Universidad ORT Uruguay

Facultad de Ingeniería

Diseño de Aplicaciones 2

Obligatorio 1

Entregado como requisito para el Obligatorio 1 de Diseño de Aplicaciones 2

Diego Acuña - 222675

Felipe Briosso - 269851

Nicole Uhalde – 270303

https://github.com/IngSoft-DA2-2023-2/222675-269851-270303

Tutores:

Francisco Bouza

Juan Irabedra

Santiago Tonarelli

2023

Contents

Estructura de este documento:

El siguiente documento se encuentra organizado de la siguiente manera. En la primera sección, se explicitan los bugs detectados. Luego, el documento se organizará en secciones, cada una haciendo énfasis en una vista del modelo 4+1, y en la última sección se incluirán algunas aclaraciones generales.

Descripcion general Bugs y reports:

Defecto 1:

Dado como EF creó nuestras tablas a partir de code first, la entidad Product contiene el id de su compra. A partir de esto nos surgió el problema de que al traer los productos, este nos traía los productos asociados a una compra y al momento de filtrar por sus atributos el único que los diferencia es el id de la compra (que en el caso del producto original es null).

Sin embargo, la id de la compra no puede ser accedido a través del repositorio del producto, debido a que es un atributo de la compra.

Nuestra solución rápida fue traer el primer producto, aunque este puede pertenecer a una compra.

Vista de implementación

Diagrama de paquetes y arquitectura de la solución:

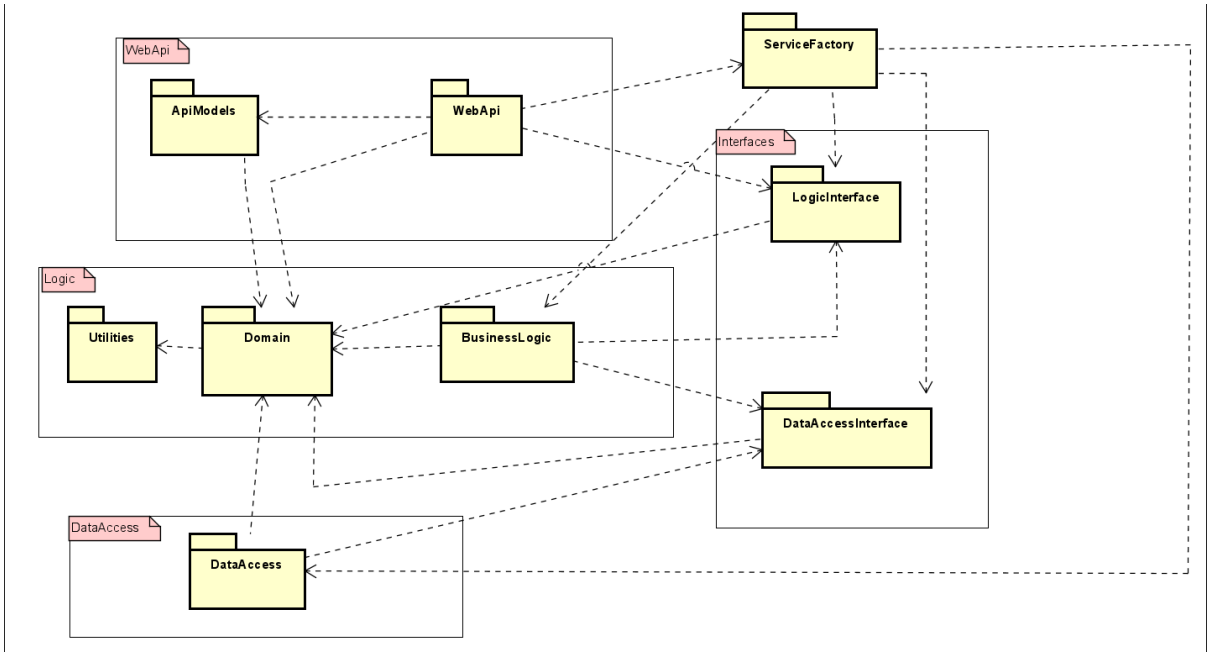

Figura 1: Diagrama de paquetes que muestra las distintas capas del sistema.

El equipo decidió separar la solución en cuatro capas bien diferenciadas según su propósito.

La primera de ellas es la WebApi. Dentro de esta capa tenemos el paquete de WebApi, que incluye los controladores y sus filtros. A su vez, también tenemos los Api Models, encargados de la comunicación con el cliente.

En segundo lugar, se encuentra la lógica. El dominio, businessLogic y utilities son los paquetes que componen esta capa. Son clases de alto nivel y encargadas de modelar las entidades.

En tercer lugar, está la capa de DataAccess. Es la capa encargada de la persistencia de datos del sistema.

En cuarto lugar, los paquetes LogicInterface y DataAccessInterface contienen las interfaces de sus respectivos paquetes. Esto fue realizado con el fin de ocultar la implementación y por ende minimizar el impacto al cambio.

Un aspecto que puede llamar la atención de la figura anterior es el ServiceFactory. Este es un paquete que se encuentra por afuera de los anteriores y se encarga de inyectar dependencias, aunque será explicado con mayor profundidad más adelante.

Con respecto a cómo están conformados los paquetes de la figura 1, a continuación, se presenta un diagrama de ello:

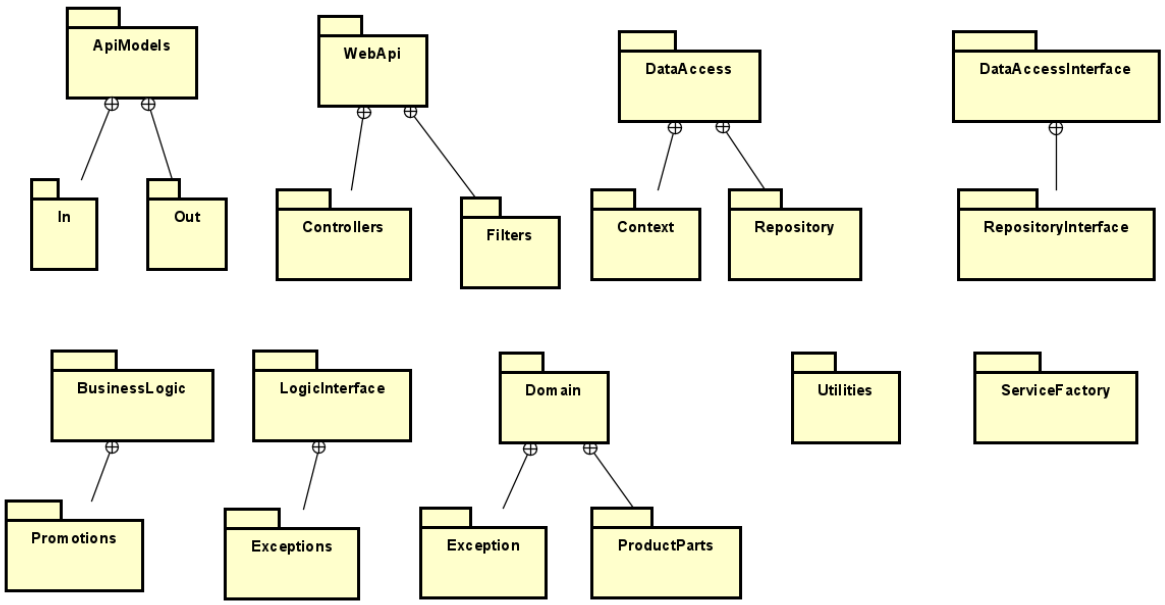

Figura 2: Diagrama de paquetes utilizando nesting.

Con el fin de que este diagrama sea claro y entendible, se optó por eliminar las dependencias entre paquetes. En la siguiente sección se analizará cada uno de ellos.

Diagrama de componentes:

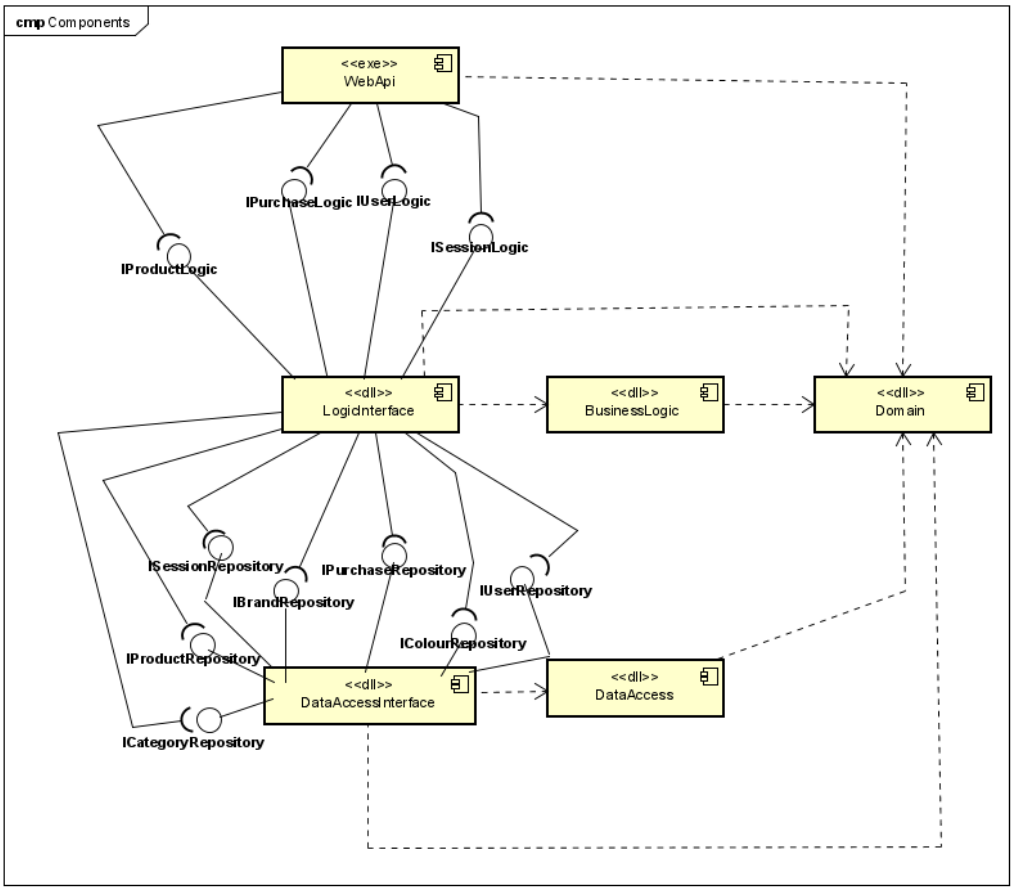

Figura 3: Diagrama de componentes

En el diagrama anterior, se observa cómo están relacionados los componentes de la aplicación. En primer lugar, destacamos el uso de interfaces entre DataAccessInterface, LogicInterface y WebApi, esto realizado con el fin de mejorar la separación entre componentes, que será explicado con mayor profundidad más adelante.

Vista lógica

Diagramas de clase por paquetes:

Domain

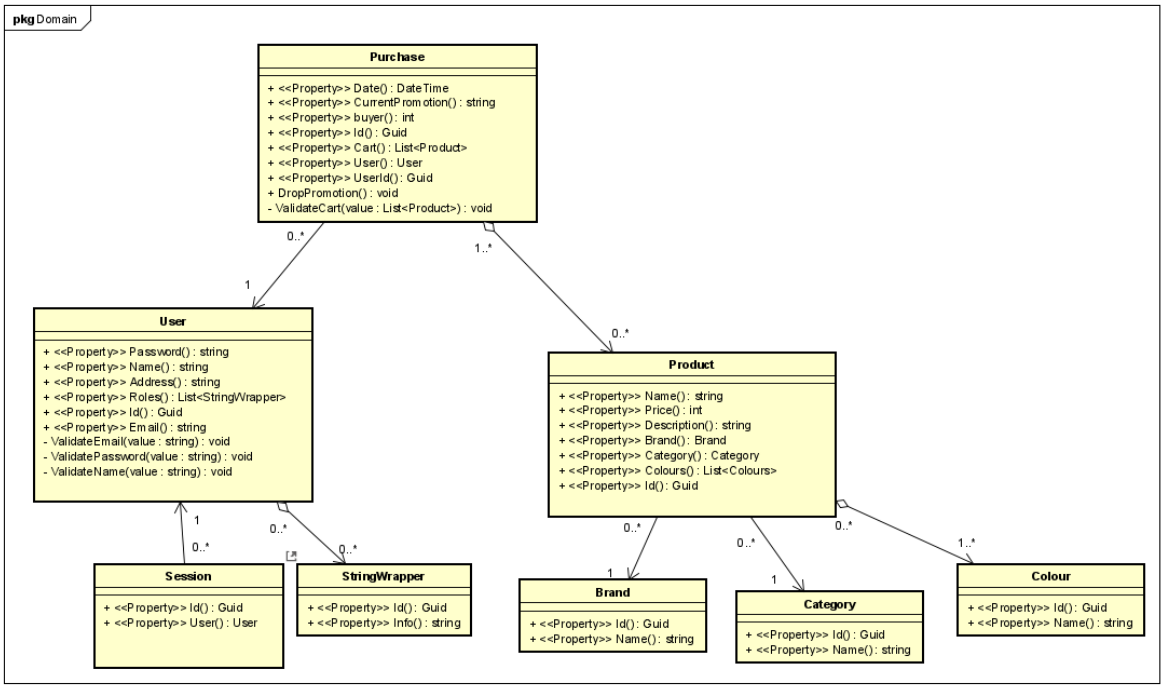

Figura 4: Diagrama de clases de Domain

En el diagrama anterior se observa loas clases del dominio. Con respecto a las clases creadas, la mayoría de estas surgen de la letra del obligatorio.

