

Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 1

Diseño de aplicaciones 1

Fecha: 9/5/2024

Nicolás Russo 227286
Victoria Chappuis 278311

Docentes: Gastón Mosqués, Martín Radovitzky,
Facundo Arancet

Grupo: M5A

[Repositorio](#)

2 - Descripción general del trabajo y del sistema	2
3 - Justificación de Diseño	4
Diagrama de paquetes	4
Diagramas de Clases	4
Paquete Providers	4
Paquete DepoQuick.Backend.Repos	5
Paquete DepoQuick.Backend.Models	6
Paquete DepoQuick.Backend.Services	7
Paquete DepoQuick.Backend.Dtos	8
Explicación de los mecanismos generales y descripción de las principales decisiones de diseño tomadas	8
Análisis de los criterios seguidos para asignar las responsabilidades	9
4 - Mantenibilidad y Extensibilidad	10
5 - Análisis de Dependencias	10
6 - Cobertura de Pruebas Unitarias	11
Test Implementado	13
Código de función testeada	14
Cambios en función en el commit GREEN	14
Image	14
Cambios en función en el commit REFACTOR	14
7 - Casos de prueba para baja de depósito y alta de depósito	14

2 - Descripción general del trabajo y del sistema

DepoQuick es una innovadora iniciativa que pretende transformar el mercado de alquiler de depósitos, ofreciendo una solución eficaz y accesible para quienes requieren espacio adicional de almacenamiento. La misión de DepoQuick consiste en simplificar el proceso de alquiler de depósitos, equiparándolo a la sencillez de realizar una compra en línea.

La solución de DepoQuick, a excepción de la interfaz de usuario, se desarrolló utilizando TDD (Desarrollo Guiado por Pruebas).

Este mismo consiste en escribir las pruebas antes de escribir el código. El proceso se divide en 3 fases:

- Red: Escribir un test que falle, representando el comportamiento deseado aún no implementado
- Green: Escribir el código mínimo y necesario para que la prueba pase.
- Refactor: Optimizar y mejorar el código ya escrito.

Todas las pruebas unitarias impletadas cumplen con las características F.I.R.S.T.

Con TDD logramos cubrir el 100% del código realizado.

Ambos componentes, la lógica y la interfaz, fueron desarrollados utilizando C# en .NET Core 6. La plantilla *Blazor Server App* fue empleada para la creación del proyecto de frontend (DepoQuick.Frontend), luego se utilizó la plantilla de *Class Library* para el proyecto de backend (DepoQuick.Backend), se utilizó la plantilla de *MSTest* para el proyecto de pruebas unitarias (DepoQuick.Test) y se utilizó *GitHub* como repositorio de código.

