Universidad ORT Uruguay

Facultad de Ingeniería

Bernard Wand Polak

Licenciatura en Sistemas - 2023

Diseño de aplicaciones 2

Obligatorio 2

https://github.com/IngSoft-DA2-2023-2/203832-241699-238927

Agustín Fioritto	**Nro est. 238927**
Joaquín Calvo	**Nro est. 203832**
Chiara Sosa	**Nro est. 241699**

ÍNDICE

Descripción general del trabajo

Este proyecto se estructuró en dos etapas principales. La primera fase se dedicó a la implementación del Backend y la API REST, estableciendo así los cimientos necesarios para la ejecución de la segunda fase: el desarrollo del Frontend de una aplicación web integral destinada al comercio electrónico.

Las principales funcionalidades del proyecto incluyen:

- Gestión de usuarios: registro de nuevos usuarios, inicio de sesión y cierre de sesión de usuarios ya creados, actualización de datos de usuarios, historial de compras.

- Manejo de productos del comercio: creación, eliminación y actualización de productos, búsqueda de productos según sus características.

- Gestión de carritos de compras: agregar y eliminar productos a los carritos, cálculo y aplicación de la mejor promoción aplicable al carrito disponible, distintas opciones de métodos de pago.

- Ejecución de pedidos: procesos de creación y confirmación de pedidos.

- Persistencia de datos: la información sobre clientes, productos, carritos y compras es almacenada en una base de datos, con la cual se interacciona en todo momento para actualizarla según las operaciones realizadas.

En esta fase final, nos enfocamos en cumplir en su totalidad los requerimientos planteados y luego desarrollar el front end, incluyendo los datos de la aplicación, que fueron accedidos mediante los endpoints instanciados en el back end.

En este documento se detallan distintos diagramas respectivos al modelo 4 + 1 de la solución, los cuales nos permitieron tanto a nosotros como equipo de trabajo comprender y asociar ciertos aspectos del sistema, como también comunicarlos a terceros interesados en el proyecto. Se incluyen diagramas de clases de los distintos paquetes para modelar la estructura interna del sistema desde una perspectiva *lógica*. Respecto a la *vista de procesos,* se detallan diagramas de secuencia de dos casos de uso que se consideran relevantes, ayudando a visualizar cómo se ejecutan los mismos y cómo se coordinan las actividades internas del sistema. Por último, para ilustrar la vista de implementación, se adjuntan los diagramas de componentes y paquetes.

Bugs

Se eliminaron los bugs pendientes respecto a la primera entrega.

- Si se aplica una promoción a un carrito, el precio con descuento no es mostrado en el frontend.

- Por cuestiones de tiempo, no se pudo implementar la lógica de agregar más de un mismo producto al carrito (para realizar la compra), pero el stock es actualizado con normalidad al comprar.
- La búsqueda de productos por rango fue realizada en el backend (controladores), pero por razones de tiempo no se logró poner a disposición del frontend.

Diagrama general de paquetes

La solución se diseñó de manera tal que la organización y la estructura de la misma fomenten la eficiencia, la escalabilidad y la mantenibilidad del sistema, evitando dependencias innecesarias entre paquetes para lograr un bajo acoplamiento entre ellos. Esta estructura en capas (layers) facilita la comprensión y el mantenimiento del código, organizándolo de manera coherente y lógica. Cada capa tiene responsabilidades claras y se comunica con las capas adyacentes de manera que no se intercambie información innecesaria. Esto ayuda a la modularidad y la facilidad de mantenimiento del sistema a medida que se desarrolla y escala.

A continuación se proporciona una breve descripción de cada una de las capas:

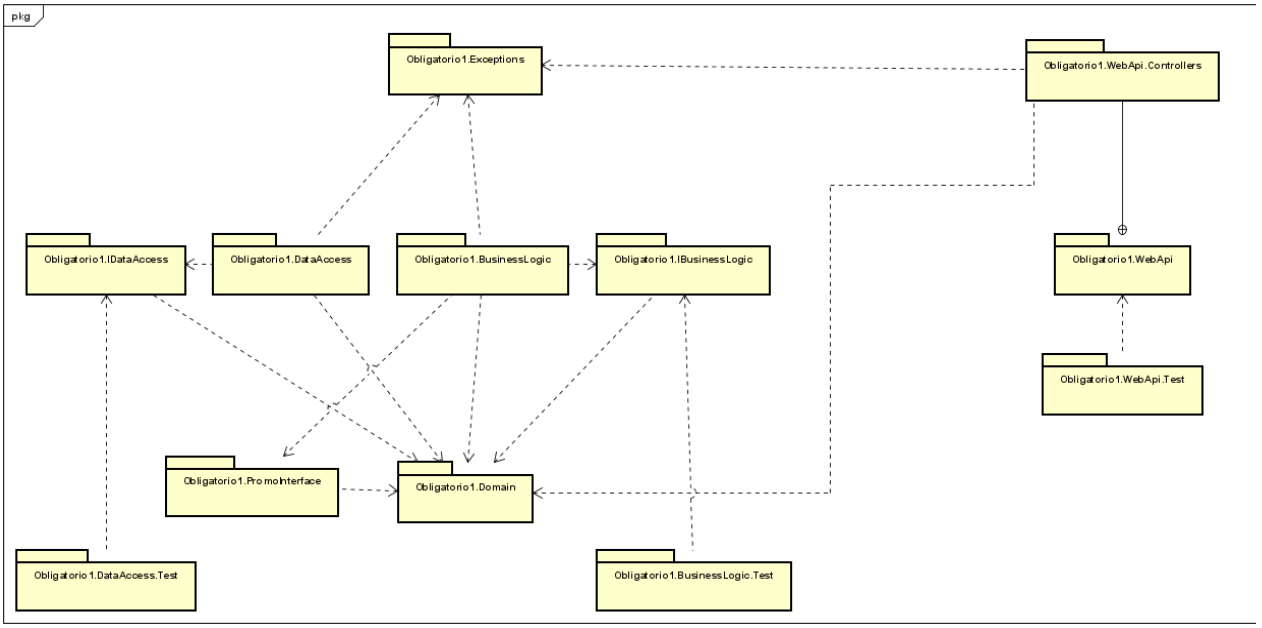
Capa del Dominio: El paquete "Obligatorio1.Domain" contiene las entidades del dominio, como "User", "Cart", “Product”, "Purchase", “Session”, “CartProduct” y “PurchaseProduct”. Estas entidades representan los objetos utilizados. La capa del dominio se encarga de definir la estructura de datos fundamental de la aplicación.

Capa de Lógica de Negocio: El paquete "Obligatorio1.BusinessLogic" alberga la lógica de las operaciones del sistema. Cada clase en este paquete contiene métodos para realizar operaciones relacionadas con las entidades, habiendo una clase de servicio para cada entidad. Además, se creó un proyecto con la interfaz relacionada a las promociones, la cual implementará todas las promociones que se incluyan en la solución.
Esto ayuda a mantener la lógica del negocio separada de otras capas. El paquete "Obligatorio1.IBusinessLogic" contiene interfaces implementadas por BusinessLogic, y será el punto de acceso a estos servicios sin tener que depender de las implementaciones. Esta lógica también incluye un paquete con tests.

Capa de Acceso a Datos: El paquete "Obligatorio1.DataAccess" se encarga de gestionar el acceso a la base de datos. La clase GenericRepository de este paquete interactúa directamente con la base de datos y, mediante su respectiva interfaz contenida en “Obligatorio1.IDataAcces” ayuda a mantener la lógica de acceso a datos separada de su utilización e interacción con la lógica de negocio. La clase GenericRepository contiene operaciones genéricas para todas las clases del dominio, que permite a cada servicio obtener los datos necesarios para realizar sus operaciones, sin tener que procesar los datos en la capa de acceso a datos.

Capa de Web Api: En el paquete "Obligatorio1.WebApi," se encuentran los controladores necesarios para permitir que los usuarios interactúen con la aplicación a través de una interfaz RESTful. Estos controladores manejan las solicitudes HTTP y comunican las acciones del usuario con la lógica de negocio. También se incluye un paquete de tests para estas operaciones.

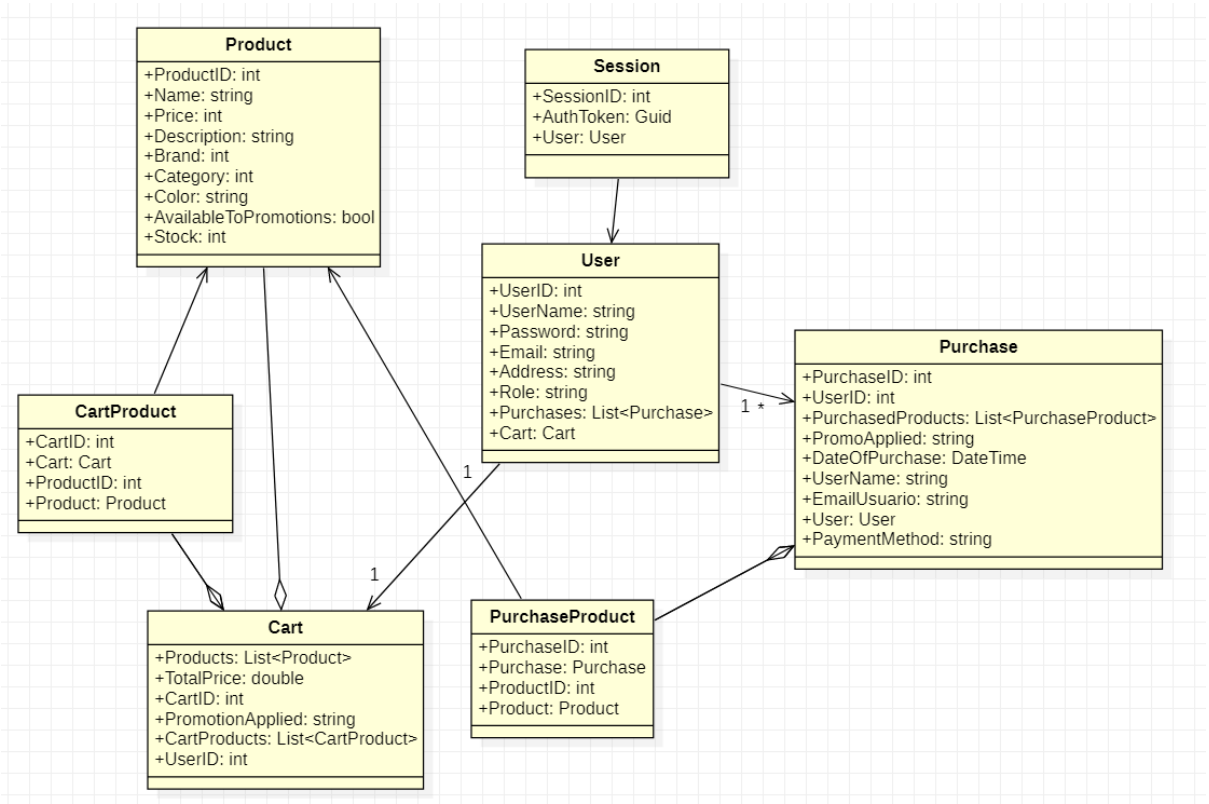
A continuación se detalla un diagrama general de los paquetes que componen el proyecto conformando cada una de estas capas.



Paquete Obligatorio1.Domain

Dentro de Domain, implementamos todas las entidades del dominio siendo estas las representaciones de los objetos del mundo real en nuestra aplicación. Estas clases únicamente contienen sus atributos y su constructor. El siguiente diagrama de clases representa los atributos de cada entidad y las relaciones entre las clases (incluyendo su respectiva cardinalidad). Se evitó incluir para cada una de ellas los constructores sin y con parámetros para descomplejizar el diagrama.

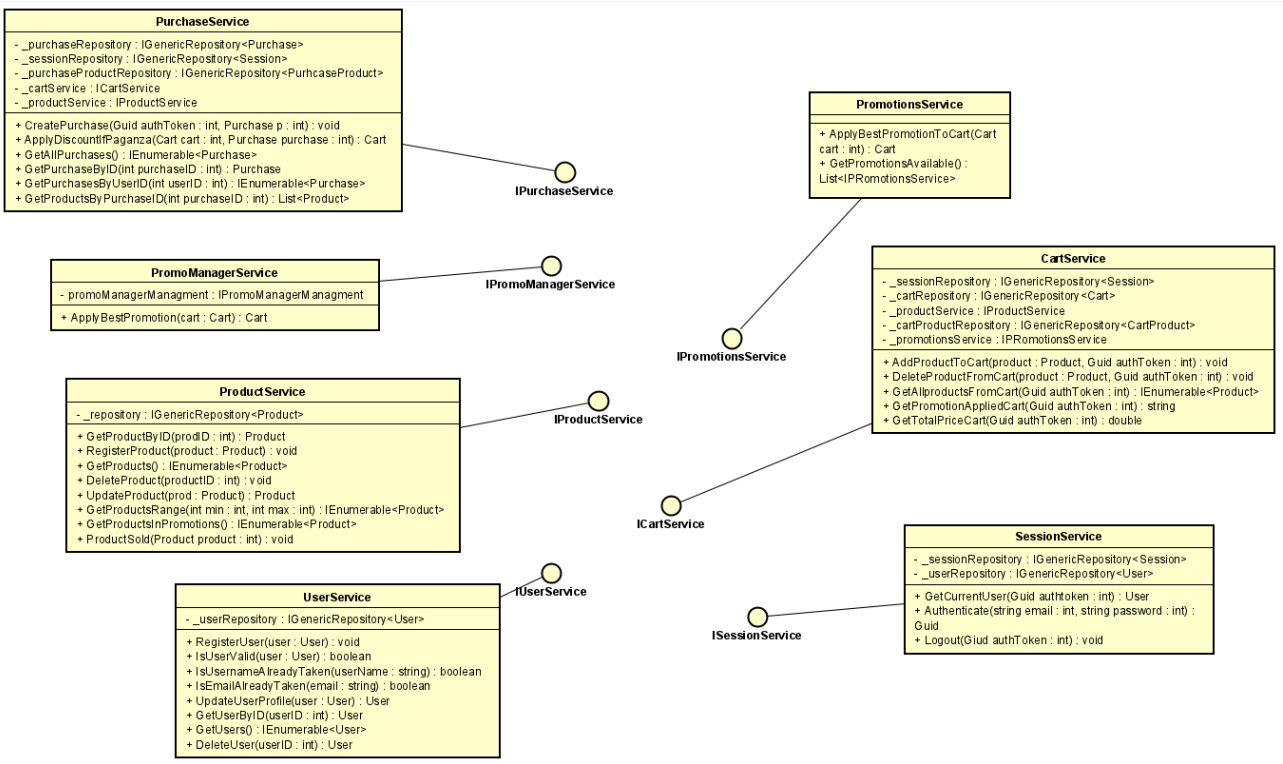
La clase User tiene una relación de asociación con las clases Cart y Purchase, haciendo uso de las mismas para instanciar algunos de sus atributos. La clase Cart está asociada a Product, y la clase Purchase tiene una dependencia de agregación con la clase Product, dado que está formada por productos y no pueden crearse instancias de Purchase si no existen productos en nuestro proyecto.



