Obligatorio 2

Diseño de Aplicaciones 2

Ingeniería en sistemas

Docentes: Daniel Acevedo

Federico Gonzalez



Francisco Fernández (301270)

Agustin Varela (303129)

Joaquín Mesa (295871)

Link a Repo

UNIVERSIDAD ORT
Uruguay

Índice

Descripción general del trabajo

El proyecto SmartHome se centra en el desarrollo de una plataforma integral, que abarca desde la creación de dispositivos inteligentes hasta la gestión dentro de un hogar, con el objetivo de crear un entorno automatizado y centralizado.

La aplicación está diseñada para ser utilizada por múltiples usuarios con diferentes roles (administradores, dueños de empresas y dueños de hogares), cada uno con distintos niveles de acceso y permisos.

El objetivo principal es crear una experiencia unificada que facilite la interacción y el control de múltiples dispositivos inteligentes, independientemente del fabricante, mientras se mantiene una estructura de permisos sólida para garantizar la seguridad y el control adecuado de la información y los dispositivos.

Errores conocidos

Hasta el momento, no se han identificado fallas críticas en el sistema. Cualquier comportamiento anómalo o error reportado ha sido abordado y solucionado durante el desarrollo.

Sin embargo, enfrentamos un inconveniente significativo relacionado con las migraciones de la base de datos. En un momento, generamos migraciones corruptas o accidentalmente eliminamos alguna, lo que resultó en la imposibilidad de actualizar la base de datos correctamente.

Para resolver este problema, nos vimos obligados a eliminar todas las migraciones existentes y recrearlas desde cero, siendo cautelosos para que no ocurra otra vez. Como consecuencia, la solución actual no incluye las migraciones de la primera entrega.

En el aspecto de diseño visual, reconocemos que el front-end no alcanza el nivel estético que hubiésemos deseado. Esto se debe principalmente a restricciones de tiempo que nos llevaron a priorizar los requerimientos funcionales, el diseño sólido del código y la implementación de buenas prácticas de desarrollo.

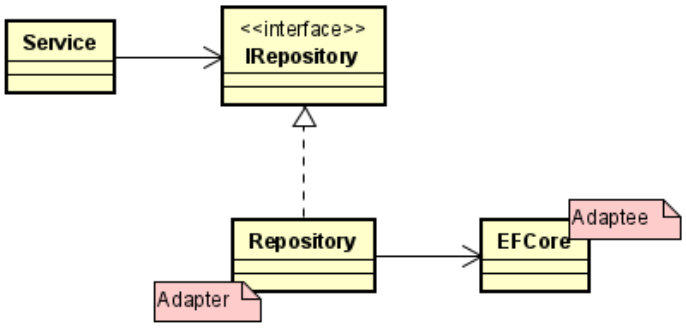
Nos enfocamos en garantizar que el sistema sea mantenible, extensible y esté fundamentado en principios sólidos de programación, como SOLID y GRASP. Además, nos aseguramos de contar con una cobertura adecuada de pruebas para ofrecer un producto robusto y confiable. Si bien el diseño visual quedó relegado en esta etapa, consideramos que hemos sentado una base firme para futuras mejoras en esta área.

Vista Lógica:

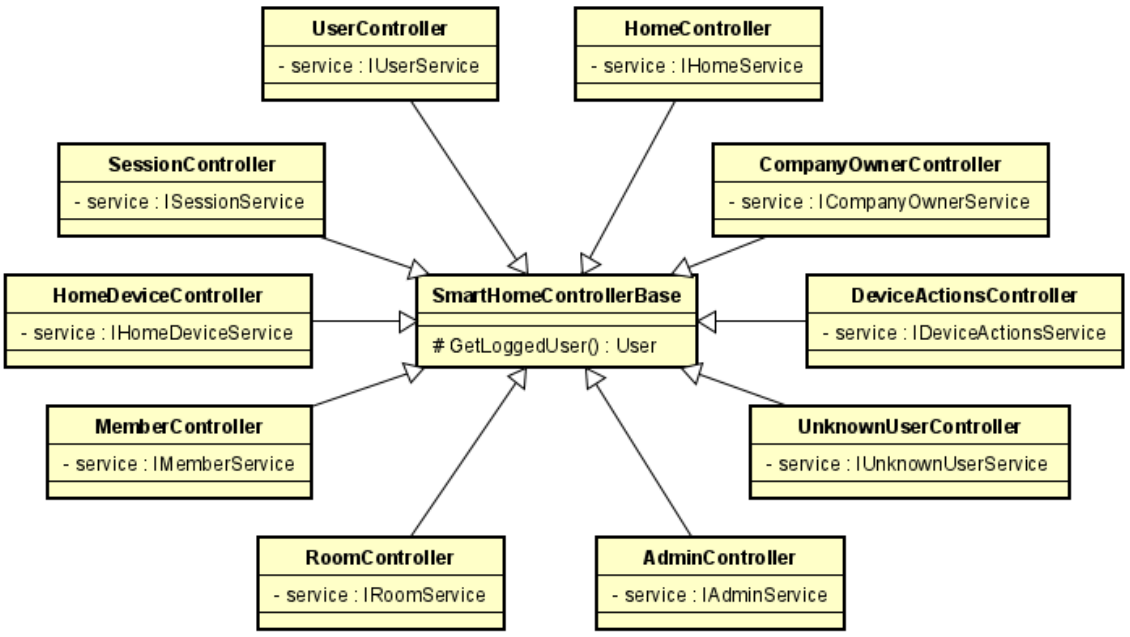
El diseño de la solución se planteó con una arquitectura horizontal (por tipos). Sin embargo, actualmente muchos proyectos tienden hacia una arquitectura vertical (por funcionalidad), debido a sus ventajas en términos de escalabilidad, menor acoplamiento, mayor cohesión y mejor desempeño en métricas. Esta sería una posible mejora para el proyecto, aunque no fue factible implementarla por limitaciones de tiempo.

Para distribuir las responsabilidades en la solución, dividimos el proyecto en tres componentes principales:

DataAccess: este proyecto se encarga del acceso a datos y de la generación de la base de datos, encapsulando la tecnología de Entity Framework Core. Utilizamos el patrón Adapter para lograr este encapsulamiento y mantener flexibilidad en la elección de la tecnología de base de datos.



BusinessLogic: aquí se encapsula toda la lógica de negocio, incluyendo las clases de mayor nivel (clases de dominio), validadores, lógica de load assembly para reflection y los servicios. Para reducir el acoplamiento en esta capa, no priorizamos la reutilización de código y la especialización de los servicios en repositorios específicos. Por esta razón, cada servicio recibe inyección de varios repositorios y no se comunican entre ellos. Una desventaja de esto es que cada servicio no será experto en un repositorio lo que lleva a disminuir la cohesión.



WebApi: este proyecto contiene un controller por servicio, donde priorizamos SRP inyectando un único servicio por controller. Para reutilizar código, creamos un Controller Base, que devuelve el usuario actual (current user) y centraliza la lógica común, el resto de Controllers heredan del Base, como muestra se en la figura. Aquí también se encuentran los filtros de excepción, que capturan todas las excepciones lanzadas en el back-end y las retornan en una respuesta adecuada siguiendo el estándar RESTful. Además, agregamos filtros de autenticación y autorización, eliminando esta responsabilidad de los servicios y validando previamente al usuario actual. Como posible mejora, se podría añadir un filtro de autorización específico para validar las acciones relacionadas con el hogar. Siguiendo las mejores prácticas, nuestros controllers solo se encargan de transferir datos y no contienen lógica de negocio.

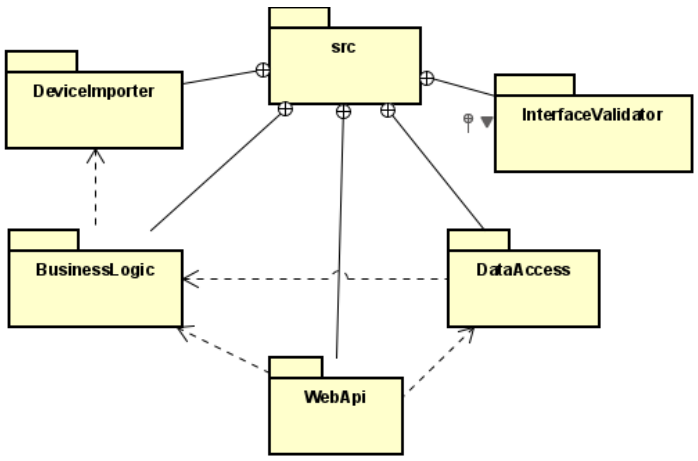
Para el diseño de APIs, siguiendo la guia RESTful, tuvimos en cuenta las características:

1. Las solicitudes del cliente no mantengan ningún estado a nivel del servidor, es decir que cada solicitud es completamente independiente, esto cumple con que sea Stateles.
2. Los recursos son identificados por URL únicas.
3. Uso de operaciones estándar Http (get, post, put, delete, patch,etc)
4. Interfaz uniforme, todas las interacciones entre cliente y servidor siguen un modelo consistente, facilitando su comprensión y uso.

