Universidad ORT Uruguay

Diseño de Aplicaciones 2

Obligatorio 1

Link al repositorio:
https://github.com/IngSoft-DA2/278311_227286_266889.git

Docentes:
Nicolás Fierro
Alexander Wieler
Máximo Halty

Alumnos:
Nicolás Russo 227286
Matias Hirschfeld 266889
Victoria Chappuis 278311

DOCUMENTO: Descripción del diseño

LAS DEMÁS DOCUMENTACIONES SE ENCUENTRAN EN GITHUB.

SMARTHOME

Esta aplicación fue creada para centralizar y controlar dispositivos inteligentes del hogar, como cámaras de seguridad y sensores de movimiento, permitiendo su gestión desde una única plataforma. Smarthome dispone de varios roles, desde dueños de hogar como también dueños de empresa, entre otros. Los usuarios no autenticados pueden registrarse y autenticarse para acceder a sus cuentas y gestionar sus hogares.

Para esta primera entrega, se desarrolla el backend de la aplicación, el cual tiene el alcance de gestionar usuarios, empresas y la administración de los dispositivos inteligentes. La aplicación cuenta con un sistema de autenticación y autorización que permite a los usuarios registrados acceder a sus cuentas y manejar sus funcionalidades (según su rol) de manera segura. La base de datos utilizada almacena información sobre los usuarios, dispositivos, empresas, y la configuración de los hogares inteligentes. Se implementa un sistema de roles, que incluye propietarios de hogares, propietarios de empresas, y administradores, cada uno con diferentes niveles de acceso y permisos dentro de la plataforma.

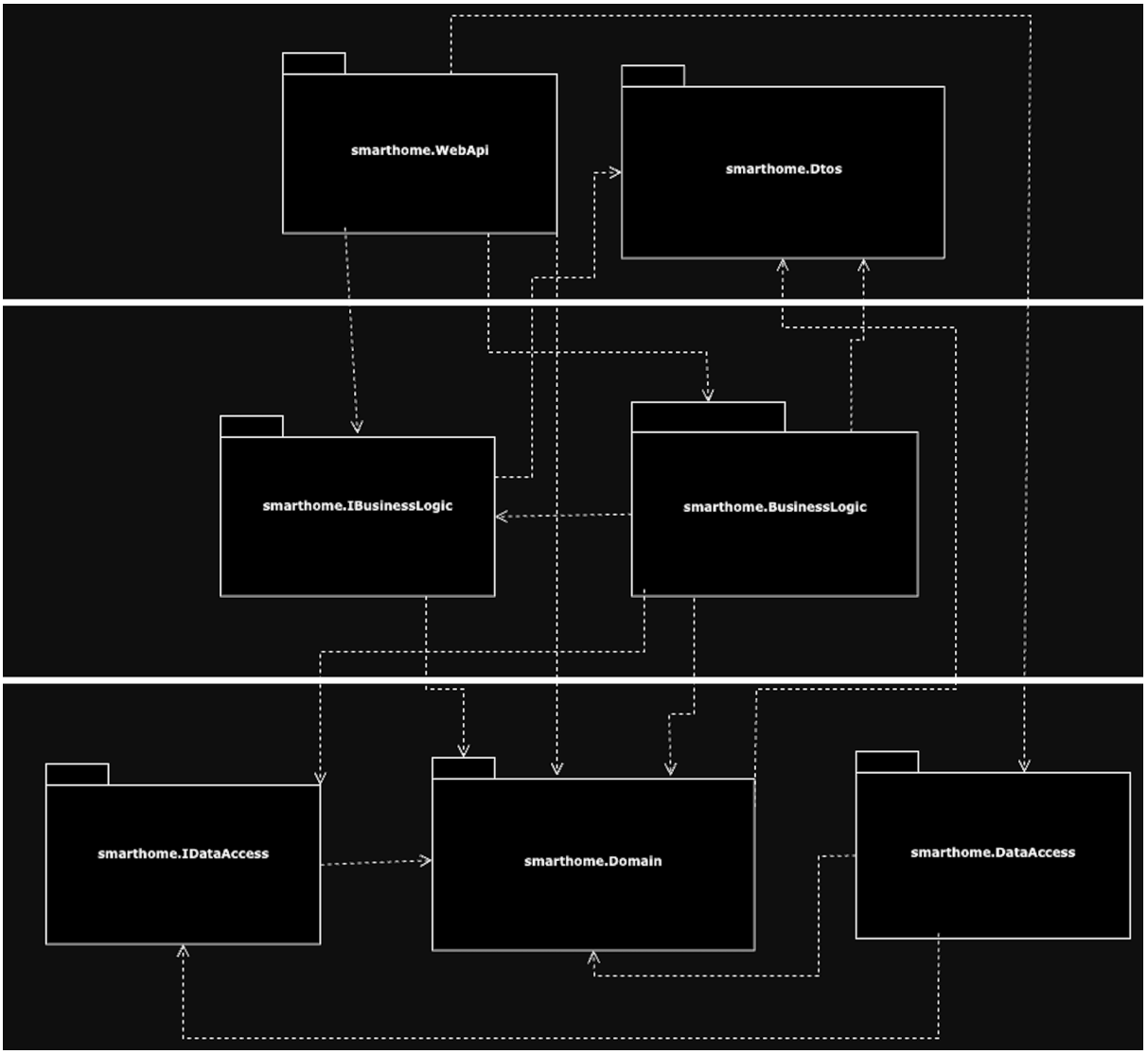
El backend fue diseñado utilizando una arquitectura de microservicios con un patrón de repositorio genérico e inyección de dependencias, garantizando que la gestión de datos y las operaciones CRUD sean escalables y mantenibles. Además, se han implementado pruebas unitarias para validar todas las funcionalidades y asegurar la robustez del sistema.

La base de datos utilizada es Azure SQL Edge, estructurada para almacenar eficientemente la información relacionada con los hogares inteligentes, dispositivos y usuarios. También se utilizan API REST para que los usuarios puedan interactuar con la aplicación a través de clientes móviles o web.

Diseño del sistema

Modelo 4+1

Diagrama de paquetes



smarthome.WebApi

Se encarga de gestionar la comunicación, ruteando los pedidos del cliente a los distintos servicios. Además, se encarga de chequear los permisos del cliente, mediante un filtro por rol.

smarthome.Dtos

Es el paquete responsable de definir los Data Transfer Objects (DTOs) que se utilizan para la transferencia de datos entre las capa de presentacion (webApi) y la capa de lógica de negocio (businessLogic). Estos DTOs actúan como modelos simplificados que contienen

solo la información esencial, asegurando que los datos se transmitan de manera eficiente y segura entre el cliente y el servidor.