Paquete Obligatorio1.BusinessLogic

En el apartado de BusinessLogic, se pueden encontrar todas las operaciones disponibles en el proyecto, para cada una de las clases. La sintaxis de los nombres de las clases que contiene BusinessLogic, es el nombre de la clase seguido de la palabra service, a modo de ejemplo, UserService que contendrá todas las operaciones correspondientes a User y el manejo de usuarios: registro, logueo, actualización de datos, etc.

En el diagrama se representan, además de las clases incluidas en este paquete, qué interfaces de IBusinessLogic son implementadas por cada una de las clases.



Paquete Obligatorio1.IBusinessLogic

IBusinessLogic es un conjunto de interfaces, que contienen las definiciones de las funcionalidades, implementadas luego dentro de cada una de sus clases dentro de BusinessLogic. Esto se realizó no solo para abstraer la implementación de la lógica y poder hacer uso de las funcionalidades en otras partes de la solución sin depender de la misma, sino que también nos permitió flexibilidad a la hora de obtener distintas implementaciones en simultáneo, en el caso de las promociones. Por ejemplo para la clase userService, su interfaz será IUserService.

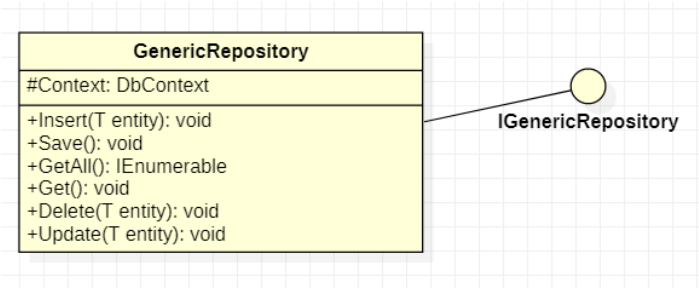
Paquete Obligatorio1.BusinessLogic.Test

En BusinesssLogic.Test, realizaremos todos los tests asociados a Business Logic.

Paquete Obligatorio1.DataAccess

La principal responsabilidad de DataAccess, es la gestión del contexto de la base de datos con la que vamos a trabajar. Contiene las implementaciones para las operaciones descritas en la interfaz de IDataAccess: IGenericRepository.

El siguiente diagrama muestra la clase GenericRepository (contenida en la carpeta Repositories), y sus métodos para acceder a la base de datos, siendo estos genéricos para poder ser utilizados por todas las entidades del dominio. Además se muestra la interfaz que implementa, IGenericRepository, incluida en el paquete IDataAccess.



Paquete Obligatorio1.IDataAccess

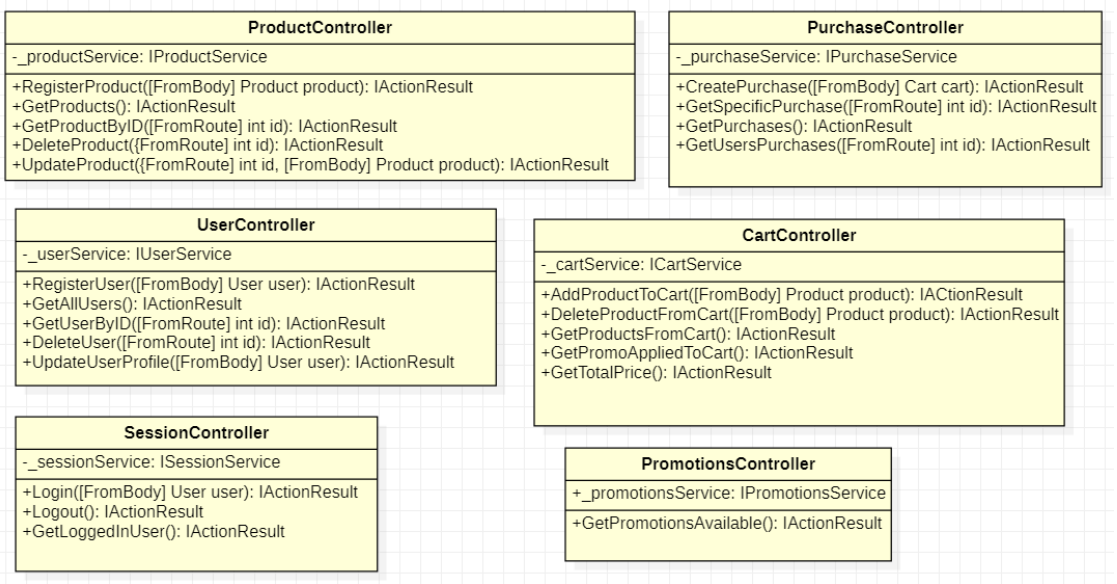
IDataAccess contiene la interfaz implementada en DataAccess. Al hacer uso de interfaces, nos permitió generar una abstracción de la lógica de las funciones que no son necesarias para la manipulación de datos en la base de datos.

Paquete Obligatorio1.DataAccess.Test

En DataAccesTest, realizaremos todos los tests asociados a DataAccess.

Paquete Obligatorio1.WebApi

Dentro del proyecto Obligatorio1.WebApi encontraremos los controladores. Los dividimos en seis clases, cada una haciendo referencia a los endpoints relacionados a una clase del dominio.



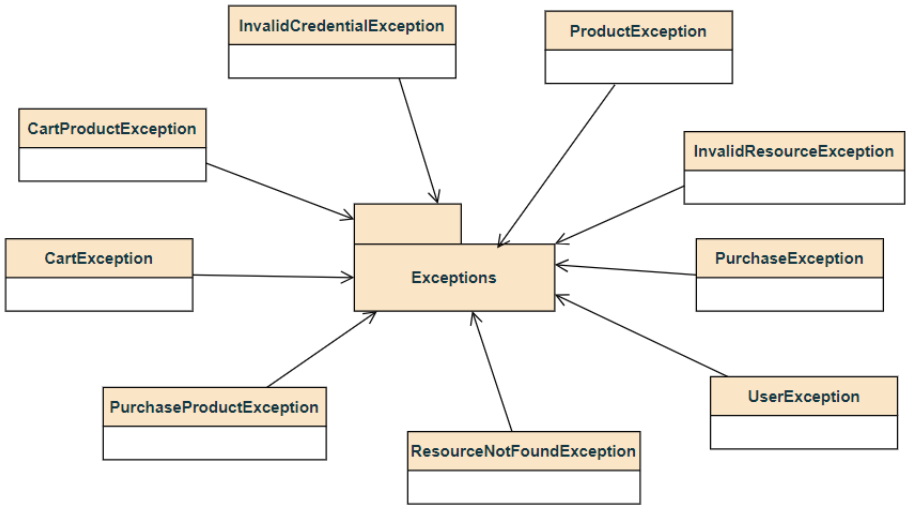
Los 6 controllers de la solución heredan la clase abstracta ControllerBase (perteneciente a Microsoft.AspNetCore.Mvc) para poder definir acciones, enrutamiento y gestionar solicitudes y respuestas HTTP.

Paquete Obligatorio1.WebApi.Test

En Obligatorio1.WebApi.Test, realizamos todos los tests asociados a los endpoints de los controladores en la WebApi.

Paquete Obligatorio1.Exceptions

Este paquete contendrá clases definidas para distintas excepciones. Cada una de estas clases implementa una excepción para cada objeto definido en el obligatorio. A modo de ejemplo, para los usuarios se definió la excepción UserExceptions.



Paquete Obligatorio1.PromoInterface

Este paquete contiene la interfaz que es implementada por las promociones que se agregan a la solución, mediante la cual se comunicará el resto de la aplicación para aplicar las distintas promociones sobre los productos. En el diagrama detallado anteriormente en la sección de diagrama de paquetes, se puede visualizar como las 4 promociones incluidas en la solución hasta el momento implementan la interfaz contenida en este paquete.

Además se incluye una clase PromoUtility que contiene métodos en común que utilizan las distintas promociones. La creación de esta clase fue con el objetivo de reutilizar código y evitar la redundancia de métodos.

Paquetes de promociones

Cada vez que se quiera agregar una nueva promoción a la solución, se debe ingresar como un nuevo proyecto independiente, cumpliendo con el requisito de que la clase que lo contenga implemente la interfaz IPromoService del paquete PromoInterface para ser reconocida como una promoción a lo largo de la solución.

Paquete Obligatorio1.ServiceFactory

Este paquete contiene la clase ServiceFactory que está diseñada para gestionar la configuración y registro de servicios en una aplicación .NET, utilizando el patrón de inyección de dependencias. Esta clase se encarga específicamente de registrar servicios relacionados con la gestión de CORS (Cross-Origin Resource Sharing).

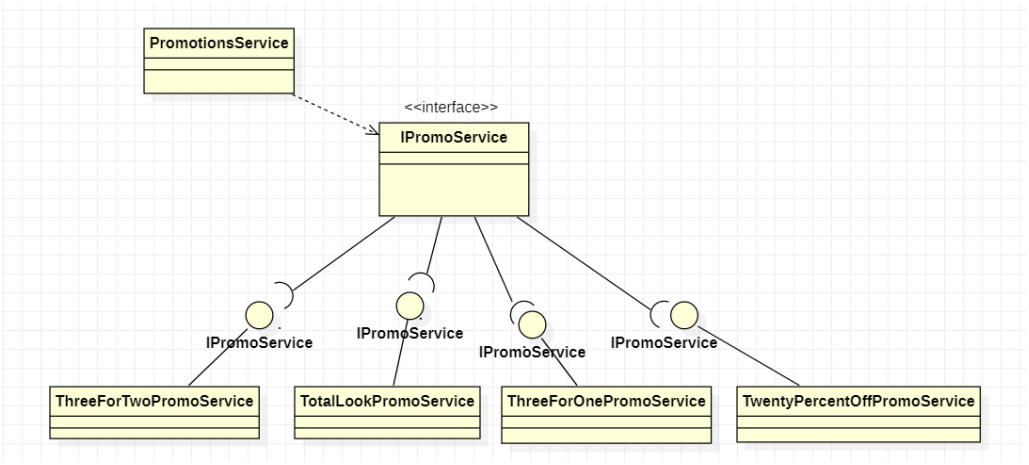
Jerarquías de herencia

La estructura de herencia en nuestro proyecto se basa principalmente en el uso de interfaces. A lo largo del trabajo, nos enfocamos en utilizar interfaces para separar la lógica de las operaciones de su implementación. Esto nos permite tener un código más flexible y fácil de mantener. Esta técnica se aplicó tanto en la parte de la lógica de negocio como en el acceso a datos, donde cada clase tiene una interfaz asociada. De esta manera, podemos realizar ciertas operaciones en todo el proyecto sin necesidad de conocer los detalles de cómo se ejecutan.

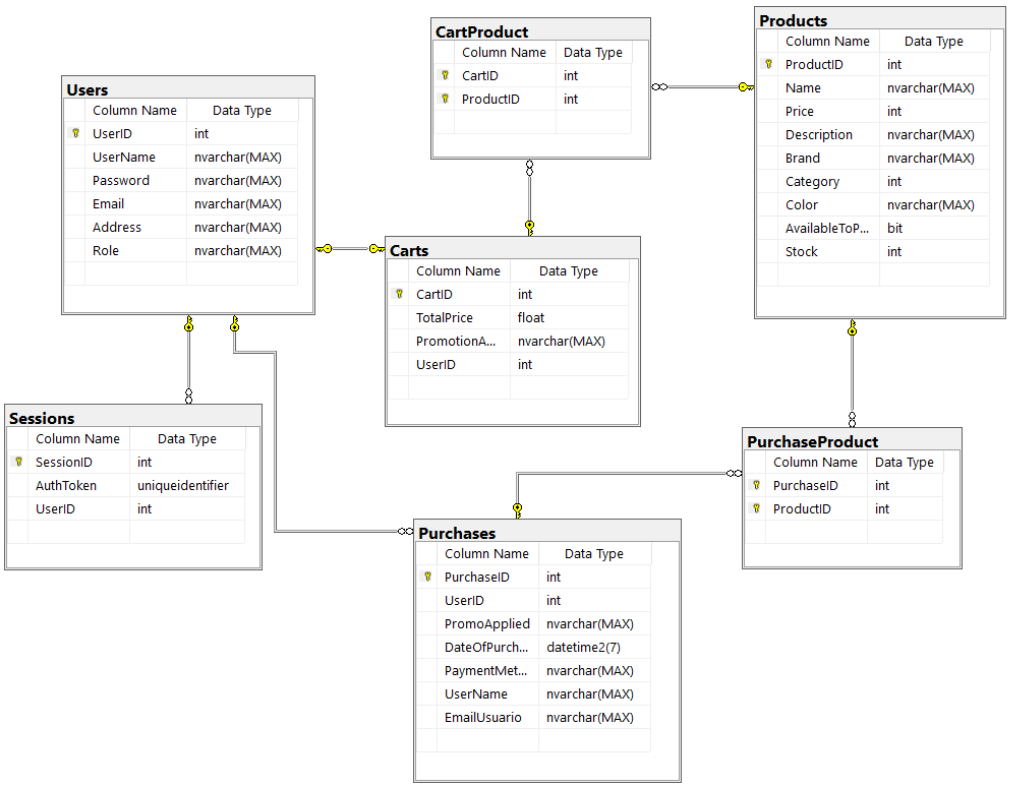
Un ejemplo interesante en el cual se aplicó esta técnica es en el manejo de las promociones. Implementamos una única interfaz llamada IPromoService encontrada dentro del proyecto PromoInterface, que contiene un método para calcular el precio del carrito aplicando una promoción y otro para devolver el nombre de la promoción. Esta interfaz es implementada por cada nueva promoción que se agregue a la solución, de manera que cada una de ellas es encargada de calcular el precio final del carrito según el tipo de promoción que representa y de asignar un nombre a la promoción. Al momento de aplicar la mejor promoción al carrito de compras después de haber agregado o eliminado un producto, se comparan los resultados que devuelven cada una de las promociones vigentes, todas del tipo IPromoService. Estos resultados se obtienen mediante la llamada al método en común instanciado en la interfaz, sin necesidad de conocer cómo están implementados los algoritmos internamente.

