Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1 Diseño de Aplicaciones 2

Enzo Izquierdo (283145)

Manuel Graña (285727)

Martín Salaberry (294238)

Repositorio

2024

Índice

Descripción del diseño

Introducción

HomeConnect es una plataforma digital diseñada para centralizar y gestionar diferentes dispositivos inteligentes en el hogar, creando un entorno integrado y automatizado. Este sistema permite a los usuarios, con distintos roles, gestionar y controlar dispositivos de forma eficiente.

Funcionalidades Implementadas

Se implementaron todas las funcionalidades previstas para esta entrega.

- Registro y gestión de usuarios
- Administración de dispositivos y cuentas de empresas
- Listado y filtrado de cuentas, empresas y dispositivos
- Creación y gestión de hogares, miembros y notificaciones
- Registro de dispositivos inteligentes como cámaras y sensores
- Autenticación de usuarios
- Configuración de notificaciones y permisos para miembros

Instalación

La base de datos del proyecto corre en un contenedor de Docker, por lo que lo primero es configurarlo. Iremos al archivo “Aplicacion/docker-compose.yml” en el directorio raíz y lo podremos configurar. Hay que indicar la contraseña en **SA_PASSWORD** y el puerto en **ports**. Por defecto son *“Passw1rd”* y *“1433”*, respectivamente.

Luego debemos configurar los connection strings para poder realizar la conexión a este. En la carpeta Aplicacion, en el archivo appsettings.json configuraremos el string para la versión de producción. Buscaremos la línea “DefaultConnection”. Aquí podremos configurar la IP (Server), nombre de la base de datos (Database), User Id y Password, esta última debe ser la misma que utilizamos en el archivo “/docker-compose.yml”.

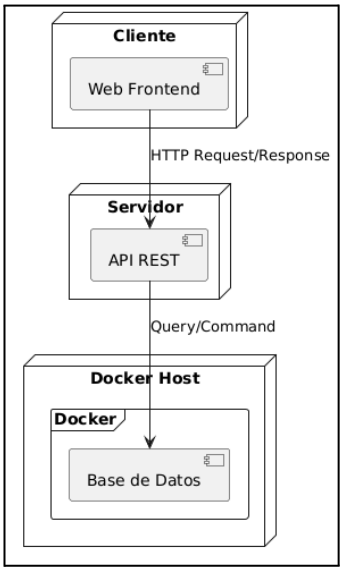
Por último, debemos ejecutar HomeConnect.WebApi.exe y la API ya se estará ejecutando y escuchando peticiones en el puerto y la dirección URL especificada en consola.

Ubicación de los scripts SQL con datos de prueba: Se incluyen los backups de una base de datos sin ningún dato adicional (solo con roles, permisos y cuenta de administrador) así como un backup de la base de datos con datos de prueba precargados.

Cuenta de administrador por defecto (con la que se puede empezar a registrar dueños de empresa u otros administradores):

- **Administrador:** *admin@admin.com* - contraseña: *Admin123@*

Vista de Deployment



Descripción de la Arquitectura física

La arquitectura física de la aplicación se basa en 3 elementos, donde los componentes están diseñados para ser independientes y escalables. En este sistema, la API REST actúa como el núcleo de la aplicación, gestionando las solicitudes y respuestas entre el cliente y la base de datos. Esta API está alojada en un servidor dedicado, mientras que la base de datos se despliega en un contenedor Docker, lo que permite un manejo eficiente y cohesivo de los recursos y una fácil escalabilidad.

Componentes de la Arquitectura

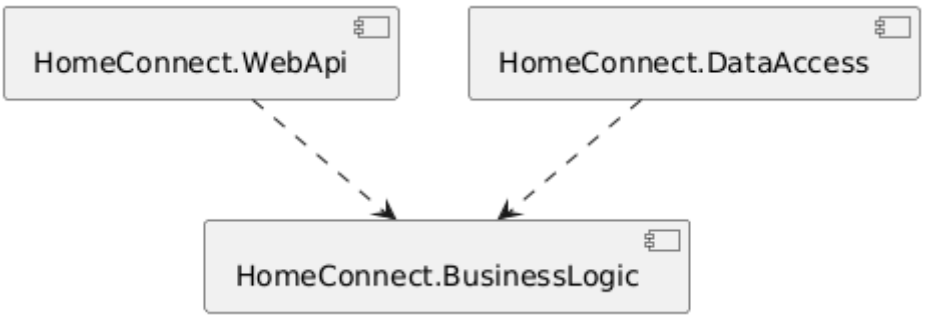
1. **Cliente Web Frontend**:
    - Aunque el cliente es una parte crucial de la arquitectura, actualmente no está implementado. La intención es que el cliente sea una aplicación web que se comunique con la API REST a través de solicitudes HTTP. En esta fase, solo se exponen los endpoints de la API, permitiendo interactuar con ella.
2. **API REST**:
    - La API REST es el intermediario entre el cliente y la base de datos. Implementa endpoints que permiten realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos.
3. **Base de Datos**:
    - La base de datos está alojada en un contenedor Docker, lo que facilita su gestión y escalabilidad. Este enfoque permite que la base de datos se ejecute

en un entorno aislado, minimizando conflictos y asegurando la integridad de los datos.

Vista de Implementación

Nuestra solución se organiza en 3 componentes principales: BusinessLogic, DataAccess y WebApi.

