Universidad ORT Uruguay
Facultad de Ingeniería

Obligatorio 1
Diseño de Aplicaciones 2

Repositorio: https://github.com/IngSoft-DA2-2023-2/242493-260956-281651

Romina Arour - 242493

Daiana Aysa - 281651

Ana Gutman - 260956

Tutores:

Francisco Bouza, Juan Irabedra, Santiago Tonarelli

2023

ÍNDICE

Descripción general del trabajo

En el transcurso de este informe, detallaremos el diseño integral de la aplicación web que desarrollamos como parte del proyecto obligatorio 2 de Diseño de Aplicaciones 2. A lo largo de estas páginas, proporcionaremos justificaciones fundamentadas para cada una de nuestras decisiones de diseño.

Este proyecto surge como respuesta a la creciente demanda de las tiendas de ropa en Uruguay de expandir su presencia en línea, brindando a los usuarios la posibilidad de explorar y adquirir productos a través de la web. Además, está enfocado en la implementación de promociones estratégicas que fortalezcan la fidelidad de la clientela.

La plataforma fue diseñada con modos de navegación específicos en función de si el usuario es comprador o administrador en el sistema. A los usuarios compradores se les brinda la capacidad de armar un carrito de compras y realizar transacciones de compra de productos de manera sencilla y segura. En cambio, los administradores tienen funcionalidades avanzadas como la posibilidad de acceder al histórico de compras del sistema y la capacidad exclusiva de visualizar y editar la lista de usuarios registrados. De este modo se les otorga un control integral sobre la gestión de la plataforma. Asimismo, se contempla la posible existencia de usuarios que son tanto administradores como compradores en la página, quienes acceden a todas las funcionalidades existentes en la misma. De esta manera se logra ofrecer una experiencia personalizada y adecuada tanto a las necesidades específicas de cada tipo de usuario como a las de la empresa.

En cuanto a la implementación técnica, usamos C# como lenguaje para el desarrollo del backend, aplicando el desarrollo guiado por pruebas (TDD) como metodología central.Para la realización de las aplicaciones de consola, la API Rest, y los proyectos de pruebas unitarias, optamos por utilizar .NET Core 6.0. Las pruebas unitarias fueron elaboradas con MSTest y Moq, proporcionando un marco robusto y eficiente para la validación de la lógica del código.

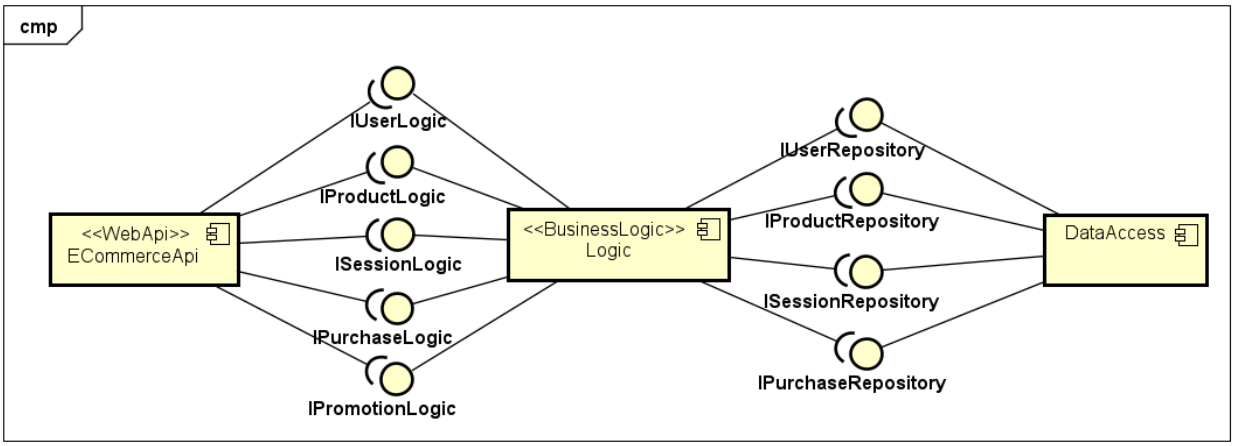
La persistencia de datos se llevó a cabo a través de Entity Framework Core, garantizando una gestión segura y eficiente de la información en la base de datos. Para el frontend, diseñamos una aplicación de página única (SPA) haciendo uso de AngularJS. Este enfoque no solo proporciona una experiencia de usuario fluida, sino que también simplifica la gestión dinámica de plantillas sin necesidad de recargar la página.

En las secciones posteriores, explicaremos en detalle cada uno de estos componentes, destacando las decisiones de diseño que respaldan la funcionalidad y eficiencia de la aplicación.