Con respecto a las decisiones de diseño tomadas en este paquete, se tomaron las siguientes.

- Si bien el email del usuario es único, este puede cambiar en el futuro. Por ende, para evitamos tener claves que potencialmente pueden cambiar, se optó por utilizar un identificador (guid).
- De igual forma, se incluyó un id en las demás entidades, con el fin de facilitar el manejo de la base de datos.
- El equipo decidió crear una clase Session, encargada del manejo de login y logout. Si bien a primer momento se consideró que el token de acceso para los usuarios sea su propio identificador, esto no es lo más correcto. Por lo tanto, se optó por crear una entidad nueva, que contenga un id de sesión y el usuario que la inicia. De esta forma, podemos lograr que un usuario se conecte en más de un dispositivo y que sus login/logout no afecte las sesiones de sus otros dispositivos.
- Otro aspecto que puede llamar la atención es la clase StringWrapper. Originalmente, el usuario poseía una lista de roles (guardados como strings). Debido a que Entity Framework no lo reconocía, se optó por crear una clase nueva, que representa los roles. Se la llamó de esta forma debido a su propósito y para promover el reuso de código.
- Por último, no se incluyó en el diagrama relaciones con clases de otros paquetes, pero varios de estos utilizan Utilities para realizar las validaciones.

Utilities

Estamos al tanto que el uso de Utilities era una práctica muy común en la industria, aunque últimamente ha caído en desuso. En nuestro caso contiene una única clase que tiene como objetivo ayudar al dominio a realizar validaciones, por ejemplo, para los formatos de mail o contraseña. De

esta forma logramos mejorar SRP y promover el reuso de código. Además, nos parece más prolijo que tener estos métodos directamente en las clases que los necesiten.

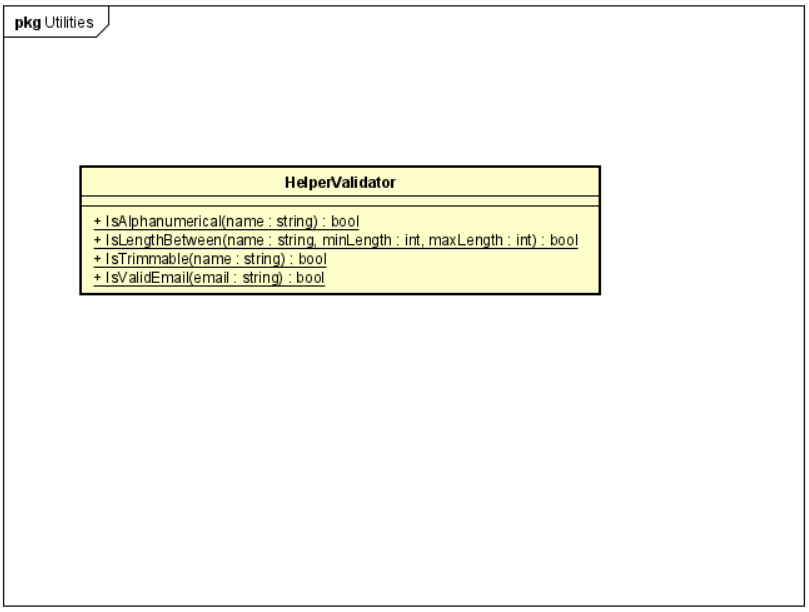

Figura 5: Diagrama de Utilities

Los métodos de esta clase son todos estáticos ya que la clase no tiene atributos propios. Sus métodos son utilizados por clases externas y su comportamiento es el de auxiliar a otras clases a realizar validaciones.

BusinessLogic

En este paquete se encuentra toda la lógica del sistema, incluyendo los descuentos. Originalmente habíamos pensado en dejarlos en el dominio. Pero debido a que estos no se modelan a base de datos nos pareció mas correcto dejarlos en este paquete.

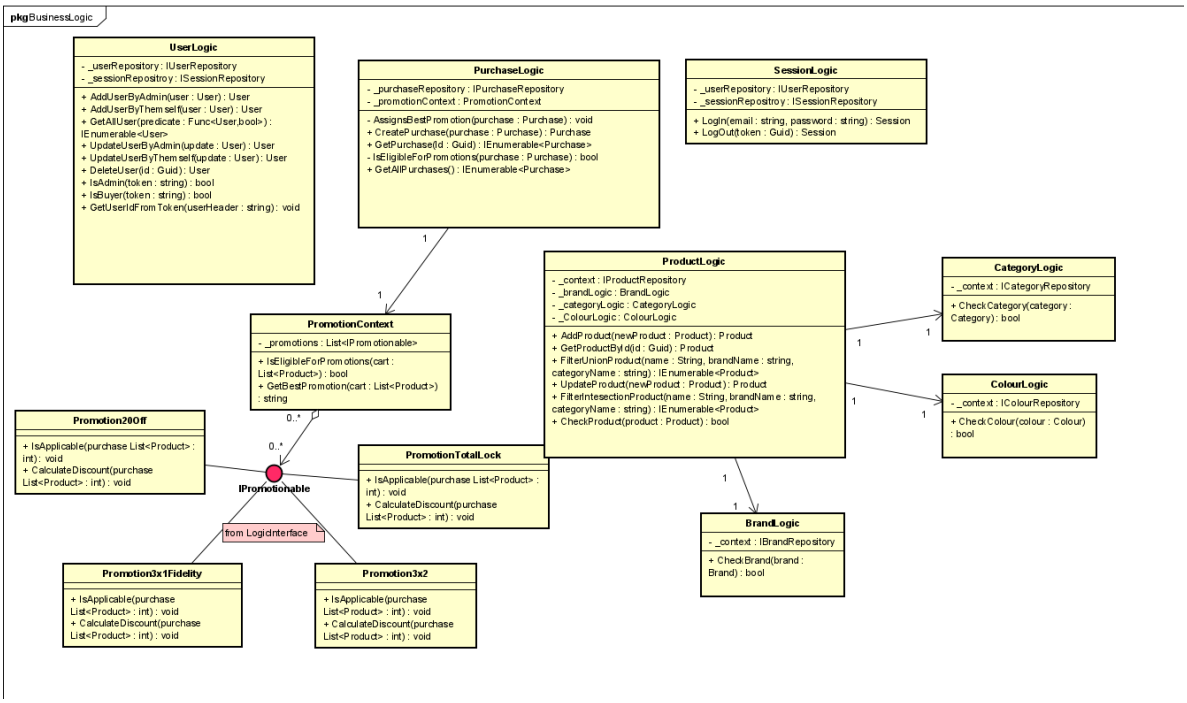

Figura 6: Diagrama de BusinessLogic

La interfaz que se encuentra en rojo, se encuentra diferenciada de las demás entidades debido a que no pertenece a este paquete. No obstante, se reconoció que las promociones tenían los mismos métodos, por lo que se decidió que implementaran una interfaz.

