Universidad ORT Uruguay

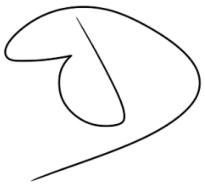
Facultad de Ingeniería

OBLIGATORIO 1

DISEÑO DE APLICACIONES 2

-

Descripción del diseño



Nicolás Edelman
251363

Juan Diego Etcheverry
252443

Mateo Goldwasser
239420

Tutores: Francisco Bouza, Juan Irabedra, Santiago Tonarelli

2023

Indice:

Link al repositorio:
https://github.com/IngSoft-DA2-2023-2/239420-252443-251363

Descripción general del trabajo:

Nuestro trabajo consistió en la creación de un sistema capaz de ayudar a las empresas de la industria textil mediante el comercio electrónico, en cuanto a implementar promociones sumado a lo que ya hace una tienda física. Esto lo logramos a través del desarrollo de un e-commerce, en el cuál se le permite a los usuarios agregar productos al carrito y en base a los productos agregados y las promociones del sistema calcular el precio total para posteriormente permitir que se realice la compra.

Para ello, creamos una WEB API a la cuál el cliente podrá realizar consultas a través del protocolo HTTP. También hacemos uso de una base de datos, lo que nos ayudará a persistir los datos de los usuarios, compras, productos y diferentes actores del sistema. En cuanto a los actores del sistema, contamos con tres tipos de usuarios: Compradores, Administradores o Ambas. Cada uno tendrá distintos permisos con los que podrá ir efectuando las requests, dependiendo del pedido y la autorización.
Entre las acciones que tiene permitido hacer el **comprador** se encuentran:

- SignUp
- LogIn
- LogOut
- Añadir un producto a su carrito
- Pedir el historial de compras del usuario
- Ver el listado de todos los productos y poder filtrarlos a partir de un texto y/o
- una marca y/o una categoría
- Ver el detalle de un producto
- Efectuar una compra

Por otro lado, el usuario con rol **Administrador** puede:

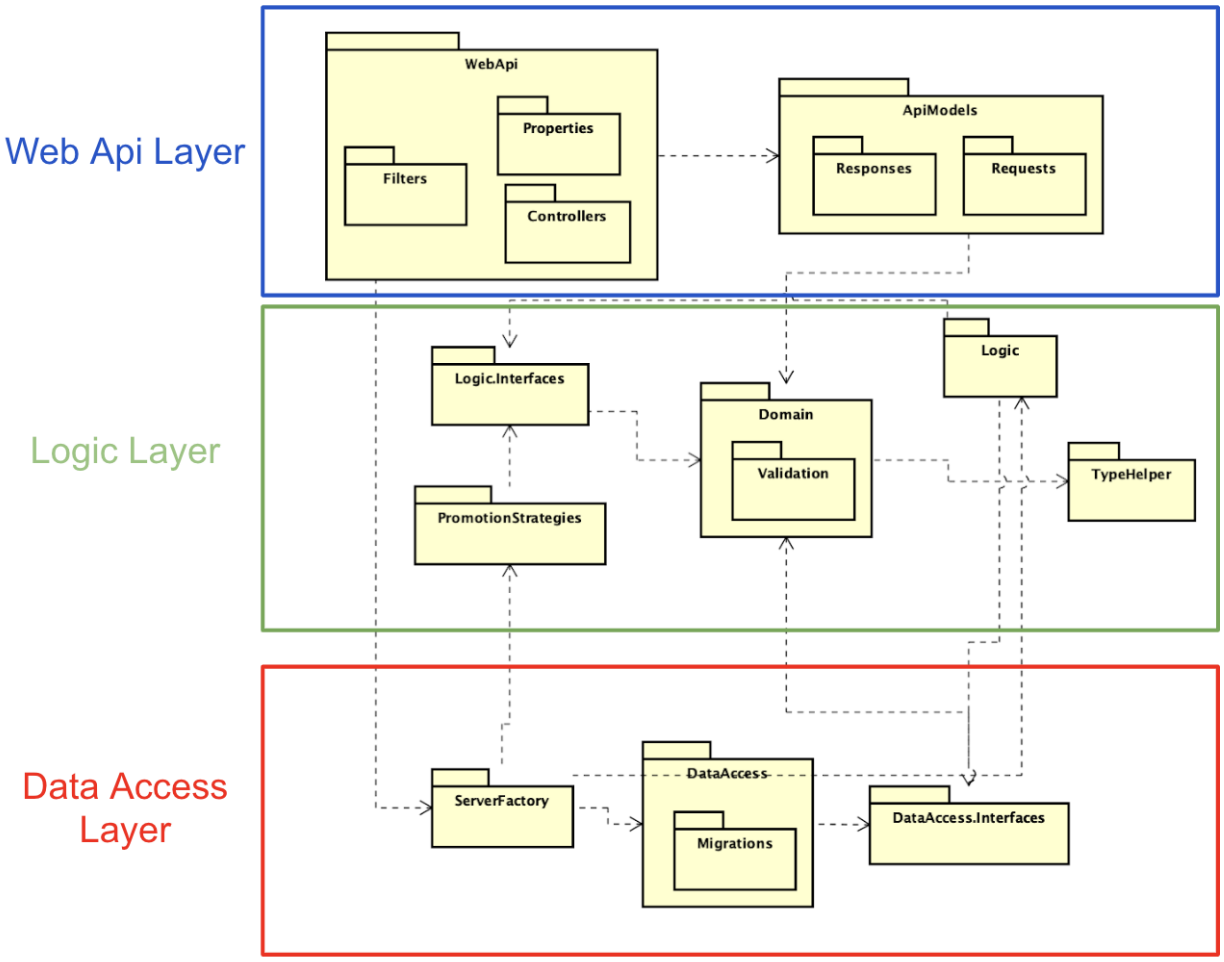
- Crear, modificar o eliminar productos
- Eliminar usuarios
- Ver el historial de todas las compras realizadas
- Ver los usuarios registrados en el sistema, crear, modificar o eliminarlos

El usuario con rol “Ambos” claramente puede hacer todas las opciones.

Nuestro sistema permite ser usado sin haber iniciado sesión, lo que quiere decir que uno puede crear un carrito y agregar o eliminar productos de él sin siquiera tener una sesión. La gran ventaja de tener una sesión (por ende un usuario), es que se puede efectuar una compra, y guardar la información en una base de datos, elemento que es fundamental para la persistencia de nuestra aplicación

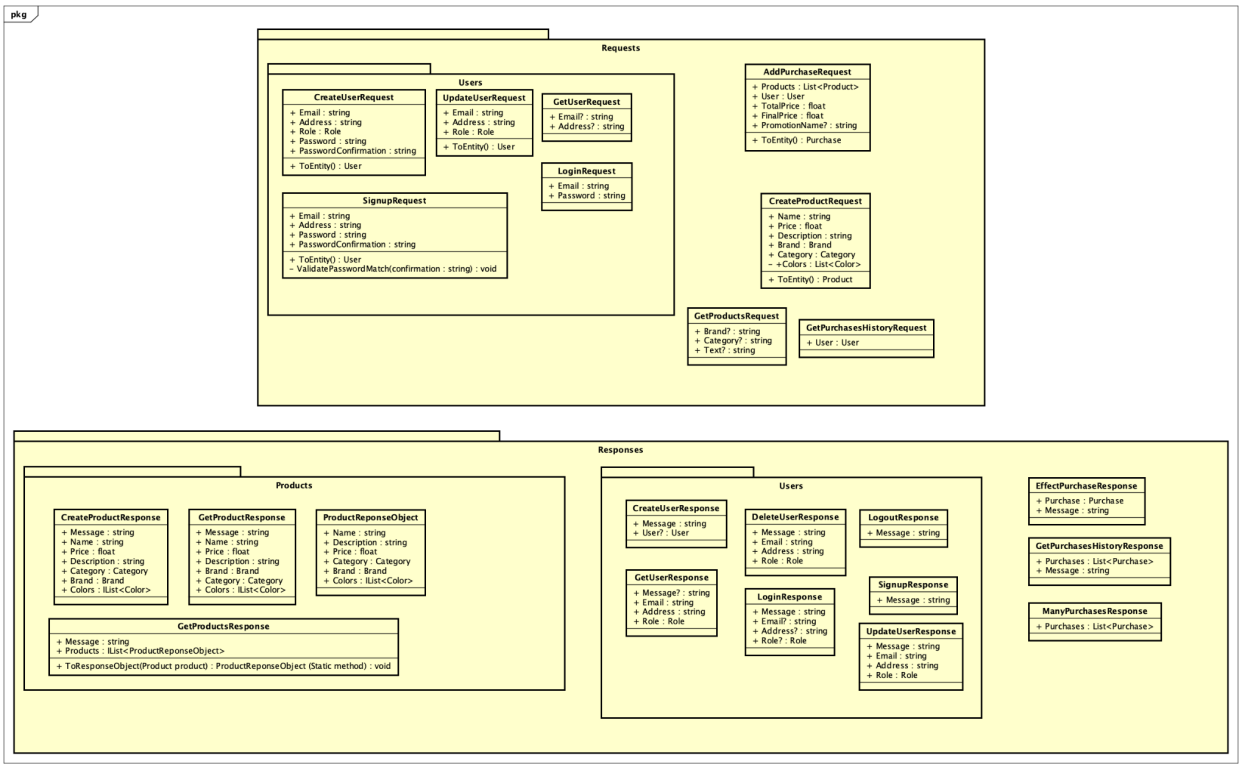
Diagrama general de paquetes:

Intentamos dividir nuestro sistema en las 3 “capas” que se ven a continuación: Primero la Capa de Web API donde vive todo lo relacionado a los controladores, filtros, request-response, etc. Debajo de ella se encuentra la capa lógica, encargada de justamente la lógica del sistema. Aquí podemos encontrar las clases del dominio, el modelaje de los objetos que usamos en nuestro sistema (usuario, producto, compra, etc), y distintos proyectos que agregan al proceso lógico de nuestra aplicación.
Por último, la Data Access Layer. Esta es la capa encargada de manejar la persistencia de nuestro sistema mediante una base de datos, vinculada con el ORM “Entity Framework”.



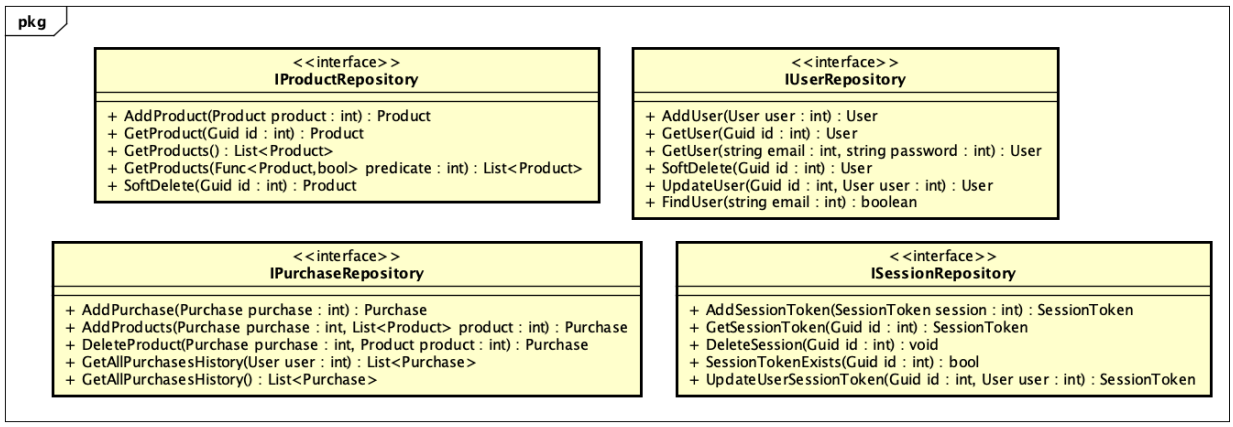
ApiModels:

En este paquete se encuentran los data transfer objects, los cuales son modelos que utilizan la capa de WebApi para no exponer los datos originales. En nuestro sistema, hacemos uso de los “Request” para ingresar los modelos de la petición a la capa lógica, y por otro lado los “Response”, para devolver las respuestas del cliente. Cada posible Request /Response tiene su propia clase para poder manejar los datos entrantes/salientes.



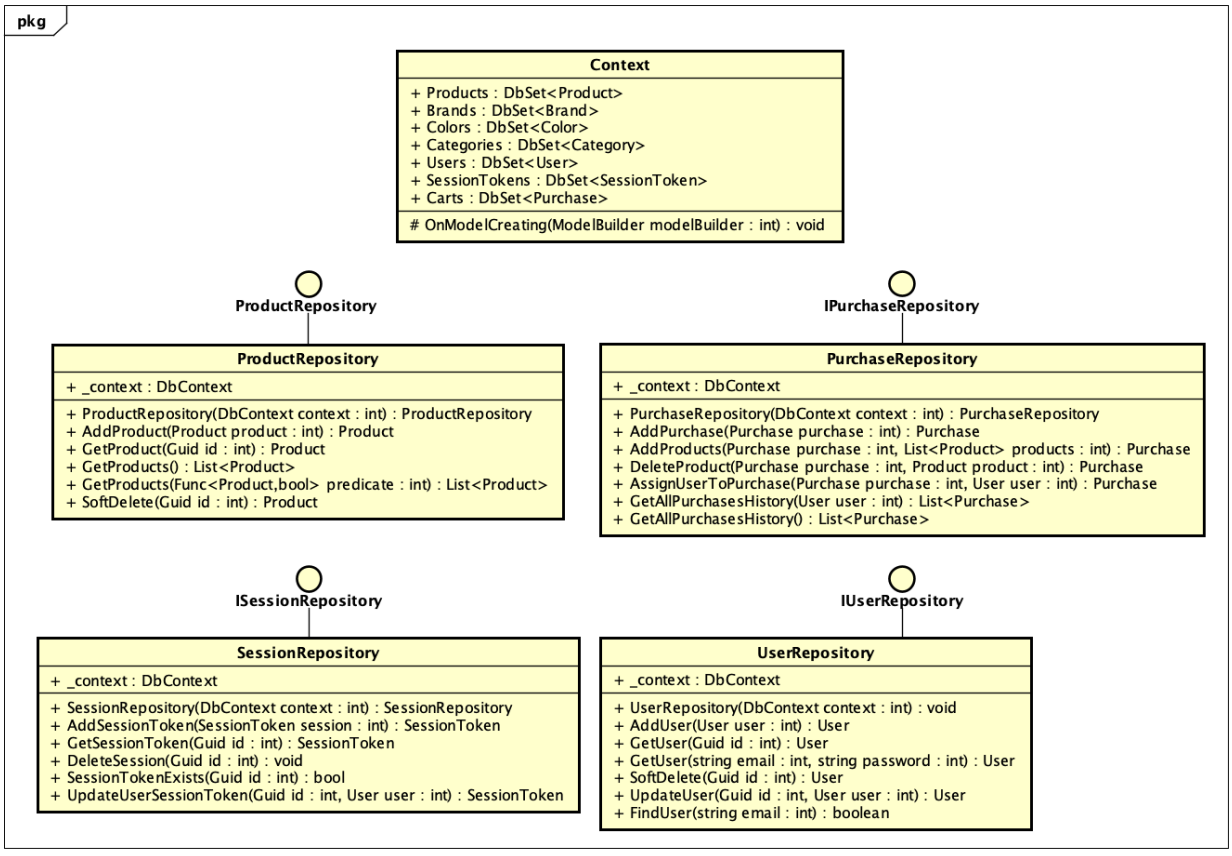
DataAccess.Interfaces:

Este paquete tiene como función principal definir las interfaces para el acceso a la base de datos. Cada una de estas interfaces son implementadas en el paquete DataAccess.