Descripción de paquetes

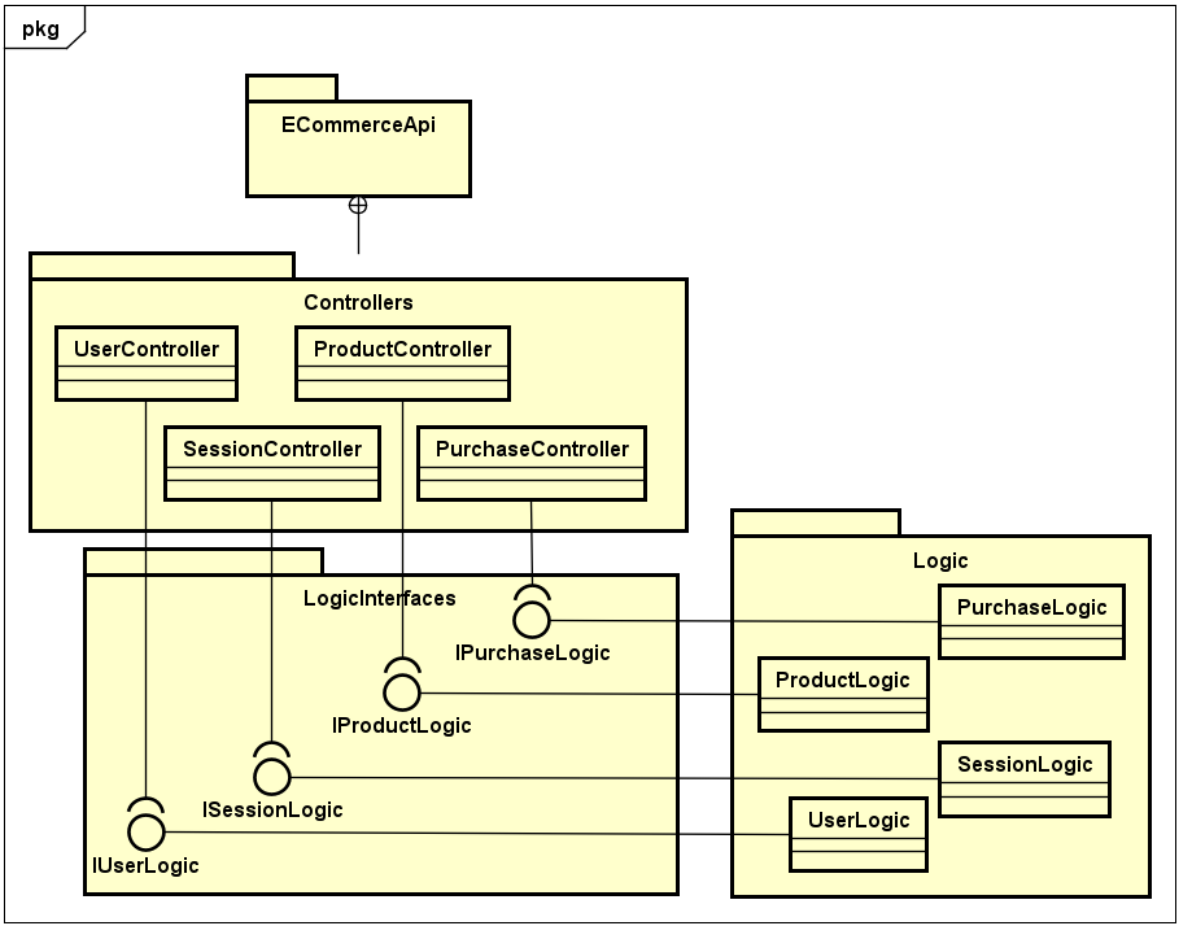
Vista de Desarrollo

Para la vista de desarrollo se diseñó el siguiente diagrama de componentes



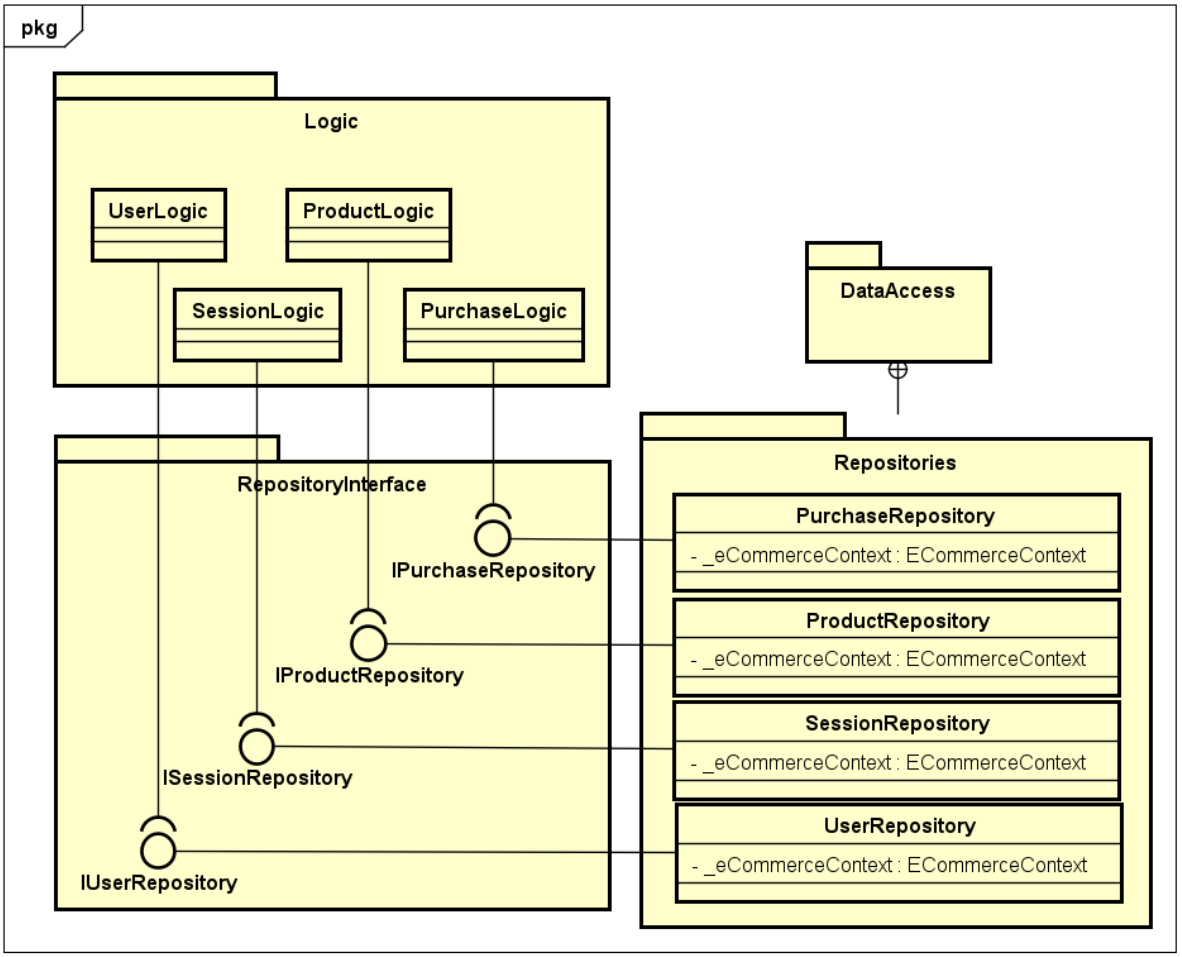
Como se puede observar en la imagen, los paquetes ECommerceApi y Logic interactúan a través de Interfaces basadas en cada clase de Logic que se usa.
Pasa lo mismo entre Logic y DataAccess, donde se usan interfaces para acceder a las funciones del repositorio.

A su vez, diseñamos los siguientes diagramas de paquetes:



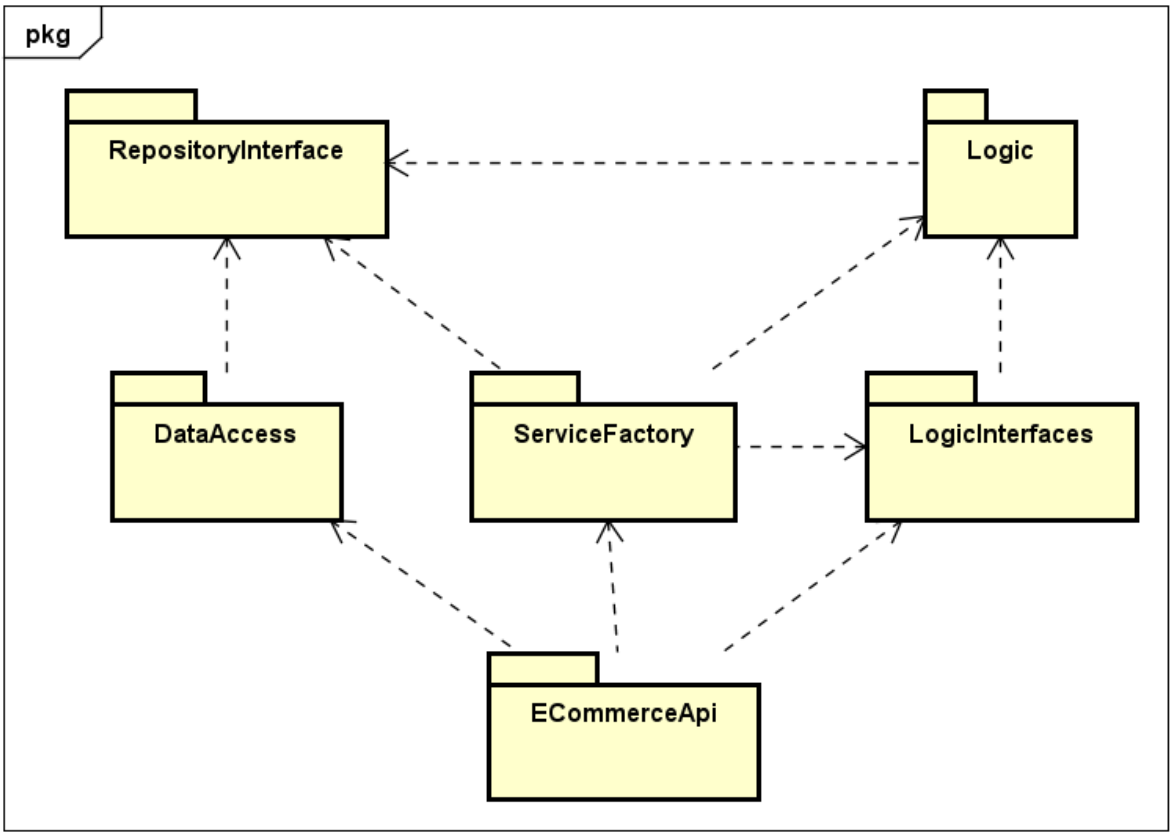
Acá entramos más en detalle en esa interacción de ECommerceApi y Logic, donde se puede observar como por ejemplo ProductController usa IProductLogic para poder utilizar

los métodos definidos en ProductLogic, sin accederlos directamente. Pasa exactamente lo mismo con las otras clases de controllers.



