

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 2 - Diseño de Aplicaciones 2

Descripción del diseño

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

Federico Rodriguez - 255981

Santiago Salinas - 266628

Manuel Morandi - 271568

Tutores: Daniel Acevedo

Rafael Alonso

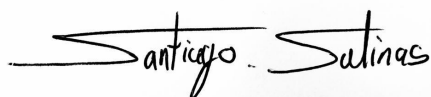
Franco Galeano

Diciembre 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:

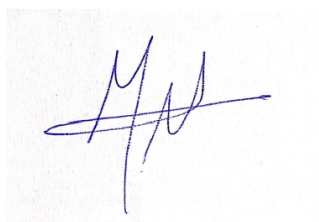
- La obra fue producida en su totalidad mientras construimos el segundo obligatorio de Diseño de Aplicaciones 2
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la 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

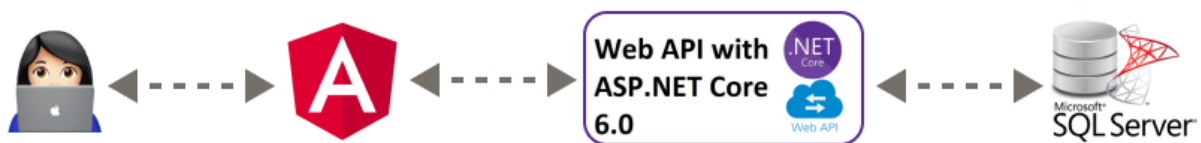
El objetivo de este documento es demostrar que el equipo fue capaz de diseñar y documentar el diseño de la solución.

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.

Abstract	3
Descripción general y errores conocidos	5
Diagrama general de paquetes	6
Modelo de tablas de la estructura de la base de datos	8
Explicación de cada paquete individual	9
DataAccess	9
Factory	9
Services	10
Rest Api	11
Nuevos Requerimientos	12
Promociones	12
Apply Discount	13
Stock de producto	13
Nuevos filtros de búsqueda	16
Métodos de pago	17
Frontend	19
Services	19
Routing	20
Guards	20
Patrones Aplicados	20
Patrón Adapter	20
Patrón Abstract Factory	21
Mejoras de diseño	22
Análisis de métricas y aplicación de principios	24
Cálculo a Mano	24
Cálculo mediante NDepend	27
Anexo	29
Especificación de endpoints nuevos o modificados	29
Get Cart Price and Info	29
Buy Cart	30
Filter Products y Get Products	31
Cobertura de Pruebas	32

Descripción general y errores conocidos

Para esta segunda entrega se mantuvo la solución del backend y de la Api mayoritariamente sin cambios, salvo por las nuevas funcionalidades. En cambio, se agregó una aplicación frontend desarrollada en Angular. Esta aplicación se presenta como una página web de una tienda de ropa, en donde se puede explorar la selección de artículos disponibles para comprar.



Los usuarios que inicien sesión pueden agregar los productos a un carrito para luego realizar la compra en línea, así como ver su historial de compras pasadas o modificar la información de su cuenta.

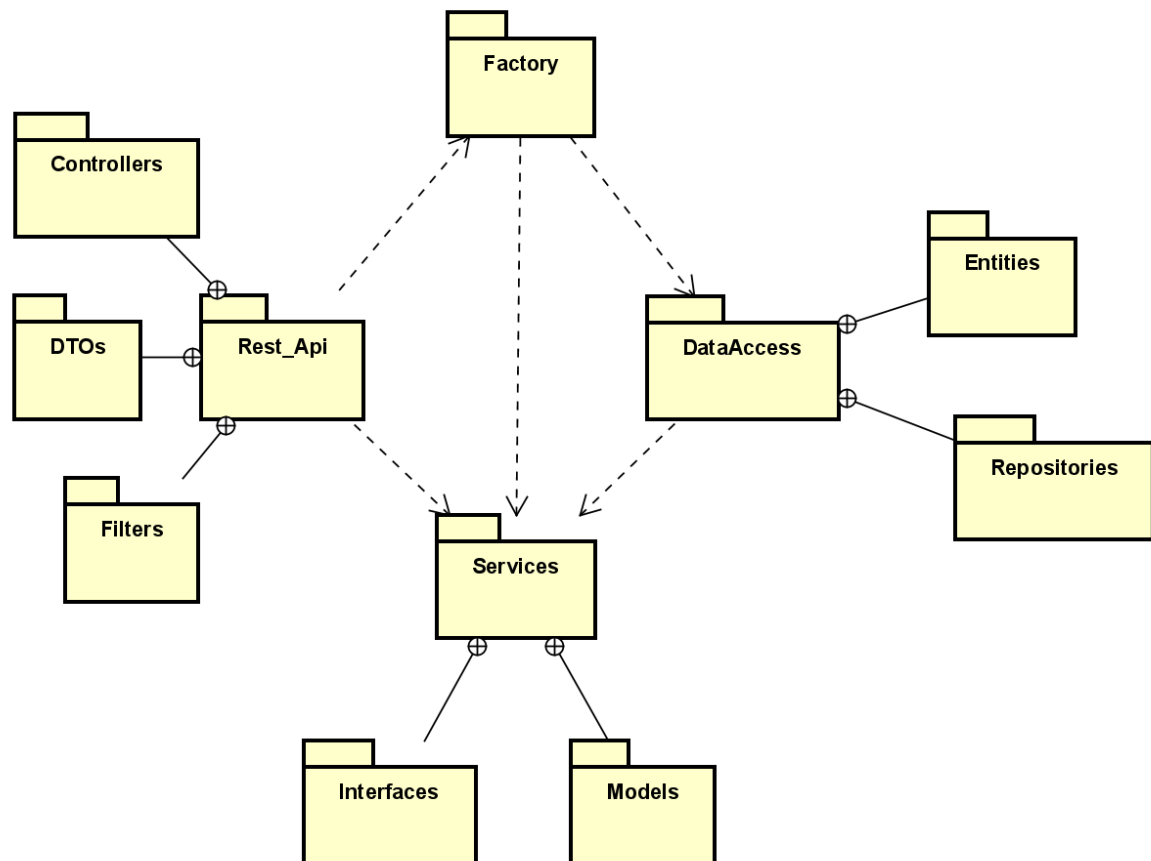
En adición, los usuarios con el rol de administrador pueden agregar, modificar y quitar productos. También pueden administrar las cuentas de los demás usuarios y ver todas las compras realizadas a través de la página.

Estas funciones se mantienen todas gracias a una comunicación mediante requests HTTP entre la aplicación frontend y el servidor backend, siendo intermediadas por la API previamente desarrollada. Aunque la API pasó por ciertos cambios y adiciones que surgieron al trabajar en el frontend.

Cerca del final del desarrollo hemos descubierto que si se retiran las .dll de las promociones y se trata de visualizar una compra en el historial, estas no muestran la promoción que fue utilizada. Una posible solución sería modificar el objeto Purchase para que además de guardar el objeto IPromo en sí, también guarde un string con el nombre de la promoción. De esta manera se podría mostrar el nombre guardado en la base de datos incluso si la promoción deja de estar en el sistema.

Diagrama general de paquetes

La aplicación backend quedó organizada de manera semejante a como lo estaba en la anterior entrega, sin grandes cambios. Se adjunta diagrama, ahora con la inclusión de los paquetes anidados:



Notamos que hay cuatro grandes paquetes (se explicarán más en profundidad posteriormente en el informe):