Diagrama de componentes



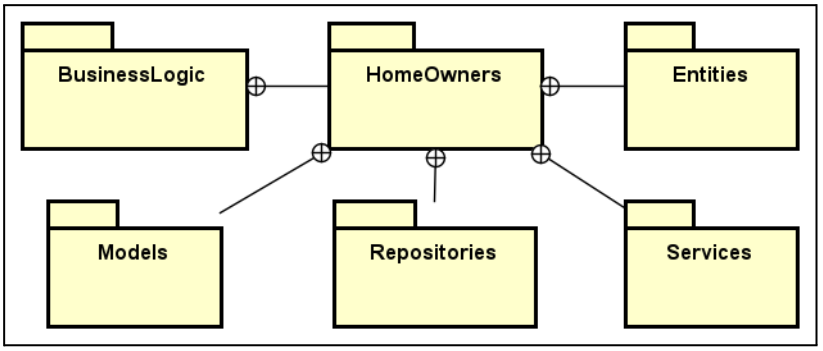
Arquitectura lógica

La solución está dividida en capas de manera que agrupen responsabilidades similares, siguiendo el modelo de **Clean Architecture**. En la **capa de presentación**, donde se manejan las interacciones con el usuario, tenemos a **WebApi** que contiene la API web, junto con sus controladores y los filtros (middleware).

Luego está la **capa de persistencia**, que maneja el acceso a la base de datos, donde tenemos a **DataAccess** que gestiona los repositorios y el contexto de la base de datos. Por último tenemos la **capa de aplicación**, que gestiona la lógica de negocio y el dominio. En esta tenemos a **BusinessLogic** que es donde se encuentran la lógica negocio que abarca los servicios y clases del dominio. Decidimos no separar el dominio en una capa separada porque no nos aportaba ningún beneficio. El dominio solo va a ser usado por nuestra aplicación, no va a ser reusado por otras, además por la transitividad, aunque lo separemos, los proyectos que dependan de BusinessLogic, que depende del dominio, también dependen de este.
Nos decidimos entonces por organizar nuestras clases en **vertical slices**, agrupando en namespaces las clases que colaboren entre sí para cubrir la misma responsabilidad. En la siguiente tabla ilustramos esto.

Namespace	Responsabilidad
BusinessLogic.Admins	Eliminación de administradores. Obtención de lista de usuarios y empresas.
BusinessLogic.BusinessOwners	Creación de empresas Creación de dispositivos
BusinessLogic.Devices	Gestión y validación de datos de Device y Camera Gestión y validación de datos de la relación OwnedDevice
BusinessLogic.HomeOwners	Creación de Hogares. Gestión y listado de miembros de hogar Gestión y listado de dispositivos de hogar
BusinessLogic.Notifications	Creación y asignación de notificaciones Notificar a usuarios que tienen los permisos necesarios para recibir notificaciones
BusinessLogic.Roles	Declaración de las clases de roles y permisos, así como la interfaz del repositorio de roles
BusinessLogic.Users	Creación y validación de datos de User (Administradores, dueños de empresa y dueños de hogar).
DataAccess.Repositories	Implementación de los repositorios declarados como interfaces en los namespaces de BusinessLogic Configuración del contexto de la base de datos
WebApi.Filters	Aplicación de filtros para manejo de autenticación y autorización Manejo de excepciones en las solicitudes
WebApi.Controllers	Gestión de todos los endpoints, cada uno en un namespace llamado de acuerdo a su ruta correspondiente. Ej: WebApi.Controllers.Admin

A su vez, dentro de cada namespace están las interfaces que utilizan (si se cumple el caso que solo ellos la utilicen) tanto de servicios como de repositorios, los servicios que utilizan, las entidades del dominio y los modelos, Dtos que utilizamos para hacer que las funciones no reciban muchos parámetros. HomeOwners, por ejemplo, tiene los 4 posibles subpaquetes dentro.



Independencia de librerías

Para abordar el principio de **independencia de librerías** y minimizar el impacto de los cambios en los componentes físicos de la solución, hemos diseñado la arquitectura en módulos independientes, organizados en **assemblies** separados:

1. BusinessLogic

Este paquete contiene la lógica de negocio y las entidades del dominio, así como las **interfaces de los servicios y de los repositorios**. Aquí se define cómo deben comportarse los servicios y cómo interactúan con los repositorios.

La razón por la cual mantenemos las **interfaces de los repositorios** dentro de BusinessLogic es porque esta capa también contiene las **clases del dominio**, lo cual significa que **DataAccess**, que maneja la persistencia de datos, siempre va a conocer este dominio. No tiene sentido mover estas interfaces a otro assembly, ya que siempre habrá una dependencia hacia BusinessLogic, dado que ambos manejan el dominio.

2. DataAccess

Este paquete es responsable de implementar los repositorios que interactúan directamente con la base de datos. Implementa las interfaces que se definen en **BusinessLogic**, permitiendo así un claro desacoplamiento entre la lógica de negocio y la lógica de persistencia.

Al mantener la implementación en este assembly independiente, permitimos que la lógica de acceso a datos pueda ser modificada o reemplazada sin afectar a otras capas del sistema.

3. WebApi

Aquí es donde se exponen los servicios a través de endpoints. La WebApi depende de las **interfaces de los servicios** que están en **BusinessLogic**. Usamos estas interfaces para hacer la inyección de dependencias y desacoplar la API de las implementaciones concretas de los servicios.

Si bien **WebApi** podría beneficiarse de que las interfaces de los servicios estuvieran en un assembly separado de BusinessLogic, no vemos una justificación clara para hacerlo. Actualmente, no estamos considerando múltiples implementaciones de estos servicios, y la

complejidad añadida de mover las interfaces no parece aportar beneficios significativos en este contexto.