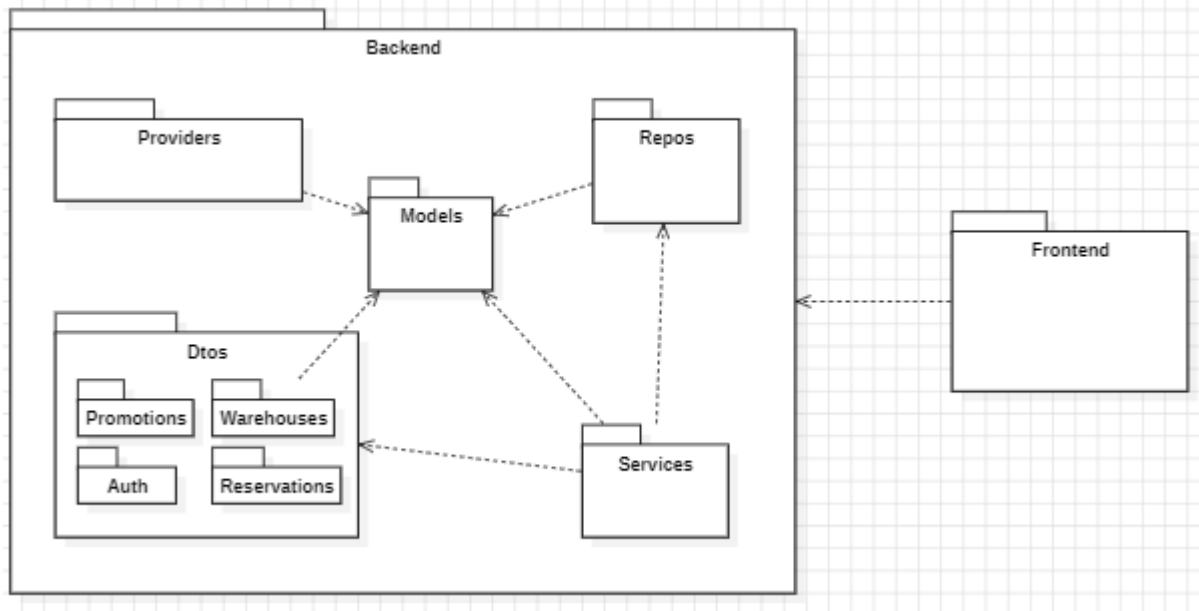
En nuestro proyecto, utilizamos Git Flow como modelo de ramificación de Git. En el proyecto hay dos ramas principales: 'master', que representa la versión estable en producción, y 'develop', que es la rama de desarrollo. Además, creamos ramas de características ('feature') para cada nueva funcionalidad o mejora específica. Asimismo, creamos ramas adicionales con nombres específicos para abordar necesidades particulares (por ejemplo: 'feature/warehouses-admin-dashboard' o 'feature/edit-promotions'). Esta estructura nos permitió trabajar en paralelo en diferentes áreas del proyecto y mantener un flujo de desarrollo ordenado. Además para hacer los mensajes de commits utilizamos [Conventional Commits](#) (por ejemplo: 'feat: add sign in signout flow with navbar').

Para garantizar la independencia de la lógica de negocio, de los datos y la interfaz de usuario, se separaron en distintos proyectos dentro de la solución y namespaces con responsabilidades claras dentro de estos. De esta manera, se logró una arquitectura modular y organizada que facilita el desarrollo y mantenimiento del proyecto.

Además, se implementaron repositorios en la solución. Esta decisión se tomó anticipando la futura transición de la base de datos actual (en memoria) a una base de datos permanente (con Entity Framework) en etapas posteriores del proyecto. Los repositorios proporcionan una capa de abstracción que facilitará la migración sin afectar significativamente la lógica de negocio y la interfaz de usuario. Funcionan como capa intermedia entre los datos y la lógica de negocio.

