Universidad ORT Uruguay

Facultad de Ingeniería

Primer obligatorio

Diseño de Aplicaciones II

Entregado como requisito para la obtención del título de

Licenciatura en Sistemas

Guillermo Diotti - 285578

Juan Peyrot – 275076

Nicolás Toscano - 220264

Profesores:

Daniel Acevedo

Pablo Geymonat

https://github.com/IngSoft-DA2/285578-220264-275076

2024

Descripción general del trabajo 4
Errores conocidos 4
Funcionalidades no implementadas 5
Justificación de la estructuración del proyecto: 5
Vista de Implementación 5
Diagramas de paquetes 5
Diagrama general de paquetes 5
Diagrama por capas (layers) 7
Diagrama de componentes 7
Vista de diseño 8
Diagramas de clases 8
Diagrama general de entidades 8
Breves Aclaraciones del diagrama de clases 9
Diagrama de clases del paquete BusinessLogic para la entidad Session 10
Diagrama de clases del paquete WebApi para la entidad Session 10
Diagrama de clases del paquete Exceptions 10
Diagrama de clases del paquete DataAccess 10
Diagrama de clases del paquete Utility 10
Descripción de jerarquías de herencia utilizadas 10
Modelo de tablas de la estructura de la base de datos. 11
Justificaciones de diseño 11
Inyecciones de dependencias 11
Patrones y Principios de diseño 12
Adapter 12
Domain-Driven-Design 12
Layer SuperType Pattern 13
Separation of Concerns (SoC) 13
Principios SOLID 13
Single Responsability Principle 13
Liskov substitution 14
Interface Segregation 14
Dependency Inversion Principle 15
Decisiones de diseño 15
Ubicación de la interfaz IRepository 15
Atributos de detección de movimiento, persona y ventana 16
Atributo Type en Device 16
Out-Side In 17
Paginación 17
Uso de DTO´s 17
Implementación de clase Session y clases Filters 18
Atributos de dirección y ubicación geográfica 18
Descripción del mecanismo de acceso a datos utilizado. 19
Descripción del manejo de excepciones 19
Anexo 20

1. Estructuración previa del diagrama de clases con del patrón observer 20
2. Diagrama de clases de las entidades de dominio 21
3. Diagrama de clases del paquete de BusinessLogic 22
4. Diagrama de clases del paquete de WebApi 23
5. Diagrama de clases del paquete de Exceptions 23
6. Diagrama de clases del paquete de DataAccess 24
7. Diagrama de clases del paquete de Utility 24
8. Diagrama general de paquetes 25
9. Diagrama de paquetes por layers 25
10. Ejemplo del patrón adapter 26
11. Modelo de la base de datos 26
12. Diagrama de interacción del listado de cuentas 27
13. Foto de las dependencias usadas 28
14. Estructura de las clases de BusinessLogic 28
15. Ejemplo de clase Custom de Excepcion y ExceptionFilter 29
16. Diagrama de componentes 30

Descripción general del trabajo

Homify es una API Web **RESTful** encargada de la administración de dispositivos inteligentes en hogares, siguiendo requerimientos como la gestión de roles con distintos privilegios y la notificación de acciones captadas por los dispositivos mencionados.
Decimos que Homify cumple con el estilo arquitectónico **REST** porque presenta recursos claramente definidos, que son gestionados mediante las operaciones **CRUD** básicas **(POST, PUT, GET, DELETE)**. Estos recursos son accedidos a través de **URLs**, siguiendo buenas prácticas en su diseño. Además, cada solicitud es **independiente** (stateless) de las anteriores, lo que significa que incluye toda la información necesaria para ser procesada. Finalmente, las respuestas del servidor contienen los **códigos de estado HTTP** adecuados, indicando si la operación fue exitosa o si ocurrió algún error.

Errores conocidos

No tanto por el lado de error, pero quizás no es tan buena práctica la implementacion de tanto la clase Admin como Sensor, Ambas clases forman parte de una jerarquía de herencias. Como ambas no presentan ningún tipo de atributo distintivo de la clase base padre, no sería tan buena práctica su implementación y probablemente se podría haber evitado. Sin embargo, Implementamos ambas clases para, aunque no tener información propia respecto al padre, encapsular el tipo del objeto concreto en la clase, al igual que tener una estructura **más clara en la base de datos** y permitir extensibilidad en el futuro.

Una mala práctica que hemos implementado es la reutilización innecesaria de código, algo que debe evitarse. Si bien no ha sido algo recurrente, tenemos casos aislados donde ha ocurrido, en las clases donde se encuentran los argumentos utilizados para la creación de usuarios (*CreateUserArgs*, *CreateAdminArgs*, etc). Actualmente, hemos creado clases específicas para cada tipo de usuario (administrador, dueño de empresa, dueño de compañía), lo que ha resultado en una duplicación de código. Cada una de estas clases verifica propiedades comunes como nombre, apellido, y contraseña, cuando en realidad, una clase genérica podría haber sido utilizada para evitar esta redundancia. Esto habría permitido reutilizar el código de manera más eficiente, evitando la repetición de validaciones comunes entre los diferentes tipos de usuarios. Sin embargo, es la única excepción a la regla de no repetir código, ya que en el resto del proyecto nos hemos mantenido dentro de los estándares de las buenas prácticas.

Por último, el único “bug” del que tenemos registro ocurre en la feature de obtener las notificaciones y aplicar distintos filtros, al momento de filtrar las notificaciones de un usuario en concreto, todos los filtros funcionan correctamente, exceptuando el filtrado por

fecha, ya que en dicho caso, se deberá ingresar un string con la fecha a filtrar, pero hemos omitido la implementación de validaciones y el parseo al tipo de dato del atributo fecha de la notificación (DateTimeOffset) debido a falta de tiempo y compromisos con la fecha de entrega, lo cual solucionaremos de cara a la próxima instancia.

Funcionalidades no implementadas

Analizando exhaustivamente la letra del obligatorio, no hemos identificado ningún requerimiento o funcionalidad faltante según nuestra investigación. Por supuesto, se toman en cuenta los aspectos identificados previamente tomándolos en cuenta como áreas de mejora.

Justificación de la estructuración del proyecto:

En cuanto a la estructuración de la solución propuesta, el código y las clases implementadas están distribuidos en diferentes paquetes, siguiendo una arquitectura vertical que separa por funcionalidades en lugar de por tipos. Esto nos permite lograr una alta cohesión y un bajo acoplamiento. Los paquetes implementados son: [**BusinessLogic, WebApi, Exceptions, Tests y Utility**].

Es importante destacar que hemos decidido agrupar las entidades de dominio junto con la lógica de la aplicación dentro del mismo paquete, **Homify.BusinessLogic**, ya que ambas conforman el núcleo del negocio. En otras palabras, el dominio no puede existir sin la lógica, y la lógica carece de sentido sin el dominio.

Vista de Implementación

Diagramas de paquetes

Diagrama general de paquetes

Diagrama general de paquetes

El paquete **Homify.WebApi** es parte de la parte central de la aplicación, encargándose de la exposición de servicios a través de una API web. Dentro, se encuentran la subcarpeta **'filters'**, donde se definen clases que gestionan filtros de autenticación y autorización, para denegar el acceso indebido del usuario a métodos no correspondientes con su naturaleza. Además, contiene la subcarpeta **'Controllers'**, organizada en varias

carpetas, cada una representando un recurso específico de la WebApi. Dentro de estas subcarpetas, están las clases controladoras correspondientes a cada entidad del dominio, encargadas de gestionar las solicitudes **HTTP** y de interactuar con la lógica de negocio. Además, el paquete incluye clases como **'CreateResponse'**, **'CreateRequest'**, **'UpdateResponse'** y **'UpdateRequest'**, diseñadas para manejar las solicitudes y respuestas relacionadas con la creación, actualización y eliminación de recursos a través de la API, asegurando un flujo de datos consistente y eficiente en el contexto de la aplicación. Estos **DTO´s** ayudan también a manejar solo la información deseada y no exponer información de más.

El paquete **DataAccess** es fundamental para la gestión de la base de datos de la aplicación. Tiene un **DbContext** que permite interactuar con la base de datos. Dentro de este contexto, se definen varios **DbSet**, cada uno representa una entidad a mapear en la base de datos relacional. Además, este paquete es responsable de la generación y gestión de migraciones, lo que permite evolucionar la estructura de la base de datos de manera controlada y sin problemas a medida que cambian se implementan cambios de la aplicación. Es en el paquete DataAccess donde se almacena el **Repository**, repositorio global dela aplicación, como también repositorios custom diseñados para entidades en concreto.

Dicho paquete, **Homify.Tests**, se encarga de encapsular todas aquellas clases dedicadas a las pruebas unitarias y testeo de la lógica de la aplicación.
Para ello, se ha creado una clase de testeo, por un lado, por cada controlador implementado, como también por cada servicio lógico implementado.

Este paquete es de gran importancia ya que sus clases se encargan de alcanzar la mayor cobertura posible de sus respectivas clases, es decir, por ellas se pueden verificar los correctos funcionamientos de distintos métodos. Cabe remarcar la diferencia entre cobertura y eficacia del código. El tener, por ejemplo, una cobertura del 100% no implica el tener un código de alta calidad el cual nunca fallase, en cambio significa que las pruebas unitarias implementadas abarcan la totalidad de los métodos de la repectiva clase

Este paquete, **Homify.BuisnessLogic**, en conjunto con el paquete de **WebApi**, es de los principales paquetes implementados. En él se centran todas las operaciones relevantes relacionadas con la lógica del negocio y el manejo de datos. Tiene, para cada una de las clases de dominio queridas, una clase encargada de las validaciones y verificaciones del formato en cuanto a la creacion del objeto, siendo esta clase posteriormente implementada por el controlador. Este paquete también tiene una clase con implementación de métodos, definidos en una interfaz, encargados de actuar como intermediarios entre los métodos del controlador y los métodos de recolección de la información en la base de datos (métodos de la clase **Repository**).

