Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Diseño de Aplicaciones 2

Obligatorio 2

Integrantes:

Garia Sotto (191065)
Fabio Ramirez (218566)
Martin Vidal (68694)

Link a nuestro repositorio GIT:

https://github.com/IngSoft-DA2-2023-2/191065-218566-68694

Grupo N6A - Noviembre 2023

DECLARACIÓN DE AUTORÍA

Nosotros, Garia Sotto, Fabio Ramirez y Martin Vidal, declaramos que el trabajo que se presenta en esta obra es de autoría de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos la materia diseño de aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué no fue aportado por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Introducción

El problema planteado consiste en desarrollar un sitio de comercio electrónico de venta de ropa. Con la particularidad que deben ser fácilmente implementables distintas estrategias promocionales que acompañen la dinámica del sector.

Es así, que el sitio debe permitir, que los usuarios agreguen productos al carrito y en base a los productos agregados y las promociones habilitadas, el sistema calcula el precio final para posteriormente permitir que se realice la compra

Debe existir una gestión de productos, promociones, usuarios y compras; con funcionalidades específicas indicadas en la letra pero cuyo diseño considere la posibilidad de que el sistema sea extensible.

Para cumplir con esto, se creó una API desarrollada en .NET SDK 6.0 / ASP.NET Core 6.0 usando el lenguaje de programación C# y las librerías de Entity Framework Core. Los endpoints de la API fueron probados con la herramienta POSTMAN en la primera entrega. Por su parte, la segunda entrega, consiste en un frontend para acceder al backend de la primera entrega, el cual fue desarrollado en Angular. Asimismo la persistencia de los datos se implementó sobre Microsoft SQL Server Express 2017. Los IDEs utilizados fueron Rider, MS Visual Studio Community 2022 y Visual Studio Code v1.84.2.

En cuanto a la documentación, los diagramas que se muestran en este documento fueron realizados con las herramientas Draw.io y StarUML.

Para la recolección de métricas se utilizó NDepend v2023.2.1

A efectos que quede documentado el alcance del sistema, indicar que se implementaron todos los requisitos incluidos tanto en el primer como en el segundo obligatorio. Igualmente se deja constancia de problemas con el manejo de colores de los productos. En este sentido, indicar que en ciertas condiciones, los colores de los productos no se devuelven apropiadamente. Asimismo, mencionar que la promoción “TotalLook” que refiere al descuento por compra de tres productos del mismo color, no se aplica correctamente. En cualquier caso, la misma ha sido implementada.

Diseño

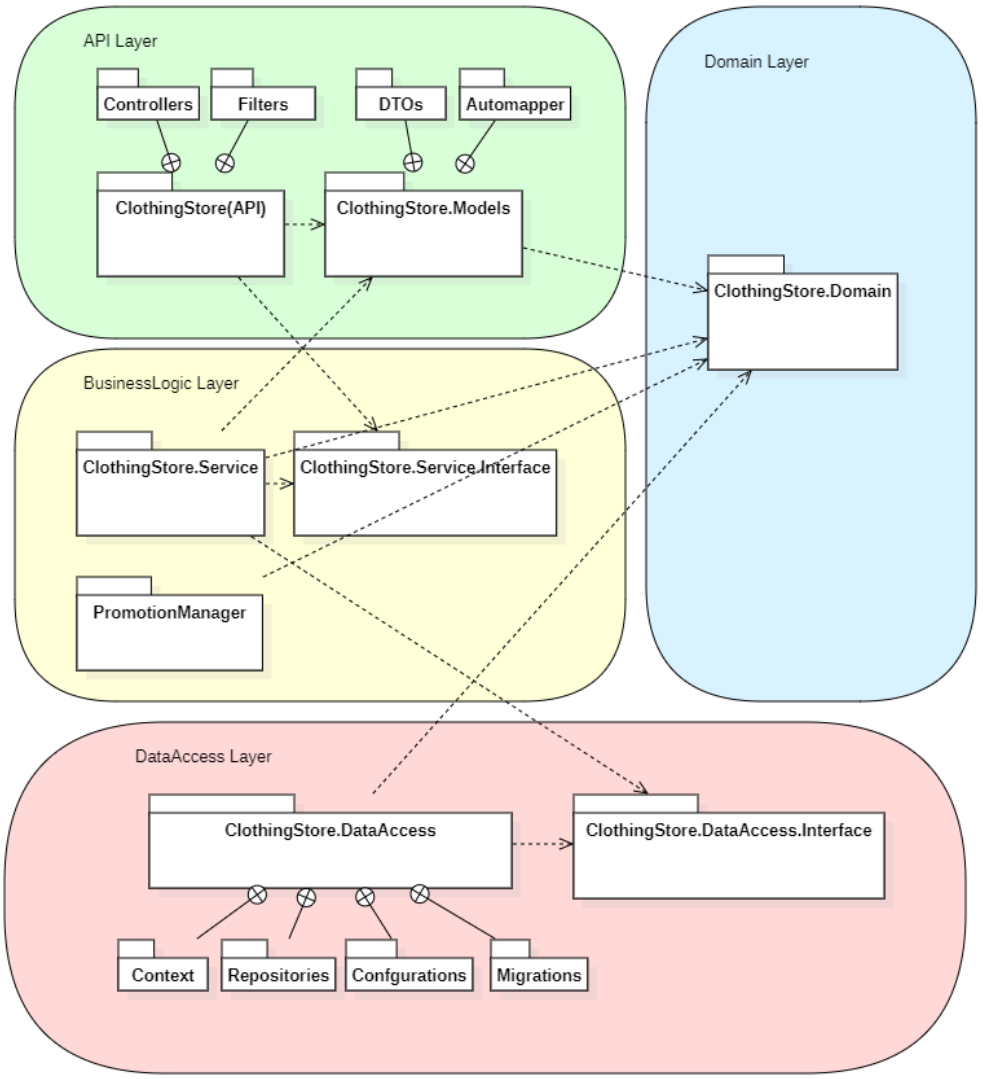
Tal como se establece en la letra del obligatorio, se intenta que el documento esté organizado siguiendo el modelo 4+1. El modelo consta de cuatro vistas más una quinta vista que se relaciona con las cuatro anteriores. Tiene como objetivo brindar perspectivas complementarias sobre la arquitectura de un sistema.

Este documento hace especial hincapié en las vistas lógica y de implementación.

Vista Lógica

Esta vista hace referencia a la representación de las clases y objetos del sistema, ofreciendo una perspectiva estructural del sistema. Se centra en los aspectos funcionales y muestra cómo las clases y los objetos interactúan.

Diagrama de paquetes



Los paquetes se organizan en capas. La intención de desarrollar por capas, tiene varias ventajas, particularmente la posibilidad de mantener relativamente desacoplados los paquetes (consecuentemente las clases) para evitar que modificaciones en un lugar impacten en el resto del código.

ClothingStore (WebApi)

En el paquete llamado ClothingStore se encuentran las configuraciones clave de la Web API. Aquí se definen los métodos que especifican los endpoints, esto es, los puntos de acceso a los recursos de la API. También aloja los controladores relacionados con las entidades y las URIs que los usuarios utilizarán para consultar mediante peticiones (requests). Además, en este paquete se incluyen los filtros, como por ejemplo, un filtro de excepciones que permite manejar de manera específica ciertos tipos de excepciones, aplicando criterios de filtrado para una gestión precisa. Este filtro no solo mejora la gestión de excepciones, sino que también tiene la ventaja de eliminar la necesidad de usar bloques 'try-catch' en los controladores, simplificando así el manejo de errores.

ClothingStore.Models

El paquete que contiene los modelos DTOs se denomina ClothingStore.Models. Los DTOs, o Data Transfer Objects, tienen como objetivo principal crear objetos planos que contengan una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación. Esto optimiza la forma en que la información viaja entre el cliente y el servidor. Se han implementado modelos DTOs para cada una de las entidades en nuestro sistema. Para cada entidad, se creó un modelo de entrada y otro de salida (request y response), diseñados específicamente para mostrar los datos necesarios en cada caso. Además de los modelos DTOs, este paquete también contiene la clase 'AutoMapperConfig', encargada de gestionar las configuraciones de mapeo entre los modelos DTOs y sus respectivas entidades en el dominio. Esta clase es importante para garantizar la correcta traducción y transferencia de datos entre diferentes partes del sistema.

ClothingStore.Service

En el paquete ClothingStore.Services se implementan todos los métodos necesarios para cumplir con las funcionalidades principales de la API. Las clases dentro de este paquete son las implementaciones concretas de interfaces definidas en ClothingStore.Service.Interface. Aquí es donde encontramos la lógica específica para cada funcionalidad, las cuales interactúan con los datos y aplican las reglas de negocio según lo definido en la interfaz de servicios.

ClothingStore.Service.Interface

Una interfaz actúa como un contrato que debe ser cumplido por el paquete ClothingStore.Service. Las otras capas de la API necesitarán acceder a las funcionalidades de la lógica de negocio a través del uso de estas interfaces que se han definido aquí. Tal como se mencionó anteriormente, las clases en el paquete ClothingStore.Service implementan precisamente estas interfaces, estableciendo así una conexión entre la lógica de negocio y las demás partes de la aplicación.