Justificación de cómo se cumple con la independencia de librerías:

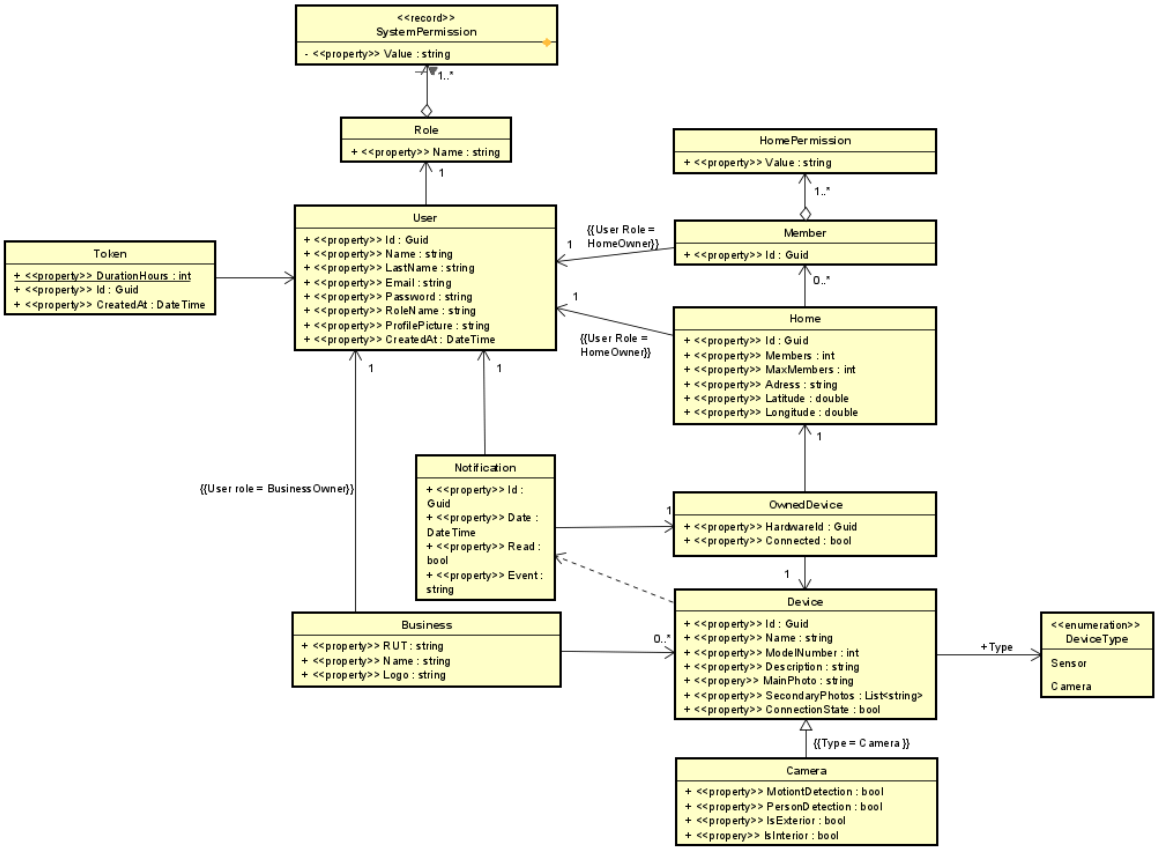
- **Mínimo impacto de cambios**: Si es necesario modificar la implementación de un repositorio o servicio, podemos hacerlo dentro del assembly correspondiente (**DataAccess** o **BusinessLogic**), sin afectar a otros componentes. Los cambios en el código de la WebApi no impactan en la lógica de negocio o de acceso a datos, y viceversa, debido al uso de interfaces y a la clara separación de responsabilidades.
- **Inyección de Dependencias**: Aprovechamos la inyección de dependencias para desacoplar las implementaciones concretas de las interfaces, asegurando que cualquier cambio en una implementación no tenga un efecto en las demás capas del sistema.

Vista Lógica

La **Vista Lógica** describe la organización interna del software, enfocándose en cómo los módulos y componentes principales interactúan entre sí para satisfacer los requisitos funcionales.

BusinessLogic

Diagrama de clases del dominio en BusinessLogic



Herencia de Cámara con Device

La clase Device fue elegida como una clase padre debido a que contiene todos los atributos comunes entre sensores y cámaras, mientras que los atributos específicos de las cámaras están incorporados en la clase Camera de la que extiende Device.

La estructura de herencia elegida garantiza que los dispositivos compartan características clave, mientras que los detalles específicos de cada tipo de dispositivo se manejan en las clases derivadas, promoviendo la reutilización de código y asegurando un diseño coherente, mantenible y extensible. Más adelante si los atributos de Sensor cambian, se podría implementar Sensor como extensión de Device. De igual manera se podría implementar nuevos tipos de dispositivos como extensiones de Device en caso de necesitar más atributos o como parte de este, añadiendo su nombre a DeviceTypes, un enum que se utiliza para validar el tipo de dispositivo. Este mismo es el que utilizamos para mostrar los dispositivos soportados por el sistema.