LogicInterface

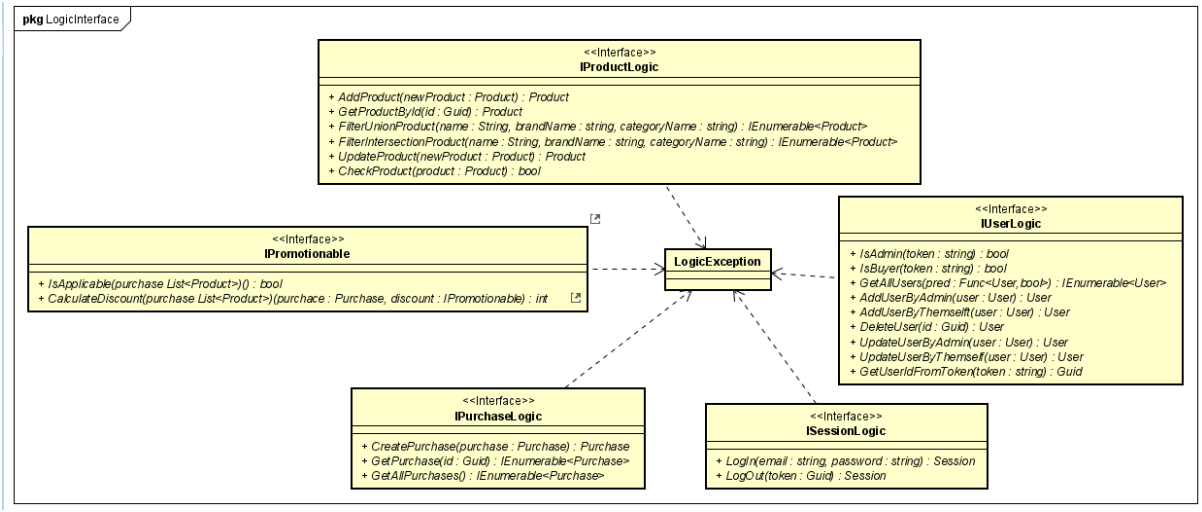

Figura 7: LogicInterface

Este paquete contiene las interfaces que son implementadas por BusinessLogic. Además, tiene la clase que crea las excepciones a nivel de lógica. Se utilizaron interfaces con el fin de minimizar el impacto a cambio y de favorecer DIP.

Se optó por crear una excepción por capa de la solución, esto con el fin de poder saber de dónde viene el error y poder manejarlo de mejor manera.

DataAccess

Contiene el contexto y la implementación de los repositorios. En este paquete no se tomaron mayores decisiones de diseño.

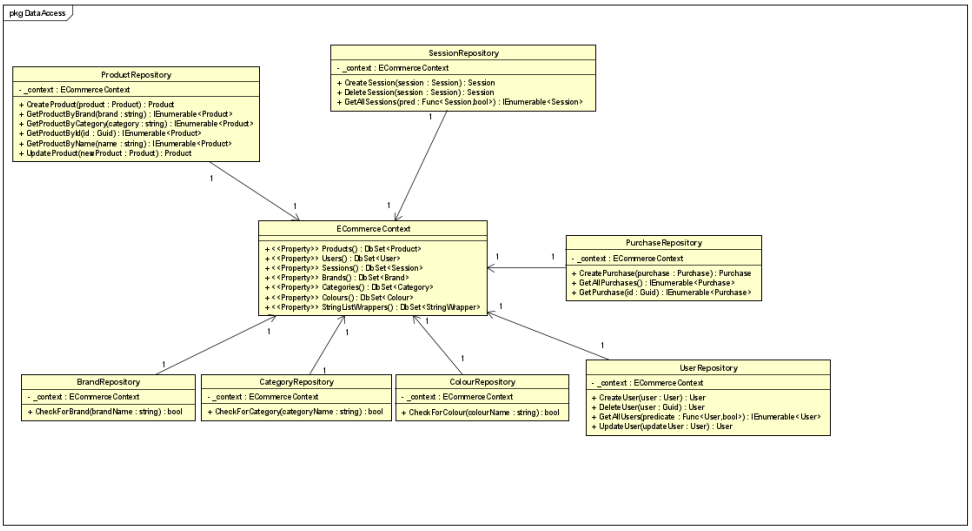

Figura 8: DataAccess

DataAccessInterface

Contiene las interfaces de DataAccess y una clase encargada de crear excepciones relacionadas con la base de datos. Fue diseñado con el objetivo de promover el Principio de Inversión de Dependencias (DIP).

Se utilizaron interfaces por diversos motivos. En primer lugar, nos permitió generar un nivel de abstracción al definir contratos. En segundo lugar, porque promueve el polimorfismo. Por último, utilizando interfaces, podemos mockear su comportamiento, lo cual nos permite implementar tests de capas superiores sin tener que implementar las capas de niveles inferiores.