ClothingStore.DataAccess

Este paquete es responsable de la gestión del acceso a datos en la API. Aquí es donde se implementan los repositorios para las clases y entidades, utilizando diversas herramientas para la

lectura y escritura de los datos. Estos datos son cargados o utilizados por otros paquetes que dependen de ClothingStore.DataAccess. Además, en este paquete se encuentran las “Migrations”. Una “migration” proporciona una forma de modificar de manera incremental el esquema de una base de datos para mantenerlo sincronizado con el modelo del sistema, conservando la información ya existente en la base de datos. Este enfoque permite que la API sea extensible, ya que permitiría realizar cambios en el modelo base, sin tener que realizar modificaciones manuales, complicadas, en la base de datos.

ClothingStore.DataAccess.Interface

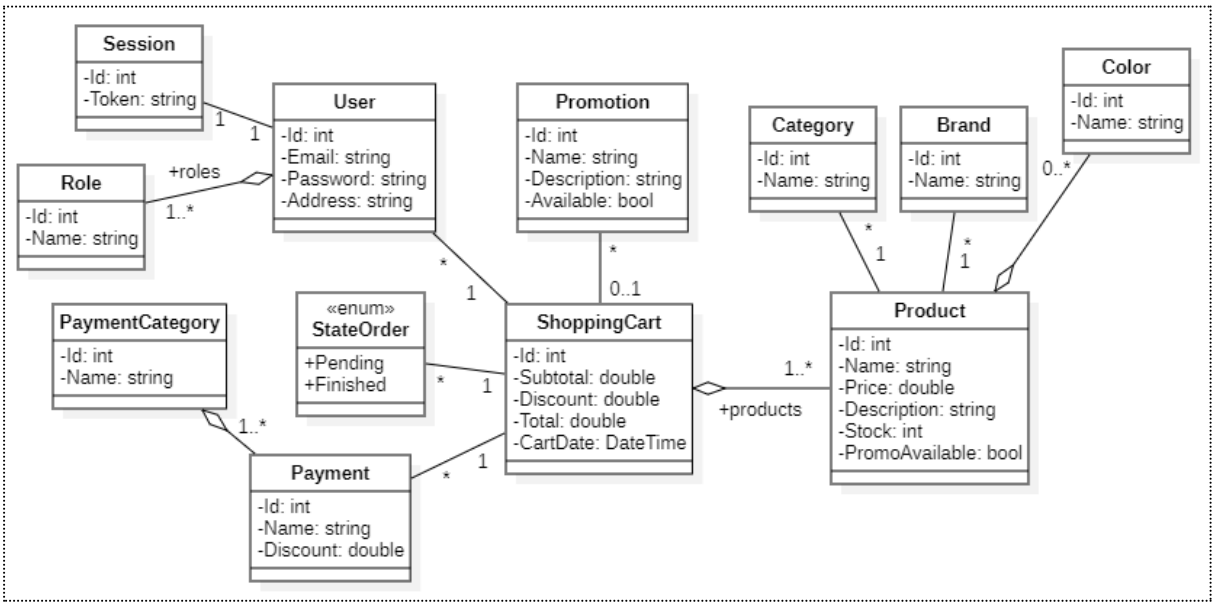
Así como las interfaces de acceso a datos, este paquete también contiene los contratos que deben ser cumplidos por el paquete ClothingStore.DataAccess. Cualquier otra capa que necesite acceder a las funcionalidades de acceso a datos deberá hacerlo mediante el uso de estas interfaces que se han definido en este paquete. Estas interfaces actúan como un puente entre las funcionalidades de acceso a datos y las otras partes de la aplicación, estableciendo una forma estandarizada de interacción y asegurando la consistencia en la comunicación entre capas.

ClothingStore.Domain

La función principal de este paquete es la gestión de las entidades de negocio. Para definir estas entidades de negocio, se efectuó un análisis de los requerimientos proporcionados y utilizamos esta información como base para diseñar un diagrama que describe las clases que componen el modelo de la API. En la próxima sección, presentaremos en detalle el proceso de creación de este diagrama de clases, destacando cómo se han modelado y organizado las clases para representar de manera efectiva las entidades y relaciones clave del sistema.

<u>Diagrama de clases</u>

En esta sección se comentan aquellos requerimientos relevantes a la hora de tomar decisiones acerca del diseño de las clases y sus relaciones.



Clases del dominio

Por razones de claridad para explicar los conceptos y las decisiones de diseño tomadas, se consideró apropiado mostrar el diagrama de clases en dos partes. Seguramente no es lo habitual o incluso lo más ajustado a la metodología, pero se entendió conveniente mostrar por un lado las relaciones entre las clases del dominio y por otro favorecer el entendimiento del sistema mostrando el esquema de desarrollo en capas.

1. Promociones:

Las promociones no tienen definido un actor. Es decir, ninguno de los usuarios se explicita que tenga derechos para realizar el ABM de las mismas. Por otro lado, se describieron las promociones para tanto para la primera como para la segunda entrega y no parece posible parametrizar el descuento teniendo en cuenta la enorme cantidad de posibilidades de describir las reglas propias de cada promoción. Este es un elemento que cambió en la manera como se procesan de una entrega a la otra. Para la primera entrega, se optó por guardar una referencia a cada promoción en la base de datos (id, nombre y descripción) con el objetivo de poder guardar la promoción aplicada en cada venta, vinculándola con un registro en la base. En este caso, las reglas propias de cada promoción y como se aplicaban a cada carro de compras, estaban hardcodeadas en la clase ShoppingCart. En cualquier caso, lo relevante es que las promociones se corren directamente cuando se calcula el total de una lista de productos en un carrito de compras (y/o cuando se efectiviza la compra).

Las reglas definidas en el código verifican las siguientes promociones relativas a la primera entrega:

- Promoción 20% OFF:
    - Condición en el carrito: Tener al menos 2 productos cualesquiera en el carrito.
    - Acción: Aplicar un descuento del 20% en el producto de mayor valor en el carrito.
- Promoción Total look:
    - Condición en el carrito: Tener al menos 3 productos del mismo color en el carrito.
    - Acción: Aplicar un descuento del 50% en el producto de mayor valor en el carrito.
- Promoción 3x2:
    - Condición en el carrito: Tener al menos 3 productos de la misma categoría en el carrito.
    - Acción: Hacer que el producto de menor valor en el carrito sea gratuito.
- Promoción 3x1 Fidelidad:
    - Condición en el carrito:** Tener al menos 3 productos de la misma marca en el carrito.
    - Acción: Hacer que los dos productos de menor valor en el carrito sean gratuitos.

La segunda entrega mantuvo las promociones solicitadas en la primera, pero en lugar de correrlas desde el código compilado; el nuevo requerimiento establece que las mismas puedan ser cargadas (o eliminadas) en tiempo de ejecución mediante reflection. Este cambio es muy significativo; por lo cual se creyó apropiado describir el mecanismo mencionado en una sección independiente, más adelante.

2. Productos:

- Cada producto debe tener nombre, precio, descripción, marca, categoría y colores.
- Se debe poder ver un listado de todos los productos y filtrarlos por texto, marca o categoría.

Para resolver estos requerimientos se creó una clase Product que permite guardar los datos solicitados en el primer ítem, además en la lógica de negocio se dispone de una función que permite obtener una lista de productos filtrada por nombre, marca y categoría. La segunda entrega demanda dos requerimientos adicionales; por un lado poder seleccionar si un producto determinado es o no elegible para ser considerado al aplicar las promociones; y por otro llevar un control de stock disponible al agregar cierto producto a un carrito o a la hora de cerrar la compra. En este sentido indicar que los requerimientos adicionales tuvieron un impacto bajo en las clases y en los métodos afectados.

3. Carrito de Compras:
El concepto más relevante en el sistema es sin dudas el carrito de compras. En este sentido, indicar que esta entidad funciona tanto como carrito de compras propiamente dicho, así como orden de compra. El estado de la orden determina si es un carrito de compras (estado “pending”) o una orden concluida, es decir, una venta (estado “finished”). En este último caso, representa una instancia no modificable. Cuando se procede con la compra, el carrito desaparece como una versión accesible y la órden se guarda en base.

Detalles sobre el carrito y la orden de compra:

- Los usuarios pueden agregar productos al carrito.
- Si el carrito tiene productos, cualquiera de ellos pueden ser eliminados.
- El subtotal del carrito hace referencia a la suma de los importes de todos los productos agregados considerando el importe del campo “Price” de la base en la tabla “Product”.
- El sistema calcula el precio total en base a los productos agregados y las promociones aplicables.
- Solo se aplica una promoción con el mayor valor de descuento al carrito.
- En caso de no aplicarse promoción alguna, el total y el subtotal tienen el mismo valor y se guarda la compra con el identificador de la promoción “Sin Promo”, en el campo “PromotionId”.
- Al realizar una compra, se registra el usuario comprador, los productos comprados, la promoción aplicada y la fecha de la compra.
- Al realizar una compra el carrito pasa a estar en un estado cerrado (StateOrder “finished”) y no accesible. Se crea un nuevo carrito si se desea volver a comprar.
- La segunda entrega, prevé la implementación del medio de pago como parte del proceso de compra.

Para resolver este caso, se creó una clase ShoppingCart que permite guardar los productos que el usuario va a comprar, también se dispone de una función que calcula el mejor descuento a aplicar y el monto total a pagar. Indicar que al igual que la inclusión del manejo de stock; en cuanto a la implementación del medio de pago, el nuevo requerimiento tuvo bajo impacto sobre el código ya desarrollado.

5. Usuarios:

- Los usuarios pueden loguearse y desloguearse.
- Se tiene un rol “Client” y “Administrator” y una dirección de entrega.
- Podrán visualizar su historial de compras con sus detalles.