Una posible mejora para los endpoints es que retornamos 200 ok para todos los endpoint, y podríamos utilizar otros códigos como por ejemplo: 201- created, para identificar cuando un dispositivo o un usuario se crea.
Otra posible mejora es hacer el endpoint de tipos de devices con la restricción cacheable ya que sabemos que no cambia frecuentemente. Esto nos daría ventajas en performance del servidor ya que volvería a consultar en caso de que hayan cambiado los tipos de dispositivos o ante alguna condición específica.

Para adherirnos al principio stateless de las Web API, los controllers se configuraron utilizando el patrón AddScoped en el archivo Program.cs. Esta implementación garantiza que cada solicitud HTTP crea una nueva instancia del controller, evitando que el backend almacene cualquier tipo de estado.

De esta manera, la responsabilidad de gestionar el estado recae completamente en el frontend, lo que no solo simplifica la lógica del backend, sino que también mejora la escalabilidad y asegura el cumplimiento de las buenas prácticas en arquitecturas REST.



Como se observa en el diagrama, además de los componentes principales, la solución incluye otros dos módulos diseñados específicamente para soportar las funcionalidades de importar dispositivos y validar su modelo.

Aunque BusinessLogic interactúa con InterfazValidator, esta interacción no establece una dependencia directa en términos de arquitectura. Esto se logra al hacer referencia explícita al archivo DLL de InterfazValidator, lo que permite mantener una estructura desacoplada y flexible.

DevicesImporter: este proyecto contiene una interfaz y los DTOs necesarios para la misma. El objetivo de la interfaz es que todos los posibles nuevos importadores cumplan con ella, permitiéndonos manejar correctamente las funcionalidades y no lograr impacto a nuevas clases concretas en Business Logic.
Además, el paquete Business Logic es un paquete muy estable (no es propenso a cambios), por lo tanto no debería de depender de clases que tengan posibles cambios (principio de dependencias estables).
También, cumple con el principio de **polimorfismo**, ya que cualquier nueva clase concreta que implemente esta interfaz podrá ser utilizada de manera uniforme e intercambiable, sin afectar al resto del sistema.

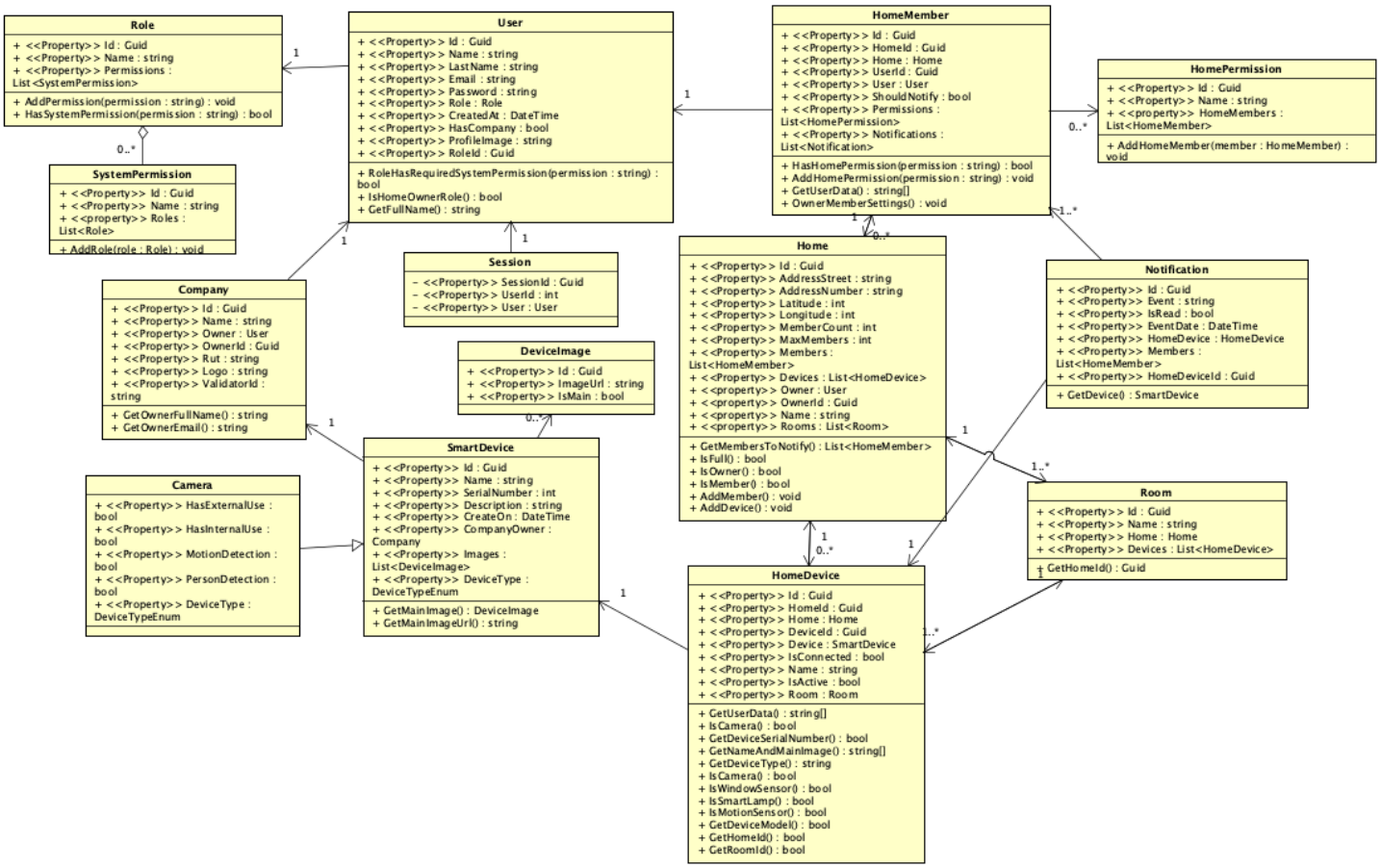
Conclusión:
La estructura actual, organizada por responsabilidades, facilita la separación de la lógica, reduce dependencias innecesarias y promueve un código modular, escalable y fácil de mantener. Este enfoque permite que cada componente cumpla un propósito específico, minimizando el riesgo de acoplamiento excesivo y mejorando la claridad del diseño.

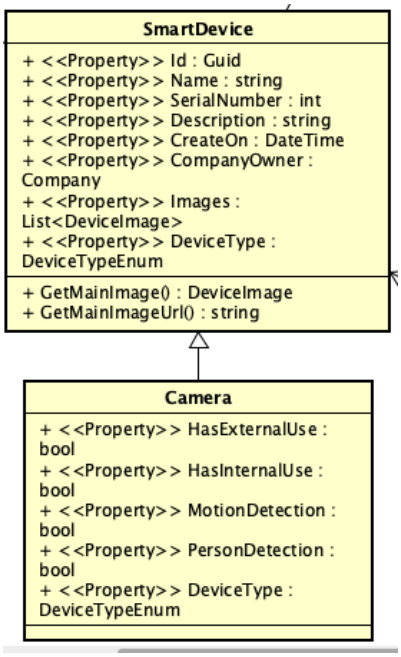
Observación
En la solución aparecen dos proyectos adicionales "ModelValidator" y "JsonDevicesImporter" que, aunque podrían eliminarse sin afectar directamente el funcionamiento del código principal, son fundamentales para las funcionalidades de importar dispositivos y validar el modelo de un dispositivo. Estos proyectos son necesarios para

soportar tareas relacionadas con reflection y requieren que sus archivos DLL sean utilizados durante la ejecución.

El paquete Dominio contiene nuestras clases principales del problema, las cuales están diseñadas con una alta cohesión. Cada clase tiene una responsabilidad claramente definida, lo que garantiza que sus comportamientos estén estrechamente relacionados con su propósito. Sin embargo, a pesar de esta cohesión, estas clases presentan un cierto grado de acoplamiento entre ellas.

