Universidad ORT Uruguay

Obligatorio 1

Diseño de Aplicaciones 2

Descripción del Diseño

Martin Edelman - 263630

Tatiana Poznanski - 221056

Tomas Bañales - 239825

Profesores: Nicolás Fierro, Alexander Wieler, Marco Fiorito

2023

Link al repositorio:

https://github.com/IngSoft-DA2-2023-2/263630_221056_239825.git

1. Link al repositorio

https://github.com/IngSoft-DA2-2023-2/263630_221056_239825.git

2. Declaración de autoría

Nosotros, Tatiana Poznanski, Martin Edelman y Tomas Bañales declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

1. La obra fue producida en su totalidad mientras cursamos Marketing de Tecnologías.
2. Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad.
3. Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra.
4. En la obra, hemos acusado recibo de las ayudas recibidas.
5. Cuando la obra se basa en el trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y que fue construido por nosotros.
6. Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

3. Abstract

El proyecto fue desarrollado en Visual Studio 2022, más precisamente en el lenguaje C#. Durante toda la implementación, se aplicaron las prácticas de TDD (Test Driven Development) y Clean Code. Asimismo, utilizamos Microsoft SQL Server Express 2019 para almacenar los datos de la aplicación.

Con el fin de agregar nuevas funcionalidades al código, se emplearon las herramientas de GitHub y se siguió un flujo de trabajo de git. A medida que se deseaba trabajar en nuevas características, se fueron creando ramas (branches), una vez completadas, las mismas se fusionaban con la rama principal llamada "develop" para mantener el código actualizado.

Asimismo, identificamos tres componentes clave en el código y aplicamos la metodología GitFlow para dividir el trabajo de manera efectiva. La asignación de tareas se llevó a cabo de la siguiente manera:

- Martín Edelman asumió la responsabilidad de trabajar en el dominio, los servicios, el DataAcces y los controladores del usuario y de las compras.
- Tomás Bañales se encargó de desarrollar el dominio, los servicios, el DataAcces y los controladores de los productos y de las compras.
- Tatiana Poznanski asume el trabajo de encargarse de desarrollar e integrar las promociones en todo el sistema, asegurando que funcionaran de manera efectiva y se integraran sin problemas con las funcionalidades existentes.

4. Descripción general del sistema

Desarrollamos un ecommerce para tiendas de moda en Uruguay que aborda la implementación de promociones. El mismo permite a los usuarios agregar productos al carrito y, en función de los mismos, la plataforma aplica la promoción correspondiente y calcula el precio total previo a finalizar la compra. Asimismo, los usuarios tienen la capacidad de visualizar la lista de todos los productos, aplicando filtros basados en texto, marca o categoría.

Para el desarrollo de nuestro sistema hicimos uso de Entity Framework Core junto con la interacción de SQL Server para la base de datos y ASP .Net para la aplicación de las APIs.

Con el fin de reducir el acoplamiento y aumentar la cohesión, implementamos 4 módulos principales (Api, Servicios, DataAccess, Dominio), los cuales contribuyen significativamente a la aplicación, y otros 3, encargados de las pruebas unitarias, la inyección de dependencias y la migración de datos a la base de datos.

4.1 Instalación

La instalación del sistema se logra instalando una serie de paquetes de NuGet y configurando el connection string dentro del appsettings.json.

Los paquetes a instalar son los siguientes:

- Microsoft.AspNetCore.Authentication.JwtBearer
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.InMemory
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

- Microsoft.Extensions.DependencyInjection
- MSTest.TestFramework
- Swashbuckle.AspNetCore
- Swashbuckle.AspNetCore.Swagger
- Newtonsoft.Json

Después de instalar los paquetes, procedemos a configurar el appsettings.json en el proyecto de API y DataAccess, en estos archivos se agrega el connection string en el apartado de EcommerceDB.

```
"ConnectionStrings": {
  "ECommerceDB": "Server=127.0.0.1,1433; Database=[ElNombreDeTuBD]; User=sa; Password=[TuContrasena]; TrustServerCertificate=True"
},
```

De esta manera, estamos realizando el connection string en un archivo de configuración, lo que nos permite cambiar la conexión en tiempo de ejecución.

Además, se debe actualizar la migration. Esto se logra en la terminal del proyecto, entrando al proyecto de migrations y ejecutando el comando:

$ dotnet ef update -p ../DataAccess

4.2 Justificaciones de diseño

A continuación detallaremos, en cada proyecto, las justificaciones de diseño.