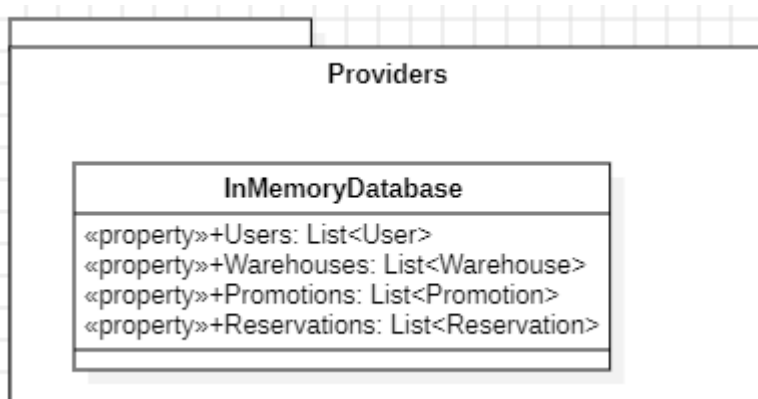
3 - Justificación de Diseño

Diagrama de paquetes



Diagramas de Clases

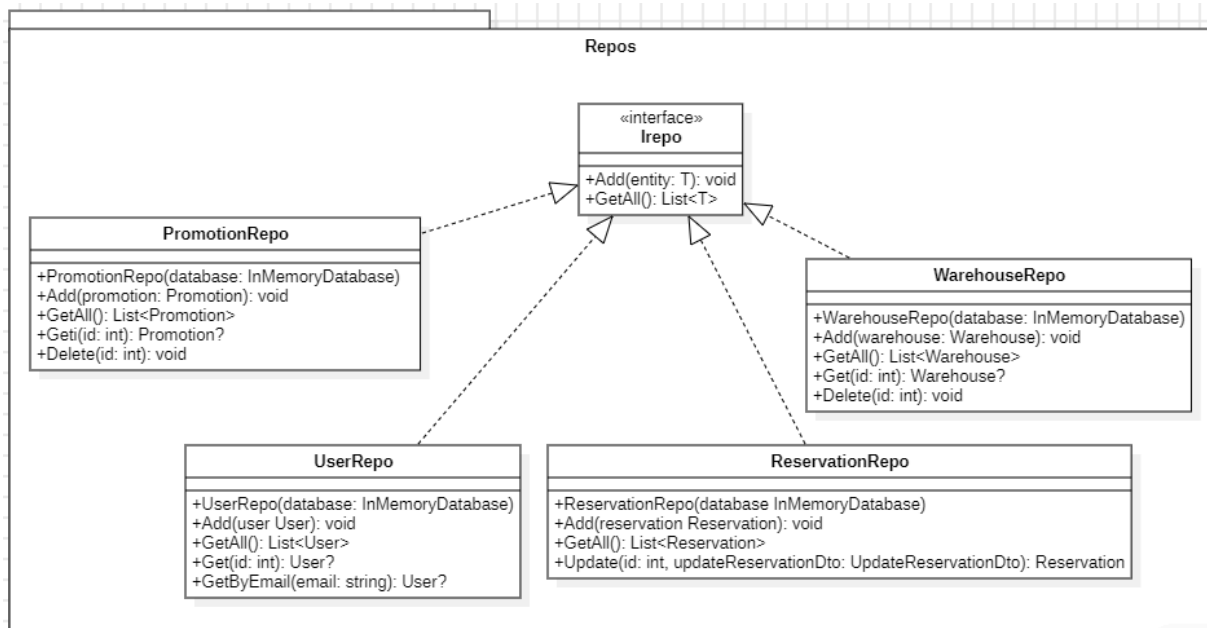
Paquete Providers



Clases: InMemoryDatabase

Responsabilidad: Proporcionar una implementación de base de datos en memoria para el proyecto. Actúa como una solución de almacenamiento temporal para los datos de la aplicación durante la ejecución. Forma ligera y eficiente de almacenar y recuperar datos sin la necesidad de una solución de almacenamiento persistente.

Paquete DepoQuick.Backend.Repos



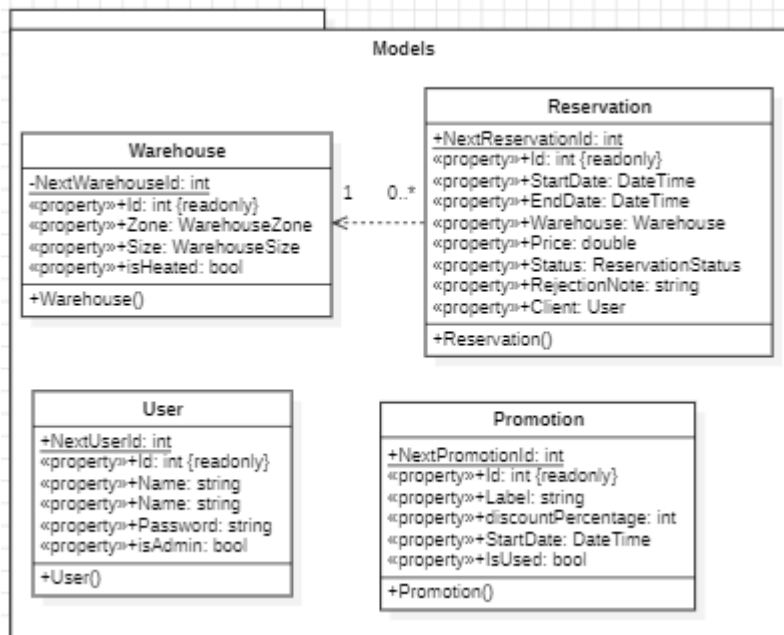
Clases: PromotionRepo, ReservationRepo, UserRepo, WarehouseRepo