En este caso, seguimos el patrón de diseño "strategy", que sugiere crear una interfaz única con las funcionalidades que necesitamos y permitir que cada clase que la implementa desarrolle su propia implementación según sus necesidades específicas. Esto nos ha ayudado a mantener un código más organizado y fácil de extender en nuestro proyecto.

En el siguiente diagrama de clases se puede visualizar claramente el uso del patrón. Se representan las clases que participan en esta jerarquía, al igual que la clase CartService que es la que hace uso de la misma y se ve beneficiada por el uso de interfaces.



Modelo de tablas



Diagramas de interacción

Añadir un producto al carrito

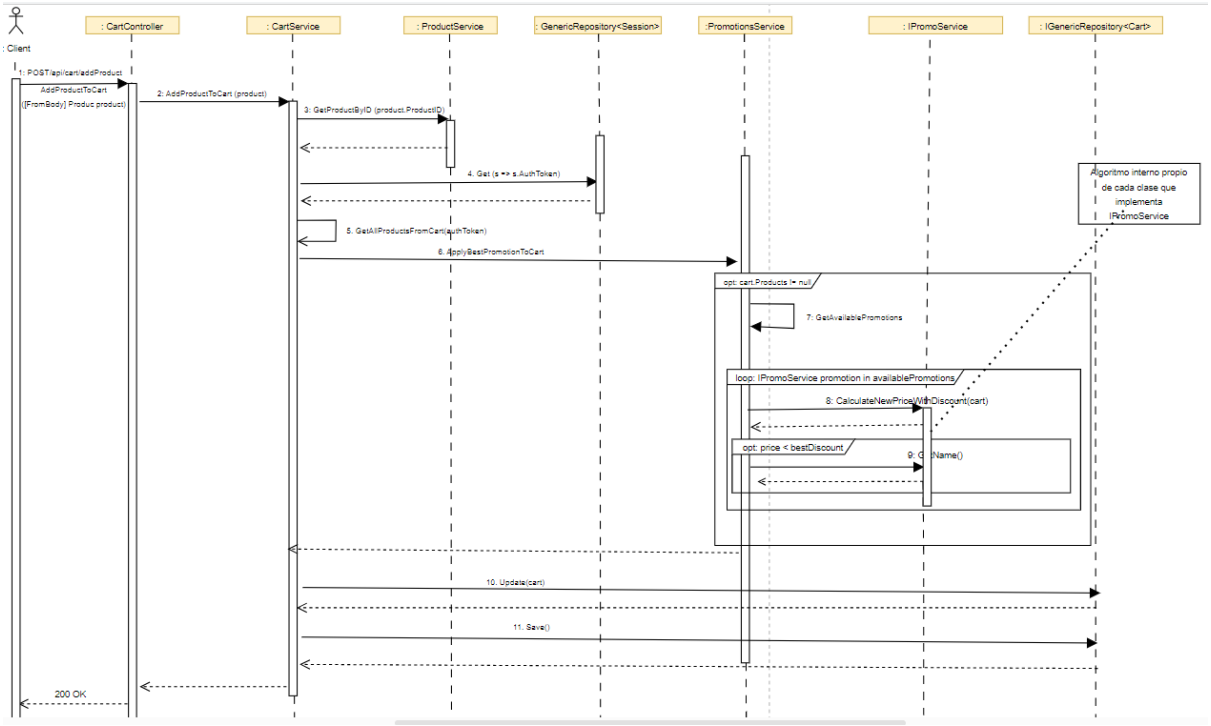
Se decidió analizar y diagramar el caso de uso de agregar un producto al carrito y cómo esta acción del usuario da comienzo a la comunicación entre las distintas capas del sistema y sus clases. Mediante el diagrama de interacción se puede visualizar todo el proceso completo desde que el cliente desea agregar un producto al carrito mediante la interfaz, traduciéndose a la solicitud de una operación POST y como a partir de eso, el

controlador del carrito en la capa de Web Api se comunica con la capa de lógica de negocio (más específicamente la clase CartService) y luego con la capa de acceso a datos. Se decidió también incluir el algoritmo para calcular la mejor promoción como proceso que sucede cada vez que se inserta o elimina un producto del carrito.

La capa de lógica de negocio se comunica con la de acceso a datos y esta con la base de datos dos veces durante este proceso: una vez cuando se agrega el producto y otra vez cuando se actualiza el carrito del usuario.

Los algoritmos específicos de cómo cada promoción calcula el nuevo precio del carrito no se incluyeron en este diagrama debido a la complejidad de los mismos y a que no resulta interesante representarlos en un diagrama de interacción ya que se ubican en clases que no interaccionan con las demás de la solución.

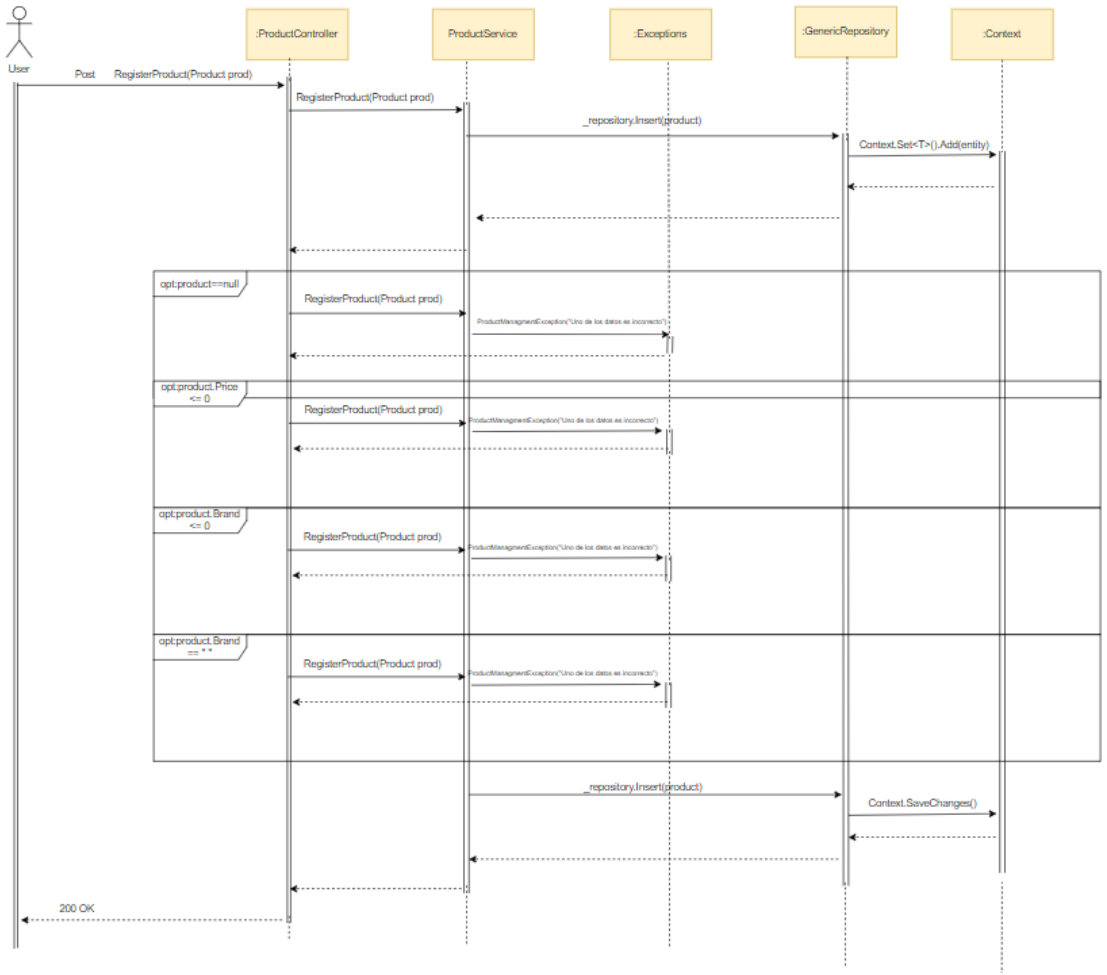
En el diagrama se muestra la interacción entre las clases para el caso en que la request es exitosa.



Registrar un producto

El segundo diagrama, se optó por realizar el caso de uso registro de un producto en el sistema, pasando por la base de datos e interactuando con las distintas capas del proyecto. Cuando el usuario realiza POST para realizar la solicitud de registro, interactuando con Web Api y las capas de BusinessLogic con DataAccess. En este caso, se muestra un ejemplo de registro válido, el cual queda persistido en la base de datos (GenericRepository e interactuando con el context). Si todos los datos del producto son correctos, se envía una respuesta 200 OK al usuario indicando que se logró registrar al producto.

Adicionalmente se muestra cómo se interactúa con casos de datos erróneos, los cuales se lanzarán excepciones indicando cuál fue la equivocación.



Actualizar datos de un usuario

El siguiente diagrama muestra el proceso de actualización de datos de un usuario, desde el momento en que el usuario realiza la solicitud, la misma comienza a comunicar las distintas capas del sistema, modifica la base de datos y finalmente se recibe una respuesta de éxito OK. Además se agrega el caso en que no se encuentra el usuario y la capa de servicio se comunica con el paquete de excepciones para devolver el error correspondiente.

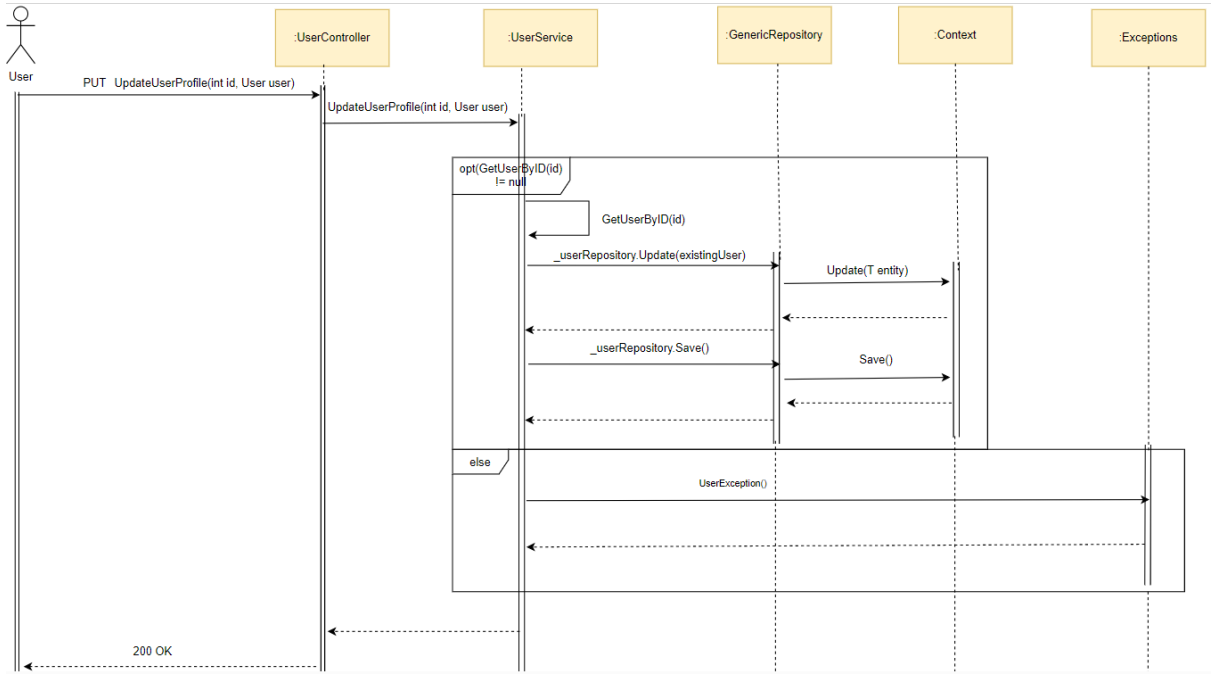


Diagrama de componentes

A continuación se muestra el diagrama de componentes de la solución, proporcionando una vista de alto nivel de la arquitectura del sistema. El mismo ayuda a comprender la organización de los módulos que componen el proyecto y cómo unos utilizan a los otros.

