

OBLIGATORIO 2

DISEÑO DE APLICACIONES 2

Universidad ORT Uruguay

Pedro Azambuja 270218

Martin Felder 248264

Tomas Zelaschi 281344

Link al repositorio:

<https://github.com/IngSoft-DA2/270218-281344-248264>

Índice

Descripción General.....	4
Correcciones con respecto al primer obligatorio.....	4
Errores conocidos.....	5
Oportunidades de mejora.....	5
Deuda técnica.....	6
Supuestos.....	6
Vista de implementación.....	6
Componentes.....	7
Vista Lógica.....	7
Implementación de los nuevos requerimientos.....	7
Asignar nombre al hogar.....	7
Nombre custom a los dispositivos.....	7
Administradores y dueños de empresa como dueños de hogar.....	8
Selección de lógica de validación de modelo.....	8
Nuevos tipos de dispositivos.....	9
Agrupar dispositivos por cuarto.....	9
Importar dispositivos inteligentes.....	9
Responsabilidades por paquete.....	10
Domain.....	10
Constants.....	10
CustomExceptions.....	10
DeviceImporter.....	11
DTOs.....	11
ExtraRepositoryInterfaces.....	11
GenericRepositoryInterface.....	12
InitialSeedData.....	12
Controllers e Interfaces.....	13
LoadAssembly y Model Validators.....	13
Services.....	13
Repositories.....	14
Vista de procesos.....	14
Métricas de diseño.....	14
Inestabilidad y cohesion a nivel de componentes.....	15
Inestabilidad y cohesion a nivel de namespaces.....	17
Patrones.....	18
Principios de diseño.....	19
SOLID.....	19
GRASP.....	20
Jerarquía de herencias.....	22
Mecanismo para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio.....	22
Instructivo para importar dispositivos.....	23
Instructivo para seleccionar un nuevo modelo validador.....	24

Resumen de las mejoras de diseño.....	24
Especificación de la API.....	25
Modelo de tablas de la base de datos.....	27
Análisis de la cobertura de los tests.....	27
Anexo.....	29
Diagrama de clases Domain.....	29
Diagrama de secuencia creación de una notificación de detección de persona.....	30
Diagrama de secuencia borrado de un administrador.....	30
Diagrama de secuencia crear un cuarto.....	31
Nuevos endpoints.....	31
Endpoints modificados.....	35

Descripción General

SmartHome es una aplicación con una arquitectura cliente-servidor para administrar dispositivos inteligentes en los hogares en la que los usuarios pueden utilizarla mediante un front-end implementado con angular. Este front-end se comunica con la WepApi que implementamos en la entrega anterior y la modificamos para aceptar las nuevas funcionalidades.

Esta aplicación permite diferenciar los tipos de usuario y en base a esto que puedan acceder a distintos tipos de funcionalidades. Para esto utilizamos los tokens que nos permiten identificar que tipo de usuario está logueado en base a los roles, también utilizamos los permisos asociados al rol para autorizar acceso a pantallas.

Hay 4 tipos de usuario en el sistema, los administradores, los dueños de hogar, los dueños de empresa y los miembros de hogar.

El administrador puede crear o borrar otras cuentas de administrador, crear cuentas de dueños de empresas, listar todas las cuentas y listar todas las empresas.

El dueño de empresa puede crear una única empresa asociada a su nombre, y registrar distintos tipos de dispositivos como cámaras de seguridad, lámparas inteligentes, sensores de ventana y sensores de movimiento.

El dueño de hogar puede listar todos los dispositivos disponibles, listar todos los tipos de dispositivos disponibles, crear una casa, agregar una casa, listar sus casas, cambiarle el nombre a una casa, asignarle un nombre a un dispositivo de su casa, crear un cuarto y agrupar dispositivos por cuarto.

El miembro de hogar con los permisos asociados correspondientes puede agregar un miembro al hogar, agregar un dispositivo al hogar, listar los dispositivos asociados al hogar, recibir notificaciones de los dispositivos asociados al hogar, asignar nombre al hogar, asignar un nombre a los dispositivos del hogar.

Los administradores y los dueños de empresa pueden operar como dueños de hogar si así lo desean manteniendo sus permisos que tenían anteriormente.

Correcciones con respecto al primer obligatorio

Un error que tuvimos en la primera entrega fue no crear un proyecto para testar el proyecto de DataAccess pero para esta segunda entrega lo hicimos.

