Universidad ORT Uruguay

Facultad de Ingeniería

DISEÑO DE APLICACIONES 2

OBLIGATORIO 2

Daniel Komés - 225841

DOCENTES:

Ignacio Valle, Marco Fiorito, Matías Salles

Link del repositorio:

https://github.com/IngSoft-DA2-2023-2/225841.git

2023

Declaración de autoría

Yo, Daniel Komés, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba Diseño de Aplicaciones 2;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y que fue contribuido por mí;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Resumen

Documentación acerca de la solución construida para el obligatorio 2 de Diseño de Aplicaciones 2. El sistema es el frontend, backend y base de datos de una aplicación para comercio electrónico de tiendas de ropa centrada en aplicar promociones basadas en los productos de cada compra.

1. Descripción general del trabajo

El trabajo es el frontend, backend, y base de datos de una aplicación web para tiendas de ropa, diseñado para ser adaptable a los constantes cambios de la industria.
El sistema permite a los usuarios ver los productos disponibles, sus detalles y precios, y agregarlos y quitarlos de su carrito de compras, ver su perfil y modificarlo.
Al carrito podrá ser aplicada una de las promociones que haya disponibles, proveyendo un descuento en el precio total de la compra. El usuario podrá crear una cuenta o iniciar sesión con una cuenta existente y realizar la compra de los productos que haya guardado en su carrito.
Los usuarios pueden ser administradores y/o clientes. Los administradores pueden ver la lista de todos los clientes registrados, ver la lista de todas las compras realizadas, y crear, borrar y modificar a otros clientes que no sean administradores, pero no pueden realizar compras.
Los usuarios con el rol de “Cliente” pueden crearse una cuenta, editar sus datos, ver los productos, agregarlos al carrito y realizar compras.

La base de datos tiene pre-guardada una cuenta de administrador (email: “admin@admin.com”), password: “” (vacía), con la que se puede hacer operaciones CRUD sobre usuarios no administradores, y ver todas las compras realizadas.

La base de datos guarda los datos de los usuarios cuando se crean, las sesiones con sus respectivos tokens cuando se inician, las compras con su lista de productos, usuario, promoción (si es que se aplicó alguna), método de pago, total y fecha de la compra, cuando se efectúan, los productos con sus nombres, descripciones, categorías, marcas, colores y precios, y las promociones, con sus nombres y descripciones.

1.1. Errores conocidos y funcionalidades faltantes

- No se aplicó filtro de paginado en las requests de la base de datos ni en la devolución de la API.
- No se usó el código 204-NoContent recomendado por Microsoft en las buenas prácticas de diseño de API.
- No se validó que el Email tenga un formato correcto.

- No se utilizaron Guards en el frontend. Se entiende que mejoran la seguridad al impedir que usuarios sin autorización accedan a páginas que requieren autorización. No se aplicó por falta de tiempo.
- No se utilizaron Services en el frontend. Se entiende que mejoran la mantenibilidad moviendo la lógica de requests a una clase fuera del componente. No se aplicó por falta de tiempo.
- Muchos tipos de datos faltan en el frontend, lo que requirió utilizar “any”. Se entiende que lo ideal sería crear tipos de datos compatibles con los del backend para ayudar a la mantenibilidad.
- No se puede cambiar el rol de un usuario una vez creado, solamente al momento de ser creado por un admin se puede elegir el rol.
- Las promociones de la versión anterior se movieron al paquete a importar mediante Reflection, por lo que se desvincularon las dependencias y se tuvieron que editar las pruebas. Las tres promociones de la primera versión, junto con otra nueva desarrollada para comprobar la funcionalidad, existen todas dentro del mismo proyecto. Se entiende que la funcionalidad requiere desactivar las promociones individualmente, y por lo tanto, cada una debería estar en un proyecto separado.
- WebApi posiblemente contiene referencias a un proyecto de tipo Blazor. Se entiende que lo esperado era un proyecto ASP.NET, pero por falta de tiempo se dejó así. Las funcionalidades no parecen estar afectadas.

2. Diagramas

2.1. Vista lógica

2.1.1. Diagrama de paquetes

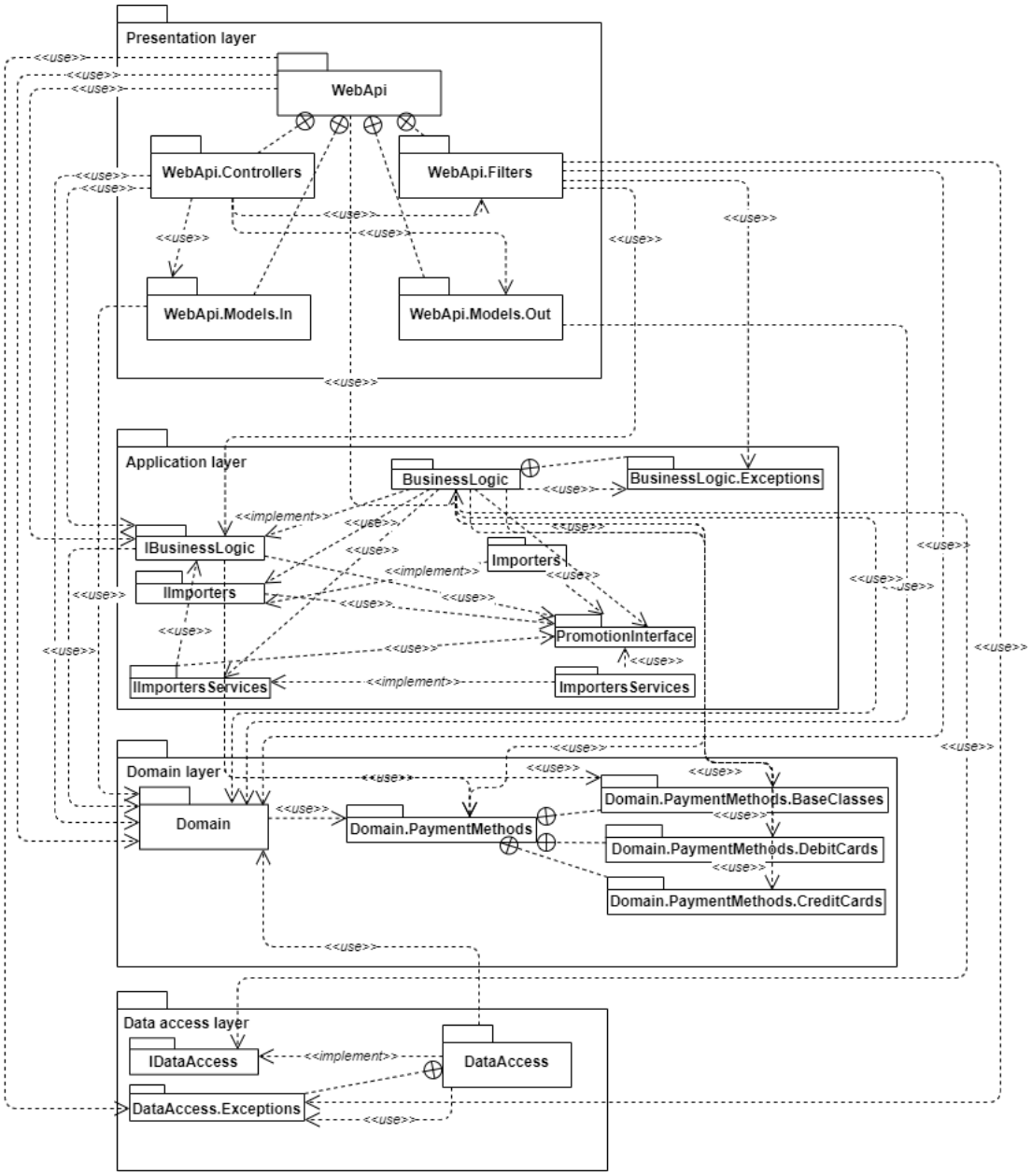


Imagen 1, diagrama de paquetes.

2.1.2. Diagramas de clases

2.1.2.1. WebApi

WebApi es el proyecto de inicio. Contiene además los controladores que reciben las requests de los usuarios y otras clases que los controladores necesitan.

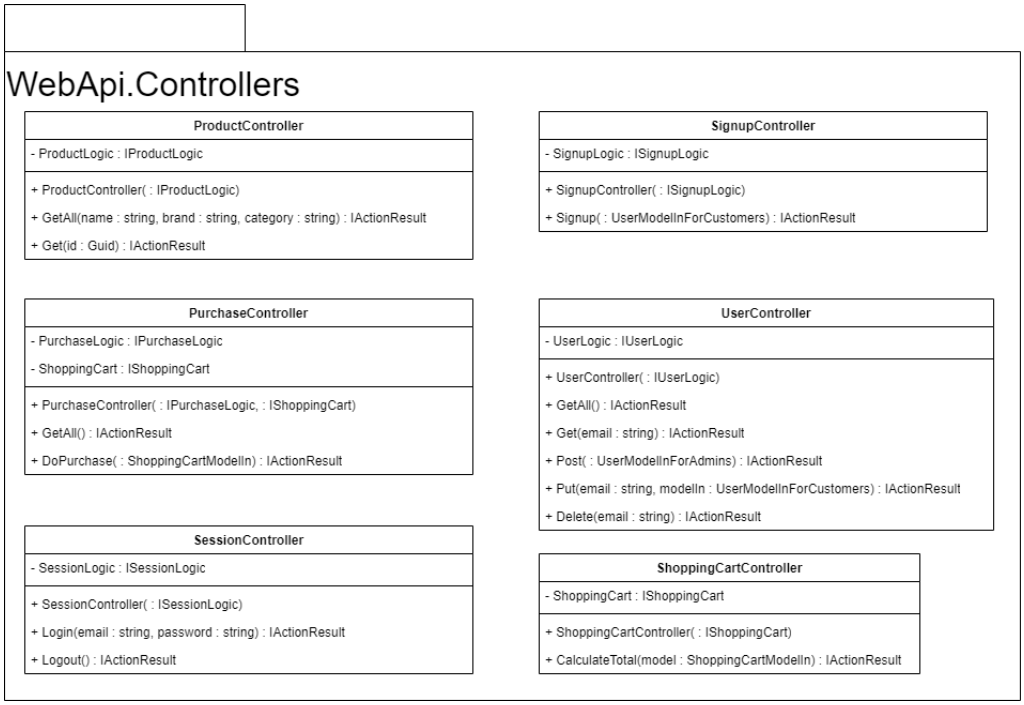


Imagen 2, diagrama de clases de WebApi.Controllers.

