

# Universidad ORT Uruguay

## Facultad de Ingeniería

### Obligatorio 1 - Diseño de Aplicaciones 2

### Descripción del diseño



<https://github.com/IngSoft-DA2-2023-2/266628-255981-271568>

Federico Rodriguez - 255981

Santiago Salinas - 266628

Manuel Morandi - 271568

**Tutores:** Daniel Acevedo

Rafael Alonso

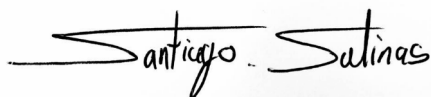
Franco Galeano

Septiembre 2023

# Declaración de autoría

Federico Rodriguez, Santiago Salinas y Manuel Morandi declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

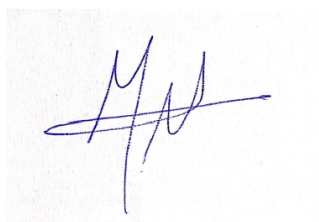
- La obra fue producida en su totalidad mientras construimos el primer obligatorio de Diseño de Aplicaciones 2
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra 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 contribuído por otros y qué fue contribuído por nosotros
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes



Santiago Salinas 26/9/2023



Federico Rodriguez 26/9/2023



Manuel Morandi 26/9/2023

## Abstract

El objetivo de este documento es demostrar que el equipo fue capaz de diseñar y documentar el diseño de la solución.

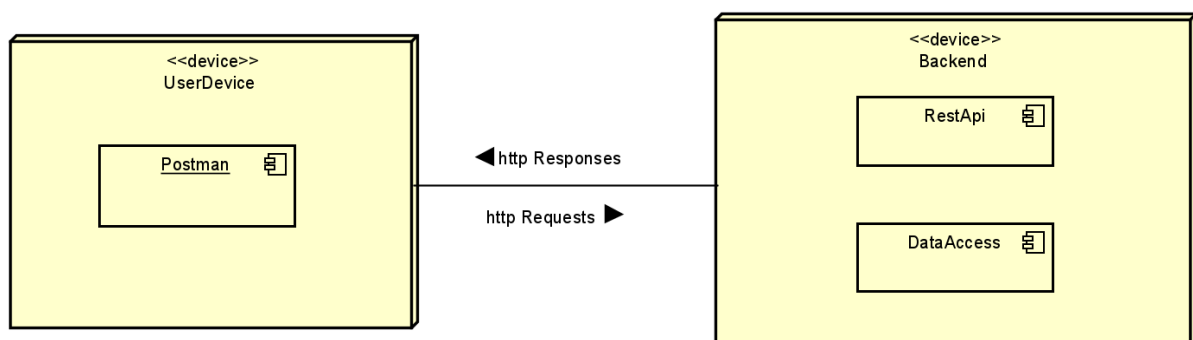
Pensada para que un tercero (corrector) pueda en base a la misma comprender la estructura y los principales mecanismos que están presentes en el código. O sea, debe servir como guía para entender el código y los aspectos más relevantes del diseño y la implementación.

<b>Abstract</b>	<b>2</b>
<b>Descripción general del trabajo</b>	<b>4</b>
<b>Errores conocidos o funcionalidades no implementadas</b>	<b>5</b>
<b>Diagramas de dependencias</b>	<b>5</b>
<b>Diagrama general de paquetes</b>	<b>7</b>
DataAccess	9
Rest Api	10
Services	12
<b>Diagramas de interacción</b>	<b>14</b>
<b>Uso de patrones</b>	<b>17</b>
Patrón Factory	17
Patrón Repository	18
Patrón Strategy	19
<b>Arquitectura de Filtros</b>	<b>20</b>
Authentication filter	20
Authorization filter	20
Exception filter	21
<b>Conclusión</b>	<b>21</b>

# Descripción general del trabajo

El objetivo de este trabajo es desarrollar una Web API para un negocio de venta de ropa. Esta API debe ser RESTful, esto implica que implemente las recomendaciones presentadas por las soluciones API REST. Se implementan un set de operaciones para manejar los distintos recursos de la tienda a través HTTP, haciendo uso de los verbos get, put, post y delete. Adicionalmente, la solución debe ser stateless, cliente-servidor, y el resto de criterios de REST. Esto se expande en la documentación “Evidencia del diseño y especificación de la API”. A continuación presentamos a modo de desarrollo el trabajo realizado, decisiones tomadas y sus implicancias.

Con el siguiente diagrama de deploy pretendemos mostrar cómo el funcionamiento del sistema se divide en dos principales nodos: el servidor/backend y el cliente/usuario. Toda la lógica de negocio y de acceso a base de datos se encuentra en el dispositivo del backend, así como la implementación de la Api. Mientras que el usuario puede hacer iniciar operación mediante el uso de HTTP requests y ver los resultados mediante responses HTTP, incluso sin nosotros haber desarrollado frontend específico para el proyecto.



## Errores conocidos o funcionalidades no implementadas

Como las operaciones destinadas a ser usadas por administradores requieren enviar en el header un token correspondiente a un usuario con el rol de Admin en la base de datos, se requiere agregar manualmente un usuario con el rol Admin en la base de datos antes.

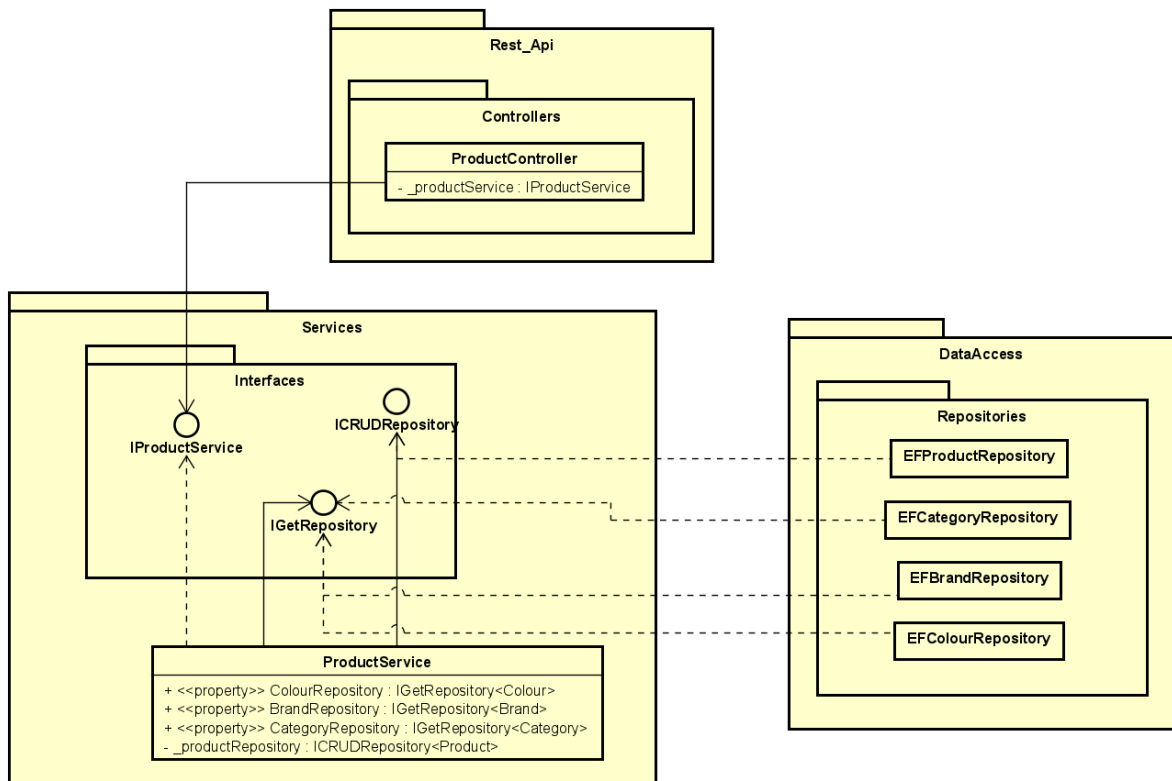
Debido a cómo implementamos la asignación de roles para un usuario, se requiere que para poder crear un usuario y subirlo a la base de datos estén ya en la tabla Role Entities ingresados los roles que se les quiera asignar a un usuario.

