Universidad ORT Uruguay

Facultad de Ingeniería

Documentación Obligatorio 1

Descripción del diseño

Diseño de Aplicaciones 2

Link al Repositorio: IngSoft-DA2/301178_231810_280070 (github.com)

Integrantes:

- Angelina Maverino - 280070
- Yliana Otero - 301178
- María Belén Rywaczuk - 231810

Tutores:

- Juan Ignacio Irabedra De Maio
- Juan Pablo Barrios García
- Ignacio Valle Dubé

Grupo N6A - 2024

ÍNDICE

Introducción

En esta ocasión tuvimos que diseñar y desarrollar una aplicación para gestionar diversos dispositivos inteligentes en un hogar creando así un entorno integrado y automatizado. Para ello diseñamos una solución capaz de cumplir las funcionalidades y requerimientos demandados por la letra. Dentro de los mismos podemos encontrar requerimientos funcionales como el mantenimiento y la creación de cuentas, y requerimientos no funcionales tales como la inicialización de datos predefinidos.

Para cumplir con la problemática planteada, utilizamos distintas herramientas tales como Github, Plant Text, Docker, Rider, Postman, Dbeaver, entre otras. Las mismas habían sido introducidas para instancias de evaluaciones anteriores en Diseño de Aplicaciones 1.

Todo este trabajo se realizó utilizando TDD y aplicando los conocimientos adquiridos en las clases teóricas y prácticas. Nos basamos en principios de Clean Code, SOLID y GRASP para mejorar el diseño y la calidad del código. La base de datos fue implementada con Entity Framework, lo que garantiza la persistencia de los datos.

Descripción general del trabajo y errores conocidos

Nuestro primer acercamiento con el proyecto fue identificar los distintos endpoints en la letra y anotarlos en un documento para luego dejar un excel armado con los distintos recursos, las acciones, las rutas y los modelos de entrada y salida. Una vez tuvimos los endpoints pensados y las distintas clases identificadas con sus respectivas dependencias, nos ocupamos de organizar las primeras pruebas siguiendo TDD para las clases del dominio. El flujo de trabajo fue

Creemos importante definir algunos de los términos comentados anteriormente ya que son herramientas definidas en el curso actual. Entendemos por endpoint una dirección de una API que se encarga de manipular distintas respuestas o solicitudes HTTP vistas en el curso. Queda implícito entonces que utilizaremos el protocolo de capa 7 del modelo OSI ligado con el estilo arquitectónico REST. Para cumplir con los distintos principios, exponemos a nuestra API a manejar diversos verbos HTTP manteniéndola así con un estilo RESTful. Los verbos utilizados para el desarrollo fueron GET, POST, PUT y DELETE.