Además, los DTOs en este paquete utilizan Data Annotations para definir validaciones y restricciones directamente sobre las propiedades de los objetos. Esto permite asegurar que los datos cumplan con los requisitos necesarios antes de ser procesados, mejorando la integridad y seguridad del sistema.

smarthome.BusinessLogic

Es el paquete encargado de implementar la lógica del negocio de la aplicación. Aquí se encuentran las reglas y procesos que determinan cómo interactúan los usuarios con el sistema y cómo se gestionan las operaciones de los dispositivos inteligentes. Este paquete es responsable de coordinar el flujo de información entre las capas de datos (repositorios) y las capas de presentación (WebApi), asegurando que todas las operaciones cumplan con los requisitos del negocio.

smarthome.IBusinessLogic

Interfaz de businessLogic. Define los contratos para la lógica de negocio, asegurando consistencia y facilitando la inyección de dependencias.

smarthome.DataAccess

Maneja el acceso a datos mediante un solo repositorio genérico. Este permite realizar operaciones CRUD de manera eficiente y reutilizable. Este enfoque simplifica la interacción con la base de datos, mantiene el principio de separación de responsabilidades y crea una capa de abstracción entre la lógica del negocio y la base de datos.

smarthome.IDataAccess

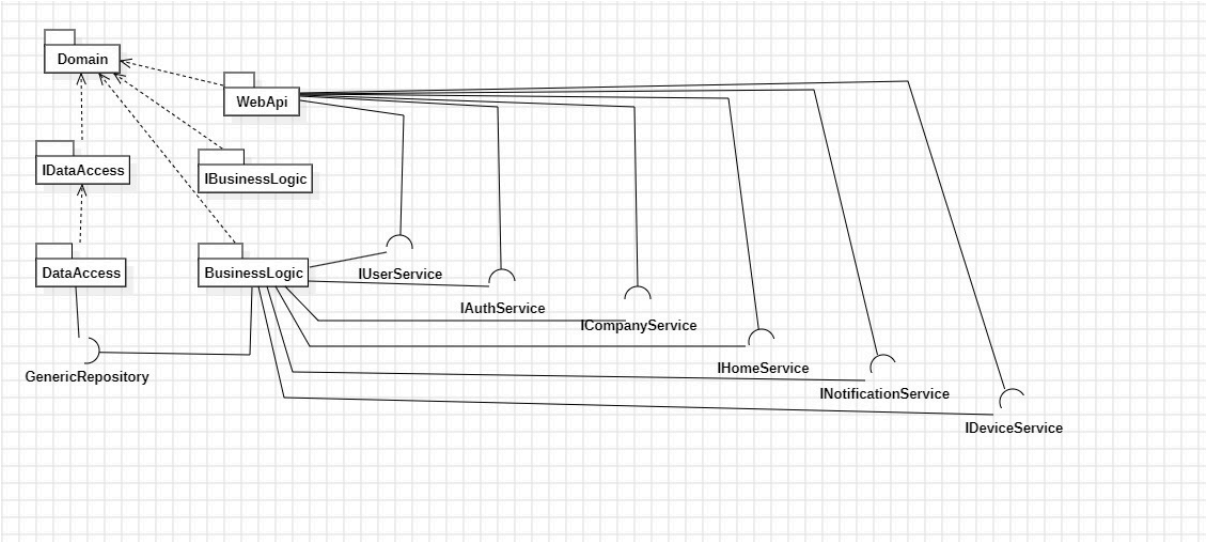
Interfaz de dataAccess. Define los contratos, facilita el desacoplamiento y asegura consistencia.

smarthome.Domain

Define las entidades centrales del sistema, representando los datos clave del negocio, como usuarios, dispositivos y empresas. Estas entidades se centran en la estructura de los objetos y sus propiedades, mientras que la lógica de negocio y el comportamiento asociado

se gestionan en otras capas. Esta organización permite mantener las entidades claras y sencillas, promoviendo la separación de responsabilidades dentro del sistema.

Diagrama de componentes



Justificación de los componentes:

Domain:

- **Justificación:** Separar el dominio garantiza que las reglas de negocio sean independientes de los detalles técnicos de implementación, permitiendo que el núcleo del sistema permanezca consistente y adaptable. Esto facilita la reutilización de la lógica de negocio en diferentes contextos y hace que la aplicación sea más fácil de extender en el futuro.

DataAccess/GenericRepository:

- **Justificación**: Esta capa aísla el acceso a los datos, evitando que la aplicación dependa directamente de una base de datos específica. Al usar un repositorio genérico, se fomenta la reutilización del código y se simplifican las operaciones CRUD para las distintas entidades del dominio, lo que mejora la flexibilidad y el mantenimiento del sistema.

IDataAccess y IBusinessLogic (interfaces):

- **Justificación**: Las interfaces establecen contratos claros entre capas, permitiendo cambiar implementaciones sin afectar a las capas superiores. Son esenciales para la inyección de dependencias, ya que permiten gestionar dependencias de forma

abstracta, haciendo que las clases no dependan de implementaciones concretas. Esto también facilita las pruebas unitarias, al permitir simular (mockear) los servicios con flexibilidad. Así, se mejora la modularidad y el mantenimiento del sistema.

BusinessLogic/Servicios:

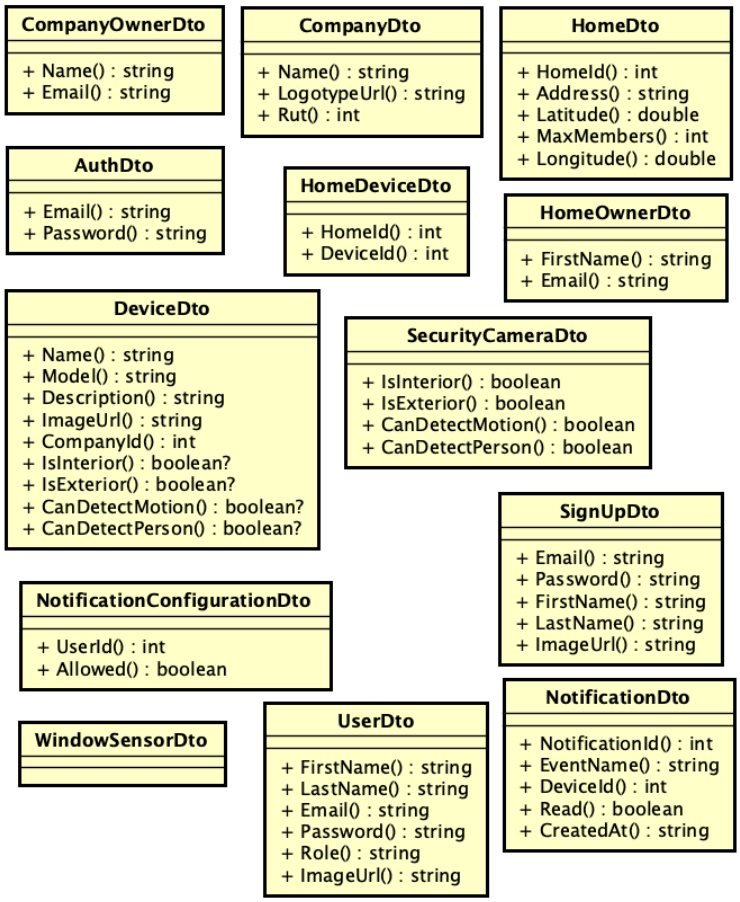
- **Justificación**: Centraliza toda la lógica de negocio de la aplicación. Esto facilita su gestión y escalabilidad. La creación de servicios separados para diferentes áreas de la aplicación permite una mejor organización del código y facilita el mantenimiento. Cada servicio se enfoca en una parte específica de la aplicación, asegurando una clara separación de responsabilidades.

WebApi:

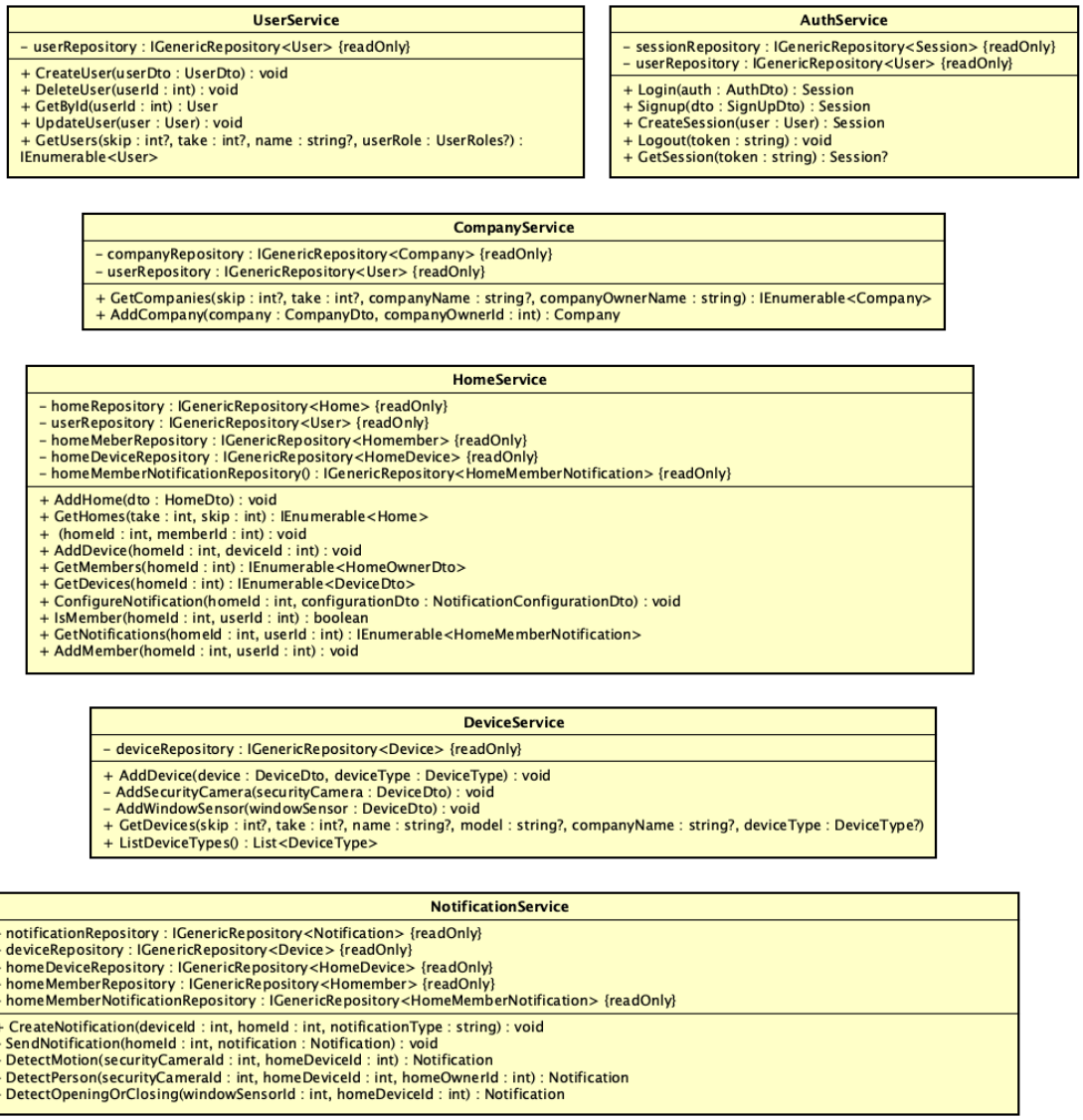
- **Justificación**: WebApi se encarga de la comunicación entre el cliente y el servidor, exponiendo los endpoints de manera que las otras capas no necesiten conocer detalles sobre las interfaces de usuario. Esto promueve una arquitectura más limpia, donde la API solo se preocupa de servir como intermediaria entre el cliente y la lógica de negocio.