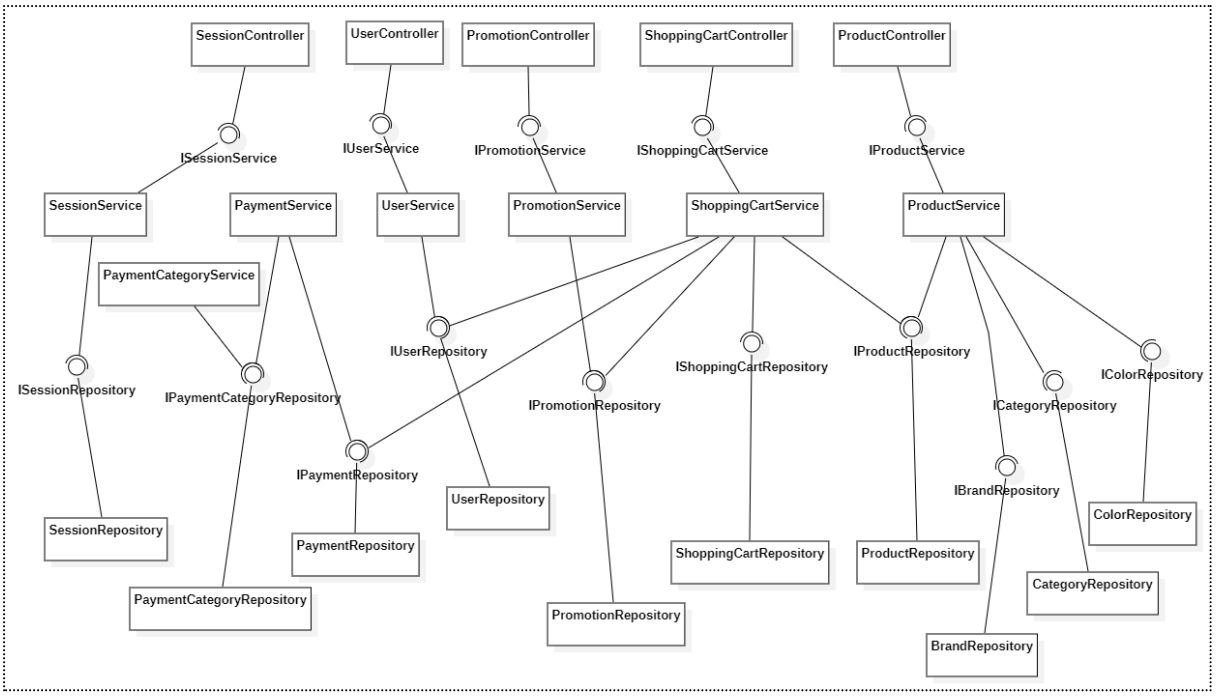
Se creó una clase “Session” que permite realizar el login y logout del usuario, cuando el usuario ingresa su email y contraseña en el login, le retorna un token de autenticación. El mismo es persistido en la base de datos. Al momento del usuario desloguearse se elimina ese token de la base. Para poder visualizar sus compras existe una función en la clase ShoppingCart que devuelve el historial de compras a partir del id del usuario.

6. Administración:

- Los usuarios administradores pueden ver, crear, modificar y eliminar usuarios en el sistema.
- Los usuarios administradores pueden ver todas las compras realizadas.
- Los usuarios administradores pueden gestionar productos; crear, modificar y eliminarlos.

En este caso se aplica una un filtro de autenticación mediante un token que permite validar si el usuario es administrador y puede realizar la operación solicitada, ya fuese gestión de usuarios, de productos o reportes de ventas. Indicar además, que la gestión de productos involucra todo lo referente al manejo de stock requerido para la segunda entrega. En este sentido, se aprovechan las operaciones de modificación de productos para hacer lo propio con el stock.

Como se mencionó anteriormente, este segundo diagrama muestra cómo las clases se disponen en capas. Dichas capas se comunican entre interfaces manteniendo así desacopladas las clases.



Asimismo, el diagrama evidencia cierto grado de sobrecarga en la clase ShoppingCartService, lo que hace suponer en principio, que dicha clase tienen demasiadas responsabilidades. Oportunamente se había indicado que el concepto “ShoppingCart” era el centro de la aplicación. Como tal, la mayoría de las funciones lo tienen como referencia. En cualquier caso, parece haber una oportunidad para mejorar el código refactoreando no solo esa clase, sino todo el esquema relacionado al “ShoppingCart” en las distintas capas del sistema.

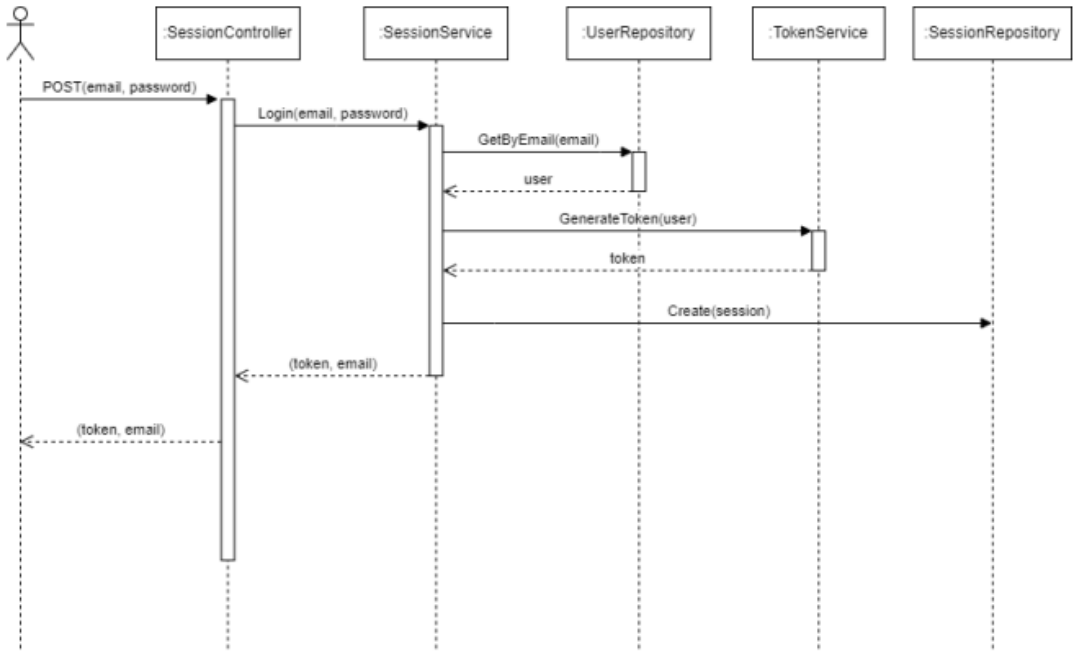
Independientemente de los elementos particulares del sistema, la intención en todo momento fue intentar seguir las mejores prácticas de diseño tratadas durante la asignatura. En especial, el uso de interfaces y el uso de inyección de dependencias.
El uso de interfaces a modo de contrato, en las cuales se define qué se debe hacer pero no como implementarlo. Esto permite desacoplar una implementación particular de las clases que la usan. De esta manera, se puede lograr la comunicación entre la capa de acceso a datos y la lógica del negocio, y de esta última con API Layer; a la vez que se oculta la implementación de las funcionalidades que provee.

Por su parte, el uso de inyección de dependencias es en el contexto en el cual el sistema presenta múltiples relaciones entre clases. Esto significa un fuerte acoplamiento, producto de que ciertas clases tenían la responsabilidad de la instanciación de sus dependencias. Al aplicar el patrón las dependencias de un objeto se proporcionan desde afuera en lugar de ser creadas internamente. Como beneficio inmediato de este desacoplamiento está el hecho que el sistema pasa a ser más mantenible al tiempo que favorece la extensibilidad. Además presenta otras ventajas relacionadas, por ejemplo, al reuso, ya que cada componente se encuentra más independiente y por tanto es más fácil de llevar hacia otras partes del sistema o incluso a otro desarrollo. Asimismo, esta independencia hace más fácil el testing debido a que es más sencillo proporcionar dependencias simuladas durante las pruebas unitarias.