Diagrama general de paquetes

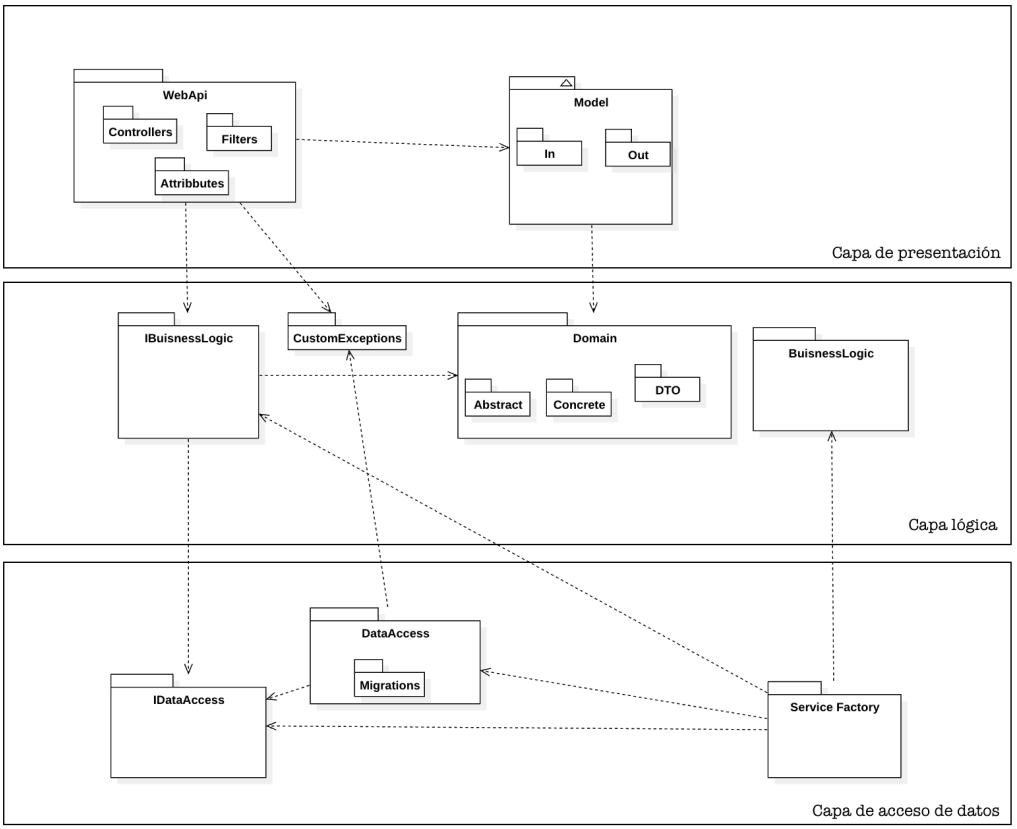
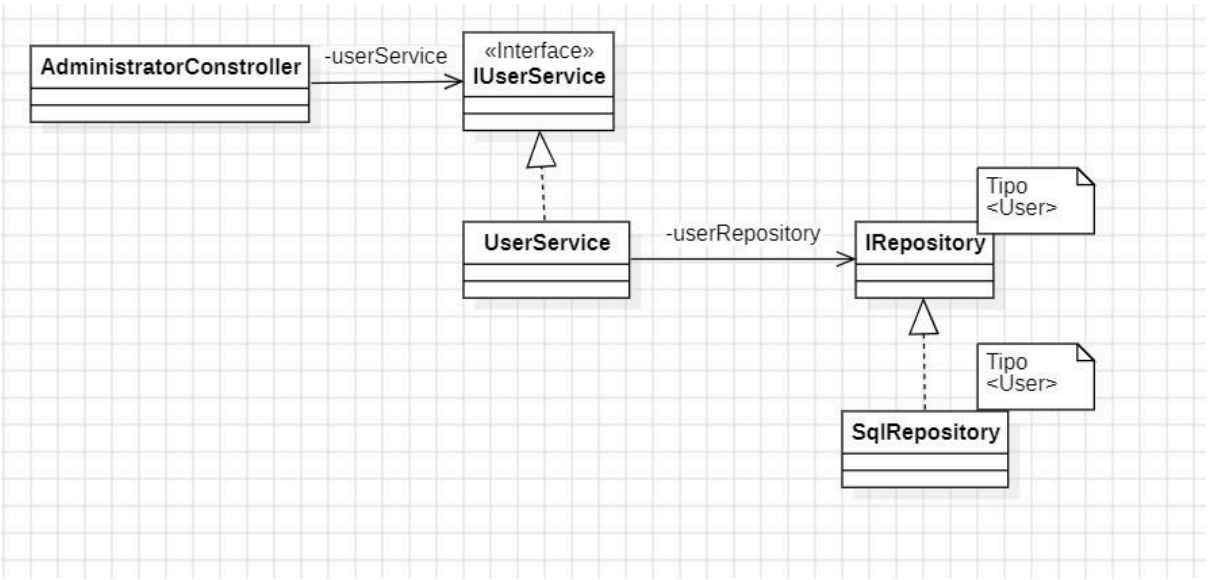
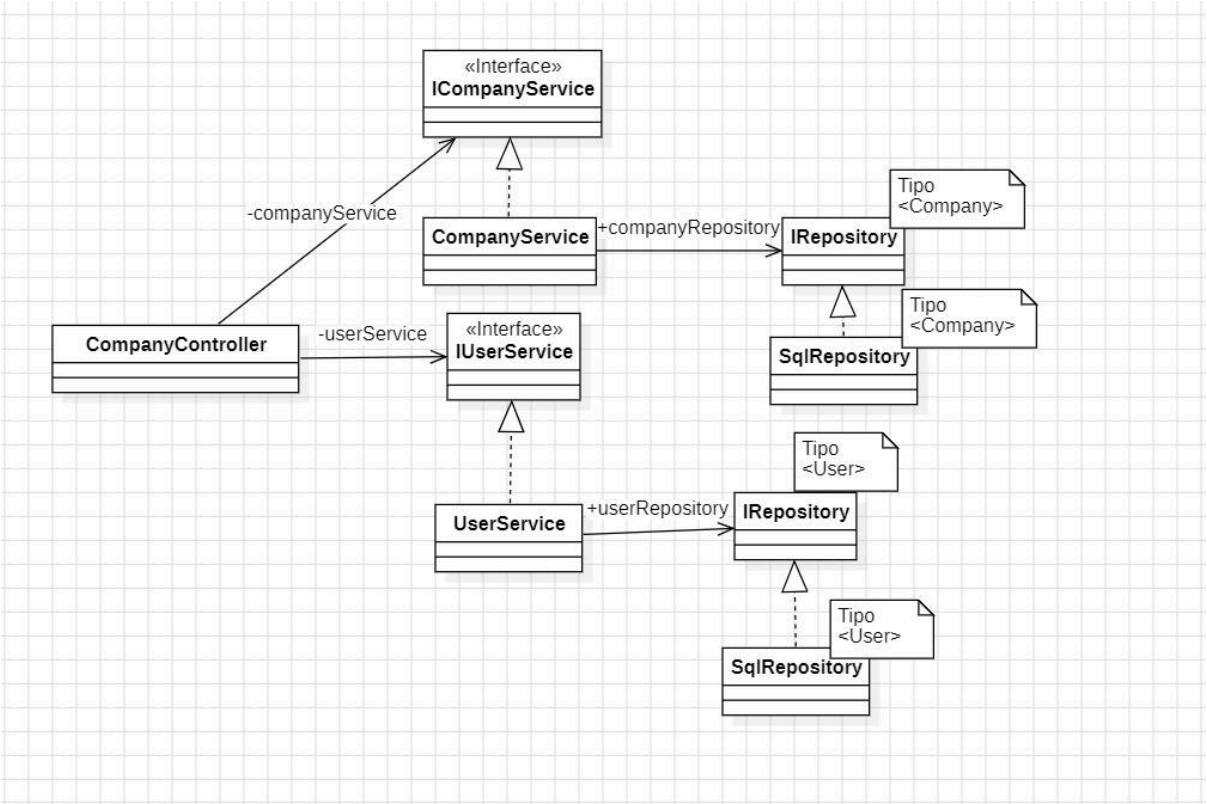
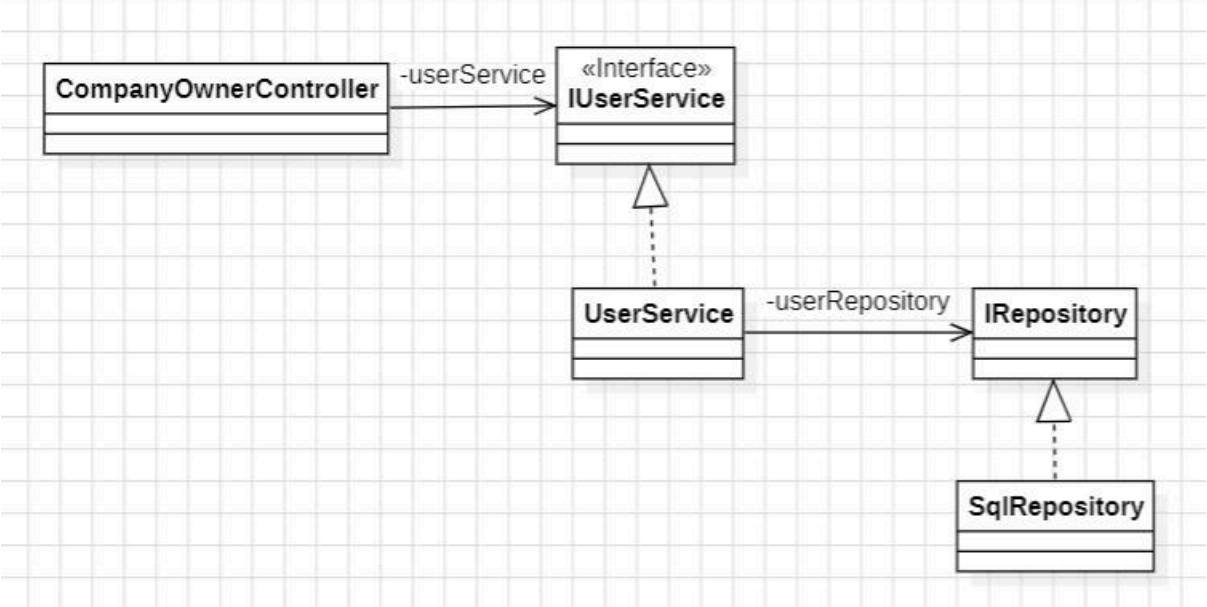
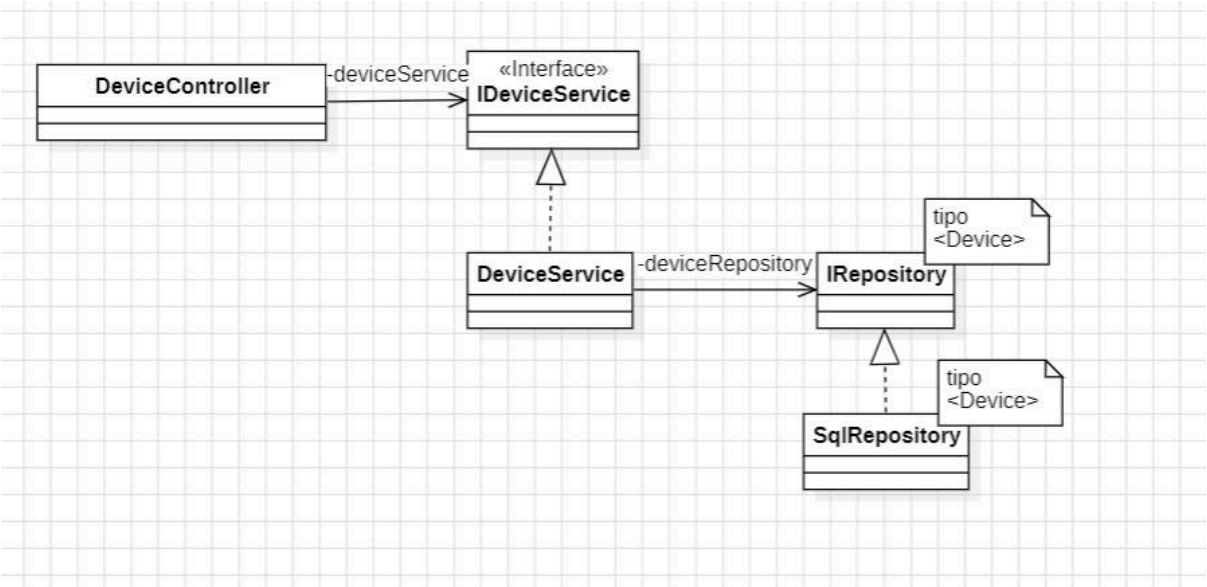