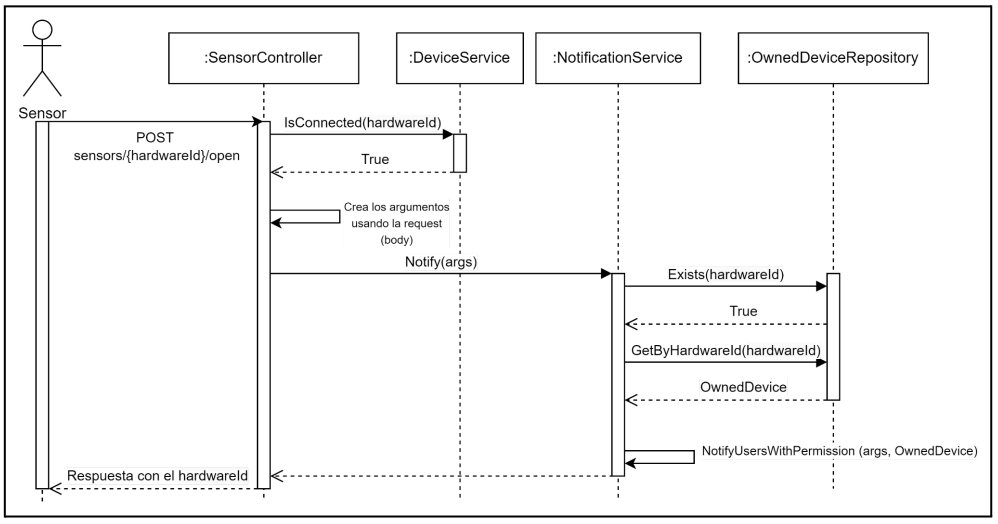
Manejo de notificaciones

La estructura de notificaciones complementa la arquitectura de dispositivos. Gracias a la herencia proporcionada por Device, las notificaciones se pueden asociar con cualquier tipo de dispositivo (sea una cámara, un sensor, o cualquier futuro tipo de dispositivo que se añada al sistema), ya que todas heredan de la clase base Device. Esto significa que cuando se añaden nuevos tipos de dispositivos al sistema, no es necesario modificar la lógica de notificaciones, manteniendo así el principio de abierto/cerrado.

La clase Notification actúa como el punto central para el manejo de notificaciones, actuando como una asociación entre los dispositivos y usuarios del sistema, añadiendo atributos relativos a la notificación, las notificaciones se asocian a un OwnedDevice, clase que asocia dispositivos y hogares, de la que hablaremos mas adelante.

El proceso de notificación se inicia a través del método Notify del NotificationService. Este método verifica la existencia del dispositivo, obtiene los miembros que deben ser notificados basándose en sus permisos, y crea las notificaciones correspondientes.

Diagrama de secuencia centrado en el controlador, mostrando la comunicación entre los tres componentes para notificar a los usuarios:



DataAccess

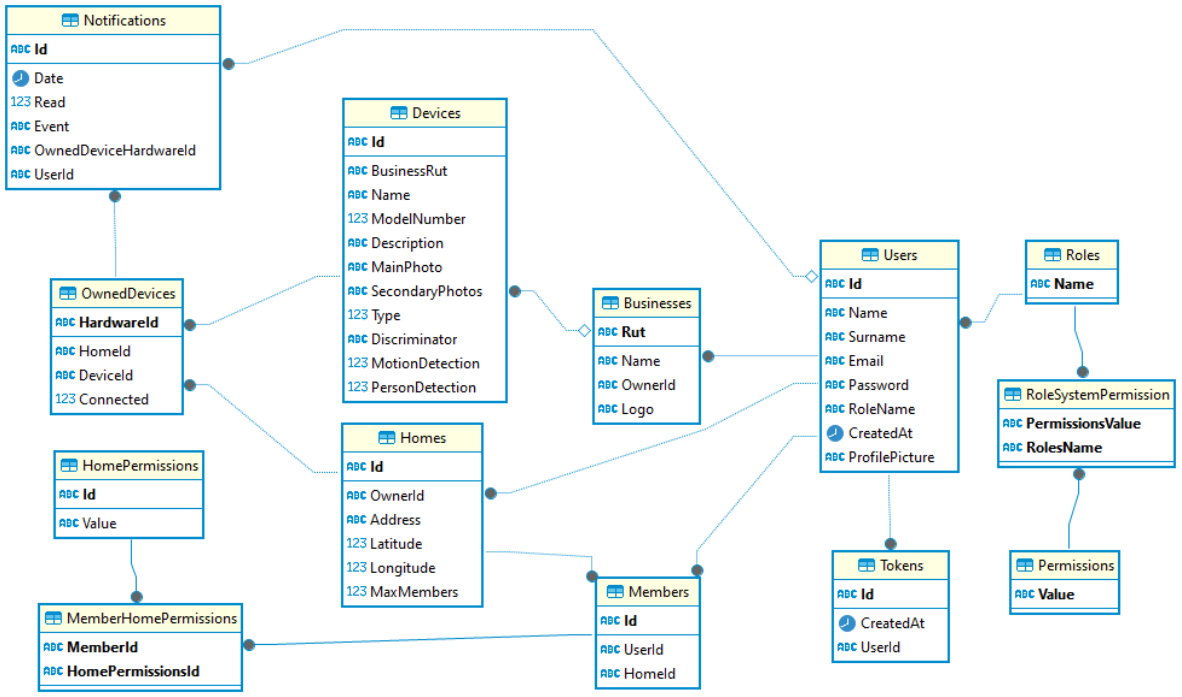
DataAccess solo se encarga de implementar las interfaces de repositorios que son declaradas en BusinessLogic, escondiendo en sí la complejidad del manejo de datos en la base de datos.

Como no nos aporta mostrar un diagrama de clases de DataAccess, el siguiente diagrama de componentes ilustra lo que sucede con UserRepository.