WebApi.Controllers contiene los controladores que reciben las requests de los usuarios.

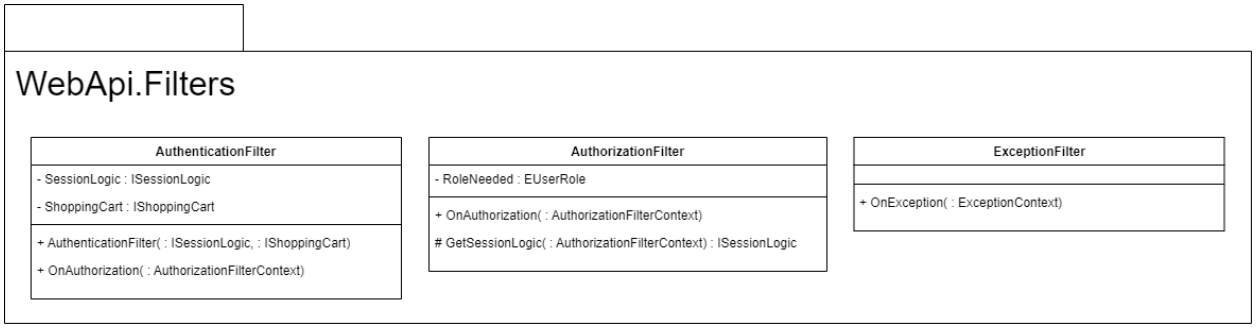


Imagen 3, diagrama de clases de WebApi.Filters.

WebApi.Filters contiene las clases de filtro que se pueden aplicar a los endpoints en los controladores.

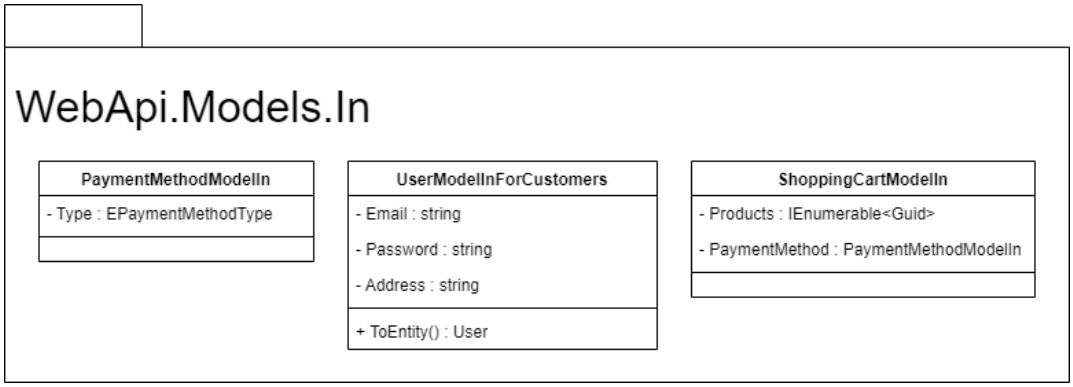


Imagen 4, diagrama de clases de WebApi.Models.In.

WebApi.Models.In contiene las clases que actúan de modelos que los usuarios pueden crear para enviar a los endpoints si corresponde. La existencia de estos modelos da seguridad, ya que ayudan a prevenir posibles ataques de inyección de código al no permitir a los usuarios asignar propiedades potencialmente importantes para el funcionamiento del sistema.

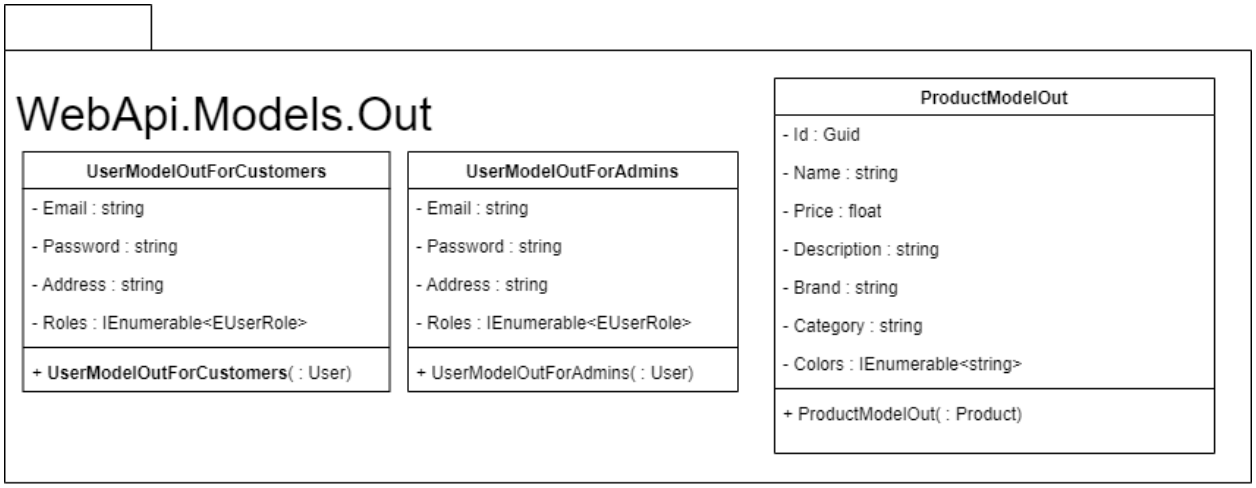


Imagen 5, diagrama de clases de WebApi.Models.Out

WebApi.Models.Out contiene las clases que actúan de modelos que se mostrarán a los usuarios en lugar del objeto original correspondiente en Domain. La existencia de estos modelos da seguridad y privacidad, ya que es posible que algunas propiedades de los objetos no se deban revelar al usuario.

2.1.2.2. IBusinessLogic

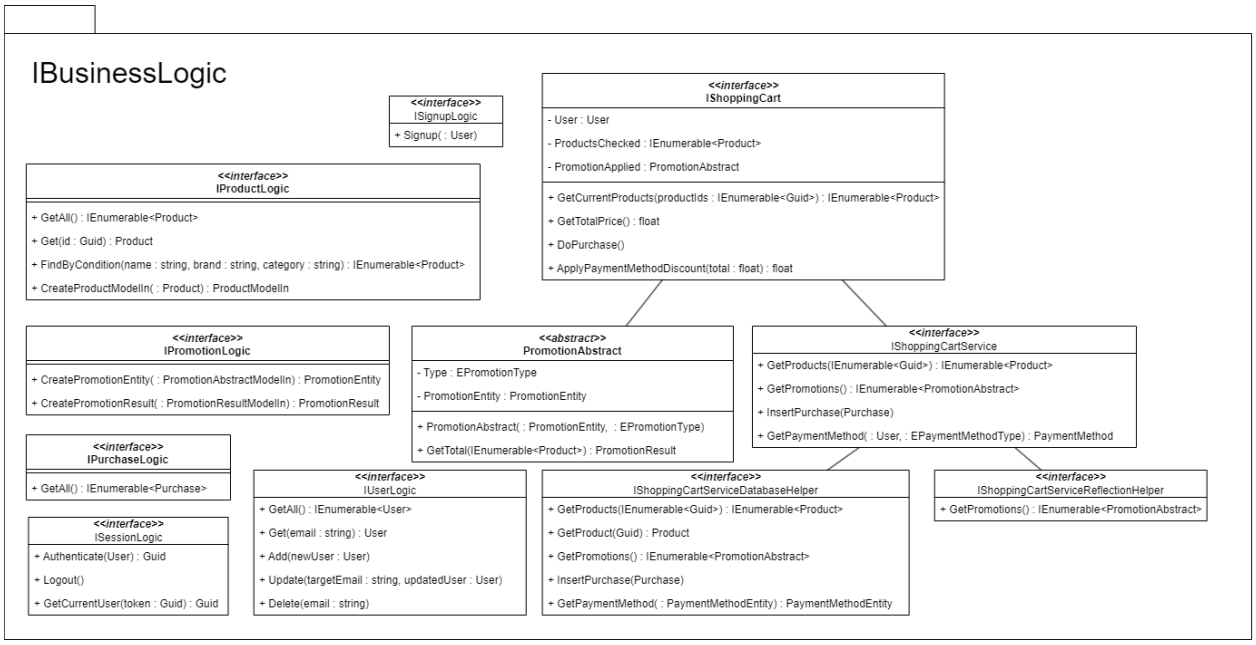


Imagen 6, diagrama de clases de IBusinessLogic.

IBusinessLogic contiene las interfaces (y clase abstracta, en el caso de PromotionAbstract) que son implementadas en la lógica.