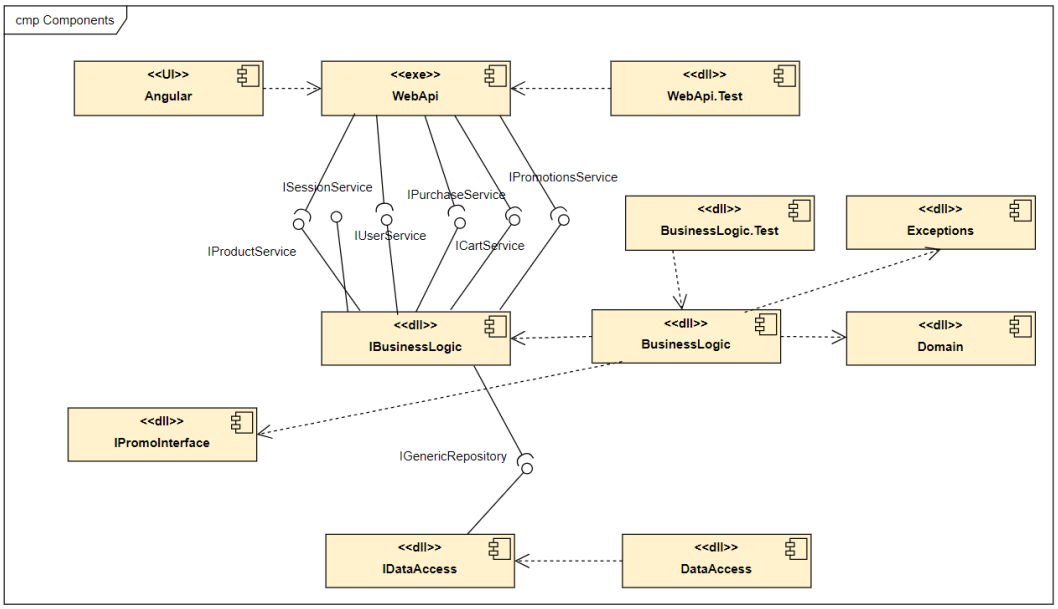
Se muestran las relaciones de dependencia entre los componentes, indicando qué componentes se basan en otros para su funcionamiento:

- La Web Api utiliza las interfaces instanciadas en IBusinessLogic para desarrollar sus operaciones de interacción con el usuario.
- IBusinessLogic hace uso de las interfaces proporcionadas por IDataAccess para establecer la conexión entre la lógica del negocio y el acceso a la base de datos.
- Tanto DataAccess como BusinessLogic dependen de los módulos que proveen las interfaces respectivas para cada uno de ellos.
- BusinessLogic también depende del dominio y las excepciones.
- Ambos módulos de Test dependen de los módulos que están testeando, en este caso, WebApi.Test de WebApi y BusinessLogic.Test de BusinessLogic.
- La UI Angular depende de la Web Api ya que necesita de las solicitudes http para mostrar los datos al usuario.

La solución se dividió en dichos componentes para mejorar la modularidad, la mantenibilidad y la escalabilidad del sistema. Esta organización nos permitió un desarrollo

más eficiente al separar las preocupaciones, ya que cada componente tiene una responsabilidad clara y definida en el proyecto.

El diagrama de componentes sirvió como herramienta de visualización de nuestro sistema para comprender bien la comunicación entre todo los módulos que por momentos se hacía medio confuso. Esto permitió un mayor comprendimiento del mismo y contribuyo al desarrollo más eficiente.



Justificación del diseño - uso de principios y patrones de diseño

Patrones de diseño

Strategy

Para la realización de este proyecto, fue utilizado uno de los patrones vistos en clase, el *patrón de diseño strategy*. Strategy nos permite determinar y controlar el intercambio de mensajes entre objetos. Específicamente con strategy buscamos que el objeto "cliente" tenga a disposición prestaciones o algoritmos (funciones) de las cuales pueda optar por implementar.

Ahora llevando este patrón a nuestro proyecto, fue utilizado para la creación y utilización de las distintas promociones (threeForTwo, TrheeForOne, ToltalLook, 20%Off y cualquier otra que se desee agregar/ eliminar a futuro), las cuales según el comportamiento que tenga la compra (Purchase) se debe implementar un tipo específico de promoción, en particular, la más conveniente.

* Dentro de la sección jerarquías de herencias, se encuentra a disposición un diagrama de su implementación con una breve descripción del comportamiento.

Fachada

Se utilizó el patrón de diseño Facade para el acceso a los servicios de la aplicación, que se encuentra en la capa de Web Api. La fachada expone una interfaz sencilla que los clientes pueden utilizar para interactuar con los distintos servicios de la aplicación, sin la necesidad de conocer los detalles internos de cómo se implementa la lógica de las operaciones se estos servicios. La fachada también es beneficiosa para simplificar el acceso a los datos de la aplicación.

Llevado a nuestro proyecto, un ejemplo de esta aplicación es UserController, el cual recibe solicitudes HTTP del cliente, se comunica con la capa de servicios (IUserService) para realizar las operaciones correspondientes (y comunicarse con la capa de acceso a datos) y devuelve una respuesta mediante código de estado. Este controlador encapsula la complejidad de la lógica de negocio y expone métodos en su interfaz pública para reflejar las distintas acciones y simplificar la interacción.

Observer

El patrón de diseño Observer desempeña un papel crucial en la arquitectura de Angular al establecer una relación de dependencia uno a muchos entre objetos. Esta relación permite que cuando un objeto experimenta un cambio de estado, todos sus dependientes sean notificados y actualizados automáticamente. En el contexto de Angular, este patrón se implementa estratégicamente en diferentes partes del proyecto, siendo un ejemplo destacado el archivo AuthGuard.

```
18
19    canActivate(
20      route: ActivatedRouteSnapshot,
21      state: RouterStateSnapshot
22    ): boolean | UrlTree | Observable<boolean | UrlTree> {
23      if (this.localStorageService.isLogged()) {
24        return this.userService.getUserFromToken().pipe(
25          map((user: User) => {
26            if (user && user.role === 'Administrador') {
27              return true;
28            } else {
29              return this.router.createUrlTree(['/inicio-sesion']);
30            }
31          })
32        );
33      }
34
35      return this.router.createUrlTree(['/inicio-sesion']);
36    }
37  }
```

En este marco, los Guards (protectores de rutas) y los interceptors (interceptores de solicitudes HTTP) actúan como observadores. Observan los cambios en las rutas y las solicitudes HTTP, respectivamente, y ejecutan acciones específicas en respuesta a estos cambios. Esta capacidad de observación y reacción contribuye significativamente a la baja acoplación de componentes en el proyecto.

Al emplear el patrón Observer, se logra una arquitectura más modular y mantenible. Esto favorece una mayor flexibilidad y evolución de componentes individuales sin comprometer la estabilidad del conjunto del sistema.

Principios a nivel de paquetes - apuntando a la cohesión

Principio de reuso común (CRP)

Respecto a los principios a nivel de paquetes en términos de cohesión, al comienzo de la creación del proyecto se buscó favorecer el *principio de reuso común (CRP)*, partiendo de la base de que las clases que pertenecen a un mismo paquete se rehúsan juntas, apuntando al reuso.

Principio de equivalencia reuso/ liberación (REP)

A partir de la segunda entrega, donde se añadieron más requerimientos, se comenzó a aplicar el *principio de equivalencia reuso/ liberación (REP)*. El tema de las promociones, que se pueden agregar o eliminar en tiempo de ejecución, nos llevó a separar cada una de las promociones en paquetes distintos para permitir liberarlas por separado. Esto hace que no afecte a otras clases del proyecto, minimizando el impacto del cambio si se quiere eliminar promociones existentes, solo afectando las clases que estén en el paquete o que las usan (ninguna clase usa directamente las promociones ya que se levantan los dll disponibles en la carpeta Promotions a la hora hacer uso de las promociones disponibles).

Principio de clausura común (CCP)

Por último, se partió de la base de que las clases candidatas a cambiar por el mismo motivo se deben agrupar en un mismo paquete de forma de minimizar el impacto del cambio, siguiendo el *principio de clausura común (CCP)*. Esto apunta al mantenimiento del código. Por ejemplo: al agregar una nueva entity, la coloco en el mismo paquete donde están las demás. Al agregar un nuevo controlador, se ubica junto a los demás de la solución.

Principios a nivel de paquetes - apuntando al acoplamiento

Principio de dependencias cíclicas (ADP)

Se siguió este principio a la hora de diseñar el esquema de la solución, creando proyectos que no presentan dependencias cíclicas entre sí. La estructura de dependencias entre paquetes forma un grafo dirigido y no cíclico.

Principio de dependencias estables (SDP)

Aplicamos este principio al establecer que los paquetes más inestables (DataAccess y BusinessLogic) dependan de sus respectivas interfaces, siendo estas más estables (IDataAccess y IBusinessLogic). Además, estas interfaces se comunican con Domain que presenta gran estabilidad. Llamamos inestables a los paquetes que presentan mayor cantidad de dependencias hacia otros paquetes.

Principio de abstracciones estables (SAP)

Dado un paquete abstracto, es probable que otros paquetes lo utilicen ya sea para implementarlo o extenderlo. Se aplica al establecer que los paquetes concretos que implementan los paquetes estables son menos estables. Llevado a nuestro proyecto, IBusinessLogic es más estable que BusinessLogic.

Principios SOLID

Por otro lado, se siguieron los principios SOLID. Se respetó el *Single Responsibility Principle* asignando una única responsabilidad a cada clase y manteniendo así una única razón por la cual cada una debería cambiar. El *Open Close Principle*, fue aplicado también, visible en el tema de las promociones, creando un código que es totalmente abierto a la extensión permitiendo que se agreguen futuras promociones y cerrado a la modificación ya que no es necesario cambiar el código existente para extender la cantidad de promociones. En el mismo contexto de las promociones, se respeta *Liskov Substitution Principle*, permitiendo que las clases hijas (las 4 clases que hasta el momento implementan los servicios de las promociones) cumplan con las especificaciones del padre (interfaz). Todas implementan codigo sin violar el contrato que establece la clase padre. Respecto a *Interface Segregation Principle*, en este trabajo se tuvo especial atención a ese principio, generando una gran colección de interfaces, cada una personalizada para cada entidad/ necesidad del cliente. Por último, el *Dependency Inversion Principle* también se cumple generando una abstracción entre los módulos de bajo nivel y los de alto nivel. La lógica de negocio se comunica con la lógica de acceso a datos a partir de interfaces, de manera que no dependa directamente de los detalles de implementación.

Consideraciones sobre el obligatorio:

- Se interpretó que los productos podían tener un color en string.
- El atributo category, es registrado como int, ya que hacen referencia al ID en la base de datos. A modo de ejemplo una category de valor 1 tendrá en la base de datos ID con valor 1 y nombre de la categoría inalámbricos.
- Al insertar objetos en la base de datos mediante Postman, no se debería mandar como atributo el ID de esos objetos ya que la base lo autogenera.

Se optó por la utilización de excepciones, para continuar con las buenas prácticas de Clean Code. Para ello, se creó el paquete Excepciones, el cual contiene un conjunto de ellas, pensadas para implementar en distintos sectores del proyecto. Fueron creadas excepciones para los usuarios, los productos, las compras, etc, las cuales son utilizadas en varios sectores del código, como puede ser BusinessLogic, DataAccess y WebApi. La variación que se tiene en cada excepción, es el mensaje que este devuelve, dependiendo para que se desee implementar (capturar una acción específica para devolver un mensaje específico).

Mecanismos de extensibilidad

Para esta entrega abordó el tema de la extensibilidad con respecto a las promociones. Se diseñó una solución que permite, en tiempo de ejecución agregar y eliminar promociones al sistema según las necesidades del momento. Esto es un aspecto clave ya que nuestro e commerce puede adaptarse a distintas épocas del año donde las promociones existentes van variando y no es necesario volver a compilar todo el proyecto para modificarlas. Si se quiere agregar una nueva promoción, únicamente se tiene que incluir el dll de la misma en la carpeta Promotions de Obligatorio1.WebApi y el sistema la

reconoce como disponible. De lo contrario, si se quiere quitar algunas de las actuales, tan solo con remover el dll de dicha carpeta será removida.

El diseño del código también contempla casos en que en la carpeta Promotions se agreguen archivos que no refieran a promociones, en esos casos no tomándose en cuenta. Para que un archivo dll sea considerado como promoción, la clase deberá de implementar la interfaz común de las promociones: IPromoService.

El siguiente método ubicado en PromotionsService de BusinessLogic es el encargado de obtener los archivos del directorio donde se encuentra la carpeta Promotions, analizar cuales son promociones y retornarlas como una lista.

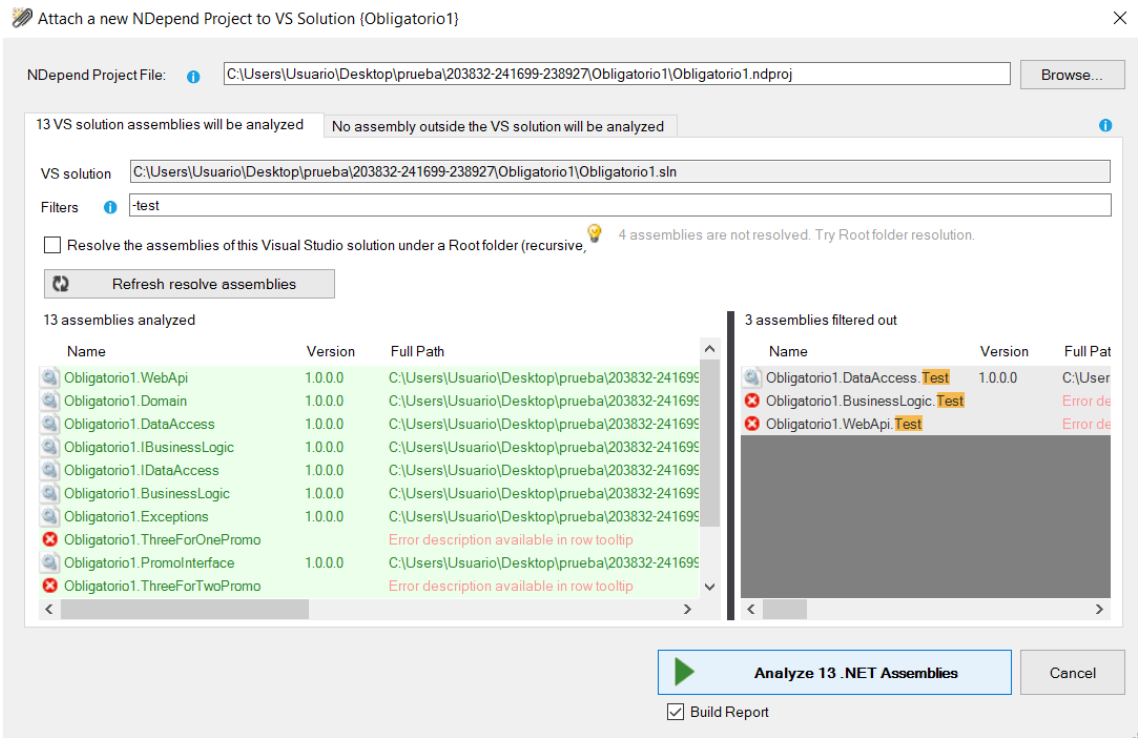
```
3 referencias
public List<IPromoService> GetPromotionsAvailable()
{
    List<IPromoService> availablePromotions = new List<IPromoService>();
    string promosPath = "./Promotions";
    string[] filePaths = Directory.GetFiles(promosPath);

    foreach (string filePath in filePaths)
    {
        if (filePath.EndsWith(".dll"))
        {
            FileInfo fileInfo = new FileInfo(filePath);
            AssemblyLoadContext context = new AssemblyLoadContext(null, true);
            Assembly assembly = context.LoadFromAssemblyPath(fileInfo.FullName);

            foreach (Type type in assembly.GetTypes())
            {
                if (typeof(IPromoService).IsAssignableFrom(type) && !type.IsInterface)
                {
                    IPromoService promotion = (IPromoService)Activator.CreateInstance(type);
                    if (promotion != null)
                        availablePromotions.Add(promotion);
                }
            }
            context.Unload();
        }
    }
    return availablePromotions;
}
```

Breve explicación de los pasos realizados:

1. Se crea una lista donde se van a almacenar las promociones disponibles.
2. Mediante Directory.GetFiles, se obtienen todos los archivos encontrados en la carpeta Promotions de Obligatorio1.WebApi.
3. Para cada uno de esos archivos se evalúa, si tiene como extension .dll:
    a. Se obtiene la información del archivo mediante FileInfo()
    b. Mediante assembly se accede al código del archivo y si el tipo implementa IPromoService (es decir, es una promoción), mediante Activator.CreateInstance se crea una instancia de IPromoService.
    c. Si esta instancia creada no es nula, se agrega a la lista creada inicialmente
4. Se retorna la lista de promociones disponibles.