Los **repositorios** en nuestro proyecto actúan como adaptadores entre los servicios de negocio y el acceso a la base de datos, aplicando el **patrón Adapter** para conectar interfaces que, de otro modo, serían incompatibles. Los repositorios implementan interfaces comunes que permiten que los servicios interactúen con los datos sin conocer los detalles específicos de cómo se acceden o manipulan en la base de datos. De este modo, se abstrae la complejidad del acceso a los datos, lo que facilita que las clases con interfaces diferentes trabajen juntas sin modificar su código original.

Esta implementación del patrón Adapter también cumple con el **principio de abierto/cerrado (OCP)**, ya que los servicios están abiertos a la extensión sin necesidad de modificar su lógica interna. Es posible cambiar la implementación de un repositorio (por ejemplo, pasar de una base de datos SQL a NoSQL) sin afectar la lógica de los servicios. Los repositorios permiten que los servicios interactúen con diferentes tipos de fuentes de datos, manteniendo el sistema flexible y extensible sin cambios disruptivos en el código de las clases que manejan la lógica de negocio.

Tablas de la base de datos



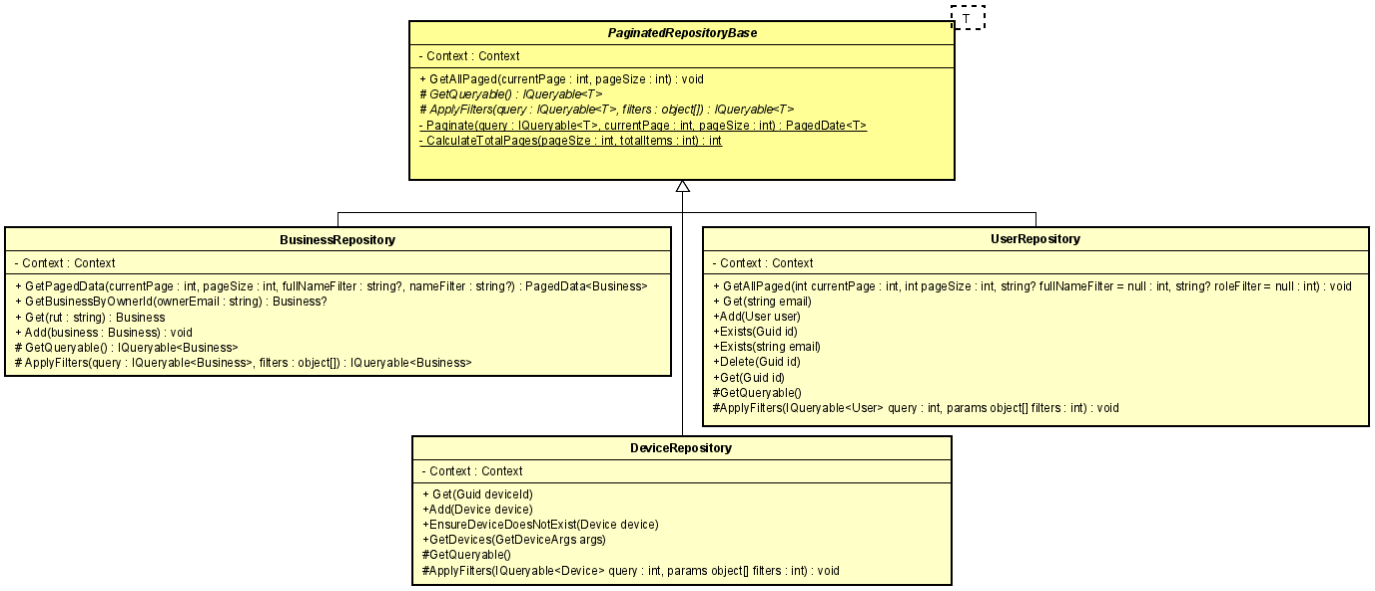
Herencia de PaginatedRepositoryBase en Repositorios

En vistas de que la estructura de los algoritmos de paginación eran los mismos en las tres clases que necesitaban paginar (BusinessRepository, DeviceRepository, UserRepository) decidimos implementar el patrón Template Method con la finalidad de reutilizar código y reducir la complejidad.

Dividimos el algoritmo de paginación y filtrado en los siguientes pasos:

1. **GetQueryable()**: Retorna el IQueryable donde se van a aplicar los filtros y la paginación posteriormente. Por ejemplo, en el caso que queremos obtener una lista de usuarios con paginación y filtrado, retornamos el propio DbSet de los usuarios. *Este método debe ser implementado por las clases que heredan.*
2. **ApplyFilters():** Aplica los filtros. Nuevamente, retorna un IQueryable luego de aplicarle los filtros necesarios. *Este método debe ser implementado por las clases que heredan.*
3. **Paginate():** Una vez obtenidos y filtrados los datos, este método se encarga de devolvernos la lista ya paginada con información el tamaño de página, la página actual y el total de páginas en un record PagedData. *Este método ya contiene una implementación por defecto que es usada por todas las clases hijas*.

De esta manera, se expone el template method **GetAllPaged()** y las clases que heredan solo tienen que redefinir **GetQueryable()** y **ApplyFilters()**, sin necesidad de redefinir el orden del algoritmo ni el método **Paginate()**.



