Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 1 - Diseño de Aplicaciones 2 - Agosto 2023

https://github.com/IngSoft-DA2-2023-2/194592-275723-246326

Luis Sanguinetti – 246326

Facundo Rodríguez - 194592

Leandro Olmedo - 275723

Docentes: Ignacio Valle, Marco Fiorito, Matías Salles

2023

Índice

Descripción general del trabajo **3**
Bugs o funcionalidades no implementadas 3
Diagramas de paquetes **3**
Diagrama general de paquetes de la solución 3
Diagrama de paquetes de Data 4
Diagrama de paquetes de Logic 5
Diagrama de paquetes de ApiModels 6
Diagramas de clases **7**
Diagrama de componentes 7
Descripción de jerarquías de herencia utilizadas **8**
Inyección de dependencias 9
Patrones 9
Principios de diseño 10
¿Cómo estos mecanismos apoyan a la mantenibilidad de la aplicación? 11
Oportunidades de mejoras **13**

Descripción general del trabajo

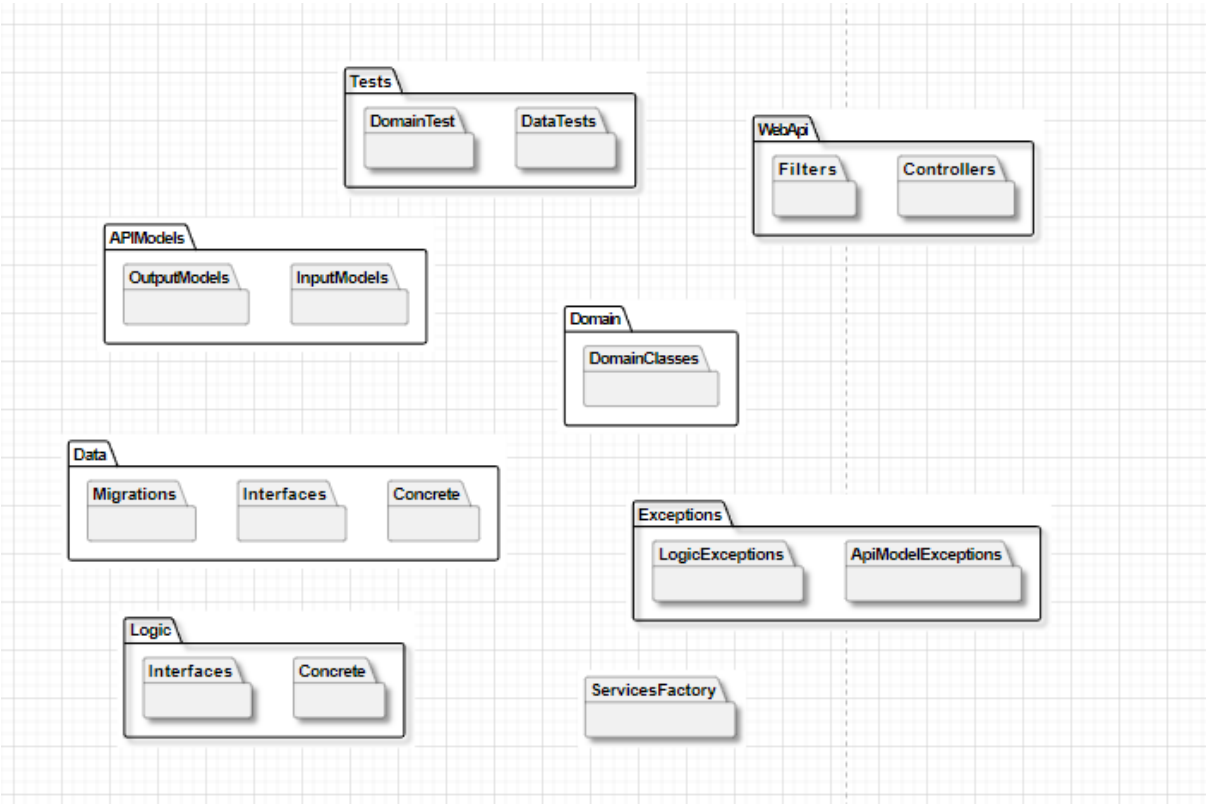
La solución que presentamos tiene muchas funcionalidades, algunas de estas son el manejo de usuarios, sus compras, carritos y productos. Estas se pueden crear, editar o destruir para adaptarse a las necesidades del sistema en un futuro. También maneja las promociones de las compras eligiendo la promoción aplicada para cada carrito. A estas partes principales se les generaron secciones auxiliares para poder manejar mejor el sistema por ejemplo el shoppingCartProduct que tiene los productos para cada shopping cart.

Bugs o funcionalidades no implementadas

En nuestro caso solo vimos un bug en el delete de promociones, al intentar borrar una Promocion, no se puede por los PromotionCondition que tiene anidados y deberíamos borrarlos en cascada. En el caso de las funcionalidades no implementadas tuvimos la parte de compra que no logramos implementar. Esta prácticamente terminada pero estamos teniendo algunos problemas con la misma asi que no va a ser agregada en este momento