Homify.Utility es un paquete que brinda ciertas herramientas. Herramientas útiles en distintas partes del código. El motivo principal de la creación de este paquete es el ahorro de código innecesario repetido, así como también evitar dependencias innecesarias entre paquetes si es que las clases auxiliares que este ostenta se encontraran en otro sitio.

Por último, hemos implementado un paquete únicamente destinado a las excepciones. Más adelante en este documento, en especial en la sección indicada, se hablará sobre ello y los beneficios que brinda.

Diagrama por capas (layers)

Diagrama de paquetes por layers

La capa de **Presentación** representaría a los usuarios que utilizarían la aplicación. En primeras instancias, para esta entrega hemos utilizado únicamente la aplicación de **Postman**, pero va a ser en esta capa donde se clasificaría el eventual uso de Angular para la interfaz de usuario

La capa de **Negocio de la Aplicación**, es la encargada de administrar y encapsular toda la lógica de la aplicación. La misma también abarca las entidades de **Dominio**.

Por último, la capa de **Persistencia**, es la encargada del manejo de datos y el coleccionamiento de os mismos. Actualmente se usa el motor de base de datos relacional **SQL** y el **framework** de **ef-core** para hacer posible la interacción con esta base de datos.

La modularización de la solución en distintas capas, presenta severos beneficios, entre los que se encuentran el cumplimiento de **SoC** (Separation of concerns), donde cada capa se encarga de una responsabilidad única. A su vez, este diseño modularizado favorece el mantenimiento y la escalabilidad a largo plazo, como la facilidad del desarrollo de pruebas unitarias

Diagrama de componentes

Diagrama de componentes

En el diagrama de componentes propuesto, se puede observar los principales componentes de la aplicación. Se utilizó un componente por paquete, remarcando la responsabilidad única de cada uno de estos componentes. A su vez, se hizo uso del estereotipo, señalizando el tipo de componentes, en este caso dll o exe. En el diagrama también se pueden observar las interfaces implementadas, indicando claramente que paquete la implementa o la requiere. Por un lado, se encuentran las interfaces **I<<Entity>>Service**, respectivamente para las entidades que presenten controladores. Estas interfaces funcionan como capa de abstracción entre la clase de comportamiento

lógico de la entidad y el controlador que solicite dichos comportamientos. Por otro lado, en el diagrama también se presenta la interfaz **IRepository**, encapsulando todas los métodos pertinentes para comunicarse con la base de datos. Esta interfaz es implementada, por un lado, por una clase genérica **Repository** la cual acepta cualquier entidad TEntity, y según corresponda, implementada por clases más concretas para sobreescribir ciertos métodos. Dato no menor, se resalta que en el diagrama de componentes se puede ver bien claro la dependencia entre los componentes.

Vista de diseño

Diagramas de clases

Diagrama general de entidades

En primeras instancias, para el manejo y gestión de las notificaciones implementamos el patrón de diseño **Observer**, el cual tiene como propósito definir un mecanismo en donde varios objetos, se encuentran "observando" a otro objeto, y cuando a este le ocurre cualquier evento, los objetos observadores son notificados al respecto.

A la hora de implementar dicho patrón, nos vimos con la necesidad de diseñar dos interfaces, una llamada "**IObserver**" y otra "**ISubject**". Por un lado la interfaz **ISubject** es utilizada por la clase **User**, ya que esta clase comprende a los miembros configurados de recibir notificaciones del respectivo dispositivo, esperando la emisión de la notificación por parte de la clase **Device**, utilizando la segunda interfaz, **ISubject**. Estas interfaces definen ciertos métodos los cuales obliga a las clases que utilicen dichas interfaces respectivas que los implementen. El principal método, y el más importante de ISubject es el **Notify()**. Este método es el encargado de llamar al método **Update()** de **IObserver** por cada observador existente. Contamos con la siguiente abstracción del diagrama UML de como había sido implementado el patrón **Observer**

Estructuración previa del diagrama de clases con el patrón observer

Avanzando con la implementación en código del modelo proyectado, notamos que el patrón no estaría cumpliendo con la función deseada. Habíamos definido la lista de observadores dentro de la clase **Device**, pero surgía el problema de cómo identificar aquellos observadores del hogar concreto donde pertenece el dispositivo que genera la notificación. En base a esto pensamos entonces mover esta lista a clase **Home**, pero pasaría lo mismo de forma contraria. Como resolución a estos problemas, decidimos omitir la implementación de dichas interfaces, y la implementación del patrón **Observer** en sí, recolocando la lista de usuarios a notificar a la clase **Home** y cambiando el tipo de objeto de

dicha lista. Agregamos también una flag booleana a **HomeUser** para poder identificar si un miembro de hogar en particular debe recibir o no notificaciones.
Diagrama de clases de las entidades de dominio

Breves Aclaraciones del diagrama de clases

User - Role - SystemPermission

La primera de estas clases representa las cuentas del sistema. La misma puede presentar 3 tipos (hasta el momento) de roles distintos, siendo estos administrador, dueño de empresa o dueño de hogar. Como cada rol tiene una restricción de las acciones permitidas a realizar, se implementó la clase **SystemPermission**, adjudicándole cada permiso en particular a cada rol

Tablas intermedias

A lo largo de este obligatorio se crearon tablas interobjetos, tablas intermedias, que si bien no representan entidades en sí, cumplen el proposito de soporte de las relaciones entre ellas. Algunas de estas clases son **HomeUser**, **HomeDevice**, **HomeUserHomePermission**, **RoleSystemPermissions.** La primera de ellas, HomeUser, representa a los usuarios pertenecientes a un hogar dado. Se compone del id propio, como también id de las entidades Home y User como bien el nombre lo menciona. Como cada miembro de un hogar presenta distintas acciones permitidas dentro, la clase tiene una lista de HomePermission. Como último detalle de esa clase, hay un flag booleano para saber si dicho miembro recibe o no notificaciones. El resto de las mencionadas sigue una estructura similar, con un identificador, identificador de las clases que relaciona y algún atributo necesario.

User - HomeUser - HomePermission

Los miembros de un mismo hogar pueden realizar distintas acciones como bien lo indique el dueño de la misma instalación, por tanto, se implementó la clase **HomePermission** siguiendo la misma filosofía que **SystemPermission**. Por más de poseer un rol en el sistema con ciertos permisos, dentro de un hogar, los permisos difieren entre sus miembros.

Session

La clase **Session** sirve para otorgarle al usuario un token al momento de registrarse en la plataforma. De esta forma, puede ser posible identificar que usuario se encuentra con la sesión iniciada en cada momento.

PermissionGenerator - RolesGenerator

Para facilitar la creación de rol, y en concreto, la asignación de permisos a cada rol, hemos implementado estas dos clases. Por un lado, PermissionGenerator es una clase que cuenta con un atributo de tipo string por cada rol existente en el sistema. La clase a su vez

cuenta con métodos que retornan una lista de permisos agrupados por cada rol. En cuanto a la clase RolesGenerator, esta es la encargada de crear los roles en concreto, asignando a cada rol una clave e instanciando a la lista de permisos retornada por los métodos pertinentes de la clase PermissionGenerator

Diagrama de clases del paquete BusinessLogic para la entidad Session

Para el siguiente diagrama se tomó como ejemplo la entidad Session, para no generar diagramas repetitivos y tratar de ilustrar la ide lo más sencillamente posible
Diagrama de paquetes de BusinessLogic

Diagrama de clases del paquete WebApi para la entidad Session

Para el siguiente diagrama se tomó como ejemplo la entidad Session, para no generar diagramas repetitivos y tratar de ilustrar la ide lo más sencillamente posible
Diagrama de paquetes de WebApi

Diagrama de clases del paquete Exceptions

Diagrama de paquetes de Exceptions

Diagrama de clases del paquete DataAccess

Diagrama de paquetes de DataAccess

Diagrama de clases del paquete Utility

Diagrama de paquetes de Utility

Descripción de jerarquías de herencia utilizadas

En el proyecto **Homify**, se utiliza una jerarquía de herencia en el manejo de dispositivos. La clase **Device** agrupa propiedades y comportamientos comunes a todos los dispositivos inteligentes del sistema, como **Id, Name, Model**, entre otros. De esta clase los dos tipos de dispositivos existentes al momento, **Camera** y **Sensor**, que añaden características específicas de cada tipo de dispositivo.

Este enfoque ayuda en la reutilización de código innecesario, siendo este comprendido en **Device**. Además, se emplea polimorfismo para tratar instancias de **Device** como **Camara** y **Sensor** respectivamente.

Esta jerarquía de herencias mejora la mantenibilidad y escalabilidad del código, ya que agregar nuevos dispositivos solo requiere crear nuevas clases derivadas que extiendan de **Device**, sin necesidad de modificar el comportamiento compartido.