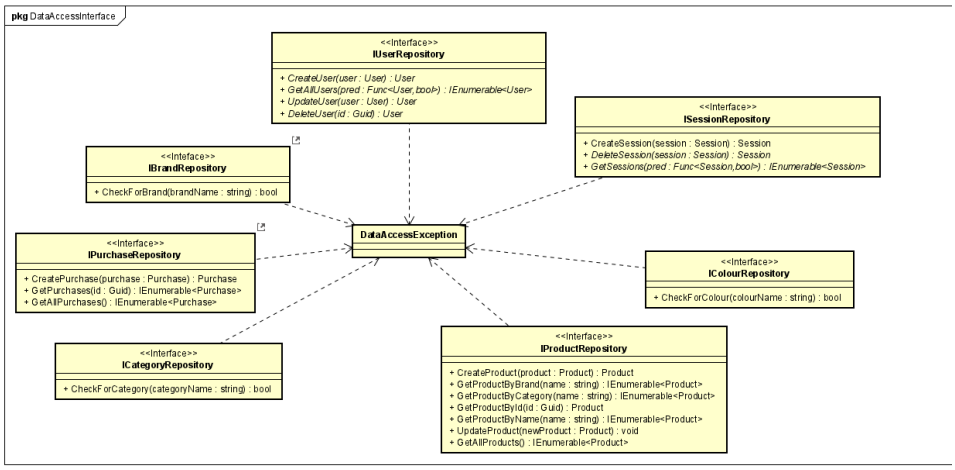

Figura 9: DataAccessInterface

ApiModels

Contiene todos los modelos in y out de la aplicación. Se encarga de recibir las requests del usuario y devolver las respuestas. Se realizó para cumplir con SRP y de poder elegir qué datos recibir y mostrar al usuario.

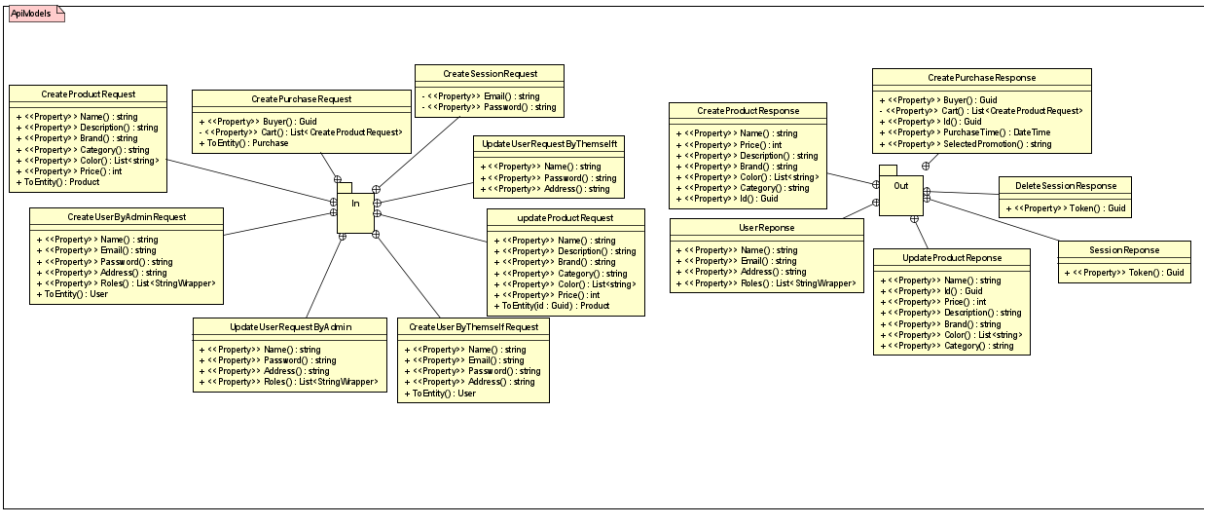

Figura 10: Clases dentro de ApiModels

En el diagrama anterior se observa que la mayoría de clases contienen únicamente properties. Por lo que varias de ellas fueron implementadas como structs.

A su vez, en algunas requests se observa el método ToEntity(), esto con el fin de convertir las requests en clases del dominio. De esta forma, la lógica recibe un objeto del dominio, generando dependencias entre la lógica y las clases del dominio, cumpliendo DIP.

En caso contrario, si optábamos que la lógica reciba una clase de request, generábamos una dependencia desde la lógica hacia los Api Models; violando DIP.

<u>WebApi</u>

Posee los controladores y filtros. Se lo separó con el fin de mejorar la organización, fomentar la modularidad y escalabilidad.

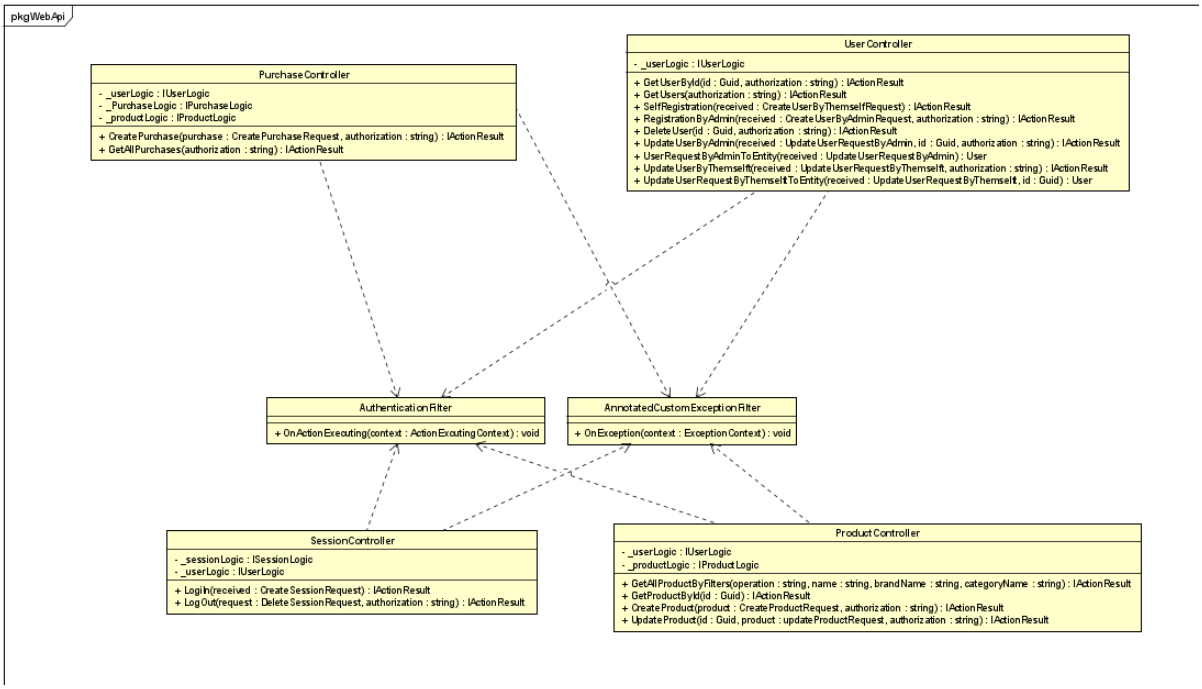

Figura 11: Clases de WebApi

Con respecto a los filtros, estos nos permitieron remover los try y catch de los controladores, mejorando la legibilidad.