Diagramas de paquetes

Diagrama general de paquetes de la solución



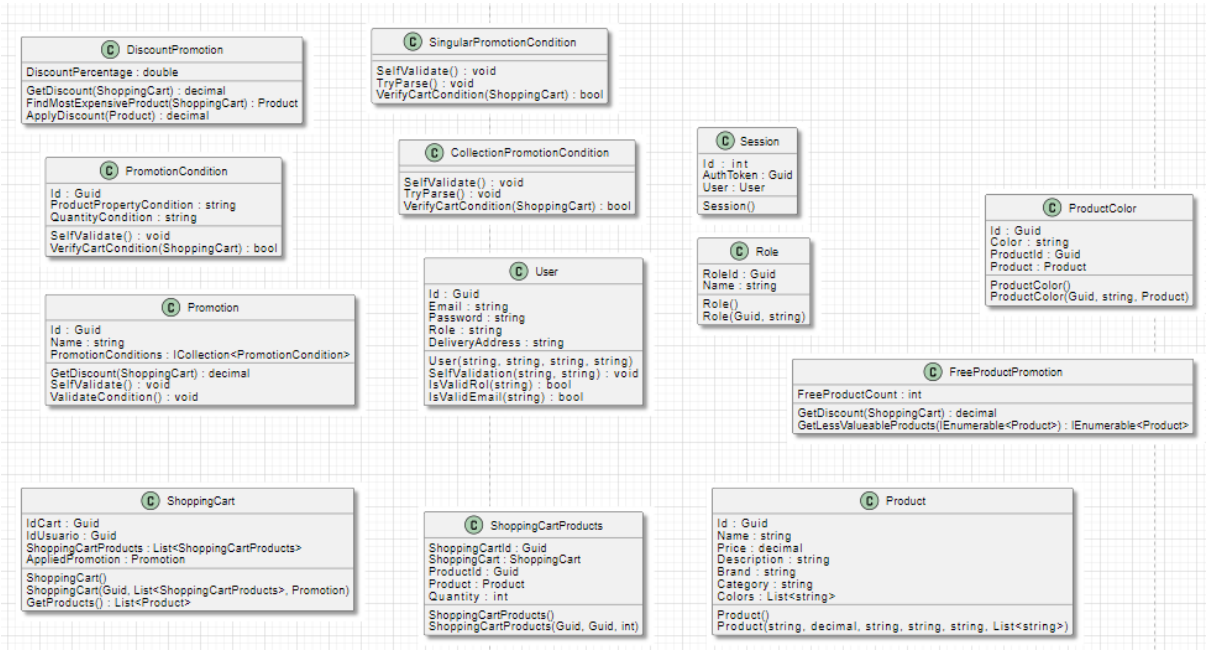


Diagrama de paquetes de Data