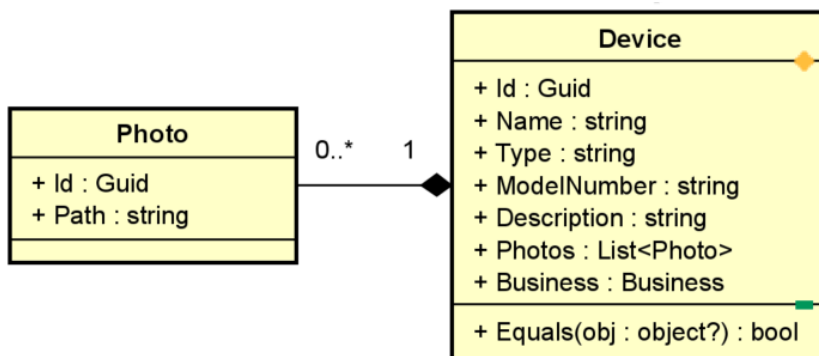
Otro error que tuvimos fue que el método que utilizamos para manejar las custom exceptions era un if extenso con cada caso y su mensaje y código de error asociado pero para esta lo cambiamos por un método que es un diccionario y a partir del tipo de excepción que es el mensaje que devuelve.

Otra cosa que tuvimos que corregir fue el manejo del filtrado y la paginación, nosotros lo hacíamos directamente en los controllers pero esto era muy poco eficiente ya que cada vez que queríamos mostrar por ejemplo 10 dispositivos, en vez de traernos solo esos 10 de la base de datos nos traíamos todos los dispositivos y ahí seleccionamos 10.

También corregimos el uso de magic strings pasandolos a variables con nombres descriptivos para que sea más mantenible en un futuro.

Otra corrección que hicimos fue el manejo de imágenes, en la primera entrega teníamos un atributo Photo de tipo string que solo permitía que el dispositivo tenga una foto asociada. Para esta segunda entrega creamos una clase Photo la cual tiene un atributo Id y un atributo Path, esta tiene una relación de composición con Device para que el dispositivo pueda tener varias fotos asociadas.

Un error que tuvimos en la primera entrega fue que en un método llamábamos dentro de un for each a la base de datos mediante un update. Esto nos generaba un problema de performance ya que la base de datos recibe muchísimas solicitudes que podría provocar que se sature. Para solucionarlo creamos un método el cual recibe los elementos actualizados. Los mapea a sus correspondientes en la base de datos y hace un solo save changes. Haciendo esto le pegamos una sola vez a la base de datos.



Errores conocidos

- No validamos que la latitud y longitud este en los valores válidos
- No mostramos las fotos de perfil de los usuarios
- Tuvimos que hacer fast-forward a main porque tenemos un error al tener duplicados los archivos generados para testear los validadores. Para correr los tests habria

Oportunidades de mejora

Un aspecto en el cual nuestro proyecto puede mejorar es que tenemos un alto acoplamiento al servicio HomeService. En parte está justificado ya que es un proyecto basado en gestión de casas pero nos da como resultado un sistema frágil. Como consecuencia de esto en el repository en los métodos Find y FindAll nos vimos con la necesidad de agregar varios .Include() y ThenInclude() por lo que cada vez que se llaman a estos métodos se está trayendo información la cual puede ser no pertinente a la llamada. Esto genera una carga sobre la base de datos la cual es innecesaria. Podríamos haber hecho métodos Find y FindAll los cuales sean más específicos o creado repositorys para todas las entidades.

Algo que optamos por no implementar es un endpoint el cual dinámicamente valide los importers disponibles y le diga al FrontEnd cuales están disponibles. Si bien agregamos el atributo DIIName para hacerlo optamos por no hacerlo ya que tuvimos la impresión de los

profesores que no era necesario porque no se iba a validar. Elegimos hardcodear en el front-end una única opción posible (JSON).

Como comentamos en la primera entrega, tuvimos problemas con una migración y quedaron mal las tablas. Tratamos de arreglarlo pero no pudimos, entonces tuvimos que borrar las migraciones y hacer nuevas, es por eso que el historial de migraciones es desde que arrancó la segunda entrega hasta ahora.

Si un admin se borra a él mismo la sesión no se cierra, pero una vez que cierra sesión no puede volver a entrar.

Las mayúsculas y minúsculas afectan a la hora de loguearte con el mail.

Deuda técnica

No implementamos el filtro de notificaciones. Nos dimos cuenta que nos faltaba esto el último día relejendo la letra y elegimos no implementarlo para no tener problemas en el código.