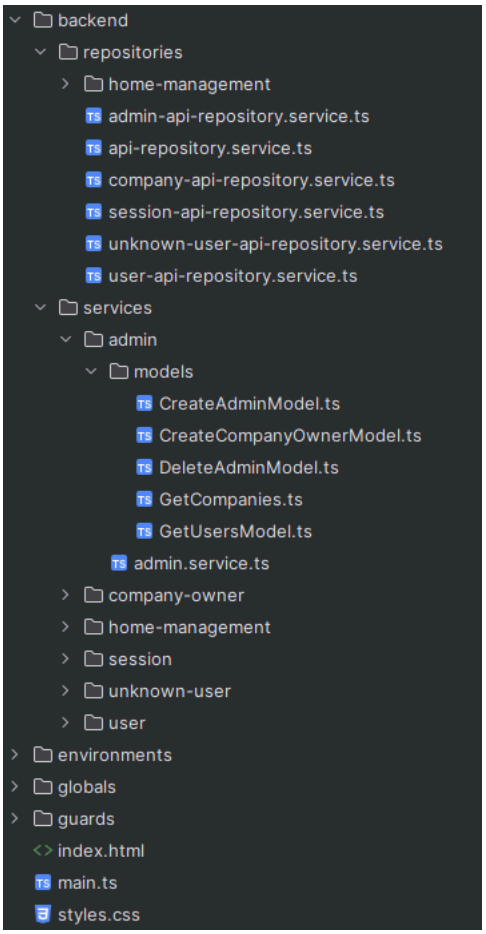
Estas clases siguen principios fundamentales como el principio del experto (cada clase es responsable de lo que mejor conoce) y el Principio de Responsabilidad Única (SRP), donde cada clase es responsable de validar y mostrar sus propios datos. Este enfoque asegura que el sistema cumpla con el principio de la ley de Demeter, ya que evita que las clases realicen tareas fuera de su área de especialización.





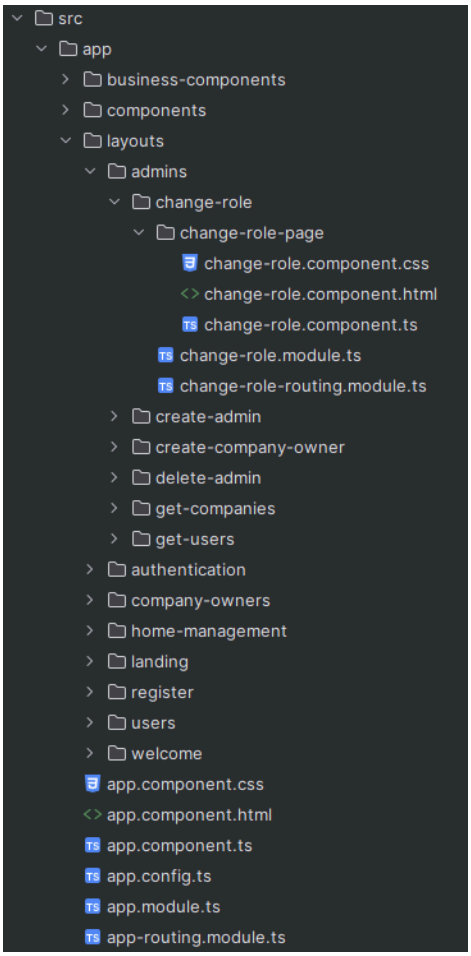
Como se puede ver en el diagrama, utilizamos **jerarquía de herencia** entre SmartDevice y Camera. Respecto al obligatorio anterior ahora le agregamos una property DeviceType a la clase SmartDevice, permitiendo que los tipos de dispositivos que no son Cámara puedan ser creados con la misma clase. Dado que sabemos que no es frecuente el cambio de tipos de dispositivos, consideramos que la solución es correcta. Camara extiende de SmartDevice dado que tiene más funcionalidades, por lo tanto, más properties.

Esta herencia cumple con el patrón SOLID Liskov ya que si intercambiamos las instancias, el comportamiento del sistema no se ve afectado.



backend/repositories: Se encargan de realizar las solicitudes HTTP al backend. Cada repositorio sigue el nombre <controller>-repository.service.ts y hereda de ApiRepositoryService, que centraliza la lógica de manejo de solicitudes y el tratamiento de errores. Los repositorios actúan como una capa de acceso a datos.

backend/services: Los servicios son responsables de la lógica de negocio del frontend. Utilizan los repositorios para interactuar con el backend y realizar operaciones específicas. Los servicios son independientes entre sí y proporcionan una API interna clara que otros componentes o módulos de la aplicación pueden consumir. Los servicios encapsulan la lógica de negocio y proveen métodos que pueden ser reutilizados en diferentes partes de la aplicación. Mantuvimos la organización que los controllers del backend, es decir, tenemos un servicio por controller.



app/layouts: Los layouts están organizados de acuerdo con los roles de usuario, lo que permite estructurar las vistas de manera eficiente. Además, mantiene la organización de los controllers del backend y los servicios de frontend.

Cada layout refleja la representación de las funcionalidades para el usuario.

Dentro de cada layout, se incluyen los módulos (.module) y las páginas (/pages), que a su vez contienen los componentes (html, css y ts). Esta jerarquía modular permite una mayor organización y reutilización del código, facilitando el mantenimiento y la escalabilidad del sistema. Cada capa tiene responsabilidades claras, lo que refuerza la modularidad y asegura que los cambios en una capa no afecten indebidamente a otras.

app/components: Contiene los componentes reutilizables y genéricos (ej. visualización de excepciones, navegación de tablas, menús, botones de routing). Estos componentes pueden ser reutilizados en cualquier otro proyecto de angular ya que están aislados a la lógica de negocio de nuestra aplicación.

app/business-components: Contiene los componentes específicos del negocio (ej. Creación de dispositivos, visualización de respuestas del servidor). Estos componentes fueron creados para poder ser reutilizados en nuestro código. Utilizan los componentes genéricos (components) con lógica de negocio vinculada**.** Esto nos permitió reutilizar el código y ganar tiempo de desarrollo.

globals: Contiene variables globales que centralizan configuraciones y constantes reutilizables en la aplicación, como los roles con su identificador y parámetros compartidos entre las distintas capas. Esto asegura que las configuraciones sean consistentes y fáciles de mantener.

environments: Gestiona las configuraciones para diferentes entornos (desarrollo, producción). Este enfoque facilita el cambio entre entornos sin necesidad de modificar el código base, permitiendo que las solicitudes se realicen a los endpoints correctos dependiendo del entorno.

guards : Los guards son una capa de seguridad que se encarga de proteger las rutas de la aplicación. También validan los roles de los usuarios, asegurando que los permisos estén alineados con la lógica de backend.

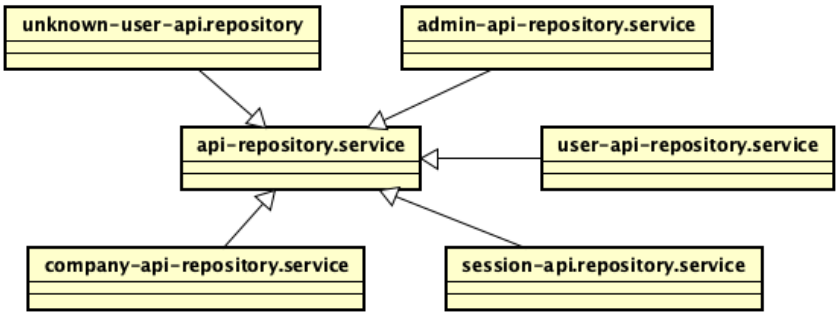
app-routing.module: Este archivo define las rutas de la aplicación, se encarga de configurar el enrutamiento, permitiendo la navegación entre las vistas de la aplicación. Definimos el mapeo de url con componentes y módulos correspondientes. En cada ruta definimos el path, las guardas y la carga de módulos de forma diferida (load children). Esta carga Lazy permite que un módulo no se cargue hasta que el usuario navegue a la ruta correspondiente, mejorando el rendimiento.

app.module.ts: Este archivo es el módulo raíz que inicializa y arranca la aplicación. Importa los módulos necesarios y declara componentes.

Esta separación nos permite que las carpetas tengan una responsabilidad definida, disminuye el impacto de cambios y fomenta la reutilización (de componentes y funcionalidades de servicios).

Además, esta forma de carga inicial y luego hacer interacción sin recargar la página sigue los requisitos de una SPA (Single Page Application) que nos da ventajas en performance y en la experiencia del usuario.