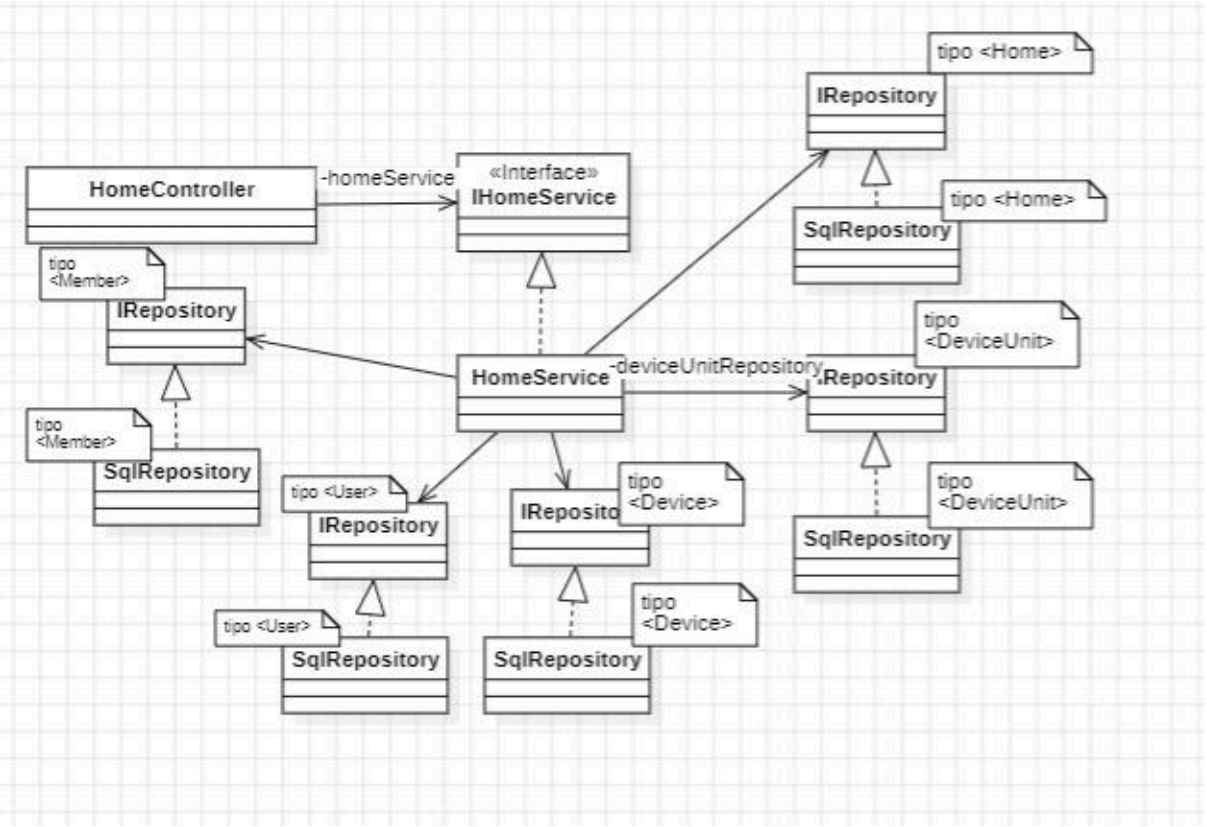
Diagrama de clases de controller (general)

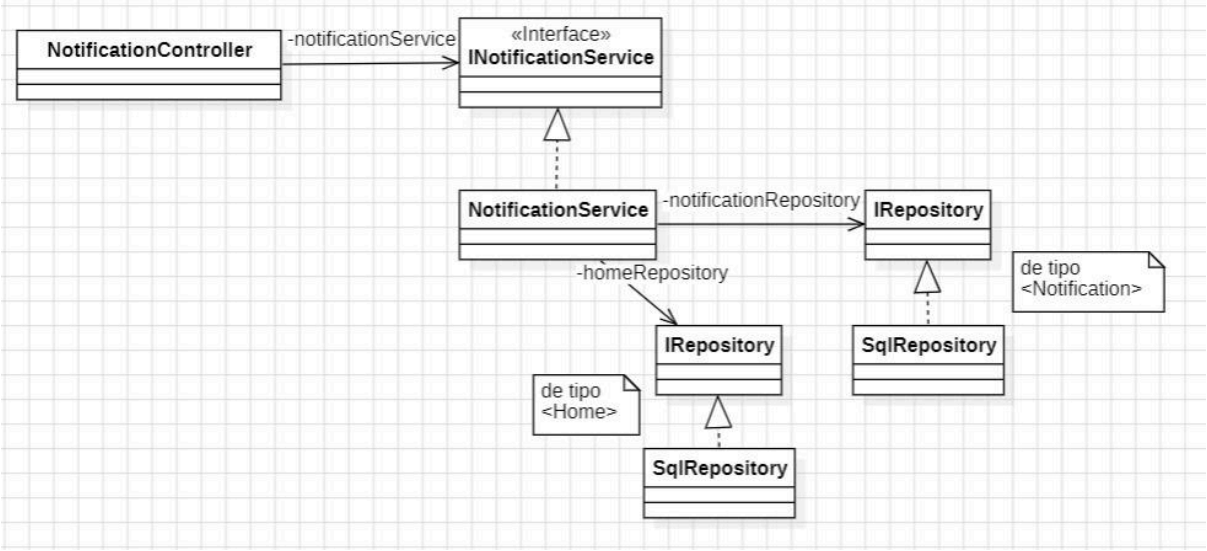


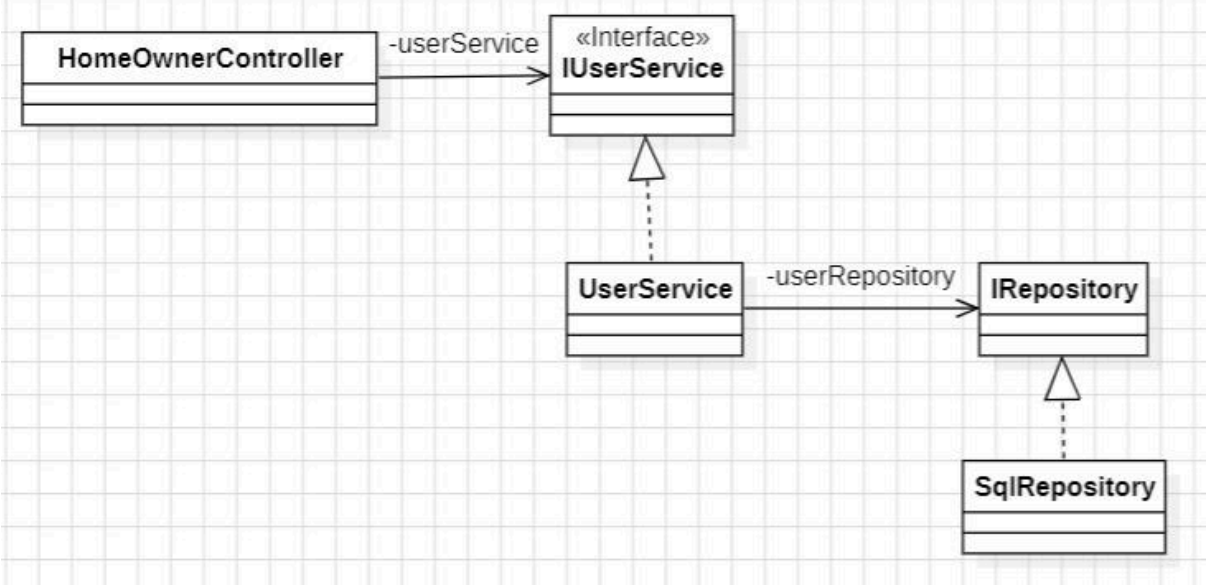

«Interface»
ICompanyService
-companyService
CompanyService
+companyRepository
IRepository
Tipo
<Company>
CompanyController
-userService
«Interface»
IUserService
SqlRepository
Tipo
<Company>
Tipo
<User>
UserService
+userRepository
IRepository
Tipo
<User>
SqlRepository

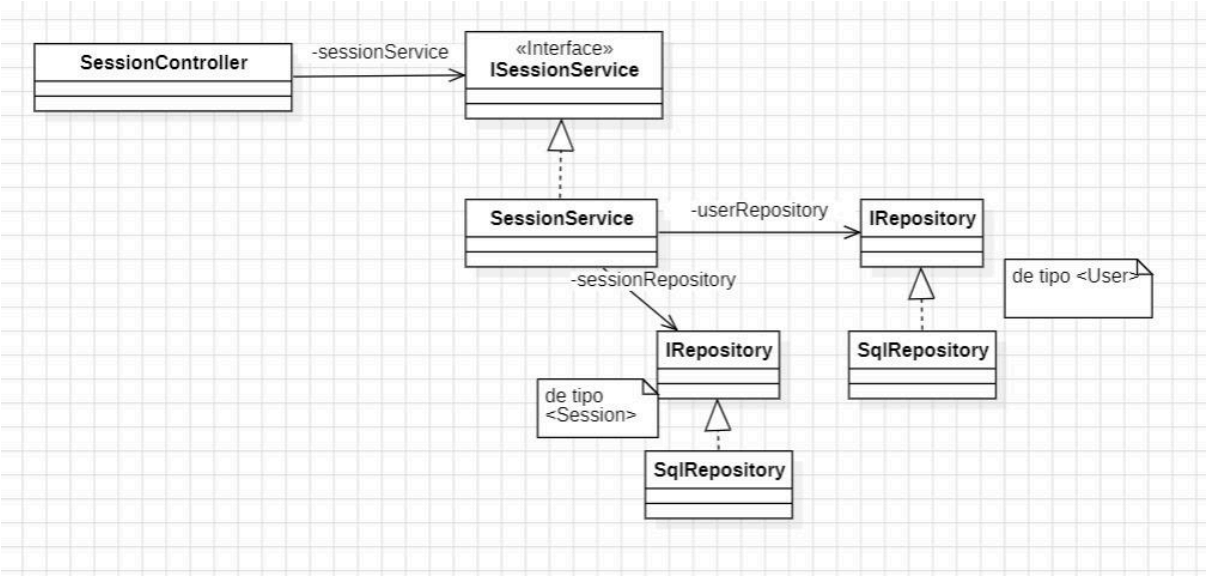

CompanyOwnerController
-userService
«Interface»
IUserService
UserService
-userRepository
IRepository
SqlRepository


