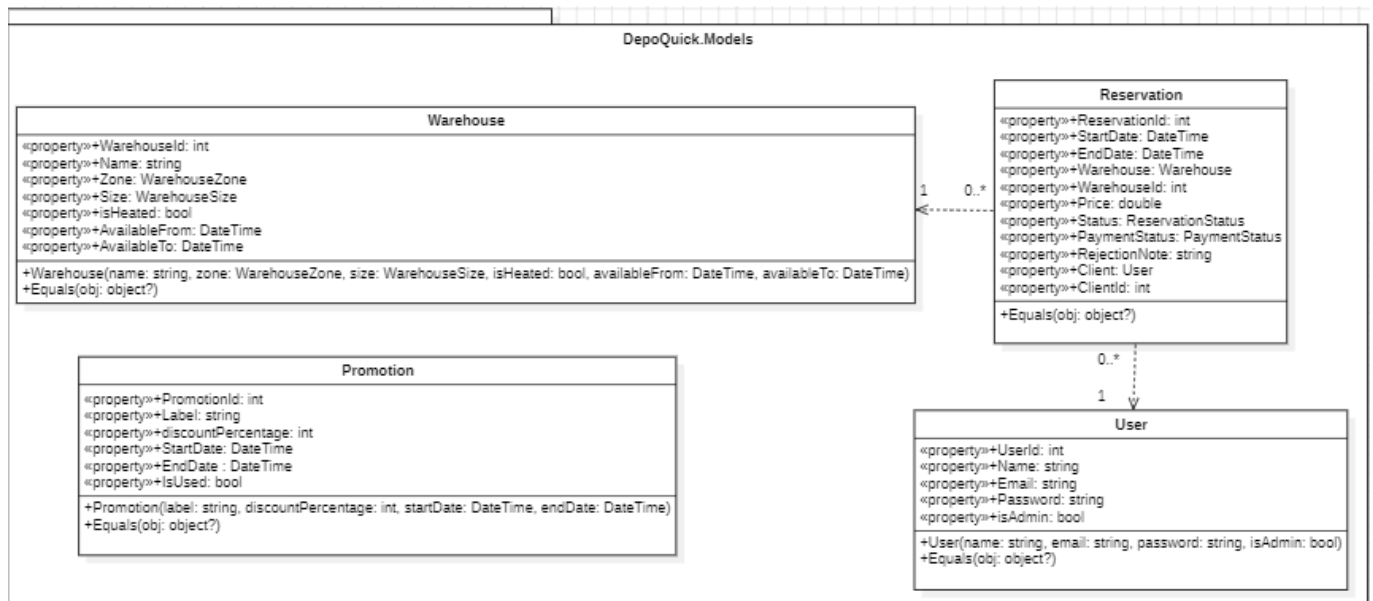
Clases: PromotionRepo, ReservationRepo, UserRepo, WarehouseRepo

Interfaces: IRepository

Responsabilidad: Agrupar las clases que se encargan de gestionar y facilitar el acceso a los datos almacenados. Estas clases, conocidas como repositorios, actúan como una capa de abstracción entre la lógica de negocio y la fuente de datos subyacente, ya sea una base de datos u otro tipo de almacenamiento. Los repositorios permiten interactuar con la fuente de datos de manera más sencilla, facilitando las operaciones de consulta, inserción, actualización y eliminación de registros.

La clase `IRepository` define una interfaz genérica para la gestión básica de entidades, mientras que las clases específicas, como `WarehouseRepo` y `UserRepo`, implementan las operaciones relacionadas con la gestión de almacenes y usuarios, respectivamente.

Paquete DepoQuick.Backend.Models



Classes: Promotion, Reservation, User, Warehouse

Responsabilidad: Agrupar todas las clases relacionadas con el modelo de datos en el proyecto DepoQuick. Estas clases representan entidades y estructuras clave en el sistema.

El objetivo principal del namespace *DepoQuick.Models* es proporcionar una organización y estructura clara para representar los datos correspondientes al modelo de la aplicación.

Comentario: Sobre la posible crítica acerca del “**Dominio Raquítrico**”. Nuestra decisión de crear el dominio de esta forma desde el principio fue consciente y tomada a sabiendas de que iba a recibir esta crítica. De todas formas decidimos **apegarnos al espíritu de la materia**, que es **crear código mantenible** y buscando seguir las mejores prácticas de cada lenguaje (como fue dicho al principio del curso). Nos apoyamos entonces en la documentación provista por Microsoft, que apunta a un dominio delgado y modelos que contengan solamente la estructura de los datos y se despeguen lo más posible de la lógica de negocio (Estos son algunos ejemplos: [Ejemplo 1](#), [Ejemplo 2](#), [Aplicación de ejemplo de uso](#) se pueden encontrar muchísimos más). Para proveer la lógica de negocio, usamos otra capa de servicios, que interactúan con estos modelos y desacoplan esta lógica, como se propone en toda la documentación oficial de .NET y Entity Framework.

En esta segunda parte, a pesar de haber sustentado la postura a favor de un dominio liviano, recibimos críticas sobre el mismo. Aprovechamos la oportunidad para investigar más a fondo la temática y de nuevo encontramos que un dominio liviano con una capa de servicios que implemente la lógica de negocios es lo que más parece apegarse al estilo y mejores prácticas de EF Core. Y no afirmamos esto meramente porque sea lo que hicimos, nos apoyamos en las fuentes más duras sobre mejores prácticas con EF. Por ejemplo, en el libro *Entity Framework Core in Action (2018)* de Jon Smith (el texto más popular de Entity Framework Core), se cita que el patrón utilizado para implementar lógica compleja de negocio y el que se utiliza a través de todo el libro son los “Transactions script”:

4.3 Using a design pattern to implement complex business logic

Before you start writing code to process an order, take a look at a pattern that will help you write, test, and performance-tune your business logic. The pattern is based on the Domain-Driven Design (DDD) concepts expounded by Eric Evans, but the business logic code isn't inside the entity classes. This pattern is known as a *transactions script* or *procedural* pattern of business logic because the code is contained in a standalone method.