Diagramas de secuencia

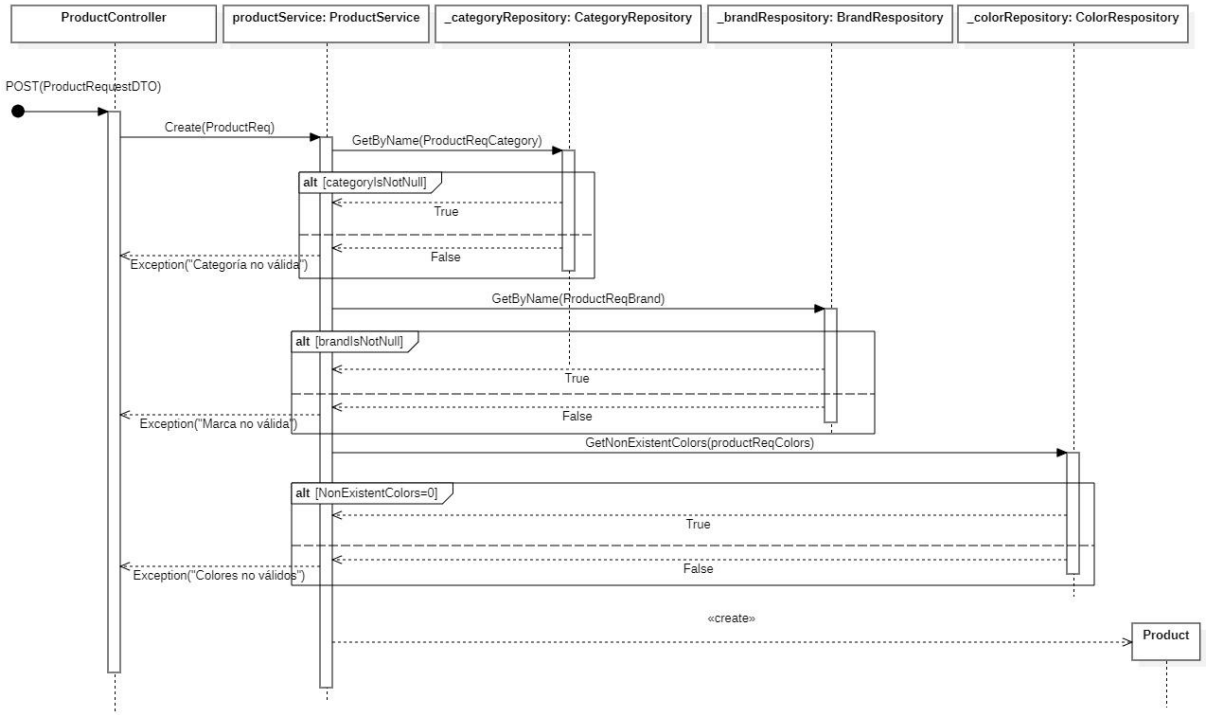
Ciertas funcionalidades ameritan ser analizadas especialmente, ya sea por su relevancia en el sistema o por la complejidad que demanda su implementación.

En el diagrama se muestra como es el proceso de login de usuario. A partir de esto se obtiene el token que da soporte al mecanismo de autenticación, necesario todas las funcionalidades administrativas del sistema y para las relacionadas a la compra de productos.



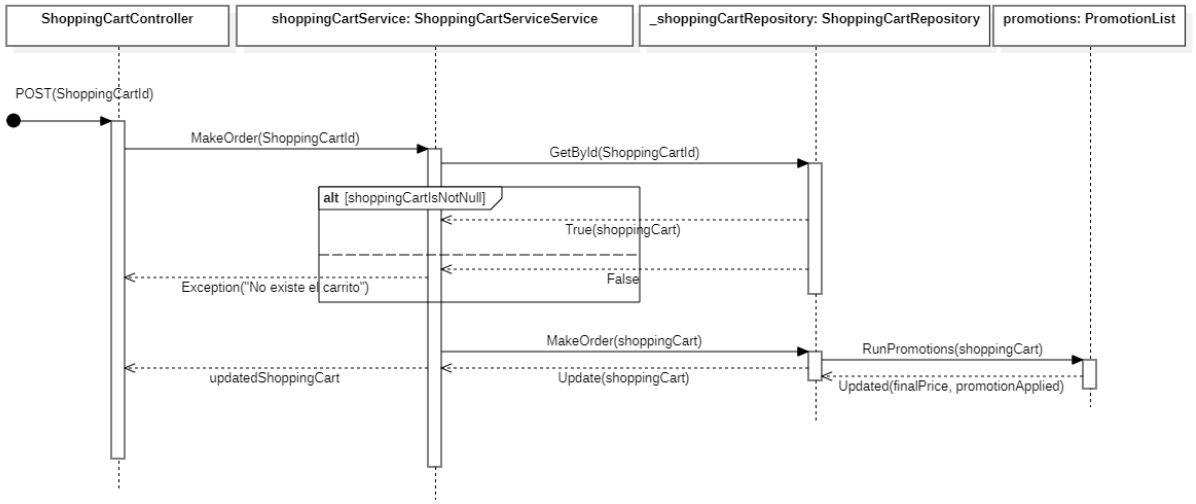
Como resultado de esto, queda determinado que puede o no puede hacer un usuario en el sistema.

A continuación se observa la secuencia para la creación de un producto. Particularmente, se hace hincapié en los controles referentes a la categoría, marca y colores a asignar. Es de hacer notar que estas propiedades ya se encuentran precargadas en la base, razón por la cual, se hacen esas verificaciones para mantener la consistencia de los datos. En la primera entrega, se controlaba específicamente que los valores de estas propiedades ya estuvieran ingresados para evitar que el usuario afectara la consistencia de la base de datos. Al disponer de un frontend en la entrega final, estas opciones se presentan a modo de lista para que el usuario administrador seleccione. Es decir, los controles ya no son requeridos, pero igualmente se optó por dejarlos, tanto en el sistema, como en el diagrama.



El tercer diagrama que se presenta es el que muestra cómo se ejecutan las promociones sobre un shopping cart. Se pasa por parámetro el id del shopping cart hasta la clase que tiene el método y la lista de promociones vigentes. De acuerdo a las reglas, se aplicará una promoción, estos datos vuelven para actualizar la información sobre el carrito de compras.

Es de hacer notar, que la lista “promotions” representa las promociones activas a ser corridas. El diagrama no especifica cómo se carga esa lista. En la primera entrega se corría desde código hardcodeado en la aplicación, mientras que para la segunda entrega, la lista representa las promociones cargadas por reflection.



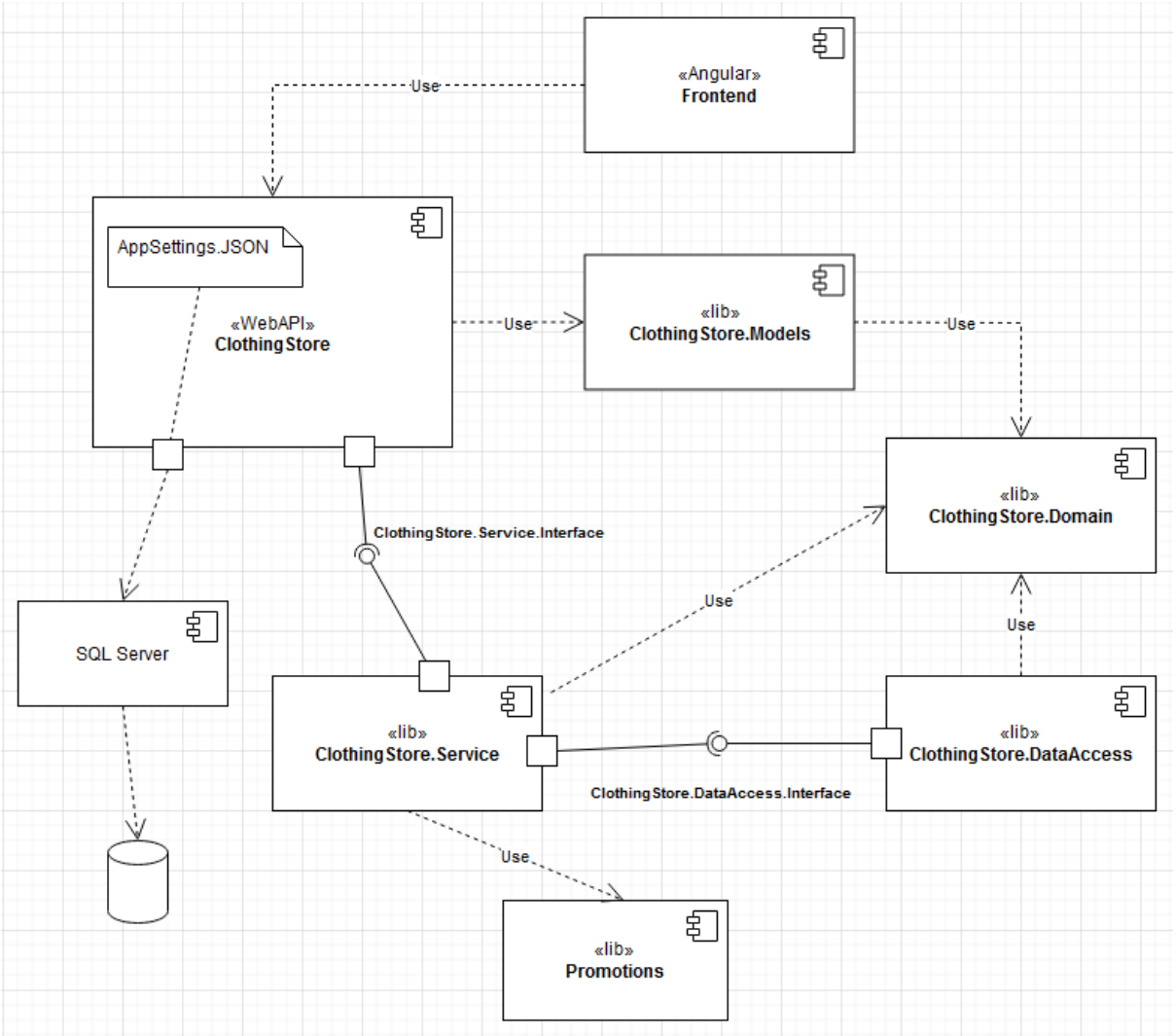
Como se había mencionado, el shopping cart funciona como carrito de compras y como orden de compra, determinado por el estado de la orden. El diagrama muestra la secuencia que incluye la finalización de la orden, pero las promociones se pueden correr para saber el costo total, aun cuando no se realice la compra. A efectos del diagrama, lo único que cambia es que en el "updateShoppingCart" se modifica o no el estado de la orden.

Vista de implementación

Esta vista permite visualizar mejor la estructura física del sistema, mostrando cómo los elementos lógicos se distribuyen.