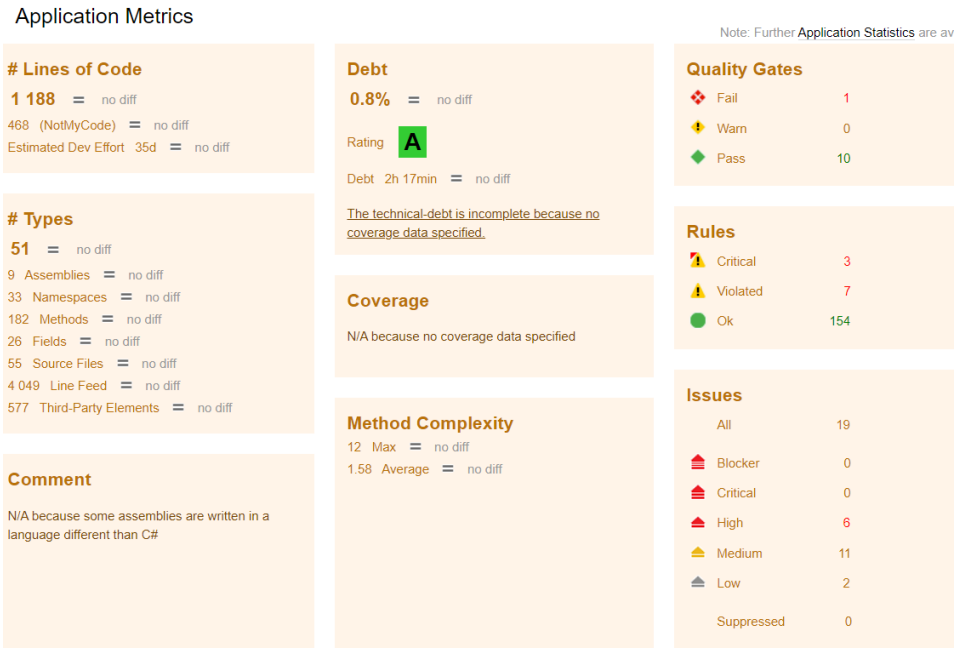
Calidad del diseño en base a métricas

Para la calidad de diseño en base a métricas, nos apoyamos en la herramienta NDepend, la cual es instalada en Visual Studio.



Luego de la instalación, corrimos la herramienta para que nos realizará un informe sobre las métricas y dependencias de nuestro proyecto. En el anexo se incluye el informe obtenido.

Los resultados fueron los siguientes:



Pasaremos a analizar el diagrama de abstracción e inestabilidad (Imagen 2 del anexo).
Una de las primeras conclusiones que obtenemos es sobre el Domain, el cual está muy próximo a la zona de dolor, siendo poco inestable y cero abstracto. Domain es concreto y no depende de nadie (otros paquetes dependen de él).

Con respecto a la métrica H (Relational Cohesion), observamos que la mayoría se encuentra en el rango deseado de 1.4-4.0, exceptuando las clases IDataAccess, PromoInterface, IBusinessLogic y ServiceFactory1 (imagen 4). Si bien no se encuentran dentro del rango deseado, la relación entre las clases de estos paquetes, no están tan lejanas a 1.4, por lo que se podría mejorar a futuro el relacionamiento de las clases dentro de un mismo paquete.

Una razón por la cual la “Relational Cohesion” se encuentra dentro del rango previsto, es por la implementación de agrupaciones por responsabilidades de clases y no de paquetes, siendo un ejemplo de esto DataAccess(relaciones por acceso a datos) y/o BusinessLogic (lógica de negocio).

En relación a la medición de la estabilidad, se busca que una clase o paquete sea lo más estable posible, para afectar en menor medida a otros elementos que puedan depender de él. Nuestro proyecto presenta buenos números, con la excepción de algunos casos como lo es el de WebApi, el cual este paquete no depende de otros, pero son varios los que dependen de él.

Otras mejoras serían las de DataAccess y BusinessLogic, los cuales también presentan un gran número de dependencias hacia ellos.

Por parte de la abstracción, uno de los principios a seguir, fue el de DIP, el cual nos dice que los módulos de alto nivel no deben depender de los módulos de bajo nivel, sino que ambos deben depender de abstracciones. Para esto fueron implementados los distintos paquetes de interfaces, a secciones del código (IBusinessLogic, IDataAccess por ejemplo).

Un cambio que hubiese mejorado los resultados de nuestro proyecto, es implementar esta última, para mejorar la estabilidad del paquete, ya que según los resultados arrojados por NDepend, aún existe un márgen de mejora (Abstractness).

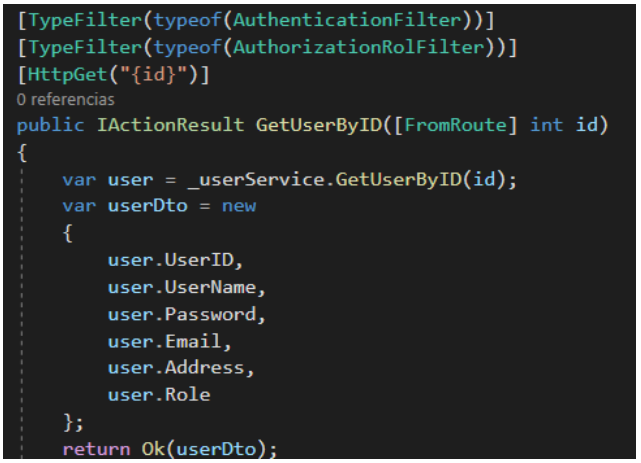
Como conclusión, si bien se pueden seguir implementando mejoras para concretar mejores resultados, se lograron obtener resultados muy decentes. Uno de los puntos importantes a mencionar, es el evitar que nuestros paquetes sean muy rígidos, concretos estables, condición que nos llevaría a la zona de dolor (no hubo paquetes en esta zona), al igual de la zona de poco utilidad la cual los paquetes son abstractos y nadie depende de ellos.

Mejoras al diseño

Respecto a la primera entrega, se realizaron varios cambios y mejoras en términos de diseño y aplicación de patrones a la solución, así obteniendo un código más estable y mantenible. Además, se pulieron ciertos aspectos para completar la totalidad de los requerimientos de la primera entrega, sumados a los nuevos.

- Filters

Para esta segunda entrega, logramos optimizar nuestro código mediante la implementación de filters. Esta aplicación se realizó implementando distintos tipos de filters, según su utilización y las necesidades de las solicitudes: autorización, autenticación, excepción. Los filters nos permitieron modificar el comportamiento de las solicitudes HTTP antes o después de que esta ocurra. A modo de ejemplo, a continuación se muestra un ejemplo mediante una captura de la implementación de los filtros Authentication y Authorization para la solicitud de la obtención de usuarios por ID.



```
[TypeFilter(typeof(AuthenticationFilter))]
[TypeFilter(typeof(AuthorizationRolFilter))]
[HttpGet("{id}")]
0 referencias
public IActionResult GetUserByID([FromRoute] int id)
{
    var user = _userService.GetUserByID(id);
    var userDto = new
    {
        user.UserID,
        user.UserName,
        user.Password,
        user.Email,
        user.Address,
        user.Role
    };
    return Ok(userDto);
```

Los filters se aplican a nivel de clase, para los casos en que se utilizan en todos los métodos de la clase, o a nivel de método para casos más específicos.

- Sesiones de usuarios

Otra mejora de diseño con respecto al obligatorio pasado, es la utilización de sesiones de usuarios, las cuales son asignadas para los usuarios que están interactuando con la aplicación. Uno de los elementos utilizados para esto son los Tokens, los cuales nos ayudan a identificar a cada una de las sesiones (útil para los roles administrador y comprador), ya que según el tipo de sesión y si pertenece a un usuario comprador o administrador se pueden realizar distintas acciones. Este cambio permitió la mejora del diseño y la seguridad.

- Manejo de las promociones

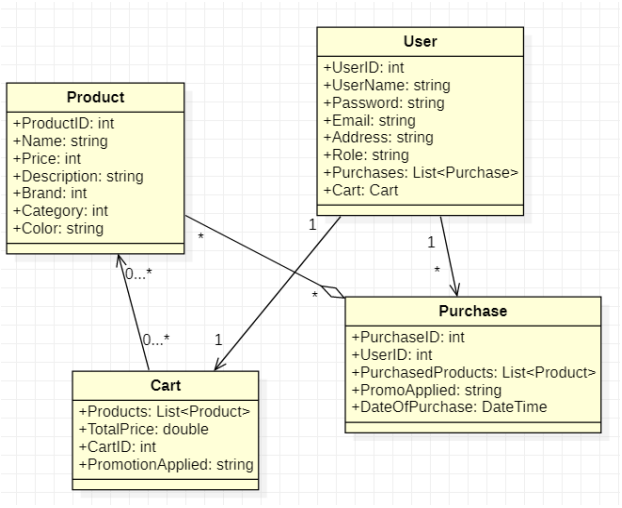
Respecto a las promociones se realizaron grandes mejoras en el diseño de las mismas. Por un lado, se extrajeron métodos en común de las mismas, los cuales fueron alojados en una nueva clase llamada PromoUtility dentro del paquete PromoInterface. Esta nueva clase es genérica a las promociones y permite la reutilización de código, evitando la redundancia del mismo que se había generado para la primera entrega.

Por otro lado, para permitir la extensibilidad del sistema en términos de flexibilidad de promociones disponibles, se decidió modificar la estructura de paquetes de la solución. Las clases de las promociones que antes estaban todas alojadas en el paquete BusinessLogic, pasan a ser ahora cada una un nuevo paquete. Esto permite que cada una de ellas se pueda liberar individualmente y sin que otras clases dependan de ellas para poder ser manipuladas en tiempo de ejecución según se quiera agregarlas al sistema o eliminarlas. Esta decisión se tomó a modo de respetar el principio de equivalencia reuso/ liberación de paquetes. La interfaz que todas las clases implementan también se encuentra en un nuevo proyecto, junto a la clase PromoUtility.

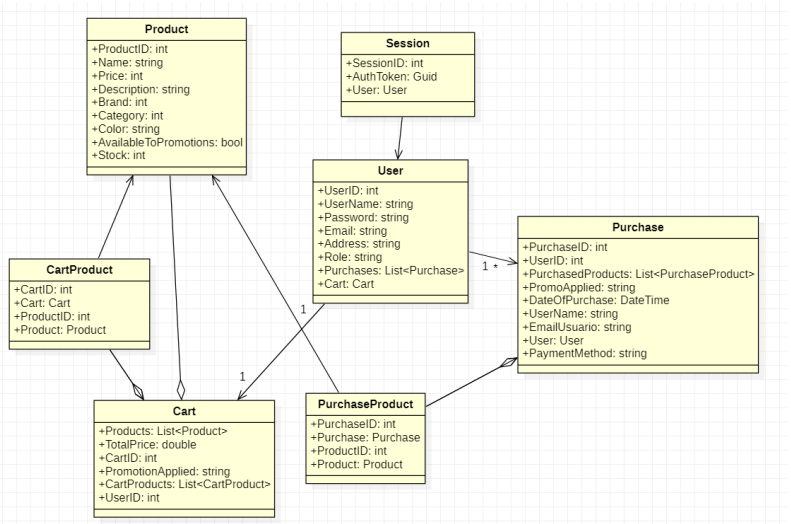
- Relaciones en tablas

En esta nueva instancia, se optó por realizar un nuevo diseño de relaciones del dominio, buscando que sea más efectiva la implementación y la lógica de los objetos.