No mostramos el logo de las cuentas de los home owners

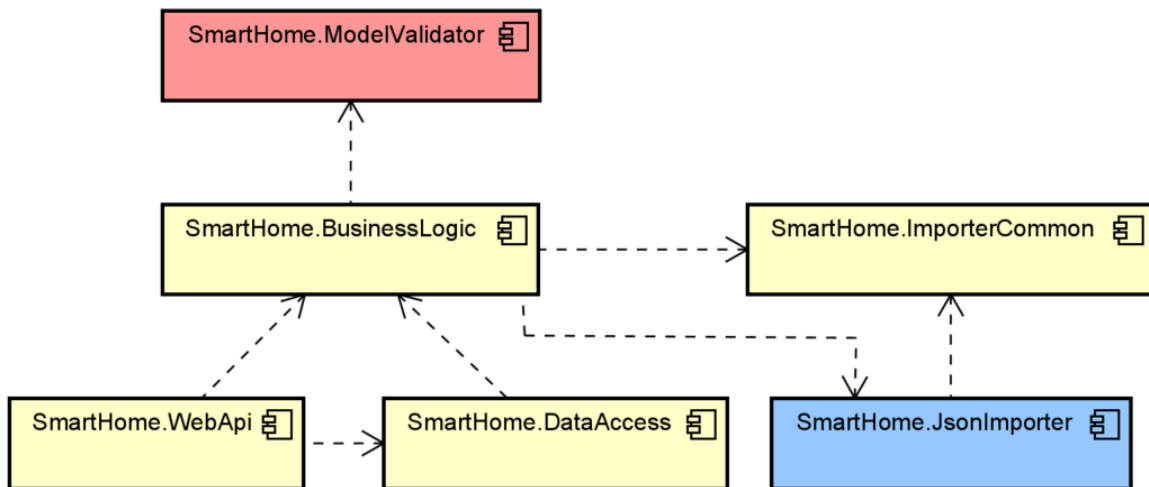
Supuestos

- El custom name no se asigna cuando el dispositivo se asocia a la casa sino que una vez que está asociado está la opción de asignarle un nombre
- Solo el dueño de la casa puede crear cuartos pero cualquier miembro puede agregar un dispositivo a un cuarto
- Solo el dueño de la casa puede asignar permisos al resto de los miembros
- El permiso para ver las notificaciones de la casa se hace para todos los dispositivos y no para cada uno en específico.
- La importación de los dispositivos se puede hacer solo una vez ya que tomamos el id que viene en la base de datos, esto lo hacemos para que no haya dispositivos repetidos en la base de datos.
- Si hay más de un administrador registrado un admin se puede borrar a el mismo.
- Tiene que haber un admin o un admin home owner en todo momento. Puede no haber ninguna cuenta siendo solo administrador.

Vista de implementación

Al igual que en la primera entrega, organizamos la documentación siguiendo el modelo 4+1 visto en el curso. A continuación se detallaremos 3 de las 5 vistas desde la de mayor nivel a la de menor.

Componentes



Acá se puede ver el diagrama de componentes. Como se puede ver, se mantienen las mismas dependencias que en la entrega pasada entre SmartHome.BusinessLogic, SmartHome.WebApi y SmartHome.DataAccess. Ahora se agregó una dependencia desde SmartHome.BusinessLogic a SmartHome.ImporterCommon y a SmartHome.JsonImporter. SmartHome.JsonImporter esta de color azul porque es un servicio de terceros que se utiliza en tiempo de ejecución y no es un desarrollo nuestro sino de terceros. SmartHome.ModelValidator es un servicio de terceros que también se utiliza en tiempo de ejecución por eso esta en color rojo, la lógica no tiene una dependencia al paquete en sí mismo sino una referencia a la dll ModeloValidador.Abstracciones.dll que este contiene.

Vista Lógica

Implementación de los nuevos requerimientos

Asignar nombre al hogar

Para asignar el nombre al hogar, el cambio fue bastante sencillo, agregamos un atributo Name a la clase Home. En el endpoint de CreateHome agregamos el atributo name para pasar también en la creación e hicimos un nuevo endpoint UpdateHomeName para modificarle el nombre a la casa. Este endpoint tiene el verbo PATCH ya que solo estamos modificando un atributo de la entidad y no toda la entidad entera.

Nombre custom a los dispositivos

Este cambio fue muy similar al de la casa, agregamos un atributo Name a la clase HomeDevice así dos dispositivos del mismo modelo pero en casas distintas podían tener distinto nombre. Al principio habíamos puesto el atributo Name en la clase Device pero nos dimos cuenta que eso era un error porque si modificamos el nombre en una casa este cambio se iba a ver en todas las casas que tienen ese mismo modelo.