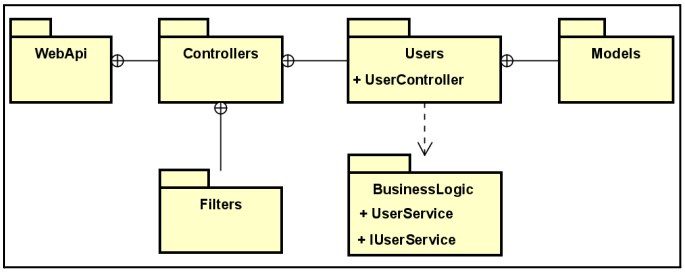
Finalmente, es oportuno mencionar que nuestra aplicación utiliza un **modelo de base de datos conectado**. A su vez, hemos configurado el ciclo de vida de los repositorios como Scoped, es decir, cada instancia del contexto se crea al inicio de la solicitud y se destruye al final de la misma. Esto implica que las entidades que se recuperen desde la base de datos permanecen vinculadas al Context mientras la request está activa, ofreciendo así varias ventajas. Primero, permite que cualquier modificación, inserción o eliminación realizada en las entidades sea trackeada por EF Core, sin necesidad de realizar ninguna operación adicional. Además, el ciclo de vida Scoped es ideal en este escenario porque cada request HTTP recibe su propio DbContext, lo que evita compartir el contexto entre múltiples solicitudes. A diferencia de un ciclo de vida Transient, donde se crea una nueva instancia cada vez que se inyecta el servicio, el ciclo Scoped nos garantiza ser más eficientes (minimizar las interacciones con la base de datos) sin sacrificar control (nos permite tener un control más preciso sobre los cambios en las entidades dentro de esa solicitud).

WebApi

Por último tenemos el namespace WebApi, el cual tiene 2 sub-namespaces principales. **Controllers** contiene las clases de que implementan los endpoints que declaramos en el archivo de especificación de la API. Estos se comunican con los servicios a través de sus interfaces. Existe solo una relación entre clases de WebApi, CameraController y SensorController ya que extienden BaseDeviceController.

Por otro lado tenemos **Filters**, son 4. Estos se encargan de abstraer al resto de clases de: la verificación de la sesión para identificar a los usuarios, la verificación de permisos generales y de permisos relacionados a un hogar. Por último un filtro que se encarga de manejar las excepciones.

Al igual que con DataAccess, no vemos conveniente diagramar las clases de WebApi, ya que no se relacionan entre sí, por lo que usaremos de ejemplo UserController, que utiliza la interfaz brindada e implementada en BusinessLogic, la cual le es inyectada. En todos los controllers se cumple la misma regla, utilizan interfaces de servicios que son brindadas e implementadas por BusinessLogic, las cuales les son inyectadas.



Utilización de Polimorfismo

En nuestro diseño, el **polimorfismo** se ha utilizado para mejorar la flexibilidad y capacidad de extensión del sistema, se puede ver especialmente en los controladores de **cámaras** y **sensores**, que heredan de una clase base llamada **BaseDeviceController**. Esta herencia permite que ambos controladores compartan funcionalidades comunes, como conectar y desconectar dispositivos, evitando la duplicación de código y promoviendo una estructura más limpia y organizada. Además, al definir las características comunes en un solo lugar, facilitamos la integración de nuevos tipos de dispositivos en el futuro; por ejemplo, si deseamos agregar un termostato, solo necesitaremos crear un nuevo controlador que herede de **BaseDeviceController,** y este ya tendrá el endpoint para conectarlo y desconectarlo. Este enfoque no solo mejora la mantenibilidad del sistema al garantizar un comportamiento coherente entre los dispositivos, sino que también sienta las bases para un crecimiento del sistema en adelante.

Autenticación y Autorización en HomeConnect API

Autenticación con "Bearer + Guid":

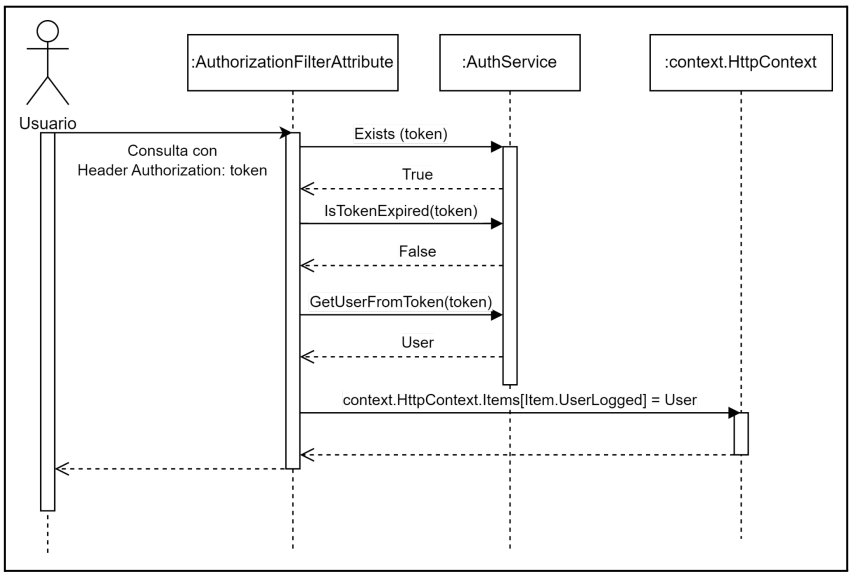
Flujo: Los usuarios inician sesión mediante un endpoint (`/auth`), proporcionando sus credenciales. Si son válidas, la API genera un token en formato "Bearer + Guid". Este token se guarda en la base de datos asociado al usuario y tiene una fecha de expiración.

Verificación del Token: Para cada solicitud protegida, el cliente envía el token en el encabezado HTTP (`Authorization: Bearer <Guid>`). La API, a través de un **middleware de autenticación**, revisa la base de datos para comprobar la validez del token y verifica si está expirado. Si el token es válido y no ha expirado, la solicitud continúa su ejecución.