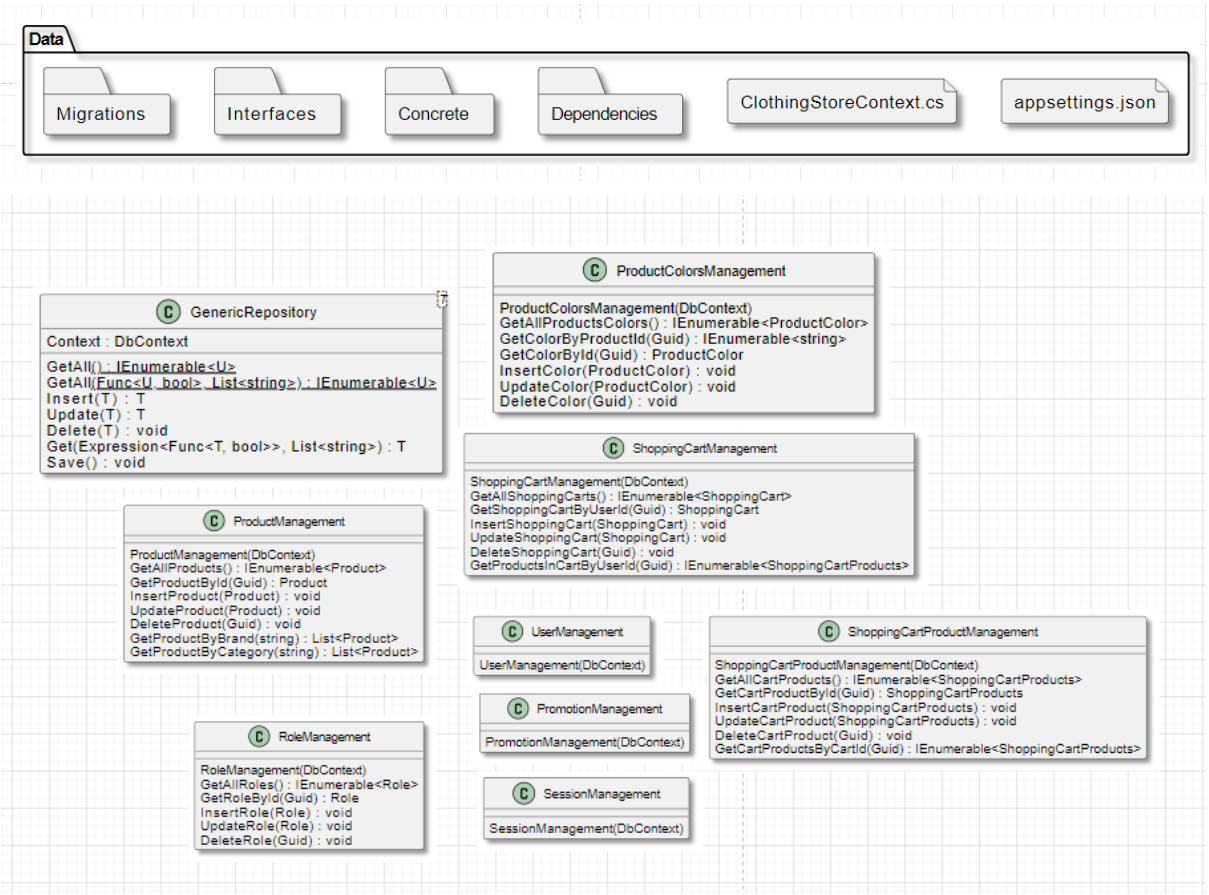


Diagrama de paquetes de Logic

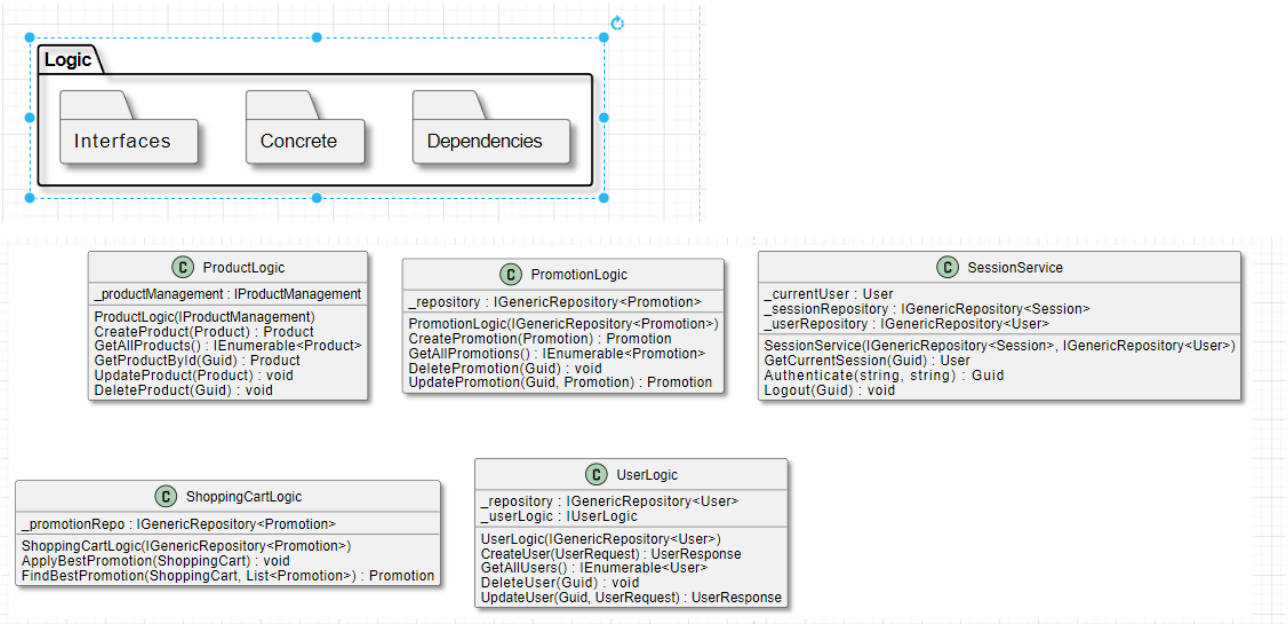


Diagrama de paquetes de Web API