2.1.2.3. BusinessLogic

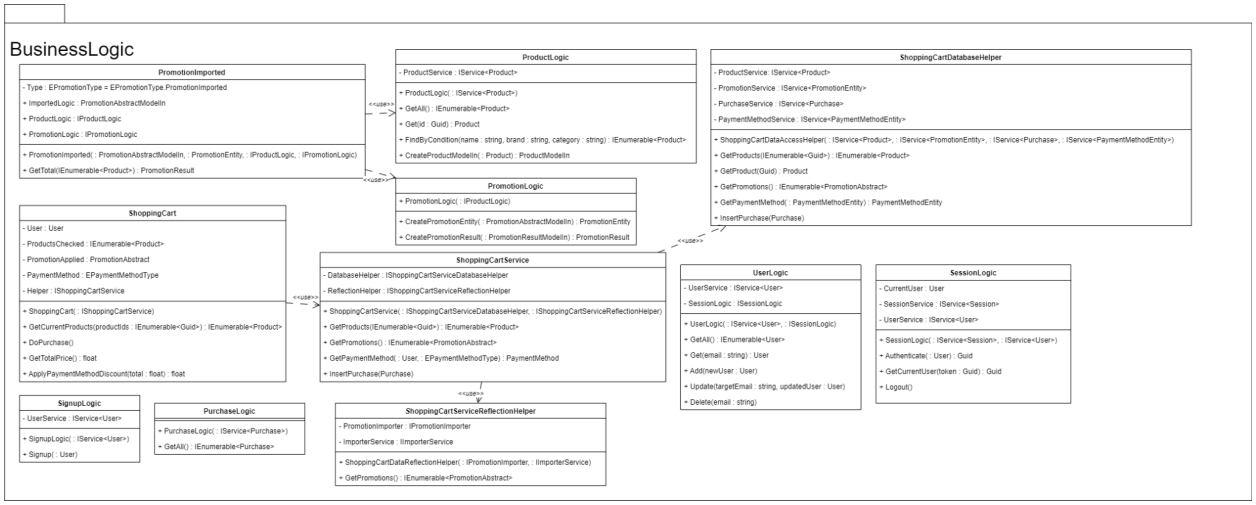


Imagen 7, diagrama de clases de BusinessLogic.

BusinessLogic contiene las clases que implementan las interfaces de IBusinessLogic, y la clase que maneja las promociones importadas, PromotionImported. Este paquete es el que contiene gran parte de la lógica vital para la aplicación.

2.1.2.4. Domain

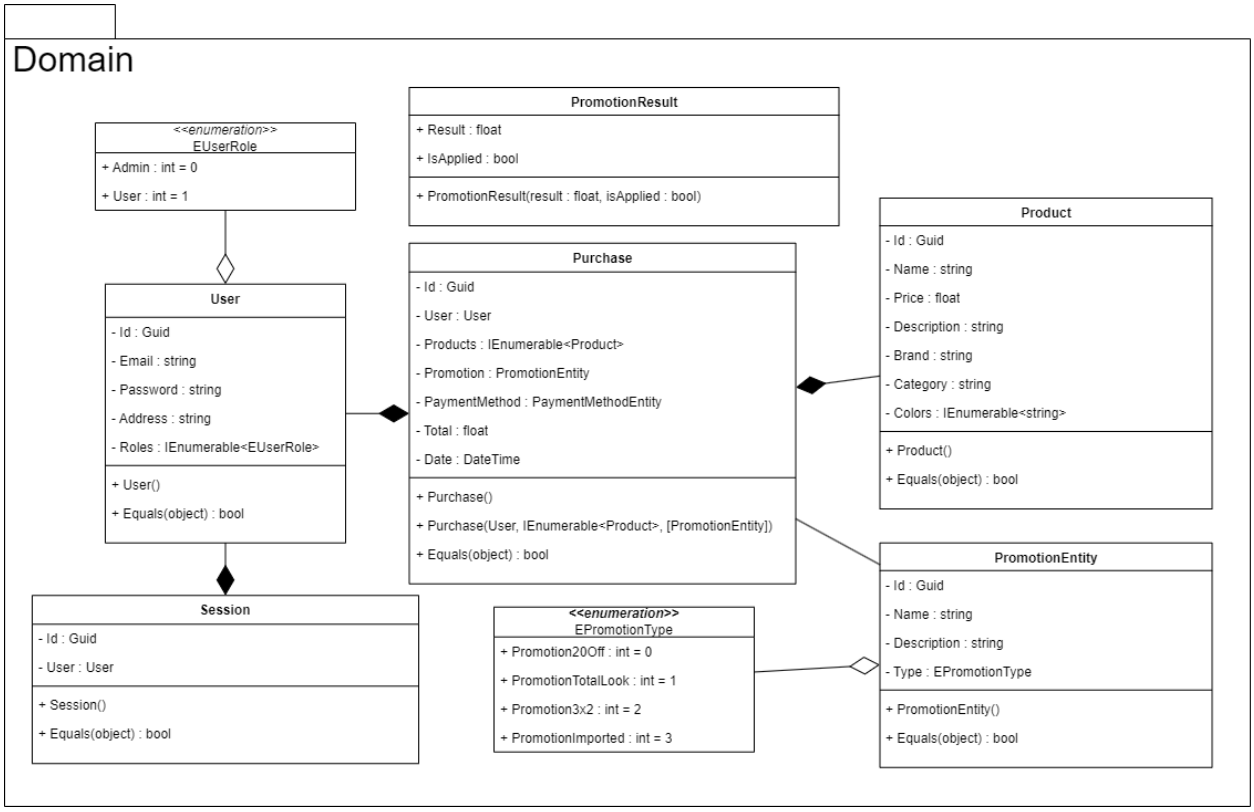


Imagen 8, diagrama de clases de Domain

Domain contiene las clases que representan objetos relevantes para la aplicación. Su utilidad está en sus propiedades, y no tanto en su lógica.

2.1.2.5. IDataAccess

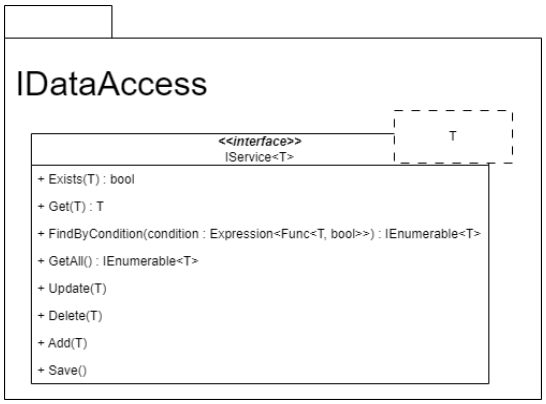


Imagen 9, diagrama de clases de IDataAccess.

IDataAccess contiene interfaces para ser implementadas por clases que interactúan con la base de datos.

2.1.2.6. DataAccess

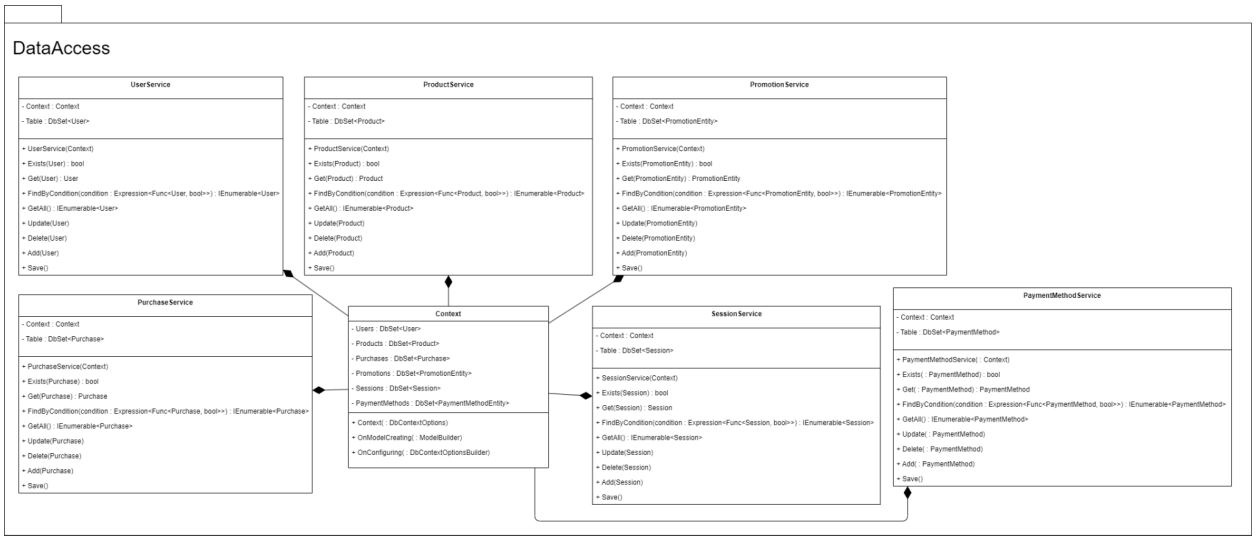


Imagen 10, diagrama de clases de DataAccess.

DataAccess contiene las clases que implementan las interfaces de IDataAccess, y la clase Context que define la estructura de la base de datos. Las clases que implementan IDataAccess son las que interactúan con la base de datos manipulando cada una una clase de objeto de

Domain (UserService manipula solamente Users, ProductService manipula solamente Products, etc.).

2.1.2.7. IImportersServices

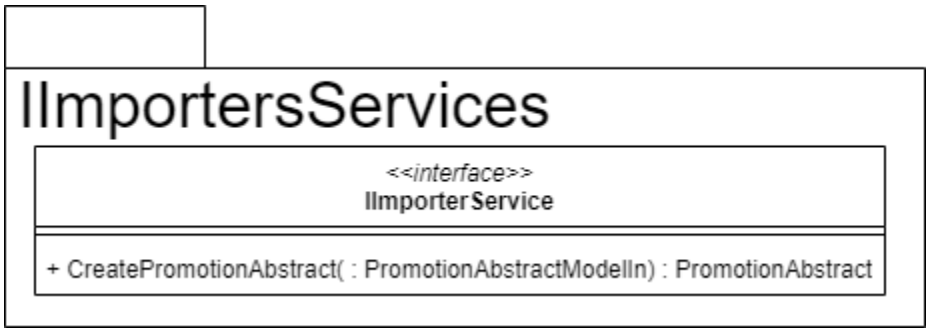


Imagen 11, diagrama de clases de IImportersServices.