En primer lugar, en el dominio existen dos paquetes: uno de productos y otro de usuario, en el que se encuentra la clase Usuario.cs, donde el id y el correo electrónico son valores únicos. Además, dicha clase contiene un valor rol de entidad, “CategoriaRol”, el cual indica si el usuario es Administrador, Cliente o ambas. Dentro del mismo paquete, se encuentra el Enum de roles en CategoriaRol.cs.

En el paquete producto, se decidió establecer la entidad producto con vínculos a Marca, Categoria, Compra y Color. Estas relaciones representan los atributos de un producto real en una página web de venta. Se separaron las funcionalidades en capas,

respetando siempre los propósitos de las mismas a la hora de decidir en cuál se ubicaría cada porción de lógica.

Tanto en la clase de servicio como la de repositorio, se diseñó una interfaz para que cada una implemente. Esto permite seguir el principio DIP de SOLID, ya que en cada inyección de dependencias se apunta a depender de la interfaz, en vez de centrarse en la instancia.

A la hora de vincular los colores al producto en la base de datos, se creó un método que vincula los colores registrados en la base de datos con los de la lista del producto siendo añadido. A pesar de ser una solución ineficiente a gran escala, se considera que para el alcance de esta entrega es un camino válido de resolver el problema de la vinculación.

Con respecto a la base de datos, la clase Producto posee relaciones N a N con Color y Compras, y 1 a N con Marca y Categoría. Las relaciones N a N se representaron con listas cruzadas entre las clases; luego, tras realizar las migraciones adecuadas, el esquema de la base implementó tablas intermedias como ColorProducto, generadas automáticamente.

En el proyecto DataAccess se tienen dos paquetes: interfaces, para la creación de la interfaz del data access de usuario y producto, por si en un futuro se desea modificar la base de datos o guardar datos localmente. El otro paquete es Migrations, en el que se encuentran todas las migraciones de la base de datos realizadas, cada vez que se cambia una entidad, se agrega una nueva migración. Asimismo, en el proyecto se plantea la implementación del contexto en ECommerceContext.cs para la creación del modelo de la base de datos en Entity Framework 6, utilizando la estrategia de Code First. El acceso a la base de datos se hace en los archivos RepositorioUsuario.cs y RepositorioProducto.cs, al que solamente accede a la base de datos sin ninguna lógica de negocio. A su vez, se conectan las distintas entidades que se relacionan, como por ejemplo, Productos y Colores que mantienen una relación N a N.

Como proyecto intermedio entre el Data Access y la Api, creamos el proyecto Servicios que maneja toda la lógica de negocio, como la validación de email y de la

contraseña, para que el usuario sea válido. Servicios también juega un papel importante en la aplicación de las promociones a las compras que hace el usuario. Esto se llevó a cabo aplicando el patrón Strategy, creando una interfaz para todas las promociones, la cual se llama desde un contexto de promociones, “PromocionContext.cs”, y el cliente, en este caso, la clase “ManejadorUsuario.cs”, interactúa con él llamando a la promoción que crea necesaria.

Previo a la aplicación del proyecto de Api, cabe explicar ServicioFactory. Este servicio utiliza el patrón de diseño Factory y, tiene como objetivo, tanto la creación de una clase que inyecta dependencias al inicio de la aplicación y se mantiene a lo largo de la misma, así como la indicación de qué Contexto utilizar de la base de datos, qué servicios de usuario y de producto, y qué data access emplear.

Api es el proyecto principal, el que se ejecuta cuando se da inicio al programa. Este proyecto contiene DTOs, los cuales contribuyen a un uso adecuado de la API al momento de ingresar los datos y validarlos, pudiendo no mostrar todo lo que tiene la entidad, sino que únicamente lo que interesa. Además, contiene filtros encargados de mantener el código más limpio, evitando las expresiones como try y catch. Estos filtros captan excepciones y las manejan devolviendo estatus de error como: 400, 401, 404 y 500.