A nivel de repositorios utilizamos un api-repository, del cual el resto de repositorios heredan. Este repository se encarga de proporcionar los métodos genéricos para las solicitudes http, manejo de errores (handleError) de las solicitudes, manejo de encabezados, etc. Dado que todos los repositorios necesitan esto, nos facilita la reutilización del código.



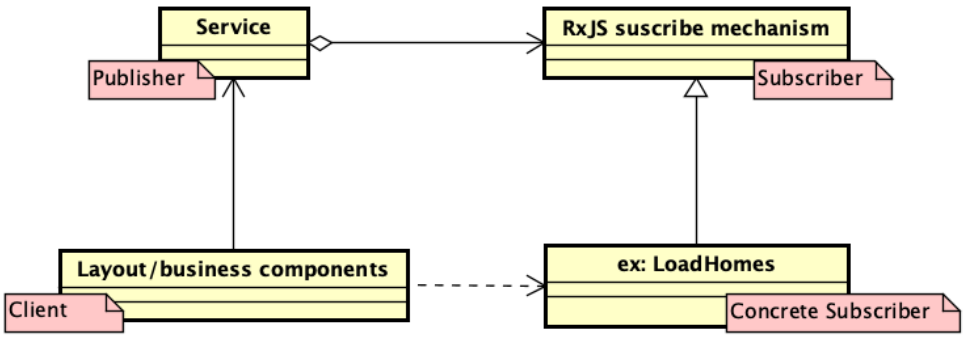
Para optimizar la notificación de cambios en los estados, hemos implementado el patrón Observer, aprovechando el servicio HttpClient de Angular y los Observable de la librería RxJS. Este enfoque permite una gestión más eficiente de las actualizaciones de estado, notificando a los componentes interesados de manera asíncrona.

Observer: Los métodos de HttpClient, como get, post, patch, etc., retornan objetos Observable.

Subscriber: Los componentes se suscriben a estos Observables para recibir datos o manejar errores. Los servicios actúan como emisores de datos (Observables), mientras que los componentes actúan como observadores, reaccionando a los datos o errores al suscribirse a los Observables.

Al implementar el patrón Observer en los servicios, se desacopla la lógica de obtención de datos de la lógica de presentación en los componentes. Esto ocurre porque los servicios actúan como emisores de datos y los componentes simplemente se suscriben a esos datos, sin necesidad de conocer cómo o cuándo se obtienen. Este enfoque facilita la mantenibilidad del código, ya que cualquier cambio en la lógica de obtención de datos solo requiere modificaciones en los servicios, sin afectar a los componentes.

Además, los métodos de los servicios devuelven Observables, lo que permite que cualquier componente se suscriba y reutilice la misma fuente de datos en diferentes contextos. Esto evita la duplicación de lógica, ya que no es necesario implementar la misma lógica de obtención de datos en múltiples lugares. Al centralizar la lógica de acceso a los datos, se promueve la consistencia en todo el sistema.



Análisis de métricas

Proyecto	Eferentes (Ce)	Aferentes (Ca)	Cohesión relacional (H)	Abstracción (A)	Inestabilidad (I)	Distancia (D)
DataAccess	113	2	1.61	0	0.98	0.01
BusinessLogic	74	25	2.90	0.17	0.75	0.05
WebApi	170	0	1.45	0	1	0
InterfaceValidator	0	0	0	0	0	0
DevicesImporter	12	2	1	0.33	0.86	0.13

Cada proyecto está representado por un color: DataAccess BusinesLogic Web api

Paquetes	Eferentes (Ce)	Aferentes (Ca)	Cohesión relacional (H)	Abstracción (A)	Inestabilidad (I)	Distancia (D)
Repositories	21	1	1	0	0.96	0.03
Domain	7	52	1.88	0	0.12	0.62
Services	33	1	0.27	0	0.97	0.02
Interfaces	27	23	0.07	1	0.54	0.38
Controllers	44	1	1	0	0.98	0.01

Conclusión a nivel de proyectos

Las métricas reflejan una correcta separación de las capas principales: base de datos (DataAccess), lógica de negocio (BusinessLogic) y capa de presentación (WebApi). Esto demuestra que se respetan los principios de acoplamiento de paquetes, como SDP (Stable Dependency Principle) y SAP (Stable Abstraction Principle), ya que ningún proyecto depende de otro menos estable.

Además, destaca un alto nivel de cohesión, especialmente en BusinessLogic, donde se cumple el principio CCP (Common Closure Principle). Esto garantiza que las clases relacionadas cambian juntas, facilitando tanto el mantenimiento como la evolución del sistema.

Conclusión a nivel de subpaquetes

En los subpaquetes, las métricas no son óptimas debido a la implementación de una arquitectura horizontal. Este enfoque organiza los paquetes por tipo de responsabilidad, lo que limita la cohesión interna y afecta negativamente el cumplimiento del principio CCP en varios de ellos.

Un cambio hacia una arquitectura vertical, agrupando paquetes según las funcionalidades que ofrecen, incrementaría la cohesión interna al fortalecer las relaciones entre las clases dentro de cada paquete.

Aunque Repositories, Services e Interfaces presentan métricas menos saludables, Domain destaca por su cumplimiento del principio CCP y métricas coherentes, siendo la base de la solución. En general, los paquetes no están lejos de la secuencia principal, lo que indica que, a pesar de las áreas de mejora, la solución es sólida.

Vista de Desarrollo:

Resumen de mejoras y adaptaciones del diseño:

Como mencionamos en la entrega pasada, tuvimos un imprevisto de último momento que nos impidió finalizar el proyecto como deseábamos. En esta segunda etapa, comenzamos enfocándonos en corregir y completar lo pendiente, trasladando los filtros de autenticación y autorización a la capa de presentación. Esto nos permitió capturar de forma temprana cualquier comportamiento inesperado y gestionar el current user en el backend de manera más eficiente.

Los nuevos requerimientos no implicaron grandes cambios, ya que el diseño de nuestra solución es mantenible y extensible. La gestión de roles solo requirió agregar dos nuevos roles, AdminHomeOwner y CompanyAndHomeOwner, asignándoles los permisos adecuados en los datos de inicialización (seed data), sin alterar la estructura del proyecto.

El nombre del hogar y de los dispositivos se reflejó en dos clases, a las que añadimos una propiedad para este fin y creamos los endpoints y funciones necesarias para permitir su modificación.

La gestión de habitaciones la implementamos en una clase separada, lo que evitó la necesidad de reestructurar el código. Este enfoque nos permitió agrupar dispositivos (lógicos) por cuarto sin hacer grandes cambios.

Para cumplir con los requerimientos de reflection, y dado que estas operaciones se ejecutan en tiempo de ejecución, únicamente incorporamos las clases y proyectos necesarios para leer la DLL y asociar las interfaces correspondientes.

En ambas funcionalidades, decidimos encapsular el código provisto por la interfaz, ya que este estaba escrito en español. Para ello, desarrollamos un Service, que actúa como un intermediario entre nuestra lógica y la implementación de la interfaz.

Este servicio no solo aísla nuestro código de los detalles de la implementación proporcionada por la interfaz, sino que también facilita el mantenimiento y la evolución del sistema,

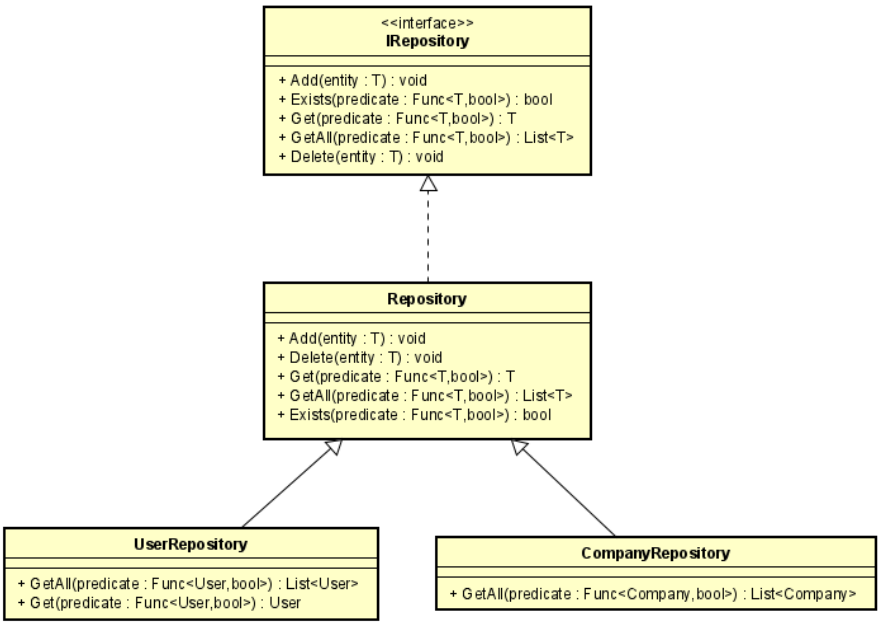
permitiendo realizar cambios en la validación/importación sin afectar al resto de la arquitectura.