De igual manera, para poder crear un producto con determinados colores, marca y categoría, estas características deben agregarse primero manualmente a sus respectivas tablas, debido a la falta de un endpoint dedicado.

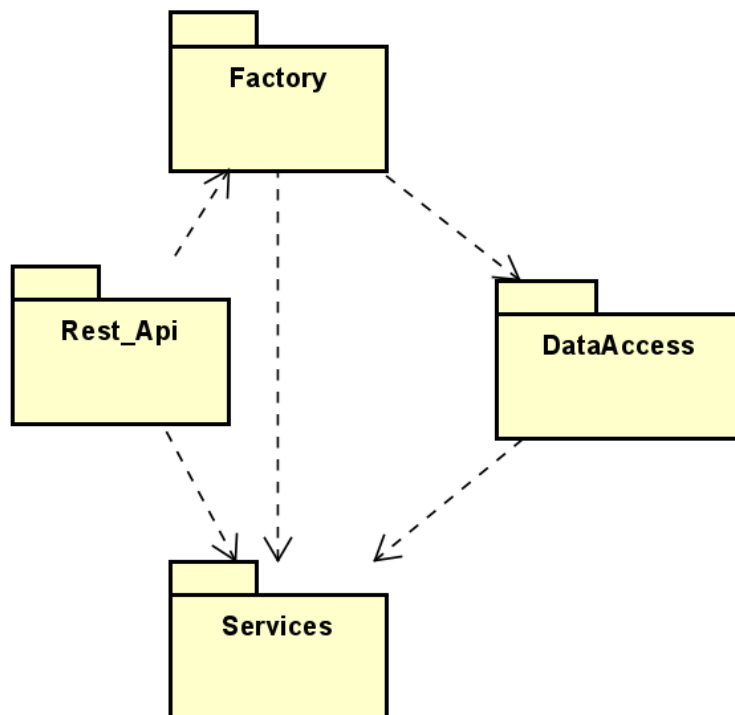
## Diagramas de dependencias

En la siguiente imagen mostramos un ejemplo de cómo se relacionan las dependencias entre clases de los proyectos. Las interfaces usadas por las distintas capas de operación existen todas en el paquete de Services dentro del sub namespace Interfaces. Así, los Controllers de Rest Api pueden usar servicios conociendo solo una interfaz, como en el ejemplo de ProductController que utiliza un servicio que implemente IProductService. Y los repositorios en DataAccess pueden implementar interfaces conocidas también por los servicios.

Los controllers utilizan cada cual uno o varios servicios a través de interfaces, mientras que los servicios a su vez utilizan uno o múltiples repositorios cada uno también a través de interfaces.



## Diagrama general de paquetes



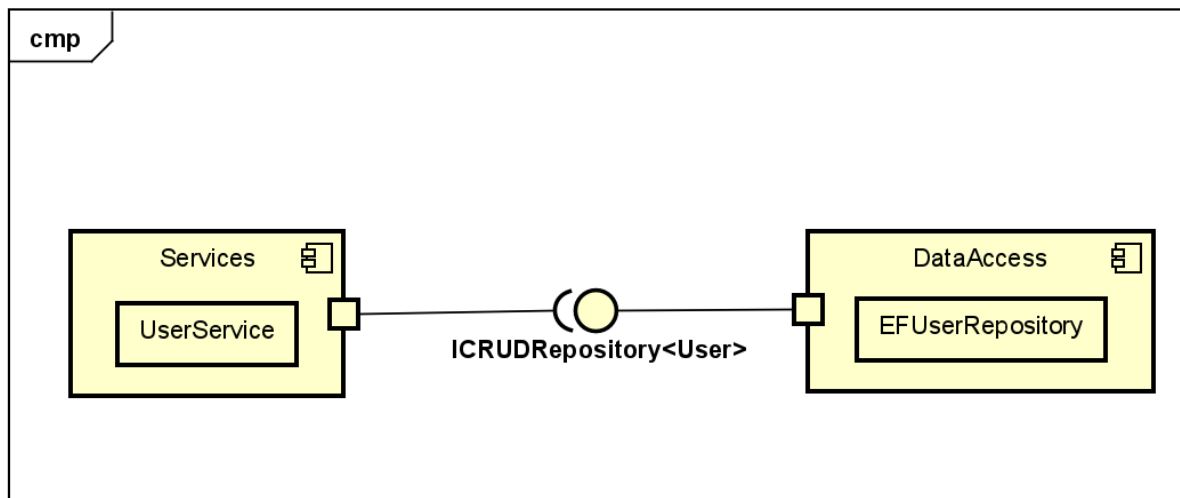
Primero, comencemos hablando de cómo se organizó el proyecto. Cómo puede verse en el anterior diagrama, nuestra solución está formada por cuatro paquetes, cada uno con su motivo único de existir. Estos se verán más a profundidad a continuación, pero como resumen: **Services** expone la lógica de negocio, el dominio y las interfaces que utilizan el resto de paquetes, **DataAccess** se ocupa del acceso a base de datos, **Rest Api** implementa los elementos que forman nuestra Api (controladores y filtros) y **Factory** tiene como responsabilidad la inyección de dependencias (instanciar distintos repositorios y servicios que la aplicación requiere). Es justo decir, entonces, que se cumple SRP, ya que cada paquete tiene una responsabilidad única.