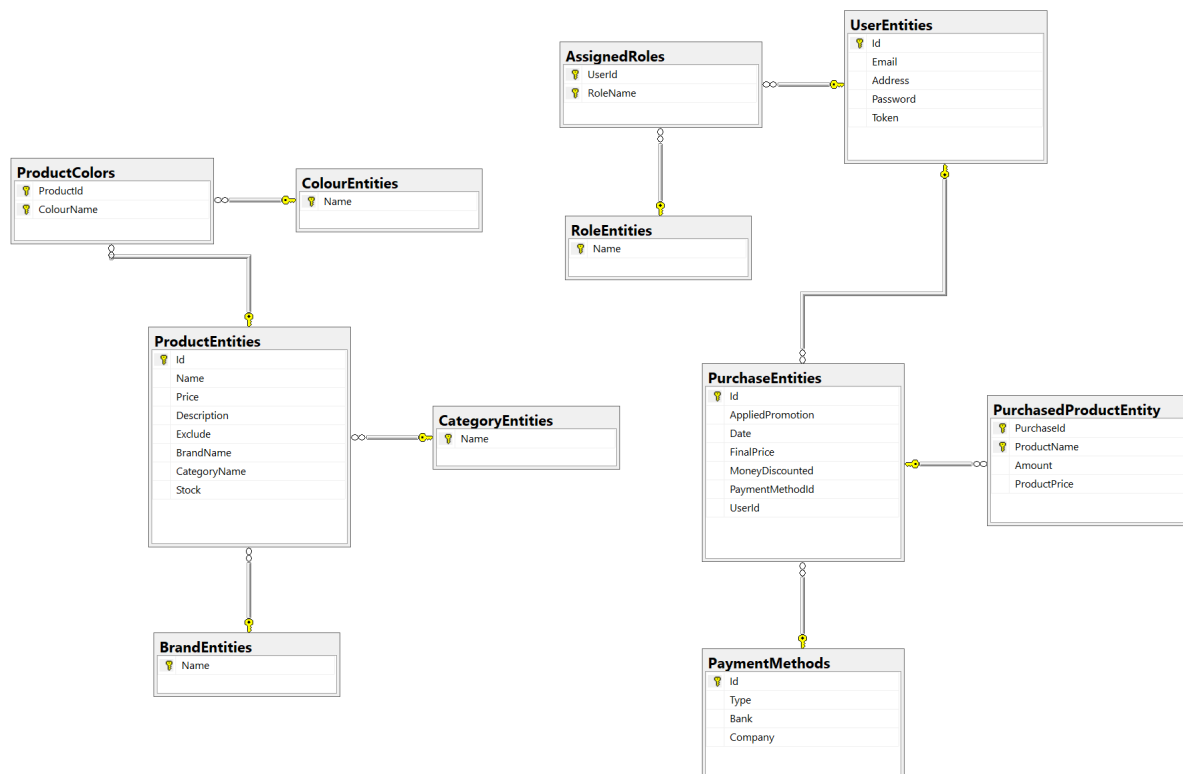
- **Factory** es el encargado de crear y suministrar todos los servicios y repositorios que se necesitan, permitiendo así la inyección de dependencias.
- **Services** brinda todos los recursos fundamentales que la API puede necesitar, como pueden ser modelos y servicios.
- **DataAccess** es el paquete encargado de realizar las consultas e inserciones a la base de datos.
- **Rest_Api** es el paquete “principal”, siendo la API en sí misma. Contiene los controladores que definen los endpoints, sobre los que luego se opera.

Vemos que cada paquete tiene una única responsabilidad, por lo que no se quebranta SRP.

Es interesante hablar sobre las dependencias que existen. Entendemos que Services es el paquete de mayor nivel, ya que plantea las bases del sistema. Debido a esto, para cumplir DIP, este paquete no puede depender de ninguno de menor nivel. Esto presenta un problema, ya que Services requiere acceder a base de datos, y para ello debe hacer uso de DataAccess, de menor nivel. Para solucionarlo, Services establece interfaces que usa, y que son implementadas por DataAccess, invirtiendo la dependencia.

Viendo el diagrama también notamos un par de cosas fundamentales. Primero, no hay dependencias cíclicas, por lo que se cumple ADP. Por otro lado, las dependencias tienden a ser de los paquetes menos estables a los que presentan más estabilidad y, a su vez, estos paquetes estables (principalmente Services), presentan abstracciones, cumpliendo también SDP y SAP respectivamente. Esto hace que se sigan correctamente los principios de acoplamiento de paquetes, llevando a una tendencia a la escalabilidad. Esto se tratará nuevamente más adelante.

Modelo de tablas de la estructura de la base de datos



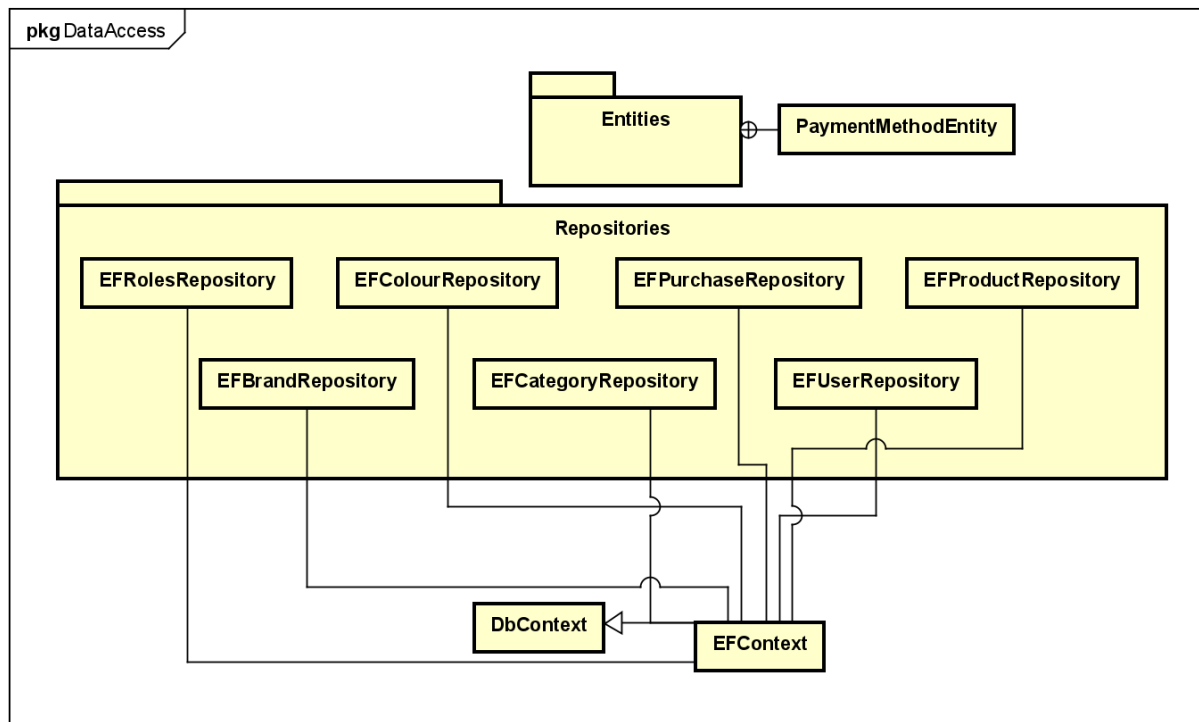
Vemos que la estructura no se vio muy afectada por las nuevas funciones. Al compararla con la estructura entregada en la primera instancia de obligatorio encontramos tres cambios únicamente. Por un lado, se incorporaron nuevos atributos a la tabla **ProductEntities**, siendo estos **stock** y **exclude**, que son fundamentales para el funcionamiento de estos nuevos requerimientos. Por otro lado, se agrega la tabla **PaymentMethods** con todos los datos que un método de pago puede tener, y su conexión a **PurchaseEntities**. La implementación de estos cambios será explicada más adelante.

Finalmente, quitamos la relación entre la tabla de **PurchasedProductEntity** y **ProductEntities**, para solucionar problemas de consistencia en las compras al modificar productos. Si uno de los productos comprados era eliminado, el sistema se caía al intentar visualizar la compra. Eliminar esta relación, guardando los productos comprados como strings en vez de referenciarlos a la tabla **ProductEntities**, hace que este problema no exista, ya que si se elimina el producto en cuestión siempre se puede leer el string.

Explicación de cada paquete individual

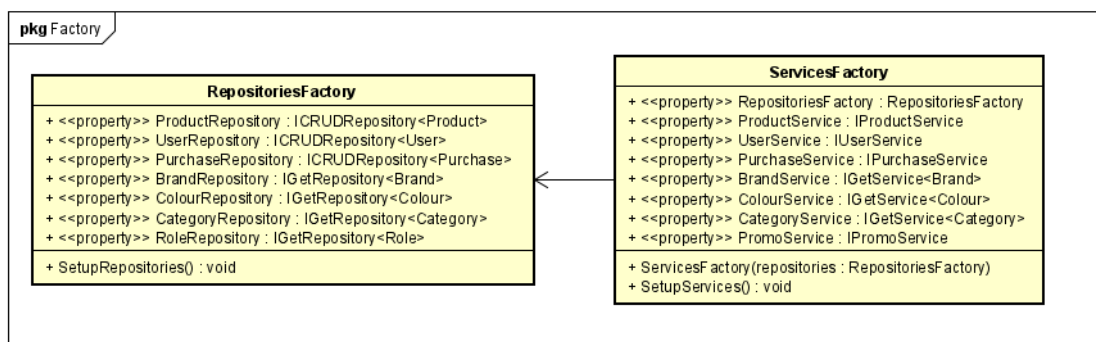
DataAccess

Este paquete se ocupa del acceso a la base de datos. Se mantiene igual que para la entrega anterior, con la única adición de una nueva entidad a persistir: los métodos de pago.