Consiguientemente, la herencia utilizada en **User** cumpliría la misma funcionalidad. Esta interfaz encapsula los atributos comunes entre los distintos roles existentes. Cada una de estas clases hijas sobreescribe el valor del atributo rol con el correspondiente. Al igual que en la interfaz anterior, a la hora de ampliar el sistema incluyendo más roles, bastaría con crear una clase heredada de User, obviamente a su vex luego configurando los permisos dados de este rol nuevo.

Modelo de tablas de la estructura de la base de datos.

Modelo de la base de datos

Diagramas de interacción

Diagrama de interacción del listado de cuentas

En el presente diagrama de puede apreciar la interacción de mensajes desde el proceso de un usuario de realizar la petición de listar las cuentas, hasta el listado de las mismas. Durante esta interacción, se comprueba que el usuario contenga los permisos adecuados para efectuar tal petición

Justificaciones de diseño

Inyecciones de dependencias

Foto de las inyecciones usadas

El método AddScoped registra cada servicio y repositorio con una vida útil de **scoped** (per-request), lo cual significa que se crea una nueva instancia de estas clases para cada solicitud HTTP. Esto es importante para una API REST porque garantiza que no haya estado compartido entre solicitudes concurrentes, lo que mejora la seguridad y fiabilidad, además de ser un requerimiento clave para cumplir con la arquitectura REST.

Usando **Scoped**, se declara que cada vez que el controlador o cualquier otra clase necesite un IRepository<User> o IUserService, se inyecte una nueva instancia respectivamente.

Además, las diferentes entidades tienen su propio servicio y repositorio, lo que permite una segregación clara de responsabilidades:

El **repositorio** se encarga de las operaciones CRUD con la base de datos.

El **servicio** encapsula la lógica de negocio específica que puede implicar más que simples operaciones CRUD.

Esta separación es importante porque permite modularidad, podemos garantizar que cada clase tiene una única responsabilidad, además de no estar acoplados a implementaciones independientes de bases de datos por ejemplo. De esta manera, logramos un diseño sólido a partir de respetar **SRP** y **DIP**.

Patrones y Principios de diseño

Adapter

El patrón adapter tiene como objetivo el limitar el impacto del cambio generado por una incompatibilidad. Para ser de mayor agrado la explicación, se cuenta con un diagrama (Ejemplo del patrón adapter). En la fotografía, se puede ver concretamente a las clases **User, IRepository<T>, Repository<T> y HomifyDbContext**. Supongamos que se desea listar los usuarios. No se quiere exponer a todos los servicios la inyección a **HomifyDbContext**. Lo que se va a poner en el adaptee es un método (**public List<User> GetAll()**) y va a ser en esa función donde se invoque al **dbcontext**. Todos los repositorios adaptarán el contexto para limitar el impacto de cambio, ya que en el día de mañana, si se desea usar una base de datos no relacional como **Mongo**, la capa de aplicación no se va a neterar de aquel cambio, porque nunca va a tener la dependencia hacia **HomifyDbContext**. La incompatibilidad en sí son los métodos **LINQ**. La interfaz **IRepository** no depende de la tecnología. Si se llega a realizar el cambio a **Mongo**, se creará un adapter para ello, que implemente de la misma interfaz, creando también un nuevo adaptee.

Domain-Driven-Design

Este es un conjunto de patrones y principios que proporcionan una forma de diseñar en torno al dominio. Se podría mencionar que esta metodología se aplicaría a nuestra solución, ya que en los comienzos del diseño de la solución requerida, nos hemos concentrado en la contrucción de diagramas de las entidades de dominio identificadas, posteriormente codificando en torno a lo modelado. Por supuesto, en reiteradas ocaciones de debió rediseñar lo diagramado para considerar una mejor solución y para resolver inconvenientes surgidos.

Layer SuperType Pattern

Layer SuperType Pattern es un patrón de diseño que involucra la creación de una superclase para todos los tipos de una layer específica. Este patrón se lo vería aplicado en la implementación de la interfaz llamada **IRepository**. Con la dicha interfaz, se logra generar un tipo genérico de operaciones (**Add**, **Get**, **Delete**, entre otras), siendo estas utilizadas por el negocio de la aplicación, para interactuar con la base de datos. Con esto se logra que las clases de la lógica no se preocupen de los detalles de las operaciones definidas en esta interfaz.

Separation of Concerns (SoC)

Aplicado a una web API promueve la creación de distintos módulos o capas que separan las responsabilidades de la aplicación. En el caso de *Homify*, este principio se implementa mediante la organización de la solución en diferentes paquetes que cumplen funciones específicas. Estos distintos paquetes son **WebApi**, **BusinessLogic**, **DataAccess**, **Exceptions**, **Tests** y **Utility**. De esta forma, cada módulo está desacoplado de los demás, permitiendo un desarrollo más flexible, mantenible y escalable.

Principios SOLID

Single Responsability Principle

El **Single Responsibility Principle (SRP)** establece que una clase debe tener una única razón para cambiar, es decir, debe encargarse de una única responsabilidad dentro del sistema. Este principio está presente en todo el proyecto, donde se han creado **clases controladoras específicas** para cada entidad que maneja solicitudes, garantizando que cada controlador se encargue de su propia funcionalidad. Asimismo, existen **clases dedicadas a la lógica de negocio** para cada uno de los objetos, asegurando una clara separación de responsabilidades y evitando que una clase asuma tareas que no le corresponden. Entre los beneficios se encuentran la mantenibilidad y un código más claro y legible, ventaja súmamamente vital al momento de trabajar en equipo.

Pongamos como ejemplo las clases pertinentes a el paquete **BusinessLogic(Estructura de clases de BusinessLogic)**. En ella, se aprecian subcarpetas, dividiendo así cada objeto existente. Es en cada de ellas en dónde se contiene la lógica implementada, en lugar de contener cada una de ellas de forma mezclada y entrelazada.

Open Closed Principle

El **Open/Closed Principle (OCP)** establece que las entidades de software deben estar cerradas para modificaciones pero abiertas para extensiones. Este principio permite que el código existente se mantenga intacto mientras se amplían sus funcionalidades.

Una de las principales herramientas para lograr esto es el uso de **interfaces**. Por ejemplo, en nuestro proyecto hemos definido la interfaz IRepository, que establece los métodos CRUD básicos para la gestión de datos. Imaginemos un escenario en el que deseamos integrar un motor de base de datos no relacional, como **MongoDB**. En lugar de modificar el código existente que interactúa con el repositorio, simplemente podemos crear una nueva clase que implemente IRepository para MongoDB. Esto permite que el sistema maneje el nuevo motor sin alterar el código ya implementado.

Otro ejemplo de la aplicación del OCP es la adición de nuevos roles o tipos de dispositivos en el sistema. Supongamos que queremos incluir un nuevo rol. En este caso, se crearía una nueva clase que herede de la clase base User, y se agregarían los permisos correspondientes para este nuevo rol. Este enfoque garantiza que al introducir un nuevo rol, se está extendiendo el comportamiento del sistema sin modificar la implementación existente.

Liskov substitution

En el diseño del sistema, el principio de sustitución de Liskov (LSP) se manifiesta claramente en la relación entre las clases User y Admin. La clase Admin hereda de User, lo que permite que un objeto de tipo Admin sea tratado como un objeto de tipo User, sin alterar el comportamiento esperado del sistema. Este comportamiento se refleja en el método AddAdmin de la clase UserService, que utiliza un objeto Admin para añadirlo al repositorio. De esta manera, se garantiza que todas las propiedades y métodos definidos en User están disponibles en Admin, lo que permite que cualquier funcionalidad diseñada para operar con User continúe funcionando de manera predecible y consistente.

Además, esta implementación favorece la extensión del sistema, ya que permite la creación de futuros tipos de usuarios, siguiendo la misma estructura sin comprometer la integridad del código existente. Por lo tanto, al adherirse al principio de sustitución de Liskov, el diseño del sistema se vuelve más robusto, mantenible y escalable.

Interface Segregation

El principio de **Interfaz Segregación** establece que una clase no debería verse obligada a implementar comportamientos que no necesita. En lugar de tener una única interfaz masiva que agrupe todos los métodos necesarios, se crearon interfaces específicas para cada tipo de objeto. Una clara implementación de este principio surge con las interfaces de las distintas clases de lógica de negocio, en lugar de definir una única interfaz

“IService” que sea implementada por todas las clases de lógica, definimos una interfaz específica para cada una de estas clases (IUserService, IHomeService, IDeviceService, etc), principalmente por el motivo que no todas las clases van a usar la misma cantidad de métodos, ni la misma lógica en ellos, por lo que crear solo una abstracción ataría a todas las clases a implementar métodos que no necesitan.
De esta manera, se evita que las clases implementen funcionalidades no deseadas, lo que resulta en un diseño más limpio y modular. Al aplicar este principio, cada clase solo implementa los métodos relevantes para su contexto, facilitando el mantenimiento y la extensión del código en el futuro.

Dependency Inversion Principle

Paquetes de alto nivel no deben depender de paquetes de bajo nivel. Se entiende de alto nivel, aquellos paquetes que no dependen de la tecnología utilizada, como **BusinessLogic.** Por contraparte, los paquetes de bajo nivel son los que dependen cercanamente de la base de datos, dependen de la tecnología. En el diagrama de paquetes, se nota que el paquete de alto nivel BusinessLogic no depende ni de WebApi ni de DataAccess, ambos paquetes de bajo nivel.

No solo a nivel de paquetes, sino también logramos implementar este principio con clases, siendo el ejemplo más claro los controladores, servicios y repositorios.