Hablando de los principios SOLID, vale la pena mencionar el principio DIP. Este establece que los módulos de alto nivel no deben depender de aquellos de bajo nivel. Los módulos de alto nivel (en otras palabras, los que no tienen tendencia a cambiar) deben en su lugar depender de abstracciones estables, para protegerse ante la variación. Vemos que esto es exactamente el caso, **Rest Api** y **DataAccess**, que son los módulos más volátiles (bajo nivel) dependen del módulo de mayor nivel que es **Services**. Si **Services** requiere llamar algún método de los paquetes de



menor nivel no lo hace directamente, ya que estos implementan interfaces definidas en el propio paquete Services, brindando esta capa de abstracción. Si en un futuro se cambia de Api o base de datos, Services no se verá afectado ya que interactúa con sus interfaces.

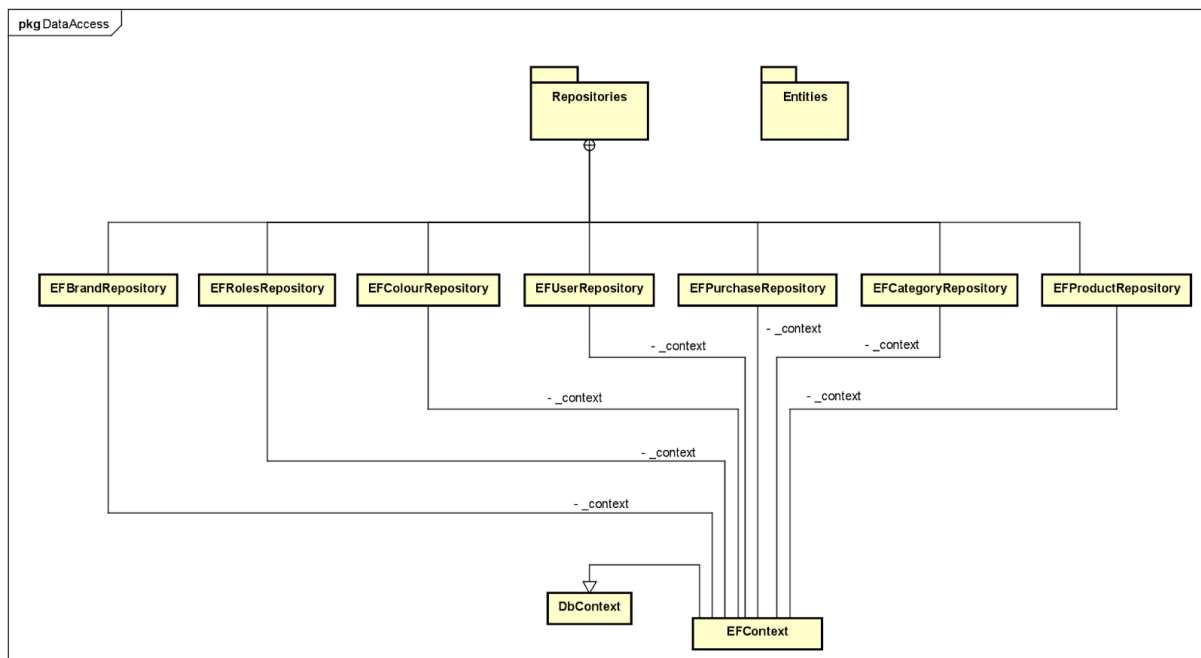
El siguiente diagrama de componentes puede ayudar a comprender el funcionamiento de estas interfaces:



Notamos que UserService requiere la interfaz de repositorio de User que está definida en el propio paquete de Services, pero que es implementada por EFUserRepository en DataAccess. Esto se trata de simplemente un ejemplo, ya que lo mismo ocurre para el resto de repositorios, y también para los controladores de Rest Api, que utilizan interfaces de los distintos servicios de los modelos.

Todo esto favorece mucho a la mantenibilidad, pero también tuvimos en mente la escalabilidad. Para explicar este apartado, veamos cómo se cumplen los distintos principios de acoplamiento. Podemos notar que el grafo de dependencias formado por el diagrama es dirigido y acíclico, por lo que se cumple ADP. Asimismo, y vinculado a lo que se habló en el anterior párrafo en relación a DIP, el flujo de las dependencias va hacia la estabilidad, un paquete estable no depende de uno inestable, cumple SDP. Este paquete estable, Services en este caso, es concreto y, como ya se mencionó, define abstracciones en forma de interfaces, llevando así al cumplimiento de SAP.

## .DataAccess



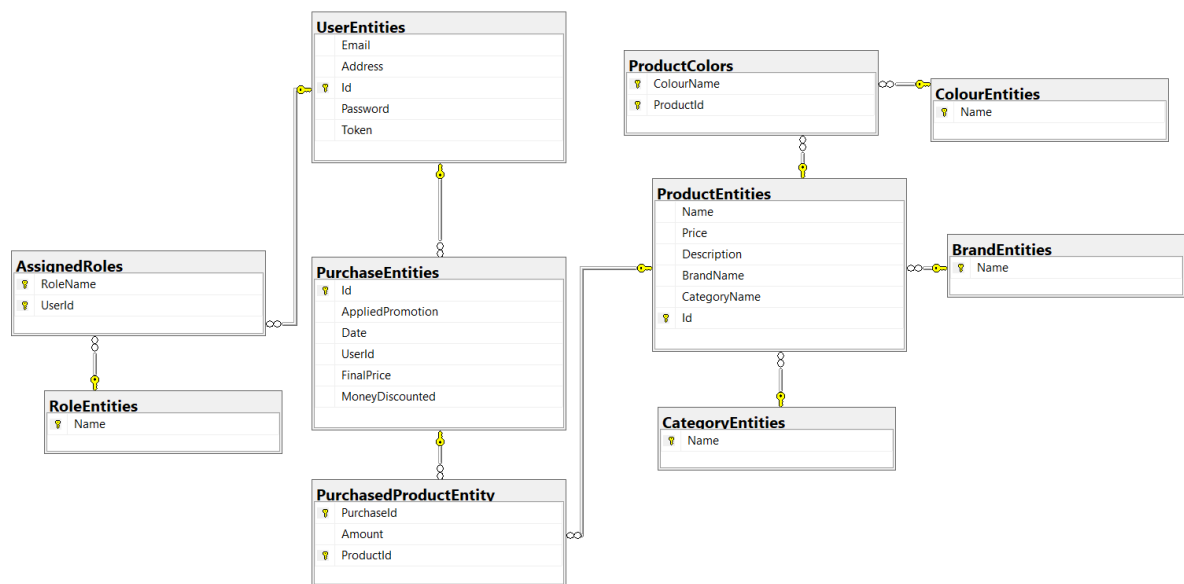
El paquete de `DataAccess` contiene la implementación basada en Entity Framework Core para las clases de tipo Repositorio. Cada una de estas clases contiene operaciones de tipo CRUD para interactuar con una tabla específica de una base de datos relacional.