Diseño entrega 1

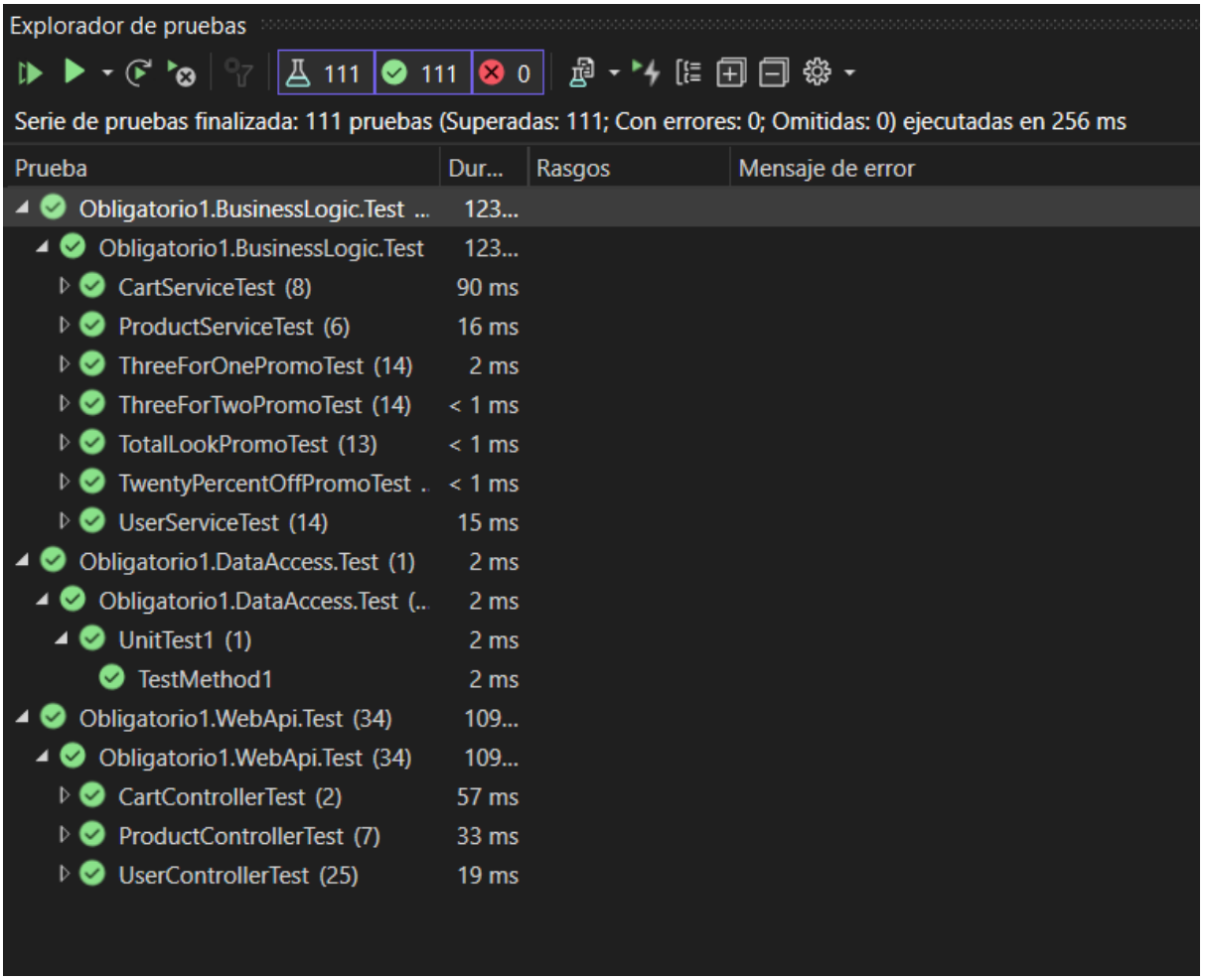


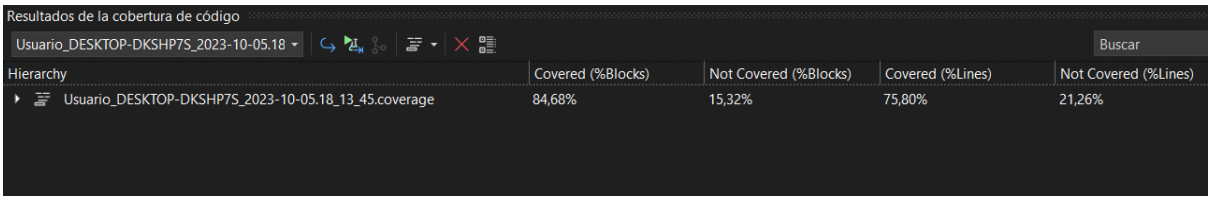
Diseño entrega 2



Anexo

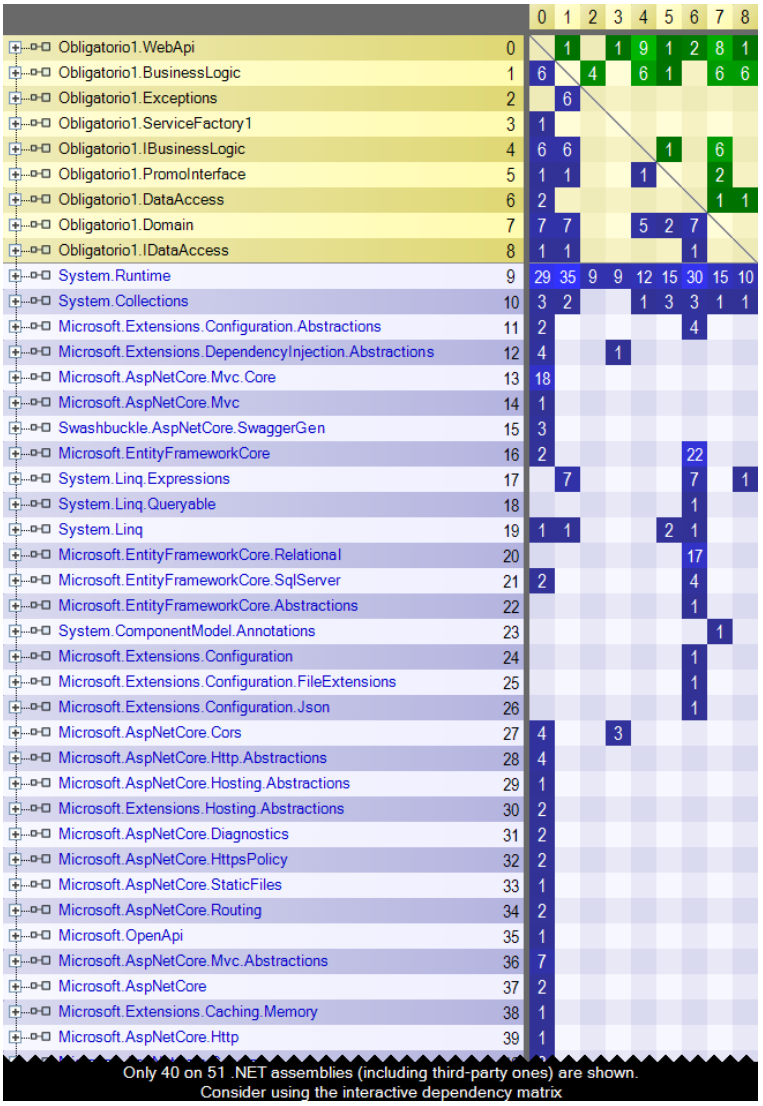
Informe de cobertura





Informe de métricas obtenido mediante NDepend

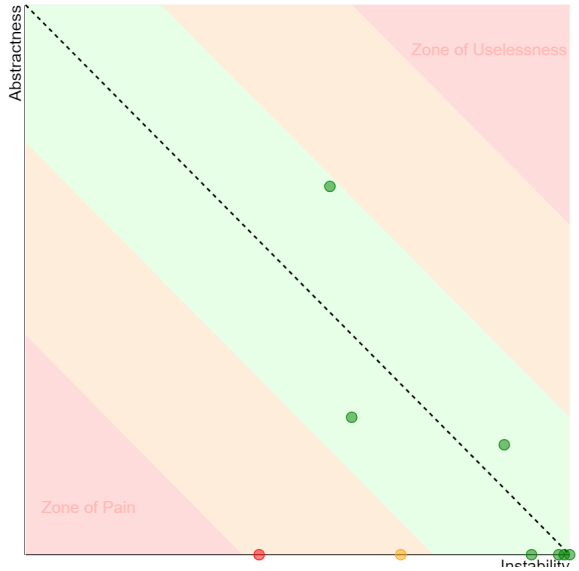
Imagen 1



		0	1	2	3	4	5	6	7	8
Obligatorio1.WebApi	0		1		1	9	1	2	8	1
Obligatorio1.BusinessLogic	1	6		4		6	1		6	6
Obligatorio1.Exceptions	2		6							
Obligatorio1.ServiceFactory1	3	1								
Obligatorio1.IBusinessLogic	4	6	6				1		6	
Obligatorio1.PromoInterface	5	1	1			1			2	
Obligatorio1.DataAccess	6	2							1	1
Obligatorio1.Domain	7	7	7			5	2	7		
Obligatorio1.IDataAccess	8	1	1					1		
System.Runtime	9	29	35	9	9	12	15	30	15	10
System.Collections	10	3	2			1	3	3	1	1
Microsoft.Extensions.Configuration.Abstractions	11	2						4		
Microsoft.Extensions.DependencyInjection.Abstractions	12	4			1					
Microsoft.AspNetCore.Mvc.Core	13	18								
Microsoft.AspNetCore.Mvc	14	1								
Swashbuckle.AspNetCore.SwaggerGen	15	3								
Microsoft.EntityFrameworkCore	16	2						22		
System.Linq.Expressions	17		7					7		1
System.Linq.Queryable	18							1		
System.Linq	19	1	1				2	1		
Microsoft.EntityFrameworkCore.Relational	20							17		
Microsoft.EntityFrameworkCore.SqlServer	21	2						4		
Microsoft.EntityFrameworkCore.Abstractions	22							1		
System.ComponentModel.Annotations	23								1	
Microsoft.Extensions.Configuration	24							1		
Microsoft.Extensions.Configuration.FileExtensions	25							1		
Microsoft.Extensions.Configuration.Json	26							1		
Microsoft.AspNetCore.Cors	27	4			3					
Microsoft.AspNetCore.Http.Abstractions	28	4								
Microsoft.AspNetCore.Hosting.Abstractions	29	1								
Microsoft.Extensions.Hosting.Abstractions	30	2								
Microsoft.AspNetCore.Diagnostics	31	2								
Microsoft.AspNetCore.HttpsPolicy	32	2								
Microsoft.AspNetCore.StaticFiles	33	1								
Microsoft.AspNetCore.Routing	34	2								
Microsoft.OpenApi	35	1								
Microsoft.AspNetCore.Mvc.Abstractions	36	7								
Microsoft.AspNetCore	37	2								
Microsoft.Extensions.Caching.Memory	38	1								
Microsoft.AspNetCore.Http	39	1								

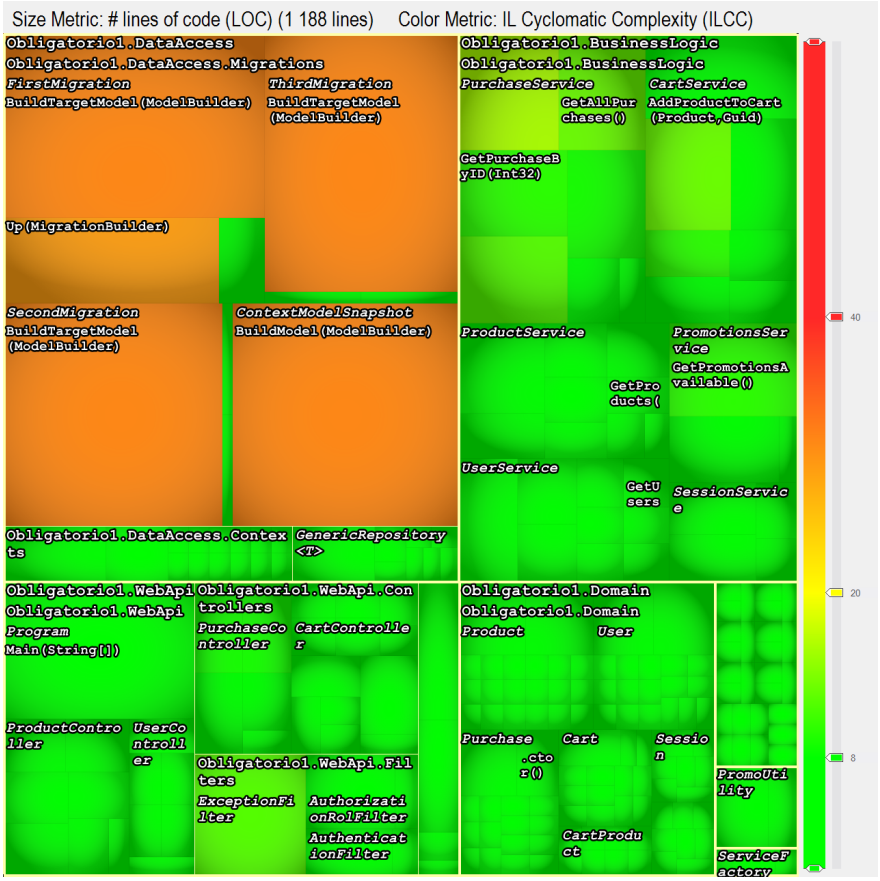
Only 40 on 51 .NET assemblies (including third-party ones) are shown.
Consider using the interactive dependency matrix

Imagen 2



	Abstractness	Instability	Dist
Domain	0	0.43	0.41
Exceptions	0	0.69	0.22
IDataAccess	0.25	0.6	0.11
ServiceFactory1	0	0.94	0.05
DataAccess	0	0.98	0.01
BusinessLogic	0	0.99	0.01
WebApi	0	1	0
PromoInterface	0.2	0.88	0.06
IBusinessLogic	0.67	0.56	0.16

Imagen 3



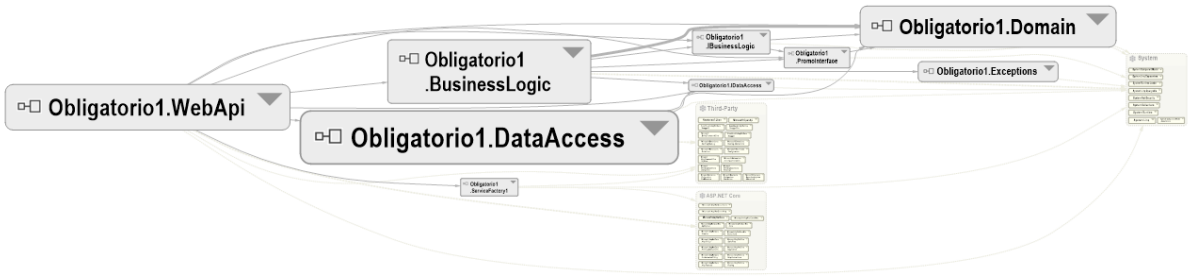


Imagen 4

Assemblies	# lines of code	# IL instruction	# Types	# Abstract Types	# lines of comment	% Comment	% Coverage	Afferent Coupling	Efferent Coupling
Obligatorio1.IDataAccess v1.0.0.0	0	30	4	1	-	-	-	8	12
Obligatorio1.Domain v1.0.0.0	134	552	10	0	0	0	-	23	17
Obligatorio1.DataAccess v1.0.0.0	443	6814	17	0	20	4.32	-	2	101
Obligatorio1.PromoInterface v1.0.0.0	12	111	5	1	0	0	-	3	22
Obligatorio1.IBusinessLogic v1.0.0.0	0	30	9	6	1	100	-	15	19
Obligatorio1.ServiceFactory1 v1.0.0.0	4	77	4	0	4	50	-	1	13
Obligatorio1.Exceptions v1.0.0.0	27	192	14	0	0	0	-	4	9
Obligatorio1.BusinessLogic v1.0.0.0	330	2550	9	0	10	2.94	-	1	68
Obligatorio1.WebApi v1.0.0.0	238	2430	21	0	181	43.2	-	0	134

Relational Cohesion	Instability	Abstractness	Distance
1	0.6	0.25	0.11
2.9	0.43	0	0.41
1.82	0.98	0	0.01
1.2	0.88	0.2	0.06
1	0.56	0.67	0.16
1	0.93	0	0.05
1.57	0.69	0	0.22
1.67	0.99	0	0.01
2.86	1	0	0

Arquitectura Cliente-Servidor en el Contexto de Angular y .NET:

1. **Cliente (Angular):**

Descripción:

- La interfaz de usuario y la lógica de presentación residen en la aplicación Angular.
- El cliente interactúa directamente con el usuario y se comunica con el servidor para obtener o enviar datos.