IImportersServices contiene las interfaces implementadas por ImportersServices.

2.1.2.8. ImportersServices

Imagen 12, diagrama de clases de ImportersServices.

ImportersServices contiene las clases que implementan IImportersServices. Actualmente sólo contiene una clase, encargada de convertir las promociones importadas con Reflection de models a clases usables por el resto del sistema.

2.1.2.9. IImporters

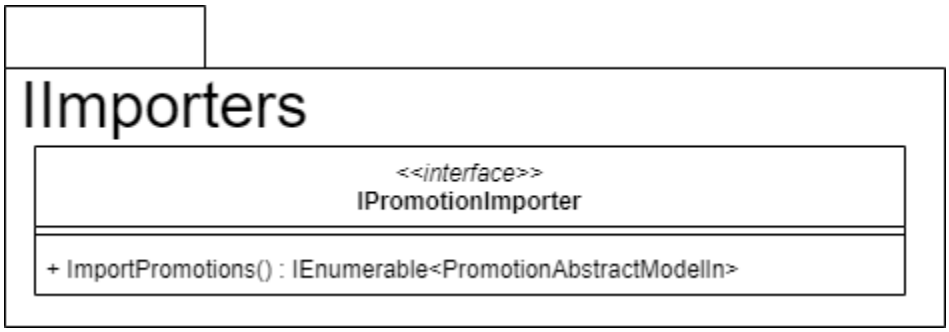


Imagen 13, diagrama de clases de IImporters.

IImporters contiene las interfaces a implementar por Importers relacionadas a la importación de interfaces con Reflection.

2.1.2.10. Importers

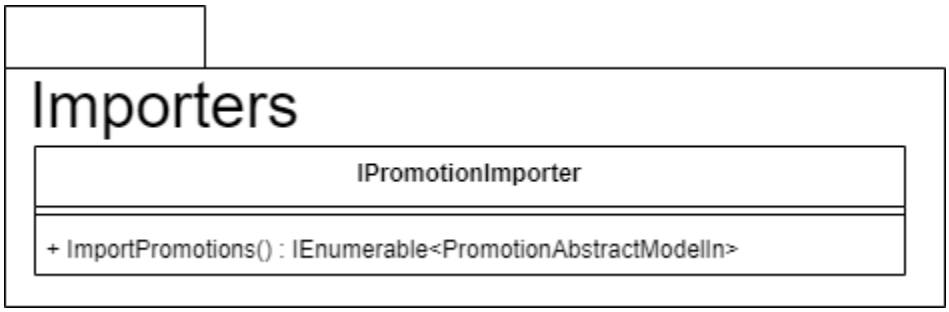


Imagen 14, diagrama de clases de Importers.

Importers contiene las clases con la lógica de importación de promociones desde Reflection. Implementan las interfaces de IImporters.

2.1.2.11. PromotionInterface

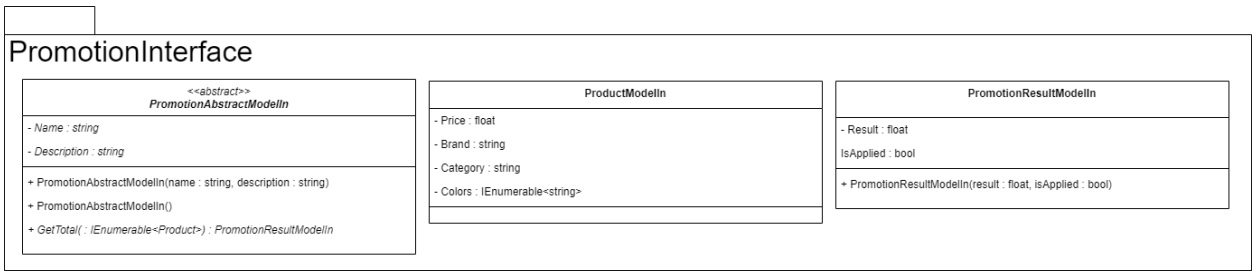


Imagen 15, diagrama de clases de PromotionInterface.

PromotionInterface contiene los modelos necesarios para el desarollo de promociones por terceros. Los modelos representan objetos del dominio, pero al ser modelos, dan mayor privacidad y seguridad al no revelar datos internos del sistema. Estos modelos son convertidos a clases usables por el resto del sistema por ImporterServices.

2.1.3. Diagrama de interacción

Aquí se muestra el diagrama de interacción para cuando el cliente realiza la compra y se logra guardar un registro en la base de datos correctamente.

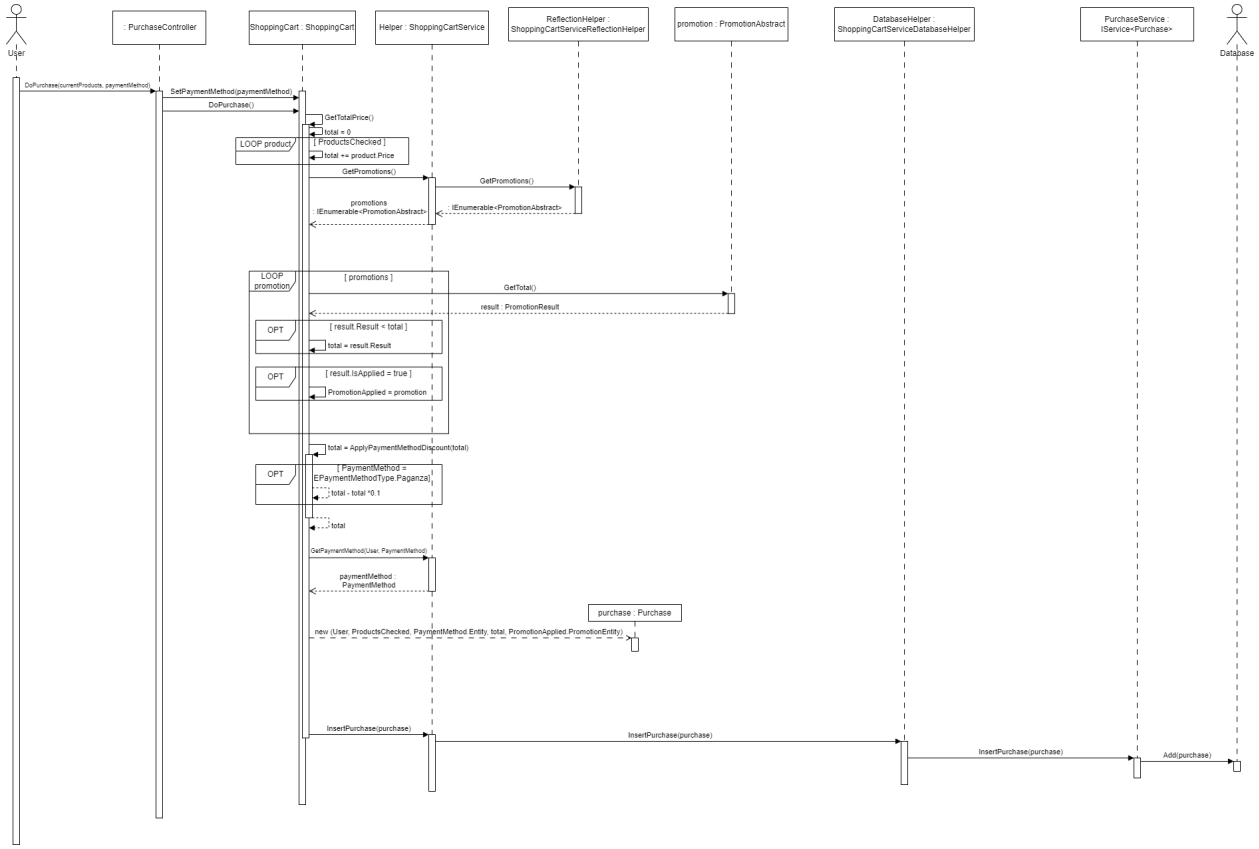


Imagen 16, diagrama de interacción para caso de uso *DoPurchase*.

2.2. Vista de componentes

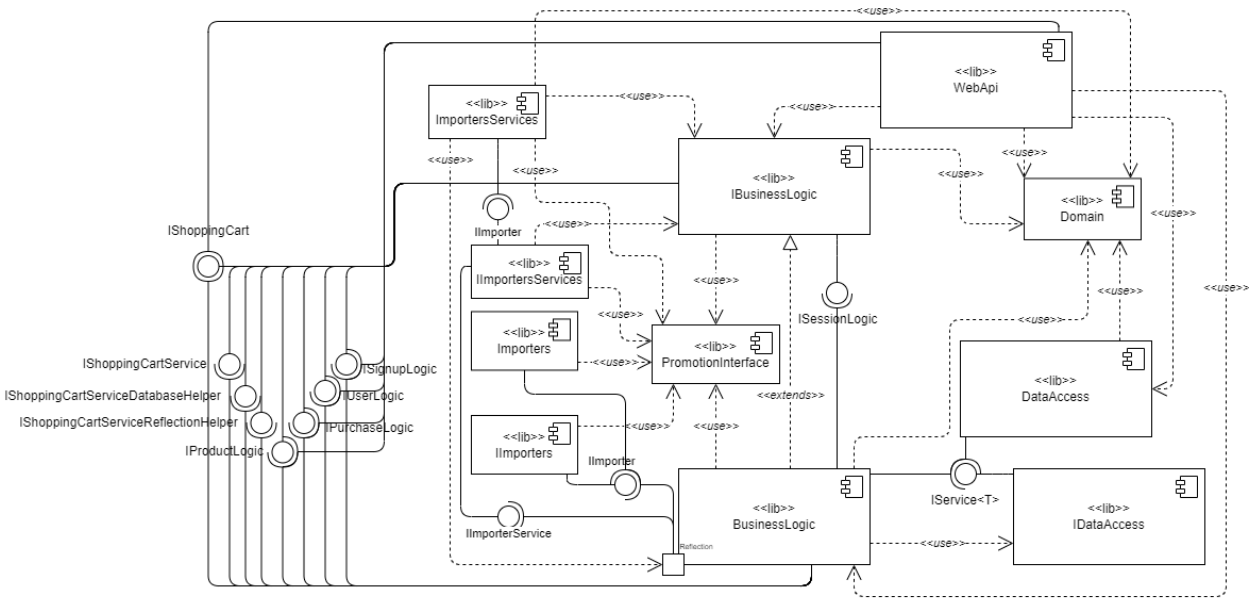


Imagen 17, diagrama de componentes.