DeviceController
-deviceService
«Interface»
IDeviceService
DeviceService
-deviceRepository
IRepository
tipo
<Device>
SqlRepository
tipo
<Device>


HomeController
-homeService
«Interface»
IHomeService
HomeService
-deviceUnitRepository
IRepository
tipo <Home>
SqlRepository
tipo <Home>
tipo
<Member>
IRepository
tipo
<Member>
SqlRepository
tipo
<DeviceUnit>
Repository
tipo
<DeviceUnit>
SqlRepository
tipo <User>
IRepository
IReposito
tipo
<Device>
tipo <User>
SqlRepository
SqlRepository
tipo
<Device>


NotificationController
-notificationService
«Interface»
INotificationService
NotificationService
-notificationRepository
IRepository
-homeRepository
de tipo
<Notification>
IRepository
SqlRepository
de tipo
<Home>
SqlRepository


HomeOwnerController
-userService
«Interface»
IUserService
UserService
-userRepository
IRepository
SqlRepository



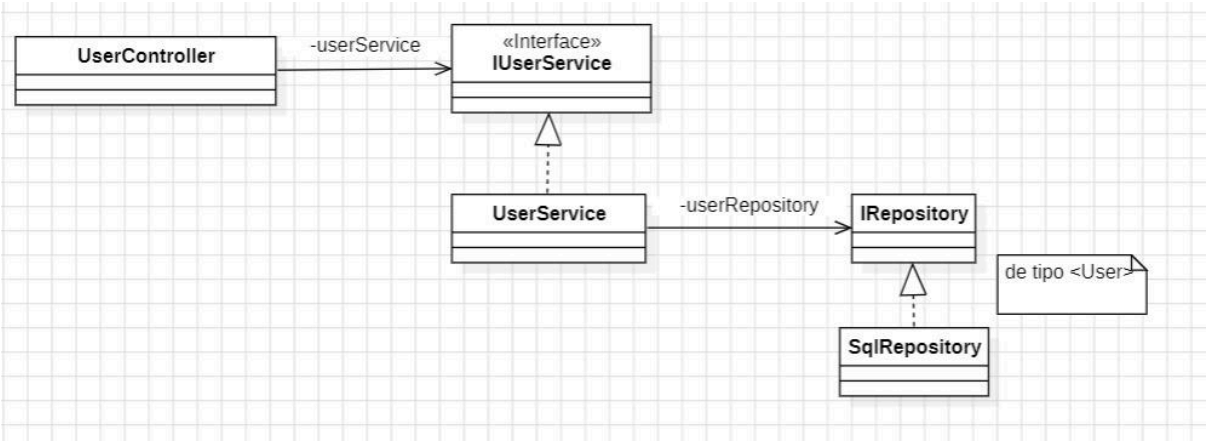
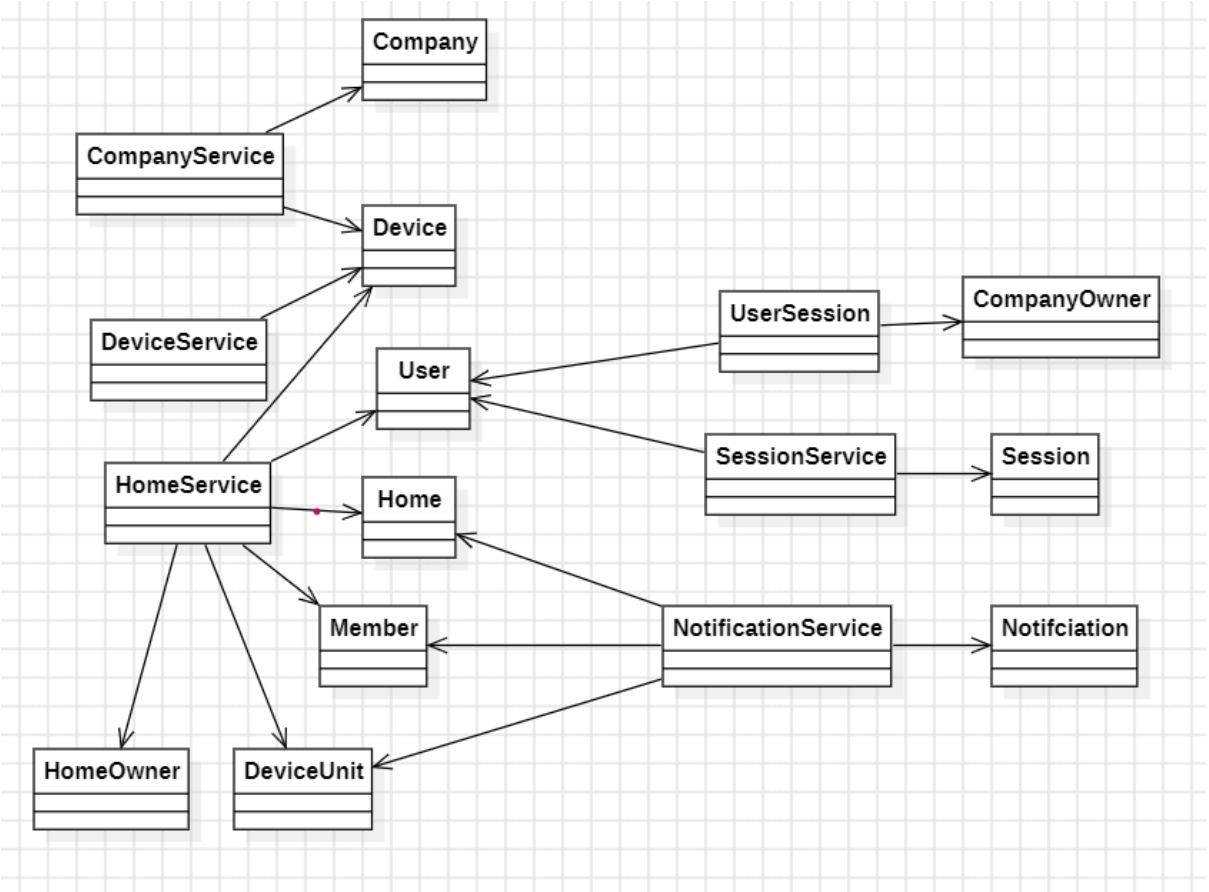


Diagrama de clases y dominio

Relación Services con Domain

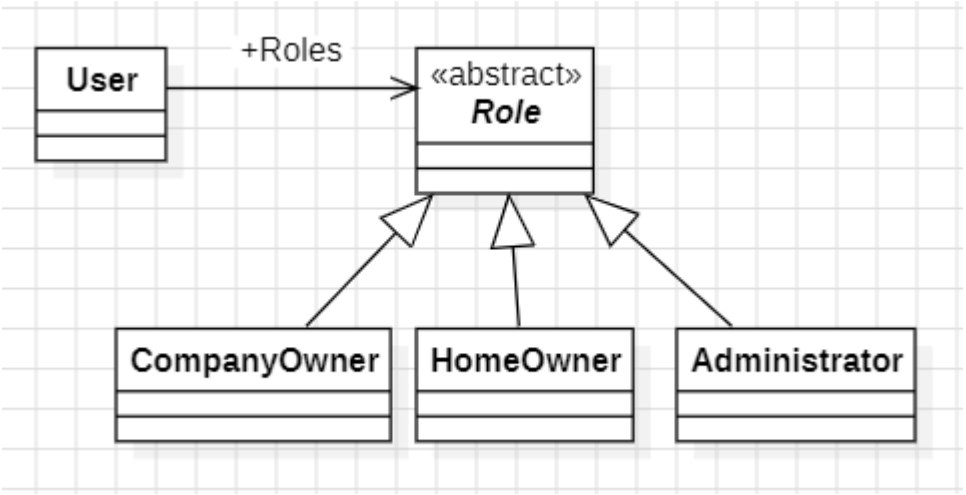


Se deja el link con el diagrama de clases con atributos y métodos, más detallado.