This procedural pattern is easy to understand and uses the basic EF Core commands you have already seen. But many people see the procedural approach as being a DDD antipattern, known as an *anemic domain model* (see <http://mng.bz/nM7g>). Later, in part 3 of this book, you will extend this approach to a fully DDD design.

This section and chapter 13 present my interpretation of Evans' DDD approach and plenty of other ways to apply DDD with EF. Although I offer my approach, which I hope will help you, don't be afraid to look for other approaches.

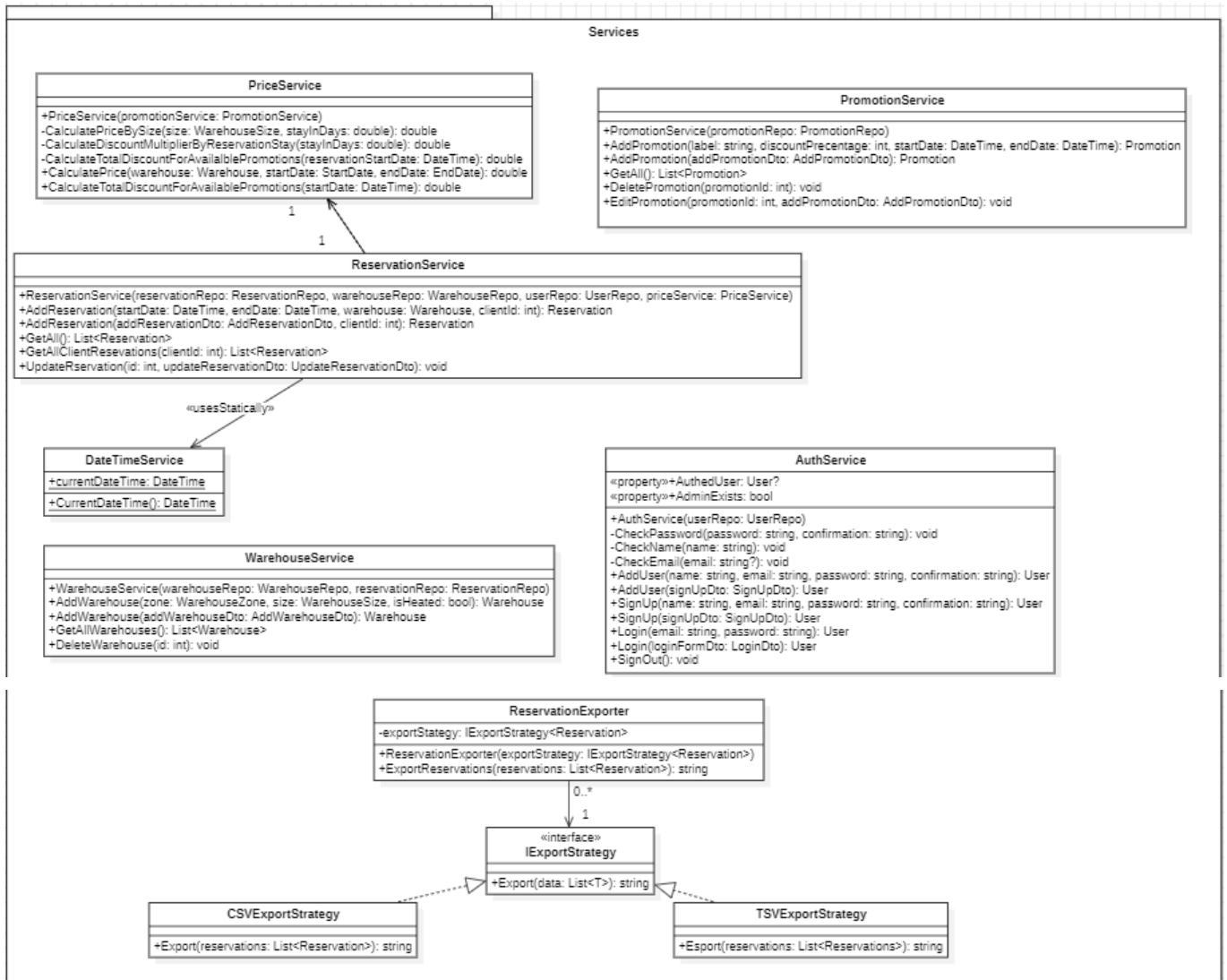
Este patrón de *Transaction Scripts* fue exactamente el elegido para implementar desde la primera parte en nuestro obligatorio ya que la lógica que requiere es simple y como reiteramos, buscamos las mejores prácticas. El patrón sale del libro *Patterns of Enterprise Application Architecture* (P of EAA) de Martin Fowler (recomendado en clase), en el que se describe como una alternativa al *Domain Model* (o Dominio Rico) con sus ventajas y desventajas.

Además Jon Smith muestra que este patrón es **una alternativa**, de otras que hay, y eso es exactamente lo que nosotros buscamos proponer. **No hay una sola alternativa**, como vimos en clase Martin Fowler en su sitio cita que prefiere el *Domain Model* (o Dominio Rico) por encima de otros, pero no es una verdad absoluta en lo más mínimo. Hasta el mismo Martin Fowler describe en P of EAA (Capítulo 9.1) que es una alternativa viable para sistemas con lógica simple, como el nuestro:

However much of an object bigot you become, don't rule out Transaction Script. There are a lot of simple problems out there, and a simple solution will get you up and running much faster.