El **Controlador** es el responsable de manejar las peticiones HTTP entrantes. En lugar de depender directamente de una implementación concreta del servicio (por ejemplo, UserService), el controlador depende de una **abstracción**: IUserService.

El **Servicio** es responsable de la lógica de negocio de la aplicación. Este servicio no interactúa directamente con la base de datos ni conoce detalles de implementación específicos. En lugar de eso, el servicio depende de una **abstracción** del repositorio (IRepository<User>), lo que no lo acopla a un modelo de base de datos específico, sino que simplemente sabe *qué* operaciones puede realizar, sin depender del *cómo* las hace.

El **Repositorio** es la capa de bajo nivel que interactúa directamente con la base de datos (por ejemplo, utilizando Entity Framework). Como es un detalle de bajo nivel, cumple su función proporcionando los datos al servicio, pero la **abstracción** (IRepository<User>) que implementa asegura que no se impongan detalles técnicos en módulos de alto nivel.

Decisiones de diseño

Ubicación de la interfaz IRepository

En el transcurso de desarrollar las clases necesarias para el manejo y comunicación con la base de datos, en primeras instancias, habíamos colocado la interfaz **IRepository**

dentro del paquete **DataAccess**, particularmente dentro de la subcarpeta **Repositories** dentro de ella. Fue eventualmente, cuando descubrimos que esa ubicación de ese archivo traía problemas. Colocando **IRepository** en **DataAccess**, generaba una dependencia circular. Como resultado final, colocamos tal archivo en el paquete **BusinessLogic**

Atributos de detección de movimiento, persona y ventana

Una de las primordiales funcionalidades de las camaras es la habilidad de captar movimiento y personas. Por contraparte, los sensores tienen la funcionalidad de detección de ventana. Por efecto a esto, se crearon flags booleanas en la clase Device, no representando si se captó o no movimiento/persona, sino para representar la capacidad del dispositivo de presentar las funcionalidades. Es entonces que las clases hijas, Sensor y Camera, en el constructor le asignan los valores a estas properties. La camara setearía los valores de MovementDetection y PersonDetection a true, y el de ventana a false, y la clase sensor haría lo mismo inversamente.

Atributo Type en Device

Una de las tantas funcionalidades de los dispositivos es el listado de los mismos, listado donde se pueda filtrarlos por tipo. Es entonces que consideramos pertinente la creación de un atributo **Type** en **Device**, que funcionaría por así decirlo como **Discriminator**. Cuando se cree un nuevo dispositivo, por supuesto además de almacenar su valor en la tabla **Device**, se pondrán sus valores en **Sensors** y **Cameras** según corresponda, tablas independientes. Pero llegado al caso de filtrar, de antemano no se sabría que tipo de dispositivo se está tratando, y por ende, se desconocería si buscar en **Sensors** o **Cameras**. Se podría preguntar por el tipo de el objeto para así identificarlo, pero esa es una práctica algo “sucia”. Es allí donde juega rol el atributo **Type**, al momento de crear una **Camera**, Type tomará el valor de **“Camera**”, y evidentemente ocurrirá algo propio para **Sensor**. Allí el filtrado por tipo de dispositivo sería mucho más manejable. Es oportuno aclarar que para saber si un **Device** es una cámara o un sensor, no se usan “magic strings” al momento, sino que se comparan valores expuestos en una clase estática perteneciente al paquete Utils, donde se almacenan varios valores constantes, con fin de hacer de esto una buena práctica.

Out-Side In

El enfoque **Outside-In** se centra en construir software a partir de las necesidades del negocio hacia los detalles técnicos, lo que se alinea muy bien con la estructura de una **API REST .NET** separada en capas como **Business Logic**, **Data Access** y Web **API**. Este enfoque facilita la separación de responsabilidades al asegurarse de que estas capas estén alineadas con el dominio y los casos de uso del sistema, lo que resulta en un diseño modular y fácil de mantener. Con **Outside-In**, el diseño comienza desde los casos de uso y las necesidades del usuario final, enfocándose primero en lo que necesita el negocio y en las acciones que los usuarios desean, antes de adentrarse en la implementación técnica. Esto ayuda a evitar acoplamientos innecesarios y mantiene la lógica del negocio más independiente del resto de la aplicación.

Además, el enfoque **Outside-In** promueve un diseño altamente testable debido a la clara separación de la lógica de negocio, lo que facilita la escritura de pruebas unitarias. También permite la evolución del sistema sin mayores complicaciones, ya que los cambios en la lógica de negocio no afectan las capas inferiores. Por ejemplo, si las reglas cambian, las **API** y las interacciones con la base de datos pueden mantenerse sin alteraciones. Asimismo, la sustitución de tecnologías subyacentes, como cambiar de un proveedor de base de datos **SQL** a **NoSQL**, se puede realizar sin necesidad de modificar la lógica de negocio ni la **API**, ya que estos detalles están abstraídos en la capa de acceso a datos.

Paginación

La paginación fue implementada utilizando dos parámetros en la query: **offset** y **limit**. El parámetro offset indica cuántos registros se deben omitir antes de comenzar a devolver los resultados, mientras que limit especifica el número máximo de registros que se devolverán en una sola respuesta. La lógica valida que los valores proporcionados sean correctos, y en caso contrario, utiliza valores predeterminados de 0 para offset y 10 para limit. Esta solución nos proporciona control directo sobre los resultados, permitiendo consultas dinámicas y navegación eficiente por los datos, un enfoque que muchas **API REST** utilizan para manejar grandes volúmenes de información, como es el caso de **PokeAPI** (https://pokeapi.co/).

Uso de DTO´s

El uso de DTOs (Data Transfer Objects) en el proyecto permite transferir datos de manera controlada entre las capas de la aplicación y el cliente. Como se mencionó anteriormente, los DTOs nos brindan un mayor control sobre la información que se envía al cliente, garantizando que solo los datos relevantes sean expuestos, evitando exponer

información sensible o innecesaria. Esto también mejora la seguridad y eficiencia de las operaciones, ya que se limitan los datos transmitidos en cada solicitud.

Implementación de clase Session y clases Filters

Las clases **Session** y **Filters** son fundamentales para gestionar la autenticación y autorización de los usuarios en la aplicación. La **clase Session** se encarga de manejar las sesiones de los usuarios, asignándoles un token que permite al servidor identificarlos en cada solicitud. Este token es clave para verificar la identidad del usuario y definir qué acciones puede realizar dentro del sistema, lo que asegura que cada usuario interactúe de manera segura con los recursos.

Junto a la clase **Session**, también implementamos varias clases **Filters** que validan el estado de autenticación y los permisos antes de que el usuario pueda acceder a determinados endpoints. Entre ellas, destacan **AuthenticatedFilter**, **AuthorizationFilter** y **NonAuthenticationFilter**, cada una encargada de un aspecto específico de la seguridad en la aplicación. Estas clases aseguran que solo los usuarios autorizados puedan interactuar con ciertas partes del sistema.

La **NonAuthenticationFilter**, por ejemplo, se encarga de verificar si en la solicitud viene el encabezado de autorización. Si es así, extrae el token y verifica si es válido. Si el token es válido (es decir, corresponde a alguna sesión de usuario), el filtro bloquea la solicitud y devuelve un mensaje de "Unauthorized" con un código 401. Este filtro es útil para rutas a las que deben acceder usuarios no autenticados, garantizando que no se permita el acceso a aquellos ya autenticados.

En cuanto a la **AuthorizationFilter**, esta añade una capa adicional de verificación, comprobando que el usuario autenticado tiene los permisos necesarios para acceder a un recurso. Utiliza un código de permiso (Code) para verificar si el rol del usuario tiene los privilegios adecuados. Si no los tiene, la solicitud se bloquea con un mensaje de "Unauthorized" y un código 403, lo que indica que, aunque autenticado, no tiene permiso para realizar esa acción.

Estas clases ayudan a mantener la seguridad de la aplicación de forma modular y permiten gestionar la autenticación y autorización de manera clara y eficiente, lo que facilita la escalabilidad y el mantenimiento del sistema.

Atributos de dirección y ubicación geográfica

Uno de los tantos campos a capturar en la creación de los hogares son su dirección y ubicación geográfica, Por letra, la dirección comprendería la calle principal y el número de puerta, y la ubicación geográfica, latitud y longitud. Nuestro diseño se llevó de tal forma de

que los hogares presentaran estos atributos, pero de forma independiente, es decír, un campo única para cada valor de ellos.

¿Qué es un usuario autenticado?

Para que un usuario esté autenticado, necesita cumplir con dos requerimientos, siendo el primero de ellos tener una cuenta propia en el sistema, y el segundo, hacer login en ella al menos una vez. La razón detrás de esto es que al hacer login se genera una Session para ese usuario en concreto, la cual contiene un AuthToken autogenerado con Guid, el cual se debe especificar en los headers con la key Authorization para poder tener acceso sobre las operaciones que permite la WebApi. Con el objetivo de simplificar estos detalles, estas AuthToken no tienen fecha de vencimiento, ni se vuelven a generar, solo se crean la primera vez que el usuario accede con su cuenta al endpoint "*/sessions*", en caso de acceder más de una vez, siempre se le devolverá el mismo token para que pueda seguir usándolo.

Descripción del mecanismo de acceso a datos utilizado.