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



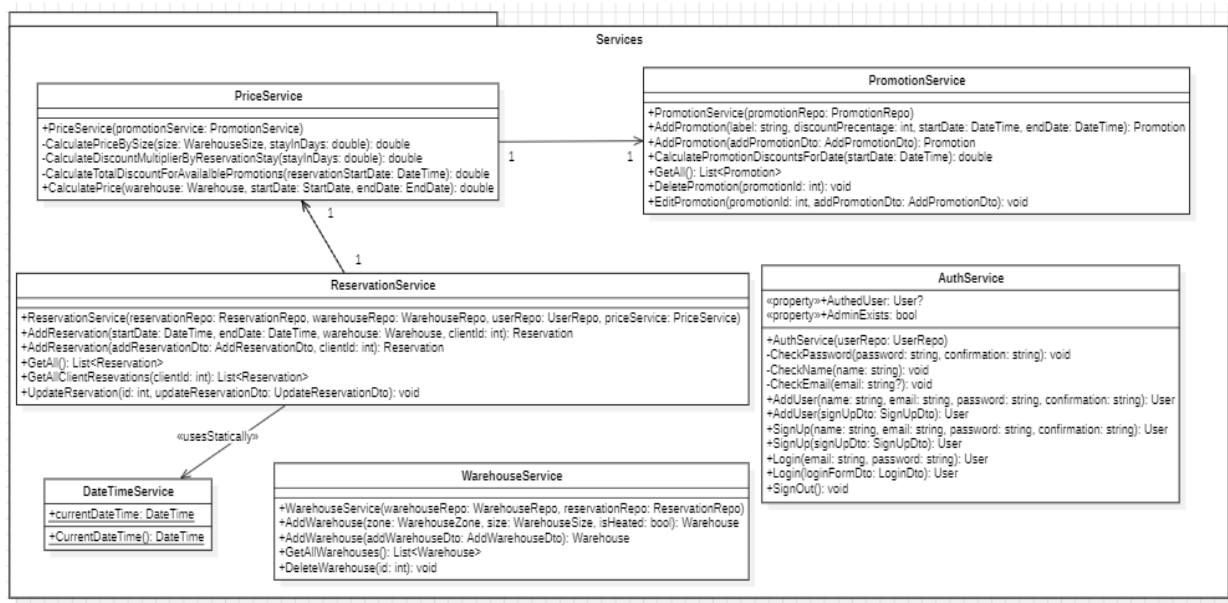
Clases: Promotion, Reservation, User, Warehouse

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 Raquítico”. 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 apegado a 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.

Paquete DepoQuick.Backend.Services

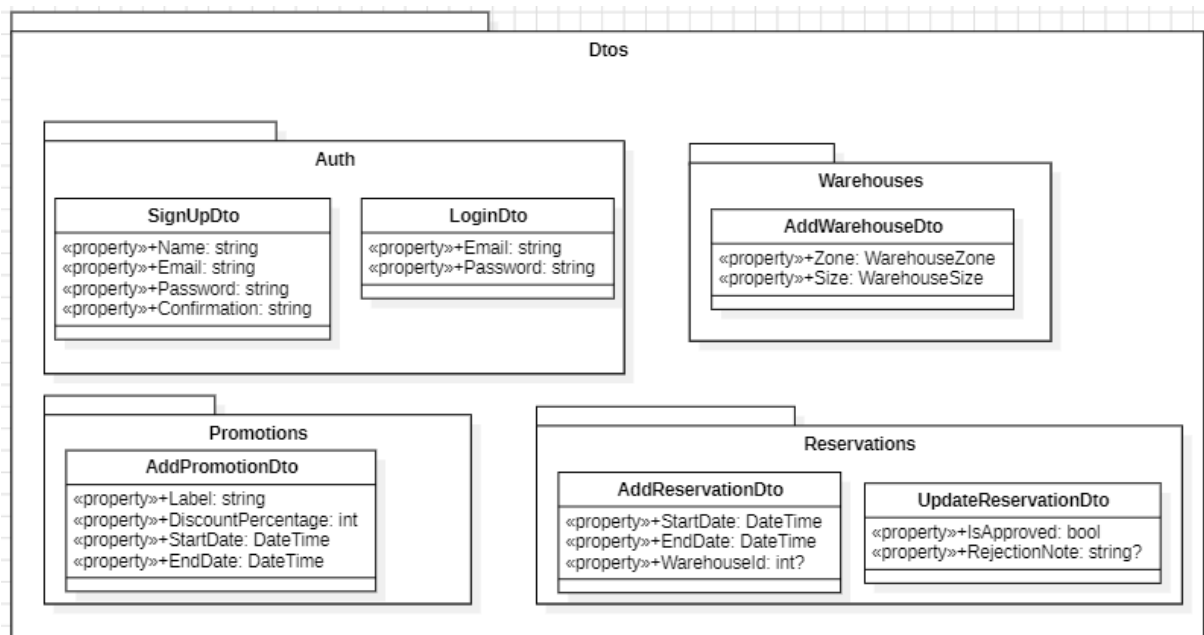


Clases: AuthService, PriceService, PromotionService, WarehouseService, ReservationService, DateTimeService

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.