Por último, se encuentran los controladores, cada uno con su propio paquete. Optamos por dividir los controladores en cuatro, uno por cada uri principal: usuarios, producto, autenticación y compra. Por otra parte, decidimos crear un controlador solamente para el inicio de sesión, separado del de usuario, con el fin de bajar el acoplamiento del proyecto y cumplir con los principios de Single Responsibility Principle y Open/Closed Principle. Como esta clase utiliza aspectos de Json Web Token, utilizamos JWT dado que uno de los integrantes del proyecto tenía experiencia con el mismo, pero en otro lenguaje de programación. Hacer uso del JWT implicó evitar tablas extra que guarden los tokens, la información de login del usuario y permitió ahorrar espacio en la base de datos. Asimismo, compras también tiene un controlador dedicado, por más que los endpoints de esta no sean tan numerosos como los de las entidades “principales” Producto y Usuario. Esto se debe a que, a

pesar de tener las funcionalidades de “Mostrar compras del usuario” en el controlador de Usuario, justamente; la función de retornar todas las compras del sistema, reservada unicamente para los usuarios con permiso de administrador, no pertenece con las ya mencionadas.

Tanto el proyecto Pruebas como Migrations son ajenos a la ejecución del proyecto. Migrations, por su parte, se implementó ya que el proyecto Api es la encargada de procesar las request, por lo que su responsabilidad es correr las migraciones y asignarle otra responsabilidad que se adapte a la tecnología EF Core, lo que rompería con SRP. Parecería lógico realizar esta tarea en DataAccess, pero surge la preocupación de que si esta no está vinculada al framework Net6, realizar migraciones podría volverse complicado en el futuro. Cambiar el framework de netstandard2.0 a netcoreapp3.1 en el archivo DataAccess.csproj permitiría ejecutar las migraciones ubicadas en ese directorio, pero esto presenta desafíos debido a las diferencias de estructura y tecnología.

Las pruebas fueron divididas en 3 paquetes: aquellas relacionadas con los productos, con las promociones y con los usuarios. Con el fin de trabajar de manera más prolija, dentro de cada proyecto se crearon archivos de prueba dependiendo de qué clase se iba a probar, por ejemplo, uno para probar controladores y otro para probar el servicio, nuevamente, para poder mantener el SRP.

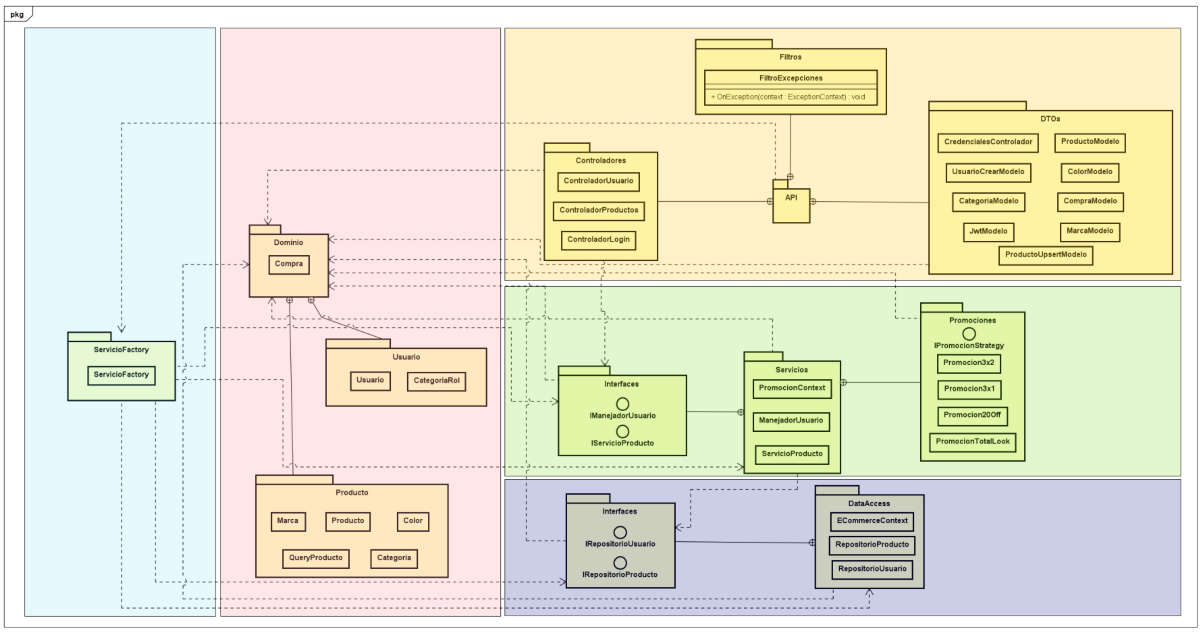
4.3 Mejoras a futuro

A pesar de ser un atributo singular al momento de la primera entrega, el “Rol” de usuario se podría convertir en una lista de Roles, para así evitar soluciones ad hoc como “ClienteAdministrador”.