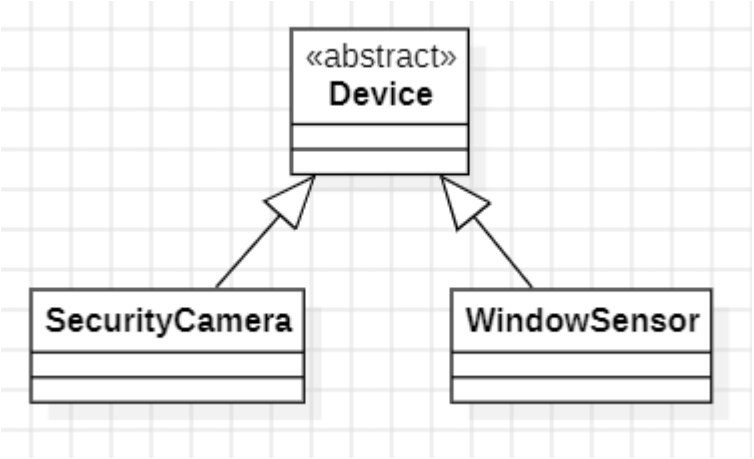
https://drive.google.com/drive/folders/1IFdQXejgw7j3gp98NeOBxo_XbPwXAb22?usp=drive_link

Descripción de jerarquías de herencia utilizadas

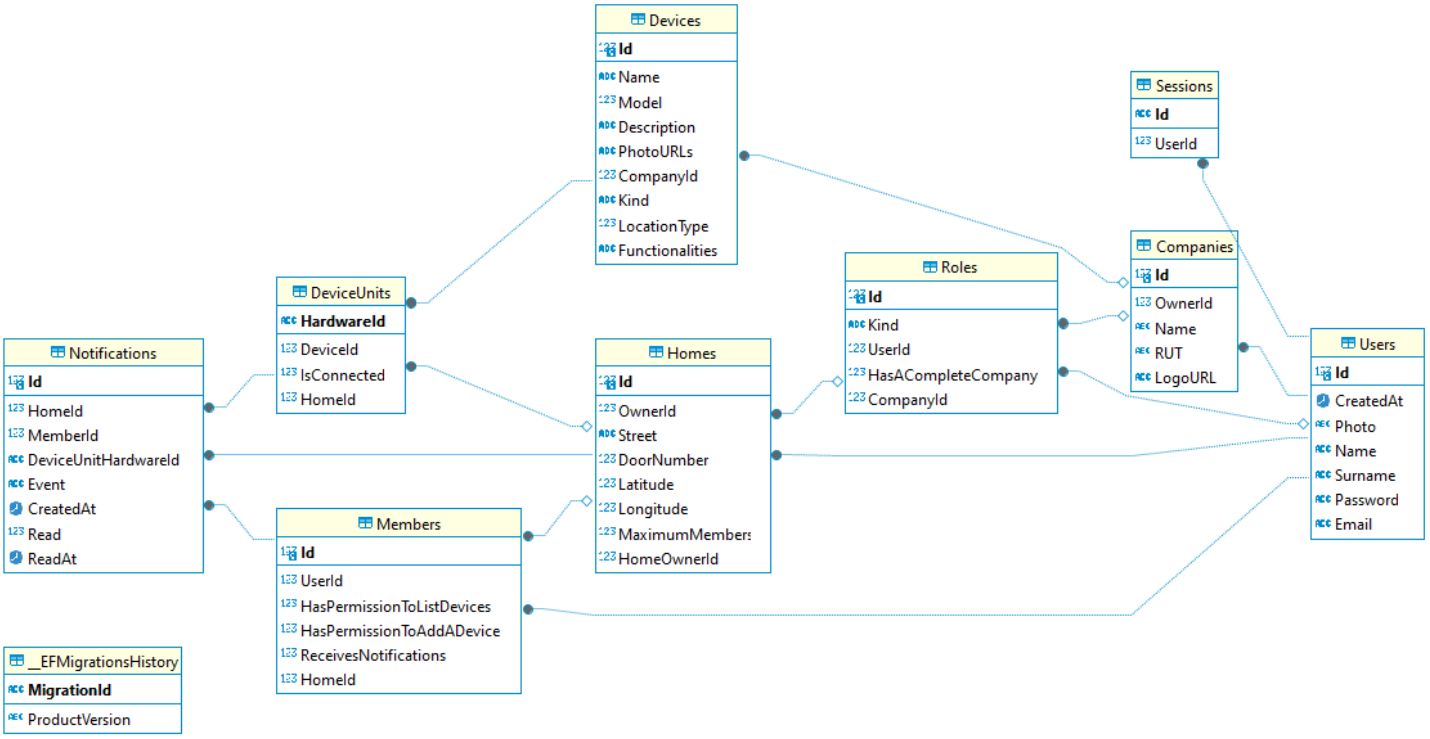
- **Roles:** Se implementó una relación de herencia en la clase Role para facilitar la adición de nuevos roles en el futuro sin necesidad de modificar el código existente. Esto permite extender la funcionalidad simplemente creando nuevas clases que hereden de Role, manteniendo el código flexible y escalable. Para más detalles, consulta la sección "Justificación del diseño explicando mediante texto y diagramas en la Implementación de la clase Role."



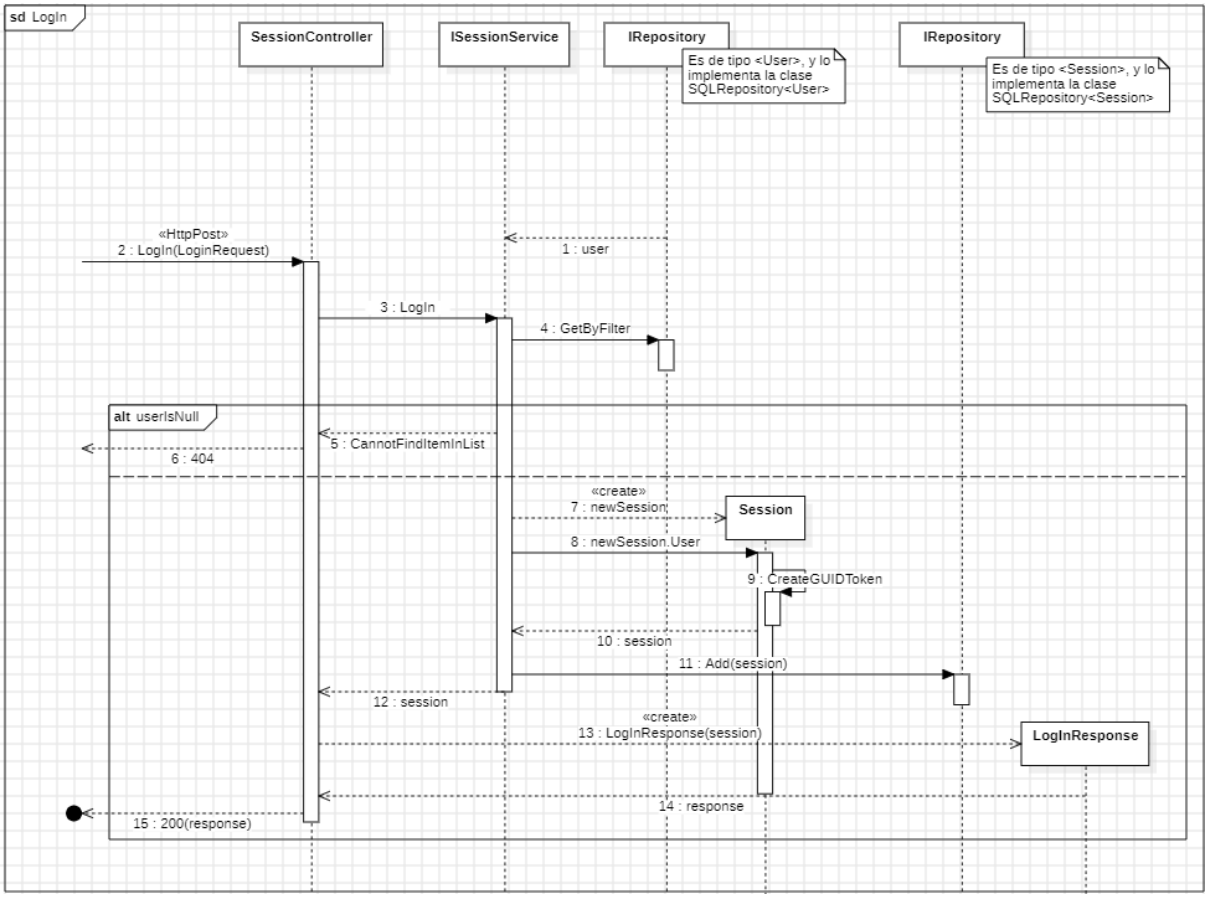
- **Devices:** Se implementó una relación de herencia para gestionar de manera más eficiente los diferentes tipos de dispositivos, facilitando la extensión del sistema al permitir la adición de nuevos dispositivos en el futuro sin modificar el código existente. Este diseño sigue el principio de abierto/cerrado (OCP) de SOLID, donde las clases están abiertas a la extensión pero cerradas a la modificación. Además, centralizar la gestión de los dispositivos bajo una estructura común simplifica el manejo, ya que una casa puede contener una lista de dispositivos sin necesidad de tratarlos de forma individual, evitando duplicación de código y manteniendo el diseño más limpio y manejable.