Comentario: Podría haber sido parte de *Models* también, como se sugiere en la [documentación de DTOs de Microsoft](#). Este cambio quedará para la parte dos.

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

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. Podría verse una estructura similar a MVC (Modelo Vista Controlador).

En esta estructura, la interfaz de usuario, el paquete *DepoQuick.Frontend* actúa como la vista, el paquete *Services* desempeña el rol del controlador y el paquete *Models* (junto con la base de datos) representa el modelo. Esta arquitectura permite una mejor organización y mantenimiento del código, así como una mayor independencia entre los diferentes componentes.

2. **Persistencia de datos**

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, como la base de datos en memoria que usamos o bases de datos persistentes como SQL Server.

3. **Manejo de errores y excepciones**

Se ha implementado un mecanismo sólido de manejo de errores y excepciones en todo el sistema. Los errores Finalmente 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.

4. **Polimorfismo**

En el caso de DepoQuick, el polimorfismo puede verse en los repositorios a través de una interfaz común, *IRepo*. Al definir una interfaz para los repositorios, se establece un contrato que especifica los métodos y operaciones que deben ser implementados por las clases concretas de los repositorios. Esto permite tratar diferentes implementaciones de repositorios, como *WarehouseRepo* y *UserRepo*, de manera uniforme a través de la interfaz común.

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, extensibilidad y escalabilidad del sistema.

Análisis de los criterios seguidos para asignar las responsabilidades

Se siguieron criterios como:

1. **Principio de responsabilidad única:** 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 de reserva 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, los enumerados en DepoQuick.Enums, los modelos de datos de autenticación en DepoQuick.Dtos.Auth, 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, las clases de servicios dependen de los repositorios para acceder a los datos, no deben 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.

5 - Análisis de Dependencias

- 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`.

Tenemos un acoplamiento bajo dentro de todo en el servicio `PriceService`. Se busca reducir al mínimo teniendo la menor cantidad de dependencias posibles, las necesarias.

Separando la funcionalidad de cálculo de precio de los depósitos en el servicio `PriceService`, las responsabilidades de la misma están altamente relacionadas, por lo tanto tenemos una cohesión alta. Los métodos dentro de `PriceService` están directamente relacionados con esta funcionalidad y se complementan entre sí para lograr el objetivo general. No hay métodos innecesarios o que estén fuera de contexto dentro de la clase.


Incluso, se delegan ciertas funcionalidades específicas a otros servicios, como la búsqueda de promociones a `PromotionService`. Esto permite mantener una mayor cohesión en `PriceService`, ya que cada método se enfoca en su tarea específica sin llevar a cabo tareas que no le corresponden.

6 - Cobertura de Pruebas Unitarias

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 documentación el TDD en `PriceService` ya que a esa altura ya nos encontrábamos 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 +     }
27 }
```

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.

7 - Casos de prueba para baja de depósito y alta de depósito

Casos de prueba elaborados para la funcionalidad de **agregar un depósito**:

- Con campos válidos debe guardarlo en la base de datos
- Con campos válidos debe retornar la instancia creada
- Dos depósitos con datos distintos deben ser creados con ids distintos
- Dos depósitos con datos iguales deben ser creados con ids distintos

Casos de prueba elaborados para la funcionalidad de **eliminar un depósito**:

- Con campos válidos debe eliminarlo de la base de datos
- Con campos válidos no debe eliminar otros depósitos de la base de datos
- Con depósito no encontrado, no debería eliminar ningún depósito
- Depósito con reserva aprobada no debería ser eliminado
- Depósito con reserva pendiente no debería ser eliminado

Anexo

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