En cuanto al diseño, dividimos el HomeOwner Service/Controller en cuatro clases, dado que, con los nuevos requerimientos, estaba acumulando demasiadas responsabilidades. Ahora, una clase gestiona miembros, otra cuartos, otra se especializa en dispositivos, y la última se encarga de la administración general del hogar.

Con el objetivo de que nuestro código siga siendo extensible y fácil de adaptar para cualquier desarrollador, decidimos mantener nuestro enfoque en TDD (Test Driven Development). Esta práctica no solo nos da confianza para realizar cambios, sino que también nos brinda la seguridad de que cada ajuste está respaldado por una batería de tests sólidos.

Optamos por estándares de C# que permiten a cualquier programador experimentado en el lenguaje comprender nuestro código de forma rápida y sencilla. Además, seguimos las mejores prácticas de Clean Code, utilizando nombres de variables mnemotécnicas, separando en funciones auxiliares, reutilizando código y asegurándonos de escribir en inglés.

Siguiendo el principio de evitar la duplicación de código y trabajar de manera más eficiente, implementamos un repositorio genérico que hereda de IRepository. Dado que en algunas consultas con EF Core es necesario aplicar la estrategia eager loading para obtener tanto la tabla principal como las tablas relacionadas, creamos un EntityRepository específico para ciertas entidades. Este hereda del repositorio genérico, permitiéndonos **reutilizar** métodos que no necesitan eager loading y sobrescribir aquellos que sí lo requieren. Al hacerlo de esta manera, junto con la **inyección de dependencias**, mejoramos la mantenibilidad del proyecto, facilitando que más



clases puedan utilizar eager loading en el futuro si es necesario. A continuación se muestra un diagrama reducido.

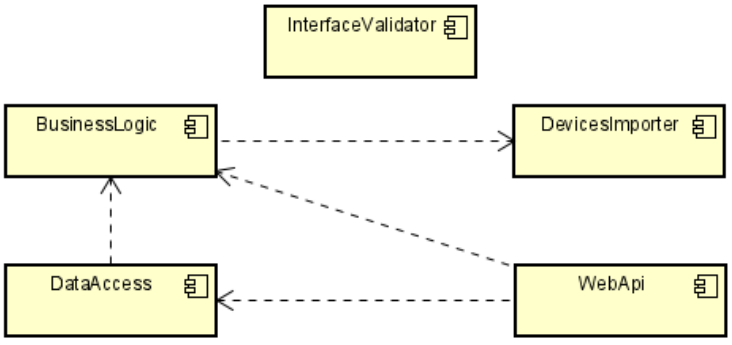
Diagrama de componentes:

Para organizar los componentes, nos guiamos por la premisa de evitar divisiones sin propósito claro, buscando encapsular las responsabilidades dentro de cada proyecto. Esto nos permite, en el futuro, reutilizar componentes individuales o, mediante reflection, utilizar una DLL sin necesidad de vincularla directamente en la solución, eliminando así la dependencia directa de esa tecnología. Por ejemplo, SmartHome.DataAccess.dll se puede excluir del proyecto simplemente haciendo referencia a la DLL, lo cual reduce las dependencias tecnológicas.

Por esta razón, decidimos crear solo tres proyectos:

- BusinessLogic: encapsula toda la lógica del negocio.
- Web API: contiene los filtros de autenticación y autorización, captura excepciones, y se encarga de recibir y procesar solicitudes HTTP.
- DataAccess: maneja la creación de la base de datos y las operaciones CRUD.

Esta estructura organiza claramente las responsabilidades, permitiendo una fácil expansión y mantenimiento del proyecto. Por ejemplo, nos da la flexibilidad de desarrollar un frontend móvil que consuma nuestra API sin necesidad de realizar cambios en el servidor. Además, mediante el uso de reflection (como se mencionó antes), es posible aprovechar la misma base de datos en distintas lógicas de negocio, evitando dependencias rígidas entre ellas.



Distribución de paquetes y proyectos:

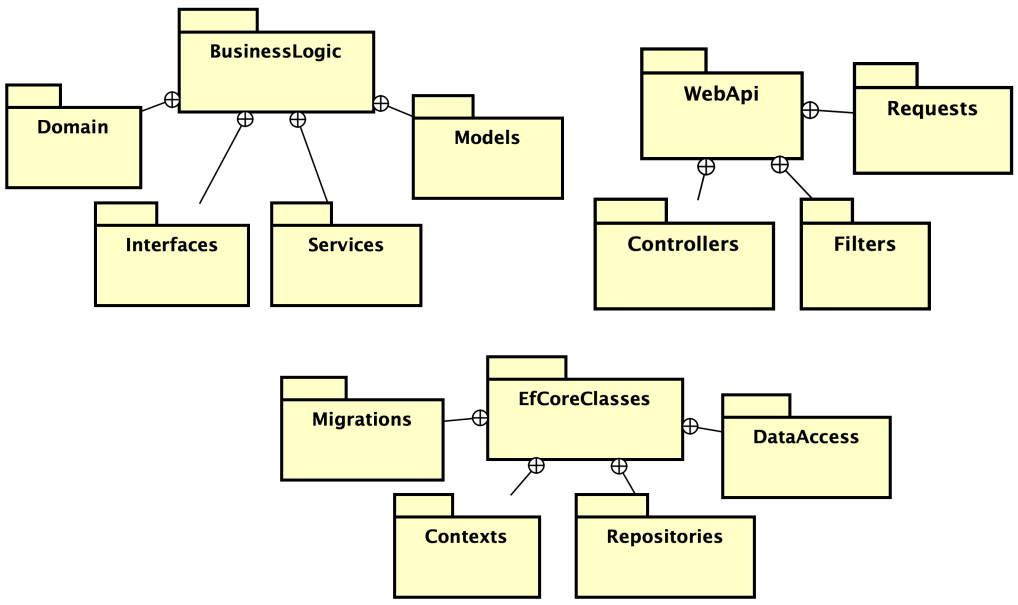
Backend

Para facilitar la comprensión del código, organizamos la solución en dos carpetas principales: SmartHome, que contiene el back-end y la API REST, y SmartHomeWeb, donde se encuentra el front-end. A su vez, el back-end se distribuye en dos carpetas: src y tests.

La carpeta src está compuesta por tres proyectos clave:

- WebApi: aquí se encuentra la capa de presentación, incluyendo los controllers, filters, models y el archivo Program.
- DataAccess: gestiona el acceso a la base de datos mediante EF Core. Define las tablas y sus relaciones, carga los datos iniciales (seedData), y contiene los repositorios que procesan las consultas a la base de datos, adaptando la carga de datos según sea eager o lazy.
- BusinessLogic: alberga la lógica de negocio, las clases de dominio y los servicios.

Cada uno de estos proyectos está dividido en paquetes (como se muestra en los diagramas) para mantener una estructura organizada y prolija, facilitando tanto la navegación como el mantenimiento del código. Además de estos proyectos principales, la solución incluye otros proyectos con responsabilidades más específicas que ayudan a evitar dependencias innecesarias, como los destinados a reflection o DTOs.



Frontend

Dentro del proyecto de Angular tenemos el paquete src y public, dentro de public únicamente dejamos la imagen del icono de la aplicación.

Dentro de src tenemos las siguientes carpetas:

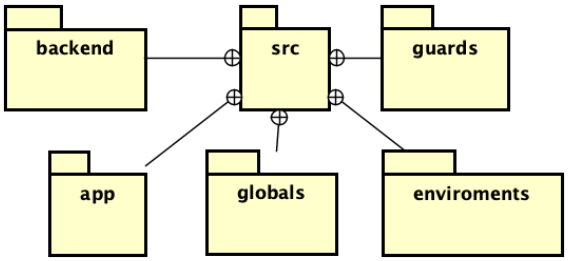
backend: dentro se encuentran nuestros servicios y repositorios, donde nos dan las funcionalidad necesaria

app: se encuentran los componentes y los layouts, donde se encuentran los módulos y permite exportar las funcionalidades.

globals: dentro de ella están los recursos globales del proyecto por ejemplo el id de los roles.