Factory

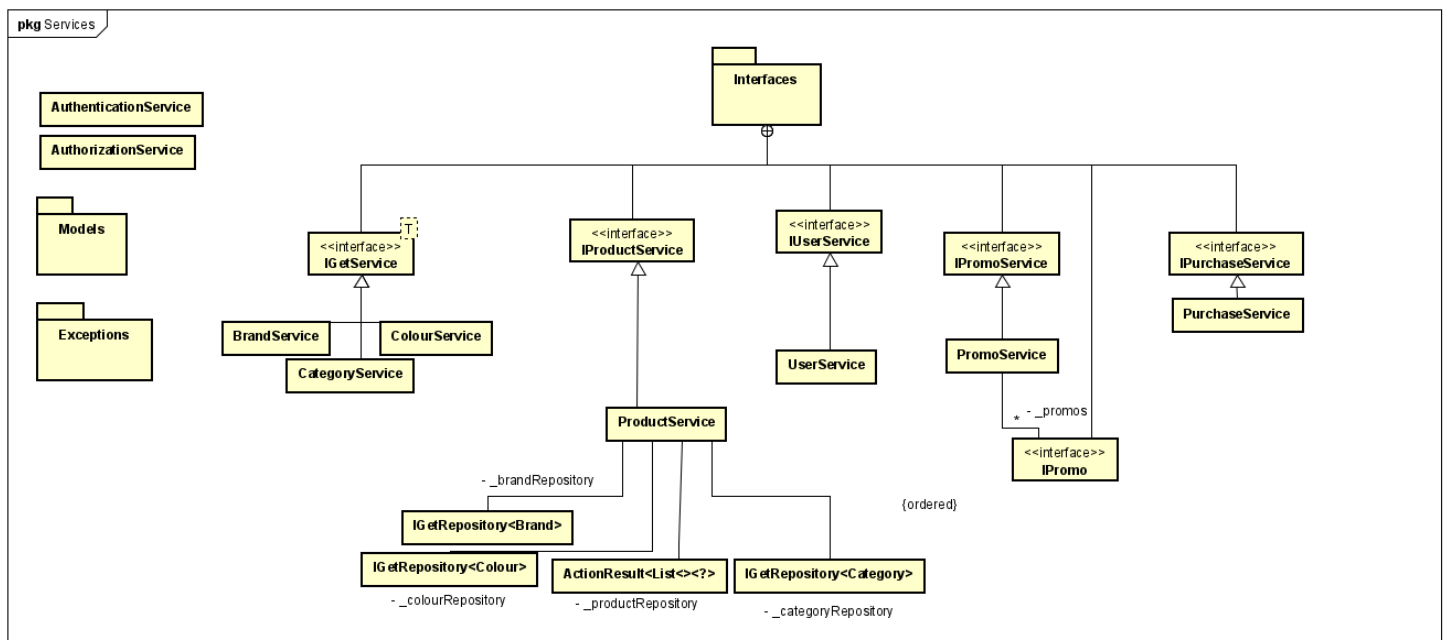
Este paquete no sufrió ninguna modificación, pues su única función es la de inyección de dependencias, instanciando los servicios y repositorios utilizados por la API. Cuenta con dos clases:



Aunque algunos servicios y repositorios sufrieron cambios, las clases y las relaciones se mantienen como surgieron en la primera entrega. Más adelante, en la sección de patrones, se hablará más en detalle del funcionamiento de este paquete.

Services

Este paquete es el que incluye las reglas de negocio, teniendo todos los modelos, interfaces y servicios necesarios para el correcto funcionamiento del sistema. Estos elementos se relacionan entre ellos de la siguiente manera:

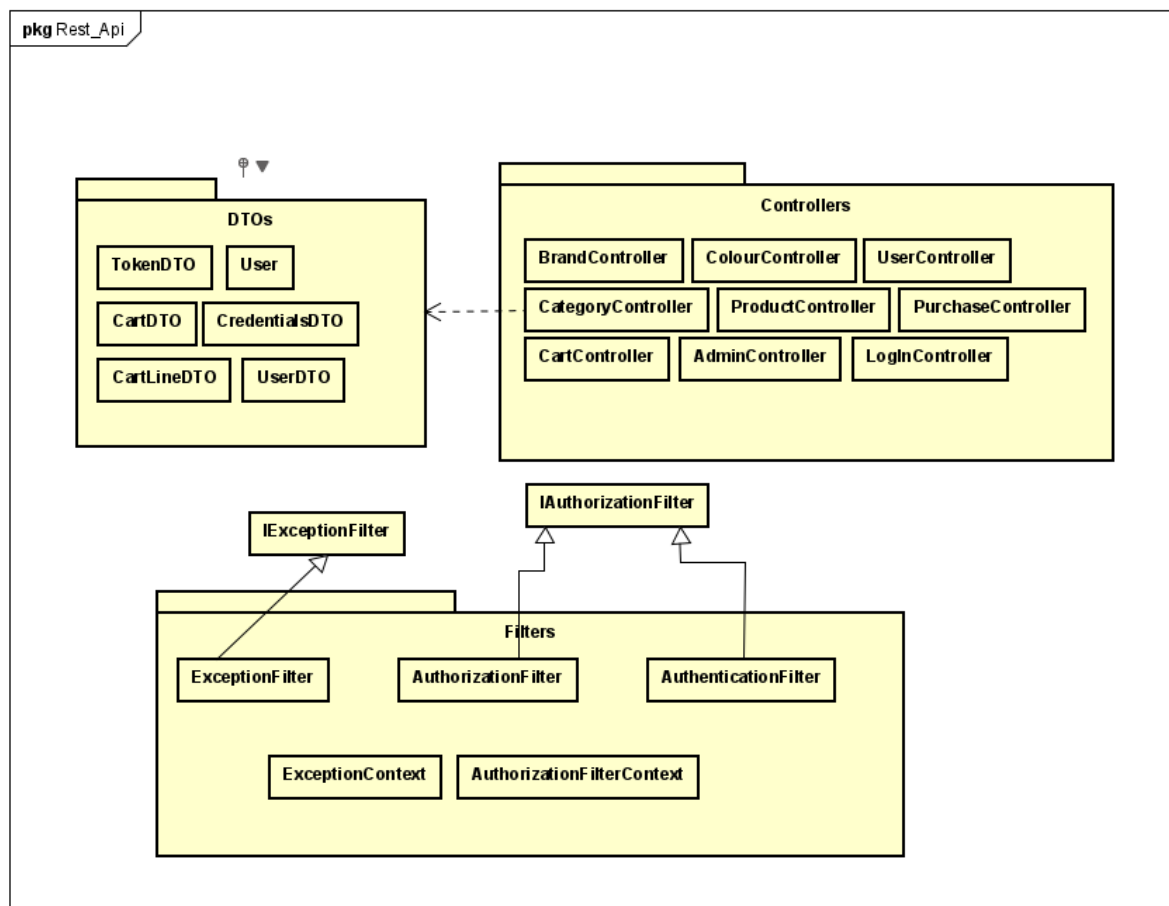


Como notamos, tampoco sufrió modificaciones importantes en sus relaciones con otros paquetes. Aún así, encontramos la nueva interfaz IPromo, implementada por todas las promociones ya disponibles y que debe ser implementada por cualquier nueva promoción que se desee agregar al sistema.

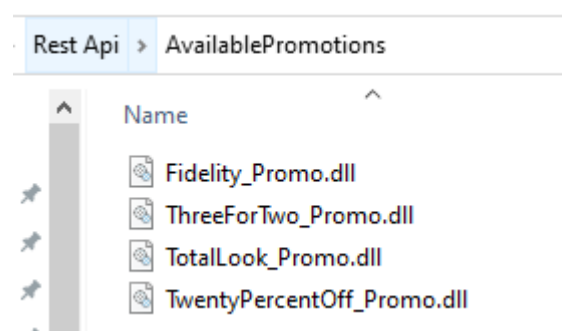
Paralelamente, el PromoService fue modificado para que utilice un mecanismo de *reflection* para leer un directorio específico en donde puede encontrar los ensamblados de las promociones ya compiladas en archivos .dll individuales, durante tiempo de ejecución. Esto nos permite activar y desactivar promociones mientras se ejecuta la aplicación, se expande este tema más adelante.