Funciones Principales:

- Representación y presentación de datos.
- Interacción con el usuario.
- Realización de solicitudes al servidor a través de servicios HTTP.

Herramientas:

- Angular para el desarrollo del frontend.
- Postman (opcionalmente) para probar y validar solicitudes HTTP desde el lado del cliente.

2. **Servidor (.NET):**

Descripción:

- El backend está construido con tecnologías .NET.
- Contiene la lógica de negocio, gestión de datos y servicios para atender las solicitudes del cliente.

Funciones Principales:

- Procesamiento de solicitudes del cliente.
- Acceso y manipulación de datos.
- Implementación de la lógica de negocio.

Herramientas:

- .NET para el desarrollo del backend.
- Postman (opcionalmente) para probar y validar las API desde el lado del servidor.

3. **Comunicación Cliente-Servido**r:

Descripción:

- Las solicitudes y respuestas entre el cliente y el servidor siguen el protocolo HTTP/HTTPS.
- Funciones Principales:
- El cliente envía solicitudes al servidor para obtener o enviar datos.
- El servidor procesa estas solicitudes y envía respuestas al cliente.

Herramientas:

- Angular utiliza servicios HTTP para realizar solicitudes desde el cliente.

Diseño de la API

Para el diseño de la APi nos basamos en los siguientes criterios y principios REST. Se adjuntan recortes de nuestro código para demostrar las decisiones implementadas.

- Buenas Prácticas

El diseño de la API sigue las mejores prácticas de diseño de API REST de la industria. Esto garantiza que la API sea coherente, escalable y fácil de mantener a medida que evoluciona con las necesidades del sistema. Se han evitado redirecciones (códigos de estado 3xx) ya que no son necesarias en el contexto de la API, lo que contribuye a la eficiencia en el tráfico de datos.

- Recursos Bien Definidos

Cada recurso se representa como una URL única y está asociado con un conjunto de operaciones permitidas (GET, POST, PUT, DELETE).

- Comunicación Sin Estado

Cada solicitud HTTP contiene toda la información necesaria para comprender y procesar la solicitud, lo que hace que la API sea independiente del estado del servidor.

- Verbos HTTP Apropiados

Se utilizan los verbos HTTP estándar para indicar la acción deseada en cada solicitud (GET para lecturas, POST para creaciones, PUT para actualizaciones, DELETE para eliminaciones.

Ejemplo de una solicitud POST, para crear un usuario.

```
33      [HttpPost]
        2 referencias
34      public IActionResult RegisterUser([FromBody] User user)
35      {
```

Ejemplo de una solicitud GET, para obtener el producto con cierto ID.

```
57      [HttpGet("{id}")]
        0 referencias
58      public IActionResult GetProductByID([FromRoute] int id)
59      {
```

Ejemplo de una solicitud PUT, para actualizar los datos de un usuario.

```
399     [HttpPut("UpdateUserProfile")]
        2 referencias
400     public IActionResult UpdateUserProfile([FromBody] User user)
401     {
```

Ejemplo de una solicitud DELETE, para eliminar un producto del carrito.

```
74      [HttpDelete("deleteProduct")]
        1 referencia
75      public IActionResult DeleteProductFromCart([FromBody] Product product)
76      {
```

- Respuestas Coherentes

Se utilizan códigos de estado HTTP adecuados en las respuestas para indicar el resultado de cada solicitud.
Algunos ejemplos:

```
return BadRequest($"Error al obtener el historial de compras: {ex.Message}");
```

```
return NotFound(new { Message = "There are no purchases" });
```

```
return Ok(purchases);
```

```
return CreatedAtAction(nameof(GetUserByID), new { id = createdUser.UserID }, createdUser);
```

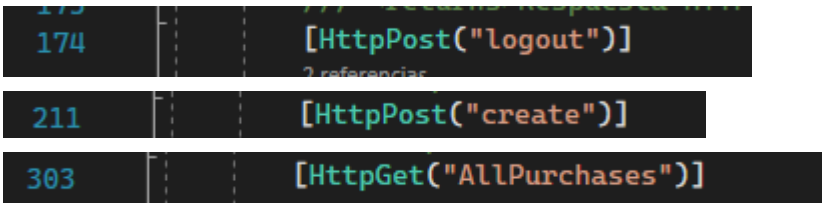
- Convención de Rutas

Se ha seguido una convención de nomenclatura clara y uniforme para las rutas de acceso a los recursos. Por ejemplo, los usuarios se acceden a través de la ruta /api/users, lo que proporciona una organización lógica y predecible.

```
14      [Route("api/users")]
```

Las rutas de acceso a las distintas operaciones también respetan esta convención:

```
136     [HttpPost("login")]
```



Además, se mejoraron aspectos respecto a la primer entrega en términos de especificación de ruta (ejemplo: api/users/{id}/getUserByID en vez de api/users/getUserByID/{id})

Mecanismo de autenticación

La autenticación de solicitudes a la API ha experimentado una actualización significativa en la forma en que se lleva a cabo. Ahora, nuestros servicios requieren la presentación de un token de autenticación obtenido mediante una sesión válida. Este token actúa como un identificador seguro que se utiliza para validar la autenticidad del usuario y otorgar acceso a las funcionalidades protegidas.

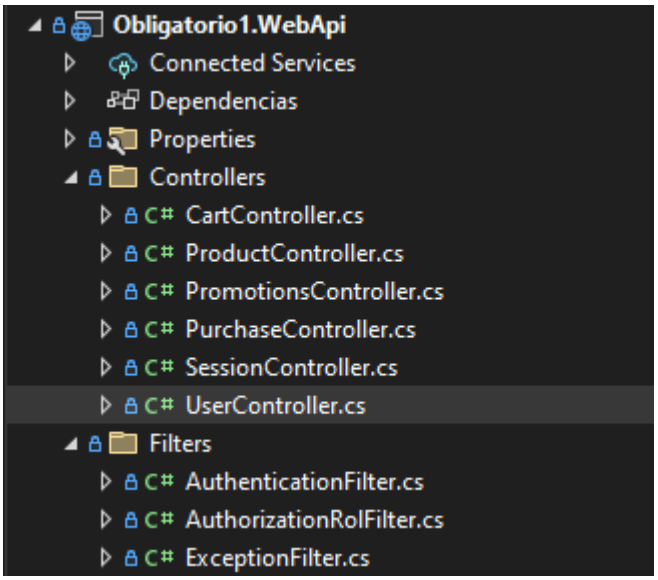
Con la implementación del nuevo sistema, los usuarios deben obtener un token de autenticación válido a través de su sesión activa. Este token se utiliza para autorizar las solicitudes y garantizar que solo los usuarios autenticados y autorizados tengan acceso a ciertas operaciones.

Adicionalmente, se han incorporado filtros de seguridad para verificar los roles de los usuarios, asegurando así que solo aquellos con los roles adecuados puedan realizar determinadas acciones. Por ejemplo, acciones administrativas como la creación, eliminación y modificación de productos, o la visualización de todas las compras realizadas, están ahora restringidas a usuarios con el rol "Administrador".

A continuación, se presenta un ejemplo claro de cómo se realiza la verificación de roles en una solicitud para eliminar un usuario. La identidad del usuario se valida para permitir la eliminación solo si el rol es "Administrador" (línea 268). En caso de no cumplir con esta condición, se devuelve un mensaje de error indicando que el usuario no tiene permiso para llevar a cabo la operación de eliminación de usuarios.

```
90
91          /// <summary>
92          /// Delete a user registered in the system by their ID.
93          /// </summary>
94          /// <remarks>
95          /// Allows a user with administrator permissions to delete a user by user ID.
96          /// </remarks>
97          /// <param name="id">The ID of the user to delete.</param>
98          /// <response code="204">The user was successfully deleted.</response>
99          /// <response code="404">The user with the specified ID was not found.</response>
100         /// <response code="400">Error in the request or deleting the user.</response>
101         [TypeFilter(typeof(AuthenticationFilter))]
102         [TypeFilter(typeof(AuthorizationRolFilter))]
103         [HttpDelete("{id}")]
            0 referencias | fiioro25, Hace 4 días | 2 autores, 9 cambios
104         public IActionResult DeleteUser([FromRoute] int id)
105         {
106             try
107             {
108                 _userService.DeleteUser(id);
109                 return Ok(new { message = "User successfully deleted." });
110             }
111             catch (Exception ex)
112             {
113                 return BadRequest(new { message = "Error deleting user: " + ex.Message });
114             }
115         }
```

Evidencia de los filtros utilizados:



Ejemplo de sesiones en la BD:

	SessionID	AuthToken	UserID
1	1	E195860F-0B29-4C32-B2BB-BC5D30850F1F	1
2	2	ADAC4EAB-31EE-4F04-8C31-6200092CCB2E	2
3	3	96534D1C-DAF2-46CF-9BF4-ACEEFFAEBA62	1
4	4	EF5D575D-9616-4798-B437-CAD7D2DAF705	2
5	5	1B9FAC5B-0C5A-42A6-9D92-927BD8019E8E	1
6	6	B4776145-F471-4004-AD70-7F71A0C22F14	1

Códigos de Estado

La API utiliza códigos de estado HTTP estándar en sus respuestas para comunicar el resultado de las solicitudes. Estos códigos de estado han sido seleccionados

cuidadosamente para proporcionar información significativa sobre los errores o éxitos en las operaciones. Esto facilita la comprensión de errores en el lado del cliente y promueve una comunicación efectiva entre el servidor y los clientes.

Se decidió restringir la variedad el uso de los códigos de estado a 4 o 5 a modo de comprender entre los integrantes del equipo el uso concreto que le estamos dando a cada uno y evitar situaciones de confusión. Apuntamos a una comunicación clara y concisa. Estos son los códigos utilizados y sus significados:

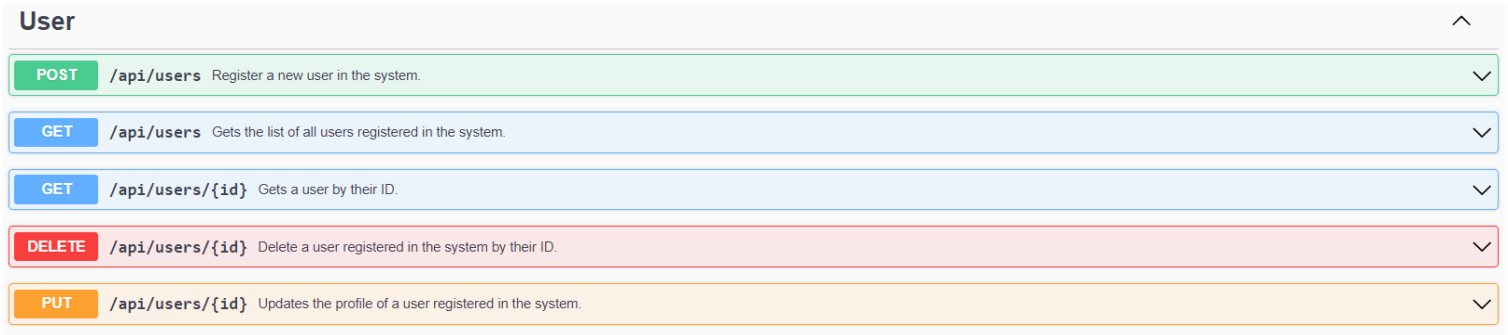
- 200 OK: Operación exitosa.
- 201 Created: Recurso creado exitosamente.
- 400 Bad Request: Solicitud incorrecta o error en el procesamiento.
- 404 Not Found: Recurso no encontrado.
- 500 Internal Server Error: Error interno del servidor.

Recursos de la API

USER

URL Base	/api/users				
Resource	User				
Description	Representa a un usuario registrado en el sistema.				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	POST	/api/users	Registra un nuevo usuario en el sistema.	*User:* Cuerpo de la solicitud (Datos del usuario a registrar)	200 OK: Registro exitoso.
	GET	/api/users	Obtiene la lista de todos los usuarios registrados en el sistema.	-	200 OK: Lista de usuarios.
	GET	/api/users /{id}	Obtiene un usuario por su ID.	*id:* En la ruta (ID del usuario)	200 OK: Usuario encontrado.
	DELETE	/api/users /{id}	Elimina un usuario por su ID.	*id:* En la ruta (ID del usuario)	200 OK: Usuario eliminado.
	PUT	/api/users /{id}	Actualiza el perfil de un usuario registrado.	*id:* En la ruta (ID del usuario), *user:* cuerpo de la solicitud (usuario	200 OK: Usuario modificado.