Diagrama de clases

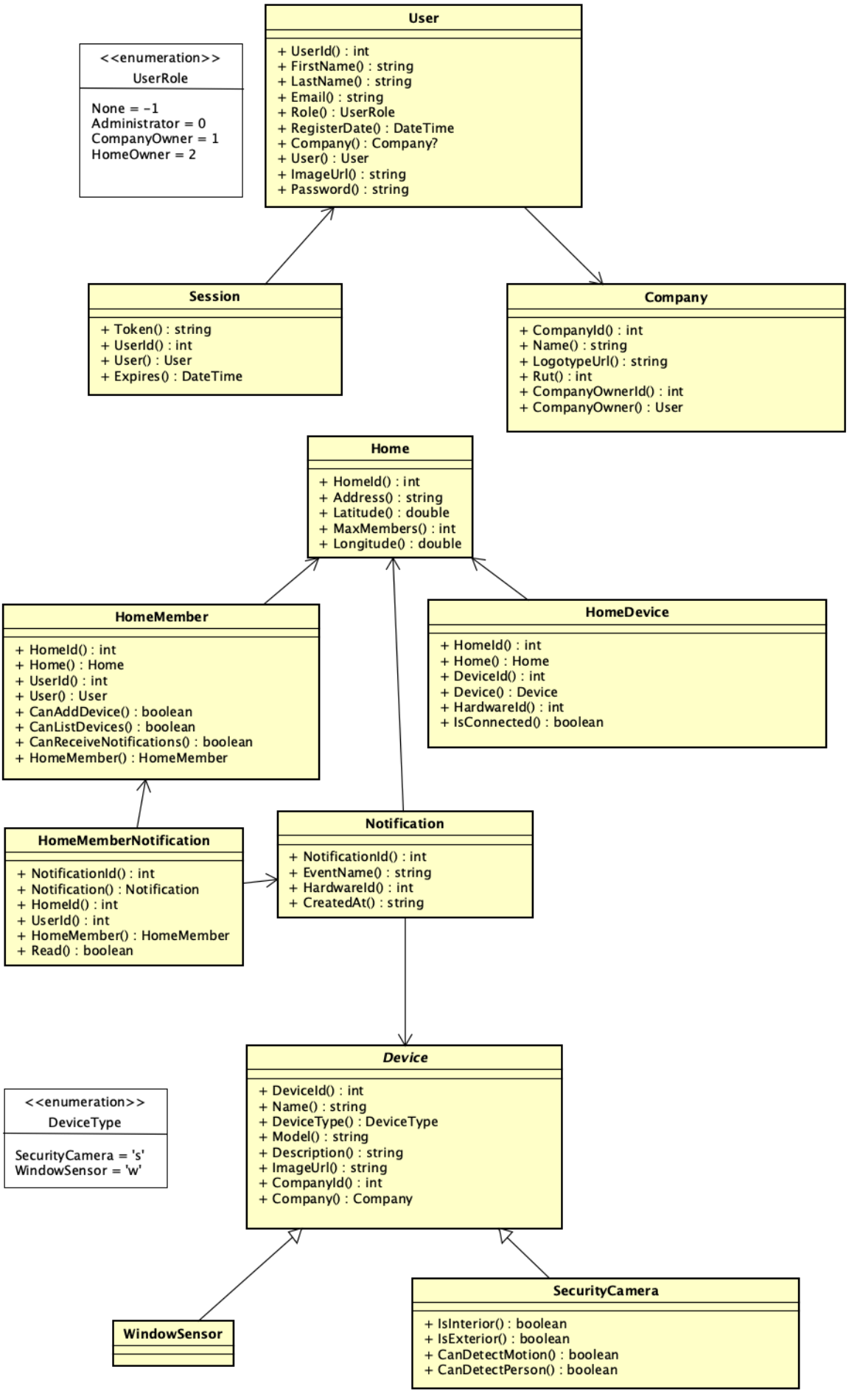
Paquete: smarthome.Dtos



Paquete: smarthome.BusinessLogic



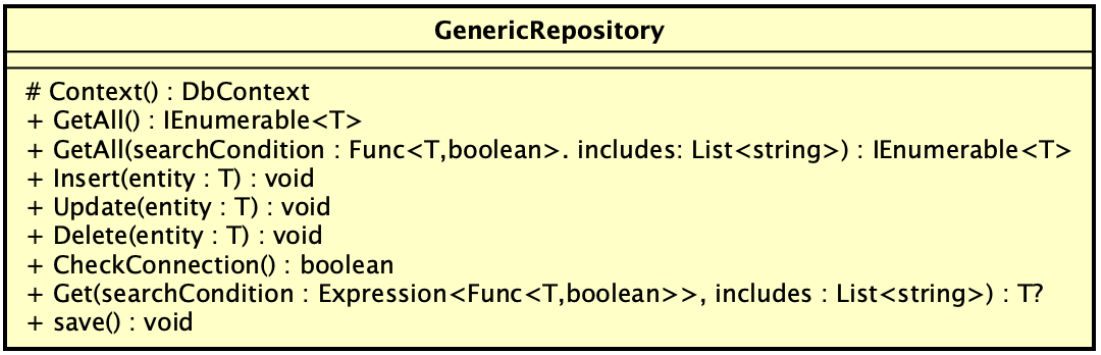
Paquete: smarthome.Domain


User
+ UserId() : int
+ FirstName() : string
+ LastName() : string
+ Email() : string
+ Role() : UserRole
+ RegisterDate() : DateTime
+ Company() : Company?
+ User() : User
+ ImageUrl() : string
+ Password() : string
<<enumeration>>
UserRole
None = -1
Administrator = 0
CompanyOwner = 1
HomeOwner = 2
Session
+ Token() : string
+ UserId() : int
+ User() : User
+ Expires() : DateTime
Company
+ CompanyId() : int
+ Name() : string
+ LogotypeUrl() : string
+ Rut() : int
+ CompanyOwnerId() : int
+ CompanyOwner() : User
Home
+ HomeId() : int
+ Address() : string
+ Latitude() : double
+ MaxMembers() : int
+ Longitude() : double
HomeMember
+ HomeId() : int
+ Home() : Home
+ UserId() : int
+ User() : User
+ CanAddDevice() : boolean
+ CanListDevices() : boolean
+ CanReceiveNotifications() : boolean
+ HomeMember() : HomeMember
HomeDevice
+ HomeId() : int
+ Home() : Home
+ DeviceId() : int
+ Device() : Device
+ HardwareId() : int
+ IsConnected() : boolean
HomeMemberNotification
+ NotificationId() : int
+ Notification() : Notification
+ HomeId() : int
+ UserId() : int
+ HomeMember() : HomeMember
+ Read() : boolean
Notification
+ NotificationId() : int
+ EventName() : string
+ HardwareId() : int
+ CreatedAt() : string
Device
+ DeviceId() : int
+ Name() : string
+ DeviceType() : DeviceType
+ Model() : string
+ Description() : string
+ ImageUrl() : string
+ CompanyId() : int
+ Company() : Company
<<enumeration>>
DeviceType
SecurityCamera = 's'
WindowSensor = 'w'
WindowSensor
SecurityCamera
+ IsInterior() : boolean
+ IsExterior() : boolean
+ CanDetectMotion() : boolean
+ CanDetectPerson() : boolean

En cuanto a la herencia en device:

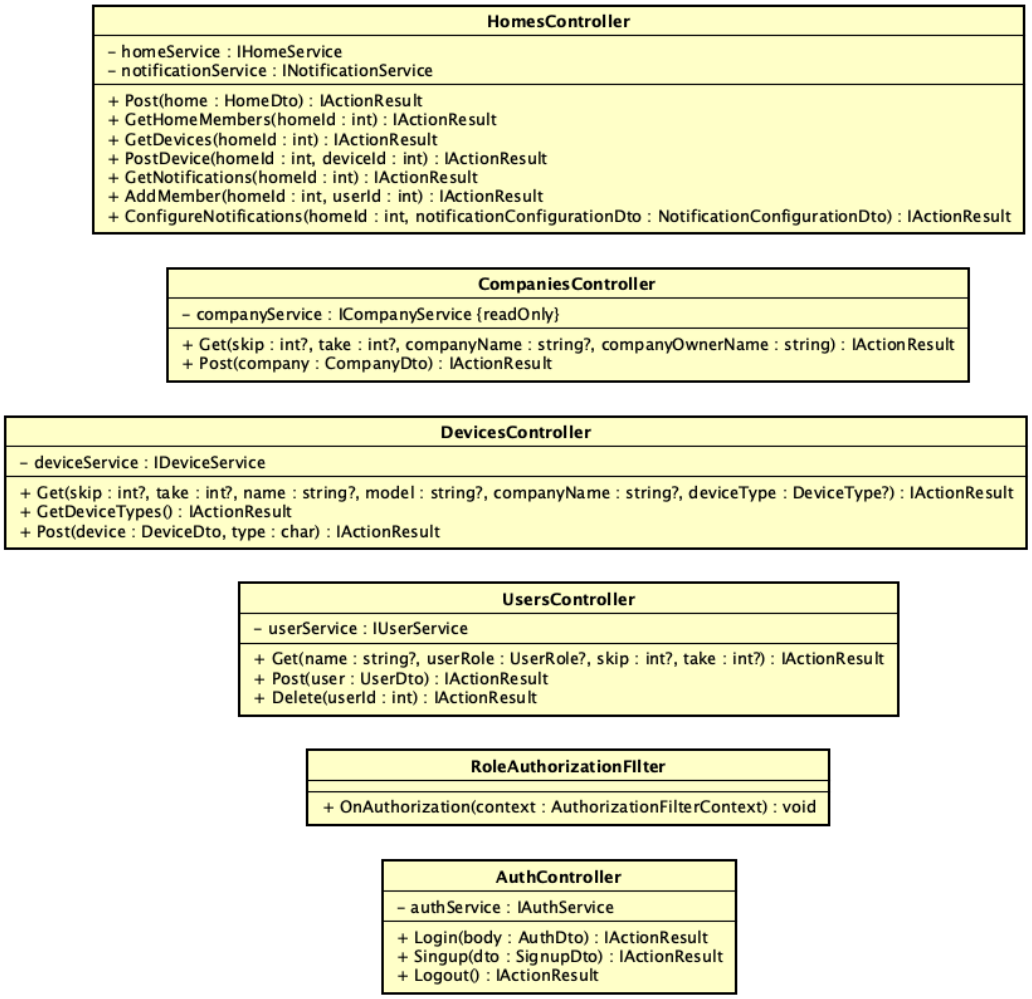
Device es una clase abstracta que define los atributos esenciales de cualquier dispositivo tecnológico. Optamos por este enfoque porque prevemos que es muy probable que la empresa quiera añadir nuevos dispositivos con características particulares. Esta estructura garantiza que todas las subclases compartan los atributos comunes, manteniendo la consistencia y evitando duplicación de código. Además, sigue el principio de Open/Closed, ya que permite extender el sistema con nuevos dispositivos sin modificar las clases existentes, asegurando flexibilidad para futuras expansiones.

Paquete: smarthome.DataAccess



Los servicios, para utilizar el repositorio genérico, crean una instancia del repositorio y especifican el tipo genérico T con la entidad que desean manejar (por ejemplo, User, Company, etc.). De esta manera, el repositorio genérico se adapta a la entidad concreta que el servicio necesita gestionar, permitiendo realizar operaciones CRUD sobre esa entidad específica. Este enfoque tipificado asegura que el repositorio maneje correctamente las propiedades y el comportamiento de cada entidad, mientras reutiliza la lógica genérica para diferentes tipos de objetos. Esto simplifica la estructura del código y evita duplicar métodos CRUD en distintos repositorios, mejorando la flexibilidad y escalabilidad del sistema.