Con respecto a AnnotatedCustomExceptionFilter, esta pregunta por el tipo de la excepción y dependiendo del tipo de esta retorna una response distinta. Sabemos que preguntar por rtti no es buena práctica, pero dada la baja cantidad de excepciones que manejamos no nos pareció que valía la pena solucionarlo con polimorfismo.

ServiceFactory

Tiene únicamente la clase que se encarga de las inyecciones de dependencia de las interfaces y de marcar el connection string de la base de datos.

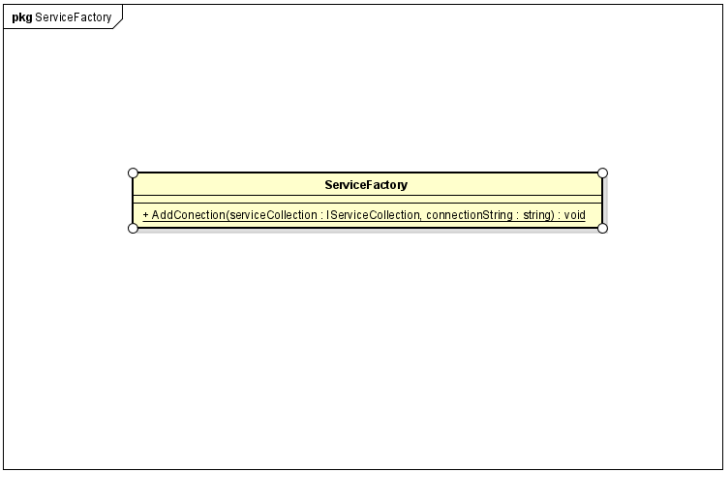

Figura 12: ServiceFactory

Modelo de tablas Base de datos

El modelado de la base de datos fue creado por Entity Framework, por lo que a continuación se presenta un diagrama de las tablas resultante:

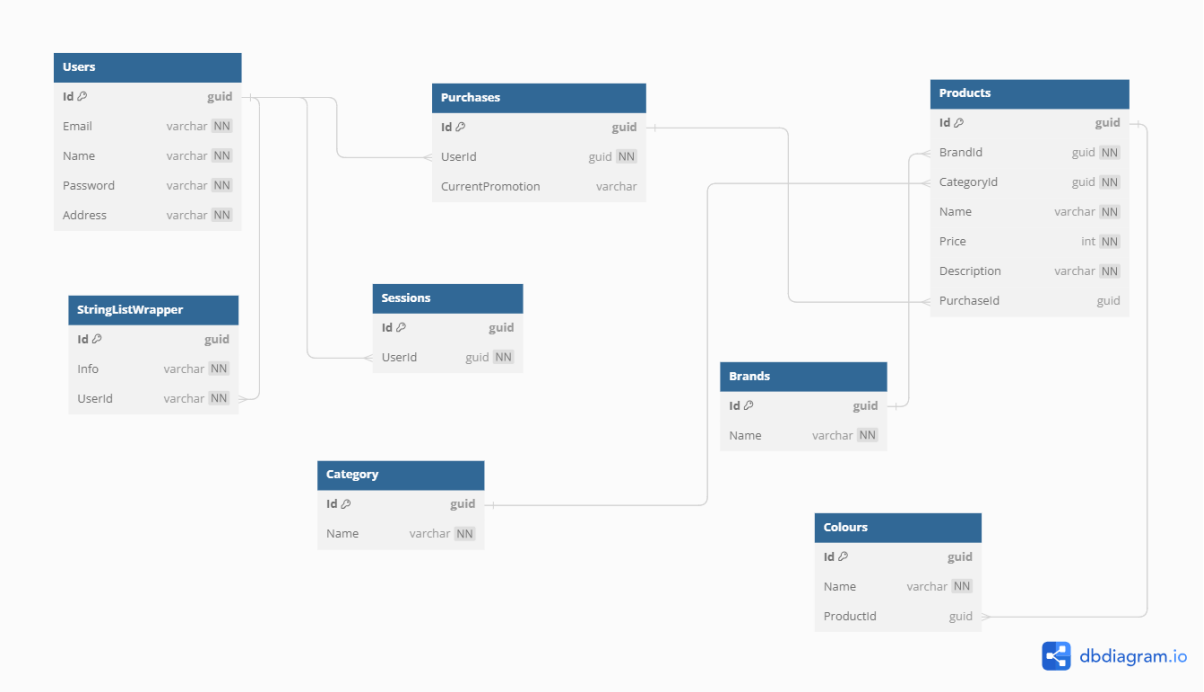

Figura 13: Modelado base de datos

El modelo de la base de datos esta hecho en base a code first modelando únicamente las clases del dominio.

En el diagrama se observan las tablas, sus atributos y tipos respectivos. A su vez, los atributos que tienen a la derecha “NN” significa que no son nulleables. Por último, en el diagrama no aparece que atributos tienen que ser únicos ni la cardinalidad de las relaciones, por lo que sugerimos entrar al siguiente enlace donde además de observar el diagrama, se indican dichos detalles:

https://dbdiagram.io/d/651c90ceffbf5169f0f8df4f

Con respecto al modelado, se observa que, en el pasaje a tabla de EF, hay ciertas relaciones que nos llaman la atención. Por ejemplo, que un Colour conozca a su producto o que un Product tenga su Purchase.