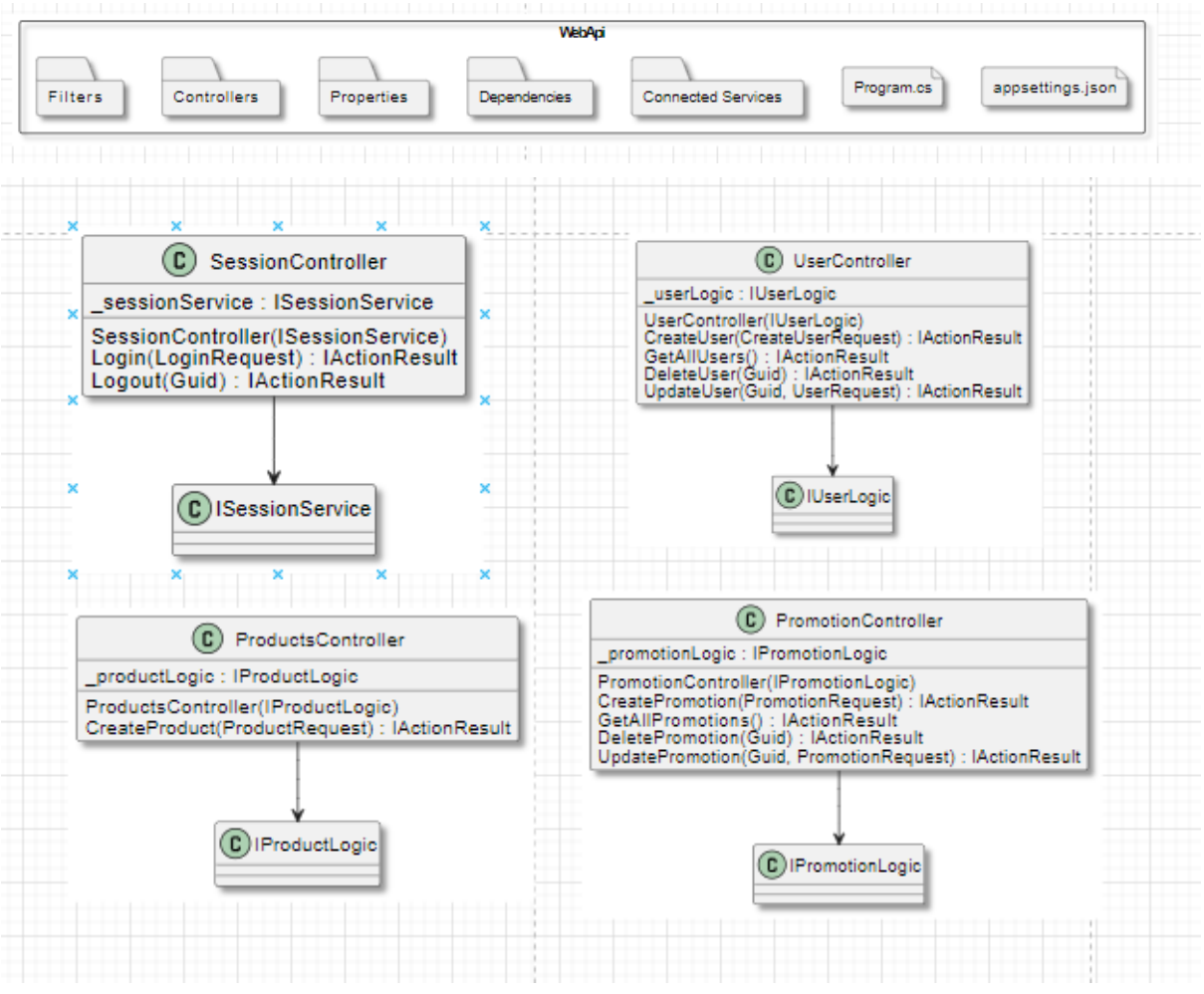
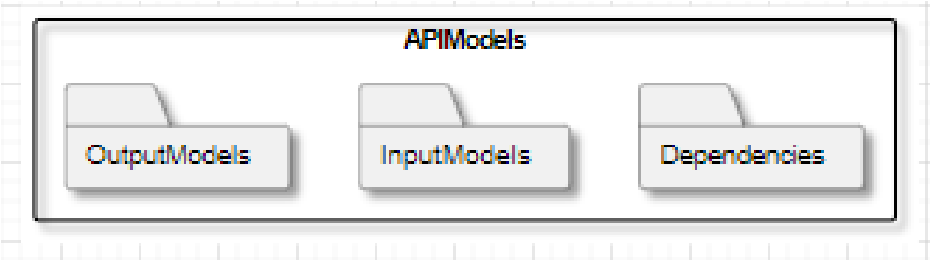
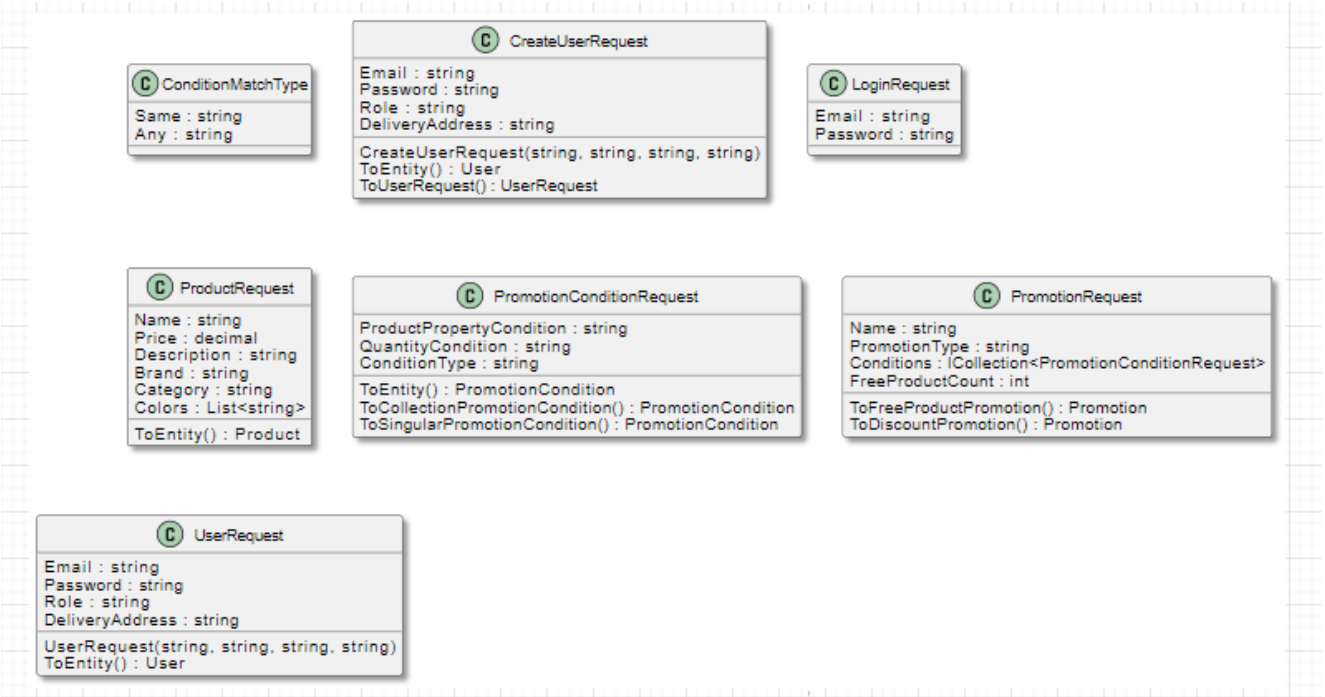