Diagrama de componentes

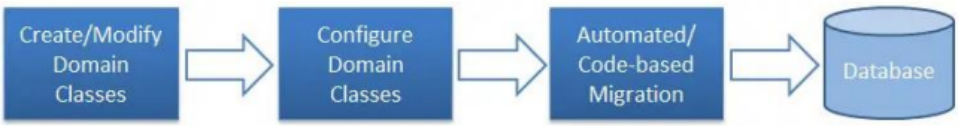
A diferencia de otros diagramas UML que describen la funcionalidad de un sistema, los diagramas de componentes se utilizan para modelar la estructura y organización de los componentes de un sistema.



Siguiendo el diseño en capas que se mostró en el diagrama de paquetes, el diagrama de componentes muestra las dependencias entre estos y evidencia las relaciones mediante interfaces. Si bien los conceptos se repiten de diagrama a diagrama, hay sutiles diferencias que ofrecen distintas perspectivas del sistema. En cualquier caso, lo importante sigue siendo la manera en cómo se reflejan en los diagramas los principios de diseño fundamentales que se intentaron aplicar.

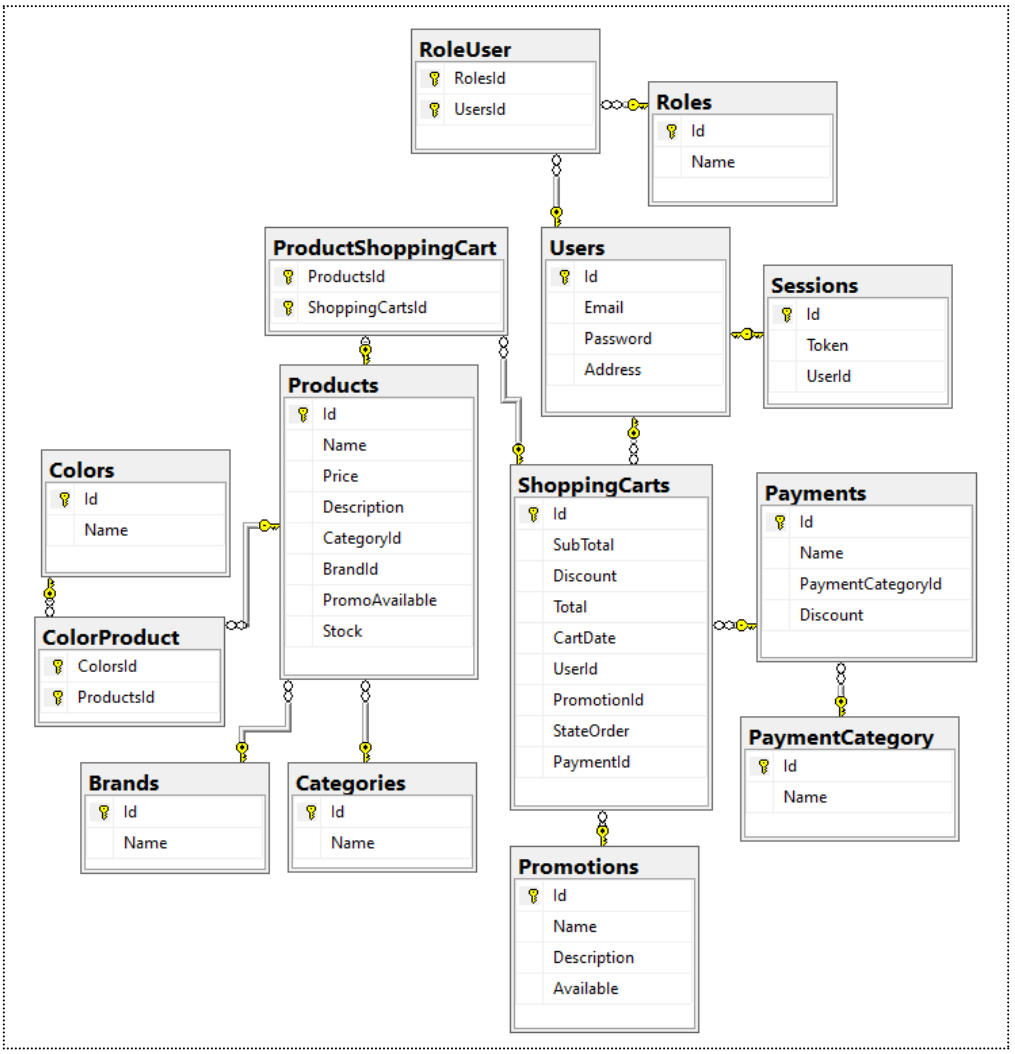
Modelado de la base de datos

Se trabajó con Entity Framework bajo la modalidad Code First. Este es un enfoque en el que el modelo de datos se define mediante código en lugar de configuración.



Entity Framework genera automáticamente las tablas de la base siguiendo la estructura (y relaciones) de las entidades del sistema. Naturalmente, para que Entity Framework comprenda qué modelo debe generar, deben seguirse ciertas convenciones en torno a cómo se definen las entidades y particularmente cómo referenciar una entidad en otra para generar la tabla que represente la relación que efectivamente se quiere implementar.

La base de datos va evolucionando con el desarrollo mediante el mecanismo de migraciones. Básicamente las migraciones son scripts que describen cómo actualizar la estructura de la base de datos en función de los cambios que se van produciendo a nivel de entidades y sus relaciones.



Es necesario hacer algunas precisiones respecto al manejo de los datos. La estrecha relación entre las entidades del sistema (incluyendo la base de datos) hace que la eliminación de un elemento pueda afectar la relación con otro elemento, dejando la base inconsistente. A modo de ejemplo, la eliminación de un producto puede corromper aquella orden de compra, cerrada, que lo tuviese como elemento en la lista de productos de la orden. Si bien en la primera entrega, la eliminación de productos de la base se hizo físicamente, para la segunda entrega se decidió lo siguiente:

- Los usuarios con órdenes cerradas (compras realizadas) o carrito de compras activo, no pueden ser eliminados.
- Los productos que se desean eliminar, se hace de forma lógica siguiendo el siguiente criterio; se pone el stock en cero y se deshabilita para participar de las promociones, modificando el campo “PromoAvailable” a false. Esto no impide que surja como resultado de una búsqueda, pero no permite ser agregado a un carrito.

Promociones y Reflection

Para cumplir con el requerimiento: "*Al trabajar con la primera versión del sistema, surgieron dos necesidades que deben resolverse en esta nueva versión:*

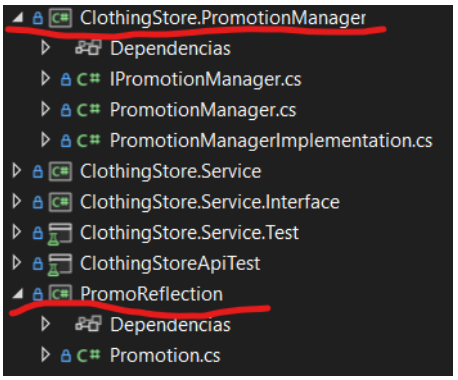
1. *La posibilidad de activar y/o desactivar una promoción.*
2. *La capacidad de agregar nuevas promociones.*

Debido a restricciones en el desarrollo e infraestructura, deseamos que estas funcionalidades puedan utilizarse de manera dinámica, sin necesidad de realizar ninguna configuración adicional ni depender de nuevas versiones del sistema.

Para lograrlo, el sistema deberá tomar las promociones alojadas en el servidor (se puede asumir una ruta específica) y realizar el cálculo basándose en las promociones disponibles en ese momento, ya sea al modificar el carrito o al efectuar una compra.

Como en la primera versión, cada promoción tendrá sus propias condiciones que determinarán si se aplica o no. Por lo tanto, si desactivamos una promoción (eliminamos el archivo del servidor), el sistema debe seguir funcionando y calcular el total de la compra sin incluir la promoción que se desactivó."

Intentado resolver todos las partes del problema se implementó una solución nueva basada en reflection y otra para la creación de promociones.



Estas dos, PromotionManager y PromoReflection nos ayudan a, por una parte, cargar promociones, eliminar las mismas, poder listar y ejecutar las que se encuentran en el servidor, y por otro lado, crear promociones dadas las condiciones y descuento requerido.

Primero, si necesitamos crear una promoción como elemento, debemos hacer uso de **Promotion.cs.** Allí es donde se agrega la información de la nueva promoción y se crea según un formato predeterminado, el algoritmo para deducir el descuento a aplicar sobre el conjunto de productos que hay en el carrito. Deben cambiarse los datos como **name** y **description** y se debe rellenar la función modelo **TotalWithDiscount** que debe devolver el descuento en tipo double que se aplica al conjunto de productos que se reciben como parámetro. Este listado de productos proviene de un carrito y

contiene sólo los elementos habilitados para aplicar descuentos. A continuación la clase a modificar y compilar para generar una dll que pueda ser usada por el sistema luego de ser cargada:

```
using ClothingStore.Domain.Entities;

namespace PromoReflection
{
    0 referencias
    public class Promotion
    {
        private static string name = "Descuento de muestra";
        private static string description = "Descripcion de muestra";

        0 referencias
        public string GetName() { return name; }
        0 referencias
        public string GetDescription() { return description; }
        1 referencia
        private double TotalAmount(List<Product> products)...
        0 referencias
        public double TotalWithDiscount(List<Product> products)
        {
            double discount = TotalAmount(products);

            //Calculate here your discount over products and return it
            //The list contains only available products for discount

            return discount;
        }

    }
}
```

Esta promoción luego de ser compilada se guarda como una dll en la carpeta /bin/Debug/net6.0 de la solución PromoReflection con el nombre *PromoReflection.dll*.