La solución se dividió en estos componentes para separar el sistema en capas y así cumplir con las buenas prácticas de programación.

En Domain están las clases que representan objetos dentro del sistema, con las que el resto de los componentes interactúan para cumplir los requerimientos.
IBusinessLogic provee interfaces de la lógica del negocio que BusinessLogic implementa.
IDataAccess provee interfaces de acceso a la base de datos que DataAccess implementa.
WebApi contiene las clases que reciben las consultas de los clientes.
Promotions contiene las promociones programadas en la versión anterior. Se separaron en un paquete aparte para cumplir con CCP.
IImporters contiene las interfaces para importar promociones desde dlls que Importers implementa.
IImportersServices contiene las interfaces que ImportersServices implementa. Estos paquetes tienen lo necesario para convertir las promociones importadas que vienen con un tipo de dato intermedio a tipos de datos usables por el resto de la aplicación.
PromotionInterface contiene las clases necesarias para que un tercero sin acceso a todo el sistema pueda programar una promoción. La promoción resultante será de un tipo intermedio

que debe ser convertido por ImportersServices para poder ser usado por el resto de la aplicación.

Separar la solución de esta forma ayuda a cumplir varios principios de programación, como los patrones SOLID y GRASP.

2.3. Vista de entrega

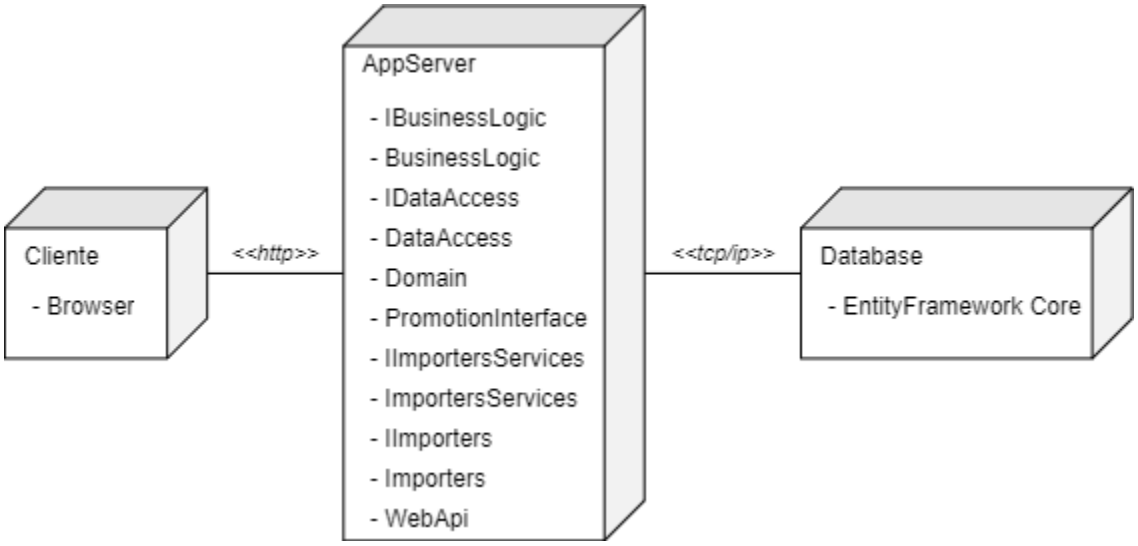


Imagen 18, diagrama de entrega.

2.4. Modelo de tablas de la base de datos

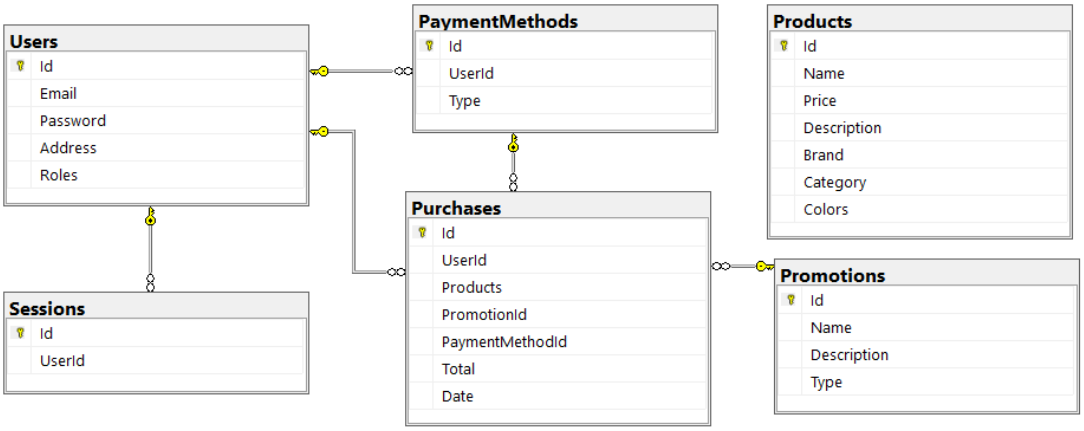


Imagen 19, diagrama de tablas de la base de datos.

3. Justificación del diseño

El principal foco de las decisiones en cuanto a la estructura de clases fue cumplir con los principios Expert, Bajo Acoplamiento y Alta Cohesión de GRASP y SRP de SOLID.

En cuanto al código en general, se siguieron las convenciones de Clean Code relevantes a C#, sobre todo las convenciones de nomenclatura de propiedades y métodos.

3.1. Mecanismos de diseño

- Se usa inyección de dependencias al iniciar la ejecución de la aplicación. Debido a que las clases con lógica implementan interfaces, fue posible aplicar la inyección de dependencias, que a su vez hace cumplir el principio DIP de SOLID.

- La clase Purchase es una clase de asociación que asocia productos con un usuario. Se decidió hacer esto en vez de mantener listas de Product en la clase User y listas de User en la clase Product. Esto permite entender el código más fácilmente y mejora la mantenibilidad. Si hubiera que agregar más información cuando el usuario hace una compra, se pueden fácilmente agregar más atributos en esta clase de asociación sin afectar al resto del código.

- La clase ShoppingCartService existe como fachada para las consultas relevantes que ShoppingCart deba hacer acerca de los usuarios, productos y promociones. A su vez, esta clase usa otras dos, ShoppingCartServiceDatabaseHelper, para interactuar con la base de datos, y ShoppingCartServiceReflectionHelper, para interactuar con las promociones obtenidas por Reflection. Esta separación ayuda a cumplir SRP y la mantenibilidad del código. Si en el futuro las clases Service que hacen las consultas a la base de datos cambian, sólo afectará a una clase. De la misma forma, los cambios en la lógica de Reflection son recibidos por una sola clase y no afecta el resto del código.

- El proyecto PromotionInterface es un proyecto independiente del resto del sistema que contiene los modelos necesarios para programar una promoción por terceros. La nueva promoción deberá heredar de la clase abstracta indicada en ese paquete, y se podrá usar un model que representa un producto para implementar la lógica de la promoción, y un model para devolver el resultado. Todas las clases del paquete son modelos, de

forma que no se revela información interna a terceros. Al momento de importar una promoción tercerizada, la lógica de importación convierte los modelos y la promoción (que implementa un modelo) en clases usables por el resto del sistema. Esta decisión ayuda a mantener la seguridad y privacidad, además de la mantenibilidad del código, al tener modelos extensibles que se puedan modificar si se requiere revelar datos diferentes a terceros.

- Se crearon nuevas clases de lógica para procesar los pedidos de los Controllers. Así, los controllers sólo se encargan de convertir el modelo recibido y pasarlo a la lógica para ser procesado, y al recibir la respuesta de la lógica, convertirla a un modelo para enviar al cliente. Esto aplica SRP y mejora la mantenibilidad del sistema, ya que si hubiera cambios en la lógica, sólo las clases de lógica serían afectadas, y no los Controllers.

3.2. Decisiones de diseño

Para los identificadores únicos de las clases del dominio se usó Guid. Se consideró innecesario verificar si una Id de ese tipo era duplicada o no debido a la naturaleza altamente aleatoria de ese tipo de dato. En el futuro, si se quisiera verificar esto, se haría responsable a las clases más cercanas a la base de datos, es decir las hijas de IService, cada una verificando la Id de su tipo de dato del dominio correspondiente (UserService verificaría las Id de los User, ProductService las de Product, etc.)

Además de cumplir con el requerimiento de que un User tenga un Email único, se agregó una Id de tipo Guid, que es usada internamente por la base de datos. Esto ayuda a que los usuarios puedan cambiar su correo electrónico sin causar problemas en las relaciones de la base de datos. Aún así, se controla que los Emails ingresados por los usuarios sean únicos en el sistema.