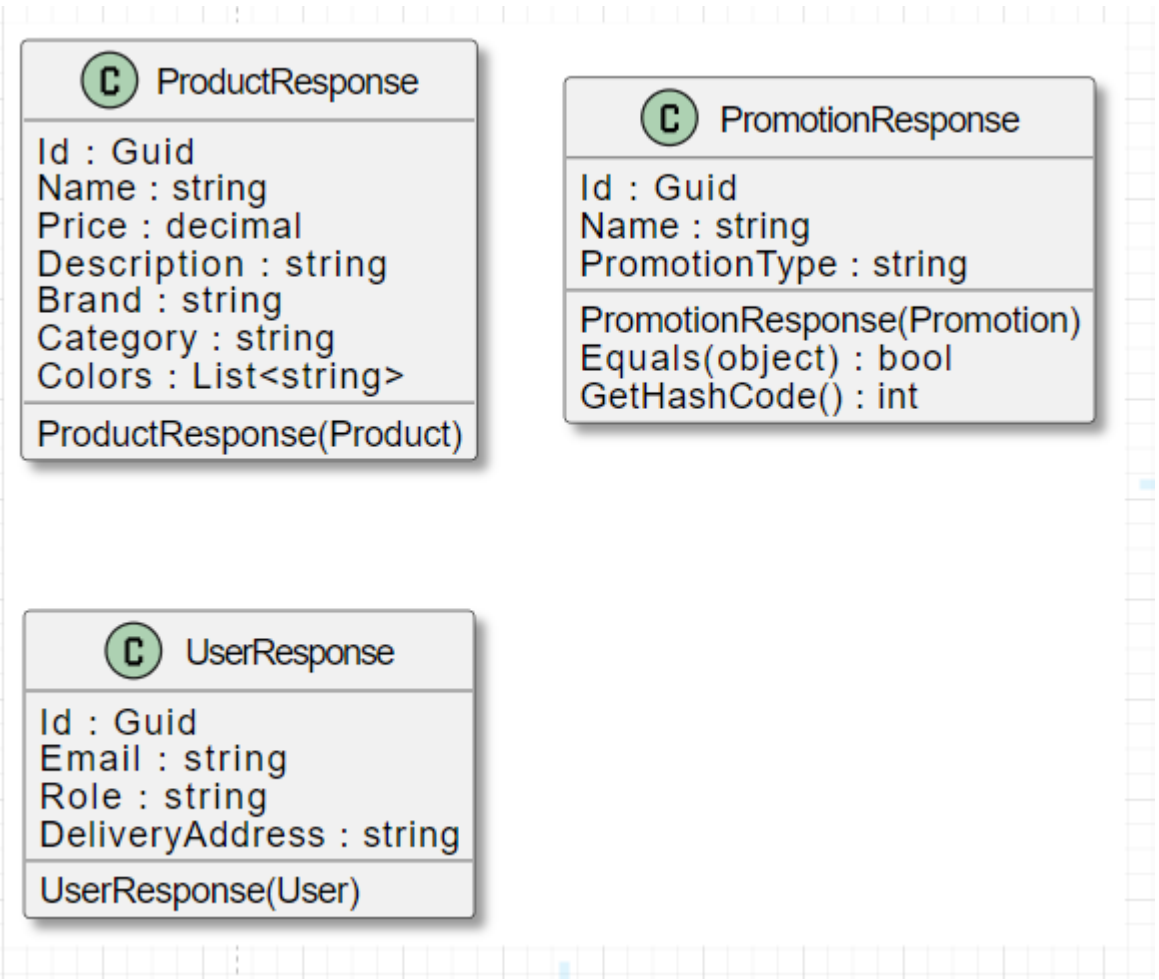
Diagrama de paquetes de ApiModels



Input model



Output Model


ProductResponse
Id : Guid
Name : string
Price : decimal
Description : string
Brand : string
Category : string
Colors : List<string>
ProductResponse(Product)
PromotionResponse
Id : Guid
Name : string
PromotionType : string
PromotionResponse(Promotion)
Equals(object) : bool
GetHashCode() : int
UserResponse
Id : Guid
Email : string
Role : string
DeliveryAddress : string
UserResponse(User)