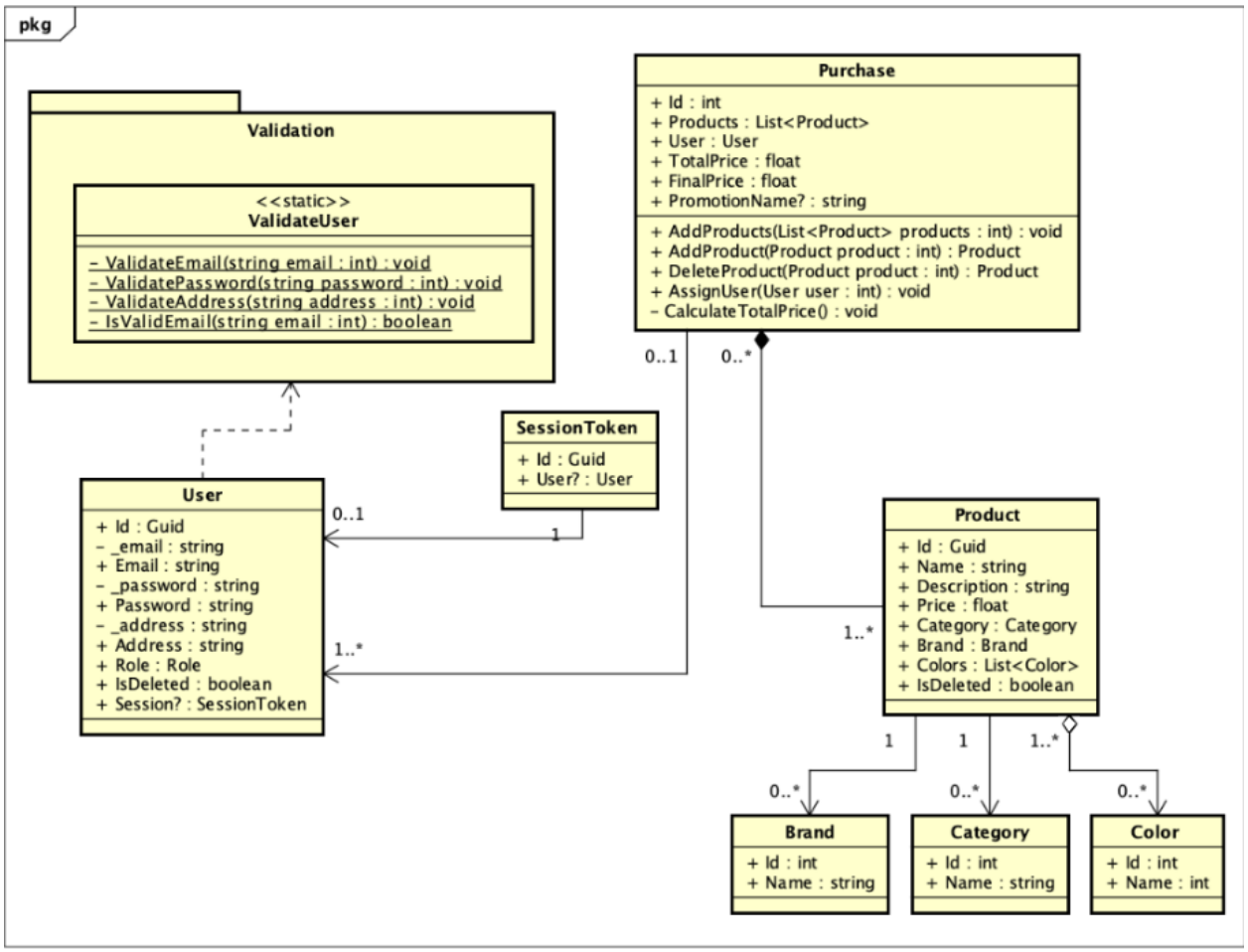
DataAccess:

Aquí se encuentran las implementaciones de las interfaces correspondientes a DataAccess.Interface. Son las entidades para el acceso a los datos de la base de datos para cada entidad de sistema.



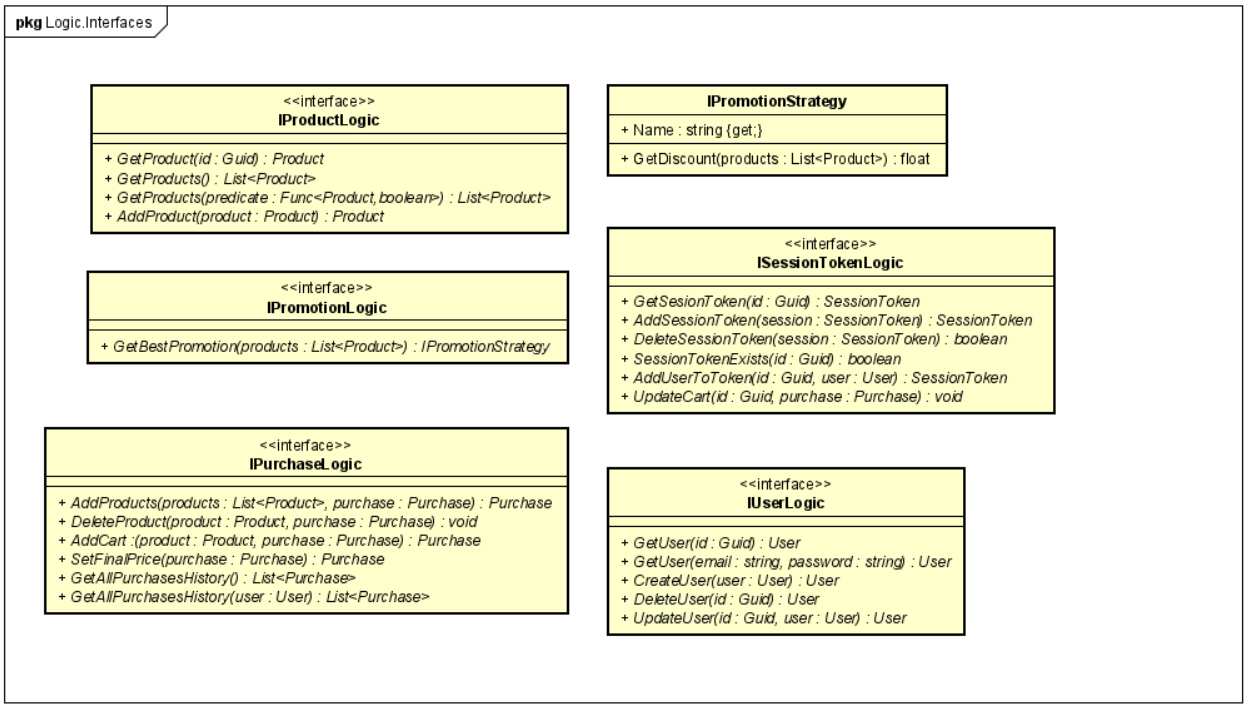
Domain:

Este es el paquete que contiene todas las entidades del negocio de nuestro sistema con relaciones incluidas. Cada uno de estos representará una “tabla” en base de datos.



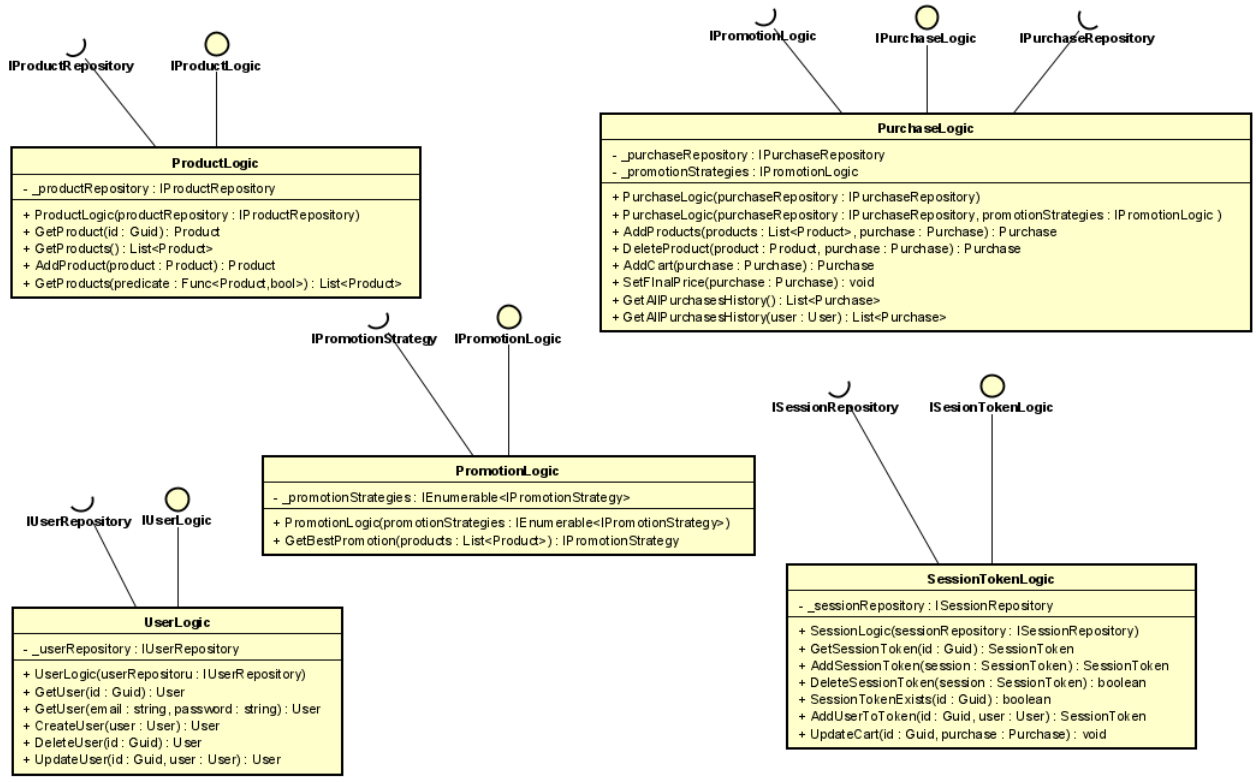
Logic.Interfaces:

La función principal de este paquete es definir las interfaces para el acceso a lógica de nuestro sistema. Cada una de estas interfaces son implementadas en el paquete “Logic”.



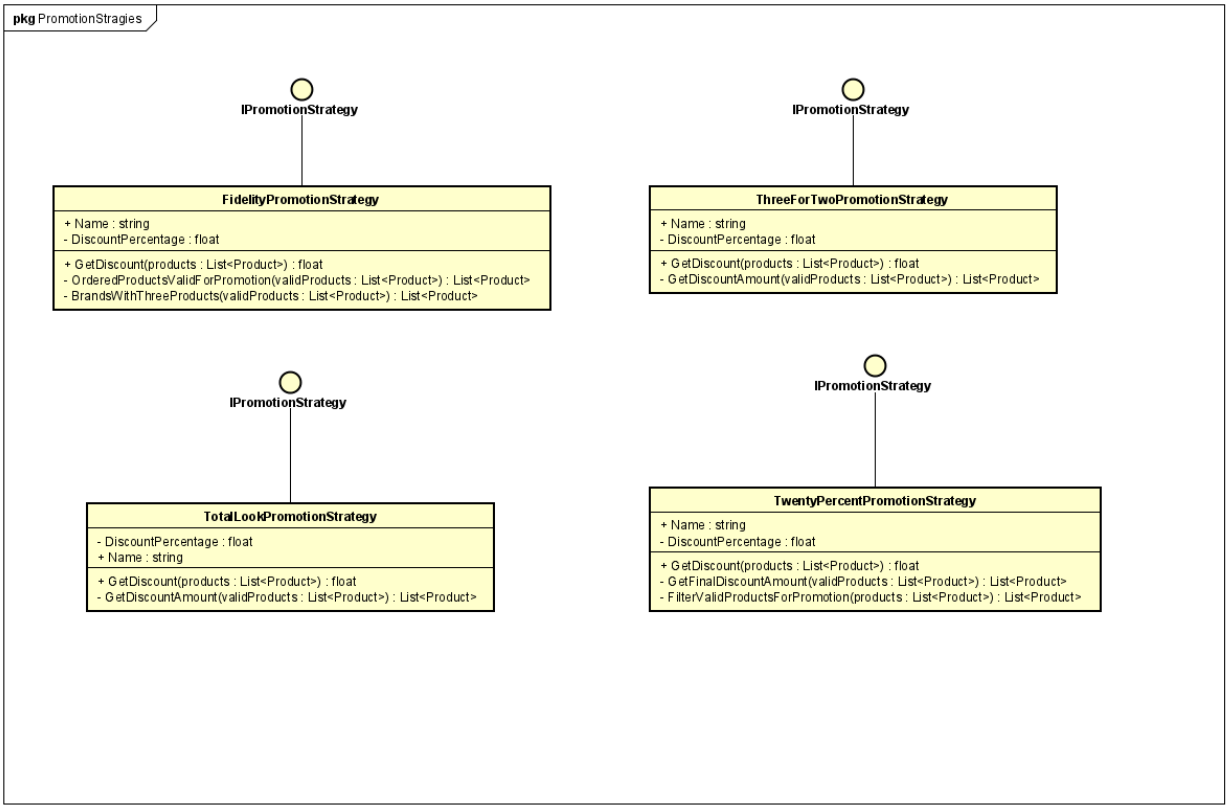
Logic:

Este es el proyecto encargado de manejar las lógicas de nuestra aplicación. Cada una implementa las interfaces mencionadas anteriormente. Son la implementación de las lógicas de negocio, y son una abstracción sobre la capa de datos.



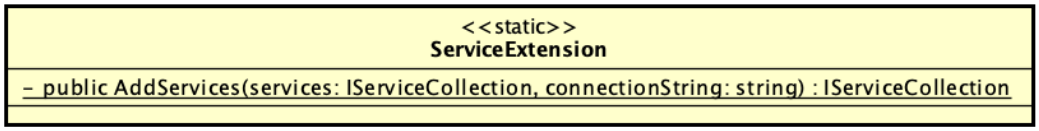
PromotionStrategies:

Aquí se almacenan las distintas promociones aplicables para una compra. Para esto utilizamos el patrón de diseño “strategy” en el cuál, mediante la función GetBestPromotion decidimos en tiempo de ejecución cuál de las promociones (aplicables a la compra) le va a dar un mayor beneficio al usuario.

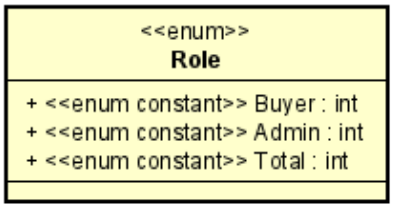


ServerFactory:

Es el encargado de inyectar las dependencias en el proyecto.



TypeHelper:



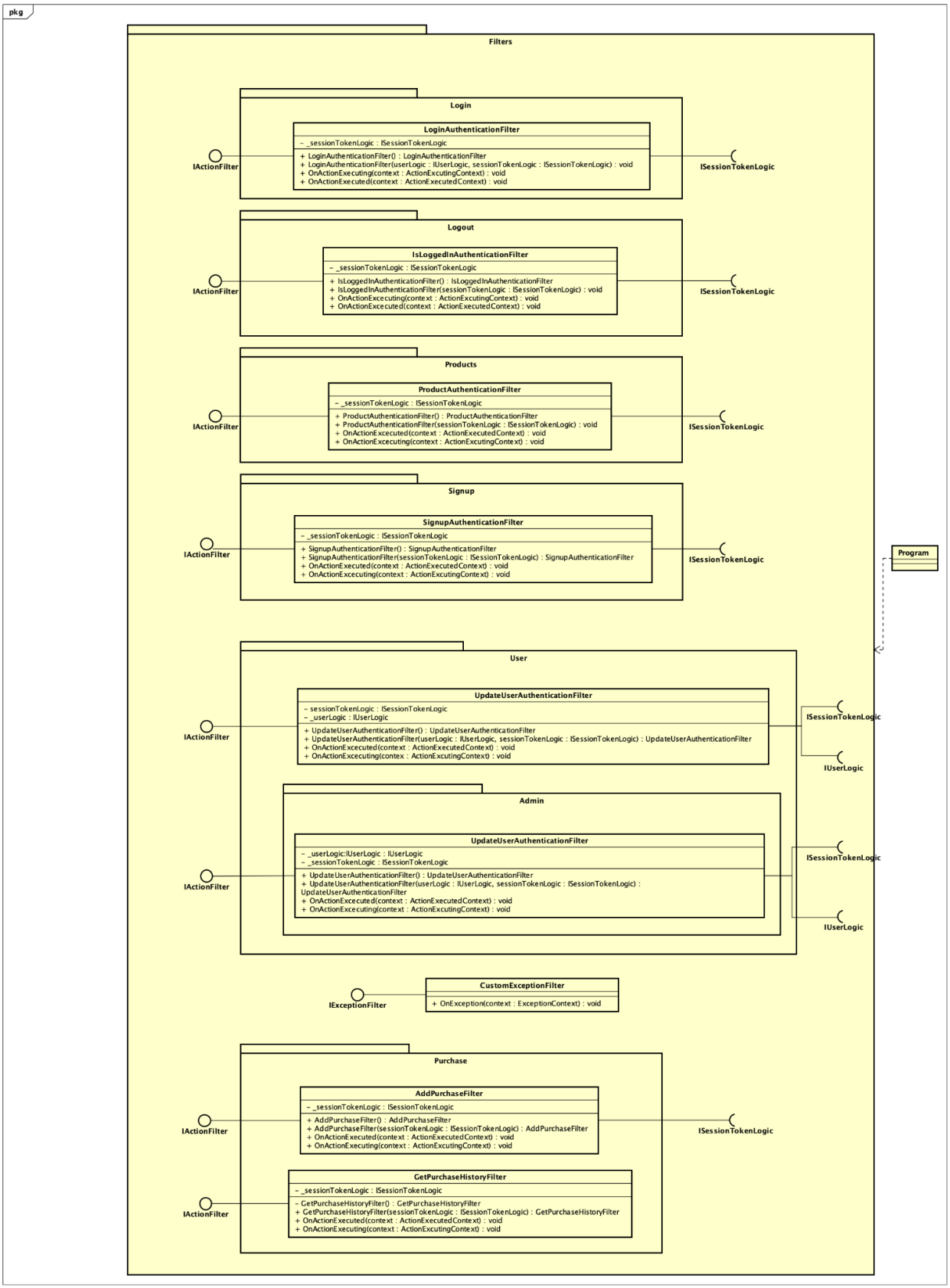
Existe para separar los elementos del dominio que se deben modelar en la base de datos, y los que existen para brindar una comodidad al equipo de desarrollo y seguir lineamientos de clean code.