Para el acceso a datos, se ha utilizado el framework **Entity Framework Core (EF Core)**, que permite la comunicación con la base de datos relacional SQL. EF Core soporta **migraciones**, lo que facilita la actualización estructurada de las tablas en la base de datos. Existen dos enfoques principales para la carga de datos relacionados: **eager loading** (carga anticipada) y **lazy loading** (carga diferida). En nuestro proyecto, aplicamos **eager loading**, utilizando el método .Include() en los repositorios para traer datos relacionados en una sola consulta LINQ. El contexto de base de datos, representado por la clase **HomifyDbContext**, gestiona la interacción entre las entidades y la base de datos, permitiendo realizar consultas, insertar, actualizar y eliminar registros. Además, se cuenta con datos semilla precargados para inicializar la base de datos.

Descripción del manejo de excepciones

A lo largo de la matoría de las clases, se implementaron bloques try catch para evitar comportamientos surgidos no deseados. Es en la sección de catch dónde llanzamos excepciones en caso de que se llegue hasta allí. Podríamos haber simplemente lanzado excepciones genéricas, cada una con un mensaje descriptivo, por supuesto, pero no hicimos eso. Fuimos más allá. Desarrollamos clases específicas para cada tipo de excepción que se nos ocurriece que se pudiera lanzar, de esa forma encapsulamos los tipos de error. Asímismo, también implementamos un filtro de excepciones, ocupado de atrapar las excepciones lanzadas por el sistema para transformarlas en mensajes y códigos de status HTTP más ilustrativos y convenientes.

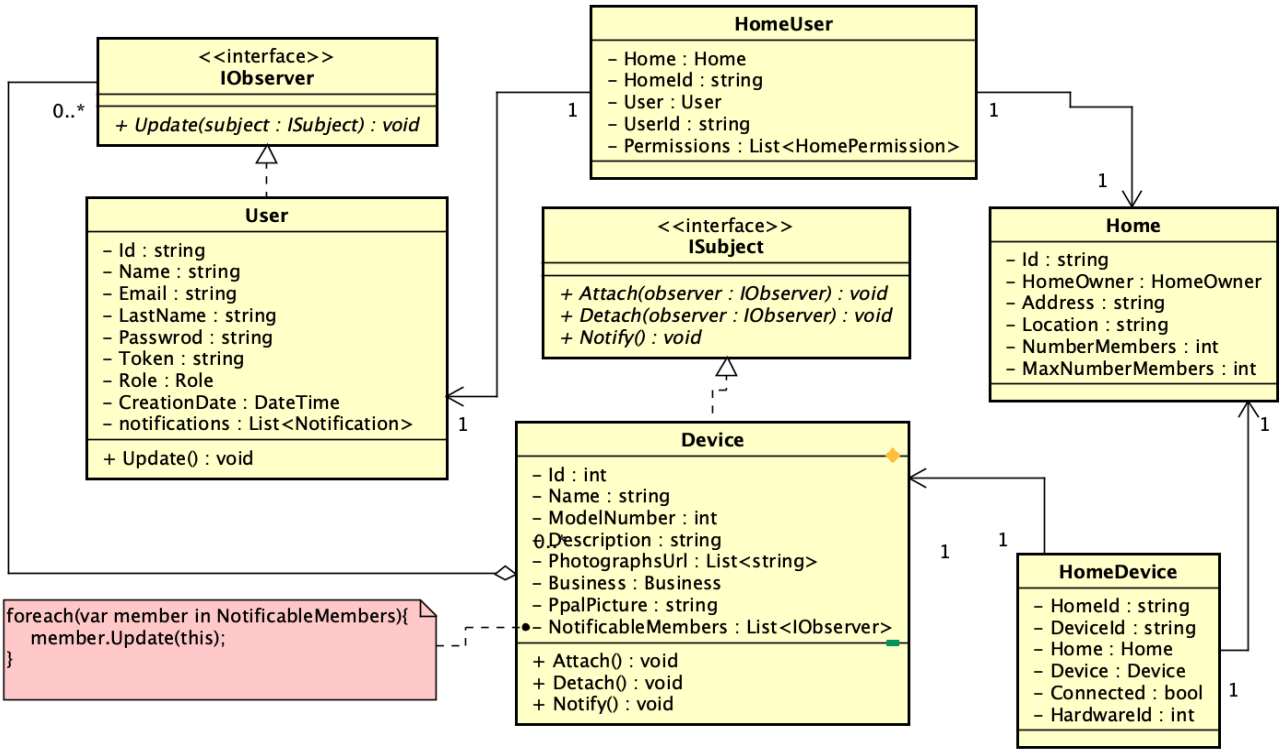
Desde la perspectiva de desarrolladores, esto favorece a que, en el momento de estar ejecutando la aplicación, en el caso de ocurrir alguno de estos casos, se agilizaría mucho más el tiempo de detección del error.

A continuación se puede observar un ejemplo de estas clases previamente nombradas

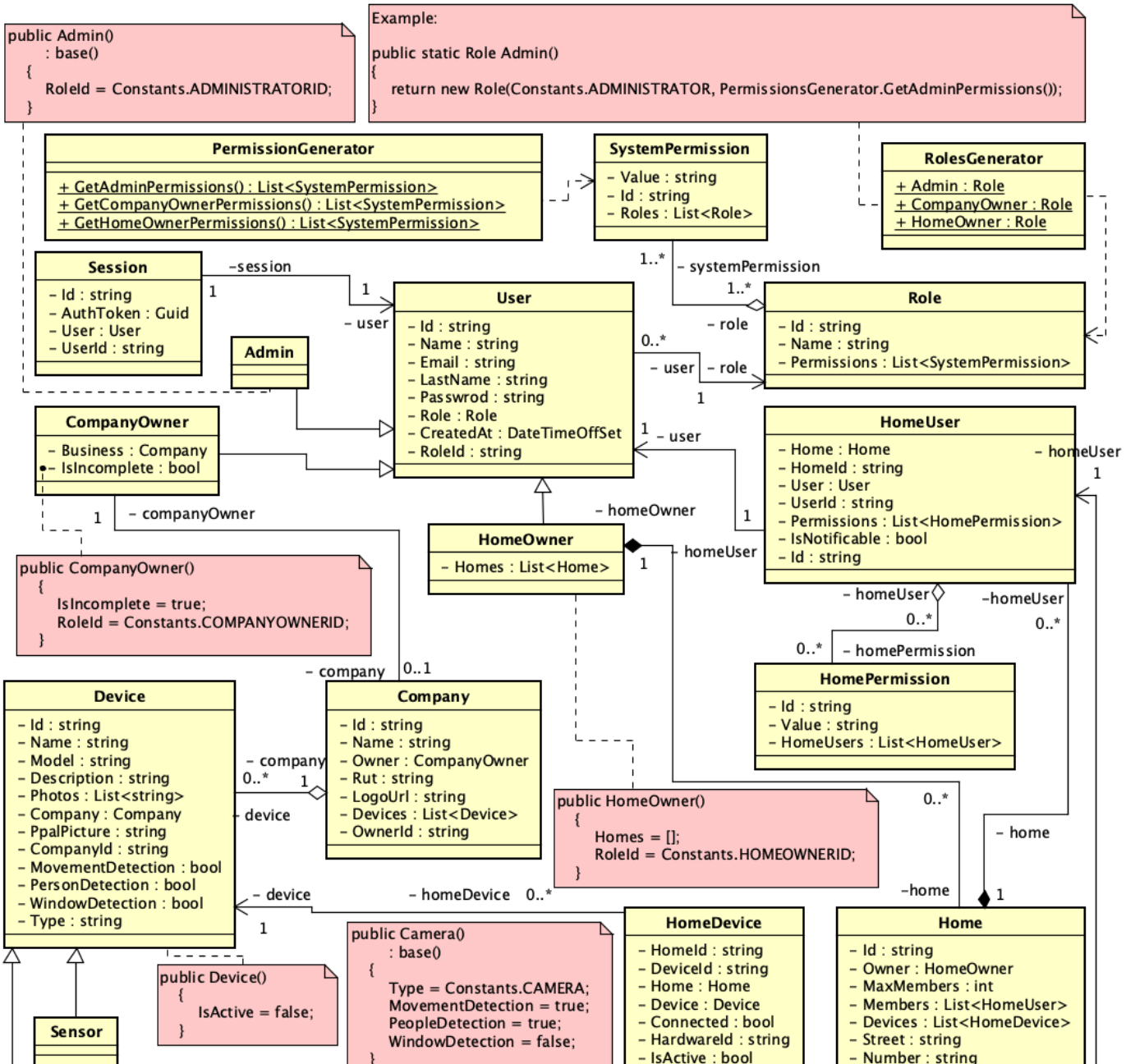
[Ejemplo de clase custom de excepción]