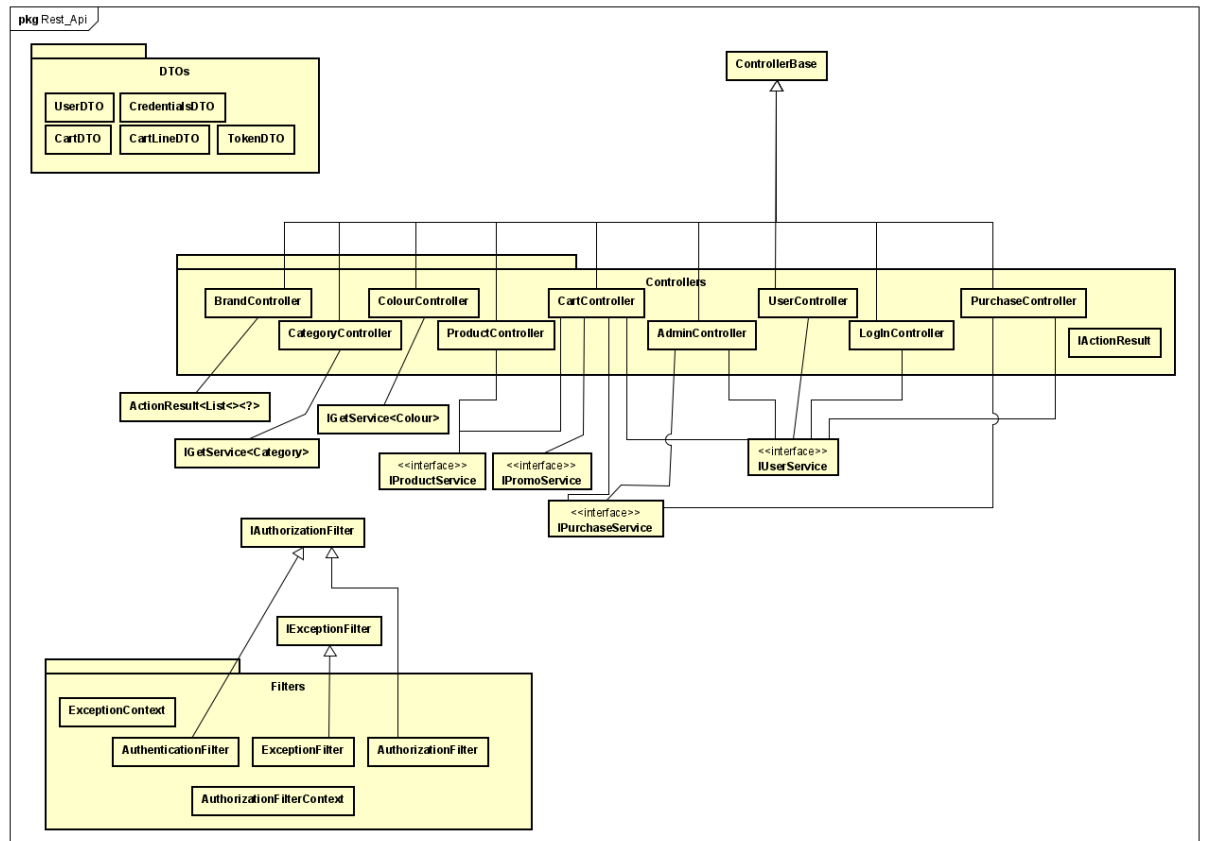
Rest Api

Este paquete se divide en tres grandes subpaquetes. Primero, tenemos uno para los DTOs. Estos son usados por los controladores, que a su vez implementan distintos filtros, permitiéndonos así controlar distintos factores, tales como si el usuario que realiza la request es Admin o no.



En el directorio de este paquete ahora encontramos una nueva carpeta utilizada como ubicación para guardar y acceder a los ensamblados que corresponden a cada promoción disponible para ser utilizada por la aplicación. En el directorio “AvailablePromotions” se pueden colocar o remover los archivos con la terminación “_Promo.dll”, y cada uno corresponde a una promoción para aplicar a las compras.





Nuevos Requerimientos

Promociones

El uso de promociones en la aplicación para aplicar descuentos a las compras realizadas evolucionó desde la última entrega. Con el propósito de permitir la expansión de las promociones disponibles, así como quitarlas de las aplicables a las compras, se implementó un algoritmo de Reflection para leer de un directorio las promociones disponibles.

En el proyecto Rest Api existe un directorio de nombre “AvailablePromotions” en donde se deben de colocar los archivos de extensión .dll que contiene compilada una clase que debe aplicarse como una promoción. El PromoService del proyecto luego leerá los archivos cuyo nombre termine con “_Promo.dll” dentro de ese directorio e intentará instanciar la promoción que dentro encuentre. Así, cada vez

que se vaya a realizar una compra, se vuelve a revisar las promociones disponibles en el momento, de forma de que la aplicación funcione de manera actualizada.

A continuación se describen en detalle las instrucciones para la creación de una clase que pueda ser utilizada apropiadamente como una promoción. La promoción a crear debe implementar esta interfaz IPromo, teniendo propiedades que detallen su nombre, la condición que debe tener el conjunto de productos para que se aplique la misma y de qué forma se aplicará.

```
public interface IPromo
{
    string Name { get; }
    string Discount { get; }
    string Condition { get; }
    double ApplyDiscount(ICart cart);
}
```

Apply Discount

Recibe(ICart cart): Recibe un objeto que implemente la interfaz ICart, que contiene únicamente el conjunto de productos participantes de promociones. Objetos excluidos de las promociones son removidos previamente, no es responsabilidad de la promoción saber esto.

Debe: Analizar los productos y si cumple con la condición descrita, calcular el precio dado el descuento planteado. El Cart es una copia del original, por lo que si se desea se puede manipular.

Devuelve double: Devuelve el precio final luego de aplicar la promoción.

Stock de producto

Para poder comenzar a manejar el stock asociado a un producto, los primeros cambios ocurrieron en las clases más básicas de Product y ProductEntity. Específicamente, se les agregó una nueva propiedad int Stock:

```

20 references
public class ProductEntity
{
    [Key]
    7 references
    public int Id { get; set; }
    11 references
    public string Name { get; set; }
    6 references
    public double Price { get; set; }
    6 references
    public string Description { get; set; }
    10 references
    public BrandEntity Brand { get; set; }
    10 references
    public CategoryEntity Category { get; set; }
    4 references
    public int Stock { get; set; }
    11 references
    public IList<ProductColors> Colours { get; set; }
}

```

```

public class Product
{
    52 references | 22/22 passing
    public Product()
    { }
    7 references
    public int Stock
    {
        get { return _stock; }
        set
        {
            if (value < 0)
            {
                throw new Models_ArgumentException("Product stock cannot be below zero");
            }
            _stock = value;
        }
    }
}

```

Luego, se modificaron los métodos Buy y Create del CartController para que den nuevas respuestas dependiendo de la situación del stock de un producto. Si se intenta comprar más stock del disponible se devolverá una BadRequest. Si se intenta crear un carrito con una cantidad de un producto mayor a la disponible se responderá con un código 206, dado que se devuelve un carrito pero modificado para ajustarse al stock disponible como máxima cantidad de un producto.

Si el carrito cumple las condiciones para completar la compra correctamente, la operación Buy también modificará el stock de los productos afectados en la base de datos.