Basándonos en todo esto, cuando estamos enfrentándonos a armar nuestro dominio, con lógica muy simple en EF Core no creemos que un dominio rico sea lo mejor. Sea cual sea la crítica que recibamos, sería deshonesto de nuestra parte hacer ojos ciegos a la mayoría de la documentación de EF Core, tanto de Microsoft como de consultores independientes que apunta a un dominio liviano con una capa de servicios que contenga la lógica de negocios.

Paquete DepoQuick.Backend.Services

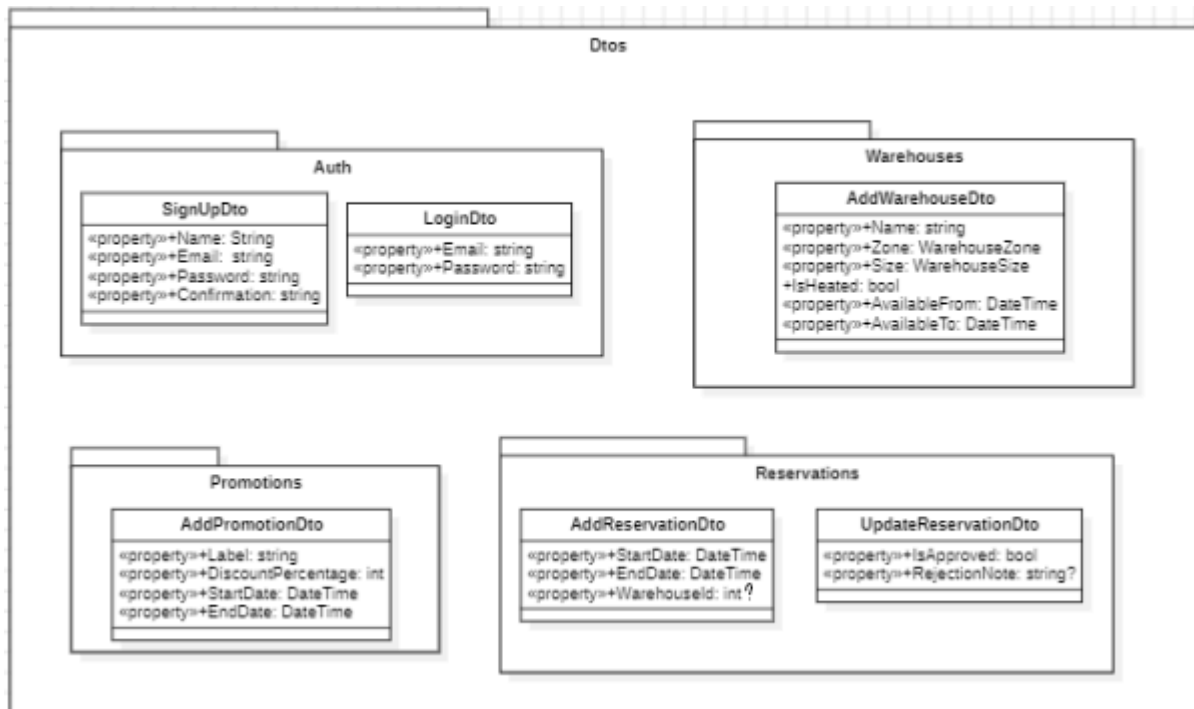


Classes: AuthService, PriceService, PromotionService, WarehouseService, ReservationService, DateTimeService

Responsabilidad: Agrupar las clases que proporcionan servicios y funcionalidades específicas dentro del proyecto DepoQuick. Los servicios son responsables de llevar a cabo operaciones sobre los modelos y lógica de negocio en el sistema.

El objetivo principal del namespace es encapsular la lógica de negocio en servicios que pueden ser utilizados por otros componentes del sistema, como las capas de presentación y repositorio. Esto ayuda a mantener un código modular, reutilizable y fácil de mantener.

Paquete DepoQuick.Backend.Dtos



Clases: Auth.LoginDto, Auth.SignUpDto, Warehouses.AddWarehouseDto, Promotions.AddPromotionDto, Reservations.AddReservationDto, Reservations.UpdateReservationDto

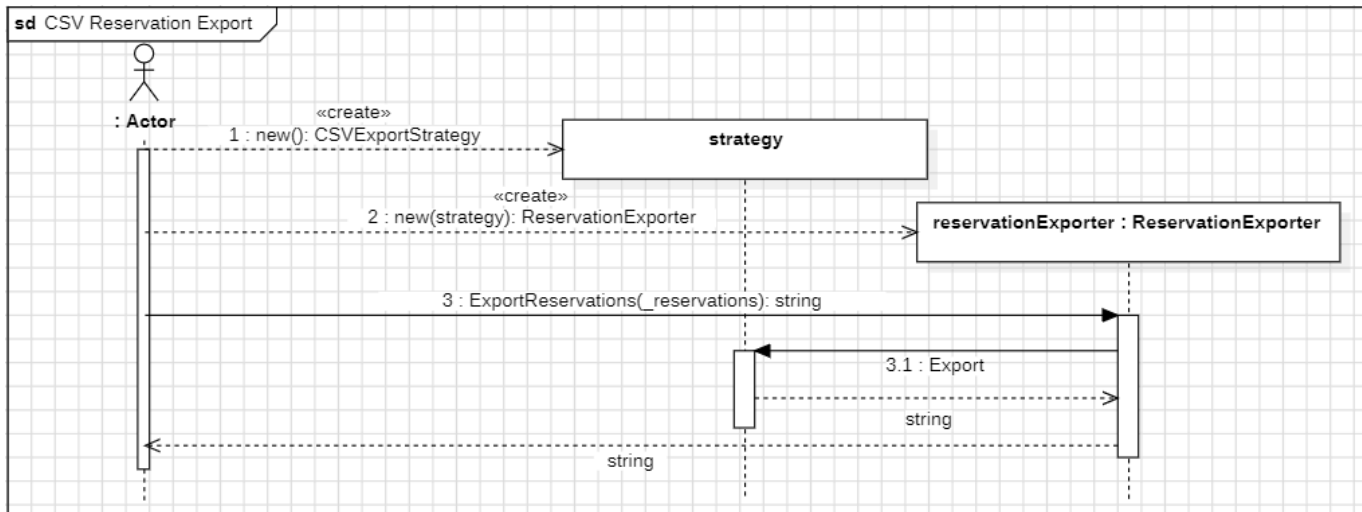
Responsabilidad: Agrupar las clases relacionadas con los modelos de datos utilizados en la transferencia de datos (desde el usuario al sistema más que nada) del proyecto DepoQuick. Estos modelos de datos, generalmente conocidos como Data Transfer Objects (DTO), se utilizan para transportar información relacionada con la autenticación y autorización entre diferentes capas o componentes del sistema.