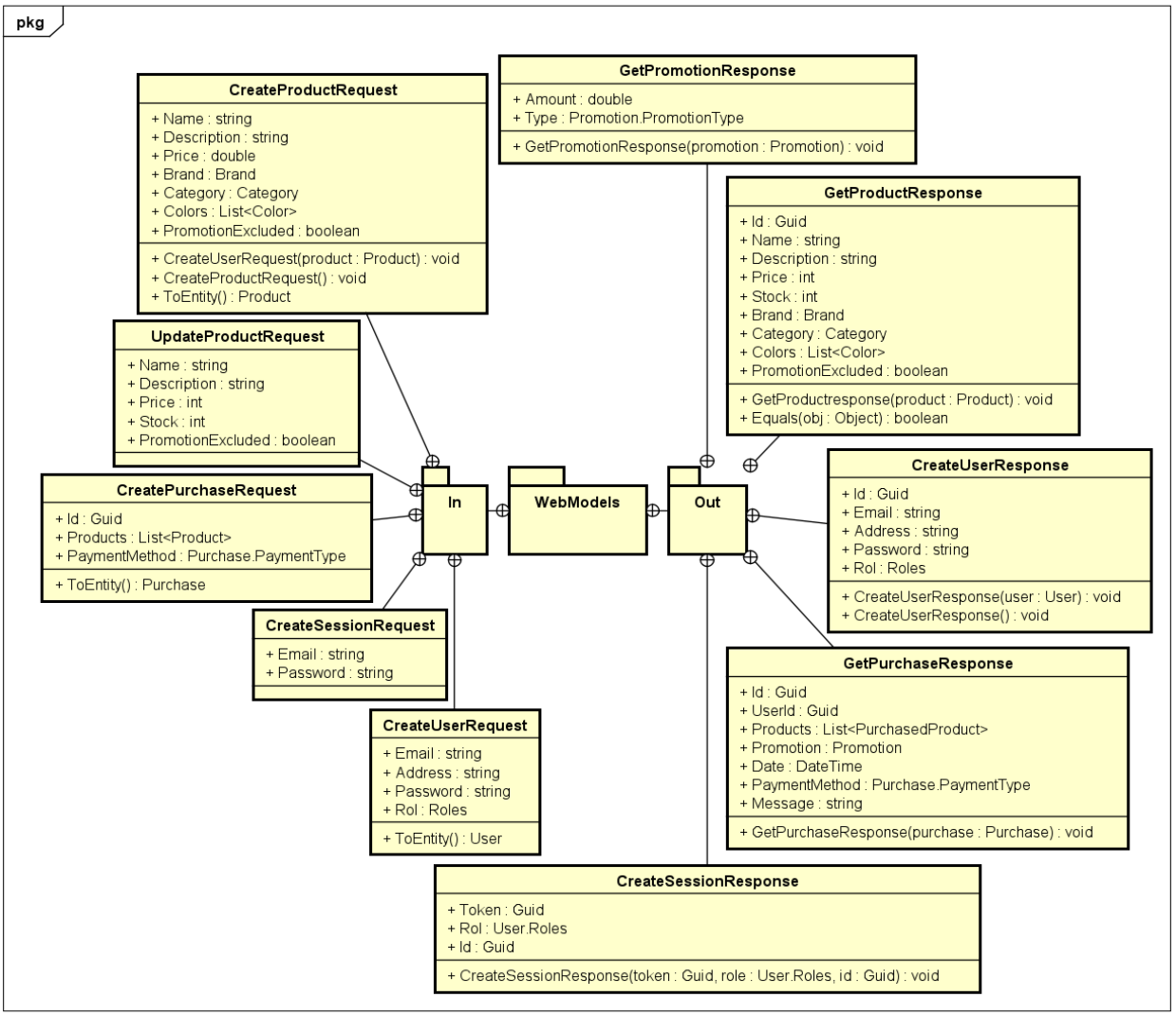
Al igual que en el diagrama anterior, en este se detalla de manera más específica la interacción entre las clases de lógica del paquete "Logic" y el paquete "Repository", siendo este último el encargado de acceder a la base de datos. Siguiendo el ejemplo previo del diagrama, una vez que el controlador de productos ("ProductController") utiliza un método a través de la interfaz "IProductLogic" de la clase "ProductLogic", esta última emplea la interfaz "IProductRepository" para acceder al método de la clase "ProductRepository". Es en este último donde se gestionan los datos, implementando así la funcionalidad deseada, ya sea crear, modificar, eliminar o buscar datos de un producto, según el ejemplo proporcionado.

Esta misma estructura se replica para las demás clases de lógica y de repositorio de ambos paquetes, manteniendo la coherencia en la gestión de datos y garantizando la modularidad del sistema.



La figura anterior ilustra de manera específica la implementación de la inyección de dependencias en nuestro sistema.

Vista lógica

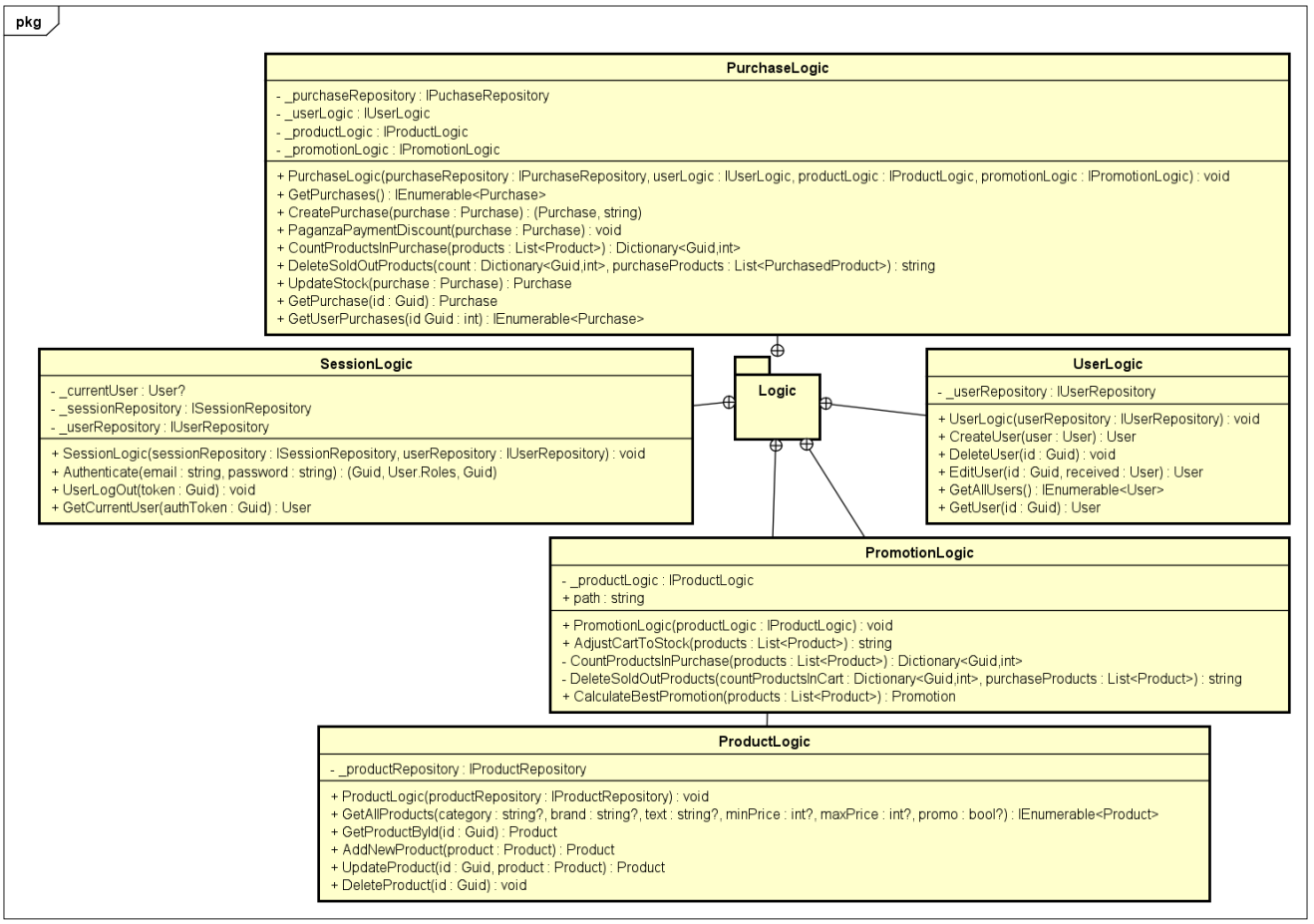


En el siguiente diagrama, se presenta la organización de los modelos de Request y Response de nuestra API. Se han definido dos entradas para el modelo de producto ("Product") según si se trata de una solicitud tipo POST o PUT, ya que los datos proporcionados pueden variar en función de la acción.

En el modelo "CreateProductResponse", se incluyen todos los atributos presentes en el dominio del producto. Por otro lado, en el modelo "UpdateProductRequest", se han seleccionado específicamente los campos que un administrador puede modificar al editar un producto. Estos campos incluyen, según lo detallado en el diagrama, el nombre, descripción, precio, stock y "PromotionExcluded".

Se tomó la decisión de no incluir los atributos de "Category", "Brand" y "Colors" en la actualización, ya que modificar cualquiera de estos elementos se considera un cambio sustancial, equiparable a crear un nuevo producto. Esta elección se fundamenta en la idea de limitar los campos modificables para mantener coherencia y evitar cambios bruscos que

transformen radicalmente un producto, como por ejemplo, de un pantalón a un par de zapatos. Se considera más apropiado crear un nuevo producto en tales casos.



Dentro del paquete de Logic, encontramos cinco clases que desempeñan funciones específicas en el sistema.