Los paquetes relacionados a la importación de promociones con Reflection (IImporters, Importers, IImportersServices y ImportersServices) contienen cada uno una sola entidad, ya sea interfaz o clase concreta. Esto afecta negativamente las métricas evaluadas, pero se decidió hacerlo de esta manera para beneficiar la mantenibilidad al separar las clases por responsabilidades. En el futuro se podrán agregar más clases a estos paquetes cuando se desarrollen nuevas implementaciones de importers.

En las clases con propiedades que no cambiarán durante la ejecución, se marcaron esas propiedades como private y readonly, de forma que sean obligatorias en el constructor y no puedan ser reasignadas. Esta decisión aplica el principio OCP de SOLID.

Se decidió que las clases que heredan PromotionAbstract, en sus métodos GetTotal, donde se requiere devolver varios valores (el total, una verificación de que se aplicó la promoción y el tipo de la promoción) devuelvan una instancia de la clase de resultado PromotionResult, en vez de tuplas, arrays, diccionarios o asignar variables de clase. Esto ayuda en la simplicidad del código y la mantenibilidad, permitiendo fácilmente agregar propiedades al resultado en el futuro. Aplica los principios Alta Cohesión y Bajo Acoplamiento de GRASP y OCP de SOLID.

Se decidió usar 403-Forbidden para casos donde se intenta crear un usuario con el mismo email que otro que ya existe.
Se investigó la posibilidad de usar 409-Conflict o 422-Unprocessable entity, pero se decidió por 403 para mantener la diversidad de status codes baja, tal como se recomendó, además de que se considera que 409 y 422 no son lo suficientemente correctos para este caso. 409 parece hacer referencia a conflictos de versiones de entidades, y 422 hace referencia a semántica potencialmente incorrecta, además de la "incapacidad" del servidor de cumplir la request, y en este caso, el servidor no es "incapaz", sino que directamente se niega por razones de lógica del negocio.
Además, 403 es apropiado para situaciones donde el cliente puede resolver el problema modificando la consulta.
También se menciona [1] que el cliente puede repetir la request con credenciales diferentes, pero que aún así, el error podría no tener que ver con credenciales (en este caso, sería porque el email ya existe). Además, se menciona que los servidores que deseen ocultar los detalles del error (para evitar problemas de privacidad al revelar si el recurso existe o no) se puede cambiar por un 404. En este caso se podría cambiar por un 200-ok (o 201-created) y mostrar la verificación al usuario por email, pero para este obligatorio no se consideró necesario.

Para guardar en la base de datos instancias de objetos con listas de tipos nativos, se decidió explicitar la serialización y deserialización en la clase Context, en vez de crear clases en el proyecto Domain que contengan una Id y una propiedad del tipo nativo deseado. Se decidió esto debido a que no es correcto que el dominio sea afectado por las limitaciones de la base de datos. Además, esto simplifica y facilita mucho la utilización de estas propiedades a lo largo de la aplicación, aumentando la mantenibilidad.

En las clases ProductModelIn y ProductModelOut, una de sus propiedades es una Id de tipo Guid. Esta Id es la misma que la de la base de datos. Se entiende que la existencia de las clases de models In y Out son justamente para evitar revelar datos internos hacia fuera de la aplicación. Se recurrió a esto para que el usuario pueda referenciar sin ambigüedad el producto con el que desea interactuar, ya que el resto de las propiedades de los productos no son únicas. Esto se podría corregir teniendo una segunda Id que sea aceptable mostrar al exterior, similar a lo que ocurre con la clase User, su Id y Email, si fuera necesario en el futuro.

El controller de ShoppingCart debe recibir en el body las Guids de los productos que el cliente haya puesto en el carrito para mostrar el total y las promociones aplicadas. Los productos se guardan en local storage en el frontend cuando el cliente los selecciona, y se envían al backend para verificar su existencia en la base de datos cuando el cliente intenta ver el total del carrito, y cuando quiere realizar la compra. Se decidió esta forma en vez de enviar individualmente al backend cada producto que el cliente seleccione porque se consideró ineficiente por la cantidad de requests que serían enviadas, siendo que el lugar donde se requiere consolidar la información de todos los productos es en el carrito. Los productos se verifican allí cuando el cliente quiera ver el resumen de su carrito, ahorrando así envíos de requests y mejorando la velocidad y potencialmente evitando sobrecargar el sistema.

Cuando un usuario no admin intenta ver o editar el perfil de un usuario que no existe, se devuelve 403-Forbidden. Se decidió ocultar el verdadero código de error (en los casos de 404) para mantener la privacidad, impidiendo revelar si un usuario existe o no.

En varias clases de service, que implementan IService, muchos métodos no se implementaron por no ser necesarios para el sistema. Se entiende que esto es una mala práctica y viola el principio LSP de SOLID. Se decidió mantenerlo así para no complejizar el código.

Se crearon nuevas excepciones personalizadas para lanzar en casos específicos como cuando se intenta crear o modificar un usuario ingresando un correo ya registrado. Esto aumenta la mantenibilidad y permite extender los filtros de excepciones para cubrir más casos y programar comportamiento más específico basado en la excepción atrapada.

Aunque no se logró implementar Guards en el frontend para impedir que usuarios no admins vean contenido para admins, se tuvo cuidado en que el sistema inmediatamente redirija hacia la página principal si las credenciales guardadas no coinciden con las de un admin.

Para esta segunda versión, la mayoría de los endpoints de ShoppingCartController no son necesarios, ya que ahora el frontend guarda los productos del carrito en local storage. Por lo tanto se eliminaron estos endpoints innecesarios, quedando únicamente un endpoint CalculateTotal, con verbo POST, que muestra los datos del carrito.

El endpoint DoPurchase se movió desde ShoppingCartController a PurchaseController, ya que requería tener el verbo POST y causaba conflicto con el endpoint CalculateTotal, que tenía los mismos parámetros y el mismo verbo. Además, por agrupar responsabilidades, tiene más sentido que DoPurchase esté en PurchaseController, ayudando a cumplir el estándar REST.

Se creó un proyecto separado PromotionInterface, totalmente independiente, que es el que será enviado a terceros para poder programar promociones. La nueva promoción deberá heredar de la clase abstracta indicada en ese paquete, y se podrá usar un model que representa un producto para implementar la lógica de la promoción. Todas las clases del paquete son modelos, de forma que no se revela información interna a terceros. Al momento de importar una promoción tercerizada, la lógica de importación convierte los modelos y la promoción (que implementa un modelo) en clases usables por el resto del sistema. Esta decisión ayuda a mantener la seguridad y privacidad, además de la mantenibilidad del código, al tener modelos extensibles que se puedan modificar si se requiere revelar datos diferentes a terceros.

Se movieron las clases de promociones Promotion20Off, PromotionTotalLook y Promotion3x2 a un proyecto separado, que se desvinculó de la solución del sistema, para cumplir el requerimiento de desactivar las promociones en tiempo de ejecución. Esto requirió desvincular las dependencias a BusinessLogic, ya que en la versión anterior, las promociones eran usadas por ShoppingCartDataAccessHelper. Al desvincular las dependencias y el proyecto entero de la solución, varias pruebas unitarias fueron modificadas o eliminadas.

Se realizó una jerarquía de herencia al crear los distintos métodos de pago para mejorar la mantenibilidad cuando se desee desarrollar lógica para cada uno de ellos.

Al realizar una compra, es necesario enviar al backend el método de pago, que ahora mismo sólo está implementado como una opción numérica que se decide en la página del carrito. El sistema revisa si el usuario comprador ya ha realizado compras con ese método de pago, y si es así, utiliza el mismo método de pago que encontró en la base de datos para realizar esta compra. Si el sistema no encuentra en la base de datos un método de pago asociado a este

usuario, que sea del mismo tipo que el que se está intentando usar para esta compra, el sistema crea un nuevo método de pago del tipo seleccionado, lo asocia al usuario y lo guarda en la base de datos. Esto significa que cada usuario puede tener sólo tipo de método de pago, pero se decidió que esta forma beneficia la mantenibilidad cuando en el futuro se desarrolle más lógica e identificadores para los métodos de pago, y la capacidad de que el usuario ingrese los datos de sus métodos de pago mediante el frontend.

3.3. Jerarquías de herencia

Se usaron jerarquías de herencia en clases con código en común. Se observa notablemente en PromotionImported, que tiene la lógica para usar la lógica programada por un tercero en las promociones importadas para calcular el descuento sobre un grupo de productos. Esta clase hereda de una clase abstracta PromotionAbstract, con la firma del método GetTotal, dos propiedades readonly y un constructor. No tiene sentido que las hijas modifiquen esas propiedades, así que se las marcó como readonly en el padre para que no puedan ser modificadas. El constructor recibe estas propiedades por parámetro y es llamado por las hijas. Esta decisión aplica el principio OCP de SOLID.

Se aplicaron también relaciones de implementación donde las clases con lógica implementan interfaces con las firmas de los métodos a implementar. Se puede observar en las clases que interactúan con la base de datos. Existe una interfaz IService<T> con la firma de los métodos que deben ser implementados por las hijas.

Para representar los métodos de pago se usaron varias clases abstractas. PaymentMethod actúa como padre de las demás, y, CreditCard y DebitCard son hijas abstractas de PaymentMethod. Las clases concretas heredan de estas tres clases.

Estas decisiones aumentan la mantenibilidad del sistema ante cambios futuros y aplican el principio Open-Closed de SOLID y permite aplicar inyección de dependencias cumpliendo el principio Dependency Inversion de los patrones SOLID.

3.4. Análisis de métricas

Para el análisis de métricas se usó NDepend. Se obtuvieron los siguientes valores de métricas.