Proporcionan una organización y estructura clara para los modelos de datos utilizados en los procesos de autenticación y autorización, facilitando su uso y mantenimiento en el código.

C - Diagramas de Interacción

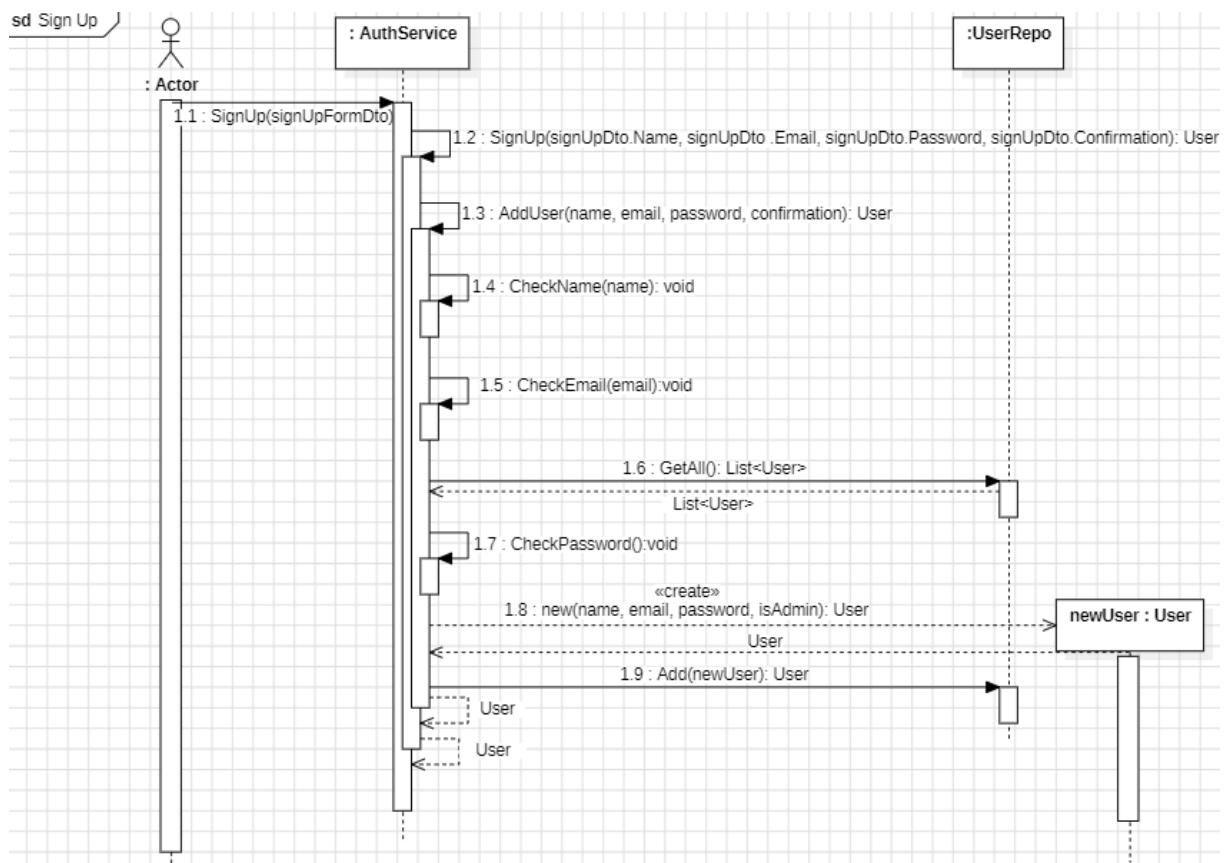
1. Exportación de reservas en CSV.