Específicamente, los repositorios respectivos para las entidades `Products`, `Purchases` y `User`, tienen todas las operaciones CRUD mientras que las de `Brand`, `Category` y `Colour` solo implementaron la operación `Get` y `GetAll`, pues no requieren de más.

Además de las implementaciones específicas para EF de los repositorios, se incluyen `Exceptions` para manejo de errores y la clase `DbContext` que no es más que una clase que hereda de la clase `DbContext` nativa de EF y contiene la especificación de las tablas de la base de datos y otras configuraciones para su funcionamiento.

También se encuentran las `Entities` que reflejan las tablas de la base de datos que utilizamos para persistir los recursos a los cuales queremos poder acceder a través de la API que se está desarrollando. Estas incluyen métodos para mapear entidades desde y hacia sus contrapartes del dominio.

Se adjunta el diagrama de las tablas de la base de datos:



La aplicación incluye dos archivos .bak (y sus equivalentes scripts sql) con respaldo de la base de datos. El primer archivo, DB.bak, incluye el set de datos de prueba creados para utilizar la aplicación. Este set contiene usuarios, productos, compras, etc. suficientes para ver todos los usos posibles de la API.

Por otro lado, encontramos emptyDB.bak, que, como su nombre indica, se trata de la base de datos “vacía”. En este respaldo encontraremos únicamente los roles Admin y Customer, junto con un usuario de id 1, email [admin@gmail.com](mailto:admin@gmail.com), contraseña admin y token tokenadmin@gmail.comsecure. Este usuario tiene asignado ambos roles.

Algo relevante sobre emptyDB.bak es que no incluye ninguna marca, color ni categoría. Todos los productos deben tenerlos, por lo que se deberán crear (desde la propia base de datos, no se contempla un endpoint para agregarlos) antes de llevar a cabo la creación de productos.

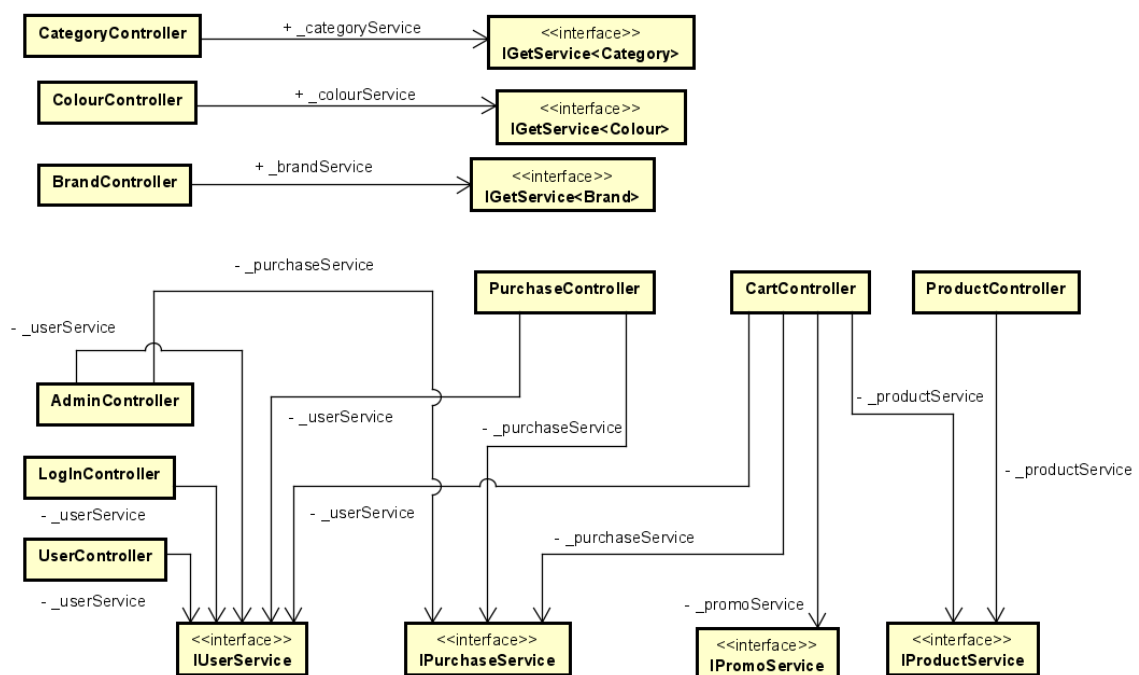
## Rest Api

Este paquete engloba todos los endpoints de nuestra API. Encontramos en él cuatro tipos de elementos distintos. Por un lado, los controladores, clases que nos permiten operar sobre los distintos recursos. Estos controladores harán uso de interfaces de servicios de los distintos recursos disponibles, para así poder llevar a cabo las acciones solicitadas.

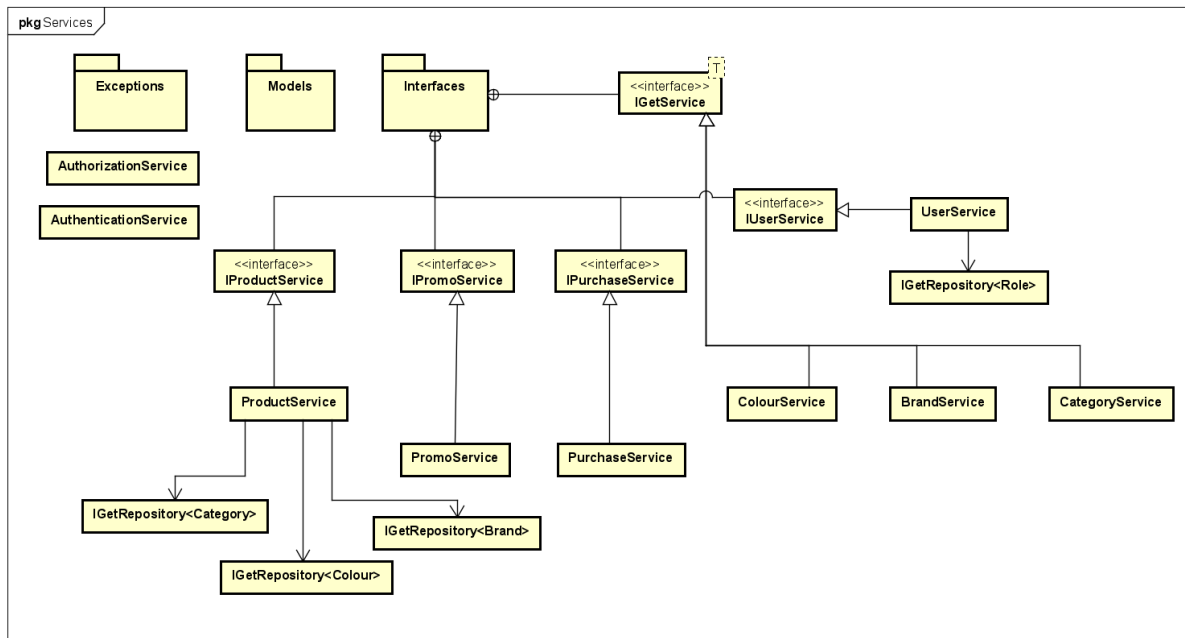
Algo a destacar es que no todos los recursos son modelos de la lógica de negocio. Por ejemplo, tenemos un controlador Admin, desde el cuál se llevan a cabo todas las distintas operaciones que sólo pueden ser ejecutadas por administradores. Es una construcción abstracta, representa un punto para que esas operaciones se concentren, no representa realmente algo físico.