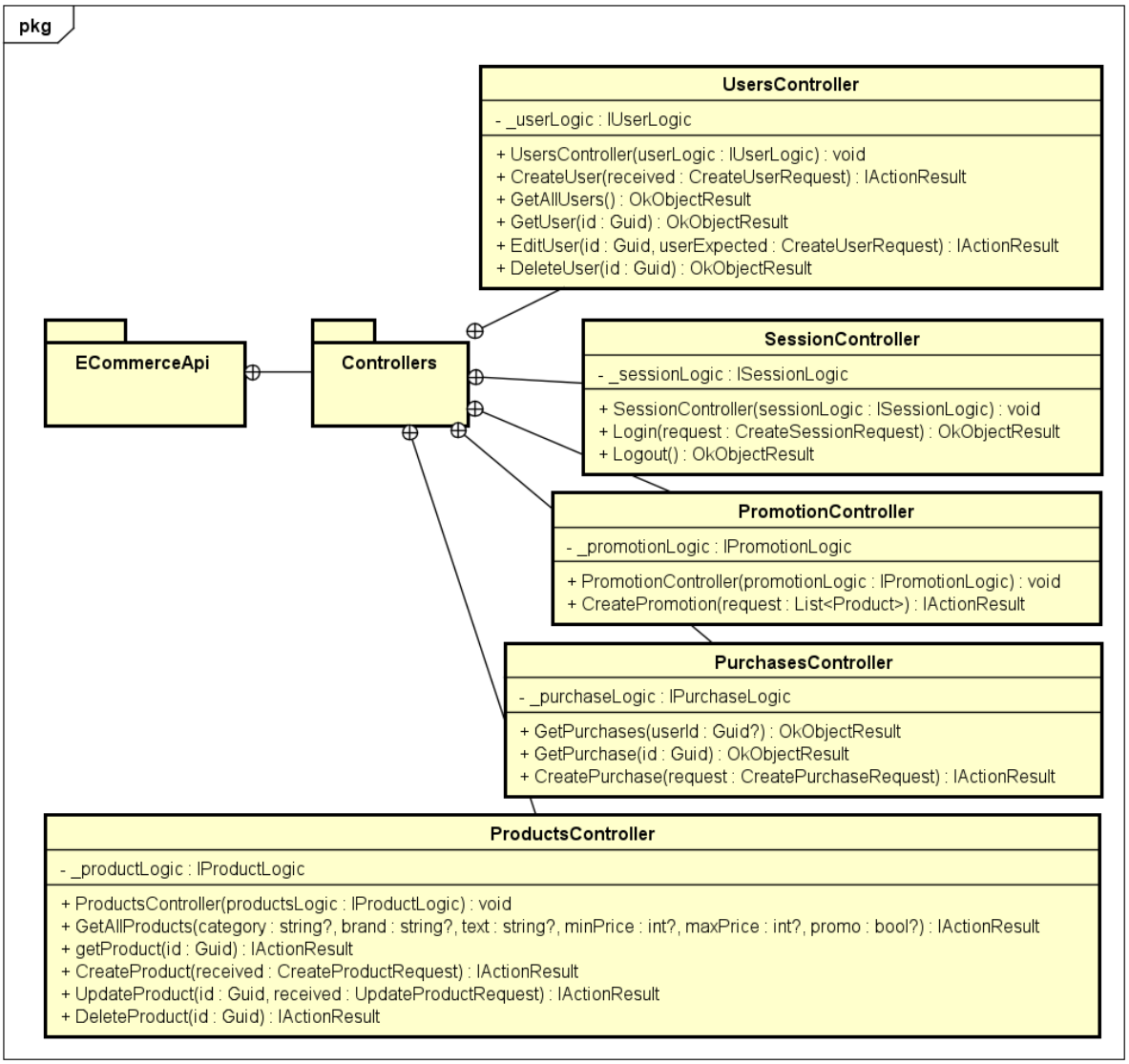
La clase ProductLogic se encarga de la creación, acceso, edición y eliminación de cada producto en el sistema, gestionando también posibles errores que puedan surgir durante estas operaciones.

La clase UserLogic, de manera análoga a ProductLogic, se ocupa del manejo de datos relacionados con cada usuario en el sistema, ya sea para crear, eliminar o modificar la información asociada a ellos.

SessionLogic es la clase responsable de gestionar qué usuarios están activos o inactivos en el sistema, controlando los eventos de inicio de sesión y cierre de sesión.

PurchaseLogic y PromotionLogic están interconectadas en cierta medida. PurchaseLogic se encarga de la creación y manejo de errores en las compras realizadas por un usuario. Dentro de PurchaseLogic, se invoca a PromotionLogic para calcular las posibles promociones aplicables a la compra. Una vez realizado este proceso, se finaliza el cálculo del descuento sobre el precio original y se determina el precio final. Es importante destacar

que PurchaseLogic registra los datos específicos de la compra en ese momento, y estos no se ven afectados en el futuro incluso si un administrador decide editar los productos relacionados.



En este diagrama, se presentan las clases dentro del paquete ECommerceApi, cada una de las cuales implementa las operaciones GET/POST/PUT/DELETE necesarias para las diversas interacciones en el sistema.

De manera general, las clases suelen gestionar dos operaciones GET (una que devuelve todos los usuarios y otra diseñada para buscar uno específico), así como POST para la creación de un nuevo usuario, PUT para la edición del usuario y, finalmente, DELETE para la eliminación del mismo. En estas clases, se manejan los errores provenientes de la lógica de los respectivos controladores, y también se verifican los permisos asociados a cada tipo de usuario para las acciones mencionadas.

En el SessionController, se gestionan un POST y un DELETE, dedicados al inicio y cierre de sesión, respectivamente.

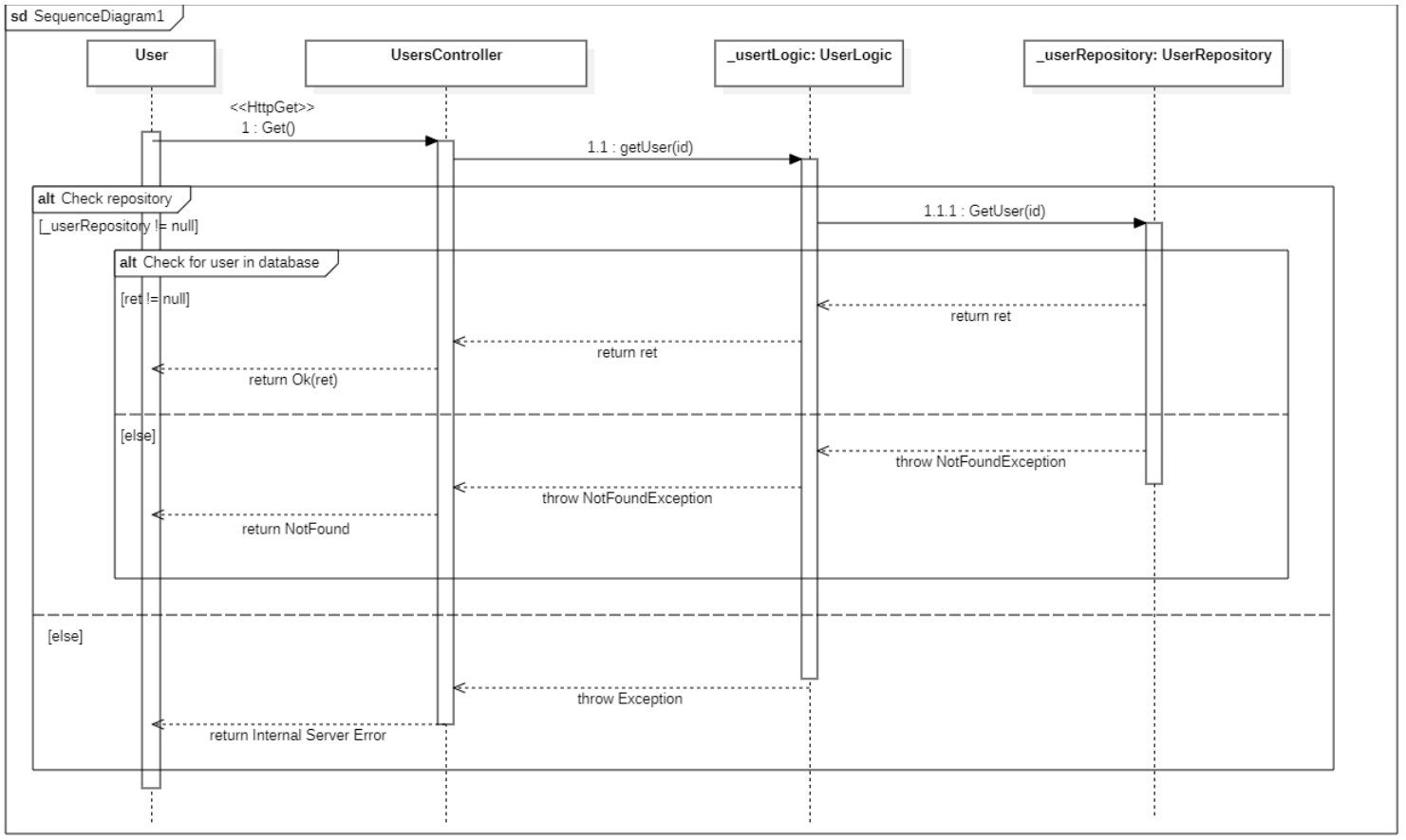
En el PromotionController, se utiliza un POST no para crear directamente una promoción, sino para enviar datos que serán utilizados posteriormente en el cálculo de la promoción.