Anexo

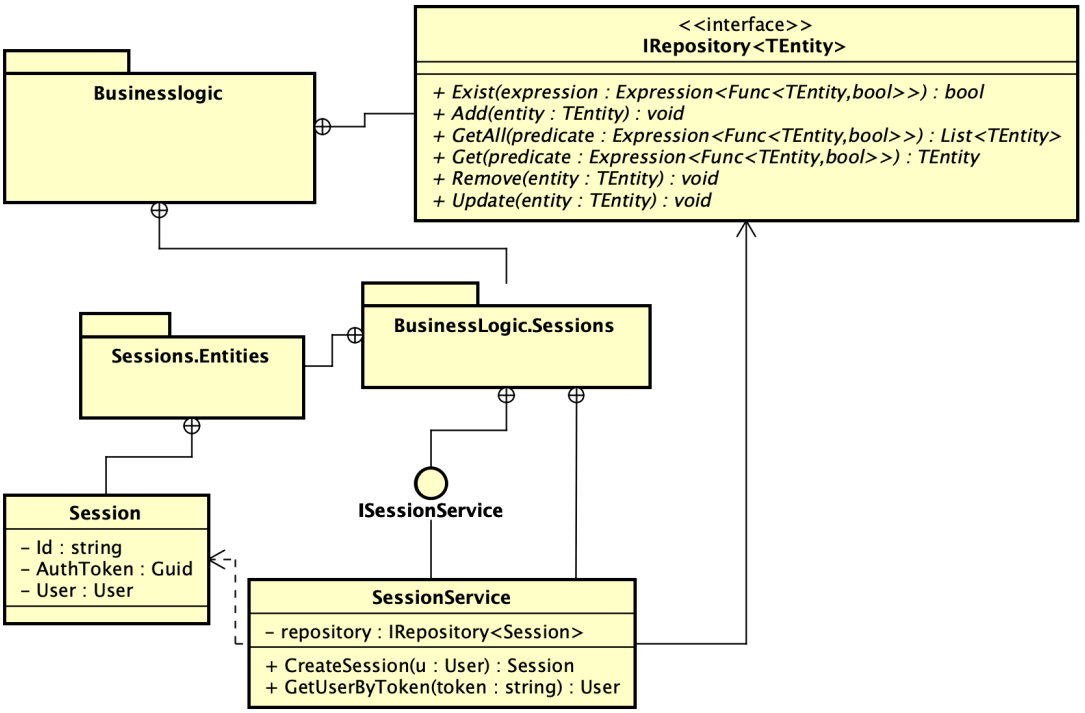
1. Estructuración previa del diagrama de clases con del patrón observer



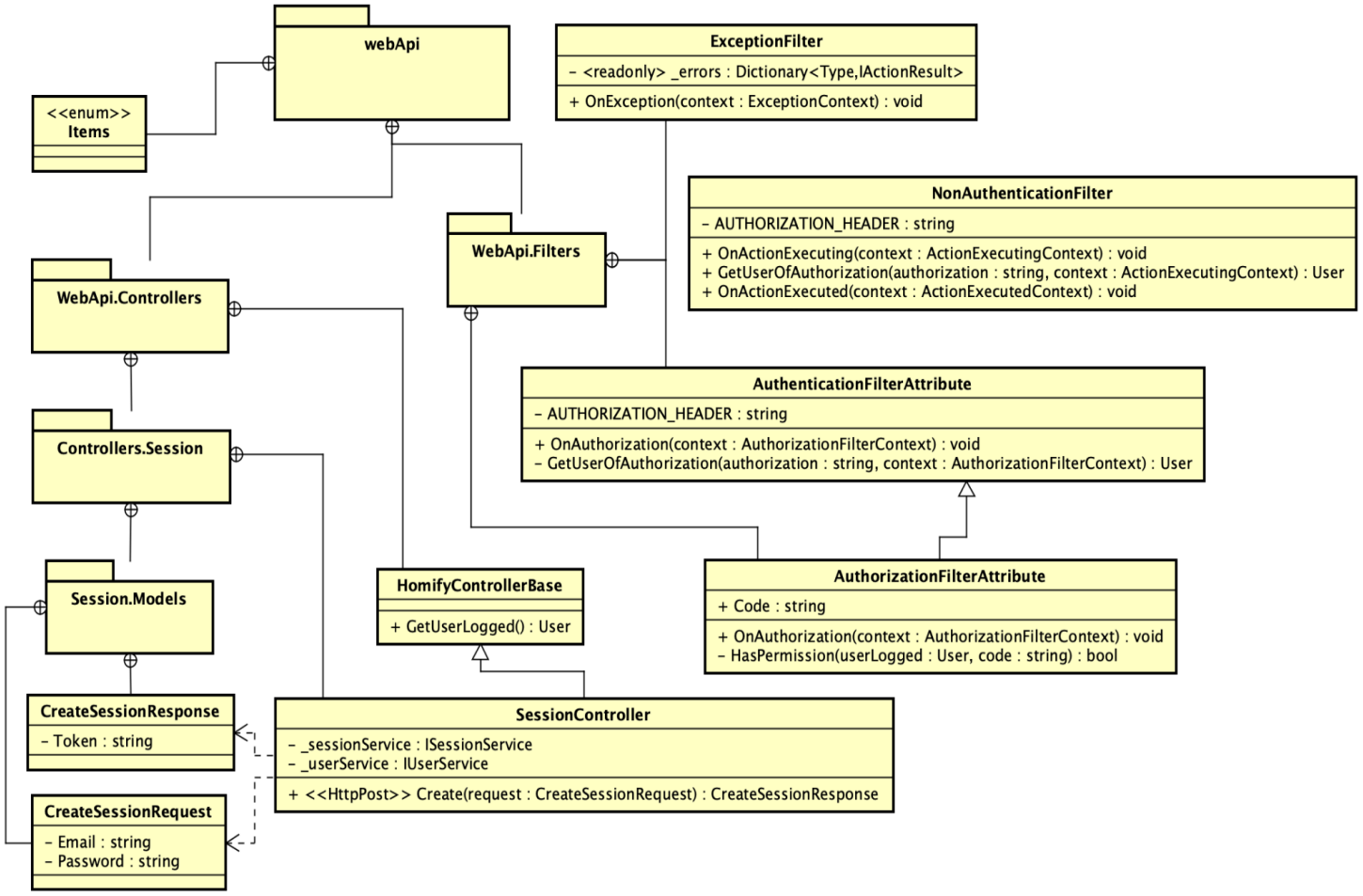
2. Diagrama de clases de las entidades de dominio



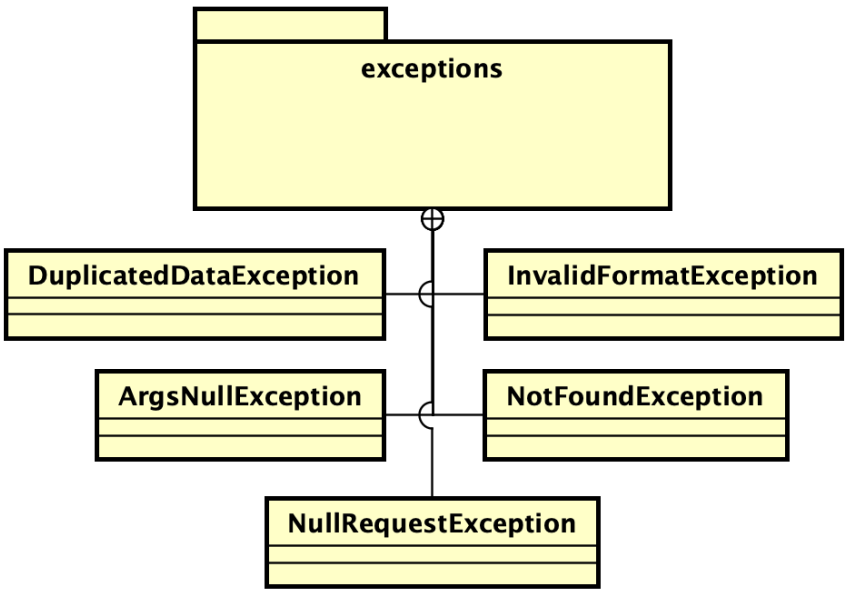
3. Diagrama de clases del paquete de BusinessLogic



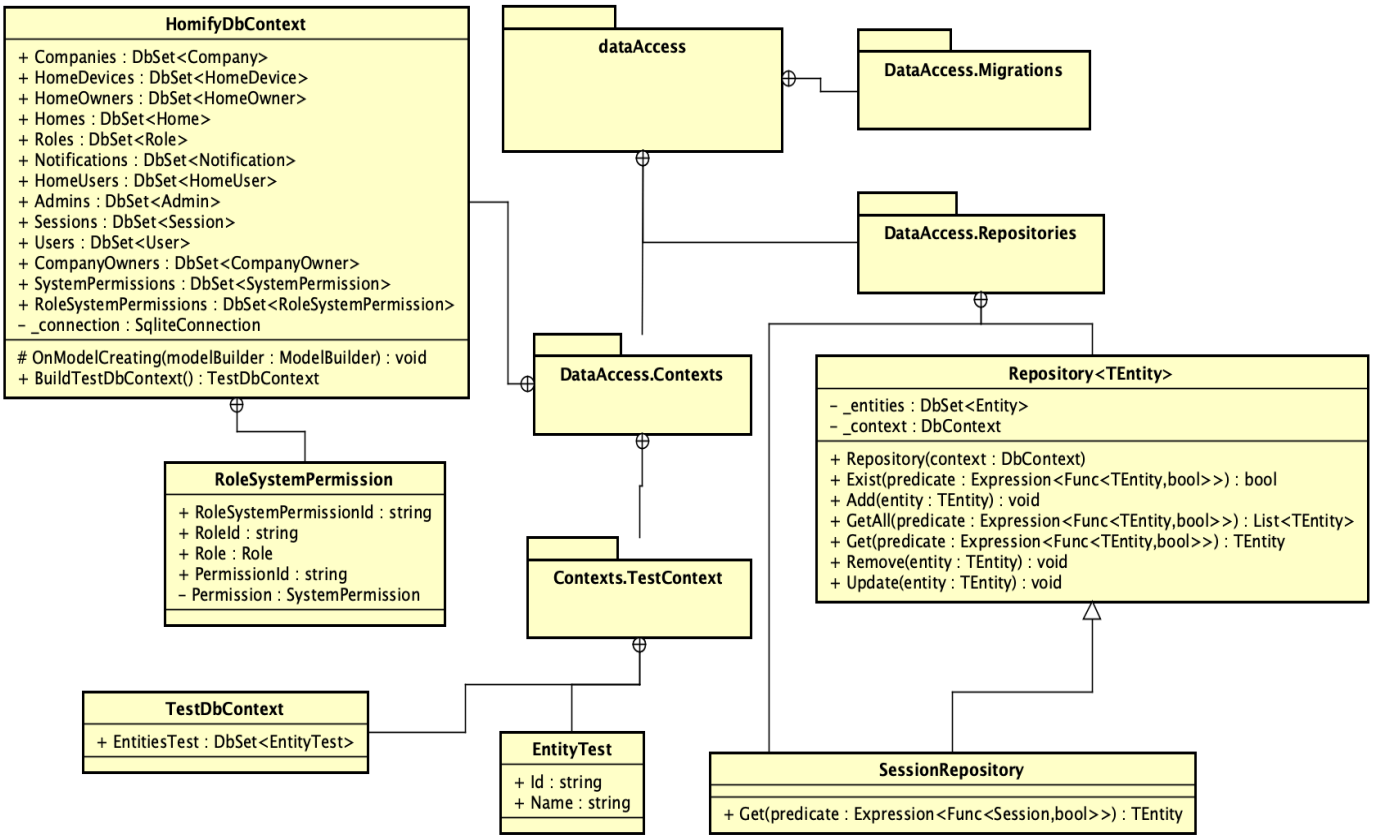
4. Diagrama de clases del paquete de WebApi