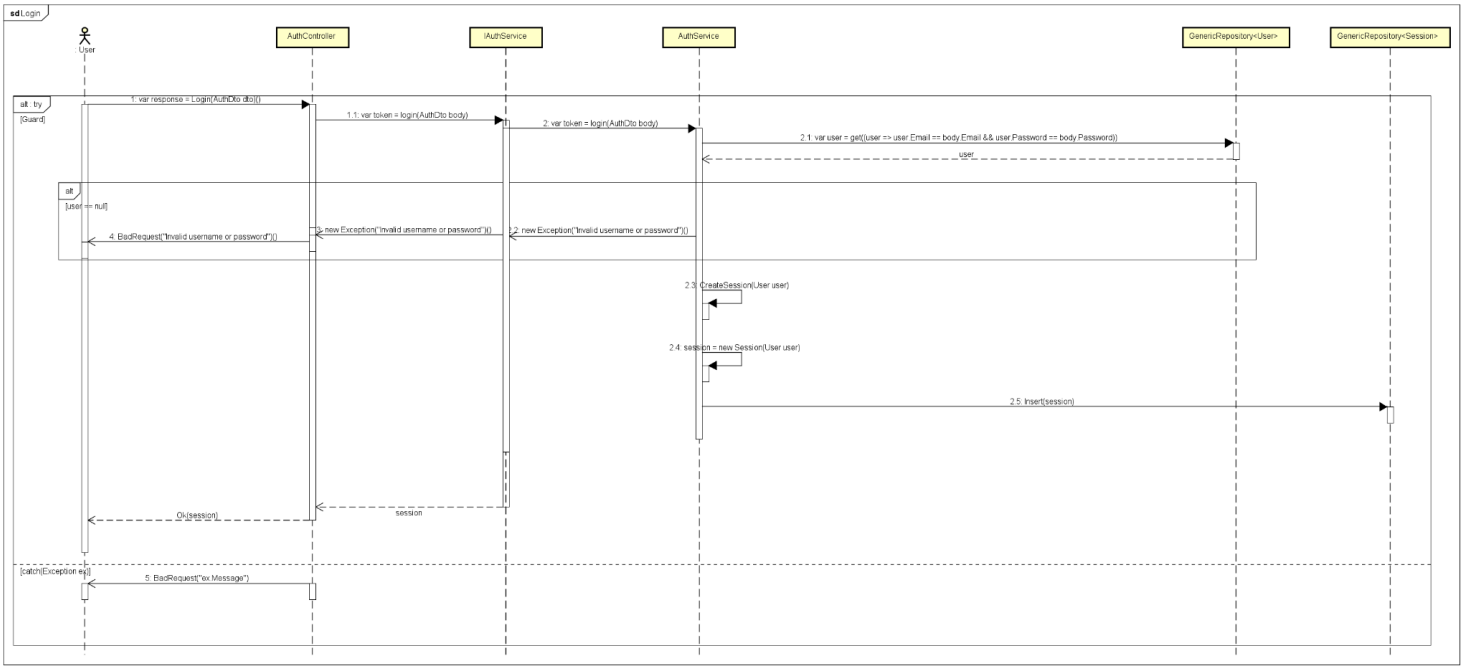
Paquete: smarthome.WebApi



<u>Nota:</u> Los diagramas de clase de interfaces, IBusinessLogic e IDataAccess, no fueron diagramados ya que son muy similares a los paquetes BusinessLogic y DataAccess respectivamente. No añaden detalles significativos que justifiquen su representación en un diagrama por separado.

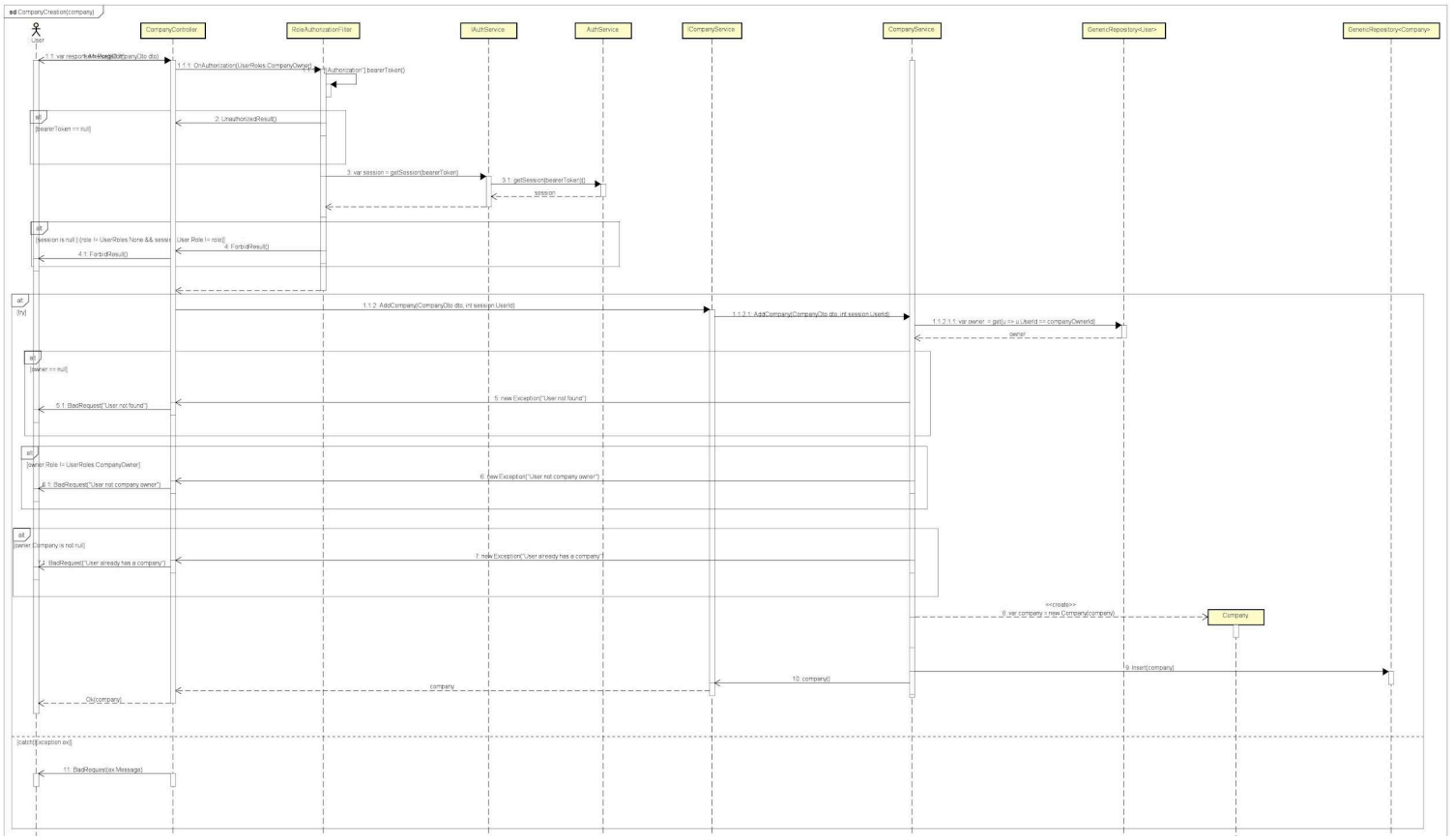
Diagramas de interacción

- **Autenticación mediante credenciales**



Para poder verlo mejor adjuntamos el siguiente drive: **https://drive.google.com/file/d/1JjAukoATVj4RFM7XLnJDza_y5eXdVKgY/view?usp=drive_link**