Desde el momento que presionamos el botón CSV de Reservations/index.razor

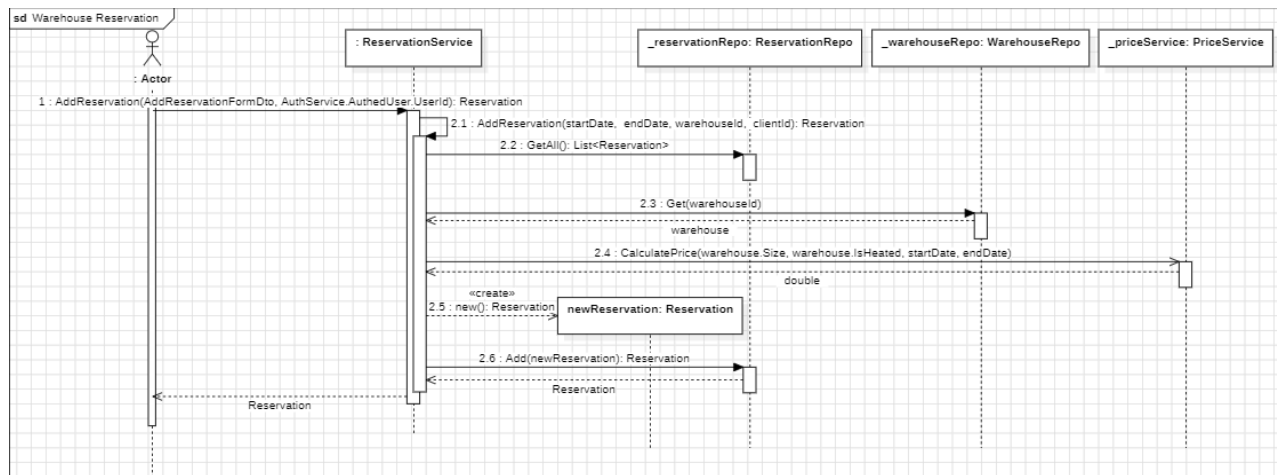


2. Registro de un usuario (no admin)

Desde el momento que se presiona el botón de submit en SignUp.razor

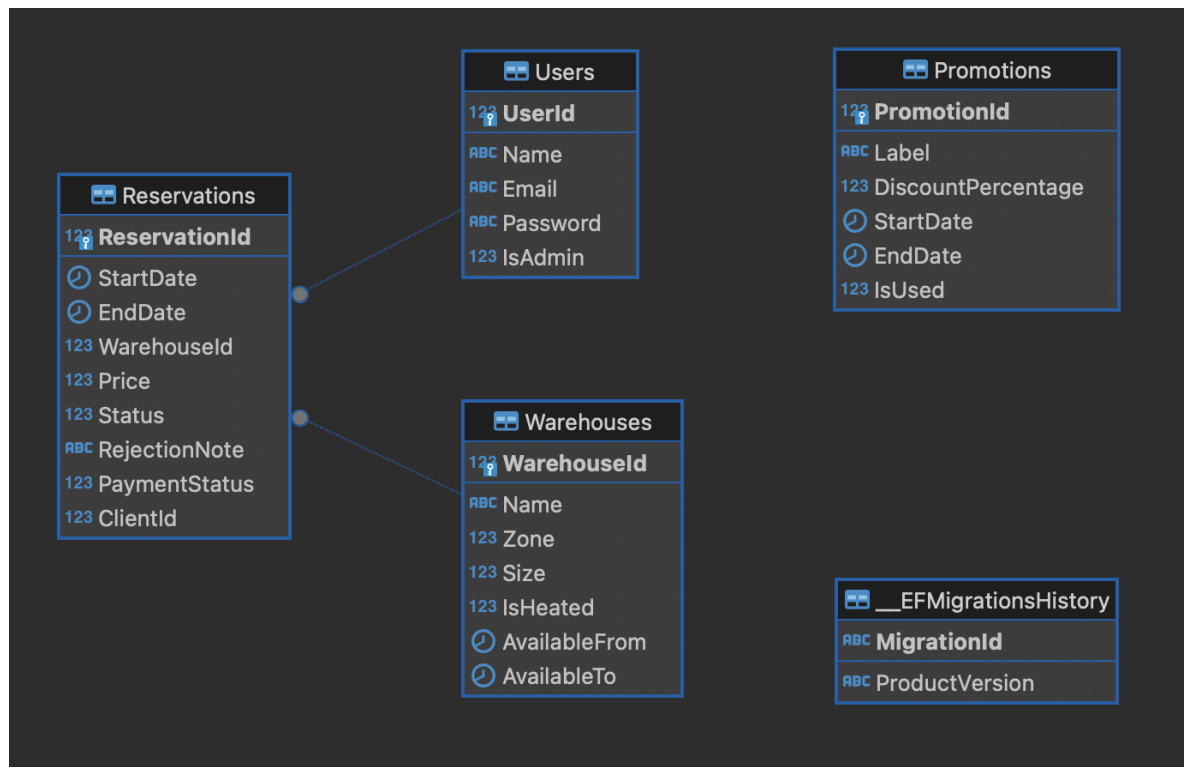


3. Reserva exitosa de un warehouse



D - Modelo de Tablas de la base de datos

ER Diagram:



El diseño de las clases es igual al de las tablas finales. Lo unico que se diferencia en el diagrama es la tabla **__EFMigrationsHistory** que es la que EF usa para mantener el historial de las migraciones aplicadas a la base de datos.

E - Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas

Actualización de la primera entrega

En el proyecto DepoQuick, se tomaron diversas decisiones de diseño para garantizar un funcionamiento eficiente y coherente del sistema. A continuación, se describen algunas de las principales decisiones tomadas en relación a los mecanismos generales:

1. **Interfaz de usuario y dominio**

Tenemos un modelado de separación por capas que permite garantizar la independencia de la lógica de negocio, de los datos y la interfaz de usuario. El usuario interactúa con la interfaz, la interfaz con los servicios y estos últimos con los repositorios. Esta arquitectura permite una mejor organización y mantenimiento del código, así como una mayor independencia entre los diferentes componentes.

2. **Persistencia de datos**

Como ya mencionamos anteriormente, en esta segunda etapa del proyecto la persistencia de datos se maneja a través de Entity Framework Core, un ORM que facilita la interacción con nuestra base de datos, Azure SQL Edge. Esta misma se encuentra alojada en un contenedor Docker para una mayor portabilidad y la facilitación del despliegue en diferentes entornos.