Los siguientes métodos se encargan de modificar el stock de los productos y de verificar si se modifica el carrito en base al stock disponible:

```
[NonAction]
1 reference
public void ModifyProductStock(List<CartLine> cartLines)
{
    foreach(CartLine line in cartLines)
    {
        Product newStock = line.Product;
        newStock.Stock -= line.Quantity;
        _productService.Update(newStock);
    }
}

[NonAction]
1 reference
public bool ProductQuantitiesWereModified(List<CartLineDTO> cartLineDTOs, List<CartLine> cartLines)
{
    foreach(CartLineDTO line in cartLineDTOs)
    {
        CartLine respectiveCartline = cartLines.Find(c => c.Product.Id == line.Id);
        if(respectiveCartline.Quantity != line.Quantity)
        {
            return true;
        }
    }

    return false;
}
```

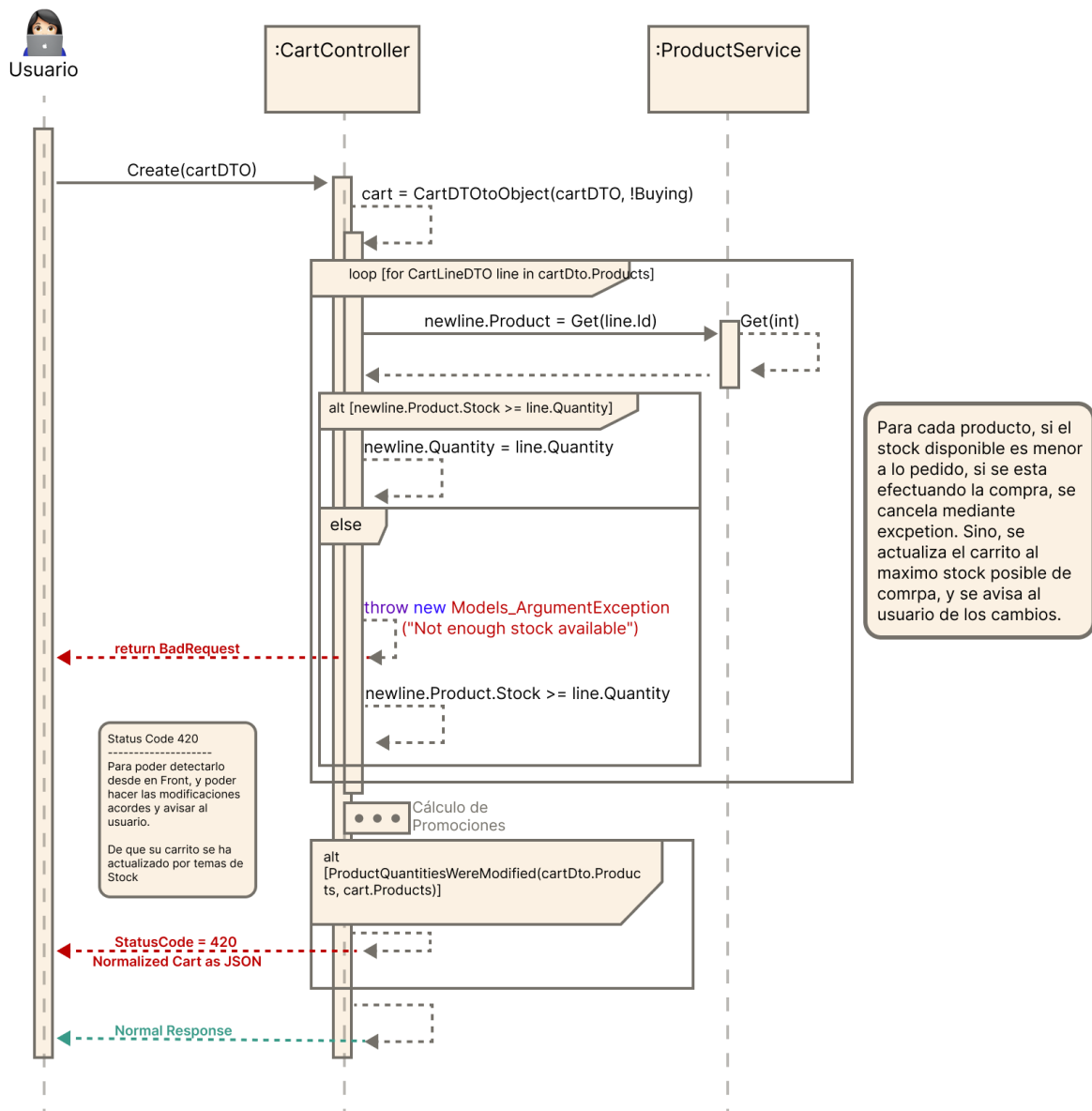
La siguiente captura muestra como en el algoritmo del método Create se retorna un 420 en el caso de que se haya modificado el carrito para el cliente debido a insuficiente stock. Al principio del desarrollo este era 206 Partial Content, pero no se logró detectarlo correctamente en el front end con Angular, por lo que definimos nuestro propio código 420, y le dimos un significado semántico propio a nuestro programa.

```
Cart cart = new Cart();
try
{
    cart = CartDTOtoObject(cartDto, false);
    cart = ApplyPromoToCart(cart);

    if(ProductQuantitiesWereModified(cartDto.Products, cart.Products))
    {
        return new ObjectResult(cart)
        {
            StatusCode = 420
        };
    }

    return CreatedAtAction(nameof(Create), cart.DiscountedPriceUYU, cart);
}
```

A continuación proveemos un diagrama de comunicación, donde se expande sobre la función que convierte DTOs a Cart explicada anteriormente. Este diagrama obvia el flujo de aplicar las promociones ya que este ya fue incluido en la documentación anterior.



Nuevos filtros de búsqueda

Para esta segunda entrega se agregó a la funcionalidad de filtrado de productos, pudiendo filtrar por nombre, precio, marca, categoría y por si están excluidos de las promociones o no. Debido a estas adiciones, se tienen un total de seis parámetros por los cuales se pueden filtrar. Consideramos que son muchos como para simplemente pasarlos por parámetro entre métodos, por lo que creamos una clase ProductFilterDTO para movilizarlos.


```

17 referencias | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
public class ProductFilterDTO
{
    6 referencias | 4/4 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public string? Category { get; set; }
    5 referencias | 3/3 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public string? Brand { get; set; }
    6 referencias | 3/3 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public string? Name { get; set; }
    3 referencias | 1/1 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public double? MinimumPrice { get; set; }
    3 referencias | 1/1 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public double? MaximumPrice { get; set; }
    3 referencias | 1/1 pasando | Fede PC, Hace menos de 5 minutos | 1 autor, 1 cambio
    public bool? ExcludedFromPromos { get; set; }
}

```

Además, los agregamos como un parámetro que puede llegar al endpoint Get /Products/, el cual se utiliza para obtener todos los productos disponibles. De esta manera, ahora el endpoint puede aceptar opcionalmente los query parameters que correspondan a términos para filtrar los productos, o traerlos todos si faltan filtros.

```

[HttpGet]
public ActionResult<List<Product>> GetAll
    ([FromQuery] ProductFilterDTO filters)
{
    if (!CheckFiltersAreNull(filters))
    {
        return _productService.GetFiltered(filters);
    }
    else
    {
        return _productService.GetAll();
    }
}

private bool CheckFiltersAreNull(ProductFilterDTO filters)
{
    var properties = typeof(ProductFilterDTO).GetProperties();
    return properties.All(property => property.GetValue(filters) == null);
}

```

Con este método también pudimos deshacernos del endpoint Get/filteredProducts que se utilizaba en la entrega anterior, a favor de este que se atiene mejor a los principios de diseño de una API Rest.

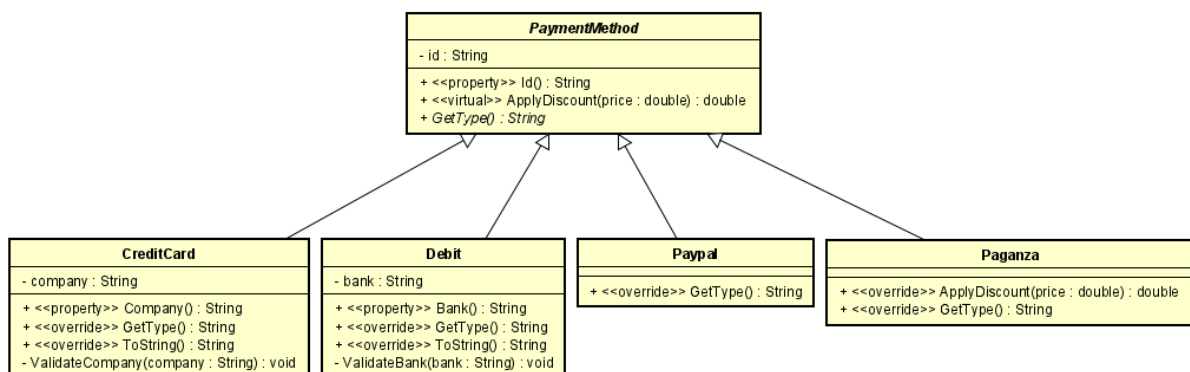
Métodos de pago

Para la implementación de los métodos de pago, se crearon 5 clases, la clase “padre” PaymentMethod y las 4 clases “hijas”, que representan cada uno de los métodos de pago aceptados en esta entrega (Paypal, Paganza, tarjeta de crédito y

débito bancario). Este enfoque también hace que se cumpla OCP. `PaymentMethod` es una clase abstracta que define que tendrá todo método de pago, cumpliendo así LSP. Todos, sin importar el tipo, tienen un id, que puede ser el número de tarjeta, o de cuenta bancaria, etc.