Detalles adicionales:

- El middleware es responsable de interceptar todas las solicitudes entrantes que necesiten autorización y realizar las verificaciones necesarias.
- TokenRepository contiene información sobre los tokens, incluyendo la fecha de expiración.

- Si el token ha expirado o no es válido, el middleware devuelve una respuesta de error indicando que la autenticación falló.



Autorización basada en Permisos

- Protección de endpoints: Utilizamos AuthorizationFilter y HomeAuthorizationFilter, otros middlewares, para asegurar que solo los usuarios con los permisos necesarios puedan acceder a ciertos endpoints. Por ejemplo:
    - Solo los usuarios con el System Permission “create-business-owner” pueden crear cuentas de dueños de empresas.
    - Solo los dueños de hogares con el Home Permission “get-devices” pueden listar dispositivos en este.

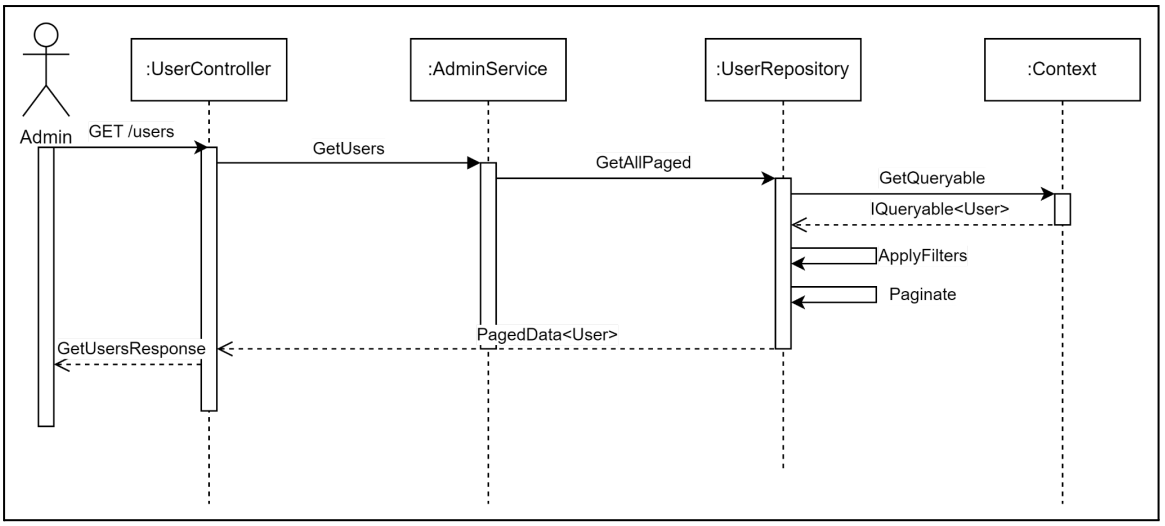
Decisiones de diseño principales

Descripción del mecanismo de acceso a datos utilizado

- **Controladores:** son el punto de entrada para las request en nuestra web API. Los controladores manejan las solicitudes HTTP y retornan respuestas HTTP. Llaman a servicios para acceder a la business logic.
- **Servicios:** encapsulan la business logic de la aplicación. Coordinan tareas y delegan trabajo a repositorios.
- **Repositorios:** abstraen la lógica de acceso a la base de datos, brindando métodos para interactuar con esta.

- **Context:** la clase Context es parte de Entity Framework, la cual es una ORM (Object Relational Mapper) que permite que trabajemos son una base de datos utilizando objetos.
- **Entidades:** son el dominio, mapean clases a tablas en la base de datos.

Se realiza una solicitud a la API web y es recibida por un controlador. El controlador llama a un método en un servicio, pasando los parámetros necesarios. El servicio utiliza un repositorio para obtener datos de la base de datos. El repositorio utiliza el Contexto para consultar la base de datos y mapear los resultados las entidades. El repositorio devuelve los datos al servicio. Luego, el servicio puede aplicar lógica adicional sobre los datos. El servicio pasa a devolver el resultado al controlador. El controlador empaqueta el resultado en una respuesta HTTP y lo envía finalmente al cliente.



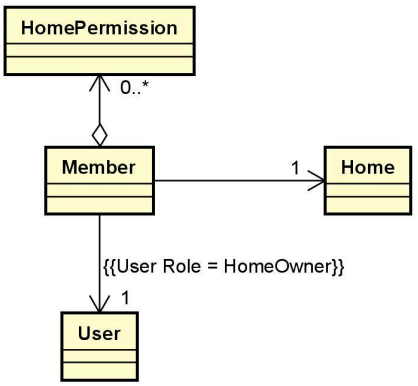
Agrupación de tipos de usuario en User

Decidimos unificar los tres tipos de usuarios registrados en la plataforma (Administrador, Dueño de empresa y Dueño de hogar) en una sola clase, ya que comparten prácticamente todos sus atributos, excepto la foto de perfil, la cual optamos por hacerla un atributo nullable. No consideramos necesario crear una clase separada para HomeOwner solo por ese pequeño detalle, ya que agregar complejidad adicional al modelo por un atributo no aportaría un beneficio significativo. Sin embargo, esta decisión de consolidar las clases nos llevó a un reto con respecto a la gestión de **roles y permisos**. Dado que todos los usuarios ahora pertenecen a la misma clase, no podemos asignarles permisos directamente. En lugar de ello,

implementamos un sistema de roles donde cada usuario tiene un rol específico (Administrador, Dueño de empresa o Dueño de hogar). Cada rol está asociado con una lista de permisos, que se configura al momento de crear la base de datos. En cada endpoint de la aplicación, verificamos si el usuario tiene los permisos necesarios para realizar la acción solicitada, asegurando así un control de acceso flexible y seguro sin necesidad de clases adicionales para cada tipo de usuario, permitiéndonos en un futuro agregar mas tipos de usuario sin tener que cambiar la clase User.