Para las pruebas, se utiliza SQLite y se crea la base de datos sin migraciones. Esto se hace con el objetivo de tener un entorno de prueba aislado, lo que aporta simplicidad y eficiencia al proceso de desarrollo. De esta manera, podemos probar nuestras funcionalidades sin afectar los datos de producción.

En el entorno de producción, se utiliza Azure SQL Edge y se aplican todas las migraciones pendientes para mantener la base de datos actualizada. Esto asegura que cualquier cambio en el esquema de la base de datos se refleje correctamente en producción.

En el proyecto DataAccess se utilizó el patrón Repository para manejar la gestión de datos. Esta capa proporciona una abstracción entre la lógica de negocio y la fuente de datos subyacente, lo que permite un fácil intercambio entre diferentes mecanismos de almacenamiento.

3. **Manejo de errores y excepciones**

Se ha implementado un mecanismo sólido de manejo de errores y excepciones en todo el sistema. Para manejar las excepciones en el front-end, se utilizan bloques try-catch al momento de enviar los datos ingresados en los formularios. Esto permite capturar y manejar cualquier excepción que pueda ocurrir durante el proceso de envío de los datos y responder de manera rápida e informativa al usuario en caso de errores.

Además, para mostrar mensajes de validación al usuario, se utilizó el componente de validación

de Blazor, `ValidationMessage`, junto con las `DataAnnotations`. Estas componentes facilitan la presentación de mensajes de error y validación relacionados con el ingreso de datos en formularios.

También se incorporan excepciones que se lanzan en casos específicos, permitiendo identificar y manejar situaciones excepcionales de manera precisa y controlada. Esto asegura que los errores sean registrados y se proporcione una respuesta adecuada al usuario, mejorando la usabilidad y la experiencia general del sistema.

En resumen, en los archivos `.razor` de DepoQuick se utilizan bloques `try-catch` para manejar excepciones, mientras que los componentes de validación de Blazor (como `ValidationMessage`) facilitan la presentación de mensajes de error al usuario. Además, en el código se implementan excepciones específicas para capturar situaciones excepcionales y proporcionar una respuesta adecuada.

Ahora también hemos incorporado el uso de `DataAnnotations` en EF Core para mejorar aún más la robustez y la eficiencia de nuestro sistema. Estos atributos se aplican a las propiedades de las entidades para configurar la base de datos y validar los mismos. Utilizamos etiquetas como `[Required]` y `[MaxLength]` para asegurarnos de cumplir con los requerimientos funcionales y no funcionales.

Estas decisiones de diseño fueron tomadas para garantizar un funcionamiento eficiente y una mayor flexibilidad. Cada una de estas decisiones se realizó pensando en la mantenibilidad y extensibilidad del sistema.

Segunda entrega

En esta etapa, prestamos especial atención a la aplicación de principios S.O.L.I.D y GRASP.

Un buen ejemplo donde se ven reflejados los principios SOLID es en la feature de exporte de reservas.

Aquí mostramos cómo integramos todos los principios S.O.L.I.D aquí:

- **Single Responsibility Principle:** Cada clase tiene una única responsabilidad. Por ejemplo, `ReservationExporter` se encarga únicamente de exportar reservas, y `reservationService` relega esa responsabilidad a él.
- **Open-Closed Principle:** La interfaz genérica `pe` Las clases `CSVExportStrategy` y `TSVExportStrategy` están abiertas para la extensión (puedes crear más estrategias de exportación), pero cerradas para la modificación (no necesitas cambiar estas clases cuando agregas una nueva estrategia de exportación).
- **Liskov Substitution Principle:** La clase `ReservationExporter` puede usar cualquier estrategia de exportación que implemente la interfaz `IExportStrategy<Reservation>`. Esto significa que un objeto de tipo `IExportStrategy<Reservation>` puede ser reemplazado por objetos de sus subtipos (`CSVExportStrategy`, `TSVExportStrategy`) sin afectar la corrección del programa.
- **Interface Segregation Principle:** La interfaz `IExportStrategy<T>` es específica, se mantiene simple y no está sobrecargada con métodos que las clases concretas no precisen/quieran implementar.

- **Dependency Inversion Principle:** La clase ReservationExporter depende de la abstracción IExportStrategy<Reservation> en lugar de las implementaciones concretas (CSVExportStrategy, TSVExportStrategy).

F - Análisis de los criterios seguidos para asignar las responsabilidades

Al asignar responsabilidades tuvimos presentes los patrones **GRASP**, que son quienes describen los principios fundamentales de la asignación de responsabilidades a objetos.

Siguiendo con el enfoque que presentamos en la primera entrega, nos enfocamos en que nuestras clases tengan **bajo acoplamiento**, mejorando aun más la versión de la entrega pasada con el refactor en PriceService. Todas nuestras clases además presentan un **alto nivel de cohesión**, presentando claramente su razón de ser y no sobre abarcando problemas.

Al querer tener el mayor nivel de cohesión, decidimos mantener PriceService como una **fabricación pura** (clase artificial) impulsando así, en todo el proyecto, la alta cohesión y bajo acoplamiento.

Utilizamos intermediarios de **indirección** tanto para trabajar con el frontend como el backend. Para el frontend implementamos el uso de DTOs (Data Transfer Objects), quienes agrupan varios datos en una sola estructura, facilitando su transporte a través de la red, y adaptando los datos a las necesidades específicas de cada capa de la aplicación, sin que estas tengan que conocer los detalles de implementación de las otras.

El intermediario de indirección para el acceso a datos son los repositorios. Los servicios de nuestra aplicación hacen uso de ellos para crear, leer, actualizar o borrar entidades de la base de datos. Ya los mencionamos anteriormente, proveen una capa de abstracción entre la base de datos y la lógica del negocio. Facilita el acceso a los datos y desacopla los servicios del acceso a datos.