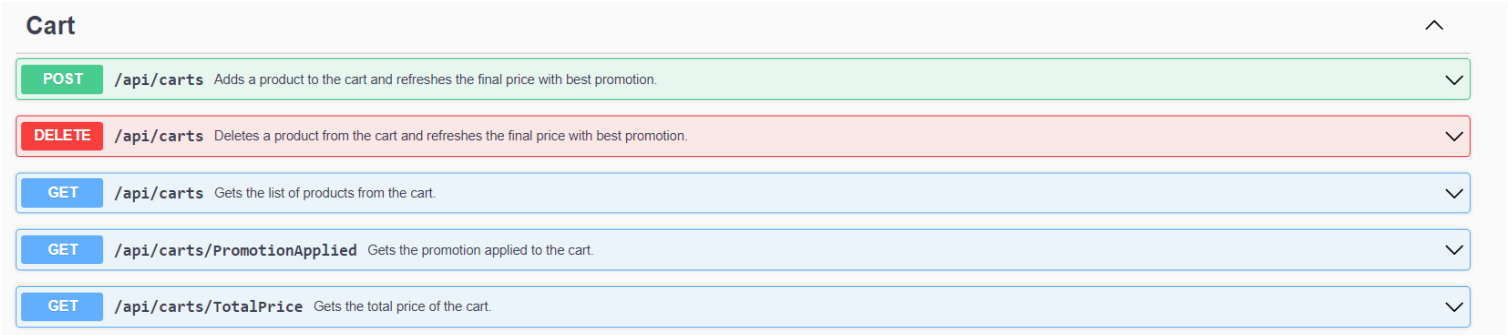
				con datos actualizados)	



CART

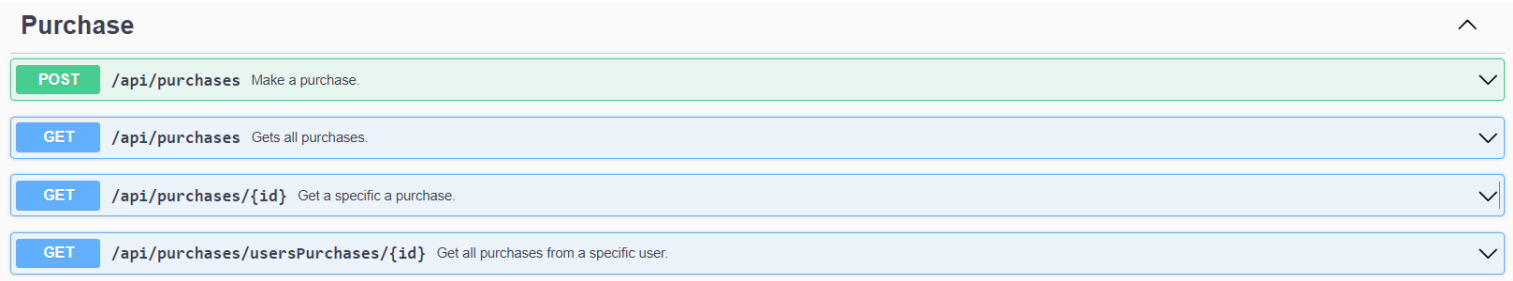
URL Base	/api/carts				
Resource	Cart				
Description	Representa el carrito del usuario registrado en el sistema.				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	POST	/api/carts/addProduct	Agrega un producto al carrito del usuario logueado.	*product:* Cuerpo de la solicitud (Datos del producto a agregar)	200 OK: Producto agregado exitosamente
	DELETE	/api/carts/deleteProduct	Elimina un producto del carrito del usuario logueado.	*product:* Cuerpo de la solicitud (Datos del producto a eliminar)	200 OK: Producto eliminado exitosamente
	GET	/api/carts	Obtiene los productos del carrito del usuario logueado.	-	200 OK: Productos obtenidos
	GET	/api/carts/PromotionApplied	Obtiene la promoción aplicada al carrito del usuario logueado.	-	200 OK: Promoción obtenida

	GET	/api/carts/ TotalPrice	Obtiene el precio total del carrito del usuario logueado.	-	200 OK: Precio obtenido



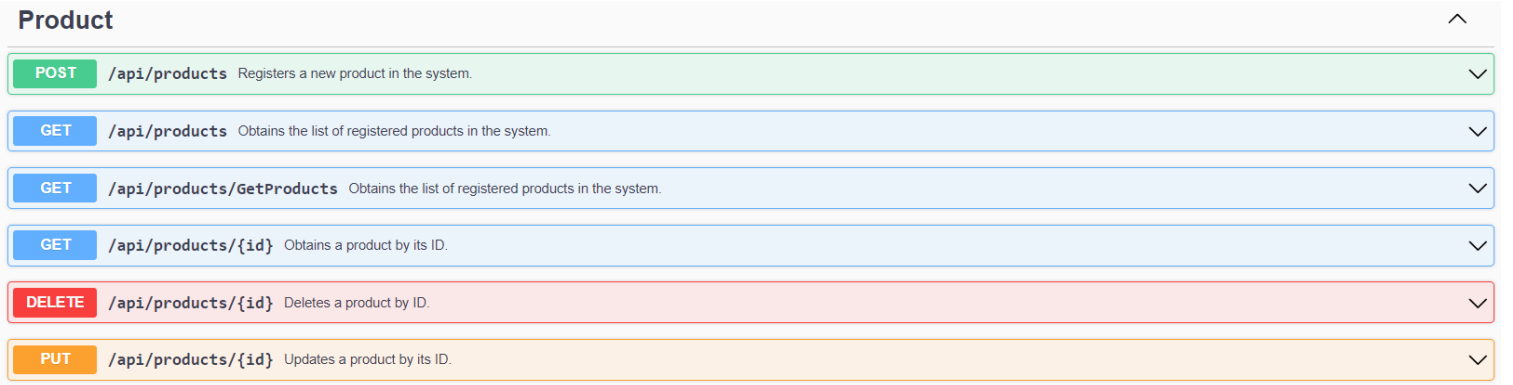
PURCHASE

URL Base	/api/purchases				
Resource	Purchase				
Description	Representa una compra realizada				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	POST	/api/purchases	Crea un purchase	*p:* Cuerpo de la solicitud (purchase a crear)	200 OK: Compra satisfactoria
	GET	/api/purchases	Obtiene todos los purchases	-	200 OK: retorna los purchases
	GET	/api/purchases/{id}	Obtiene un purchase por id	*id:* Cuerpo de la solicitud (id del purchase)	200 OK
	GET	/api/purchases/{id}/usersPurchases	Obtiene todos los purchases de un usuario	*id:* Cuerpo de la solicitud (id del usuario)	200 OK: retorna los purchases



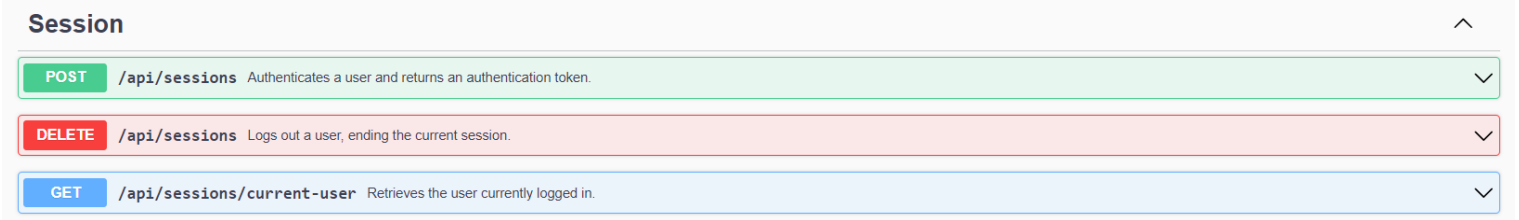
PRODUCT

URL Base	/api/products				
Resource	Product				
Description	Representa un producto del sistema.				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	POST	/api/products	Crea un producto	*product:* en el cuerpo de la solicitud	200 OK: Producto registrado correctamente
	GET	/api/products	Obtiene la lista de los productos	-	200 OK 400 Bad Request: Error al obtener productos
	GET	/api/products{id}	Obtiene un producto por id	*id:* Cuerpo de la solicitud	200 OK 400 Bad Request: Producto no encontrado
	DELETE	/api/products	Elimina un producto	*id:* Cuerpo de la solicitud	200 OK
	UPDATE	/api/products{id}	Actualiza los datos de un producto	*id:* Cuerpo de la solicitud, *product:* ruta	200 OK



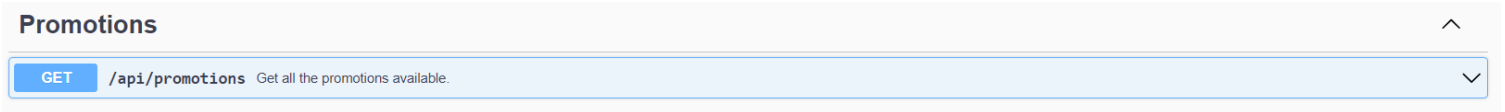
SESSION

URL Base	api/sessions				
Resource	Session				
Description	Representa una sesión activa del sistema.				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	POST	api/sessions	Inicio de sesion decierto usuario	*user:* Cuerpo de la solicitud	200 OK
	DELETE	api/sessions	Cierre de sesión de cierto usuario	-	200 OK
	GET	api/sessions/current-user	Obtiene el usuario iniciado sesión	-	200 OK 404 NOT FOUND



PROMOTION

URL Base	api/promotions				
Resource	Promotion				
Description	Representa las promociones vigentes del sistema.				
Endpoints	Operación	Ruta	Descripción	Parámetros	Responses
	GET	api/promotions	Obtiene las promociones disponibles	-	200 OK



Configuración e Inyección de Dependencias

La clase Program en la aplicación de API web es fundamental para la configuración y la inyección de dependencias. En esta sección, se describen los principales componentes y servicios que se configuran y se inyectan en la aplicación.

Configuración de la Aplicación

- Se utiliza WebApplication. CreateBuilder(args) para crear la instancia de la aplicación web.
- Se configuran los servicios esenciales, como controladores y la configuración de sesiones.

```
namespace Obligatorio1.WebApi
{
    0 referencias | fiioro25, Hace 2 horas | 3 autores, 21 cambios
    public class Program
    {
        0 referencias | fiioro25, Hace 2 horas | 3 autores, 21 cambios
        public static void Main(string[] args)
        {
            var builder = WebApplication.CreateBuilder(args);

            // Add services to the container.
            builder.Services.AddControllers();

            // Configure sessions
            builder.Services.AddDistributedMemoryCache(); // Opcional, para almacenar en caché la sesión en memoria
            builder.Services.AddHttpContextAccessor();
            builder.Services.AddSession(options =>
            {
                options.IdleTimeout = TimeSpan.FromMinutes(30); // Establece el tiempo de inactividad de la sesión
                options.Cookie.HttpOnly = true;
                options.Cookie.IsEssential = true; // Si deseas que la sesión sea esencial para la aplicación
            });
```

Configuración de Sesiones

- Se configura la administración de sesiones para mantener la información del usuario durante el tiempo de inactividad.
- Se almacenan las sesiones en memoria y se establece un tiempo de inactividad de 30 minutos.

Swagger y Documentación

- Se configura Swagger para generar documentación interactiva de la API.
- Se incluyen comentarios XML en el código fuente para proporcionar descripciones detalladas en Swagger.

```
            builder.Services.AddEndpointsApiExplorer();
            builder.Services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc("v1", new OpenApiInfo { Title = "Obligatorio 1", Version = "v1" });
                var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
                var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
                c.IncludeXmlComments(xmlPath); // Esto incluirá el archivo XML de documentación en Swagger
            });
```

Inyección de Dependencias

Se realizan inyecciones de dependencias para los siguientes servicios:

- IUserService para la gestión de usuarios.
- IProductService para la gestión de productos.
- IUserManagment para la administración de usuarios.
- IPromoManagerManagment para la administración de promociones.
- ICartService para la gestión de carritos de compra.
- IProductManagment para la administración de productos.
- DbContext para el acceso a la base de datos.
- Repositorios genéricos para las entidades User, Product, Purchase, y Cart.

```
builder.Services.AddScoped<Obligatorio1.IBusinessLogic.IUserService, Obligatorio1.BusinessLogic.UserService>();

builder.Services.AddScoped<Obligatorio1.IBusinessLogic.IProductService, Obligatorio1.BusinessLogic.ProductService>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IUserManagment, UserManagment>();

builder.Services.AddScoped<Obligatorio1.IDataAccess.IPromoManagerManagment, PromoManagerManagment>();
builder.Services.AddScoped<Obligatorio1.IBusinessLogic.ICartService, Obligatorio1.BusinessLogic.CartService>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IProductManagment,ProductManagment>();
builder.Services.AddDbContext<Context>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IGenericRepository<User>, Obligatorio1.DataAccess.Repositories.GenericRepository<User>>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IGenericRepository<Product>, Obligatorio1.DataAccess.Repositories.GenericRepository<Product>>();


builder.Services.AddScoped<Obligatorio1.IDataAccess.IUserManagment, Obligatorio1.DataAccess.Repositories.UserManagment>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IPromoManagerManagment, Obligatorio1.DataAccess.Repositories.PromoManagerManagment>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IPurchaseManagment, Obligatorio1.DataAccess.Repositories.PurchaseManagment>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.ICartManagment, Obligatorio1.DataAccess.CartManagment>();
builder.Services.AddScoped<Obligatorio1.Domain.User>();
builder.Services.AddScoped<Obligatorio1.IBusinessLogic.IPurchaseService, Obligatorio1.BusinessLogic.PurchaseService>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IGenericRepository<Purchase>, Obligatorio1.DataAccess.Repositories.GenericRepository<Purchase>>();
builder.Services.AddScoped<Obligatorio1.IDataAccess.IGenericRepository<Cart>, Obligatorio1.DataAccess.Repositories.GenericRepository<Cart>>();
```

En conclusión, la clase Program se encarga de configurar y gestionar servicios esenciales para la API web, como sesiones, Swagger y la inyección de dependencias de servicios relacionados con la gestión de usuarios, productos y compras. Esta configuración sólida y las inyecciones de dependencias permiten que la API funcione de manera eficiente y escalable.

Conclusión

La API ha sido diseñada con atención a los detalles, siguiendo las mejores prácticas de diseño de APIs RESTful. Este diseño proporciona una base sólida para el desarrollo y la expansión futura del sistema, garantizando la seguridad y eficiencia en la comunicación entre clientes y servidores. Seguiremos mejorando la API en la iteración siguiente cuando se implemente toda la parte del FrontEnd.