También sería una mejora de diseño establecer una clase para JWT en particular.

5. Vistas de Diseño e Implementación

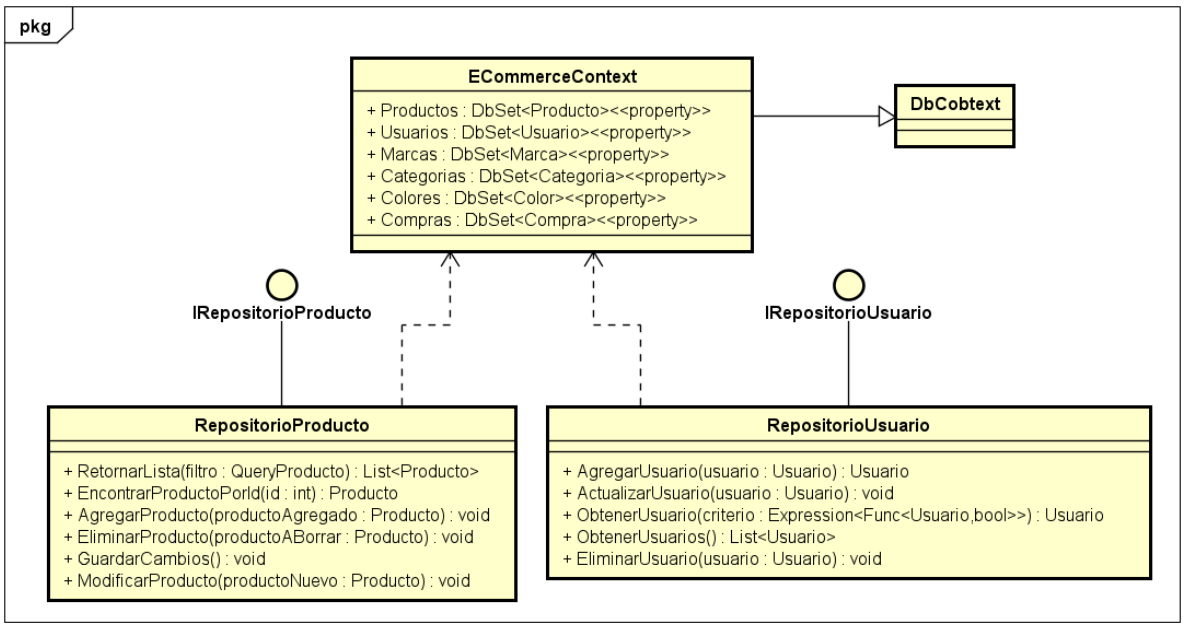
5.1. Diagrama de paquetes



5.2. Descripción de cada paquete

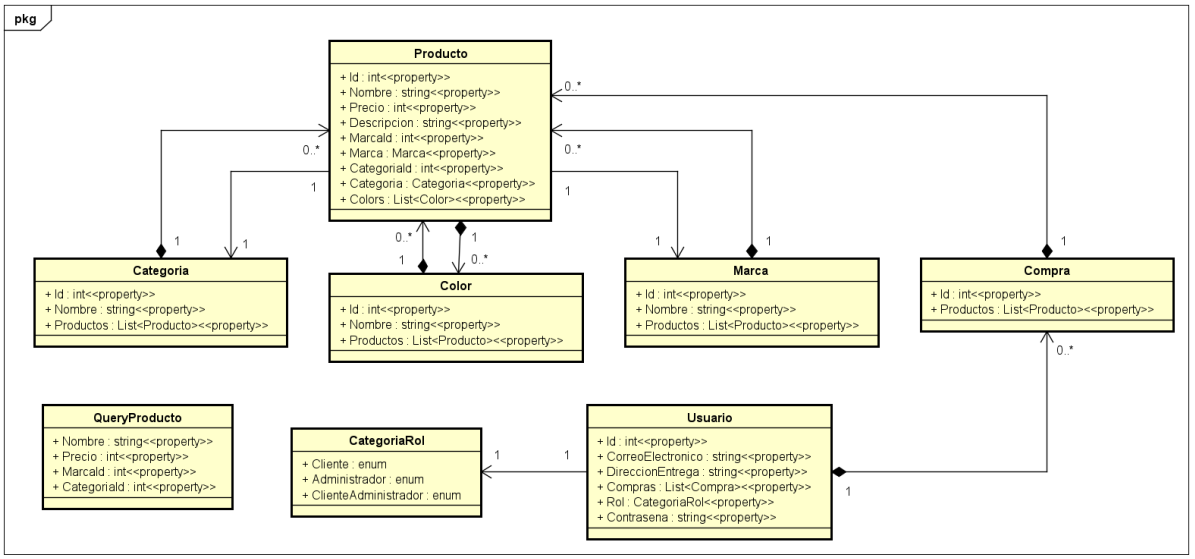
5.2.1. DataAccess

En este paquete se define “ECommerceContext”, la cual hereda de DbContext, y sirve como intermediario para la interacción de la lógica del negocio con la base de datos. Además, está la carpeta “Migrations”, la cual almacena todas las versiones de cambios de la base de datos, y diversas clases de repositorios, encargadas de llevar a cabo la creación, lectura, actualización y eliminación de entidades particulares.



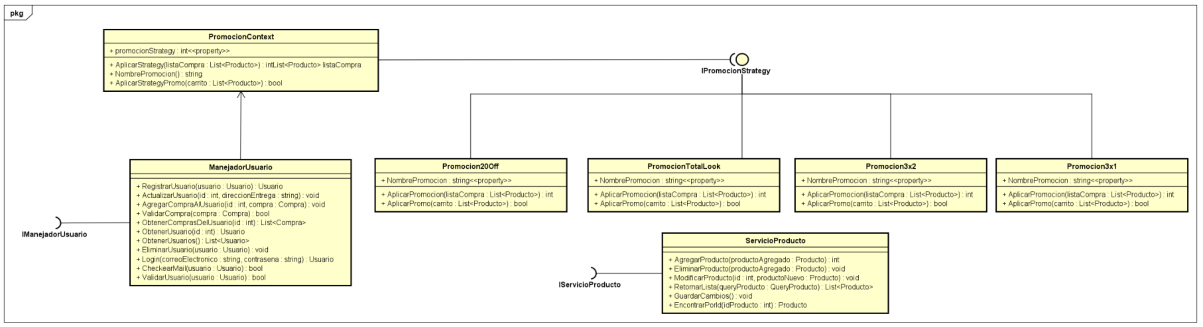
5.2.2. Dominio