Diagramas de clases

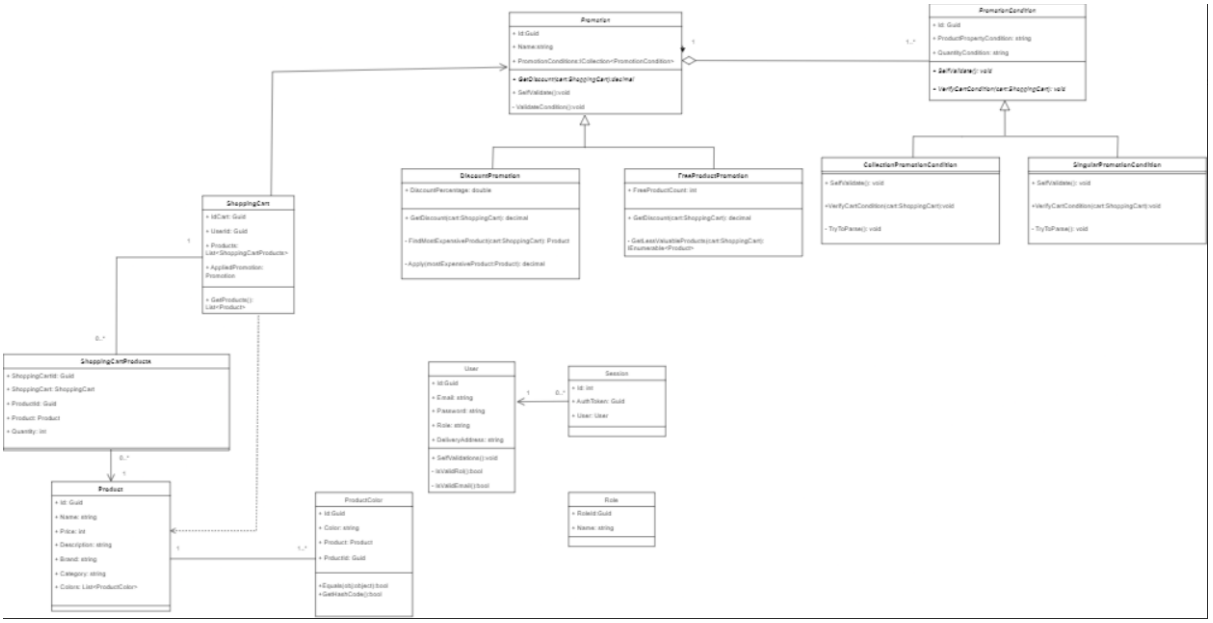
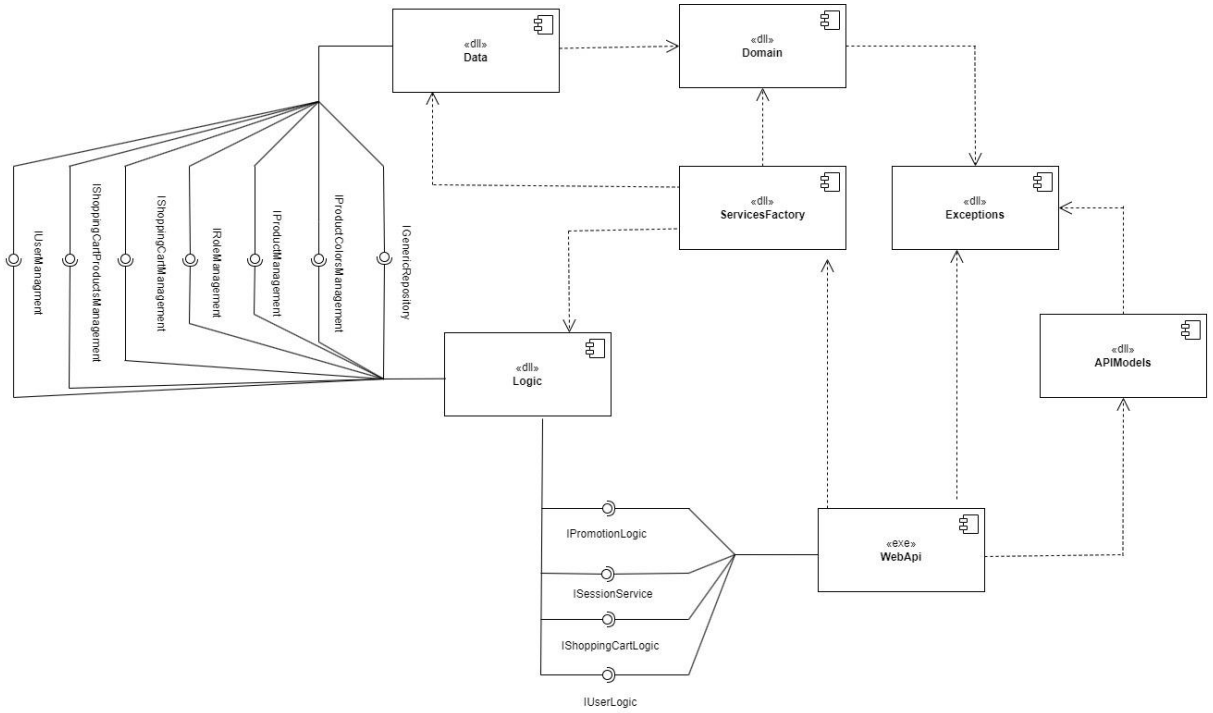


Diagrama de componentes



Diagramas de interacción

No llegamos a los diagramas de interacción

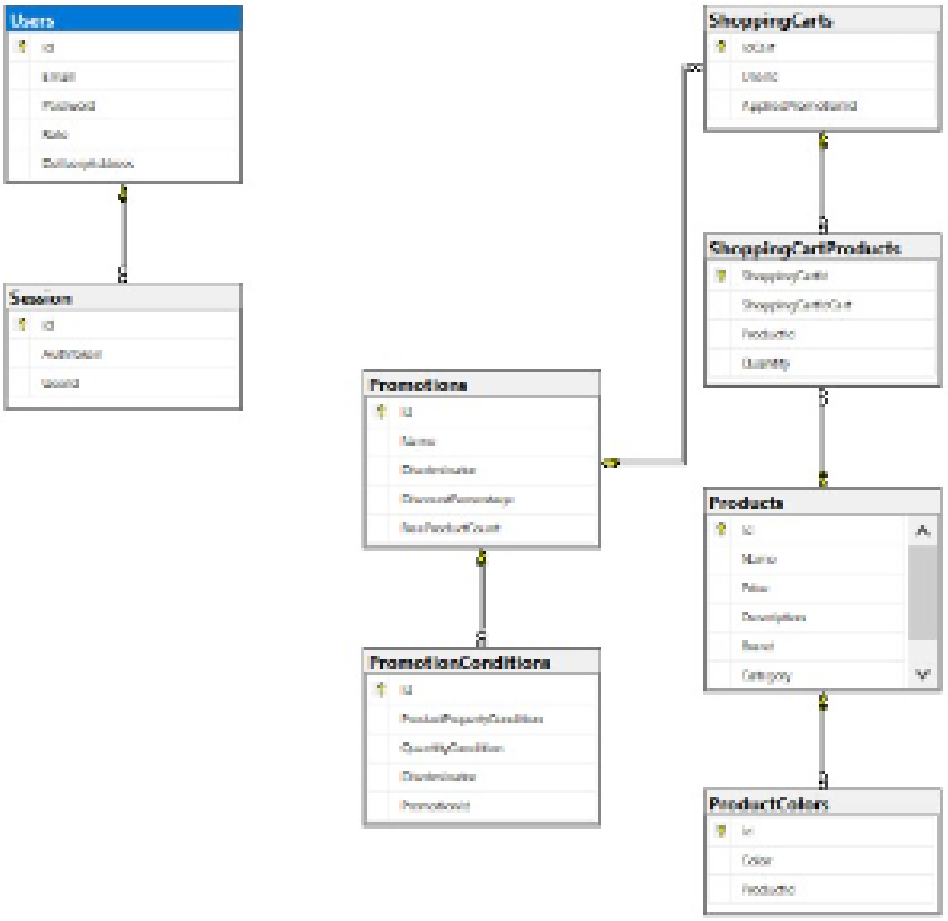
Descripción de jerarquías de herencia utilizadas