También cumplimos con la **ley de Demeter**, como se puede apreciar en los diagramas presentados en el documento, nuestras clases tienen asignadas sus responsabilidades de manera tal de minimizar su necesidad de conocer la estructura interna de objetos indirectos.

A su vez se siguieron criterios como:

1. **Principio de responsabilidad única (Solid):** Se asignan responsabilidades claras y específicas a cada componente del sistema. Cada clase o componente tiene una única responsabilidad y está enfocado en realizar una tarea específica.
Ej: PriceService, cuya única responsabilidad es calcular el precio estimado para una reserva. La clase Reservation no realiza el cálculo en sí, sino que utiliza el servicio de PriceService para obtener el precio necesario. De esta manera, se evita que la clase de reserva tenga que preocuparse por el cálculo y se delega esa responsabilidad específica al PriceService.
2. **Separación de intereses:** Se buscó separar los diferentes intereses y aspectos del sistema en componentes distintos. Por ejemplo, se separaron los modelos de datos en el namespace DepoQuick.Models, la lógica de negocio en DepoQuick.Backend, el acceso a los datos en DepoQuick.DataAccess, entre otros. Esta separación permite una mejor organización y una mayor cohesión entre los componentes relacionados.

3. **Jerarquía y relación de dependencias:** Se estableció una jerarquía y una relación de dependencias entre los componentes del sistema. Por ejemplo, el front solamente habla con los servicios, que son los que tienen la lógica de negocio y las clases de servicios dependen de los repositorios para acceder a los datos, no pueden acceder directamente.

Los criterios específicos de cada paquete fueron explicados antes en el detalle de las responsabilidades de cada paquete.

4 - Mantenibilidad y Extensibilidad

El acceso a los datos mediante el patrón Repository en DepoQuick permite una gran mantenibilidad y extensibilidad en el proyecto. Gracias a esta estructura, es sencillo extender o modificar la funcionalidad de acceso a los datos con el mínimo impacto en otras partes del sistema.

Por ejemplo, si se desea cambiar la fuente de datos de los repositorios de una base de datos en memoria a una base de datos permanente como SQL Server, este cambio puede realizarse de manera relativamente sencilla. Al tener una capa de abstracción de repositorios, las clases de lógica de negocio y de interfaz de usuario no necesitan ser modificadas en absoluto, lo que minimiza el impacto y el acoplamiento.

Además, para agregar nuevas funcionalidades relacionadas con la gestión de almacenes, usuarios u otras entidades, se pueden implementar nuevos repositorios específicos sin afectar significativamente otras partes del sistema. Esto permite que DepoQuick sea flexible y escalable, ya que se pueden agregar nuevas características y adaptar la lógica de acceso a los datos de manera modular.

En esta segunda parte, el patrón de repositorio nos facilitó muchísimo dos grandes cambios. El pasaje de la base de datos en memoria a la base de datos Azure SQL Edge con Entity Framework, y el uso de Sqlite en memoria con Entity Framework para los tests. Estos dos cambios, que a primera vista parecen estructurales, resultaron triviales (se cambiaron menos de 100 líneas en 5 archivos, más que nada en los repositorios) con el aislamiento de la lógica de negocios con respecto al acceso a datos.

5 - Análisis de Dependencias

Actualización de la primera entrega

Para recapitular, en la **entrega anterior** identificamos que:

- PriceService depende de PromotionService ya que accede a las promociones con el fin de aplicarlas al cálculo del precio. Si PromotionService presentara cambios, habría que revisar PriceService.
- PriceService.CalculatePrice recibe un Warehouse como parámetro, y lo usa en su implementación. Si Warehouse presentara cambios, habría que revisar PriceService.CalculatePrice.

En **esta segunda parte**, con respecto a la entrega anterior:

- PriceService dejó de depender de PromotionService, ya que se movió el método que usaba a uno privado de sí mismo. Ahora depende de un repositorio de Promotion, provisto a través de la interfaz IRepo<Promotion, int> buscando inversión de dependencias.
- PriceService dejó de depender de Warehouse, ya que ahora solamente se pasan los atributos necesarios para calcular el precio (tamaño y calefacción).

Segunda entrega

En cuanto a ReservationExporter (la clase responsable de exportar los datos de reserva) depende únicamente de Reservation. Además, utiliza (es decir que también depende) de la interfaz IExportStrategy<T>, quien define la firma para las estrategias de exportación.

Dependiendo de la implementación concreta (CSVExportStrategy o TSVExportStrategy), se exportarán los datos de las reservas en su formato correspondiente. Las clases CSVExportStrategy y TSVExportStrategy implementan la interfaz anteriormente mencionada. Ambas dependen de Reservation para obtener los datos específicos que van a exportar, si por ejemplo el nombre de WarehouseId cambiara en Reservation, deberíamos cambiarlo también en CSVExportStrategy y TSVExportStrategy.


6 - Cobertura de Pruebas Unitarias

Primera Entrega

En nuestro caso, **evidenciamos el procedimiento de TDD** en nuestros commits de la branch feature/login, más precisamente trabajando en la funcionalidad del signup.

Comentario: No evidenciamos en esta documentacion el TDD en PriceService ya que a esa altura ya nos encontrabamos implementando los [Conventional Commits](#).