En este se definen las clases que representan las entidades del dominio del proyecto, se dividen en dos paquetes, estos siendo Usuario y Producto donde cada paquete se encarga de complementar al Usuario y Producto.



5.2.3. Servicios

El proyecto servicios se encarga de ser la capa intermedia entre la BusinessLogic y la API, además, contiene el patrón strategy de promociones para aplicarlas en la lógica de negocio de la compra en el usuario.



5.2.4. ServicioFactory

El Servicio Factory tiene un solo programa que inyecta dependencias a lo largo de toda la solución.

5.2.5. Migrations

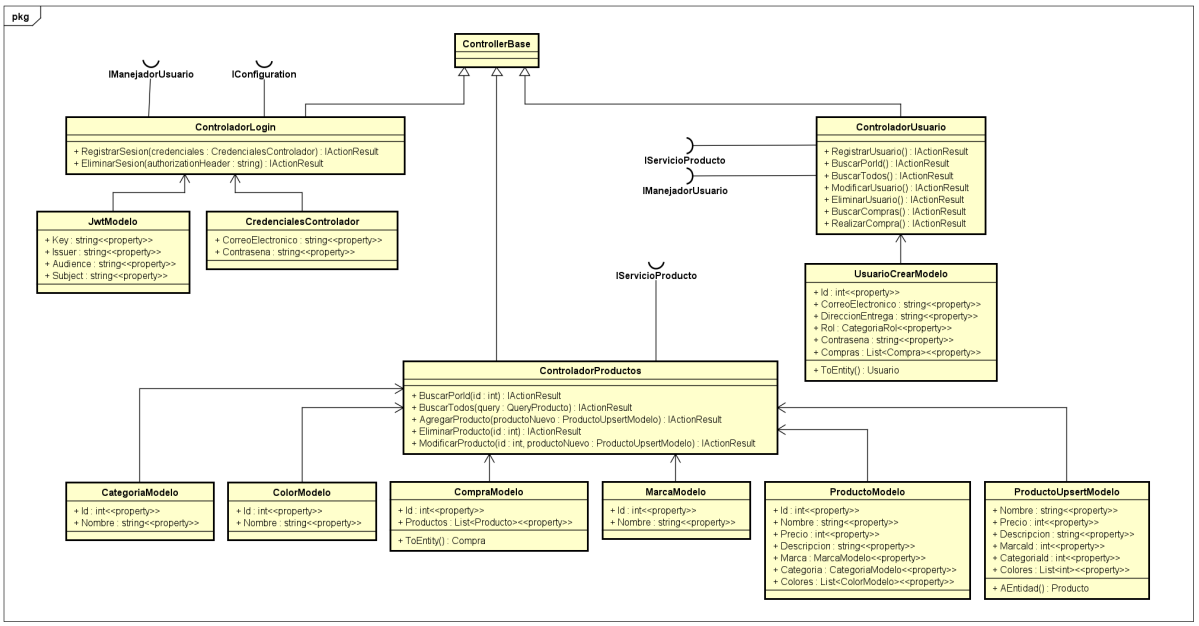
Creado para cumplir el principio de Single Responsibility Principle y crear las migraciones mediante líneas de comando.