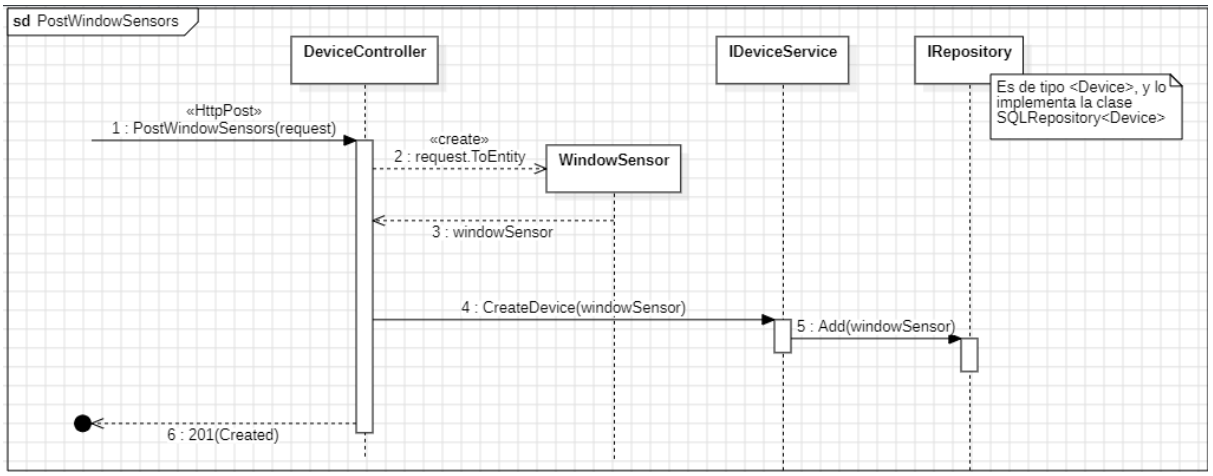


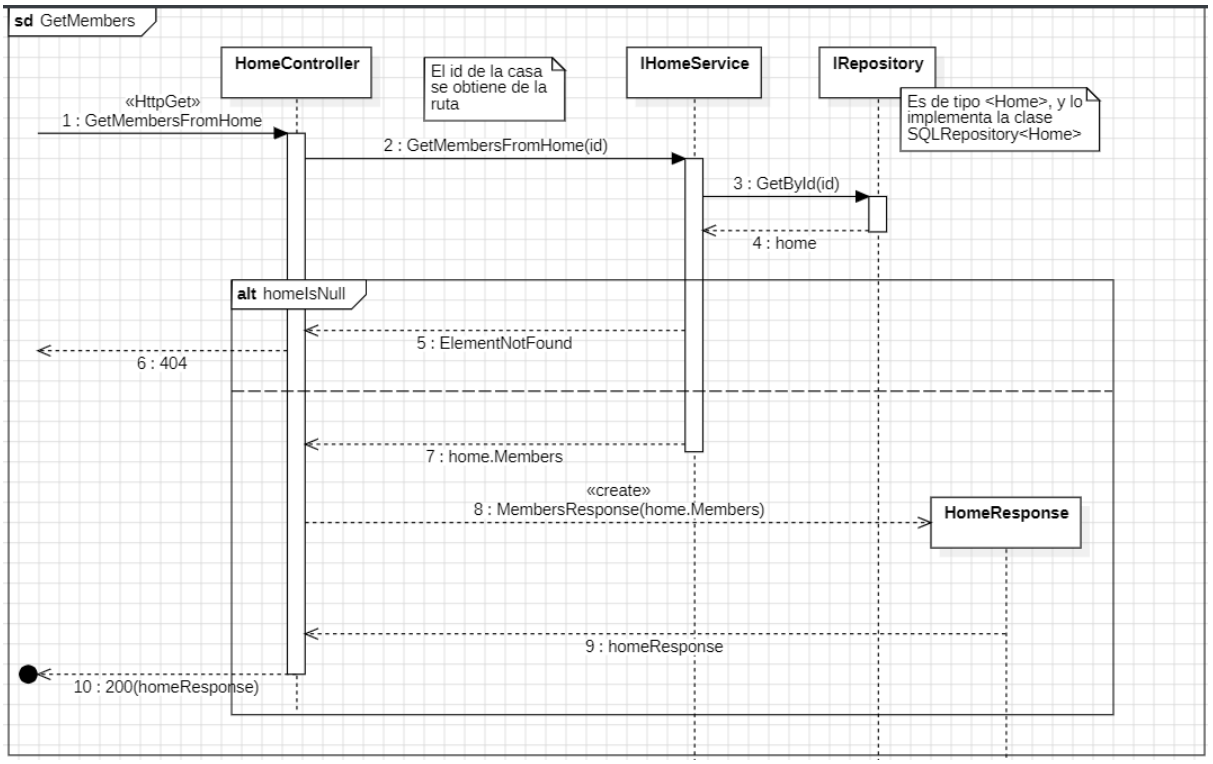
Modelo de tablas de la estructura de la base de datos



Diagramas de interacción







Justificación del diseño explicando mediante texto y diagramas

Inyección de dependencias

Decidimos usar el patrón de diseño de Inyección de Dependencias para mejorar la modularidad del código, evitar dependencias entre el proyecto WebApi y clases concretas como las definidas en Domain, DataAcces o BusinessLogic, y reutilizar las instancias de repositorios y servicios en nuestra API.

La Inyección de Dependencias es un enfoque que permite a una clase mantener referencias a sus dependencias sin tener que construirlas ella misma. En lugar de eso, las dependencias se "inyectan" en la clase (generalmente a través del constructor) en tiempo de ejecución. Nos apoyamos en este patrón porque permite desacoplar los objetos y sus dependencias, lo que facilita la sustitución y la reutilización de estos componentes. Definimos nuestros repositorios y servicios en un único lugar, y estos son utilizados en los distintos controladores y filtros de WebApi.

Los servicios son clases que contienen lógica de negocio y los repositorios abstraen operaciones de acceso a datos. Al inyectar estos servicios, también facilitamos la sustitución de una implementación de servicio por otra, lo que es útil para diseñar nuestras pruebas unitarias y asegurarnos de mantener una buena separación de responsabilidades.

Los servicios y repositorios se registran en el contenedor de Inyección de Dependencias en la clase ServiceFactory. Cada servicio o repositorio se registra con una vida útil específica, que determina

cuándo se crea y se destruye la instancia del servicio o repositorio. Por ejemplo, si un servicio se registra con la vida útil "Scoped", se creará una nueva instancia del servicio para cada solicitud HTTP.

Patrones y principios de diseño

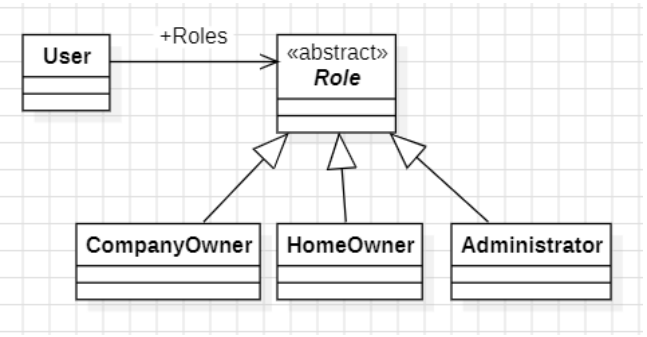
Inversión de Dependencias

Nos aseguramos de que las capas de nuestra aplicación dependieran de interfaces en lugar de implementaciones concretas siguiendo el principio de Inversión de Dependencias, uno de los cinco principios SOLID. Este principio establece que las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones. En nuestra arquitectura, esto se traduce en que la lógica de negocio y otros módulos de alto nivel interactúan únicamente con interfaces, como IRepository<T>, en lugar de clases concretas como SqlRepository<T>.