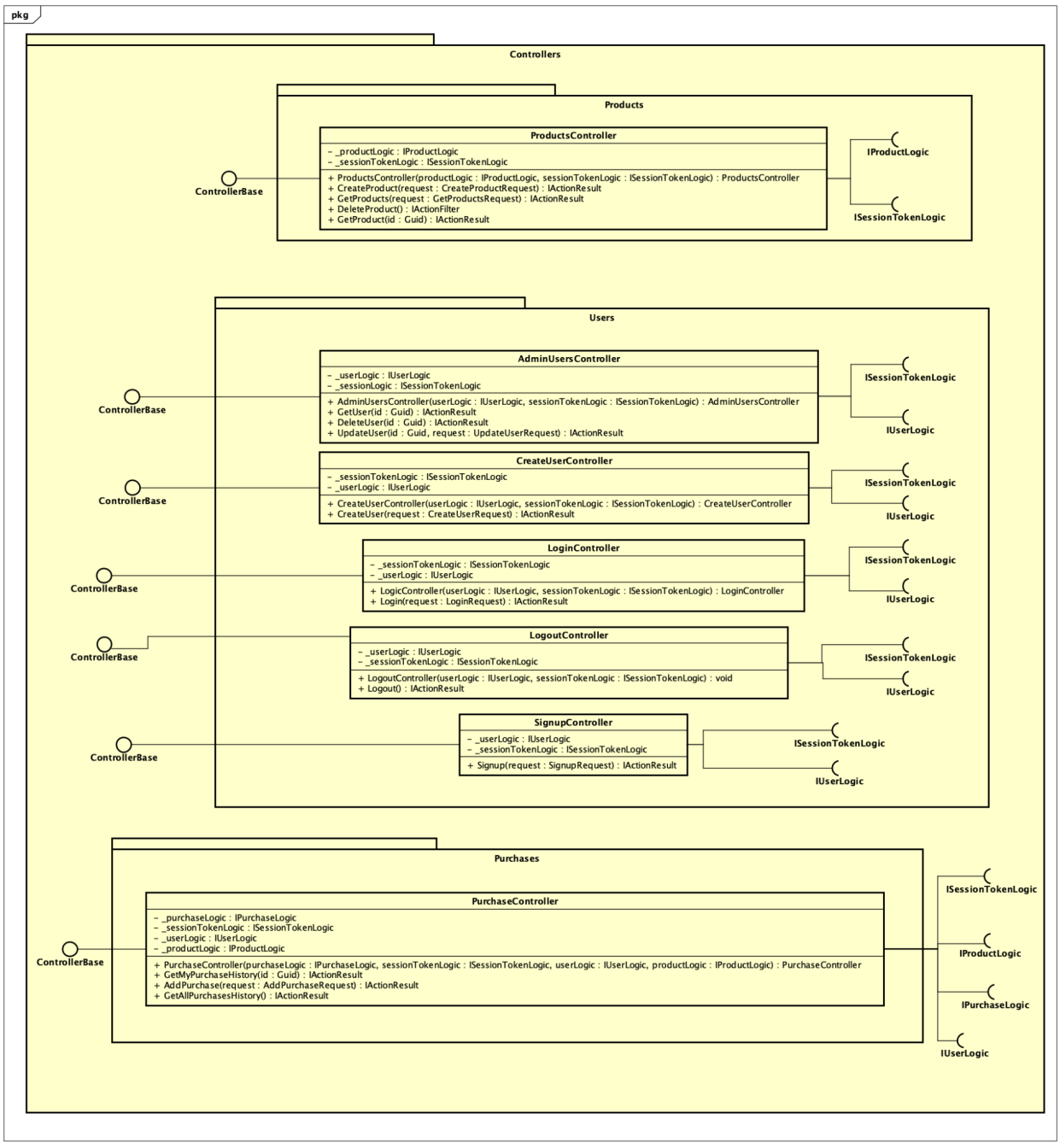
WebApi:

Paquete encargado de resolver las consultas HTTP realizadas por el cliente. Delega la lógica de negocio al paquete de servicios quien se encargará de devolver el resultado a la web api para que ésta interactúe con el cliente. Este proyecto cuenta con 2 paquetes principales: Controllers y Filters.

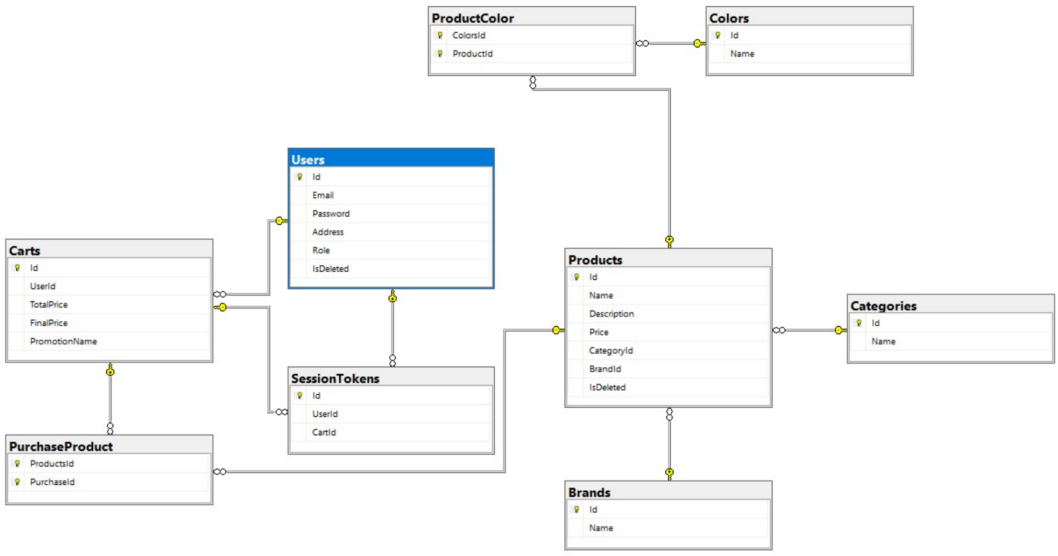
En el momento que llega una request al sistema, esta pasa por los filtros determinados para ese tipo de request, y de esa forma poder hacer una validación previa del paquete antes de llegar al controlador. Los filtros sirven para validar cosas como si el cliente está logueado, si tiene permiso para hacer la operación solicitada, etc.