Ahora veamos **PromotionManager** que tiene varias funciones para manipular las promociones en tiempo de ejecución que va a utilizar nuestro sistema. Detalle:

- *private static string dllPath = "C:/windows/temp/Promotions";* :
  Esta es la ruta predeterminada desde dónde se van a leer los archivos dll de promociones.

- *public static void PromotionDllLoad(string pathToFile)* :
  Procedimiento que se encarga de recibir la ruta de la dll a cargar, copia la misma a *dllPath* y le cambia el nombre en el destino para que no existan conflictos.

- *public static void PromotionDllUnload(string filename)* :
  Procedimiento que toma el nombre del archivo relacionado a una dll que se encuentre en *dllPath* y lo elimina.

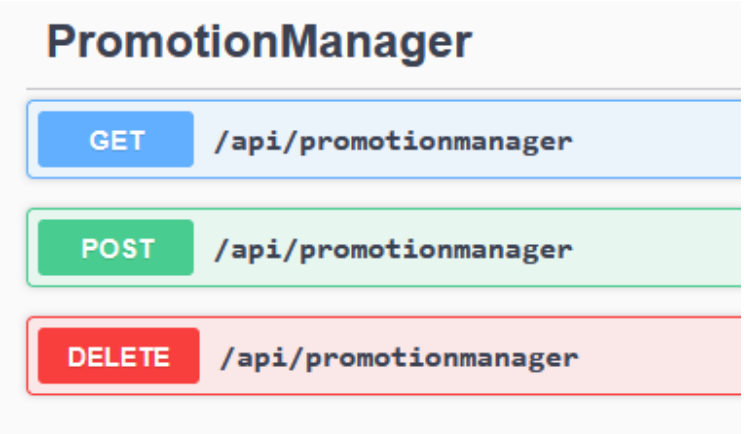
- *public List<Tuple<string, string, string>> GetPromotionList()* :
  Función que se encarga de devolver un listado con el nombre, descripción y nombre de archivo de cada promoción en el servidor.

- *public void RunPromotions(ShoppingCart sp)* :
  Este procedimiento ejecuta todas las promociones actuales sobre el conjunto de productos que se encuentran en el carrito pasado como parámetro.

Otras funciones internas no están disponibles dado que sirven solo en el contexto de **PromotionManager**.

(Usamos promo, promotion y promoción como sinónimos en el texto y en el sistema)

Se creó un controlador y un servicio para poder acceder a las funcionalidades que pueden ser usadas en el frontEnd. Como prueba se usó Swagger.



Y se ejecuta una solicitud para ver todas las promociones en el servidor. Este ejemplo incluye las promociones solicitadas en la primera entrega.

Response body


```
[
  {
    "item1": "20% OFF",
    "item2": "Condiciones del carrito: Tener 2 productos cualesquiera sean. Descuento: 20% de descuento en el producto de mayor valor.",
    "item3": "C:/windows/temp/Promotions\\Promo_20_OFF.dll"
  },
  {
    "item1": "3x1 Fidelidad",
    "item2": "Condiciones del carrito: Tener al menos 3 productos de la misma marca. Descuento: Los dos productos de menor valor son gratis.",
    "item3": "C:/windows/temp/Promotions\\Promo_3x1_fidelidad.dll"
  },
  {
    "item1": "3x2",
    "item2": "Condiciones del carrito: Tener 3 productos de la misma categoría. Descuento: El producto de menor valor es gratis.",
    "item3": "C:/windows/temp/Promotions\\Promo_3x2.dll"
  },
  {
    "item1": "Total look",
    "item2": "Condiciones del carrito: Tener al menos 3 productos del mismo color. Descuento: 50% de descuento en el producto de mayor valor.",
    "item3": "C:/windows/temp/Promotions\\Promo_Total_look.dll"
  }
]
```


La manipulación de promociones es descartada parcialmente sobre base de datos aunque se mantiene cierta compatibilidad dado el grado de especificidad que se requirió para el uso de EF Core.

Manejo de excepciones

Como se ha explicado previamente, se ha implementado un filtro encargado de gestionar las excepciones en la API. Este filtro tiene la responsabilidad de capturar cualquier excepción que se produzca en el sistema y determinar el mensaje y código de error a devolver. A continuación, se detallan las excepciones que el filtro es capaz de capturar y manejar:

- ArgumentException: En este escenario, se hace uso del código de estado 400 y se proporciona el mensaje: "Ha ocurrido un error al procesar la solicitud. Por favor, verifica que los datos enviados sean correctos."

- UnauthorizedAccessException: En caso de que se produzca esta excepción, establecemos el código de estado 401 y emitimos el mensaje: "No está autorizado para realizar esta acción."

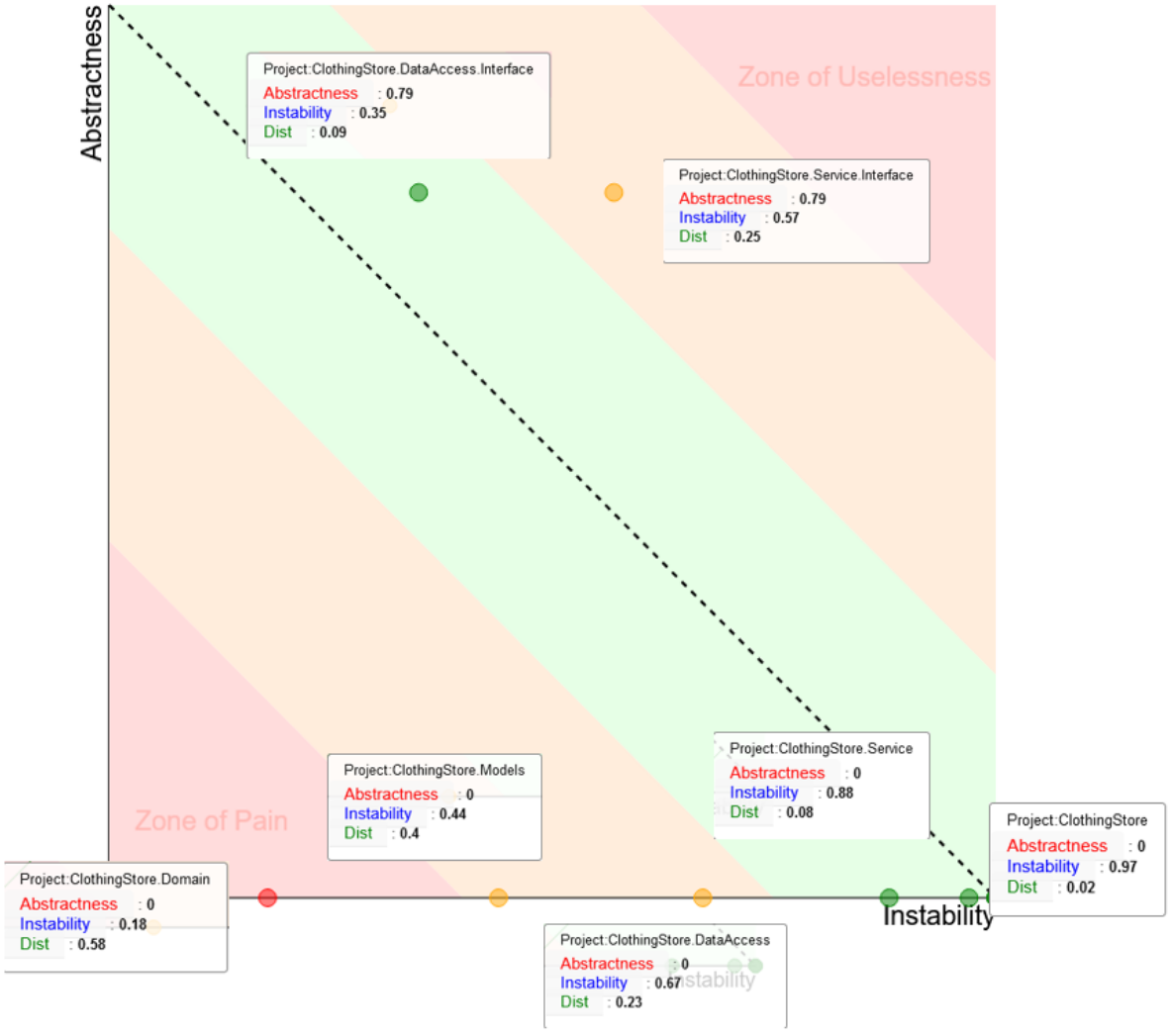
- NullReferenceException: Para esta excepción, configuramos el código de estado 404 y enviamos el mensaje: "No se encontró el recurso solicitado."

En caso de que se produzca alguna otra excepción no especificada anteriormente, se establece que la API devuelva un código de estado 500 con el mensaje: "Ha ocurrido un error interno en el servidor." Este enfoque garantiza una adecuada gestión de excepciones y una respuesta consistente en diversos escenarios de error.

Métricas

Si bien la generación de métricas en un proyecto de desarrollo de software es una herramienta que sirve para evaluar tanto el proceso como el producto final; en esta oportunidad el foco está en la mantenibilidad y el reuso.