[REFACTOR]: AuthService SignUp should throw when password does not contain a symbol

 brainsaysno authored and victoriachappuis committed 3 weeks ago


[GREEN]: AuthService SignUp should throw when password does not contain a symbol

 brainsaysno authored and victoriachappuis committed 3 weeks ago


[RED]: AuthService SignUp should throw when password does not contain a symbol

 brainsaysno authored and victoriachappuis committed 3 weeks ago

[REFACTOR]: AuthService SignUp should throw when email is invalid

 brainsaysno authored and victoriachappuis committed 3 weeks ago


[GREEN]: AuthService SignUp should throw when email is invalid

 brainsaysno authored and victoriachappuis committed 3 weeks ago


[RED]: AuthService SignUp should throw when email is invalid

 brainsaysno authored and victoriachappuis committed 3 weeks ago


[REFACTOR]: AuthService SignUp should throw when name is longer than 100 characters

 brainsaysno authored and victoriachappuis committed 3 weeks ago

[GREEN]: AuthService SignUp should throw when name is longer than 100 characters

 brainsaysno authored and victoriachappuis committed 3 weeks ago

[RED]: AuthService SignUp should throw when name is longer than 100 characters

 brainsaysno authored and victoriachappuis committed 3 weeks ago

Usando de ejemplo el primer RED-GREEN-REFACTOR:

Test Implementado

```
1 + using DepoQuick.Services;
2 +
3 + namespace DepoQuick.Tests.Services
4 + {
5 +     [TestClass]
6 +     public class AuthService_SignUp
7 +     {
8 +         private readonly AuthService _authService;
9 +         private readonly string _validName = "John Doe";
10 +        private readonly string _validEmail = "example@email.com";
11 +        private readonly string _validPassword = "Pass123#";
12 +
13 +        public AuthService_SignUp()
14 +        {
15 +            _authService = new AuthService();
16 +        }
17 +
18 +        [TestMethod]
19 +        public void SignUp_NameLongerThan100_ShouldThrow()
20 +        {
21 +            string name = new string('v', 101);
22 +
23 +            Assert.ThrowsException<ArgumentOutOfRangeException>(() => _authService.Signup(name, _validEmail, _validPassword,
24 +                _validPassword));
25 +        }
26 +    }
```

La configuración de la clase de prueba es la siguiente:

- La clase `AuthService_SignUp` está decorada con el atributo `[TestClass]` para indicar que es una clase de prueba en el marco de pruebas utilizado.
- Se crea una instancia de la clase `AuthService` llamada `_authService` para poder llamar a los métodos de prueba.
- Se definen varias variables para contener datos válidos para el nombre, el correo electrónico y la contraseña para rellenar la información que no será probada en cada test.

En el constructor de la clase `AuthService_SignUp`, la instancia de `AuthService` se inicializa para poder utilizarla en los métodos de prueba.

El método de prueba `SignUp_NameLongerThan100_ShouldThrow` se encarga de probar el caso en el que el nombre tiene una longitud mayor a 100 caracteres y se espera que se lance una excepción.

Dentro del cuerpo del método de prueba, se crea una cadena de caracteres 'v' repetida 101 veces para simular un nombre demasiado largo. Luego se utiliza el método `Assert.ThrowsException` para verificar que se lance una excepción del tipo `ArgumentOutOfRangeException` al llamar al método `SignUp` de `AuthService` pasando el nombre, el correo electrónico y la contraseña válidos.

Código de función testeada

```
1 + namespace DepoQuick.Services
2 + {
3 +     public class AuthService
4 +     {
5 +         public void Signup(string name, string email, string password, string confirmation)
6 +         {
7 +         }
8 +     }
9 + }
```

Al correr el test y recibir un “failed”, nos aseguramos que las pruebas son efectivas, significando que si luego la función ya implementada cuenta con errores, el test nos advertirá debidamente y no dará siempre true.

Cambios en función en el commit GREEN

```
4     {
5         public void Signup(string name, string email, string password, string confirmation)
6         {
7 +             throw new ArgumentOutOfRangeException(nameof(name));
8         }
9     }
10 }
```

Aunque el test nos da bien, aún no cumplimos con el verdadero sentido de la funcionalidad

Cambios en función en el commit REFACTOR

```
5     public void Signup(string name, string email, string password, string confirmation)
6     {
7 +         if (name.Length > 100)
8             throw new ArgumentOutOfRangeException(nameof(name));
9     }
10 }
```

Finalmente, añadimos el ‘if’ statement que faltaba.

Segunda entrega

Nuevamente logramos el 100% efectivo de cobertura del código, siguiendo la metodología de TDD. Evidencia de cobertura en el [anexo](#).

Instructivo para acceder a la aplicación

Descargar el release de la v2.0 de la sección de releases de github, unzippear y queda una carpeta **publish**. Dentro de publish hay un ejecutable **DepoQuick.Frontend.exe** que es el que debemos ejecutar para correr nuestra aplicación que levanta en el puerto **5000 con http** y **5001 con https**.

Para ingresar a la aplicación hay 6 usuarios, cuyos usuarios y contraseñas se pueden ver en los INSERTS provistos en el repo. Aquí dejamos dos usuarios para probar, el administrador y un usuario común:

- User: **linustorvalds@ort.edu.uy** - Password: **F3doraR0cks#**
- User: **robertcmartin@ort.edu.uy** - Password: **#N0TestN0Beer**

La base de datos se debe levantar mediante el comando **docker compose up** que se expone en el puerto 1433 y se accede con las siguientes credenciales:

- User: **sa**
- Pass: **DB_password!#**

Los datos guardados en la base de datos se replican en un volumen local **db-data** por lo que son persistentes a través de recreaciones del contenedor.

Las credenciales de acceso de la aplicación a la base de datos pueden cambiarse modificando el archivo de configuración **publish/appsettings.json**.

Anexo

- [Video de pruebas unitarias pasando y code coverage 100%](#)
- [Video de guía de la aplicación](#)