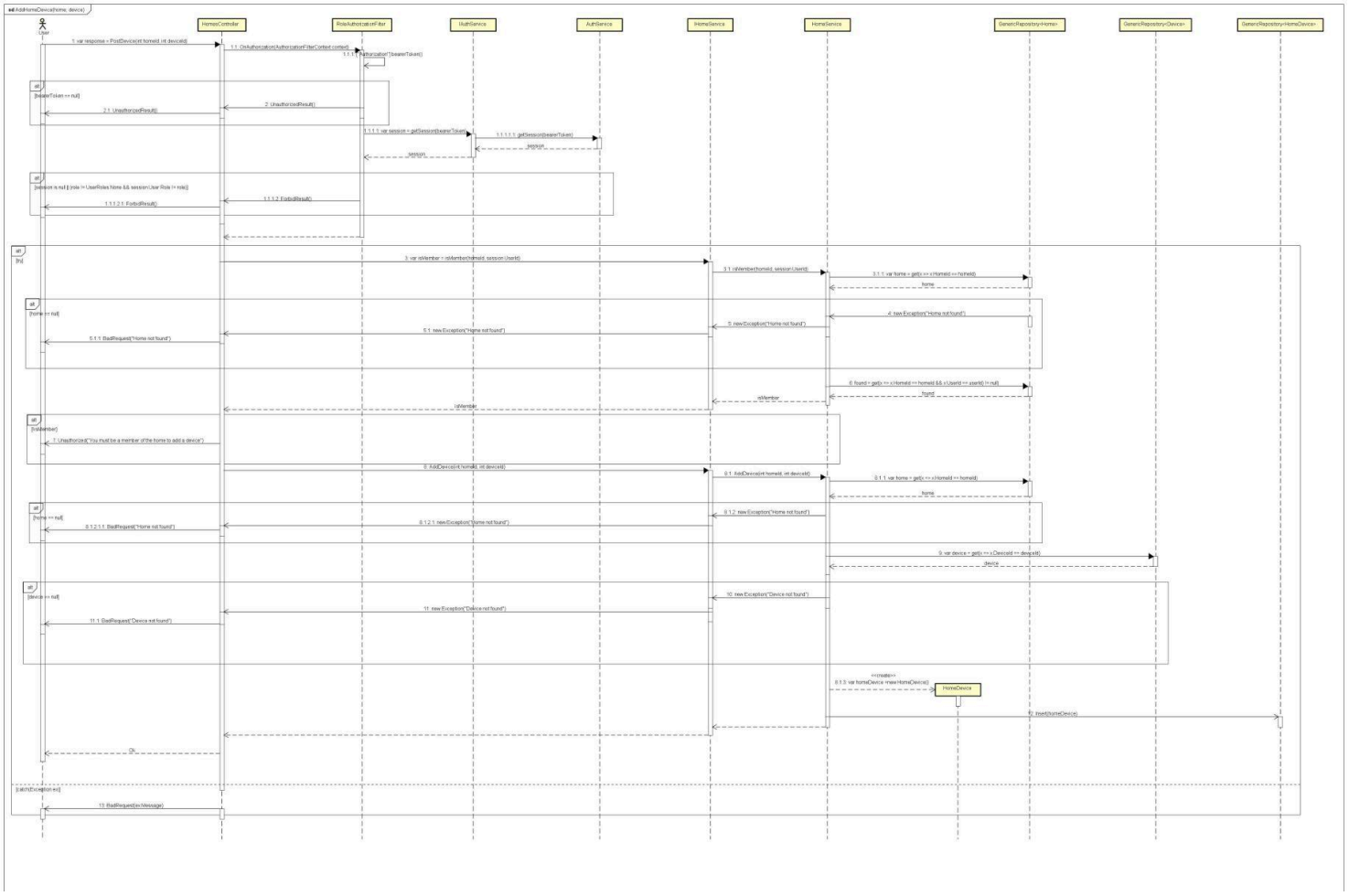
- **Creacion de una empresa**



Para poder verlo mejor adjuntamos el siguiente drive:

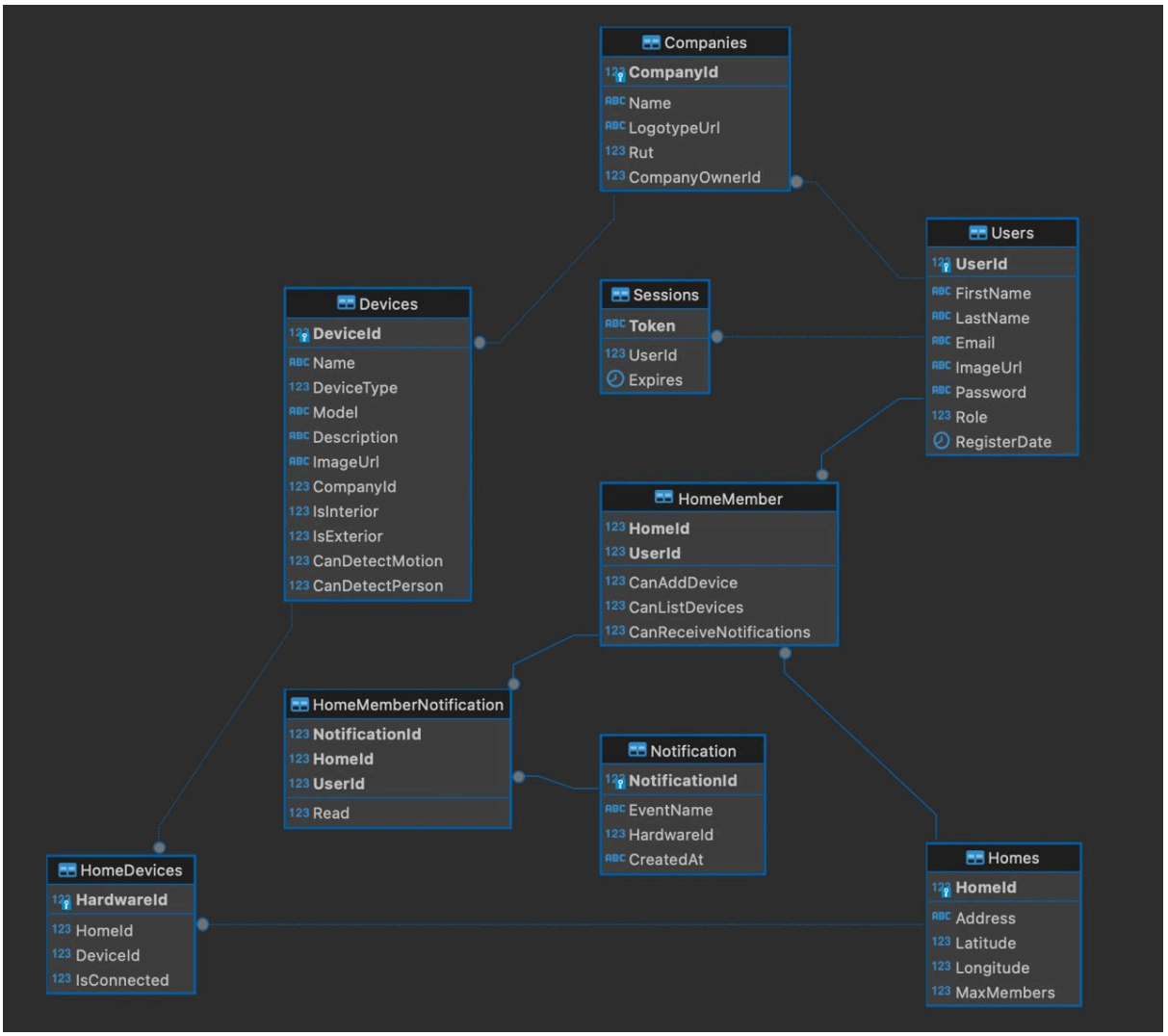
https://drive.google.com/file/d/1CN2CmuFigez6yyG8_ubfhfwv6rB1R-19/view?usp=drive_link

- **Agregar dispositivos a un hogar**



Para poder verlo mejor adjuntamos el siguiente drive: https://drive.google.com/file/d/19RnjDCRtCvY3ndor3K4oPN3pq7AqT_81/view?usp=drive_link

Diagrama de la base de datos



Principios SOLID reflejados en nuestro proyecto

Aunque se encuentran a lo largo de todo el proyecto, daremos un ejemplo concreto para cada principio.