Los valores que se presentan en esta sección fueron obtenidos con la aplicación NDepend acotando específicamente a aquellas métricas vistas en clase. Debemos precisar que esta corrida de NDepend fue realizada cuando el backend quedó terminado para todas los requisitos del primer y segundo obligatorio pero antes de implementar lo relacionado a Reflection. Esto último agrega un nuevo proyecto (paquete) que podría afectar los valores publicados, pero que entendemos no afecta conceptualmente a lo que se indica en esta sección.



En la gráfica, la secuencia principal y su zona cercana (en verde), representa la situación en la cual hay un cierto equilibrio entre abstracción y estabilidad. Es deseable que la mayoría de los paquetes se encuentren en esta zona; y aquellos por fuera, tengan una justificación apropiada a la luz de los principios de diseño de paquetes.

Esa posición viene dada por la métrica D (o D'). D' es la normalización de la métrica D, lo que permite analizar el valor dentro del intervalo [0,1]. En cualquier caso, el análisis demanda desagregar esta métrica en los valores de Inestabilidad y Abstracción que la componen y así ser más precisos en la explicación.

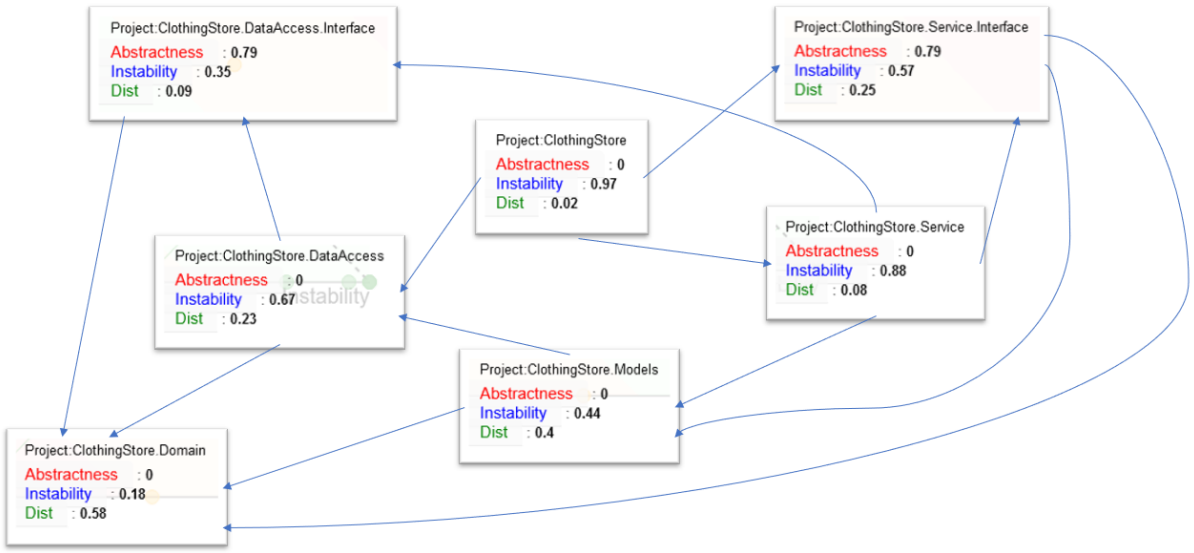
Paquete	Cohesión Relacional (H)	Inestabilidad (I)	Abstracción (A)	Distancia a la secuencia principal (D)	D normalizada (D')
ClothingStore(API)	1,18	0,97	0	0,021	0,029
ClothingStore.Service.Interface	1,07	0,57	0,79	0,25	0,36
ClothingStore.Service	1,82	0,88	0	0,082	0,12
ClothingStore.DataAccess.Interface	1,07	0,35	0,79	0,095	0,13
ClothingStore.DataAccess	1,86	0,67	0	0,23	0,33
ClothingStore.Models	1,92	0,44	0	0,4	0,56
ClothingStore.Domain	3,07	0,18	0	0,58	0,82

Los paquetes de dominio y modelos son los que más se alejan de la secuencia principal, debido a que son paquetes concretos. Asimismo presentan una baja inestabilidad (son estables). Esto es lógico teniendo en cuenta que representan una implementación específica del modelo de datos. Es decir, que es razonable que se encuentren en la zona de dolor; ya que una modificación afecta considerablemente al sistema en su conjunto.

En relación a las métricas mencionadas, hay dos principios que se deben tener en cuenta, por un lado el principio de abstracciones estables (SAP) y por otro, el principio de dependencias estables (SDP).

El primero indica que un paquete debería ser tan abstracto como es su estabilidad. Es decir, los paquetes más estables deben tender a ser más abstractos y los inestables a ser concretos. En otras palabras, los estables abiertos a la extensión y los inestables fácilmente modificables. Creemos que en este sentido, se ha cumplido con el principio.

Por otra parte, SDP indica que la dependencia debe ir en el sentido de la estabilidad. Es decir, que un paquete debe depender solamente de paquetes más estables que él. Para visualizar este principio podemos hacer uso del grafo de dependencias.



Del grafo se desprende que el proyecto Models depende de DataAccess, es decir, aquí se viola el principio SDP debido a que Models estaría dependiendo de un paquete menos estable. Es de hacer notar, que es una oportunidad de mejora ya que es razonable pensar que esta dependencia podría ser eliminada.

En cuanto al gráfico en sí, indicar que se cumple con otro de los principios en cuestión que es el principio de dependencias acíclicas; que no es otra cosa que la no existencia de dependencias circulares entre paquetes de la solución.

Las métricas analizadas están relacionadas a los principios orientados al acoplamiento de los paquetes. A continuación indicamos los principios vinculados a la cohesión y las métricas correspondientes.

En particular, mencionar la cohesión relacional. Mide el grado de cohesión interna de un paquete. Es decir, indica qué tan relacionadas están las clases de un paquete en función de la cantidad de clases e interfaces y la cantidad de relaciones entre estas. Lo deseable es que este número se encuentre entre 1.5 y 4.0. Se observa que tres de los paquetes están por debajo de 1.5, lo que significa que hay pocas relaciones para la cantidad de clases, lo que se traduce en que las clases no están suficientemente relacionadas.

La primera explicación que se presenta es el hecho que estos paquetes tal vez sean demasiado grandes o más explícitamente, tienen demasiadas responsabilidades, las cuales no están estrechamente relacionadas entre sí. Esto hace que un conjunto de clases con pocas relaciones entre sí, estén agrupadas bajo el mismo namespace.

Si se observa bajo la óptica de los principios de diseño de paquetes; en términos generales no se estaría cumpliendo con el principio de Clausura Común. Esto es un principio orientado al mantenimiento, que establece que las clases que cambian juntas deben permanecer juntas.

En el mismo sentido y en referencia al reuso; las clases que pertenecen a un paquete se reusan juntas (Principio de Reuso Común). Dado el bajo valor de cohesión relacional, podríamos estar ante la situación de tener que cargar con clases que realmente no serían requeridas en un nuevo proyecto que pretendiera reusar parte de este desarrollo.

El análisis aplicando sólo los principios de diseño de paquetes puede llevar a tomar decisiones equivocadas. Es lógico pensar que dividir los paquetes problemáticos en otros más pequeños y cohesivos es la solución inmediata. Sin embargo, estos criterios deben contrastarse con otros principios (tal vez a nivel de clases) y siempre teniendo en cuenta el contexto particular de cada sistema que se esté desarrollando.

Conclusiones

Teniendo en cuenta las dificultades para presentar la funcionalidad requerida en el primer obligatorio, se podría suponer que la nueva entrega demandaría un rediseño de la solución. Sin embargo, los problemas, aun cuando fueron muy relevantes, estuvieron acotados a la capa de acceso a datos. En cierto sentido, estamos diciendo que fue justamente el diseño lo que permitió recomponer la situación para la segunda entrega.

A lo largo de este documento, se destacó la importancia de hacer uso de los principios de diseño vistos en clase con el objetivo de hacer más mantenible el sistema. Esa fué la intención en todo momento.

El hecho de poder restringir los problemas a un entorno acotado, no es la única evidencia de que el diseño favorece la mantenibilidad. Tal vez la mejor forma de visualizar esto, sería evaluar el impacto que tuvo sobre el sistema la incorporación de nuevos requisitos para la segunda entrega.

En primer término se solicitó ampliar la gestión de productos, permitiendo la verificación de stock y controlar la posibilidad que un producto fuera elegible o no para participar de las promociones. Haciendo un análisis por capas de la aplicación, el impacto fue el siguiente: en la base de datos, la tabla productos requirió agregar dos campos, resuelto en una única migración. Los modelos de entrada y salida (crear, actualizar y mostrar producto) se ajustaron con los dos campos mencionados. A nivel del repositorio, ninguno de los métodos se vio afectado. A nivel de servicio, se agregó el método que verifica el stock. Este método es invocado desde el método que agrega producto y el que realiza la venta. Estos últimos se modificaron agregando la condición indicada. Algo similar sucede con la habilitación de productos para participar en promociones. El impacto es mínimo. Se extiende la funcionalidad, con nuevos métodos, pero las modificaciones son menores y acotadas.

Relacionado también a la gestión de productos, el requerimiento del filtrado por precio, no tuvo más impacto que el de crear el endpoint correspondiente.