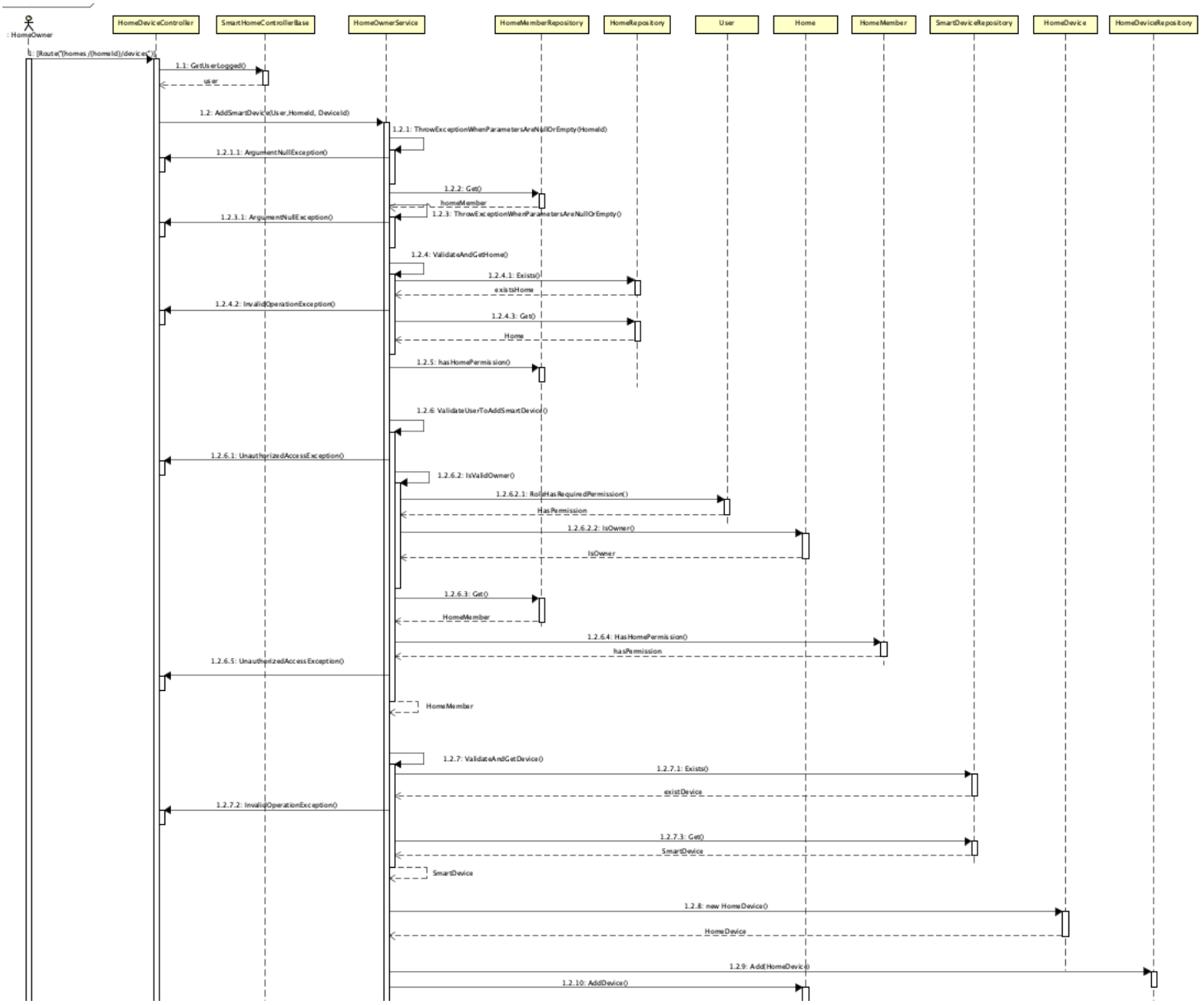
environments: son utilizados para definir configuraciones específicas para diferentes entornos en los que se ejecuta una aplicación.

guards: incluye las clases utilizadas para controlar y restringir el acceso a rutas en la aplicación

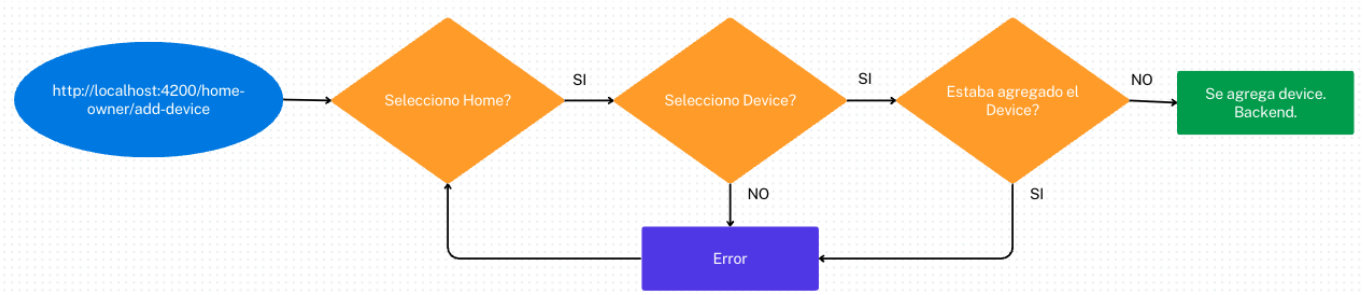


Vista de Procesos:

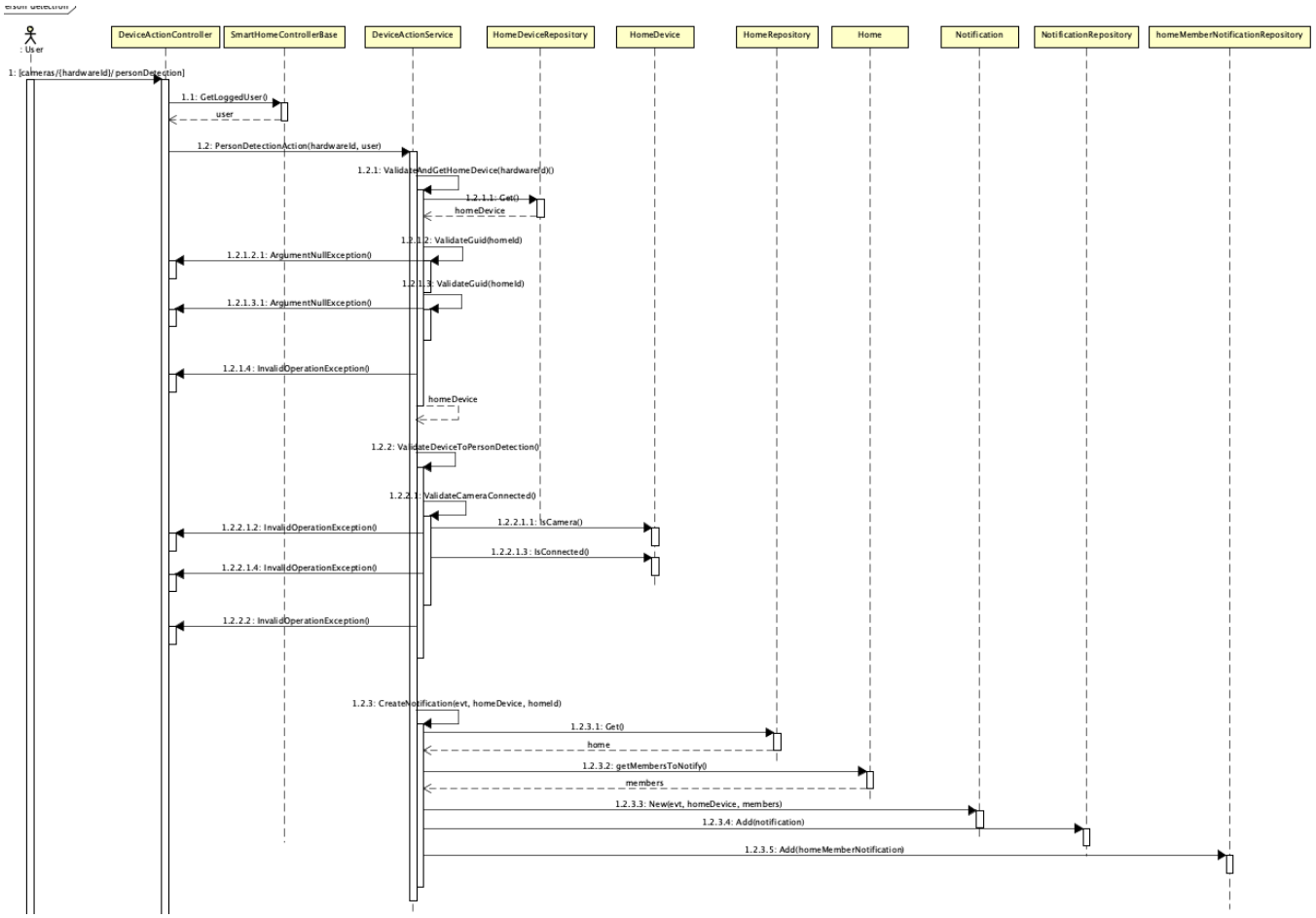
Feature agregar device a una home (backend):