En este paquete también encontramos DTOs para transferir información de los JSON recibidos en los body de las request hasta la creación del objeto; y filtros, middleware que se ocupan de correr cierta lógica en distintos niveles de la API, de los cuales se hablará más adelante. Finalmente encontramos el main.

Mostramos en este diagrama el sub namespace de Controllers, donde se guardan las clases que manejan la funcionalidad de la Api.



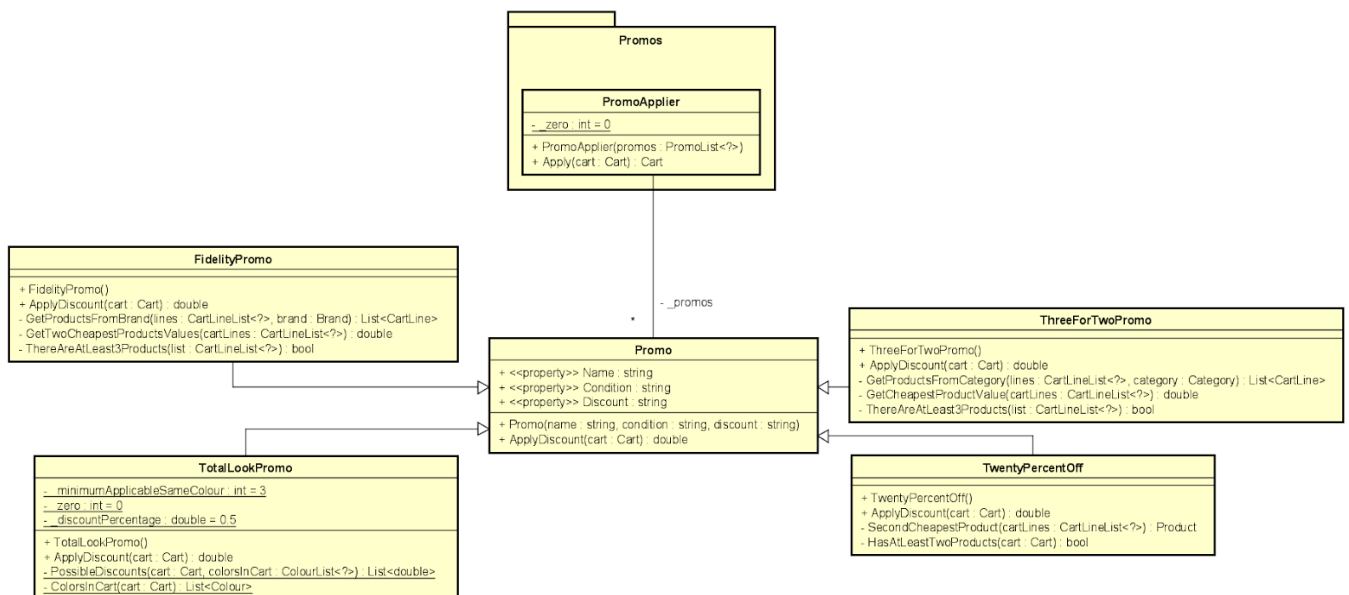
## Services



El paquete Services es uno que contiene las clases que conforman el dominio de nuestra aplicación en el sub namespace Models, es decir clases que representan o ayudan a representar los recursos que se pueden administrar con el uso de la Rest Api como Product, User, Color, etc. Por supuesto, contiene su propio conjunto de Excepciones para el manejo de errores a nivel de Servicios, Dominio o DataAccess y las interfaces que implementan tanto los servicios como las que deben implementar los repositorios que pretendemos utilizar.

De manera general definimos dos interfaces distintas que pueden implementar los repositorios: `ICRUDRepository<T>` y `IGetRepository<T>`. Se diferencian en que una la primera exige poder hacer todas las operaciones CRUD (Get, Update, Remove, Create) y la segunda sólo exige los métodos para Get y GetAll. Ambas hacen uso de Generics, es decir que se puede implementar la misma interfaz para distintos repositorios que solo varían en la clase del dominio que administran. El mismo concepto se intentó aplicar para los servicios, como se ve en la interfaz `IGetService<T>` que implementan los servicios para Colour, Brand y Category.

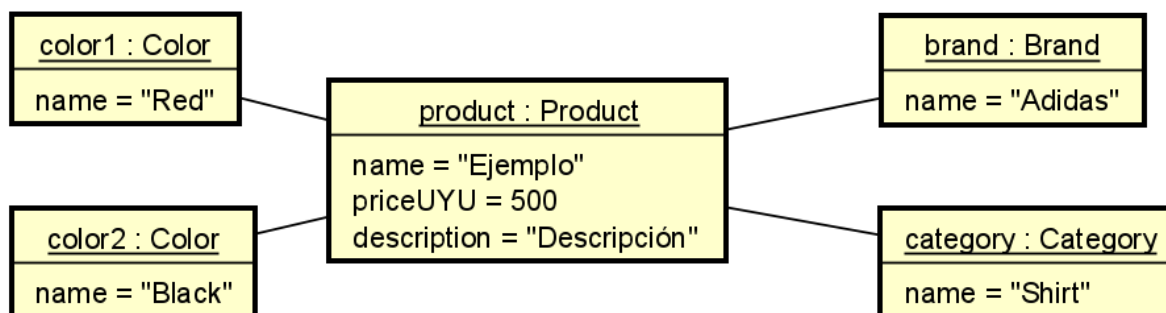
Dentro de Services, vale la pena explicar el funcionamiento de las promos. A continuación vemos el diagrama de como fue implementado:



Notamos que todas las promos disponibles heredan de una clase Promo. Esto es muy beneficioso, ya que nos permite agregar nuevas promos sin tener que modificar nada de lo ya implementado. Gracias a esto, podemos afirmar que la implementación de las promos cumple OCP. Si en un futuro se quisiera agregar una nueva promo, se podría crear una nueva clase que herede de Promo, implementando en ella su propio método ApplyDiscount.