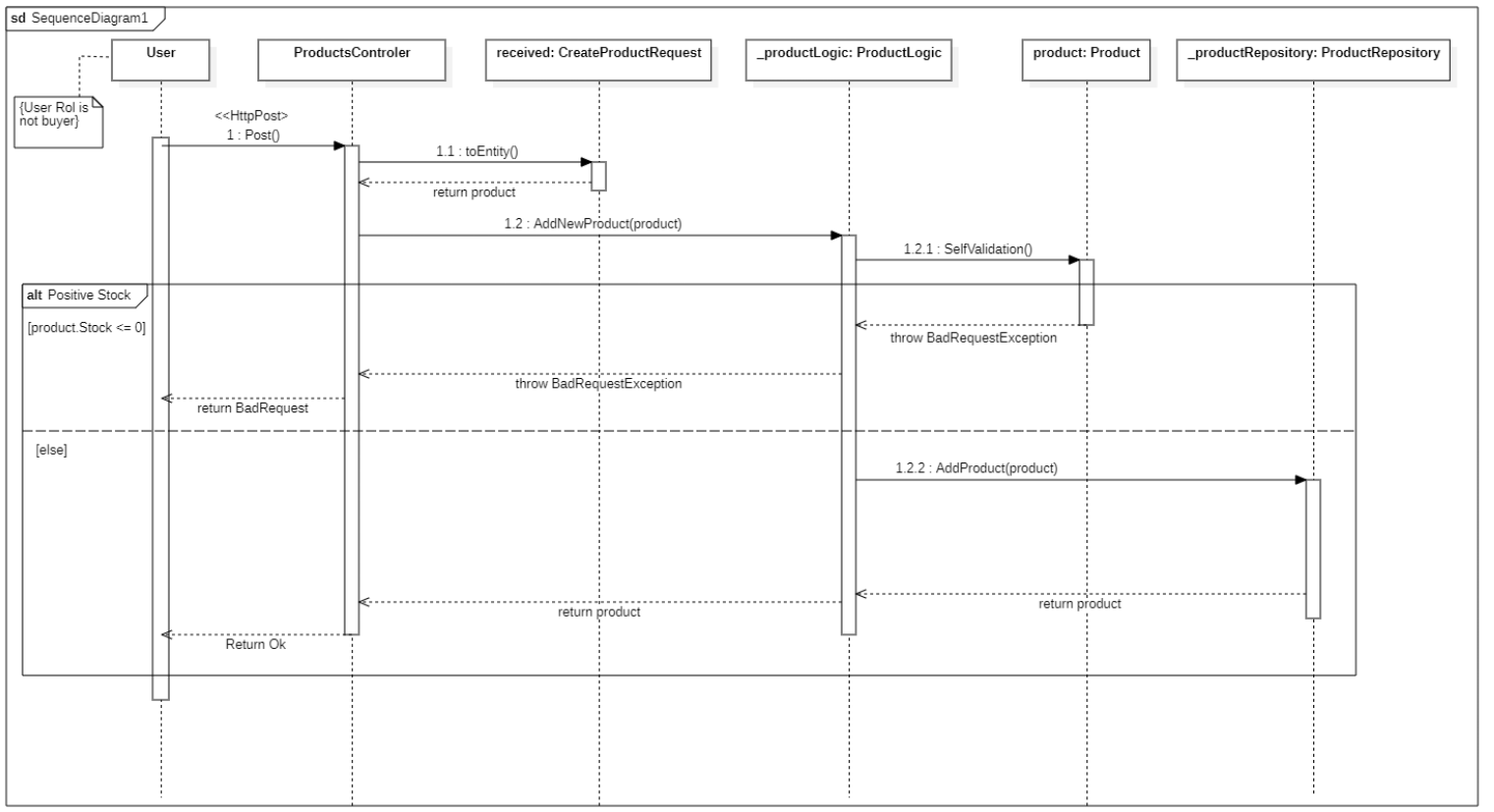
Para el PurchasesController, se manejan múltiples operaciones GET, una para revisar el historial completo de todas las compras y otra para visualizar las compras específicas realizadas por cada usuario. El POST se utiliza para la creación de la compra misma.

Por último, tanto en Users como en Products, se encuentran opciones para crear (POST), editar (PUT), borrar (DELETE) y dos operaciones GET, una para obtener todos los productos/usuarios y otra para obtener información específica sobre uno en particular (ya sea un producto o un usuario).

Vista de proceso

Para la vista de proceso se diseñaron los siguientes diagramas de secuencia:





El primer diagrama muestra el flujo de la obtención de usuarios por su id único. El diagrama de secuencia permite reflejar cómo es necesaria la existencia de la variable _userRepository en la lógica de usuario así como la obtención correcta del usuario indicado desde la base de datos para que la función pueda retornar al mismo junto con un mensaje de operación exitosa (200). También se refleja cómo se manejan las excepciones de manera individualizada, devolviendo un 404 not found en caso de que no se haya encontrado al id indicado en la base de datos o un 500 si existe una falla en el acceso al repositorio que no permita la búsqueda en un principio.

En el segundo diagrama queda reflejada la funcionalidad de incorporación de productos al sistema. Podemos ver en la restricción al User inciando la ejecución que el mismo debe ser Admin (ya sea exclusivamente, o Administrador y comprador en conjunto), dado que esta es una de las funcionalidades vinculadas a la gerencia de la empresa. El cumplimiento de este requisito se realiza antes de entrar en el get mediante un filtro de autenticación encargado de chequearlo. Una vez en ejecución, esta función post de http comienza usando la función toEntity() de la clase CreateProductRequest en Models In para, a partir de los datos ingresados por el usuario, poder obtener un objeto de tipo Product. Luego, desde product logic se accede a la función del Product de selfValidation() que corrobora que el producto ingresado cumpla la condición de poseer stock positivo. De ser así, se lo agrega en la base de datos y se retorna hasta el controlador, dónde se envía en la response con un status code 200. En el caso contrario, se lanza la excepción de bad request (status code 400), y de este modo se impide el ingreso del producto al sistema.

En ambos diagramas se puede ver la disposición descentralizada de la información en nuestro sistema, obtenida en base a la implementación satisfactoria del principio de inyección de dependencias (DIP) de SOLID.

Vista física

Esta vista no fue implementada dado que consideramos que el alcance del obligatorio no lo amerita

Patrones de diseño y principios SOLID

Uno de los principios que respetamos dentro de nuestro proyecto es el Principio de Segregación de Interfaces (ISP). Este principio establece que se deben crear interfaces más pequeñas, específicas y cohesivas que se ajusten a las necesidades del cliente.

En el obligatorio decidimos crear interfaces que permitan la comunicación entre la API y la lógica de negocio a través de las interfaces en el paquete LogicInterfaces. A su vez la comunicación entre la lógica de negocio y el acceso a la base de datos se da a través de interfaces, que se encuentran en el paquete RepostoryInterfaces. De este modo buscamos reducir el acoplamiento y aumentar la cohesión de nuestros componentes.

El Principio de Sustitución de Liskov (LSP) fue utilizado al implementar PruchaseProduct, una clase que es hija de Product y cuyo cometido es guardar los valores del los productos en el momento en que son comprados, para que luego al mostrarlos en los historiales de

compra estos no se vean afectados por los posibles cambios que se puedan hacer en el producto original. Aunque la idea principal de este principio se lleva a cabo, dado que en varias ocasiones podemos usar a products para referirnos tanto a los products como a los purchasedProducts de manera genérica, consideramos que utilizar el mismo no fue la mejor decisión de diseño para nuestro sistema en particular. El beneficio de poder almacenar todos los productos (tanto comprados como activos) en la misma tabla y poder declararlos como Products indiferentemente fue menor a las complicaciones que nos generó tener que luego diferenciar a unos de los otros. Por ejemplo, a la hora de mostrar los productos del sistema en pantalla, tenemos que distinguir y excluir a los que son de tipo PurchasedProduct.

El Open-Closed Principle indica que una entidad de software (clase, módulo, función, etc.) debe estar abierta para su extensión pero cerrada para su modificación. El mismo se cumple claramente en la implementación de las promociones con Reflection. Mediante la incorporación de esta característica de .NET, el equipo consiguió que se puedan tanto agregar como eliminar promociones existentes en el sistema, únicamente mediante el manejo de ddls que se encarguen de implementarlas. Una vez ingresado el ddl con los algoritmos que ejecutan la promoción, los mismos no pueden ser modificados.