Patrones de diseño utilizados e inyección de dependencias

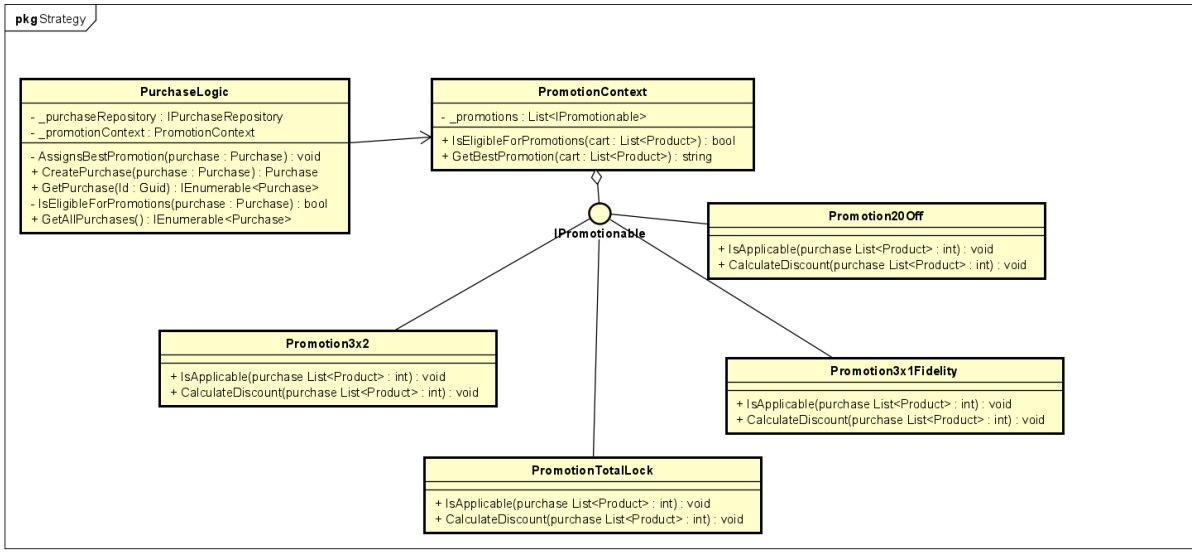

Figura 14: Modelado de las promociones

Patrón Strategy

En el proceso de aplicar descuentos en las compras, nos enfrentamos a un desafío crucial. Teníamos la familia de algoritmos de calcular descuentos, pero no sabíamos como hacerlas intercambiables para poder probarlas todas en tiempo de ejecución.

Fue en ese punto donde consideramos que el patrón de diseño Strategy era la mejor solución, ya que precisamente intenta resolver el problema que teníamos.

Además, dos factores adicionales influyeron en nuestra decisión. Primero, las promociones no eran configuradas por los compradores o administradores, y tampoco se almacenaban en una base de datos. Esto nos llevó a la conclusión de que podíamos codificar estas promociones directamente en la lógica, lo que facilitaría la adición de nuevas promociones en el futuro, simplemente agregándolas y permitiendo que el contexto tuviera acceso a ellas.

En resumen, el patrón Strategy se presentó como la solución idónea para nuestro problema de aplicar descuentos en las compras, permitiéndonos flexibilidad, fácil mantenimiento y la capacidad de agregar nuevas promociones de manera sencilla.

Inyección de dependencias

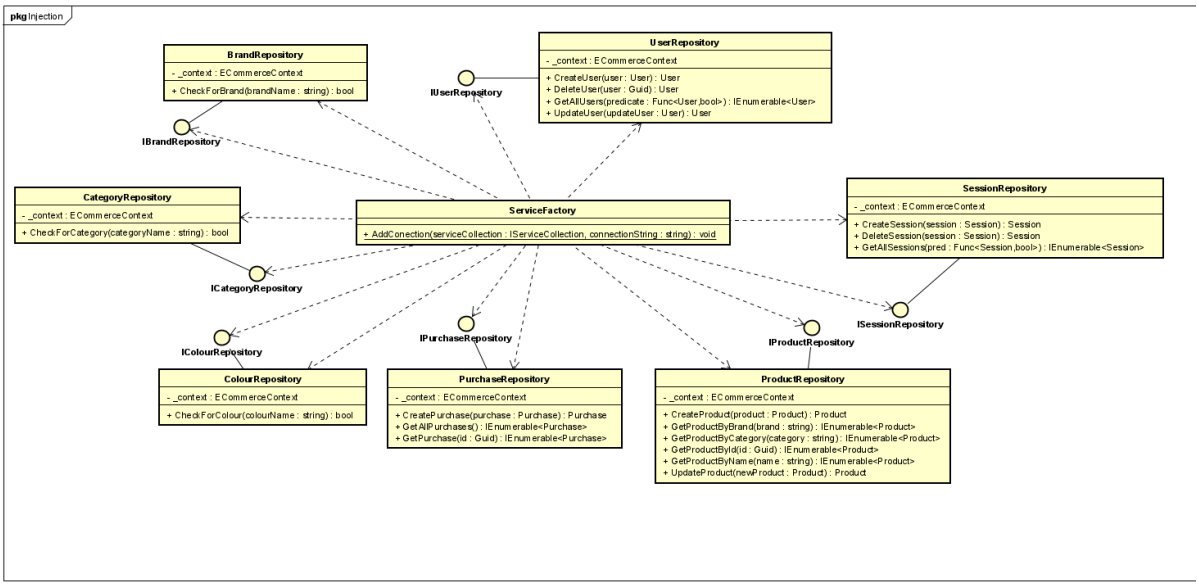

Figura 15: Inyección de dependencias

Se decidió utilizar inyección de dependencia en varias partes del obligatorio. Por un lado, entre el data Access y lógica con sus respectivas interfaces. Esto nos permitió mejorar el desacoplamiento, la reusabilidad, mantenibilidad, flexibilidad y el cumplimiento de DIP (Dependency Inversion Principle) al hacer que clases de alto nivel dependan de interfaces y no de clases concretas.