Esto último nos afirma también que nuestro sistema de promos también cumple LSP, ya que todas las clases “hijas” se comportan como el “padre”. En otras palabras, todas las promos implementan este método ApplyDiscount, ya que la clase Promo lo tiene y exige que se implemente. Posteriormente, este método será usado por el PromoApplier para hallar la mejor promo a utilizar.

Los productos, así como los usuarios, tienen atributos del tipo de clases de nuestra lógica de negocio. El siguiente diagrama de un producto genérico puede ayudar a comprender esto:

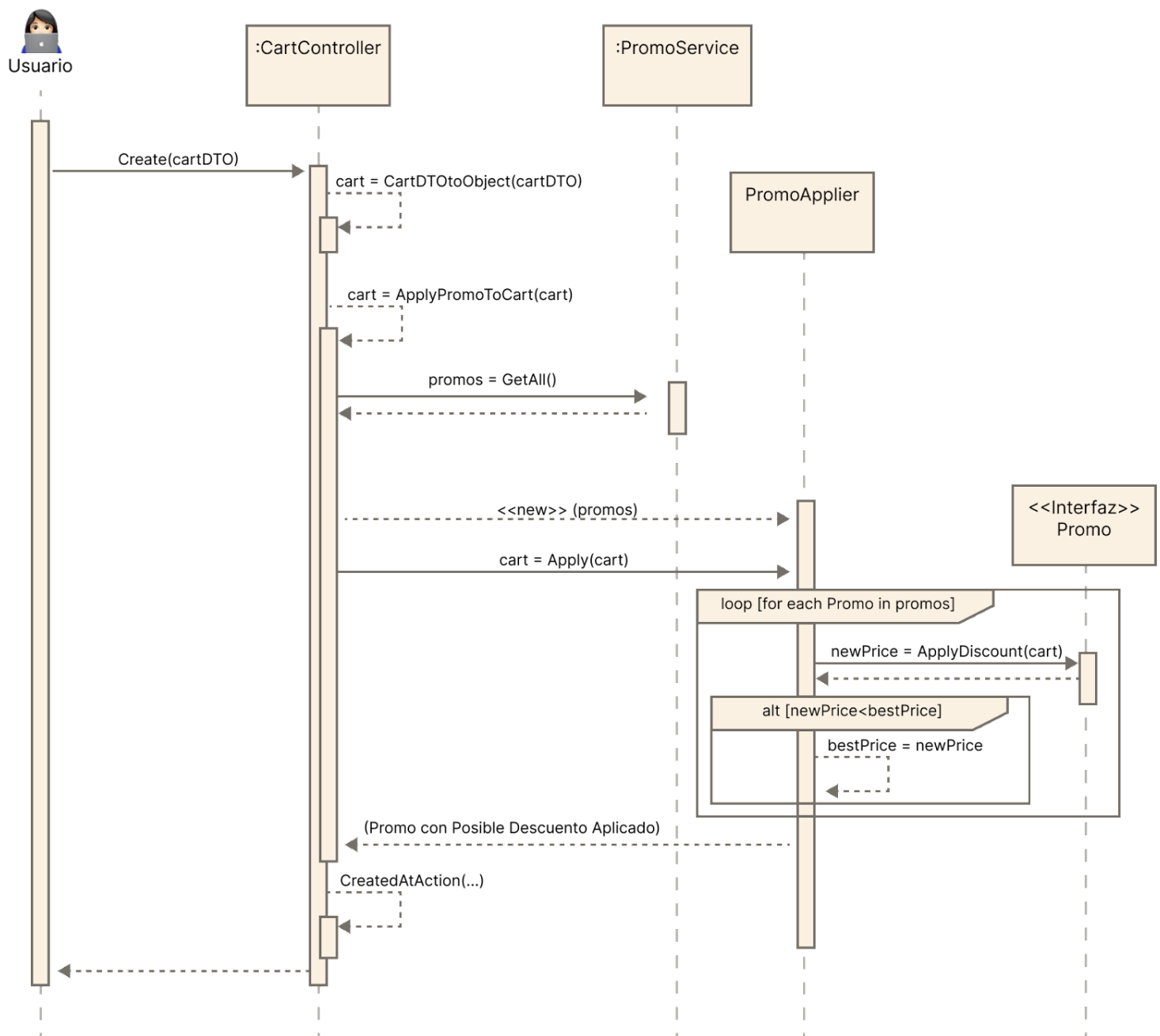


Como podemos notar, un producto tiene una marca, una categoría y uno o más colores. Paralelamente, un usuario puede tener varios roles. Para lograr esto, estos objetos que “viven” dentro del producto o usuario son instanciados por la propia clase. Esto implica, el objeto Brand de nombre Adidas que tiene el producto anterior será una instancia diferente a la que tendrá cualquier otro producto Adidas.

Ahora, pese a ser dos instancias distintas, podemos tratarlas como iguales, ya que podemos compararlas y ver que comparten el mismo nombre.

## Diagramas de interacción

Diagrama de comunicación al envío de un POST a /Cart para obtener el precio y el descuento aplicado:

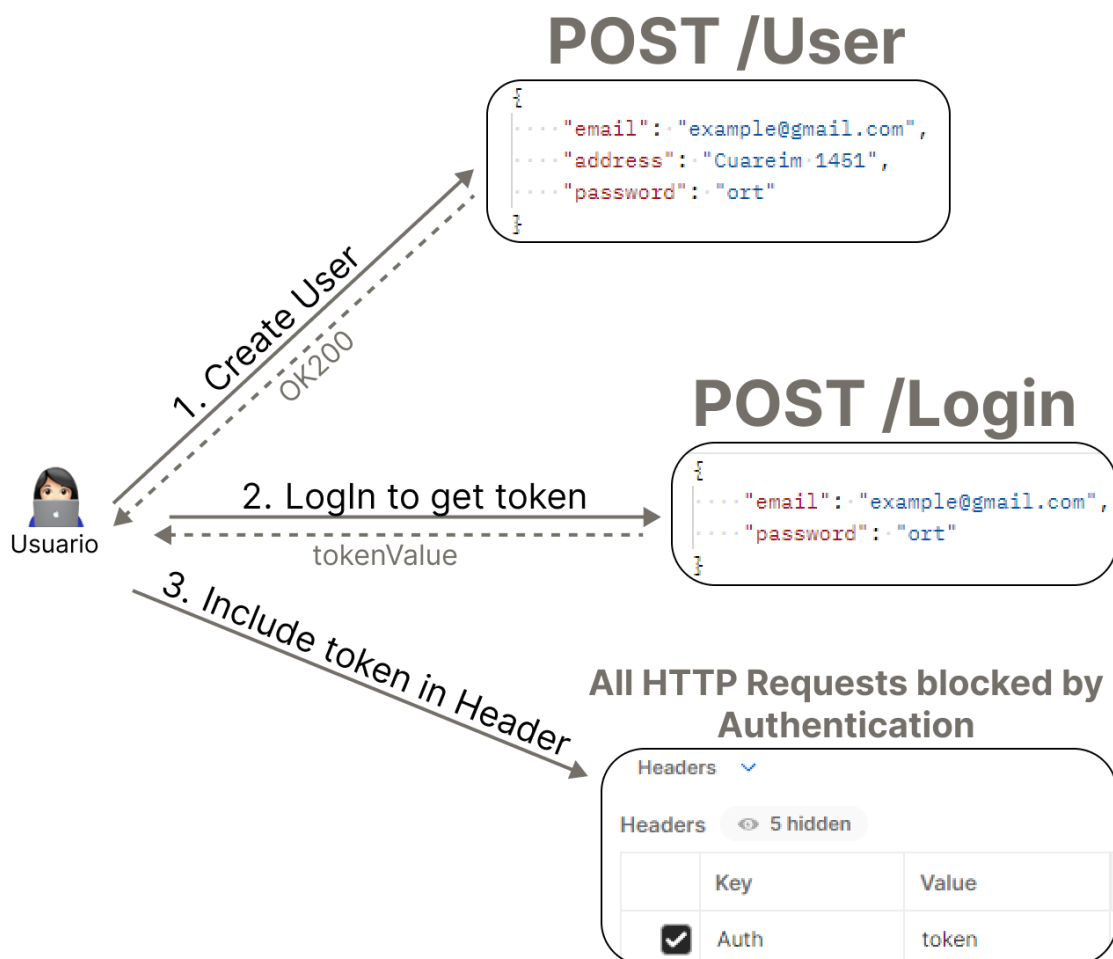


Se tomaron varias decisiones de diseño que giran alrededor del carrito de compras. Primero, no es necesario que el usuario esté logueado en el sistema para poder utilizarlo, es decir, se puede utilizar sin enviar un token auth en el header. Asimismo, el carrito no se guarda, a menos de que se efectúe la compra. Es algo efímero que solo existe del lado del cliente. Para saber su valor, se envía a la API los IDs de los productos y las cantidades seleccionadas para así calcular el precio y posible descuento, para posteriormente mostrarlo al usuario en el response. Al



efectuarse la compra el carrito es enviado, para crear un objeto Purchase a partir de él y agregarlo a la lista de compras asociada a su usuario.

¿Cómo hacer uso de los tokens? Debemos crear un usuario si previamente no lo habíamos hecho, mediante un POST en /User. Si los datos ingresados son únicos y todo es correcto, recibiremos un status code 200 OK. Para poder obtener nuestro token, debemos ir al URI de /Login y postear nuestro email y contraseña. Con este token, podremos operar en endpoints que sean bloqueantes por autenticación, agregandolo en el header como Key "Auth".



# Inyección de dependencias

La inyección de dependencias es el proceso por el cual se instancian las clases reales que harán uso de las interfaces definidas y de las cuales otras subclases dependen, sin saber que esa será la implementación final definida, ya que solo conocen a la interfaz. Esto trae como ventaja principal el desacoplamiento de las clases siguiendo el principio SOLID de Inversión de Dependencias.

Este mecanismo influye altamente en el mantenimiento de la aplicación ya que, ante GRASP, obtenemos variación protegida. Si el día de mañana deseamos cambiar nuestro DataAccess soportado por Entity Framework y SQL, por otro, por ejemplo, MongoDB, no debemos hacer cambios mayores a nuestro código. Cumpliendo a su vez, otro principio SOLID más, Open Close, dónde estamos abiertos a la extensión, pero cerrados al cambio.

Las desventajas de esta técnica es su complejidad para aplicarla desde el inicio, por lo que una vez construida base inicial de nuestro programa, se realizó un refactorio para poder aplicarla.

Para ello hicimos uso del patrón factory, desarrollado en el siguiente capítulo.

## Uso de patrones

### Patrón Factory

A la hora de instanciar los repositorios y los servicios que serán usados en el sistema surge un dilema: ¿dónde lo hacemos? Para solucionar esta problemática se establece esta capa Factory, que instancia todos repos y servicios que el resto de clases usarán. Al necesitar hacer uso de uno de ellos, se le solicita al Factory que nos devuelva la instancia que almacenó de la clase solicitada, consiguiendo de esta manera una interfaz donde se crean todos estos objetos con el mismo contexto, para que la Api pueda usar.