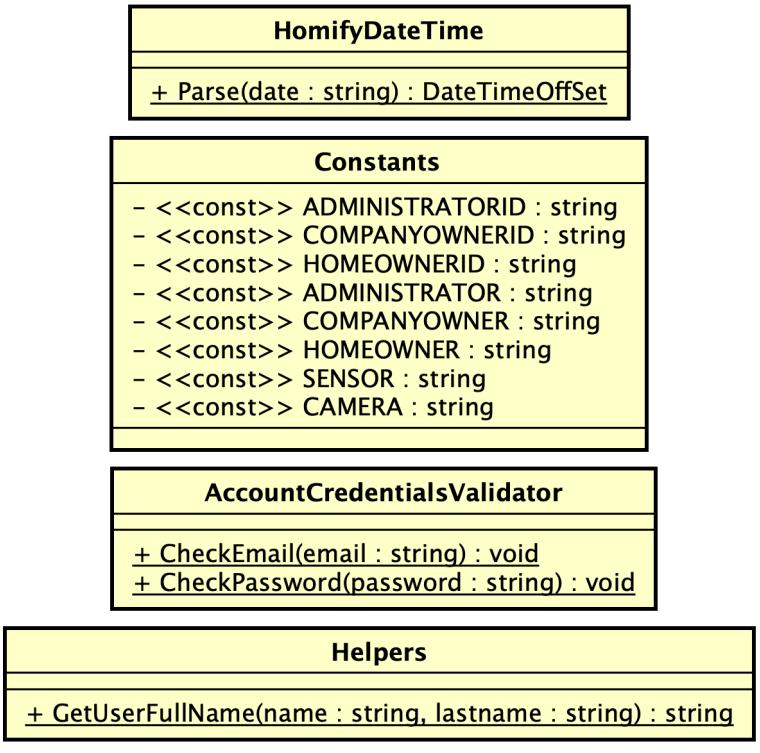
5. Diagrama de clases del paquete de Exceptions



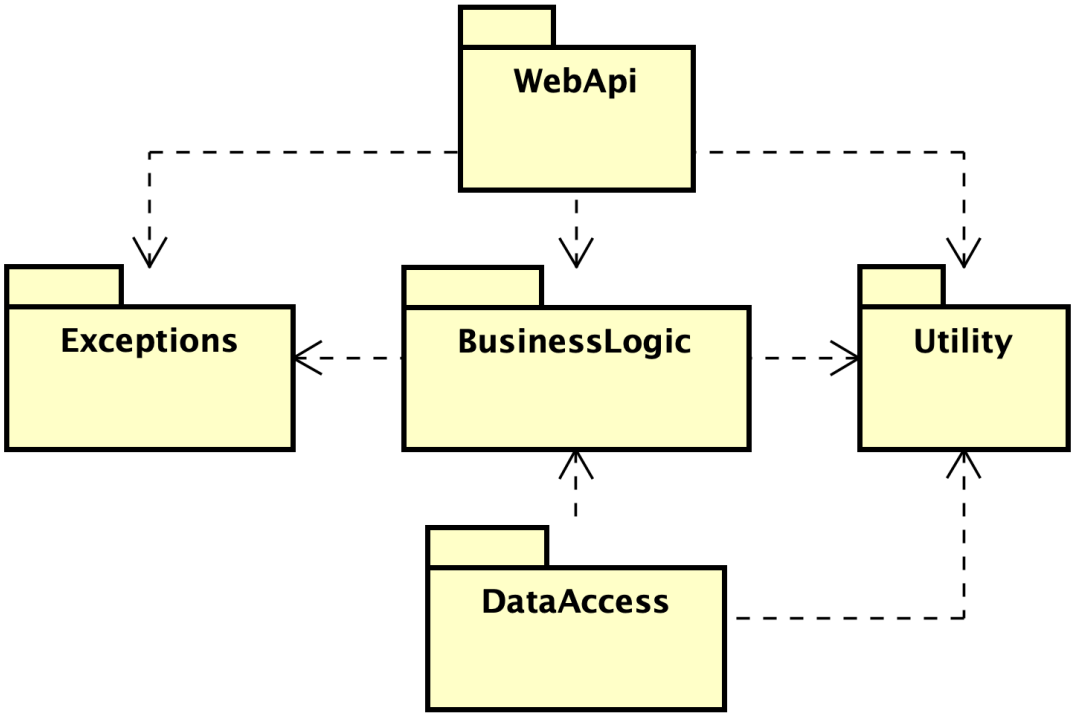
6. Diagrama de clases del paquete de DataAccess



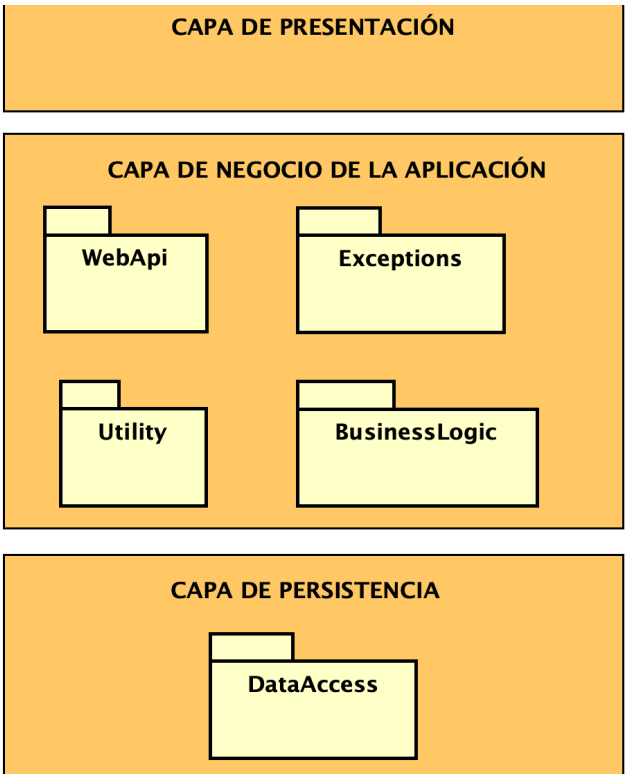
7. Diagrama de clases del paquete de Utility