Feature agregar device a una home (frontend):



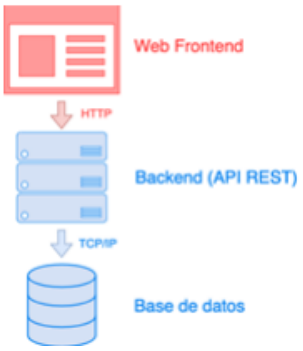
Feature Camera Person detection:



Vista Física:

La solución se divide en tres partes principales:

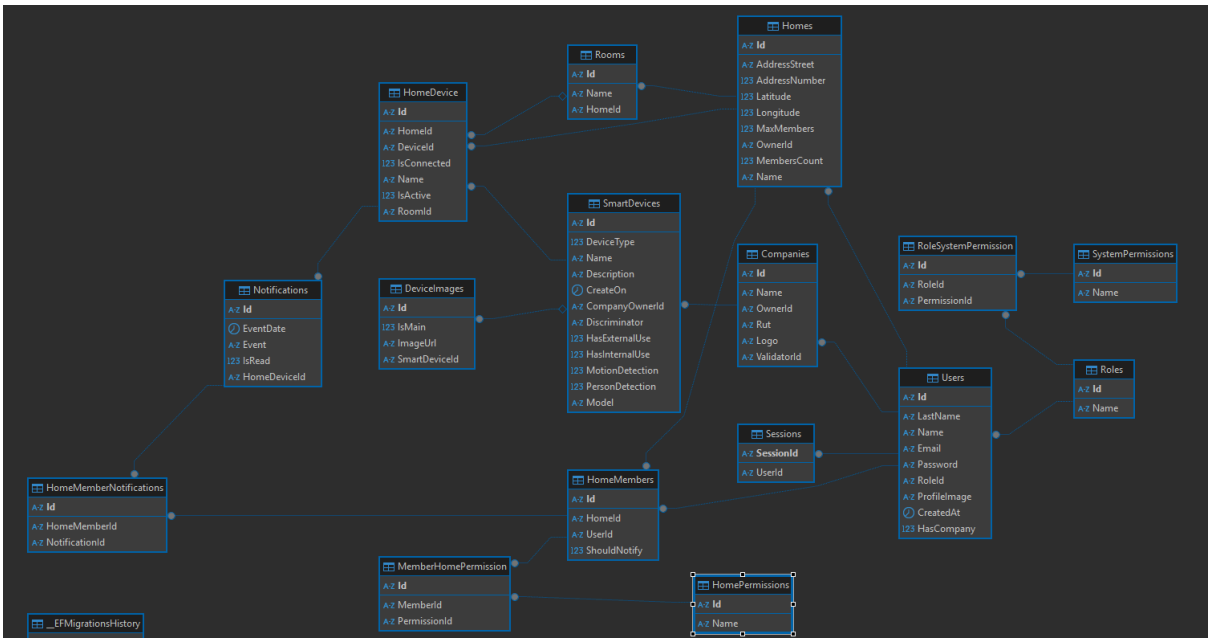
- Base de datos: Implementada como una base de datos relacional usando Entity Framework Core en un enfoque code-first.
- Backend (Api Rest):Aquí se encapsula toda la lógica de la solución, desarrollada en C# con .NET 8.0.
- Web Frontend: Contiene la interfaz de usuario, realizada con Angular 18.2.9.



La base de datos se almacena en un contenedor de Docker, lo que permite portabilidad y facilita la gestión en diferentes entornos. Su vinculación con el backend se gestiona a través de EF Core, encapsulado en el proyecto *DataAccess*. Los repositorios en esta capa realizan las operaciones CRUD, aplicando filtros y paginación antes de recuperar la información de la base de datos.

La comunicación entre el frontend y el backend se realiza mediante API REST utilizando el protocolo HTTP. En el backend, esta lógica está encapsulada en los controllers, que procesan y mapean las solicitudes HTTP, mientras que en el frontend se gestiona en los services, encargados de enviar las peticiones de usuario. Esta conexión sigue el estándar API REST, asegurando una interacción eficiente y organizada entre ambas capas.

La base de datos se creó según lo planificado, siguiendo nuestro diagrama de clases UML y añadiendo las tablas intermedias necesarias para gestionar adecuadamente las relaciones many-to-many. Las tablas intermedias esenciales para el correcto funcionamiento del sistema son: HomeMembersPermission, HomeMemberNotifications y RoleSystemPermission.



Cada entidad en el modelo tiene un atributo Id de tipo GUID, generalmente autogenerado, que se usa como clave primaria de la tabla. Esto garantiza la unicidad de cada registro y mejora la eficiencia de las consultas, ya que normalmente se utilizan los Id para buscar tuplas específicas.

Este diseño no solo optimiza la gestión de relaciones, sino que también mejora la escalabilidad y eficiencia de la base de datos. La elección de GUIDs como claves primarias, en lugar de enteros, aporta una mayor flexibilidad y reduce el riesgo de colisiones en los identificadores, lo cual es especialmente beneficioso en entornos distribuidos.

Para reforzar la seguridad y la integridad de los datos, implementamos restricciones adicionales a nivel de la base de datos. Entre estas medidas, definimos alternate keys en las tablas intermedias para prevenir la existencia de tuplas duplicadas. Esto asegura un manejo más riguroso de las relaciones y mantiene la consistencia de la información.

```
// HomeDevice
modelBuilder.Entity<HomeDevice>() // EntityTypeBuilder<HomeDevice>
    .HasAlternateKey(hd => new { hd.HomeId, hd.DeviceId });

modelBuilder.Entity<HomeDevice>() // EntityTypeBuilder<HomeDevice>
    .HasOne(hd => hd.Home) // ReferenceNavigationBuilder<HomeDevice,Home>
    .WithMany(h :Home  => h.Devices)
    .HasForeignKey(hd => hd.HomeId)
    .OnDelete(DeleteBehavior.Cascade);

modelBuilder.Entity<HomeDevice>() // EntityTypeBuilder<HomeDevice>
    .HasOne(hd => hd.Device) // ReferenceNavigationBuilder<HomeDevice,SmartDevice>
    .WithMany(d :SmartDevice  => d.HomeDevices)
    .HasForeignKey(hd => hd.DeviceId)
    .OnDelete(DeleteBehavior.Restrict);
```

Una posible optimización en este nivel sería el agregado de índices adicionales, lo que permitiría una mejora significativa en la eficiencia de las consultas. Este ajuste podría potenciar el rendimiento del sistema, especialmente en operaciones que involucren búsquedas o filtros complejos.

Con esta mejora, estaríamos reforzando un enfoque orientado tanto a la robustez como al rendimiento, garantizando una base aún más sólida y preparada para escalar en el largo plazo.

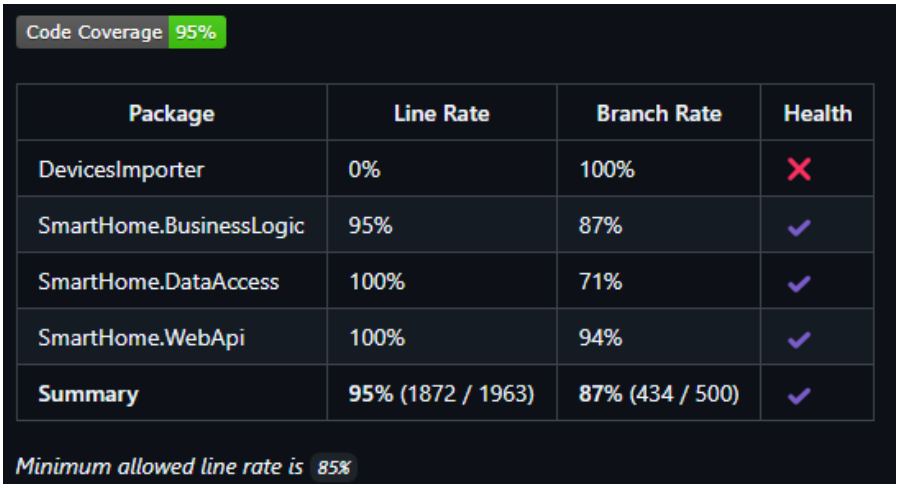
A pesar de ello, el diseño actual ya es bastante robusto, cumpliendo con los estándares establecidos y mostrando un desempeño confiable en las pruebas realizadas.