`PaymentMethod` también presenta un método virtual llamado `ApplyDiscount`. Esto nos permite que cada uno de los métodos de pago pueda tener su propio descuento. Por ejemplo, `Paganza` tiene un 10% de descuento, por lo que `Paganza` hace un override de `ApplyDiscount` y toma el precio final como el 90% del total. Si, en un futuro, hubiera otro descuento con otro método de pago, simplemente se hace un override de esta función en la clase de ese método y se implementa.

El siguiente diagrama puede simplificar la comprensión (nota, el método `GetType` simplemente devuelve un string con el tipo de `PaymentMethod`, simplifica a la hora de crear la entidad a persistir en base de datos):



El controlador de `Cart` tiene un método `Buy` por el que se realizan las compras. Este recibe un `CartDTO`. Lo que se hizo fue que este DTO reciba toda la información que un método de pago pueda tener (tipo, id, banco y compañía). Todos estos datos son opcionales, ya que pueden no ser necesarios (paypal no tiene banco, para ejemplificar) y pueden llegar a ser nulos (si uso el DTO para ver el precio, pero aún no voy a comprar, no necesito pasarle `PaymentMethod`).

Posteriormente, al convertir el `CartDTO` a `Cart`, se instancia el método de pago y se almacena en el carrito, para luego asignarlo también en la `Purchase`. Al persistir la `Purchase` en la base de datos, se guarda el ID del método de pago,

teniendo una tabla para almacenar todos los métodos de pago usados. Esta tabla guarda todos los 4 datos que un método de pago puede tener, por lo que es TPH.

Al realizar esta implementación, se vieron afectados todos los cuatro paquetes. Se afectó ApiTests ya que se añadieron tests de las nuevas funcionalidades, siguiendo las normas de TDD; DataAccess al tener agregar la tabla de métodos de pago y añadir nueva columna a la de Purchases; también se modificó el controlador de Cart y su DTO, impactando en RestApi y se crearon nuevos modelos, además de modificar algunos ya existentes, modificando así Services.

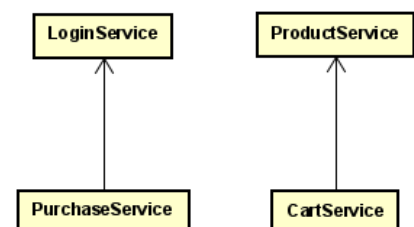
Frontend

El principal requerimiento de esta nueva entrega fue la implementación de un frontend que nos permita hacer uso de la API desarrollada. Esta nueva aplicación fue desarrollada con Angular y, pese a que no se solicita en este informe, creemos valioso explicar los puntos principales de su desarrollo.

Services

El frontend realiza requests al backend por medio de la API. Estas requests se llevan a cabo en servicios especializados para cada request, dependiendo de sobre qué recursos busca operar.

Particularmente se tienen cuatro Services: ProductService, PurchaseService, CartService y LoginService, que se ocupa del manejo de los inicios de sesión y del resto de operaciones que se pueden hacer con los usuarios. Estos servicios pueden relacionarse entre ellos. Por ejemplo, CartService necesita hacer uso de ProductService para poder conocer y operar con los productos.



Routing

Para realizar requests a través de los Services, el usuario interactúa con distintos componentes. Entre estos componentes encontramos formularios, cards y menús con los que el usuario puede hacer acciones que le permitan usar el sistema.

El usuario navega entre componentes haciendo uso de un router. Se definieron diversas rutas que llevan a cada uno de los componentes posibles. Ciertas acciones, como puede ser tocar un botón, volver al menú principal o perder la conexión con el backend, hará que el router nos lleve a una nueva ruta, permitiéndonos visualizar un nuevo componente con el que interactuar.

Guards

Algunas de las rutas definidas tienen asignadas la guarda AdminGuard. Este guard fue creado con el propósito de brindar más seguridad. El módulo de Admins es, como su nombre indica, exclusivo para administradores. Debido a esto, se le asignó AdminGuard a todas las rutas a las que un usuario no Admin no debería tener acceso. Si un Admin accede a estas rutas no habrá problema, pero si alguien que carece el rol intenta acceder, no podrá.

Patrones Aplicados

Patrón Adapter

Se consideró aplicar este patrón para convertir los DTOs que se reciben en los controladores a los objetos finales que manejan los servicios. Bajo esta mentalidad, el controlador recibiría el DTO y se lo enviaría al adapter, que sería el responsable de transformarlo al modelo y realizar todas las operaciones necesarias sobre él. Con esto logramos que el controlador se abstraiga completamente de los servicios, ya que esto será responsabilidad del adaptador.

No obstante, la idea fue descartada, ya que realmente no es tan útil como se pensó inicialmente. Al no estar trabajando con librerías externas, sino que con clases creadas por nosotros mismos, y que además están protegidas a través de interfaces, no nos brinda mayor protección ante posibles cambios. Adicionalmente,

esa abstracción de la que hablamos en el párrafo anterior simplemente se movería a otra capa, complejizando el sistema sin aportar muchas ventajas. El controlador dependería del adaptador (en lugar del servicio), y el que dependería del servicio sería ahora el adaptador, generando un pasamano que no mejora el acoplamiento pero si perjudica la performance y lo hace menos simple, afectando más de lo que aporta.

Patrón Abstract Factory

Pese a que este patrón ya fue usado y explicado para la primera entrega, desde entonces se expandió el conocimiento teórico dado en clase, permitiendo comprender mejor la solución aplicada en el trabajo. La situación es la siguiente: ¿dónde deberíamos instanciar los repositorios y servicios que se usarán a lo largo del proyecto?

Para solucionar esto, se decidió aplicar una suerte de patrón Factory, teniendo una clase que se ocupa de instanciar y almacenar los repositorios y otra para hacer lo propio con los servicios. Aquí está lo interesante, pese a que su funcionalidad es la misma, técnicamente son patrones distintos. La Factory de repositorios es más tradicional, inicializa los objetos necesarios y los devuelve cuándo se le solicitan.

Ahora, esta Factory es usada por la de los Services, ya que la mayoría de estos requieren repositorios para ser creados. Entonces, en estos casos, ServicesFactory toma el repositorio pertinente de RepositoriesFactory y lo asigna en el nuevo objeto creado, convirtiéndola en algo más parecido a una Abstract Factory. El siguiente diagrama, que muestra cómo se instancian los repositorios y servicios, puede servir como ejemplo del funcionamiento general de esta Factory Abstracta:

entrega fue satisfactorio, no se realizaron mejoras extremas para esta nueva versión. No obstante, hubo algunos cambios.

Un cambio importante fue la incorporación de las nuevas funcionalidades solicitadas. Estas mejoran el sistema y aportan mayor cantidad de casos de uso. La implementación de estos nuevos requerimientos no presentó grandes complejidades, siendo su desarrollo ameno. Esto demuestra que nuestro diseño original es prolijo y extensible.

Entre estos nuevos requerimientos encontramos la habilidad de “desactivar” promociones mientras la aplicación corre. Para esto se usó Reflections, como ya se explicó. Ahora, para esta implementación fue necesario incluir nuevas abstracciones, interfaces para promos y todo lo que estas usan. Este aumento de abstracciones en un paquete estable, tal como es Services, fomenta el principio SAP, siendo así una nueva mejora en el diseño.

La implementación de un historial de compras trajo consigo un dilema: si se borra el usuario que la realizó o uno de los productos adquiridos, esta información faltará y no se mostrará en la aplicación. Esto se debe a que la persistencia de las compras en la base de datos se mantienen mediante relaciones a las entidades del usuario y los productos pertinentes, por lo que un cambio en estas entidades afectará al historial de compras.