Clase intermediaria entre Home y User

Otros problemas que se nos presentaron fueron: cómo hacer posible que un usuario pertenezca a varios hogares y el cómo manejar los permisos que tiene un usuario sobre un hogar específico, pues cada hogar tiene miembros, y estos pueden tener o no ser capaces de añadir dispositivos, obtener la lista de dispositivos del hogar y recibir notificaciones.



En nuestra solución, un hogar tiene una lista de miembros (Member). El rol de la clase Member es vincular a un usuario con un hogar, definiendo también qué tipo de permisos tiene en este, contando con una lista de HomePermission, clase encargada de almacenar los permisos específicos que un miembro puede tener.

Al separar los permisos en HomePermission pudimos diferenciar bien a los permisos de un hogar del resto de permisos existentes en nuestro sistema. También ganamos flexibilidad pues diferentes miembros tienen permisos distintos sin que estos estén vinculados directamente a User.

Al introducir Member como intermediario entre Home y User, fomentamos la separación entre la información relacionada con los usuarios y la información específica de los hogares, reduciendo el acoplamiento entre estas entidades. Además le permitimos a un usuario “ser” Member de más de un hogar, con sus permisos correspondientes, solucionado el problema inicial.

Descripción del manejo de excepciones

En el paquete **WebApi**, implementamos un filtro llamado *ExceptionFilter*, cuya finalidad es interceptar todas las excepciones no manejadas y comunicarlas al usuario final. Este filtro nos proporciona una manera centralizada de gestionar las excepciones, asignándoles códigos de error HTTP correspondientes.

El *ExceptionFilter* utiliza un diccionario que mapea las excepciones soportadas a sus respectivos códigos de error HTTP. En caso de que una excepción no se encuentre en este diccionario, se retorna un código de error genérico 500 (Internal Server Error).

Esta implementación garantiza una respuesta consistente y controlada ante errores inesperados, mejorando la experiencia del usuario y la mantenibilidad del sistema.

La única excepción a esto es cuando se envían tipos de datos incorrectos en el cuerpo de una solicitud (por ejemplo, un número en lugar de un bool), se generan excepciones que no están siendo controladas. Aunque tenemos un filtro de excepciones (ExceptionFilter) que asigna códigos de error HTTP a las excepciones definidas, este no captura las excepciones relacionadas con tipos de datos inválidos, lo que provoca respuestas inesperadas en herramientas como Postman.

Dado que la API está destinada a ser usada por desarrolladores, consideramos que las excepciones detalladas proporcionadas automáticamente en estos casos son útiles para identificar y corregir errores en las solicitudes. A futuro, una solución posible sería aceptar todos los datos como strings nulleables y parsearlos internamente, lo que daría más control y evitaría estos errores de tipo no manejados.

Clase intermediaria entre Device y Home

Una de las funcionalidades que se nos pidió fue la asociación de dispositivos a hogares. La solución más simple hubiera sido agregar una lista de dispositivos directamente en la entidad **Hogar**. Sin embargo, esto presentaba el problema de no poder asociar el mismo dispositivo varias veces al mismo hogar, algo que podría ser necesario en algunos casos. Para resolver esto, optamos por crear una clase intermedia llamada **Own Device**, que tiene referencias tanto al **Hogar** como al **Dispositivo**, un estado de conexión y un ID único. Esta estructura nos

permitió cumplir con el requisito de tener múltiples asociaciones entre dispositivos y hogares, manteniendo la flexibilidad necesaria.

Esta decisión refuerza varios principios clave. Primero, el **Principio de Responsabilidad Única (SRP)** se respeta al separar las responsabilidades: el **Hogar** no gestiona directamente las asociaciones con dispositivos, sino que delega esta tarea a la clase **OwnedDevice**, que se encarga de gestionar las conexiones y el estado de los dispositivos. Además, siguiendo el **Principio de Abierto/Cerrado (OCP)**, esta estructura es fácilmente extensible. Si en el futuro la empresa quisiera agregar más características a la asociación entre dispositivos y hogares (como historial de conexiones o tipos de dispositivos), podemos extender **OwnedDevice** sin modificar la clase **Hogar**.

Criterio de asignación de responsabilidad

En la asignación de responsabilidades, seguimos el **principio de responsabilidad única (SRP)** para asegurar que cada componente del sistema tenga una única razón para cambiar. Los **controladores** se encargan de delegar las llamadas a la API hacia los servicios, limitándose a gestionar las solicitudes HTTP sin involucrarse en la lógica de negocio. Los **servicios**, por su parte, interactúan con los repositorios y aplican la lógica necesaria sobre los datos, siendo responsables de coordinar las operaciones del sistema.

Los **repositorios** se encargan de implementar las operaciones **CRUD** (Crear, Leer, Actualizar y Borrar) y abstraen los detalles del acceso a la base de datos, cumpliendo con el patrón de repositorio al proporcionar un acceso centralizado a los datos sin que los servicios necesiten conocer los detalles de cómo se gestionan estos datos. Además, los **filtros** actúan como middleware, delegando en sí el control de aspectos transversales como la autenticación, la autorización y la gestión de excepciones, asegurando que el flujo principal de la lógica de negocio en los servicios permanezca limpio y enfocado en las reglas específicas de la aplicación.

Vista de Procesos

No manejamos casos de concurrencia en esta entrega al momento a pesar de que en C# existe esta posibilidad