Una vez pasado el filtro, el paquete llega al controlador, y es aquí donde se empieza a trabajar en efectuar el pedido y devolver una respuesta.


pkg
Filters
Login
LoginAuthenticationFilter
- _sessionTokenLogic : ISessionTokenLogic
+ LoginAuthenticationFilter() : LoginAuthenticationFilter
+ LoginAuthenticationFilter(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
+ OnActionExecuted(context : ActionExecutedContext) : void
IActionFilter
ISessionTokenLogic
Logout
IsLoggedInAuthenticationFilter
- _sessionTokenLogic : ISessionTokenLogic
+ IsLoggedInAuthenticationFilter() : IsLoggedInAuthenticationFilter
+ IsLoggedInAuthenticationFilter(sessionTokenLogic : ISessionTokenLogic) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
+ OnActionExecuted(context : ActionExecutedContext) : void
IActionFilter
ISessionTokenLogic
Products
ProductAuthenticationFilter
- _sessionTokenLogic : ISessionTokenLogic
+ ProductAuthenticationFilter() : ProductAuthenticationFilter
+ ProductAuthenticationFilter(sessionTokenLogic : ISessionTokenLogic) : void
+ OnActionExcecuted(context : ActionExecutedContext) : void
+ OnActionExcecuting(context : ActionExcutingContext) : void
IActionFilter
ISessionTokenLogic
Signup
SignupAuthenticationFilter
- _sessionTokenLogic : ISessionTokenLogic
+ SignupAuthenticationFilter() : SignupAuthenticationFilter
+ SignupAuthenticationFilter(sessionTokenLogic : ISessionTokenLogic) : SignupAuthenticationFilter
+ OnActionExecuted(context : ActionExecutedContext) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
IActionFilter
ISessionTokenLogic
Program
User
UpdateUserAuthenticationFilter
- sessionTokenLogic : ISessionTokenLogic
- _userLogic : IUserLogic
+ UpdateUserAuthenticationFilter() : UpdateUserAuthenticationFilter
+ UpdateUserAuthenticationFilter(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : UpdateUserAuthenticationFilter
+ OnActionExecuted(context : ActionExecutedContext) : void
+ OnActionExecuting(contex : ActionExcutingContext) : void
IActionFilter
ISessionTokenLogic
IUserLogic
Admin
UpdateUserAuthenticationFilter
- _userLogic:IUserLogic : IUserLogic
- _sessionTokenLogic : ISessionTokenLogic
+ UpdateUserAuthenticationFilter() : UpdateUserAuthenticationFilter
+ UpdateUserAuthenticationFilter(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) :
UpdateUserAuthenticationFilter
+ OnActionExcecuted(context : ActionExecutedContext) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
IActionFilter
ISessionTokenLogic
IUserLogic
CustomExceptionFilter
+ OnException(context : ExceptionContext) : void
IExceptionFilter
Purchase
AddPurchaseFilter
- _sessionTokenLogic : ISessionTokenLogic
+ AddPurchaseFilter() : AddPurchaseFilter
+ AddPurchaseFilter(sessionTokenLogic : ISessionTokenLogic) : AddPurchaseFilter
+ OnActionExecuted(context : ActionExecutedContext) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
IActionFilter
ISessionTokenLogic
GetPurchaseHistoryFilter
- _sessionTokenLogic : ISessionTokenLogic
- GetPurchaseHistoryFilter() : GetPurchaseHistoryFilter
+ GetPurchaseHistoryFilter(sessionTokenLogic : ISessionTokenLogic) : GetPurchaseHistoryFilter
+ OnActionExecuted(context : ActionExecutedContext) : void
+ OnActionExecuting(context : ActionExcutingContext) : void
IActionFilter


pkg
Controllers
Products
ProductsController
- _productLogic : IProductLogic
- _sessionTokenLogic : ISessionTokenLogic
+ ProductsController(productLogic : IProductLogic, sessionTokenLogic : ISessionTokenLogic) : ProductsController
+ CreateProduct(request : CreateProductRequest) : IActionResult
+ GetProducts(request : GetProductsRequest) : IActionResult
+ DeleteProduct() : IActionFilter
+ GetProduct(id : Guid) : IActionResult
ControllerBase
IProductLogic
ISessionTokenLogic
Users
AdminUsersController
- _userLogic : IUserLogic
- _sessionLogic : ISessionTokenLogic
+ AdminUsersController(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : AdminUsersController
+ GetUser(id : Guid) : IActionResult
+ DeleteUser(id : Guid) : IActionResult
+ UpdateUser(id : Guid, request : UpdateUserRequest) : IActionResult
CreateUserController
- _sessionTokenLogic : ISessionTokenLogic
- _userLogic : IUserLogic
+ CreateUserController(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : CreateUserController
+ CreateUser(request : CreateUserRequest) : IActionResult
LoginController
- _sessionTokenLogic : ISessionTokenLogic
- _userLogic : IUserLogic
+ LogicController(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : LoginController
+ Login(request : LoginRequest) : IActionResult
LogoutController
- _userLogic : IUserLogic
- _sessionTokenLogic : ISessionTokenLogic
+ LogoutController(userLogic : IUserLogic, sessionTokenLogic : ISessionTokenLogic) : void
+ Logout() : IActionResult
SignupController
- _userLogic : IUserLogic
- _sessionTokenLogic : ISessionTokenLogic
+ Signup(request : SignupRequest) : IActionResult
ControllerBase
ISessionTokenLogic
IUserLogic
Purchases
PurchaseController
- _purchaseLogic : IPurchaseLogic
- _sessionTokenLogic : ISessionTokenLogic
- _userLogic : IUserLogic
- _productLogic : IProductLogic
+ PurchaseController(purchaseLogic : IPurchaseLogic, sessionTokenLogic : ISessionTokenLogic, userLogic : IUserLogic, productLogic : IProductLogic) : PurchaseController
+ GetMyPurchaseHistory(id : Guid) : IActionResult
+ AddPurchase(request : AddPurchaseRequest) : IActionResult
+ GetAllPurchasesHistory() : IActionResult
ControllerBase
ISessionTokenLogic
IProductLogic
IPurchaseLogic
IUserLogic

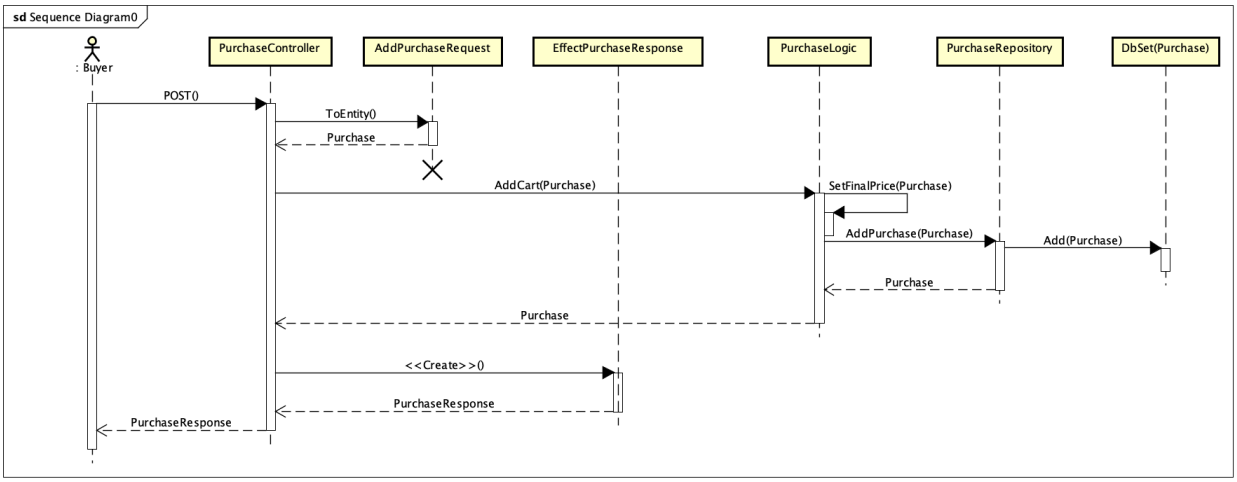
Modelo de tablas de la base de datos:



Diagramas de secuencia:

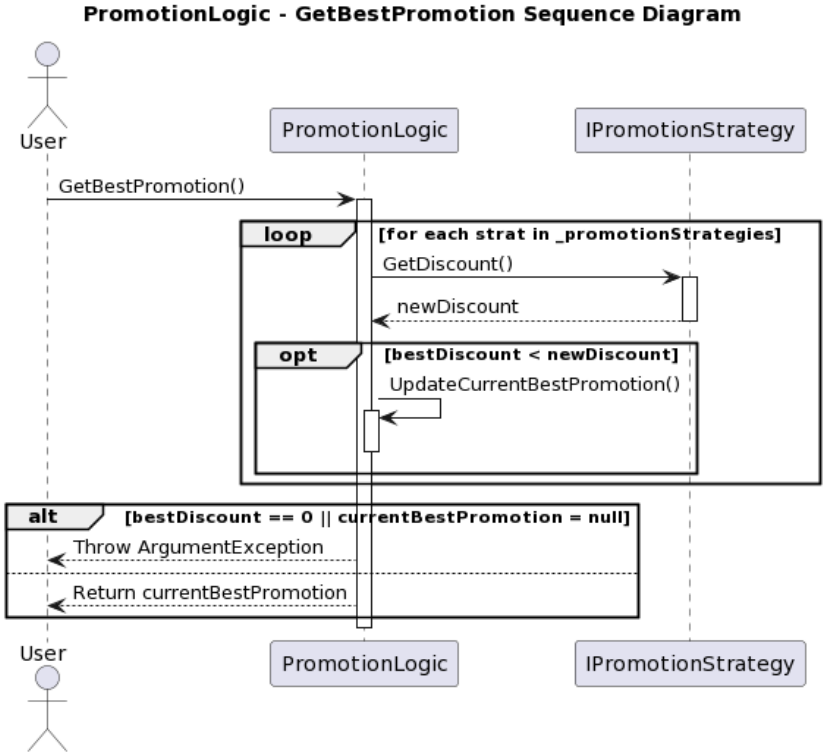
Efectuar una compra:

En este diagrama, la interacción será mediante el controlador de compras, que es quien interactúa con la capa de la lógica y esta con la capa de acceso a datos.
En el caso de efectuar una compra, recibimos peticiones http en donde los controllers serán los encargados de capturarla y delegarla a la lógica de negocios. La lógica hace todo lo que tiene que hacer, incluyendo almacenar esta nueva compra en el repositorio, y la devuelve al controlador.
El controlador crea una respuesta con una imitación de esta nueva compra, y es esto lo que se le retorna al cliente.



Elegir la mejor promoción:

Cuando se llama el metodo GetBestPromotion, el PromotionLogic itera sobre todas las IPromotionStrategy que tiene guardadas dentro de un IEnumerable<IPromotionStrategy> que se cargan por medio de inyección de dependencias, y por cada una de ellas calcula el descuento que se aplicaría a la lista de productos dada. En caso de haber promociones aplicables, retorna la que mejor descuento brinda, y en caso de no haberlo, tira una excepción



Hacer un SignUp:

En este diagrama, vemos como el controller tiene una gran influencia en cuanto a que tiene permitido y que no el cliente a la hora de hacer un signup. Lo primero que hace es preguntar si el paquete viene con una autorización. En el caso de que la tenga, va a fijarse si la sesión ya existe. En este caso se le asigna a la sesión el usuario creado. En caso de que no exista una sesión con ese “auth”, se crea una nueva sesión.

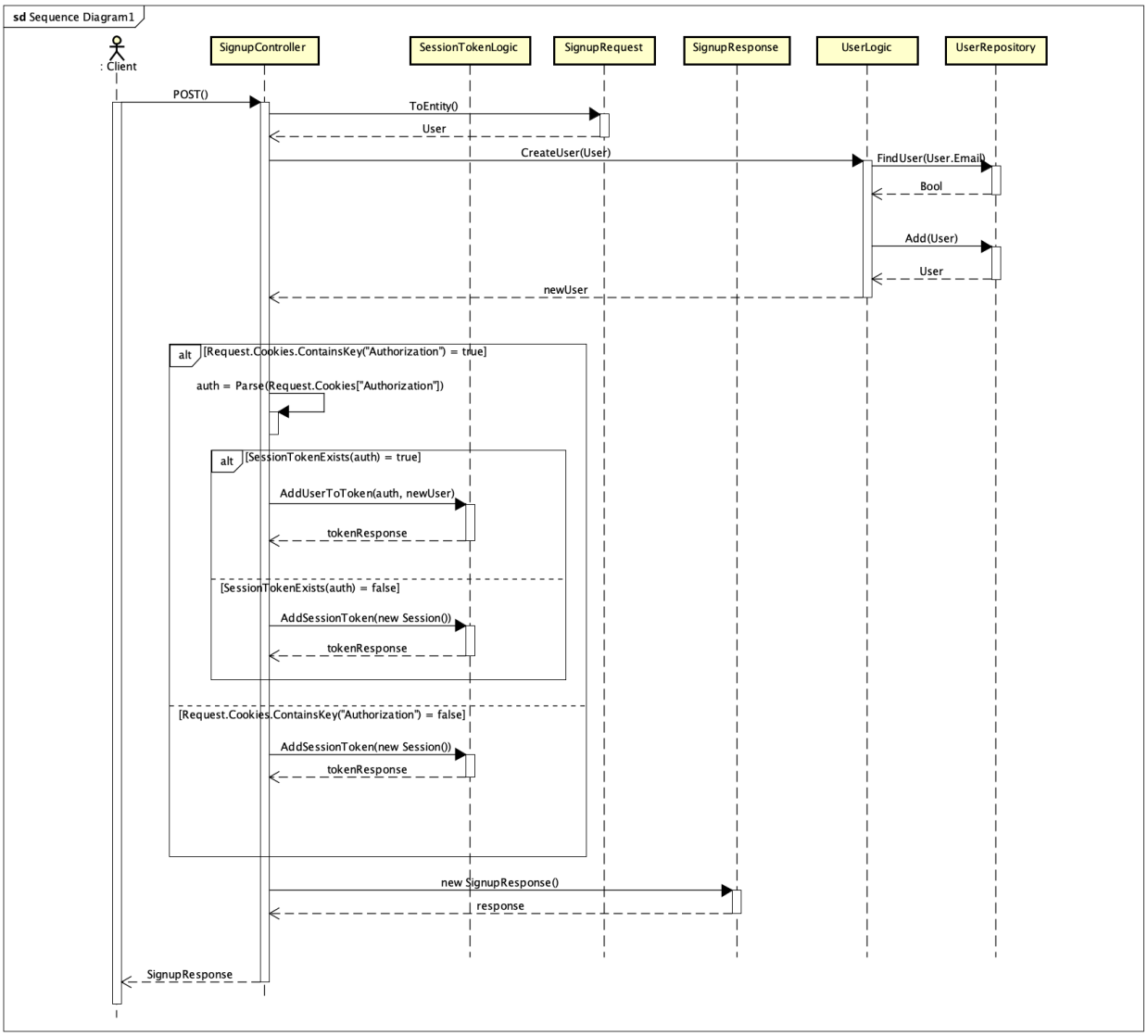


Diagrama de componentes:

En este diagrama uno de los puntos fundamentales que más tuvimos en cuenta para el desarrollo de nuestro sistema es el cumplimiento del principio SOLID: Dependency Inversion Principle (DIP). Para esto, creamos las interfaces de DataAccess y las de LogicInterface. De esta forma, ningún módulo de alto nivel dependerá de algún módulo de bajo nivel.