En nuestro proyecto, hemos adoptado una estrategia basada en los principios de diseño de patrones y código limpio. Para estructurar y organizar nuestras promociones, hemos introducido la clase base Promotion. Esta clase actúa como un punto de partida común para todas nuestras promociones, ya sean ofertas "3x1" o descuentos del "20%". Esta jerarquía nos permite gestionar de manera eficiente las distintas estrategias promocionales y garantizar coherencia en su implementación.

Adicionalmente, para representar las condiciones específicas de cada promoción, hemos diseñado una jerarquía paralela con la clase **PromotionCondition**. Esta estructura nos permite encapsular las lógicas y reglas relacionadas con las condiciones de aplicación de cada promoción, asegurando así una gestión clara y modular de las mismas.

Esta organización nos brinda la flexibilidad necesaria para extender y mantener nuestro sistema de promociones de manera eficaz, además de promover un código limpio y estructurado que es esencial en la ingeniería de software moderna.

Modelo de tablas de la estructura de la base de datos.



Justificación del diseño

Inyección de dependencias

A la hora de utilizar clases de la lógica y repositorios lo hacemos por medio de interfaces. Eso permite desacoplar las clases y facilitar la modificación o adición de funcionalidades . Para realizar la inyección de dependencias utilizamos una service factory la cual se encarga de asociar las implementaciones de las clases con sus respectivas interfaces

Patrones

Uno de los principales patrones que utilizamos fue Strategy, este puede ser visto tanto en la clase Promotion como en la clase PromotionCondition.
En el caso de la clase abstracta Promotion tiene un método abstracto GetDiscount que calcula el descuento de un carrito. Este método está implementado en dos clases concretas que calculan el descuento de una manera diferente dependiendo del tipo de promoción.

En el caso de PromotionCondition pasa algo similar ya que tiene un método que verifica si un carrito cumple con el requisito de la promoción, y la implementación de este método es diferente dependiendo del tipo de PromotionCondition.

Principios de diseño

En nuestro proyecto de desarrollo de software, hemos abrazado con entusiasmo los principios de diseño SOLID y los principios GRASP, que actúan como las bases fundamentales para lograr una arquitectura de software sólida y de alta calidad. Estos principios, respaldados por décadas de experiencia en ingeniería de software, han guiado nuestra toma de decisiones y han contribuido significativamente al éxito de nuestro proyecto.

Principios SOLID:

1. **Principio de Responsabilidad Única (SRP**): Hemos aplicado este principio asegurándonos de que cada clase tenga una única razón para cambiar. Esto ha simplificado el mantenimiento y ha mejorado la cohesión de nuestras clases.
2. **Principio de Abierto/Cerrado (OCP)**: Nuestro código ha sido diseñado de manera que sea abierto para la extensión pero cerrado para la modificación. Esto nos ha permitido agregar nuevas funcionalidades sin alterar el código existente.
3. **Principio de Sustitución de Liskov (LSP)**: Hemos garantizado que las subclases puedan ser usadas en lugar de las clases base sin cambiar el comportamiento esperado. Esto ha facilitado la creación de nuevas clases derivadas sin efectos secundarios no deseados.
4. **Principio de Segregación de Interfaces (ISP)**: Hemos dividido las interfaces en conjuntos más pequeños y específicos, lo que ha evitado que las clases implementen métodos innecesarios y ha promovido la cohesión.
5. **Principio de Inversión de Dependencia (DIP)**: Hemos utilizado la inversión de dependencia para desacoplar componentes y reducir la dependencia de detalles concretos. Esto ha facilitado la sustitución de componentes y la prueba unitaria.

Principios GRASP:

1. **Controlador**: Hemos utilizado patrones de controladores para gestionar la interacción entre el usuario y el sistema, manteniendo así una separación clara de responsabilidades.
2. **Experto**: Hemos asignado la responsabilidad a las clases que tienen la información necesaria para cumplir con esa responsabilidad, aprovechando al máximo el principio de experto.
3. **Creador**: Hemos aplicado el principio de creador para asignar la responsabilidad de la creación de objetos a las clases que tienen la información adecuada para hacerlo.
4. **Indirecto**: Hemos minimizado las dependencias directas entre clases y módulos, favoreciendo las dependencias indirectas a través de interfaces y abstracciones.
5. **Polimorfismo**: Hemos utilizado polimorfismo para permitir que distintas clases puedan ser tratadas de manera uniforme a través de interfaces comunes, lo que ha facilitado la extensibilidad y la reutilización.