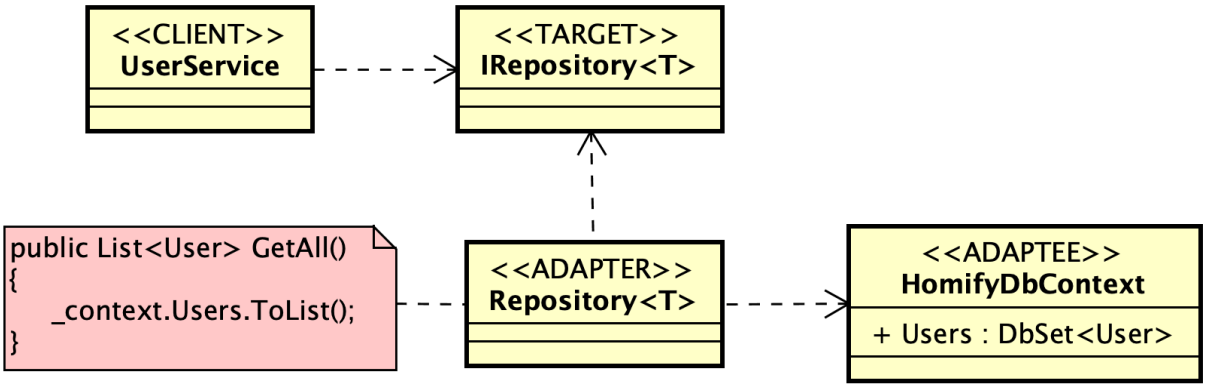
8. **Diagrama general de paquetes**



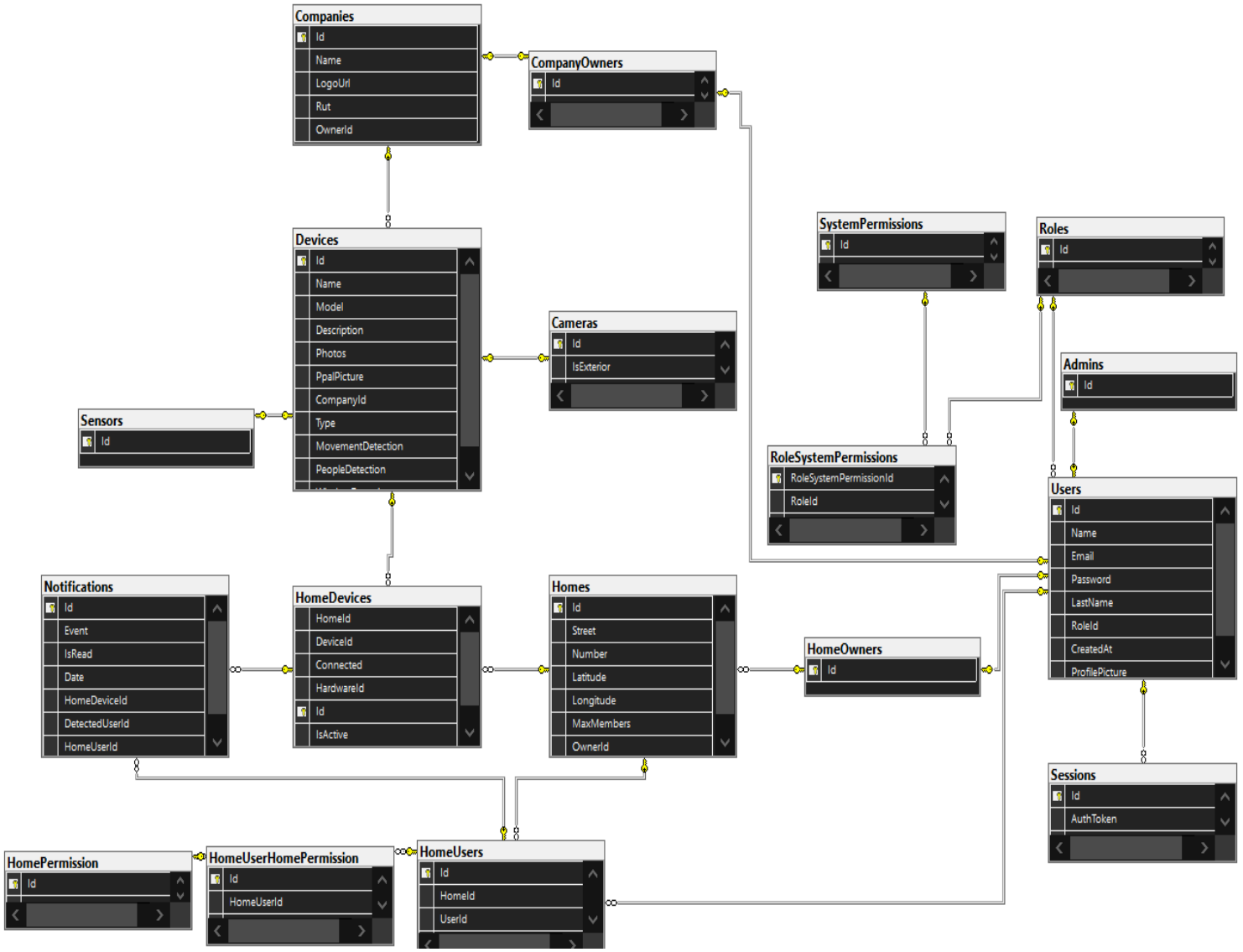
9. **Diagrama de paquetes por layers**



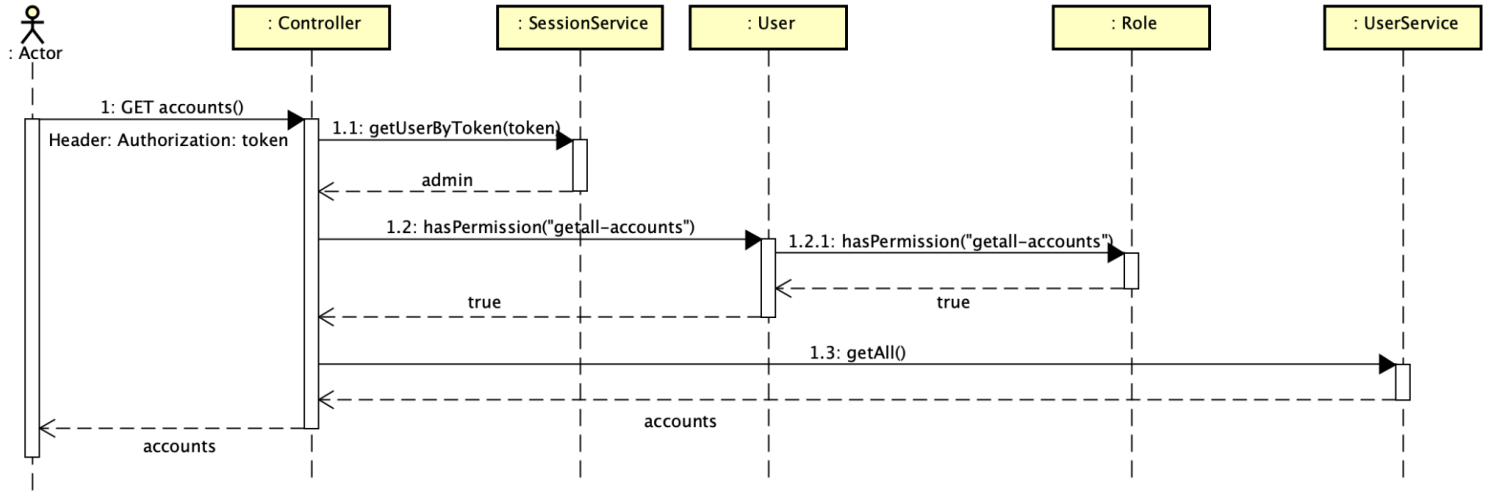
10. Ejemplo del patrón adapter



11. Modelo de la base de datos



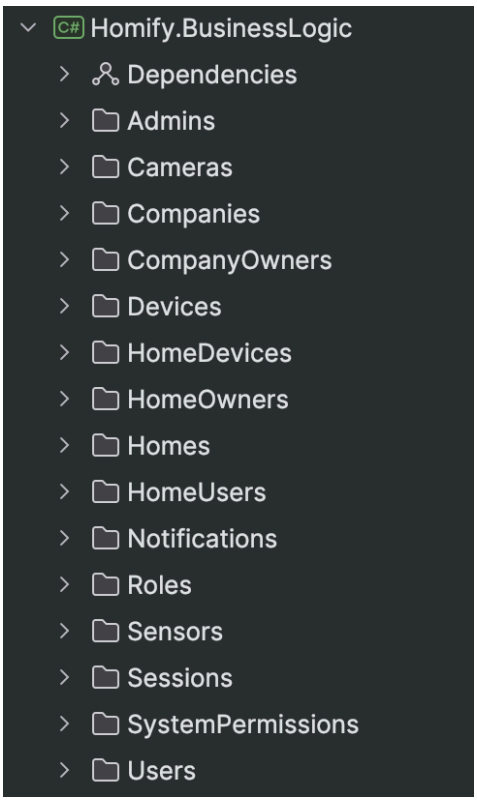
12. Diagrama de interacción del listado de cuentas



13. Foto de las dependencias usadas

```
services.AddDbContext<DbContext, HomifyDbContext>(options => opt

services.AddScoped<IRepository<User>, UserRepository>();
services.AddScoped<IUserService, UserService>();

services.AddScoped<IRepository<Company>, CompanyRepository>();
services.AddScoped<ICompanyService, CompanyService>();

services.AddScoped<IRepository<Device>, Repository<Device>>();
services.AddScoped<IRepository<Camera>, Repository<Camera>>();

services.AddScoped<IRepository<Sensor>, Repository<Sensor>>();
services.AddScoped<IDeviceService, DeviceService>();

services.AddScoped<IRepository<Session>, SessionRepository>();
services.AddScoped<ISessionService, SessionService>();

services.AddScoped<IRepository<Role>, RoleRepository>();
services.AddScoped<IRoleService, RoleService>();

services.AddScoped<IRepository<Home>, HomeRepository>();
services.AddScoped<IHomeService, HomeService>();
```

14. Estructura de las clases de BusinessLogic



15. Ejemplo de clase Custom de Excepcion y ExceptionFilter

```
public class ArgsNullException : Exception
{
    22 usages   18 tests OK *   Guillermo Dotti
    public ArgsNullException(string message)
        : base(message)
    {
    }
}
```

```
public class ExceptionFilter : IExceptionFilter
{
    private readonly Dictionary<Type, Func<Exception, IActionResult>>
     _errors = new Dictionary<Type, Func<Exception, IActionResult>>
     {
         {
             typeof(ArgsNullException), exception =>
                 new ObjectResult(new
                 {
                     InnerCode = "InvalidNullArgument",
                     Message = exception.Message
                 })
                 {
                     StatusCode = (int)HttpStatusCode.BadRequest
                 }
         },
```

16. Diagrama de componentes