Vista proceso

Diagramas de interacción

Feature 1: Crear un producto

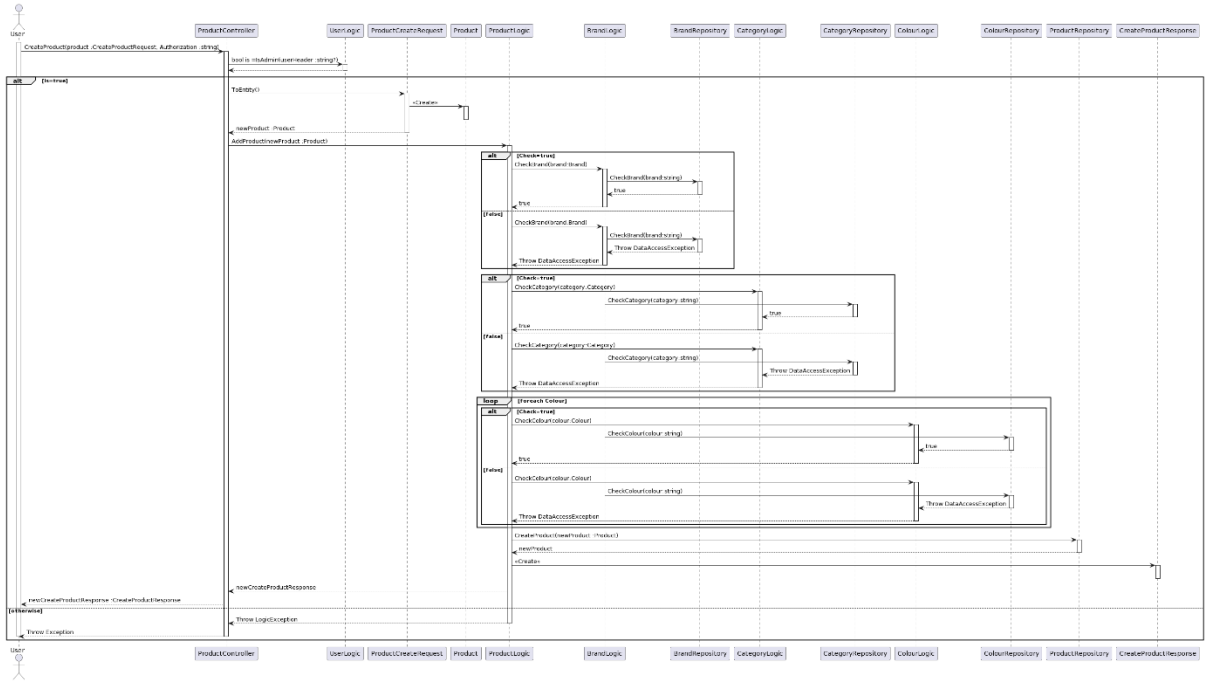

Figura 16: Diagrama de secuencia de product creation

Este primer diagrama de secuencia muestra el recorrido cuando un usuario quiere crear un producto nuevo. Decidimos tomar este diagrama dado que demuestra, como se pasa por las distintas capas, nos muestra el manejo de excepciones y por último nos muestra que cada clase tiene su responsabilidad bien marcada.

A partir del diagrama, se observa que los controladores se comunican únicamente con la lógica y con las requests, y la lógica se comunica con el repositorio. De esta forma, cumplimos la ley de Demeter, evitamos generar dependencias innecesarias y permitimos que cada capa pueda cambiar o evolucionar de manera independiente, sin afectar a las demás.

Feature 2: Log in

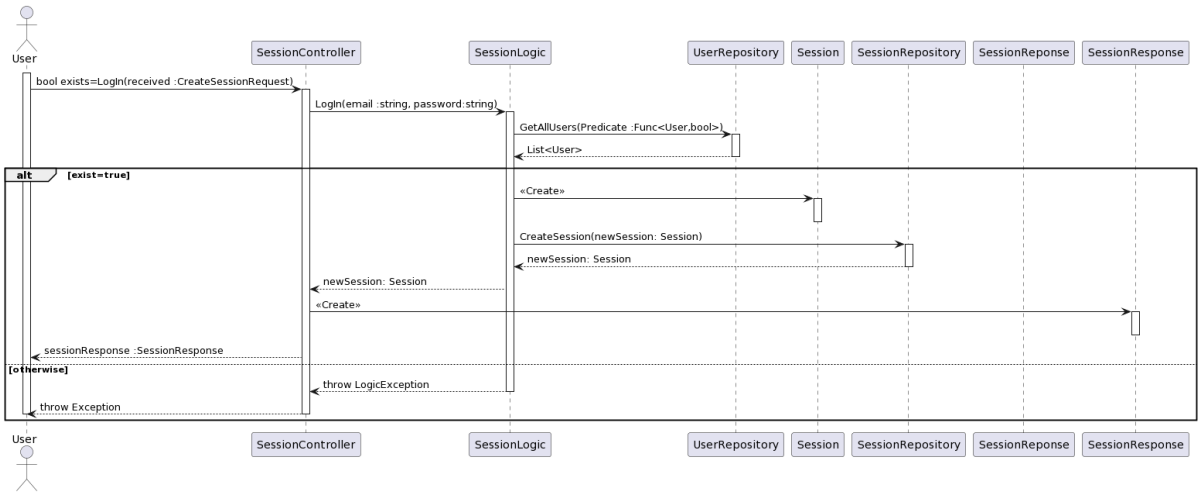

Figura 17: Diagrama de secuencia del login

Este segundo diagrama es más sencillo que el anterior y nos muestra el recorrido cuando un usuario quiere logearse(crear una nueva sesión). Nos muestra que el usuario envía sus credenciales, estas son procesadas en la lógica(validando las credenciales en el data access), y en caso de ser correcto se crea un nuevo token de sesión y se retorna un sessionResponse. En caso de ingresar credenciales inválidas, se tira una excepción.

Vista física

Diagrama de despliegue

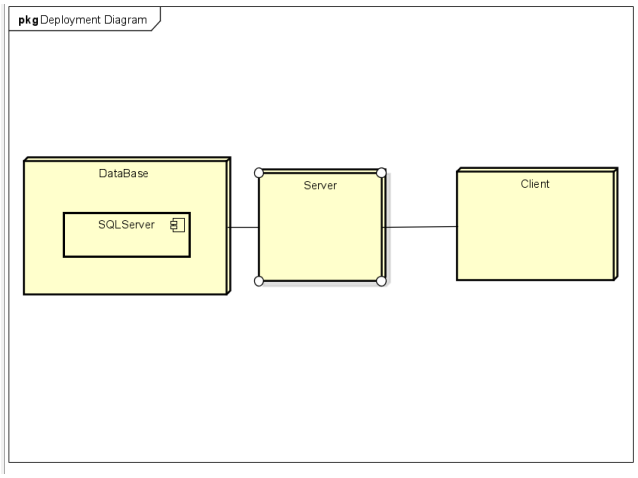

Figura 18: Diagrama de despliegue

Del diagrama se concluye que la arquitectura es cliente-servidor. El servidor se encuentra conectado a una base de datos que utiliza SQL Server.

Algunas aclaraciones

Gitflow y metodología de trabajo

El equipo decidió seguir esencialmente los fundamentos de gitflow, pero haciendo ciertas modificaciones. En primer lugar, se crearon ramas con nombres como “quickfix” o “refactor/” o “Fix…”. Esto con el fin de resolver errores, o realizar mejoras en el código. Esto se hizo con el fin de poder arreglar el código con mayor rapidez.

No obstante, a pesar de que se desvío levemente de la metodología de gitflow, se siguieron varios aspectos como por ejemplo llamar a las ramas según la feature, que las ramas salgan y vuelvan a develop y que únicamente el commit final sea en main.

Con respecto, a la metodología de trabajo, esta varió con el transcurso del obligatorio. Inicialmente, dado que no teníamos los conocimientos de implementar la api, se procedió a implementar las clases del dominio.

Luego de haber visto dichos temas en el práctico, procedimos a seguir un enfoque top-down. No obstante, en algunos endpoints se implementó de bottom-up. Esto se realizó cuando dos integrantes estaban trabajando sobre la misma slice. Con el fin de evitar que dos personas estén modificando el mismo archivo al mismo tiempo, se optó que uno modifique el controlador, utilizando top-down, mientras que el otro desde el repositorio usando bottop-up, encontrándose en la capa de lógica y mergeando sus cambios sin generar conflictos.

Pasos para probar la solución

En la carpeta release abrir el archivo appsettings.json y modificar el connection string según lo que se indique en sql server.