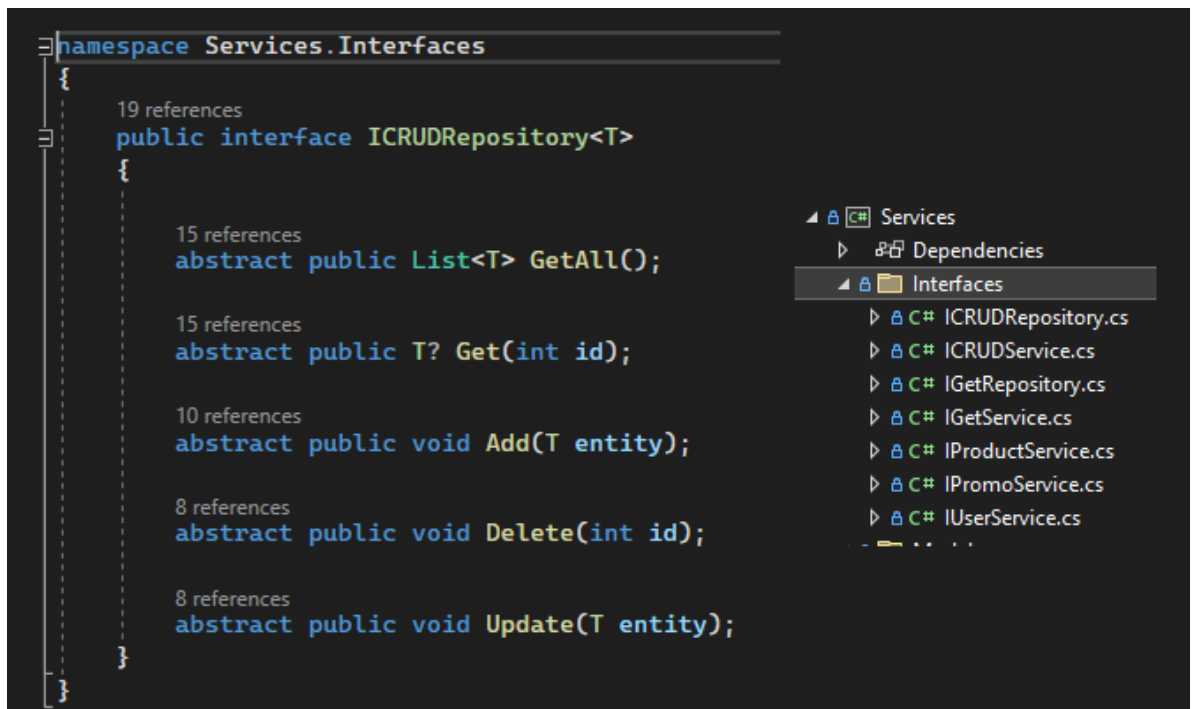
## Patrón Repository

Ya como punto de partida planteamos el uso de una interfaz dada llamada `ICRUDRepository<T>` y `IGetRepository<T>`, que incluyen las operaciones básicas CRUD Get, Update, Delete y Create, o solo la operación Get respectivamente. aprovechando el uso generics `<T>`.

Estas interfaces nos permiten implementar un pseudo repository pattern. De esta manera, el acceso a la base de datos para cualquiera de los tipos está cubierto por esta interfaz. Si en un futuro se cambia la base de datos a usar, podemos implementar una clase que realice esta interfaz, permitiendo abstraer la API de la tecnología a usar, como hablamos anteriormente con la inyección de dependencias.

Como ventaja adquirida, consideramos que en el paquete de `DataAccess` es imprescindible, ya que toda base de datos se maneja con las operaciones básicas CRUD, pero la realidad es que por fuera de `DataAccess`, como por ejemplo los `Services` no ha sido tan útil. A la hora de codificar, nos limita enormemente en nuestro desarrollo al tratar con estructuras complejas para los servicios, por lo que posteriormente se crearon en el proyecto una interfaz dedicada a cada servicio si este lo requería.

Un verdadero Repository Pattern debería realizar una implementación única de un repositorio genérico y luego aplicarla para cada recurso relevante. Esto permitiría que, si se decide cambiar de Entity Framework a otra forma de manejo de base de datos, solo se necesitaría cambiar la implementación del repositorio en una clase en lugar de una clase por cada recurso. Sin embargo, por la complejidad que conlleva implementar este patrón en su completitud, y sabiendo que no vamos a cambiar de EF, se optó por una versión simplificada con múltiples implementaciones de las interfaces.



## Patrón Strategy

El patrón strategy se define como aquel que permite que la manera de operar sea inmutable pero permitiendo hacer cambios a los algoritmos aplicados para una misma tarea según corresponda, en tiempo de ejecución.

Para llevar a cabo el análisis de las distintas promociones que se pueden aplicar sobre un carrito, se implementó efectivamente el patrón strategy. El constructor del aplicador de promociones, recibe un listado con las promociones que se ofrecen actualmente gracias al servicio de promociones.

Y este se encarga de aplicar uno por uno las promociones, y devolver un carrito con la información asociada al descuento aplicado.

```
1 reference
public PromoApplier(List<Promo>? promos)
{
    _promos = promos;
}

1 reference
public Cart Apply(Cart cart)
{
    if (cart is null || cart.Products.Count == _zero) { return new Cart(); }
    if(_promos is null) { return new Cart(); }

    double bestPrice = cart.PriceUYU;
    Promo? bestPromoToClient = null;

    foreach (Promo promo in _promos)
    {
        double newPrice = promo.ApplyDiscount(cart);
        if (newPrice < bestPrice)
        {
            bestPrice = newPrice;
            bestPromoToClient = promo;
        }
    }

    return new Cart { PriceUYU = bestPrice, Products = cart.Products };
}

23 references
public abstract class Promo
{
    4 references
    public Promo(string name, string condition, string discount)
    {
        Name = name;
        Condition = condition;
        Discount = discount;
    }

    2 references
    public string Name { get; set; }
    1 reference
    public string Condition { get; set; }
    1 reference
    public string Discount { get; set; }

    35 references | 29/29 passing
    public abstract double ApplyDiscount(Cart cart);
}
```

## Arquitectura de Filtros

### Authentication filter

Este filtro se encarga de comprobar si quien manda el request es efectivamente quien dice ser, sin necesidad de mandar sus credenciales completas. Esto se hace mediante el uso de tokens, un concepto que en criptografía y tecnologías web se basa en darle al cliente una llave de acceso. En muchos casos esta es temporal y/o pueda ser revocada, en caso de ser comprometida sin tener que cambiar la contraseña del usuario.

Este filtro utiliza una interfaz propia de .NET, donde en OnAuthorization se valida que el parámetro “auth” del header del request se encuentre asociado a un usuario.

Como resultado de usar este filtro se pueden recibir códigos de error 401 si el token está vacío y 403 si el token no fue asignado a un usuario en la base de datos.

### Authorization filter

Al igual que el anterior filtro, este middleware también implementa IAuthorizationFilter de .NET. El método OnAuthorization es utilizado en esta ocasión para verificar si el usuario asociado al token presente en el header tiene el rol de Admin. De esta manera, logramos evitar que aquellos usuarios que no tengan este

privilegio accedan a operaciones restringidas, tales como ver todas las compras o eliminar cualquier usuario.

Si se usa este filtro, se recibirá un error 401 si el token está vacío y 403 en el caso de que un usuario no administrador esté intentando acceder a algo exclusivo para Admins.

## Exception filter

Este filtro es capaz de atrapar cualquier excepción inesperada durante la ejecución de la lógica del backend, y devolver a través de la api una respuesta de código 500. Se pretende que atrape las excepciones inesperadas que logren escapar los catch ya establecidos en el flujo de interacción al momento resultante de empezar una operación a través de un controller.

Además, también es capaz de atrapar las excepciones que surgen antes de que empiece la ejecución del código del controller, por ejemplo al crearse el objeto del dominio que recibe por parámetro. En estos casos, devolvemos un código 400, señalando el error en el body enviado.

Esto nos permite disminuir la cantidad de try y catch que necesitamos para que el código no termine de manera abrupta, siendo la principal ventaja la eliminación de try y catch anidados. Se podrían reducir incluso aún más la cantidad de try/catch presentes.

## Conclusión

Se cumplió el objetivo, llegando a la construcción de la Api solicitada, de manera prolija. Se logró aplicar los conocimientos y técnicas vistas en el curso para amplificar el desempeño durante el proceso entero de desarrollo. Consideramos que el producto final está a la altura de lo esperado y es acorde a lo solicitado, dejando las puertas abiertas a imperfecciones a pulir y posibles mejoras a implementar en la siguiente entrega.