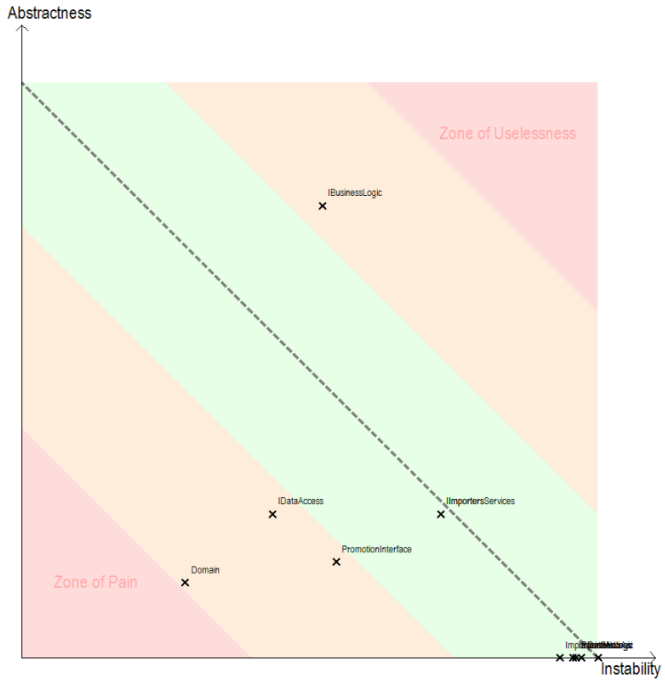


Imagen 20, gráfica Abstracción vs Inestabilidad.

Assemblies	Relational Cohesion	Instability	Abstractness	Distance
Domain v1.0.0.0	2.57	0.28	0.13	0.41
PromotionInterface v1.0.0.0	1.5	0.55	0.17	0.2
IBusinessLogic v1.0.0.0	1.43	0.52	0.79	0.22
IDataAccess v1.0.0.0	1.25	0.43	0.25	0.22
DataAccess v1.0.0.0	2.18	0.97	0	0.02
IImporters v1.0.0.0	1	0.73	0.25	0.02
IImportersServices v1.0.0.0	1	0.73	0.25	0.02
BusinessLogic v1.0.0.0	1.88	0.96	0	0.03
Importers v1.0.0.0	1	0.96	0	0.03
ImportersServices v1.0.0.0	1.25	0.93	0	0.05
WebApi v1.0.0.0	2.65	1	0	0

Imagen 21, tabla resultado de análisis de métricas.

Domain: presenta poca abstracción y poca inestabilidad. Su cohesión de 2.57 está entre los valores aceptables de 1.5 y 4. Es un paquete muy concreto y muy estable, ya que en él están las clases que representan los objetos que usa el sistema. Esto significa que incumple el principio de Abstracciones estables (SAP) por ser estable y concreto, pero ya que el propósito de este paquete es agrupar las clases por sus responsabilidades de representar objetos de la solución, se considera aceptable.
Sus clases no dependen de clases de otros paquetes, por lo que cumple el principio de Dependencias estables (SDP).

PromotionInterface: presenta poca abstracción y estabilidad media. Su cohesión de 1.5 está al límite de lo aceptable. Es un paquete muy concreto con estabilidad intermedia. Sus clases no dependen de ninguna otra ya que es el paquete a enviar a terceros, y a su vez la lógica depende de él para poder realizar las conversiones de modelos a clases usables, por lo que cumple SDP. Además, por ser muy concreto y muy estable, cumple también SAP.

IBusinessLogic: presenta estabilidad media y abstracción considerable. Su cohesión de 1.43 está en el límite de lo aceptable. Es un paquete de estabilidad intermedia y muy abstracto, ya que contiene interfaces y una clase abstracta que son implementadas por otro paquete. Sus clases dependen de IDataAcces, cuya estabilidad es mayor, por lo que cumple SDP. Su cumplimiento de SAP es dudoso, potencialmente por su dependencia a DataAccess.Exceptions, que es un paquete menos estable dentro de DataAccess. Se podría mejorar esto separando DataAccess.Exceptions a otro paquete fuera de DataAccess de forma que IBusinessLogic no dependa de un paquete menos estable.

IDataAccess: presenta estabilidad media y poca abstracción. Su cohesión de 1.25 está por debajo del valor mínimo aceptable 1.5. Esto significa que sus clases tienen poca relación entre ellas. Ya que el paquete contiene una sola interfaz que es implementada por las clases que acceden a la base de datos, se puede llegar a entender este valor. Aún así, una posible solución sería convertir esta clase a múltiples interfaces que sean implementadas individualmente en DataAccess, lo que además ayudaría a corregir el incumplimiento de LSP de las clases de DataAccess.
Según el análisis es un paquete de estabilidad media y considerablemente concreto, y con cohesión baja, lo que no coincide con lo esperado, ya que no contiene clases concretas y no depende de otros paquetes. Esto podría deberse a la baja cantidad de clases e interfaces que tiene.

DataAccess: presenta muy alta inestabilidad y ninguna abstracción. Su cohesión de 2.18 está dentro de los valores aceptables. Es un paquete muy inestable y muy concreto, ya que contiene las clases que interactúan con la base de datos y las clases de la lógica dependen de ellas.
Este paquete cumple SDP, ya que depende solamente de Domain, y SAP, ya que es tan inestable como abstracto.

IImporters: presenta alta inestabilidad y baja abstracción. Su cohesión de 1 está por debajo del valor mínimo aceptable de 1.5, que posiblemente se debe a que contiene una sola interfaz. Según el análisis, es un paquete muy inestable y concreto. Contiene la interfaz implementada por el paquete Importers para importar promociones mediante Reflection. La discrepancia entre el análisis indicando que es un paquete concreto y las entidades contenidas realmente, podría deberse a su baja cantidad de entidades (1 interfaz), pero ya que el propósito de este paquete es separar la responsabilidad de interfaces de importadores, se considera inevitable hasta que haya necesidad de más interfaces para importadores.
A pesar de que su cohesión está por debajo de lo aceptable, sus otras métricas lo dejan muy cerca de la secuencia principal. Se decidió separar esta interfaz en un paquete dedicado específicamente a ella para cumplir la separación de responsabilidades, además de que ayuda a la mantenibilidad en caso de que se agreguen más interfaces para clases importadoras de promociones.
Este paquete cumple SDP, ya que depende de PromotionInterface, un paquete más estable. Según el análisis, cumple con SAP por ser inestable y concreto.

IImportersServices: sus valores son iguales que IImporters. Este paquete contiene la interfaz implementada en ImportersServices, necesaria para convertir los modelos de promociones importadas a objetos usables por el sistema.

BusinessLogic: presenta muy alta inestabilidad y ninguna abstracción. Su cohesión relacional de 1.88 está dentro de lo aceptable. Es un paquete muy inestable y concreto ya que contiene las implementaciones de las interfaces de IBusinessLogic.
Este paquete cumple con SDP ya que depende de un paquete más estable y SAP por ser tan inestable como concreto.

Importers: presenta muy alta inestabilidad y ninguna abstracción. Su cohesión de 1 está por debajo del mínimo aceptable 1.5. Esto se debe a la baja cantidad de entidades que contiene (1 clase concreta). Se considera inevitable hasta que haya más implementaciones de

importadores de promociones.
Este paquete cumple con SDP ya que depende de IImporters, que tiene una estabilidad mayor, y cumple con SAP por ser tan inestable como concreto.

ImportersServices: presenta muy alta inestabilidad y ninguna abstracción. Su cohesión de 1.25 está por debajo del mínimo aceptable 1.5. Esto se debe a la baja cantidad de entidades que contiene (1 clase concreta). Se considera inevitable hasta que haya más servicios a importadores de promociones.
Este paquete cumple con SDP ya que depende de IImportersServices, que tiene una estabilidad mayor, y cumple con SAP por ser tan inestable como concreto.

WebApi: presenta total inestabilidad y ninguna abstracción. Su cohesión de 2.65 está dentro de los valores aceptables. Es un paquete muy inestable y concreto ya que contiene los controllers que reciben las requests del cliente.
Cumple SDP ya que depende de IBusinessLogic, que tiene una estabilidad mayor, y cumple con SAP por ser tan inestable como concreto.

A lo largo de la solución se cumplió con el principio de dependencias acíclicas (ADP).

3.5. Resumen de mejoras del diseño

Se agregaron clases de lógica en BusinessLogic (con sus correspondientes interfaces en IBusinessLogic) para manejar lógica que en la primera versión era manejada por los controllers. Esto aplica SRP mejorando la mantenibilidad al mover la lógica a clases dedicadas y permitiendo que los controllers se centren en convertir modelos de entrada a clases del sistema y clases del sistema a modelos de salida.

Se separaron los modelos de usuarios UserModelIn y UserModelOut a 4 clases: UserModelInForAdmins, UserModelInForCustomers, UserModelOutForAdmins, y UserModelOutForCustomers. Se decidió esto para separar aún más los modelos a mostrar y recibir dependiendo de si el usuario es un admin o no.

Se crearon excepciones para casos particulares (por ejemplo, ProfileMismatchException, para cuando un usuario no admin intenta acceder a los datos de otro usuario), en vez de usar una misma excepción con distintos parámetros o mensajes. Esto ayuda a la mantenibilidad ya que permite que el ExceptionFilter decida mejor cómo manejar las excepciones recibidas.

ShoppingCartDataAccessHelper pasó a ser ShoppingCartService, y se crearon otras dos clases: ShoppingCartServiceDatabaseHelper, con las responsabilidades de la antigua ShoppingCartDataAccessHelper, y ShoppingCartServiceReflectionHelper, encargada con la nueva responsabilidad de obtener las promociones importadas con Reflection. Este cambio aplica SRP, mejorando la mantenibilidad al separar responsabilidades entre clases.

3.6. Acceso a datos

Los datos se persisten en una base de datos que fue creada usando EntityFramework Core con el enfoque Code First.