En segundo lugar, las modificaciones al proceso de compra, parecía en un principio requerir modificar parte relevante del código. Naturalmente fué necesario crear nuevas entidades en el código y las tablas correspondientes en la base. Asimismo, modificar shoppingCart (orden de compra) para incorporar el medio de pago. Este requerimiento adicional atraviesa todas las capas de la aplicación. Sin embargo, implementar la forma de pago no tuvo un impacto significativo. El proceso de compra demanda integrar distintas acciones. A saber, verificar stock, calcular el subtotal, calcular el descuento mediante la corrida de las promociones vigentes, registrar la compra y actualizar el stock. El hecho de implementar el proceso de compra independizando estas acciones permitió incorporar el medio de pago sin afectar al resto. Es de hacer notar, que a su vez, se desarrolló este requerimiento pensando en la posibilidad de extenderlo a nuevos tipos de pago, evitando crear reglas específicas o haciendo RTTI. Igualmente fue necesario modificar la firma del método para tomar la forma de pago como un parámetro.

Sin lugar a dudas, el requerimiento adicional más importante, desde el punto de vista del volumen de trabajo a realizar y la complejidad del mismo, fue la gestión de promociones de manera dinámica mediante el mecanismo de reflection.

Se creó un proyecto nuevo para incorporar a la solución, pero se desarrolló independientemente del resto. De hecho se manejó como un subproyecto independiente con la intención de integrarlo cuando estuviera listo. Mientras tanto, se siguió con el trabajo, accediendo a las promociones como en la primera entrega.

Tal es así, que el Promotion Manager creado, tiene como única dependencia el dominio. A la hora de integrarlo a la solución, simplemente se reemplazó el método “RunPromotions” existente por el nuevo. Es decir, que ninguno de los procesos vinculados, como ser el cálculo de descuento o la venta misma se vió afectado. Se podría agregar además que el propio Promotion Manager está estructurado para facilitar la extensibilidad y el mantenimiento.

Por último indicar, que las métricas no necesariamente reflejan que el diseño es óptimo ni mucho menos. El solo hecho de observar los valores obtenidos, da la pauta de que hay oportunidades de mejora.

Anexo

Especificación de la API

Para la segunda entrega fue necesario modificar en gran parte del código, razón por la cual se incluye en este anexo todos los endpoints en lugar de solo los que fueron modificados o agregados.

Usuarios

Recurso: users
Acción: GET
Ubicación: /users
Headers: token
Parámetros: -
Modelos Entrada: -
Modelos Salida: List<User>
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Listado de todos los usuarios del sistema, requerimiento para el rol administrador

Recurso: users/{id}
Acción: GET
Ubicación: /users
Headers: token (admin o usuario logueado con el id parámetro)
Parámetros: [route] id
Modelos Entrada: -
Modelos Salida: User
Respuestas: 200 OK, 204 No Content, 401 Unauthorized (no hay token), 403 Forbidden (token no válido)
Justificación del Recurso: Información de un usuario particular, requerimiento para el rol administrador o cuando un usuario quiere ver sus propios datos guardados.

Recurso: users/byEmail/{email}
Acción: GET
Ubicación: /users
Headers: token
Parámetros: [route] email
Modelos Entrada: -
Modelos Salida: User
Respuestas: 200 OK, 204 No Content, 401 Unauthorized (no hay token), 403 Forbidden (token no válido)
Justificación del Recurso: Información de un usuario particular, requerimiento para el rol administrador.

Recurso: users
Acción: POST
Ubicación: /users
Headers: Si bien lo puede hacer un admin, no requiere token
Parámetros: [body] email, password, role, address
Modelos Entrada: UserRequestDTO
Modelos Salida: -
Respuestas: 201 Created, 400 BadRequest
Justificación del Recurso: Creación de usuarios.

Recurso: users/{id}
Acción: PUT
Ubicación: /users
Headers: token (de admin o el usuario id logueado)
Parámetros: [route] id; [body] password?, role?, address?
Modelos Entrada: User
Modelos Salida: -
Respuestas: 200 Ok, 400 BadRequest, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Modificación de usuarios.

Recurso: users/{id}
Acción: DELETE
Ubicación: /users
Headers: token de admin
Parámetros: [route] id
Modelos Entrada: -
Modelos Salida: -
Respuestas:200 Ok, 400 BadRequest, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Eliminación de usuarios.

Sesiones

Recurso: sessions
Acción: POST
Ubicación: /sessions/login
Headers: -
Parámetros: [body] email, password
Modelos Entrada: SessionRequestDTO
Modelos Salida: Token
Respuestas: 200 OK, 400 BadRequest,
Justificación del Recurso: Login de usuarios.

Recurso: sessions
Acción: DELETE
Ubicación: /sessions/{token}
Headers: token de usuario id logueado
Parámetros: [route] token
Modelos Entrada: token
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Logout de usuarios.

Recurso: users
Acción: GET
Ubicación: /sessions/{token}
Headers: token
Parámetros: [route] token
Modelos Entrada: token
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Obtener la session de un usuario a través del token de sesión.

Productos

Recurso: products
Acción: GET
Ubicación: /products
Headers:
Parámetros: -
Modelos Entrada: -
Modelos Salida: List<Product>
Respuestas: 200 OK
Justificación del Recurso: Listado de productos

Recurso: products
Acción: GET
Ubicación: /products/{id}
Headers:
Parámetros: [route] id (identificador del producto)
Modelos Entrada:
Modelos Salida: Product
Respuestas: 200 OK, 204 NoContent
Justificación del Recurso: Detalle de un producto específico

Recurso: products
Acción: GET
Ubicación: /products
Headers:
Parámetros: [route] text?, brand?, category?
Modelos Entrada:
Modelos Salida: List<Product>
Respuestas: 200 OK, 204 NoContent
Justificación del Recurso: Búsqueda de productos por texto, marca o categoría. La búsqueda del texto se realiza en todos los campos de la tabla de productos.

Recurso: products
Acción: POST
Ubicación: /products
Headers: token de admin
Parámetros: [body] name, price, description, brand, category, colors[], stock, promoAvailable
Modelos Entrada: ProductRequestDTO
Modelos Salida: -
Respuestas: 201 Create, 400 BadRequest, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Creación de productos. Requerimiento para el administrador.

Recurso: products
Acción: PUT
Ubicación: /products/{id}
Headers: token de admin
Parámetros: [route] id (de producto) [body] name?, price?, description?, brand?, category?, colors[]?, stock?, promoAvailable?
Modelos Entrada: ProductUpdateDTO
Modelos Salida: -
Respuestas: 200 Ok, 400 BadRequest, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Modificación de productos. Requerimiento para el administrador.

Recurso: products
Acción: DELETE
Ubicación: /products/{id}
Headers: token de admin
Parámetros: [route] id (de producto)
Modelos Entrada: -
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Eliminación de productos. Requerimiento para el administrador.

Recurso: products
Acción: GET
Ubicación: /products/price
Headers:
Parámetros: [route] startPrice, endPrice
Modelos Entrada: -
Modelos Salida: List<Product>
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Listado de productos entre dos precios determinados.

Carrito de compras / Ordenes

Recurso: shoppingCarts
Acción: GET
Ubicación: /shoppingCarts/sales
Headers: token de admin
Parámetros:
Modelos Entrada: -
Modelos Salida: List<ShoppingCartResponseDTO>
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Listado de ventas realizadas. Requerimiento para el rol administrador. Existe un endpoint similar en /shoppingCarts para ver todas los shoppingCarts (incluyendo los de estado “pending”), pero no es un requisito de letra

Recurso: shoppingCarts
Acción: GET
Ubicación: /shoppingCarts/sales/{userId}
Headers: token de usuario o admin
Parámetros: [route] id de usuario
Modelos Entrada: -
Modelos Salida: List<ShoppingCartResponseDTO>
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Listado de compras realizadas por un usuario particular. Tanto el usuario logueado como el administrador pueden verlo. Existe un endpoint similar en /shoppingCarts/{id} para ver un shoppingCart específico por id de carrito., pero no es un requisito de letra

Recurso: shoppingCarts
Acción: POST
Ubicación: /shoppingCarts
Headers: token de usuario logueado
Parámetros: [body] email de usuario logueado
Modelos Entrada: ShoppingCartRequestDTO
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Crear carrito de compra.

Recurso: shoppingCarts
Acción: PUT
Ubicación: /shoppingCarts/add/{shoppingCartId}
Headers: token de usuario logueado
Parámetros: [body] id (de producto)
Modelos Entrada: -
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Agregar producto al carrito de compras.

Recurso: shoppingCarts
Acción: PUT
Ubicación: /shoppingCarts/remove/{shoppingCartId}
Headers: token de usuario logueado
Parámetros: [body] id (de producto), [route] id de carrito
Modelos Entrada: -
Modelos Salida: -
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Eliminar producto del carro

Recurso: shoppingCarts
Acción: GET
Ubicación: /shoppingCarts/total/{shoppingCartId}
Headers: token de usuario logueado
Parámetros: [route] id de carrito
Modelos Entrada: -
Modelos Salida: double
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Calcular el precio del carrito aplicando promociones. Era una funcionalidad de la primera entrega para verificar que el total independientemente del descuento. No se accede desde el frontend.

Recurso: shoppingCarts
Acción: GET
Ubicación: /shoppingCarts/discount/{shoppingCartId}
Headers: token de usuario logueado
Parámetros: [route] id de carrito
Modelos Entrada: -
Modelos Salida: double
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Calcula el descuento, corriendo las promociones sobre el carrito.

Recurso: shoppingCarts
Acción: PUT
Ubicación: /shoppingCarts/sales/{shoppingCartId}
Headers: token de usuario logueado
Parámetros: [route] id de carrito
Modelos Entrada: -
Modelos Salida: ShoppingCartSaleDTO
Respuestas: 200 Ok, 401 Unauthorized (no hay token),403 Forbidden (token no válido)
Justificación del Recurso: Registrar la compra.