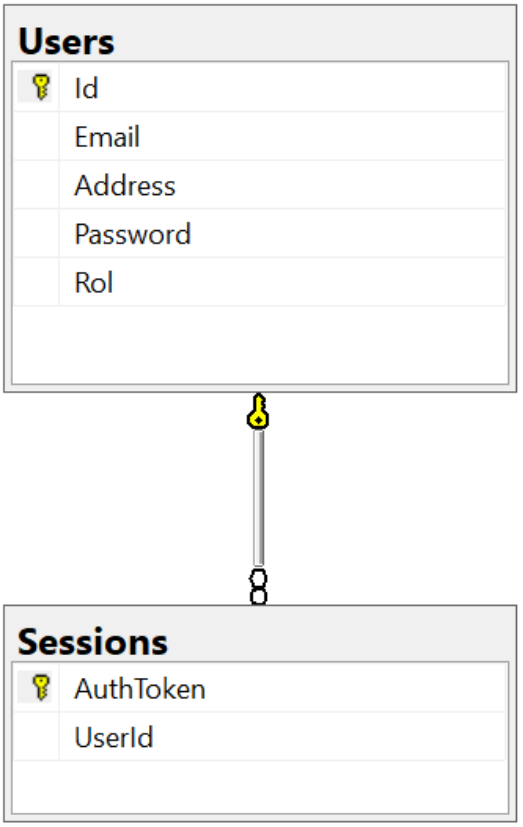
A su vez, el principio de responsabilidad única (SRP) fue utilizado para programar todas las funcionalidades en el sistema a nivel de los métodos individuales con los que se compone cada funcionalidad. Procuramos utilizar funciones cortas, encargadas de resolver un problema a la vez.En el front end este principio se ve reflejado en la naturaleza modular de Angular, ya que se hace uso de componentes individuales para cumplir funcionalidades únicas que se pueden incorporar al código varias veces.

Finalmente, como ya fue mencionado anteriormente, buscamos aplicar el Principio de Inversión de Dependencia (DIP). Este principio establece que módulos de bajo nivel no deben depender de módulos de alto nivel, sino que ambos deben depender de abstracciones. El mismo fue de utilidad tanto en el backend como en el front end, dónde lo usamos para inyectar los servicios de angular en los componentes dónde se los requería.

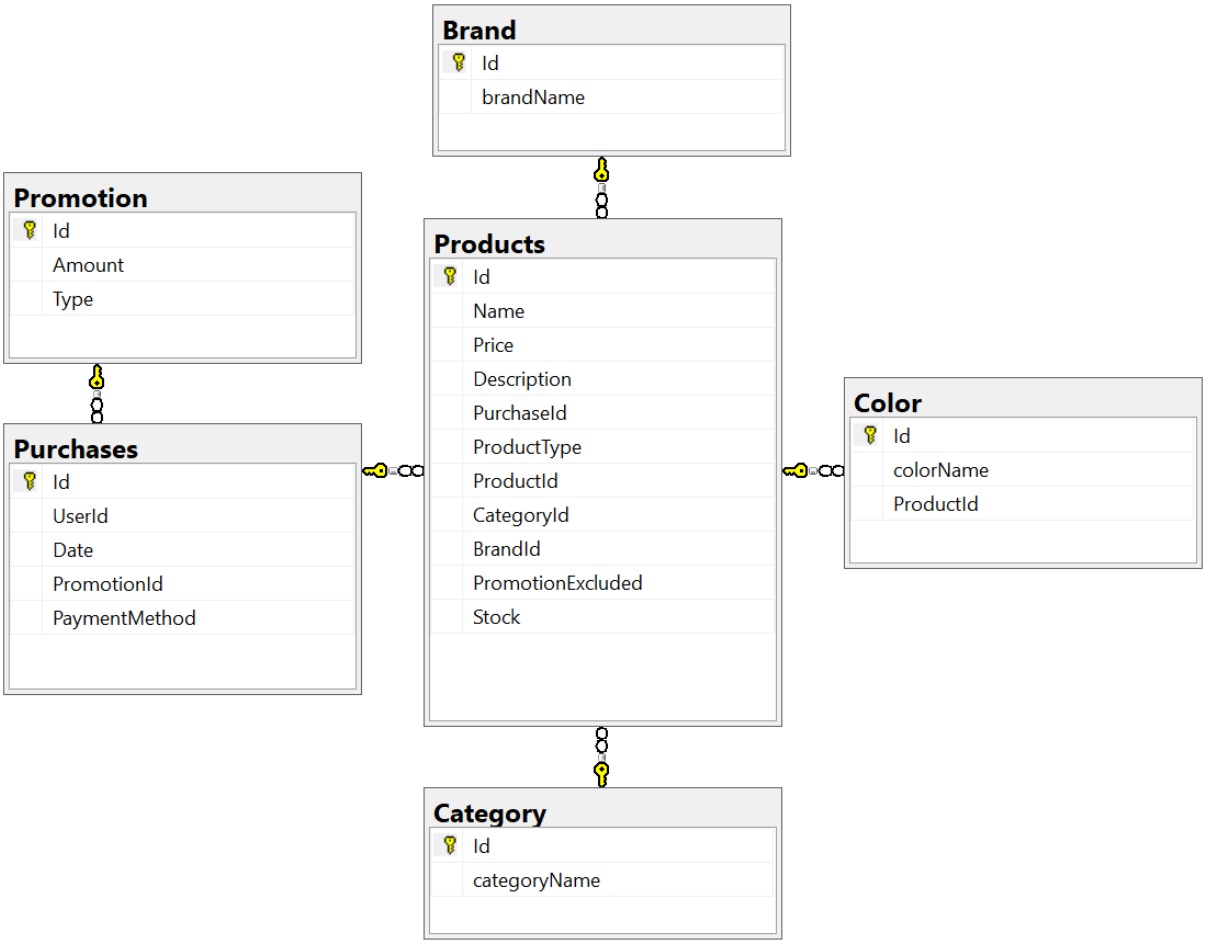
En cuanto a los patrones de diseño, se utilizó el service factory, mediante el cual también logramos implementar el patrón de inyección de dependencias. Este tiene como objetivo suministrar objetos a una clase en lugar de exigir que la clase misma los genere. Al adoptar esta perspectiva, logramos aumentar la flexibilidad y la mantenibilidad de nuestro código al separar la responsabilidad de la creación de objetos. Este enfoque ha facilitado significativamente la gestión de dependencias en nuestro sistema.

Asimismo, al asignar responsabilidades, se priorizaron los patrones de experto en información y bajo acoplamiento. La premisa fue asignar una responsabilidad a la clase que posee la información necesaria, siempre y cuando esto no resulte en un acoplamiento elevado. Un ejemplo concreto se observa en las operaciones de inserción de entidades en el contexto: los repositorios de cada entidad están familiarizados con el DbSet que el contexto (ECommerceContex) tiene para esa entidad, por lo tanto, son los encargados idóneos para llevar a cabo operaciones relacionadas con la misma.

Modelo de Tablas de la Base de Datos



En este diagrama se muestra como la Session guarda el UserId del Usuario que va a estar ingresado luego de hacer login.



En este diagrama adicional, observamos cómo los datos de "brand", "category" y "colors" ahora cuentan con sus propias tablas dedicadas para almacenar esos valores. En la tabla de "product", se guarda una única asociación a una marca ("brand") y a una categoría ("category"). En el caso de los colores, se almacena una lista de ellos en dicha tabla.