Clase abstracta Role

Dado que la aplicación tiene diferentes funcionalidades dependiendo del tipo de usuario, decidimos implementar la clase Role. Esta es una clase abstracta que permite la creación de diferentes roles mediante la herencia, lo que facilita la adición de nuevos roles en el futuro. Con este enfoque, simplemente tendríamos que crear una nueva clase que herede de Role, evitando así la modificación del código existente.

Este diseño cumple con el principio de Abierto/Cerrado (OCP) del principio SOLID, que establece que las clases deben estar abiertas para su extensión pero cerradas para su modificación. Al permitir la creación de nuevos roles sin alterar el código base, garantizamos una mayor flexibilidad y mantenibilidad en la aplicación.



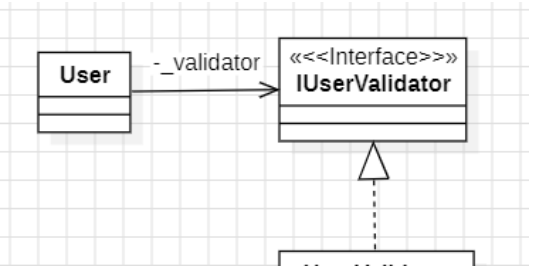
Además, la clase Usuario contiene una lista de roles, lo que permite que un usuario tenga múltiples roles, como HomeOwner y CompanyOwner. Esto se alinea con el principio de Responsabilidad Única (SRP), ya que la clase Usuario se encarga de gestionar la información del usuario y su asociación con varios roles, mientras que la clase Role se enfoca únicamente en la definición de los roles.

Al implementar este sistema de roles, también estamos facilitando la Inversión de Dependencias (DIP) al permitir que las funcionalidades de la aplicación dependan de abstracciones (Role) en lugar de implementaciones concretas. Esto no solo mejora la cohesión del sistema, sino que también facilita la implementación de pruebas unitarias, ya que podemos simular diferentes roles sin necesidad de instancias concretas.

Interfaz IUserValidator

Otro de los aspectos que consideramos propensos a cambios son las formas de validación, como la determinación de la validez de un correo electrónico o una contraseña. Por lo tanto, implementamos una interfaz IUserValidator, lo que permite que el diseño cumpla con los principios GASP y SOLID.

Creamos la clase UserValidator, que implementa esta interfaz, permitiendo la validación de atributos de



usuario de manera flexible. Este enfoque no solo facilita la creación de diferentes tipos de validadores en el futuro sin necesidad de modificar el código existente, sino que también promueve el principio de Abstracción, ya que la lógica de validación se abstrae de la clase User.

Además, la clase User incluye un atributo de tipo IUserValidator, asegurando que la responsabilidad de validar los atributos de usuario no recaiga en la propia clase User, sino en el validador correspondiente. Esto se alinea con el principio de Responsabilidad Única (SRP), ya que cada clase se encarga de una única responsabilidad: User gestiona la información del usuario, mientras que UserValidator se encarga de la validación.

Uso de Models en la API

Models: En nuestra aplicación, decidimos utilizar models como una parte fundamental de nuestra API REST. Estos models nos permiten representar de manera clara los datos que se intercambian entre el cliente y la base de datos así como ser intencional en los datos que devolvemos al cliente en lugar de devolver objetos con todas sus propiedades expuestas.

Al utilizar models, también simplificamos la interacción con los servicios y el dominio. En lugar de tener que es pasar instancias de clases del dominio o trabajar directamente con los repositorios para consultar datos a la base de datos, podemos trabajar directamente con objetos a partir de los cuales se pueden generar filtros para consultas a la base de datos, o pasarlos a objetos del dominio, lo cual hace que el desarrollo sea más ágil y menos propenso a errores. Esto también nos ayuda a no tener que depender del dominio en WebApi, ya que tenemos a Model como intermediario.

Descripción de los Proyectos y Namespaces

1. **BusinessLogic**:
    - **Descripción**: Este proyecto contiene la lógica de negocio de la aplicación. Aquí se definen los servicios que manejan las operaciones principales del sistema
    - **Razón de la Separación**: Al mantener la lógica de negocio separada de otras capas, como el acceso a datos y la presentación, se promueve una mayor claridad y organización. Esto facilita el mantenimiento y las pruebas, ya que los cambios en la lógica de negocio no afectan a otras partes de la aplicación.

2. **CustomExceptions**:
    - **Descripción**: Este proyecto se encarga de definir excepciones personalizadas que se utilizan en toda la aplicación. Estas excepciones permiten manejar errores de manera más específica y descriptiva.
    - **Razón de la Separación**: Al tener un namespace dedicado para las excepciones, se puede centralizar el manejo de errores y facilitar la reutilización de excepciones en diferentes partes de la aplicación, cumpliendo con principios de diseño limpio y promoviendo la coherencia.

3. **DataAccess**:
    - **Descripción**: Este proyecto incluye las clases y la lógica necesarias para acceder a la base de datos. Aquí se implementa la interfaz **IRepository<T>** con la clase **SqlRepository<T>** para la manipulación de datos utilizando Entity Framework.
    - **Razón de la Separación**: Al aislar la capa de acceso a datos, se mejora la capacidad de cambiar la implementación de la base de datos sin afectar la lógica de negocio.

4. **Domain**:
    - **Descripción**: Este proyecto define las entidades del dominio, que son las clases que representan los conceptos fundamentales de la aplicación, como User, Device, etc.
    - **Razón de la Separación**: Al tener un namespace dedicado para el dominio, se asegura que las entidades estén claramente definidas y accesibles para las capas superiores.

5. **IBusinessLogic**:
    - **Descripción**: Este proyecto define las interfaces que los servicios de la lógica de negocio deben implementar. Esto permite que la implementación concreta de los servicios sea intercambiable.
    - **Razón de la Separación**: Al definir interfaces, se promueve la programación orientada a interfaces, lo que facilita la creación de pruebas unitarias y la implementación de diferentes versiones de los servicios sin alterar el resto de la aplicación.

6. **IDataAccess**:
    - **Descripción**: Este proyecto contiene las interfaces para el acceso a datos, como **IRepository<T>**.
    - **Razón de la Separación**: Al tener un espacio de nombres dedicado para las interfaces de acceso a datos, se promueve una mejor abstracción y desacoplamiento entre la lógica de negocio y la implementación de acceso a datos.

7. **Model**:
    - **Descripción**: Este proyecto incluye los modelos de datos que se utilizan en la aplicación.
    - **Razón de la Separación**: Separar los modelos de datos facilita su gestión y permite mantener una clara distinción entre las entidades del dominio y los modelos que se utilizan para la presentación y manipulación de datos.

8. **ServiceFactory**:
    - **Descripción**: Este proyecto implementa la lógica para instanciar servicios en la aplicación.
    - **Razón de la Separación**: Centralizar la creación de instancias de servicio permite cambiar la lógica de creación en un solo lugar y facilita la inyección de dependencias.

9. **TestDataAccess, TestDomain, TestService, TestWebApi**:
    - **Descripción**: Estos proyectos contienen las pruebas unitarias para las distintas capas de la aplicación.
    - **Razón de la Separación**: Al tener un espacio de nombres específico para las pruebas, se mantiene la organización del código y se facilita la ejecución y mantenimiento de las pruebas. Esto permite que las pruebas sean claras y estructuradas, lo que es fundamental para asegurar la calidad del software.

10. **WebApi**:
    - **Descripción**: Este proyecto es responsable de la implementación de la API REST de la aplicación. Aquí se definen los controladores y las rutas para manejar las solicitudes HTTP.
    - **Razón de la separación**: Mantener la API en un proyecto separado ayuda a desacoplar la lógica de presentación de la lógica de negocio. Esto permite que la API sea escalable y fácil de mantener, y facilita la creación de clientes que interactúan con ella.

Descripción del mecanismo de acceso a datos utilizado

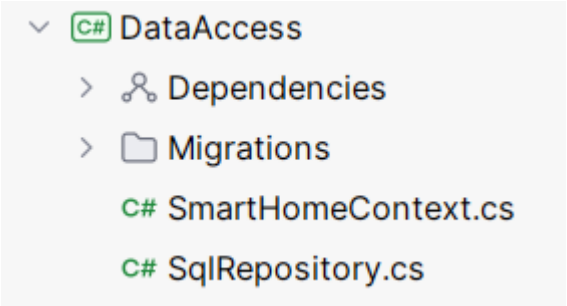
En nuestro proyecto, decidimos implementar una interfaz llamada IRepository<T>, que define las firmas para los métodos básicos de manipulación de datos (CRUD) para una entidad o clase genérica T. Esta decisión permite que todos los servicios (services), **que se ubican en el proyecto BussinesLogic,** se comuniquen con un objeto de tipo IRepository<T>, desconociendo cómo se implementan estos métodos en la práctica. Este enfoque contribuye a mantener un diseño limpio y modular, alineado con los principios SOLID, donde las capas superiores dependen de abstracciones y no de implementaciones concretas.