La aplicación de estos principios SOLID y GRASP ha sido un pilar fundamental en la construcción de un sistema de software coherente, flexible y altamente mantenible. Esto no solo ha mejorado la calidad técnica de nuestro proyecto, sino que también ha contribuido en gran medida a su éxito general.

¿Cómo estos mecanismos apoyan a la mantenibilidad de la aplicación?

Principios de Diseño SOLID:

Los principios SOLID, que incluyen Responsabilidad Única (SRP), Abierto/Cerrado (OCP), Sustitución de Liskov (LSP), Segregación de Interfaces (ISP) e Inversión de Dependencia (DIP), han guiado nuestro diseño de clases y componentes. Estos principios han promovido la modularidad, la flexibilidad y la reutilización del código, lo que resulta en una aplicación más mantenible.

Principios GRASP:

Los principios GRASP, que incluyen Controlador, Experto, Creador, Indirecto y Polimorfismo, han informado nuestras decisiones de diseño y distribución de responsabilidades. Estos principios han ayudado a mantener la coherencia en nuestro sistema y a asegurar que las clases estén correctamente diseñadas y acopladas.

Manejo de Excepciones:

Hemos implementado un enfoque efectivo para el manejo de excepciones, utilizando un ExceptionFilter a nivel de controlador para capturar y formatear excepciones de manera consistente. Además, hemos creado excepciones personalizadas para manejar casos específicos de errores, mejorando la precisión y la claridad de los mensajes de error.

Mecanismo de acceso a datos utilizado

Gestión de Datos y Acceso a la Base de Datos
En nuestro proyecto, la gestión de datos desempeña un papel central y estratégico. Para lograrlo, hemos utilizado Microsoft SQL Server como nuestro sistema de gestión de bases de datos principal. Para las pruebas y pruebas unitarias, hemos aprovechado una base de datos en memoria, lo que nos permite evaluar y validar nuestra lógica de negocio de manera eficaz.

Paquete de Acceso a Datos
Nuestro equipo ha desarrollado un paquete dedicado llamado "Data" para gestionar todas las interacciones con la base de datos. Este enfoque de encapsulación ha resultado fundamental para mantener una estructura organizada y coherente en todo el proyecto. A

través del paquete "Data," hemos establecido una jerarquía de clases de "Management," cada una de las cuales se encarga de interactuar con entidades de datos específicas.

Abstracción Eficiente: La Clase Padre "GenericRepository"
Para promover la eficiencia y la reutilización de código, hemos creado una clase base llamada "GenericRepository." Esta clase contiene métodos comunes para operaciones fundamentales de acceso a datos, como inserción, eliminación, actualización y consulta. Lo que distingue a esta abstracción es su capacidad de ser heredada por cada uno de nuestros "Management." Cada "Management" puede entonces sobrescribir estos métodos base y proporcionar su propia lógica específica.

Esta arquitectura no solo ha simplificado nuestro acceso a la base de datos, sino que también ha permitido una gestión de datos coherente y mantenible en toda la aplicación. Cada "Management" se convierte en el guardián de su respectiva entidad de datos, lo que facilita la gestión, la expansión y la adaptación a medida que evolucionamos en nuestro proyecto.

Manejo de Excepciones

En nuestro proyecto, hemos adoptado un enfoque integral para el manejo de excepciones, garantizando una experiencia de usuario confiable y una comunicación efectiva a través del protocolo HTTP. Este enfoque se divide en dos componentes clave:

1. **ExceptionFilter en el Controlado**r: Hemos implementado un ExceptionFilter a nivel de controlador que actúa como un hábil manejador de excepciones. Su función principal es capturar excepciones y formatearlas de manera que sean comprensibles en el contexto del protocolo HTTP. Esta estrategia nos permite ofrecer respuestas coherentes y significativas a través de las solicitudes web, mejorando la usabilidad y la capacidad de diagnóstico de nuestra aplicación.
2. **Excepciones Personalizadas**: Además de aprovechar las excepciones proporcionadas por el marco .NET, hemos desarrollado nuestras propias excepciones personalizadas. Estas excepciones están diseñadas para abordar casos específicos dentro de nuestra lógica de negocio, como la detección de direcciones de correo electrónico con formato inválido o la restricción de roles no permitidos. La creación de excepciones personalizadas nos permite comunicar de manera precisa y clara los errores específicos que pueden surgir en nuestra aplicación.

Este enfoque combinado de manejo de excepciones no solo mejora la robustez y confiabilidad de nuestra aplicación, sino que también facilita la tarea de depuración y mantenimiento, contribuyendo a la calidad general de nuestro código.

Oportunidades de mejoras

Reconocemos que, en la actualidad, nuestra aplicación no alcanza todos los requisitos detallados en la especificación. Este reconocimiento es un paso importante en nuestra búsqueda de la excelencia. Si bien hemos avanzado con éxito en muchos aspectos, entendemos que siempre hay margen para la mejora.

Nuestro compromiso con la evolución continua de la aplicación se manifiesta claramente en nuestra disposición a aprender y crecer a partir de esta experiencia. Consideramos esta fase inicial como un punto de partida, una base sólida desde la cual podemos construir y expandir nuestras capacidades. Estamos firmemente decididos a abordar las áreas en las que nuestra aplicación aún no cumple completamente con los requisitos.

En la próxima entrega del proyecto, aspiramos a presentar una versión mejorada y más completa que refleje no solo nuestro progreso técnico, sino también nuestro compromiso con la calidad y la excelencia en la ingeniería de sistemas.

Cada desafío es una oportunidad para crecer, y nuestra determinación de hacerlo es el motor que impulsa nuestro camino hacia el éxito.