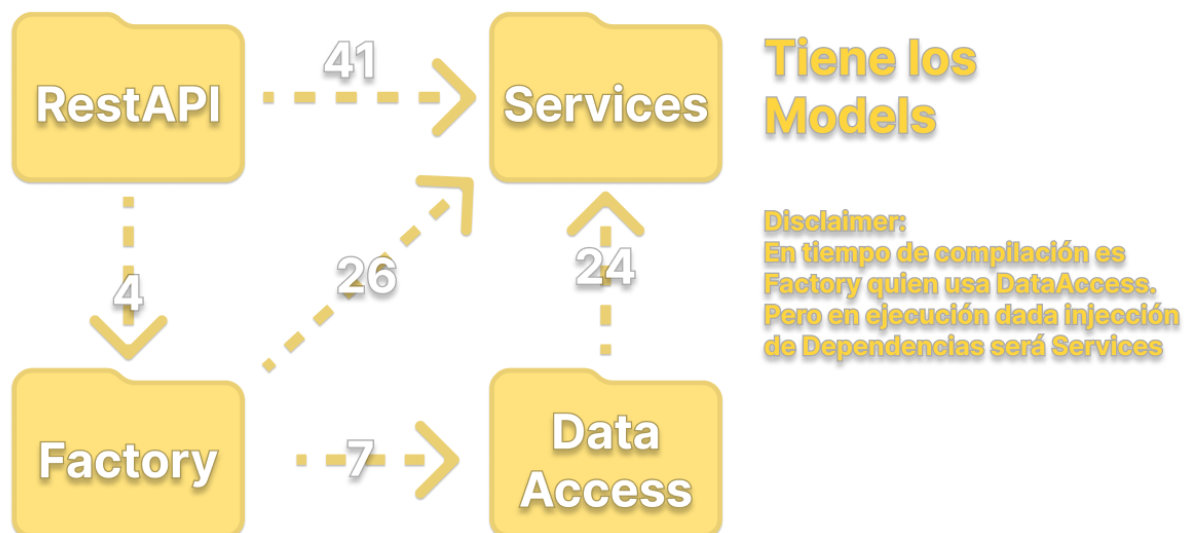
Este problema fue solucionado parcialmente. Las compras aún se mantienen relacionadas con la entidad del usuario que la realizó, por lo que borrar el usuario provocaría que a su vez se borren sus compras. Pero en el caso de los productos, ahora en lugar de haber una relación con la entidad, se guardan los datos necesarios como datos primitivos (un string para el nombre y un double para el precio). Por lo que cualquier cambio en la entidad del producto no afectará a las compras que estarían relacionadas.

Otro aspecto que admitió mejoras con respecto a la entrega del primer obligatorio fué la fusión de los endpoints GET de Product. En la primera instancia existían dos endpoints para conseguir los datos de un producto, siendo uno general que devolvía todos los productos, y otro que filtraba según distintos criterios, como marca, nombre, entre otros. Esto no es muy bueno, ya que podría unificarse en un

único endpoint. Este recibiría todos los criterios de búsqueda posibles, a través de query params. Al ser opcionales, poner cualquier parámetro de búsqueda filtraría los productos acordemente, mientras que no poner ninguno traería todos los productos de la base de datos. Quedando ambas funcionalidades en un mismo endpoint nemotécnico de Get/Products.

Análisis de métricas y aplicación de principios

Cálculo a Mano



La razón principal de esta división es separar el código estrictamente relacionado con los controladores de la api del dominio (lógica del negocio), como a su vez desacoplar el dominio del mundo de Entity Framework. Y por último de forma inevitable la inyección de dependencias trabaja en su propio paquete Factory en conjunto con el program.cs (Main) dentro de RestAPI.

El conteo de dependencias fue realizado a mano, por lo que puede haber algún mínimo error.

Paquete	R	N	H	Ca	Ce	I	Na	Nc	A	D'	
RestAPI	22	17	1,4	0	45	1,0	0	17	0,0	0,00	Zona de Dolor
Services	0	38	0,0	58	0	0	13	38	0,3	0,47	
Factory	0	2	0,5	4	33	0,9	0	2	0,0	0,71	Zona de Dolor
DataAccess	22	18	1,3	7	24	0,8	0	18	0,0	0,16	Zona de Dolor

A su vez, Visual Studio nos provee un propio análisis de código, donde se genera un índice de mantenibilidad. En forma general, nuestros paquetes tienen un score por arriba del 80

Code Metrics Results							
Filter: None							
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code	
Factory (Debug)	89	33	1	31	65	16	
Services (Debug)	88	304	2	37	1,399	375	
Services.Exception	100	3	2	1	19	0	
Services.Interface	100	30	0	6	98	0	
Services.Models.I	100	12	1	1	13	0	
Services.Models.I	100	1	2	1	6	0	
Services.Models.I	91	22	2	3	110	20	
Services.Models	84	165	2	20	613	187	
Services	79	60	1	30	479	143	
Services.Models.I	71	11	1	5	61	25	
EFTests (Debug)	83	10	1	29	162	51	
DataAccess (Debug)	81	215	2	66	1,097	321	
DataAccess.Entity	90	133	1	46	387	53	
DataAccess	89	24	2	27	58	20	
DataAccess.Repo	68	58	1	39	652	248	
Rest Api (Debug)	80	125	2	105	875	269	
Program	35	4	1	44	99	80	
Rest_Api.DTOs	97	33	1	2	69	6	
Rest_Api.Controll	78	77	2	64	609	167	
Rest_Api.Filters	78	11	2	18	98	16	
ApiTests (Debug)	68	191	1	110	3,591	1,379	

Al comienzo del semestre hemos utilizado incluso sin aún saberlo CCP, Common Closure Principle, donde ubicamos dentro del mismo paquete aquellas clases que cambian juntas.

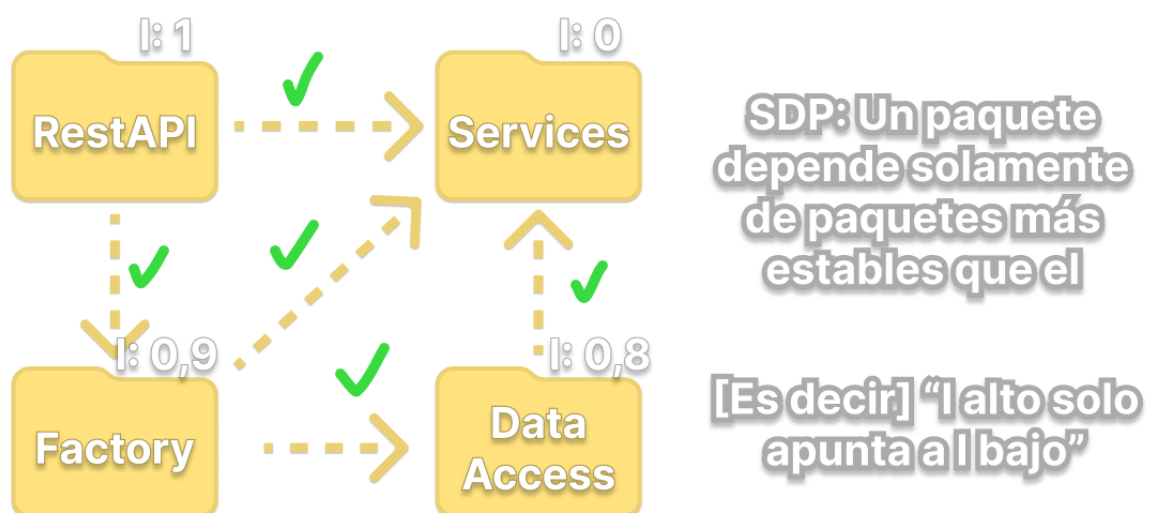
Luego fuimos haciendo algunos cambios en relación a los subpaquetes para poder cumplir mejor

Code Metrics Results	
Filter: None	
Hierarchy	Maintainability Index
Factory (Debug)	89
Services (Debug)	88
EFTests (Debug)	83
DataAccess (Debug)	81
Rest Api (Debug)	80
ApiTests (Debug)	68

con REP y CRP. Ubicado en namespaces distintos aquellas clases que sean reutilizadas en conjunto, como por ejemplo Services.Models.

En cuanto a principios de paquetes, no presentamos dependencias cíclicas, por lo que cumplimos con ADP - Acyclic Dependencies Principle. Este es obligatorio y necesario para la compilación del proyecto.

Podemos también asegurar gracias a las métricas tomadas en la tabla de arriba que cumplir con SDP - Stable Dependencies Principle.