Existe una clase Context que hereda de DbContext, en la que se especifican las relaciones de las tablas con las clases del dominio, y la forma de serializar y deserializar las propiedades de esas clases.

Las consultas se realizan únicamente desde clases concretas que implementan una interfaz IService<T> con los métodos comunes a ellas. La interfaz usa un template que las hijas reemplazan con el tipo de la clase del dominio con la que van a interactuar. Por ejemplo: UserService implementa IService<User>, y los métodos en la interfaz están definidos de forma que interactúan de alguna forma con ese template. Así, public IEnumerable<T> GetAll() se convertirá en public IEnumerable<User> GetAll() cuando sea implementada por UserService.

3.7. Manejo de excepciones

Las excepciones se manejan mediante una clase ExceptionFilter que se aplica a los Controllers (las clases que reciben los comandos de los usuarios). De esta forma, se evita el uso de los try-catch a lo largo del código, y se permite que las excepciones suban hasta que los filtros las capturan y devuelven una respuesta HTTP y un mensaje apropiados según la excepción que fue capturada.

4. Bibliografía

[1] RFC 9110, “HTTP Semantics”. [Online]. Available: https://httpwg.org/specs/rfc9110.html#status.403. Accessed on Oct. 3, 2023.

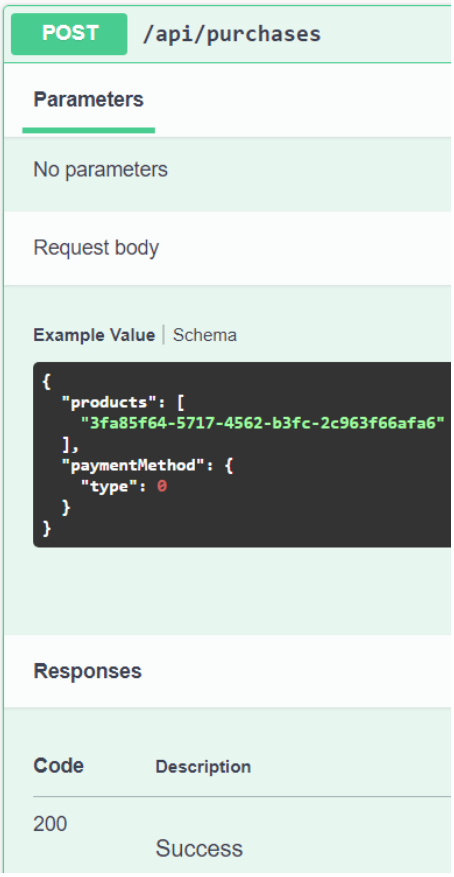
[2] Google: “Web API Design: The Missing Link”, [Online]. Available: https://cloud.google.com/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf. Accessed on: Oct 3, 2023.

[3] Robert C. Martin: “Clean Code, A handbook of agile software craftsmanship”, [Online]. Available: https://github.com/jnguyen095/clean-code/blob/master/Clean.Code.A.Handbook.of.Agile.Software.Craftsmanship.pdf. Accessed on: Nov 16, 2023.

5. Anexo

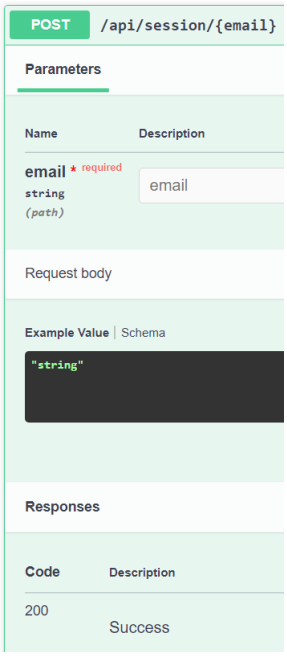
5.1. Cambios en la API

5.1.1. Purchases



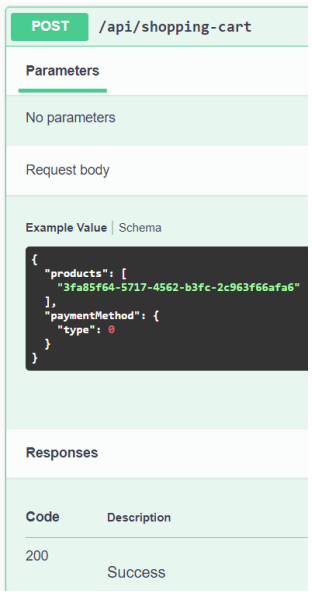
[POST] api/purchases se usa para confirmar la compra.
Anteriormente estaba en api/shopping-cart.

5.1.2. Session



[POST] api/session se usa para iniciar sesión. Ahora requiere la contraseña como string en el body. Antes no requería body.

5.1.3. Shopping cart



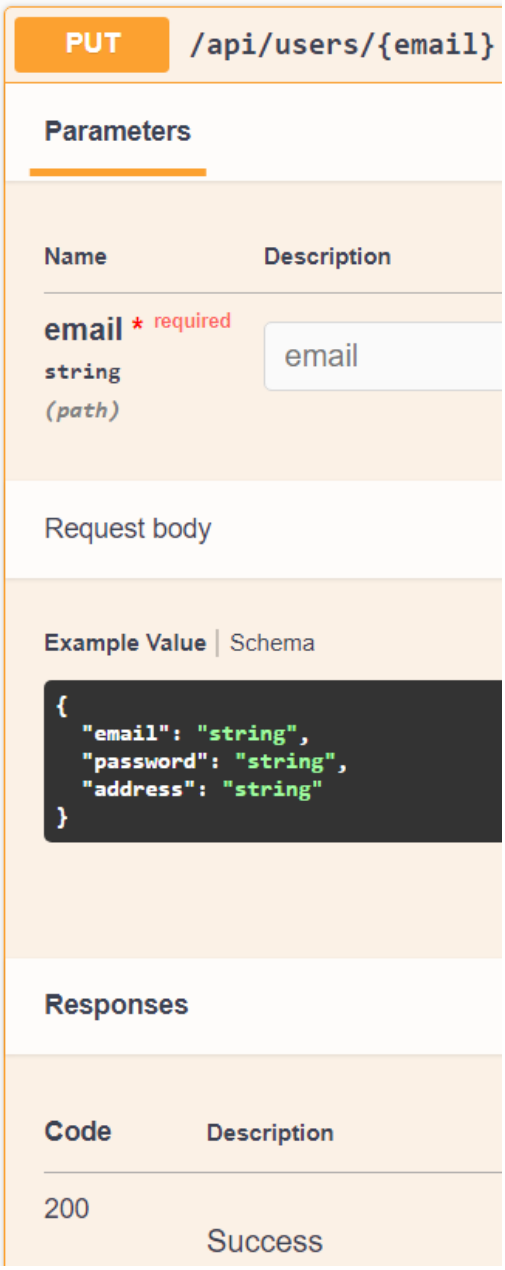
[POST] api/shopping-cart se usa para mostrar el carrito. Calcula el precio total y las promociones aplicadas.
Ahora el body requiere un JSON que incluye el paymentMethod.

5.1.4. Signup

[POST] api/signup se usa para que un usuario se registre en el sistema. Ahora el body requiere el atributo password.

5.1.5. Users

[POST] api/users es usado cuando un admin ingresa a un usuario al sistema. Ahora el body requiere password y lista de roles.



[PUT] api/users es para cuando se quiere editar los datos de un usuario, ya sea por el mismo usuario o por un admin. Ahora el body requiere el campo password.

5.2. Cobertura del código

Hierarchy	Covered (%Lines)	Not Covered (%Lines)
user_D_2023-11-16.03_25_35.coverage	60.38%	39.28%
ibusinesslogic.dll	100.00%	0.00%
domain.dll	78.18%	19.39%
businesslogic.dll	96.57%	2.00%
{ } BusinessLogic	96.51%	2.03%
ProductLogic	97.30%	0.00%
PromotionImported	100.00%	0.00%
PromotionLogic	100.00%	0.00%
PurchaseLogic	100.00%	0.00%
SessionLogic	97.56%	0.00%
ShoppingCart	86.00%	10.00%
ShoppingCartService	94.55%	3.64%
ShoppingCartServiceDatabaseHelper	100.00%	0.00%
ShoppingCartServiceReflectionHelper	100.00%	0.00%
SignupLogic	100.00%	0.00%
UserLogic	100.00%	0.00%
{ } BusinessLogic.Exceptions	100.00%	0.00%
webapi.test.dll	100.00%	0.00%
dataaccess.dll	0.00%	100.00%
webapi.dll	47.40%	52.38%
{ } WebApi	0.00%	100.00%
{ } WebApi.Pages	0.00%	100.00%
{ } WebApi.Models.Out	83.33%	16.67%
{ } WebApi.Models.In	92.00%	8.00%
{ } WebApi.Filters	0.00%	100.00%
{ } WebApi.Controllers	99.22%	0.00%
ProductController	100.00%	0.00%
PurchaseController	100.00%	0.00%
SessionController	100.00%	0.00%
ShoppingCartController	96.00%	0.00%
SignupController	100.00%	0.00%
UserController	100.00%	0.00%
{ } WebApi.Pages.Shared	0.00%	100.00%
businesslogic.test.dll	97.68%	2.05%
promotioninterface.dll	73.68%	26.32%

El principal enfoque de las pruebas fueron las clases de BusinessLogic y WebApi.Controllers, ya que contienen la mayoría de la lógica vital para la aplicación. En la imagen se observa que la cobertura de BusinessLogic es de 96.57% y la de WebApi.Controllers es de 99.22%. Ambos valores están dentro del margen aceptable de 90%-100%.
Las clases de Domain y PromotionInterface tienen su utilidad en sus atributos, y no tanto en su lógica, por lo que no se enfocaron esfuerzos en testear su lógica.