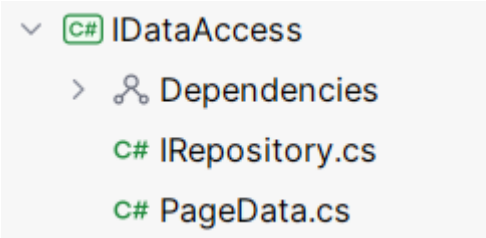
Para cumplir con los requisitos de la entrega que exigían el uso de una base de datos SQL, creamos la clase SqlRepository<T>, que se encuentra en el proyecto DataAccess, que implementa la interfaz IRepository<T>. Esta clase lleva a cabo la implementación de todos los métodos de la interfaz utilizando Entity Framework, facilitando así la interacción con la base de datos. Para conectar la clase al contexto de la base de datos, pasamos el SmartHomeContext como parámetro en el constructor de SqlRepository<T>. Esto nos permite utilizar todas las funcionalidades que ofrece Entity Framework, como la gestión de conexiones y el seguimiento de cambios en las entidades.

Además, implementamos migraciones para gestionar los cambios en el esquema de la base de datos a lo largo del tiempo. Esto nos permite actualizar la base de datos de manera controlada y reproducible, manteniendo la integridad de los datos.

Un aspecto clave de nuestro diseño es que solo los servicios se comunican directamente con IRepository<T>. Esto asegura que las partes más abstractas del código no dependan de implementaciones concretas de acceso a datos, sino que interactúen únicamente con las abstracciones definidas por la interfaz. De esta manera, si en el futuro decidimos cambiar el almacenamiento de datos, por ejemplo, a un bucket de S3, solo sería necesario implementar una nueva clase S3Repository<T> y sustituir las instancias de SqlRepository<T> por S3Repository<T> en los servicios y pruebas. Esto proporciona una gran flexibilidad y permite que el sistema evolucione sin necesidad de realizar cambios drásticos en su estructura.

Finalmente, adoptamos un enfoque Code First para el desarrollo del modelo de datos, lo que significa que primero definimos nuestras entidades en código y luego generamos la base de datos a partir de esas definiciones. Este enfoque nos permite mantener un control preciso sobre la estructura de nuestro modelo y facilita la evolución del mismo a medida que avanza el desarrollo del proyecto.





Descripción del manejo de excepciones

En nuestro proyecto, creamos un módulo separado llamado **CustomExceptions** para gestionar las excepciones de manera centralizada. Dentro de este módulo, diseñamos nuestras propias excepciones personalizadas, asegurándonos de que fueran lo más generales posible para permitir su reutilización en diferentes partes del código. De esta forma, evitamos la redundancia y promovemos un código más limpio y mantenible, siguiendo los principios de **Clean Code**.

Además, cada excepción personalizada está diseñada para recibir un mensaje descriptivo que proporciona más contexto sobre el motivo del error. Esto facilita la depuración y mejora la comprensión del flujo de errores dentro de la aplicación, ya que el desarrollador puede identificar claramente qué ocurrió y por qué falló el proceso.

Como parte de nuestra implementación de una API REST, hemos estructurado el manejo de excepciones en los controladores (controllers) de manera eficiente. En cada uno de los endpoints, utilizamos bloques try-catch para capturar las excepciones que pueden ser lanzadas por los métodos de los servicios (services). Esto nos permite gestionar cualquier error que ocurra durante la ejecución de las operaciones, asegurando que el sistema no se detenga abruptamente y que los errores sean manejados de manera controlada.

Para más información sobre los mensajes de error y endpoints, se explica en el documento “Evidencia del diseño y especificación de la API”.

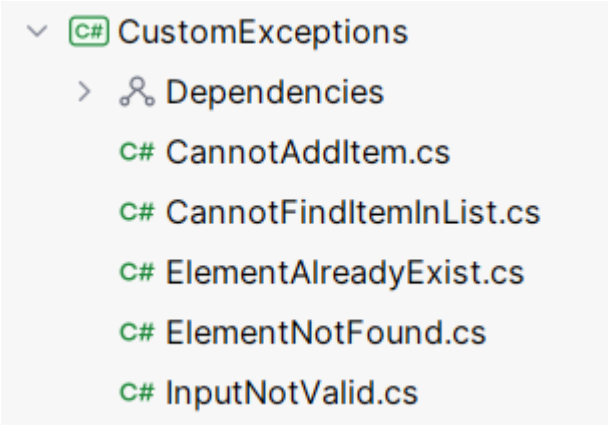
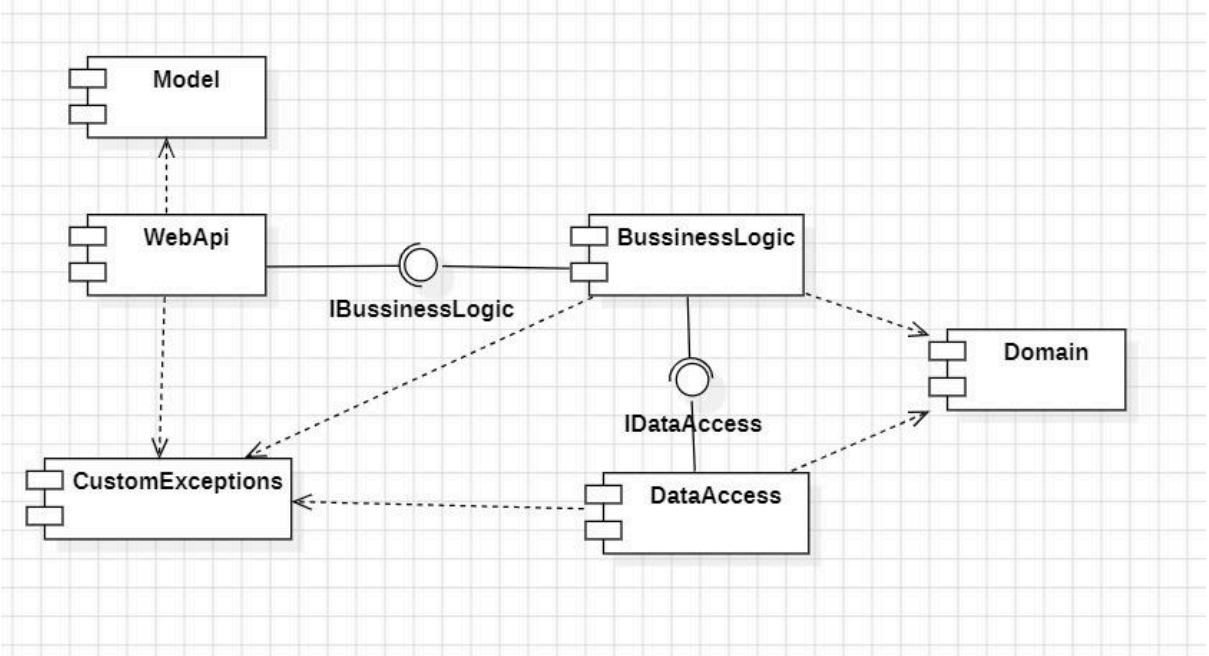


Diagrama de implementación



Decidimos estructurar nuestra aplicación para lograr una arquitectura limpia y desacoplada separando la WebApi del dominio mediante Models de modo que la API no dependa de las implementaciones del negocio y sea más fácil de mantener y extender.

Utilizamos interfaces como IBusinessLogic e IDataAccess para conectar las capas lo que nos da flexibilidad para cambiar implementaciones sin afectar el resto del sistema y facilita la creación de pruebas unitarias.

Además centralizamos las excepciones en CustomExceptions para manejar los errores de manera uniforme con mensajes específicos de nuestro negocio lo que mejora el control de errores y la depuración

Cada capa tiene un rol bien definido: la **WebApi** maneja la comunicación externa, la lógica de negocio se encapsula en **BusinessLogic**, y el acceso a datos se gestiona a través de **DataAccess**. Además, el uso de interfaces como **IBusinessLogic** e **IDataAccess** asegura un bajo acoplamiento entre las capas, permitiendo flexibilidad en la implementación y facilitando las pruebas unitarias. La

inclusión de **CustomExceptions** centraliza el manejo de errores, garantizando un control uniforme y eficiente de las excepciones específicas del proyecto.