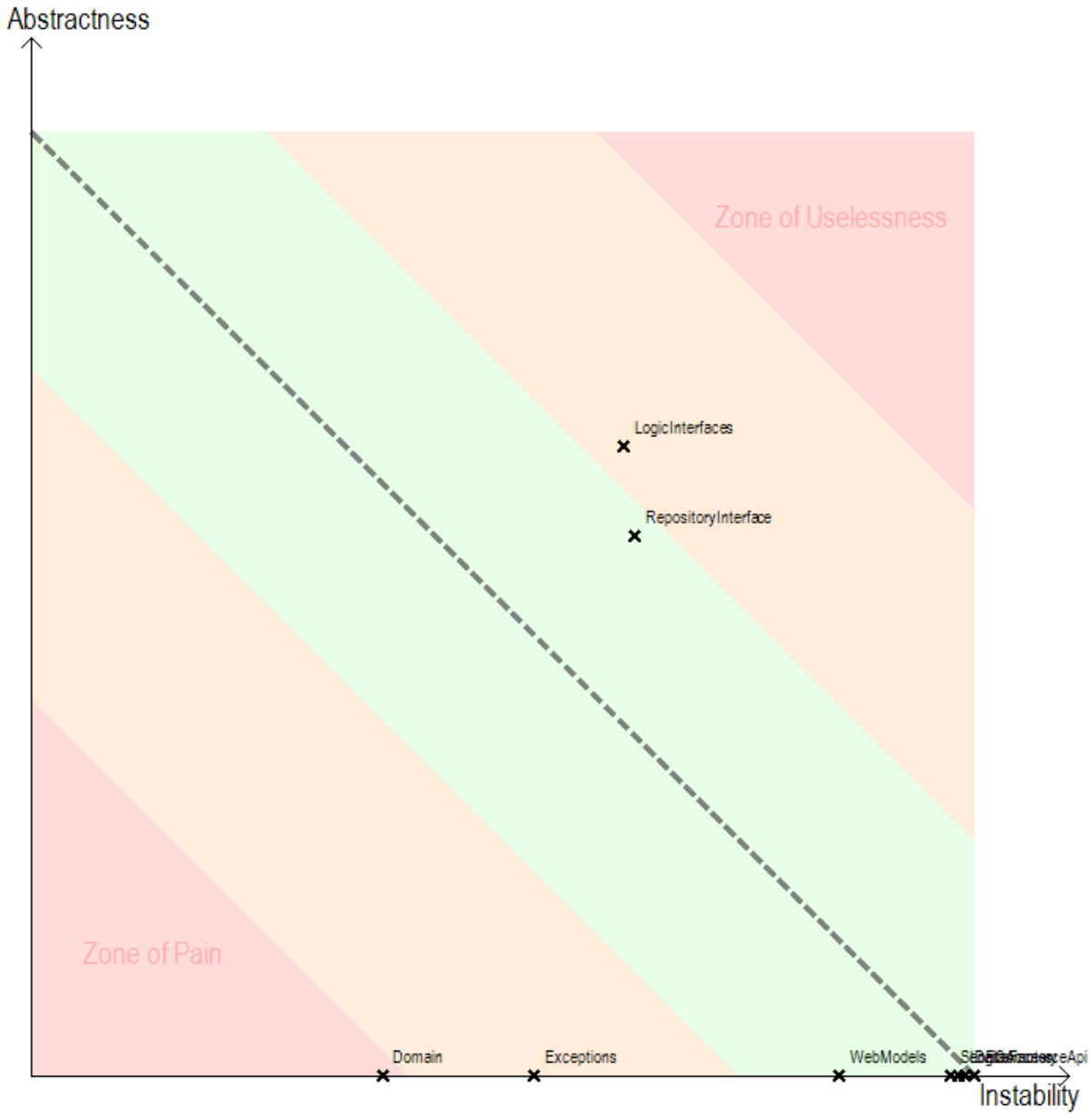
Métricas

Se utilizó la herramienta NDepend para poder analizar las métricas de diseño de nuestro proyecto.
Gracias a ese análisis se obtuvieron varios valores que nos ayudan a entender la relación entre los paquetes, entre ellos tenemos los coeficientes eferentes y aferentes (Ce y Ca respectivamente), la inestabilidad (I), la abstracción (A), la distancia (D) y también la cohesión relacional (H), las cuales representamos en la siguiente tabla

Assemblies	Ca	Ce	H	I	A	D
Exceptions	7	8	1	0.53	0	0.33
Domain	37	22	2.2	0.37	0	0.44
LogicInterfaces	13	22	1.33	0.63	0.67	0.21
RepositoryInterfaces	9	16	1.14	0.64	0.57	0.15
DataAccess	1	97	1.69	0.99	0	0.1
Logic	1	60	1.67	0.99	0	0.01
ServiceFactory	1	38	1.25	0.97	0	0.02
WebModels	5	30	1.62	0.86	0	0.1
EcommerceApi	0	110	2.25	1	0	0

Al revisar los valores dados en la cohesión racional (H), son valores relativamente bajos, lo que pudo haber sido causado por el hecho de que las relaciones entre los paquetes ocurren principalmente con interfaces y también con inyección de dependencias.
Se puede observar en base a los valores de I como estos tienden a ser medio altos en nuestros paquetes, lo cual significa que son más inestables.
Fijándonos en los datos obtenidos por A, sólo LogicInterface y RepositoryInterface son los paquetes más abstractos, mientras que el resto son totalmente concretos.
En base a estos últimos 2 valores, al analizar la distancia (D) vemos como la gran mayoría no tiene mucha distancia, y varios están cercanos a la distancia ideal para cumplir SDP (esta distancia siendo = 0). los que menos cumplen con eso son los paquetes de Domain y de Exceptions.
estos valores pueden ser observados en la siguiente tabla que fue brindada por NDepend:

Com



o ya se mencionó, mediante esta gráfica se puede apreciar como RepositoryInterface y (especialmente) LogicInterface y son los más abstractos, y se destaca como este último, Domain y Exceptions no entran en la zona verde de la Main Sequence donde se considera aceptable los valores dados. (Domain ya entrando en la Zone of Pain).
Deduciendo en base a estos datos, como no tenemos ningún paquete que esté cerca de inestabilidad 0 y abstracción 1, entonces no se cumple el caso ideal de SAP, pero si se cumple SAP ya que en su mayoría no hay paquetes que sean super inestables y abstractos y tampoco hay paquetes estables y concretos.

Mejoras de diseño

Para la implementación exitosa del frontend en esta entrega, realizamos una serie de cambios a la base implementada en la entrega anterior.

Pudimos arreglar algunos problemas que habían quedado sin resolver en la primera instancia. En primer lugar, completamos la implementación de las tablas de Brand, Category y Colors tal como fue indicada para el primer obligatorio. De esta manera también logramos que un producto pueda admitir una cantidad de colores mayor a uno. Asimismo, logramos vincular la clase Promotion a las las compras (Purchase).

Asimismo, siguiendo las correcciones obtenidas por los profesores, procuramos mejorar nuestro uso de clean code en nombres de ciertos atributos y funciones, haciendo hincapié en los nombre de las pruebas y en eliminar el uso de var como sustituto de declaración explícita de un tipo de variable. También nos aseguramos de agregar VerifyAll() en todos los mocks.

Finalmente, implementamos un endpoint nuevo llamado Promotions para que este se pueda encargar de calcular las promociones que aplican al carrito mediante un Post http cada vez que el mismo se vea modificado por el usuario comprador.

Errores Conocidos:

Se reconocen los siguientes bugs en el sistema:

- Al ingresar un producto se puede escoger si habilitan o no en el mismo las promociones. No obstante, luego de ingresar en el sistema, no se puede cambiar esta elección.
- Se escoge por defecto la promoción de 20% off (inclusive cuando no se cumplen sus prerrequisitos en el carrito).
- La promoción de 3x2 de 3 productos de misma categoría no funciona correctamente por manejo incorrecto de ids en la base de datos.
- En el Frontend, las librerías usadas del offcanvas y del modal usan como tipo de variable “any”, lo cual va en contra de los principios de clean code.
- Se tiene que usar la extensión “CORS Unblock” (https://chromewebstore.google.com/detail/cors-unblock/lfhmikememgdcahcdlaciloancbhjino?pli=1) para que funcione el deploy.
- En el deploy, no funciona el acceso al detalle de un producto en el front, aunque si lo hace al correr la aplicación localmente.
- Al agregar un producto fuera de stock al carrito, aparece un mensaje de aviso, pero el producto no se elimina automáticamente. Sin embargo, al proceder con la compra, solo se realiza con la cantidad de productos disponibles.

Anexo:

Especificación de la API con sus cambios:

- En base a lo pedido en la letra, al hacer get de products ahora se puede filtrar por precio mínimo, máximo y/o si incluye promoción o no
- Hicimos la edición de producto con más sentido para nosotras donde no se puede cambiar brand, category ni colores y se le implemento el campo de sí excluye promociones o no (el cual se puede editar también).
- Se dejó de usar error 409 en create user.