Anexo

TDD y Cobertura:

La solución se desarrolló siguiendo un enfoque riguroso de Test-Driven Development (TDD) en la mayoría de las etapas del proyecto. Esto nos permitió alcanzar una cobertura del 100% en casi todo el código. Sin embargo, el 5% restante corresponde a la funcionalidad de importación de dispositivos, implementada mediante reflection. Debido a las dificultades técnicas para realizar pruebas sobre esta característica y al limitado tiempo disponible antes de la fecha de entrega, decidimos priorizar el correcto funcionamiento del requerimiento sobre el proceso de TDD.

A pesar de esta excepción, consideramos que logramos un proyecto sólido en términos de pruebas unitarias, incluyendo casos para escenarios límite y asegurando la calidad del software en la gran mayoría de sus componentes.


Code Coverage 95%

Package	Line Rate	Branch Rate	Health
DevicesImporter	0%	100%	✗
SmartHome.BusinessLogic	95%	87%	✓
SmartHome.DataAccess	100%	71%	✓
SmartHome.WebApi	100%	94%	✓
Summary	**95%** (1872 / 1963)	**87%** (434 / 500)	✓

Minimum allowed line rate is 85%

API

HOME CONTROLLER

- **CreateHome**: Crea un nuevo hogar.
    - Ruta: POST /homes
    - Body: CreateHomeRequest:{ "name": , "addressStreet": , "addressNumber": , "latitude": , "longitude": , "maxMembers": }
- **AddMember**: Agrega un miembro a un hogar.
    - Ruta: POST /homes/{homeId}/members
    - Body: AddMemberRequest: { "memberEmail": }
- **AddHomePermission**: Agrega un permiso a un miembro del hogar.
    - Ruta: POST /homes/{memberId}/permissions
    - AddHomePermissionRequest: { "permissionId": }

- **AddSmartDevice**: Agrega un dispositivo inteligente a un hogar.
    - Ruta: POST /homes/{homeId}/devices
    - Body: AddDeviceRequest: { "deviceId":}
- **GetHomeMembers**: Recupera los miembros de un hogar.
    - Ruta: GET /homes/{homeId}/members
- **GetHomeDevices**: Recupera los dispositivos de un hogar.
    - Ruta: GET /homes/{homeId}/devices
    - Query: roomId
- **ModifyHomeName**: Modifica el nombre de un hogar.
    - Ruta: PATCH /homes/{homeId}/name
    - Body: UpdateNameRequest: { “name”: }
- **GetMineHomes**: Recupera los hogares del usuario autenticado.
    - Ruta: GET /homes/mine
    -
- **GetHomesWhereIMember**: Recupera los hogares donde el usuario autenticado es miembro.
    - Ruta: GET /homes/member
- **GetPermissions**: Recupera los permisos disponibles para un hogar.

- Ruta: GET /homes/permissions

HOME-HOME DEVICE CONTROLLER

- **ConnectDevice**: Conecta un dispositivo específico.
  - Ruta: PATCH /hardwares/{hardwareId}/connection
- **DisconnectDevice**: Desconecta un dispositivo específico.
  - Ruta: PATCH /hardwares/{hardwareId}/disconnection
- **ModifyHomeDeviceName**: Modifica el nombre de un dispositivo del hogar.
  - Ruta: PATCH /hardwares/{hardwareId}/name
  - Body: UpdateNameRequest : { “name”: }

MEMBER CONTROLLER

- **ActivateMemberNotification**: Activa las notificaciones para un miembro.
  - Ruta: PATCH /members/{memberId}/activateNotification
- **DeactivateMemberNotification**: Desactiva las notificaciones para un miembro.
  - Ruta: PATCH /members/{memberId}/deactivateNotification
- **GetNotifications**: Recupera notificaciones según los filtros especificados.
  - Ruta: GET /notifications
  - Body: FilterNotificationRequest:{ "deviceType": , "date": , "isRead": }

ROOM CONTROLLER

- **AddRoom**: Añade una nueva habitación al hogar especificado.
  - Ruta: POST /homes/{homeId}/rooms
  - Body: AddRoomRequest: { “name”: }
- **AddDeviceToRoom**: Añade un dispositivo a la habitación especificada.
  - Ruta: POST /rooms/{roomId}/devices
  - Body: HardwareIdRequest:{ "hardwareId": }
- **GetRooms**: Recupera las habitaciones de un hogar.
  - Ruta: GET /{homeId}/rooms

ADMIN CONTROLLER

- **CreateAdmin**: Crea un nuevo usuario administrador.
  - Ruta: POST /admins

- **DeleteAdmin**: Elimina un administrador por su ID.
    - Ruta: DELETE /admins/{idAdmin}
- **CreateCompanyOwner**: Crea un propietario de empresa.
    - Ruta: POST /companyOwners
- **GetUsers**: Recupera una lista de usuarios según filtros.
    - Ruta: GET /users
- **GetCompanies**: Recupera una lista de empresas según filtros.
    - Ruta: GET /companies
- **ChangeRoleToAdminHomeOwner**: Cambia el rol del usuario autenticado a AdminHomeOwner.
    - Ruta: PATCH /changeAdminRoles

COMPANY OWNER CONTROLLER

- **CreateCompany**: Crea una nueva empresa.
    - Ruta: POST /companies
- **CreateCamera**: Crea un dispositivo de cámara.
    - Ruta: POST /cameras
- **CreateMotionSensor**: Crea un sensor de movimiento.
    - Ruta: POST /motionSensors
- **CreateSmartLamp**: Crea una lámpara inteligente.
    - Ruta: POST /smartLamps
- **CreateWindowSensor**: Crea un sensor de ventana.
    - Ruta: POST /windowSensors
- **ChangeRoleToCompanyAndHomeOwner**: Cambia el rol del usuario autenticado a CompanyAndHomeOwner.
    - Ruta: PATCH /changeCompanyOwnerRoles
- **GetModelValidators**: Recupera una lista de validadores de modelos.
    - Ruta: GET /modelValidators
- **ImportDevices**: Importa dispositivos con el ID DLL especificado.
    - Ruta: POST /{dllId}/importDevices
- **GetDeviceImporters**: Recupera una lista de importadores de dispositivos.
    - Ruta: GET /deviceImporters

- **GetDeviceImporterParameters**: Recupera parámetros de un importador de dispositivos por ID DLL.
    - Ruta: GET /deviceImporters/{dllId}/parameters

DEVICE ACTION CONTROLLER

- **PersonDetectionAction**: Envía una notificación de detección de persona para una cámara.
    - Ruta: POST /cameras/{hardwareId}/personDetection
- **CameraMovementDetectionAction**: Envía una notificación de detección de movimiento para una cámara.
    - Ruta: POST /cameras/{hardwareId}/movementDetection
- **OpenWindowSensor**: Notifica que el sensor de ventana está abierto.
    - Ruta: PATCH /windowSensors/{hardwareId}/open
- **CloseWindowSensor**: Notifica que el sensor de ventana está cerrado.
    - Ruta: PATCH /windowSensors/{hardwareId}/close
- **TurnOnSmartLamp**: Enciende una lámpara inteligente.
    - Ruta: PATCH /smartLamps/{hardwareId}/turnOn
- **TurnOffSmartLamp**: Apaga una lámpara inteligente.
    - Ruta: PATCH /smartLamps/{hardwareId}/turnOff
- **MotionSensorMovementDetection**: Notifica detección de movimiento para un sensor de movimiento.
    - Ruta: POST /motionSensors/{hardwareId}/movementDetection

SESSION CONTROLLER

- **Login**: Autentica un usuario e inicia sesión.
    - Ruta: POST /sessions
- **Logout**: Cierra sesión del usuario autenticado.
    - Ruta: DELETE /sessions

UNKNOWN USER CONTROLLER

- **CreateHomeOwner**: Crea un usuario propietario de hogar.
    - Ruta: POST /homeOwners

USER CONTROLLER

- **GetDevices**: Recupera una lista de dispositivos según filtros.
    - Ruta: GET /devices
- **GetDevicesTypes**: Recupera una lista de tipos de dispositivos.
    - Ruta: GET /deviceTypes
- **ModifyProfileImage**: Modifica la imagen de perfil del usuario autenticado.
    - Ruta: PATCH /users/profileImage