5.2.6. Pruebas

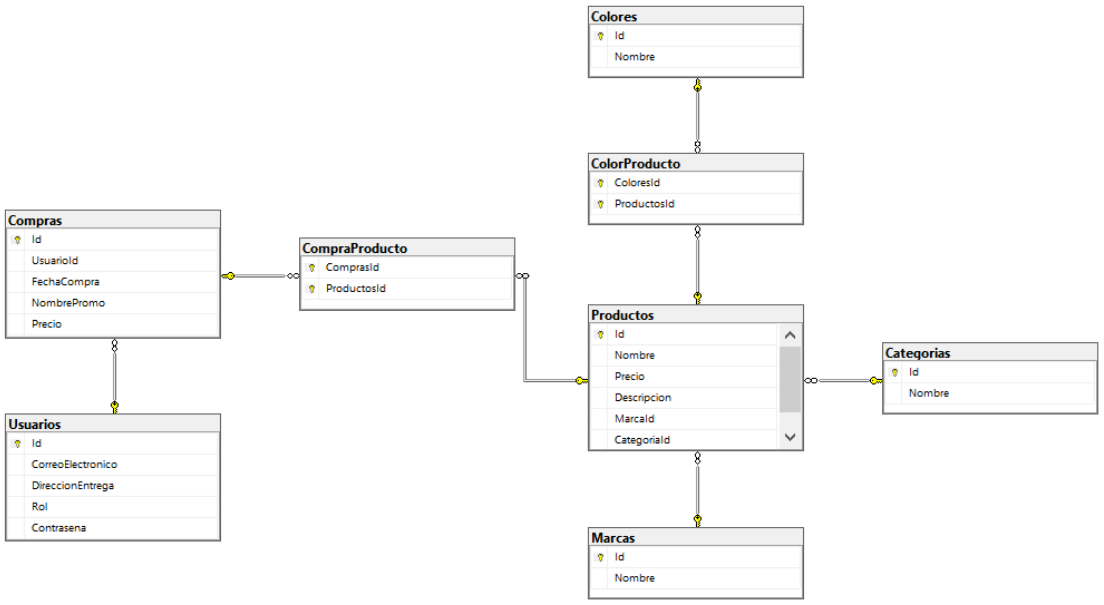
Es el paquete que contiene todas las pruebas unitarias, realizadas previamente a la implementación de las funcionalidades. Se utiliza mock con el fin de no acceder de forma directa a la base de datos.

5.2.7. Api

Es el paquete con los controladores que exponen los endpoints de la API mediante la cual el usuario puede interactuar con el sistema y modificar o consultar el estado de este. Además, se comunica con los servicios a través de interfaces para responder a las peticiones que se realizan.