Informe de cobertura de las pruebas:

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
dataaccesstest.dll	636	13	351	9	1	97.23%
ecommerceapitest.dll	916	0	541	0	0	100.00%
domain.dll	195	21	123	2	17	86.62%
dataaccess.dll	442	2610	235	1	4810	4.66%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
{ } Exceptions.LogicExceptions	6	2	9	0	3	75.00%
BadRequestException	2	2	3	0	3	50.00%
NotFoundException	4	0	6	0	0	100.00%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
{ } Logic	396	18	343	1	19	94.49%
ProductLogic	44	0	51	0	0	100.00%
PromotionLogic	87	1	69	0	3	95.83%
PurchaseLogic	166	0	130	0	0	100.00%
SessionLogic	32	6	32	0	6	84.21%
UserLogic	51	11	46	1	10	80.70%
PromotionLogic.<>c	2	0	1	0	0	100.00%
PromotionLogic.<>c__DisplayClass5	6	0	7	0	0	100.00%
PurchaseLogic.<>c__DisplayClass9_;	8	0	7	0	0	100.00%
{ } Logic.Utils	42	3	23	0	3	88.46%
PromotionHelper	33	3	22	0	3	88.00%
PromotionHelper.<>c	9	0	1	0	0	100.00%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
{ } WebModels.Models.Out	133	11	96	7	9	85.71%
CreateSessionResponse	11	0	12	0	0	100.00%
CreateUserResponse	22	2	18	0	1	94.74%
GetProductResponse	65	2	37	7	1	82.22%
GetPromotionResponse	12	1	11	0	1	91.67%
GetPurchaseResponse	23	6	18	0	6	75.00%
{ } WebModels.Models.In	144	5	112	0	3	97.39%
CreateProductRequest	53	2	40	0	1	97.56%
CreatePurchaseRequest	19	1	14	0	1	93.33%
CreateSessionRequest	4	0	4	0	0	100.00%
CreateUserRequest	31	0	25	0	0	100.00%
UpdateProductRequest	35	2	28	0	1	96.55%
CreatePurchaseRequest.<>c	2	0	1	0	0	100.00%

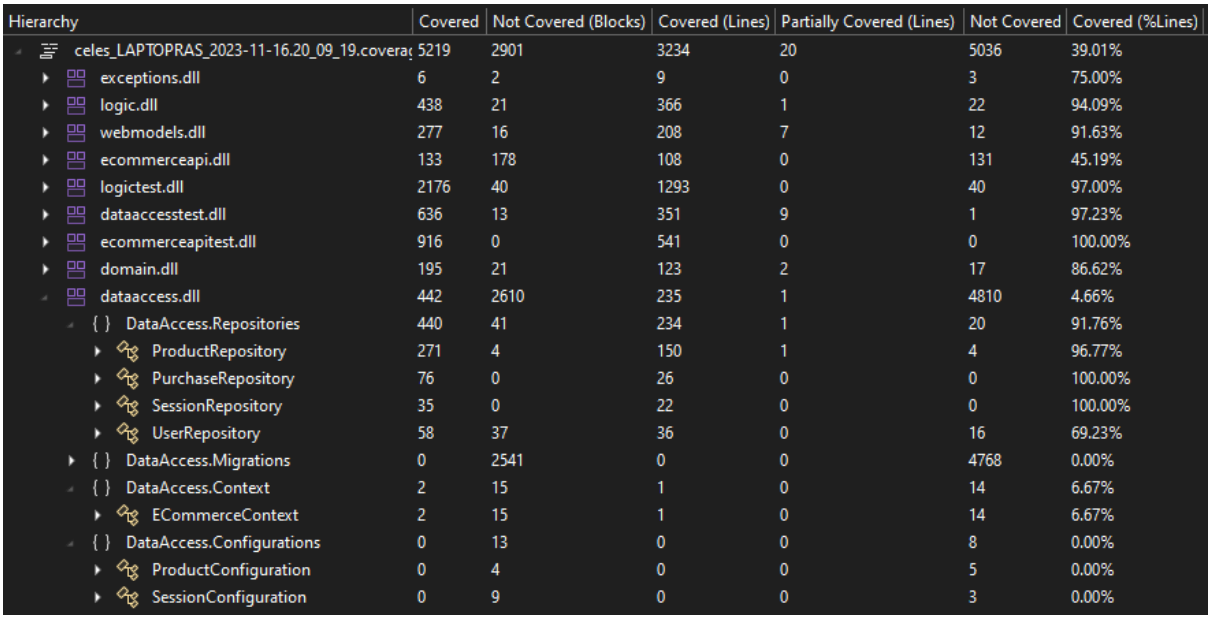
Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
{ } Global Classes	0	49	0	0	38	0.00%
Program	0	37	0	0	27	0.00%
Program.<>c	0	12	0	0	11	0.00%
{ } ECommerceApi.Filters	0	129	0	0	93	0.00%
AuthenticationFilter	0	86	0	0	63	0.00%
CustomExceptionFilter	0	43	0	0	30	0.00%
{ } ECommerceApi.Controllers	133	0	108	0	0	100.00%
ProductsController	34	0	28	0	0	100.00%
PromotionController	8	0	10	0	0	100.00%
PurchasesController	30	0	25	0	0	100.00%
SessionController	21	0	14	0	0	100.00%
UsersController	34	0	28	0	0	100.00%
ProductsController.<>c	2	0	1	0	0	100.00%
PurchasesController.<>c	2	0	1	0	0	100.00%
UsersController.<>c	2	0	1	0	0	100.00%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
{ } LogicTest	2176	40	1293	0	40	97.00%
ProductLogicTest	444	9	260	0	9	96.65%
PromotionLogicTests	409	1	186	0	1	99.47%
PurchaseLogicTest	647	5	349	0	5	98.59%
SessionLogicTest	185	5	108	0	5	95.58%
UserLogicTest	491	20	390	0	20	95.12%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.coverag	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
dataaccesstest.dll	636	13	351	9	1	97.23%
{ } DataAccessTest	636	13	351	9	1	97.23%
ProductRepositoryTest	257	7	115	7	0	94.26%
PurchaseRepositoryTest	141	1	96	1	0	98.97%
SessionRepositoryTest	83	0	41	0	0	100.00%
UserRepositoryTest	149	1	96	1	0	98.97%
ProductRepositoryTest.<>c__Display	2	4	1	0	1	50.00%
SessionRepositoryTest.<>c__Display	2	0	1	0	0	100.00%
UserRepositoryTest.<>c__DisplayCla	2	0	1	0	0	100.00%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.coverag	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
dataaccesstest.dll	636	13	351	9	1	97.23%
ecommerceapitest.dll	916	0	541	0	0	100.00%
{ } ECommerceApiTest.Controllers	916	0	541	0	0	100.00%
ProductsControllerTest	393	0	201	0	0	100.00%
PromotionControllerTest	56	0	34	0	0	100.00%
PurchasesControllerTest	196	0	140	0	0	100.00%
SessionControllerTest	84	0	43	0	0	100.00%
UsersControllerTest	183	0	121	0	0	100.00%
PurchasesControllerTest.<>c	4	0	2	0	0	100.00%

Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.coverag	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
dataaccesstest.dll	636	13	351	9	1	97.23%
ecommerceapitest.dll	916	0	541	0	0	100.00%
domain.dll	195	21	123	2	17	86.62%
{ } Domain	195	21	123	2	17	86.62%
Brand	5	2	3	0	2	60.00%
Category	5	2	3	0	2	60.00%
Color	5	2	3	0	2	60.00%
Product	70	11	35	2	8	77.78%
Promotion	7	2	6	0	2	75.00%
Purchase	16	0	14	0	0	100.00%
PurchasedProduct	20	2	13	0	1	92.86%
Session	8	0	8	0	0	100.00%
User	52	0	36	0	0	100.00%
Product.<>c__DisplayClass37_0	3	0	1	0	0	100.00%
Product.<>c__DisplayClass37_1	4	0	1	0	0	100.00%



Hierarchy	Covered	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered	Covered (%Lines)
celes_LAPTOPRAS_2023-11-16.20_09_19.covera	5219	2901	3234	20	5036	39.01%
exceptions.dll	6	2	9	0	3	75.00%
logic.dll	438	21	366	1	22	94.09%
webmodels.dll	277	16	208	7	12	91.63%
ecommerceapi.dll	133	178	108	0	131	45.19%
logictest.dll	2176	40	1293	0	40	97.00%
dataaccesstest.dll	636	13	351	9	1	97.23%
ecommerceapitest.dll	916	0	541	0	0	100.00%
domain.dll	195	21	123	2	17	86.62%
dataaccess.dll	442	2610	235	1	4810	4.66%
{ } DataAccess.Repositories	440	41	234	1	20	91.76%
ProductRepository	271	4	150	1	4	96.77%
PurchaseRepository	76	0	26	0	0	100.00%
SessionRepository	35	0	22	0	0	100.00%
UserRepository	58	37	36	0	16	69.23%
{ } DataAccess.Migrations	0	2541	0	0	4768	0.00%
{ } DataAccess.Context	2	15	1	0	14	6.67%
ECommerceContext	2	15	1	0	14	6.67%
{ } DataAccess.Configurations	0	13	0	0	8	0.00%
ProductConfiguration	0	4	0	0	5	0.00%
SessionConfiguration	0	9	0	0	3	0.00%

Al igual que en la entrega anterior se aplicó TDD siguiendo el ciclo rojo, verde refactor. Esto implica que primero se crearon las pruebas, para luego implementar el código necesario para que estas pasaran.

El análisis de cobertura de pruebas revela que, excluyendo los paquetes de Domain, migraciones, configuraciones de Data Access y las configuraciones del paquete de API, se logra un sólido porcentaje de cobertura en la mayoría de los aspectos. Aunque esto puede afectar el resultado general de la cobertura, es importante destacar que los paquetes críticos mantienen un nivel de cobertura aceptable. La exclusión de ciertos elementos puede impactar la métrica global, pero la calidad de las pruebas en las áreas esenciales permanece satisfactoria.