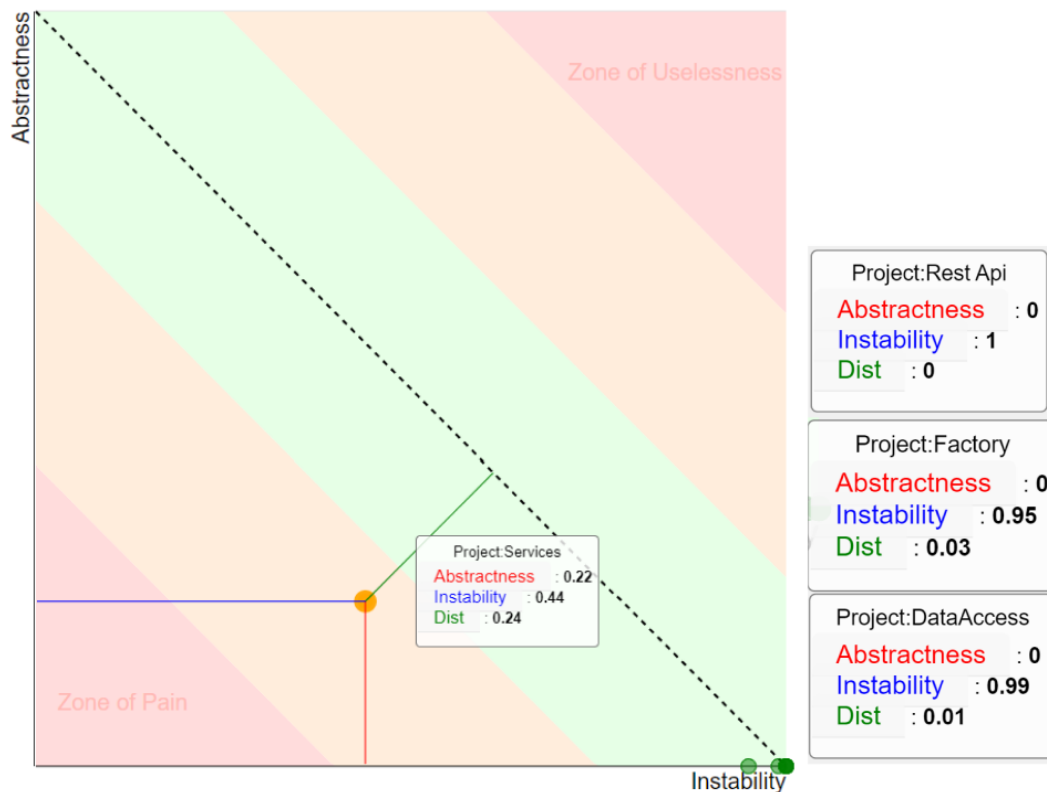


Y por último en la lista de principios tenemos a SAP - Stable Abstraction Dependencies, que nos dice que un paquete debe ser tan abstracto como estable es. Aquellas de baja abstracción, son efectivamente poco estables. Y la única que contiene abstracciones, Services, con 0,3. Es la más estable entre ellas.

Paquete	Abstracción	Estabilidad
RestAPI	0,0	0,0
Services	0,3	1
Factory	0,0	0,1
DataAccess	0,0	0,2

Pero además de un cálculo a mano, utilizamos una herramienta de análisis de código estático llamada NDepend. NDepend es una extensión para Visual Studio que permite realizar un análisis de calidad de código en entornos .NET. Proporciona información detallada sobre la calidad, la arquitectura, y las métricas asociadas.

Cálculo mediante NDepend



Paquete	H	Ca	Ce	I		Na	Nc	A	D'
RestAPI	2.96	0	117	1.0		0	17	0.0	0.00
Services	3.87	34	0	0.44		13	38	0.22	0.24
Factory	1.6	2	39	0.95		0	2	0.0	0.03
DataAccess	3.42	1	117	0.99		0	18	0.0	0.01

Vemos que la principal diferencia es la cantidad de acoplamientos eferentes y aferentes que se encuentran. Esto probablemente se deba a que tome en cuenta relaciones con de terceros utilizadas por el proyecto, y nativas a C#/.Net, más allá de las nuestras propias.

Anexo

Especificación de endpoints nuevos o modificados

Get Cart Price and Info

- Descripción: Devuelve datos de los productos en el carrito, el precio sin descuento, y el precio con descuento si aplica, y qué descuento.
- Verbo: Post
- URI: /Cart
- Headers:
- Body:

```
{  
  "products": [  
    {  
      "id": 0,  
      "quantity": 0  
    }  
  ]  
}
```

- Ejemplo: Post/User

```
{  
  "products": [  
    {  
      "id": 1,  
      "quantity": 3  
    },  
    {  
      "id": 2,  
      "quantity": 1  
    }  
  ]  
}
```

```
}
```

- Response:
 - 201: Created, se retorna la información del carrito.
 - 400: BadRequest, en caso de que el carro esté vacío, el id es incorrecto, o cantidades menores o iguales a cero.
 - 420: En caso que la cantidad de productos cambie por cambio de stock. Este cambio surge dado que antes usábamos un código 2xx que no era detectado como error desde el front end y no nos permitía actualizar la lista y avisar al usuario.

Buy Cart

- Descripción: Toma un carrito y lo guarda como una compra a nombre del usuario.
- Verbo: Post
- URI: /Cart/buy
- Headers: Auth → string
- Body:

```
{  
  "products": [  
    {  
      "id": 0,  
      "quantity": 0  
    }  
  ],  
  "paymentMethod": "string",  
  "paymentId": "string",  
  "bank": "string",  
  "company": "string"  
}
```

- Ejemplo: Post/User

```
{  
  "products": [  
    {  
      "id": 5,  
      "quantity": 2  
    }  
  ],  
  "paymentMethod": "Paganza",  
  "paymentId": "21351351365"  
}
```

- Response:
 - 200: Success, la compra fue realizada con éxito.
 - 400: BadRequest, en caso de que el carro esté vacío, el id es incorrecto, o cantidades menores o iguales a cero.
 - 403: Forbidden, el usuario no tiene el rol Customer.

Filter Products y Get Products

- Previamente se mantenían dos endpoints separados para obtener todos los productos o los productos filtrados por ciertos parámetros. Ahora son un único endpoint que puede recibir los términos de filtrado como query parameters.
- Descripción: Retorna el o los Product que cumplan los criterios de filtrado, o todos los productos disponibles si no se adjuntan parámetros para filtrar.
-
- Verbo: Get
- URI: /Product
- Headers: {}

- Body: {}
- Ejemplo:
Get/Product?Category=Shorts&Name=Bat&MinimumPrice=100&ExcludedFromPromos=true
- Response:
 - 200: Success, se retornan todos los productos que cumplen las condiciones.

Cobertura de Pruebas

Vemos en la siguiente imagen que la cobertura de código de las pruebas se mantuvo por encima del 90% de las líneas de código para el backend y la Api. Exceptuando por el archivo de inicio Program.cs que baja la cobertura del paquete de Rest Api al 75%.

Hierarchy	Covered (%Lines) ▼
<ul style="list-style-type: none"> <ul style="list-style-type: none"> Federico Rodriguez_DESKTOP-52LSNL1_2023-11-15.20_12_25.coverage apitests.dll services.dll rest api.dll <ul style="list-style-type: none"> { } Rest_Api.Filters { } Rest_Api.Controllers { } Rest_Api.DTOs { } Clases globales <ul style="list-style-type: none"> Program Program.<>c Program.<>c__DisplayClass0_0 	<ul style="list-style-type: none"> 94,01% 98,20% 91,97% 75,30% 100,00% 94,56% 90,48% 0,00% 0,00% 0,00% 0,00%

Dentro del paquete Services, el único sub namespace con menos de 90% de cobertura en líneas de códigos es el nuevo agregado en esta entrega de PaymentMethods. Pero lo consideraremos aceptable dado que los únicos métodos no testeados son simples Get:

{ }	Services.Models.PaymentMethods	77,42%
✚	Debit	84,62%
📦	get_Bank()	100,00%
📦	set_Bank(string)	100,00%
📦	ValidateBank(string)	100,00%
📦	GetType()	0,00%
📦	ToString()	0,00%
✚	CreditCard	83,33%
📦	get_Company()	100,00%
📦	set_Company(string)	100,00%
📦	ValidateCompany(string)	100,00%
📦	GetType()	0,00%
📦	ToString()	0,00%
✚	PaymentMethod	66,67%
📦	set_Id(string)	100,00%
📦	ApplyDiscount(double)	100,00%
📦	get_Id()	0,00%
✚	Paganza	50,00%
📦	ApplyDiscount(double)	100,00%
📦	GetType()	0,00%
✚	Paypal	0,00%
📦	GetType()	0,00%

```
public override string GetType()
{
    return "Paganza";
}
```

```
public override string GetType()
{
    return "CreditCard";
}
```

0 referencias | Manuel Morandi, Hace 34 días | 1 autor, 1 cambio

```
public override string ToString()
{
    return Company;
}
```