Otra ventaja de haber implementado estas interfaces, es que estamos cumpliendo con el principio de Interface Segregation Principle (ISP),

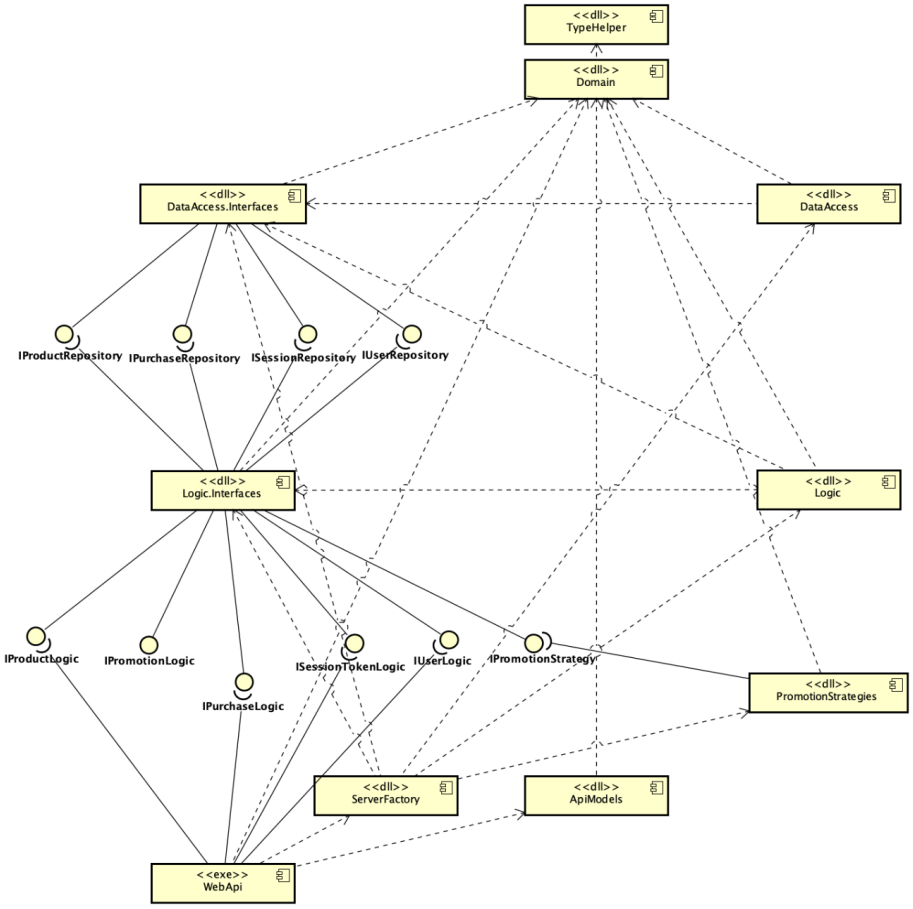


Diagrama de despliegue:

Justificación y explicación del diseño:

A lo largo de todo el sistema fuimos aplicando patrones y principios vistos en cursos anteriores como Diseño de Aplicaciones 1 y los vistos en esta materia. Intentamos que la implementación del sistema sea lo más desacoplada posible con el fin de permitir nuevas implementaciones sin que impacten de gran manera en nuestro código base.

Como el diagrama de paquetes lo muestra, todo el backend se encuentra segmentado en 3 capas con el fin de respetar el principio SRP. Cada capa está organizada según sus responsabilidades para ayudar a una mejor organización del código y en caso de necesitar solucionar un error o simplemente detectar una funcionalidad que se requiera, esto pueda ser lo menos engorroso posible.

Patrones de diseño:

Uno de los patrones que notoriamente destaca es el patrón Fachada. En nuestro caso, la fachada es el paquete “WebApi”, el cual contiene los endpoints que son los puntos de acceso al sistema para el cliente. Las únicas entradas y salidas se encuentran en este paquete con la idea de ocultar todos los llamados a las otras implementaciones y no quede todo expuesto. Podemos afirmar que dicho paquete no contiene lógica compleja, solamente se encarga de llamar a las funcionalidades más profundas de nuestro sistema.

Para elegir la mejor promoción aplicamos el patrón Strategy. Iteramos sobre una lista de IPromotionStrategy, la cual se encuentra en el mismo paquete que la lógica para aplicar inversión de dependencia, y por medio del método “GetDiscount” obtenemos la mejor promoción para la lista de productos dada.

Modelo de EntityFramework:

Utilizamos la modalidad de code-first, lo que nos permitió establecer un modelo conceptual de nuestro código para aplicarse en la generación de la base de datos. Este modelo, se basa en las clases e implementaciones de nuestro código para reflejarse como tablas en la base de datos SQL. El framework que aplicamos fue Entity Framework Core 7.0, por lo que fue el encargado de realizar dicho procedimiento, generando las migraciones y actualizando la base de datos según sea oportuno.

En cuanto a la base de datos, realizamos varias migraciones principalmente con el fin de trasladar el código a las tablas virtuales, pero también para reflejar el proceso de adaptación del código sobre todo en dominio. La idea siempre es mostrar la evolución progresiva de las tablas.

Otro detalle a resaltar es el uso de los modelos como medio de transferencia de datos (paquete ApiModels) a los clientes. Cada uno de ellos está creado con un cierto propósito, ya sea como forma de mostrar una serie de datos al cliente o simplemente tener un modelo de ingreso y salida de datos al sistema como los llamados “Request” y “Response”. La idea de esto es no tener que darle al cliente objetos 100% reales de nuestra base de datos, sino una abstracción.

Para cada repositorio, se cuenta con su correspondiente interfaz con el fin de que si en un futuro, nosotros cambiamos la capa de lógica, dicho cambio no impacte totalmente en nuestra implementación concreta de DataAccess. Lo mismo sucede con las interfaces de la lógica de negocio.

Para evitar el alto acoplamiento, hicimos uso de la inyección de dependencia. La misma consiste en una dependencia que un objeto que otro objeto necesita, donde el segundo depende del primero. Este objeto puede ser de cualquier tipo, por ejemplo clase, módulos, librerías, etc. Es muy útil, ya que nos ahorramos la declaración de muchísimos new Object() que nos acoplan a la implementación de dicho objeto, lo cual desde el punto de vista del principio de diseño Open/Closed no está nada bueno. Además, colaboramos con el principio de inversión de dependencia (DIP) donde cada implementación depende de una abstracción. En consecuencia obtenemos un código mucho más fácil de mantener, mejor para testear y limpio.

Intentamos realizar la mejor aplicación desde el punto de vista de Clean Code. Evitamos tener números mágicos o comentarios sumamente intrascendentes. Nosotros creemos que esto es de mayor importancia sobre todo para la degradación del código, que no es algo menor a la hora del trabajo en equipo.

Filtros:

Hicimos uso de dos tipos de filtros principalmente, el de excepciones y el de autorización.
Los filtros de autorización fueron implementados como filtros de acción para escapar las complejidades que nos iba a exigir el framework y que escapan el alcance de esta materia.

Implementamos distintos filtros de autorización para distintos fines. En algunos casos nos interesaba verificar que el usuario no tenga una sesión, por ejemplo en el LogIn. Por lo contrario, en casos como el LogOut, nos interesaba verificar que el usuario tenga una sesión asociada.
Otros filtros implementados son importantes para verificar el rol del usuario, ya que este puede ser Comprador, Administrador o Ambos. Por ejemplo: Para crear un producto, el usuario debe ser administrador, pero para poder hacer una compra, el usuario únicamente puede ser un comprador.

El filtro de excepciones es fundamental ya que ayuda capturar la todas las excepciones que nuestro sistema lanza. Hicimos un filtro global para las excepciones llamado CustomExceptionFilter. El mismo cumple la función de mapear cada excepción con un status code para la respuesta de error.

La clase ServiceExtension, ubicada en el paquete ServiceFactory es la encargada de la responsabilidad de inyección de dependencias. Básicamente se encarga de realizar las inyecciones de dependencias entre varias interfaces del sistema. Bajo nuestro punto de vista es una representación del patrón Fabricación Pura, ya que es una clase creada artificialmente, simplemente con el propósito de encargarse de una responsabilidad en específico y desacoplar lo más posible el sistema. Se le delega la tarea que en principio debería realizar la clase startup de WebApi. Con el fin de dejar lo más ligera posible a dicha capa se delegó la responsabilidad. Esto resuelve el problema de donde hacer el “new” y tener que pasar la instancia por parámetro por toda la aplicación.

Errores conocidos:

Sesiones con user en null:

Las sesiones están implementadas de tal forma que el usuario pueda manejarse como null. Esto es así, ya que contemplamos las sesiones para manejo de carrito, asumiendo que a nivel backend se manejaría el carrito de compras, y por ende queríamos permitir que alguien tenga un carrito sin tener una cuenta asociada. Dado que pocos días antes descubrimos que este requerimiento no sería contemplado para esta primera etapa, lo dejamos como está implementado para enfocarnos en lo restante de la aplicación, sin sumar la complejidad de cambiar esto, luego de tener todo testeado e implementado.