Single Responsibility Principle (SRP)

Damos como ejemplo las clases Notification y HomeMemberNotification. Podríamos haberlas combinado ambas en una sola, sin embargo, decidimos separarlas para mantener la responsabilidad de cada clase bien definida. Notification almacena únicamente los detalles del evento generado por un dispositivo, como el nombre del evento, el HardwareId, y la fecha de creación, mientras que HomeMemberNotification se encarga de relacionar a

un miembro del hogar con una notificación y de registrar si ha sido leída. Esta separación en dos clases asegura que los datos relacionados con las notificaciones y los miembros del hogar estén claramente organizados, lo que facilita el mantenimiento y la comprensión del código. Al seguir el SRP, nos aseguramos de que cada clase tenga una única razón para cambiar y que las responsabilidades no se mezclen, manteniendo la cohesión y claridad.

Open/Closed Principle (OCP)

En la herencia de Device, aplicamos el OCP al mantener la clase Device cerrada para modificaciones, ya que sus atributos y métodos esenciales están bien definidos y no necesitan cambiar. Sin embargo, estamos abiertos a la extensión al poder agregar fácilmente nuevos tipos de dispositivos que hereden de Device, como SecurityCamera o WindowSensor, sin alterar la estructura de la clase base.

Liskov Substitution Principle (LSP)

Tanto SecurityCamera como WindowSensor son subclases de Device y pueden ser utilizadas donde se espera una instancia de Device sin alterar el comportamiento esperado del sistema. Ambas subclases heredan los atributos y métodos de Device y agregan características específicas sin modificar la funcionalidad básica. Esto garantiza que cualquier instancia de SecurityCamera o WindowSensor pueda sustituir a Device en el código sin causar errores o comportamientos inesperados, respetando así el contrato establecido por la clase base.

Interface Segregation Principle (ISP)

En el caso de IBusinessLogic, tener una interfaz separada por cada servicio asegura que cada servicio solo implemente los métodos que realmente necesita. Esto respeta el Principio de Segregación de Interfaces, ya que evita que los servicios dependan de métodos que no utilizan. Al crear interfaces pequeñas y enfocadas para cada servicio, como IUserService, ICompanyService, o IAuthService, nos aseguramos de que las clases implementadoras no se vean obligadas a manejar métodos innecesarios, mejorando así la cohesión y reduciendo el acoplamiento. Este enfoque facilita a los serviciosaevolucionar de manera independiente sin afectar a otros, y cada interfaz sigue siendo específica y relevante para el servicio que la implementa.

Dependency Inversion Principle (DIP)

A través del uso de interfaces (IDataAccess, IBusinessLogic), desacoplamos las clases de alto nivel (como los servicios) de las clases de bajo nivel (como las implementaciones concretas de acceso a datos). Al gestionar las dependencias mediante inyección de

dependencias, garantizamos que las clases de alto nivel interactúen con abstracciones en lugar de implementaciones específicas, lo que mejora el mantenimiento y facilita la extensibilidad del sistema.

Patrones GRASP implementados en nuestro proyecto

En nuestro proyecto aplicamos varios de los patrones GRASP. Cumplimos con **bajo acoplamiento** gracias al uso de interfaces, que permiten desacoplar las clases de alto nivel de las de bajo nivel, evitando dependencias directas entre las implementaciones.

Además, logramos **alta cohesión** debido al cumplimiento del **Principio de Responsabilidad Única (SRP)**. Cada clase y componente tiene una única responsabilidad bien definida, como las entidades en la capa de dominio que representan datos, los servicios que manejan la lógica de negocio, y el repositorio que gestiona las operaciones de persistencia.

La **indirección** está presente mediante el uso de interfaces como IDataAccess e IBusinessLogic, que actúan como intermediarios entre las clases concretas. Esto nos permite utilizar inyección de dependencias para gestionar las implementaciones concretas de manera flexible, protegiendo así a las clases de alto nivel de los detalles de implementación.

También utilizamos el patrón de **variaciones protegidas** al seguir el **Principio de Abierto/Cerrado (OCP)**. La clase Device está diseñada para ser extendida sin necesidad de ser modificada. Esto permite agregar nuevos tipos de dispositivos, como SecurityCamera o WindowSensor, sin alterar la clase base. Así, protegemos al sistema de los cambios derivados de nuevas implementaciones, asegurando que el núcleo del sistema permanezca estable y no se vea afectado por futuras adiciones o modificaciones en los dispositivos.

El **polimorfismo** juega un papel clave en este diseño, ya que permite que las clases concretas como SecurityCamera y WindowSensor implementen comportamientos específicos mientras siguen la abstracción definida por la clase base Device. Esto nos permite manejar múltiples tipos de dispositivos de manera uniforme, simplificando el código y haciendo que el sistema sea más flexible y extensible ante nuevos cambios.

Manejo de excepciones

En nuestro proyecto, actualmente manejamos las excepciones en los controladores de la Web API mediante bloques try-catch. Cuando se lanza una excepción como ArgumentException, la capturamos y devolvemos un código HTTP adecuado, como 400 (Bad Request), para indicar al cliente que ha enviado una solicitud inválida. Este enfoque asegura que las excepciones sean controladas y las respuestas sean coherentes.

Sin embargo, entendemos que este método puede volverse repetitivo a medida que agregamos más controladores. Para evitar duplicación y mejorar la mantenibilidad, consideramos que se podría implementar un middleware de manejo de excepciones centralizado. Este middleware capturaría y gestionaría todas las excepciones de forma global, devolviendo códigos de estado HTTP consistentes y reduciendo la necesidad de tener try-catch en cada controlador.