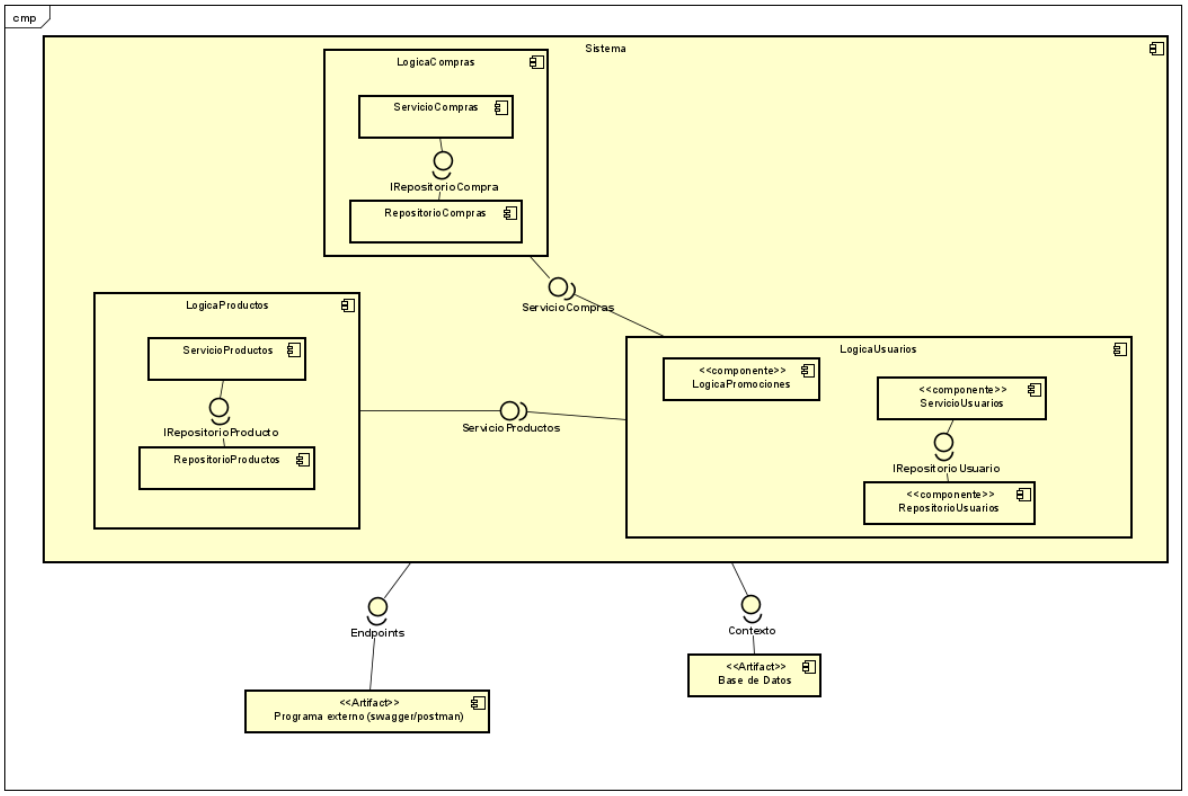
5.3. Modelo de tablas



Las tablas ColorProducto y ComprarProducto son tablas autogeneradas por la base para representar la relación N a N entre las clases de ambos pares. En ambas instancias, las clases tiene una lista de la otra para significar este vínculo; Producto tiene una propiedad Compras de tipo List<Compra> y Compra tiene una de producto, inversamente.

No se incluye en el diagrama la tabla _EFMigrationsHistory.

5.4. Diagrama de componentes



5.4 Dependencias

Como ya fue mencionado en secciones anteriores, se crearon interfaces de las cuales las clases de lógica dependen. Tanto los repositorios como los servicios; o manejadores, dependiendo de cada clase; implementan una interfaz con sus métodos públicos.

Esto se creó para fomentar y facilitar la aplicación del principio DIP de SOLID; ya que al inyectar dependencias de la capa DataAccess en la capa de servicios; y subsecuentemente al realizar esto con Servicios en la API, se establecen vínculos de dependencia con las interfaces directamente. La gran ventaja presentada por esta práctica consiste principalmente en que, en caso de requerirse la implementación de distintas variantes del mismo concepto (repositorio para productos, por ejemplo), no habría necesidad de modificar el codigo existente para permitir la integración del nuevo, mas allá de las funcionalidades y la conexión de los métodos exclusivos a

través de las capas; si es que estos fueron agregados. Solamente con crear la clase que implemente la interfaz ya podemos utilizarla en todo lugar donde la otra se encuentra.

5.5 Manejo de excepciones

Se creó un filtro de excepciones, mediante el cual se minimizó la cantidad de excepciones no controladas del sistema. Hasta el momento de la entrega, las excepciones manejadas consisten de:

> KeyNotFoundException = 404

> ArgumentException = 400

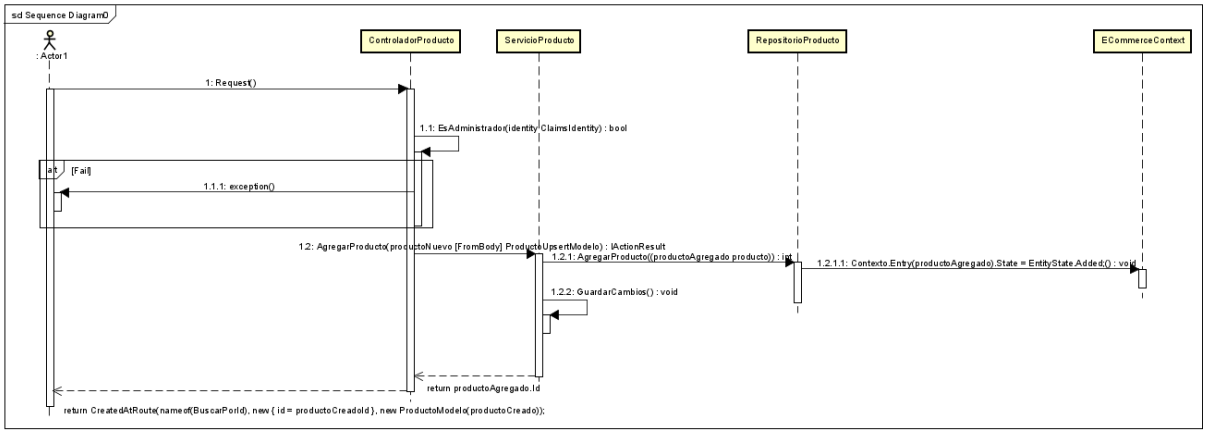
> UnauthorizedAccessException = 401

> Exception (generica) = 500

Se incluyeron en el filtro todos los tipos de excepciones encontradas durante la fase de pruebas del sistema; sin embargo, en caso de que se encuentre algún error inesperado con cual el equipo de desarrollo no dio, se implementó la cuarta opción. Esta excepción genérica se espera que contenga todo caso borde que evada los primeros tres criterios del filtro, proveyendo aun así un código de error no específico.

5.5 Diagramas de Secuencia

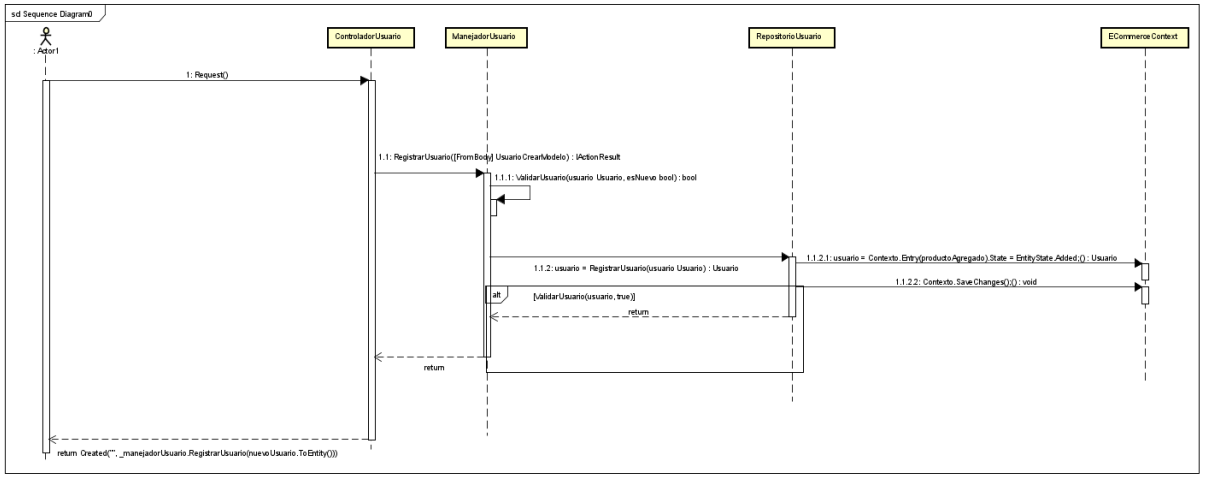
Diagrama de secuencia de la funcionalidad crear producto



Este diagrama representa el proceso de crear un Producto nuevo. Es representativo del proceso de creación de Usuario, a su vez, ya que este es casi análogo.

Se omitieron detalles como la autenticación del lado del Postman, ya que se consideró que no entraban dentro del alcance del diseño y su código, sino la comunicación de herramientas externas con el mismo.

Diagrama de secuencia de la funcionalidad registro de usuarios



>Los diagramas grandes se incluyen como imágenes PNG en la carpeta para que se pueda inspeccionar con zoom los mismos