Administradores y dueños de empresa como dueños de hogar

Para implementar este requerimiento no tuvimos muchas complicaciones debido a cómo manejamos los permisos en la primera entrega. Como habíamos mencionado, nosotros lo que hicimos fue una clase `role` que era un agrupador de permisos, entonces para asignarle los permisos de dueño de hogar a los administradores y a los dueños de empresa creamos dos roles nuevos llamados “Admin Home Owner” y “Business Owner Home Owner” y a los usuarios que querían operar como dueños de hogar se les asignaba esos nuevos roles que contenían los permisos correspondientes de ambos roles juntos.

Selección de lógica de validación de modelo

Para implementar el requerimiento de la validación de los modelos de los dispositivos tuvimos que crear 3 proyectos nuevos en la solución y hacer algunos cambios en el sistema. 2 de los proyectos que creamos fueron para armar los ddl de los validadores, el proyecto `SmartHome.SixLettersValidator` se creó para implementar la validación de 6 letras y el `SmartHome.ThreeLettersThreeNumbersValidator` se creó para implementar la validación de 3 letras al principio y 3 números al final. Para que implementen la interfaz `IModeloValidador`, tuvimos que hacer que estos proyectos tengan la referencia a la ddl “`ModeloValidador.Abstracciones.ddl`” la cual estaba alojada en otro proyecto que creamos llamado `SmartHome.ModelValidator`. Una vez que le agregamos la referencia a esa ddl las clases de ambos proyectos pudieron implementar la interfaz.

Para poder persistir los validadores, creamos una clase en el dominio llamada `Validator` la cual tiene como atributos el nombre del validador y el id y un repositorio llamado `ValidatorRepository`. Dentro de la clase `Business` creamos un atributo `ValidatorId` de tipo `Guid` que hace referencia al id del validador que la empresa va a utilizar. Igualmente, el chequeo de si el validador existe lo hacemos en tiempo de ejecución buscando en la carpeta `ModelValidators` que es donde se encuentran los dll de los validadores. No lo hacemos yendo a la base de datos ya que si un usuario borra la dll del proyecto no nos enteramos.

Los ddl en los que recibimos los validadores los alojamos en una carpeta dentro del proyecto `SmartHome.BusinessLogic` llamada `ModelValidators` como mencionamos anteriormente. Para poder cargar estos validadores, creamos un servicio llamado `ValidatorService` que tiene el método `LoadAssemblyAndGetImplementations` y carga todos los ddls que estén implementando la interfaz, una vez que los carga los guarda en la tabla de `Validators` con el nombre del archivo. El método `LoadAssemblyAndGetImplementations` utiliza la clase `LoadAssemblyClass` que contiene los métodos `GetImplementations` que devuelve la lista de validadores que implementan la interfaz y el método `GetImplementation` que recibe un nombre de un validador y crea la instancia si este se encuentra en la lista de implementaciones. Es importante que si se quiere agregar un ddl nuevo para incluir un nuevo validador, este se aloje en esta carpeta que creamos. El dll `Modelo Validador` está en español, por eso en `ValidatorService` encapsulamos todo lo que está en español para que en el resto del proyecto quede consistente el idioma que decidimos utilizar que es el inglés y no se mezcle.

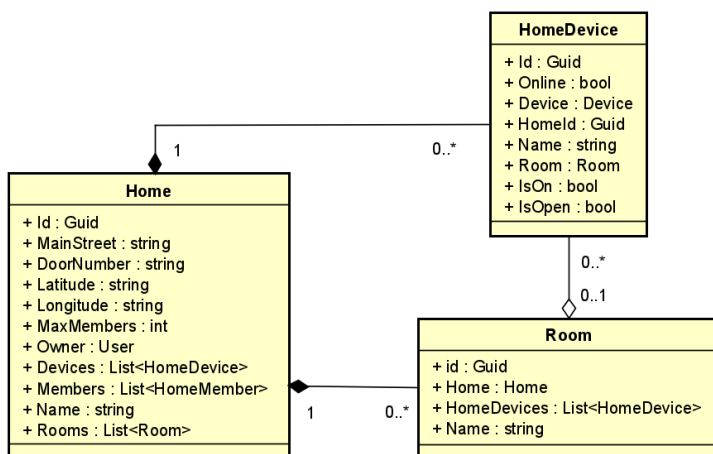
Nuevos tipos de dispositivos

Para este nuevo requerimiento, primero creamos dos clases nuevas, `IntelligentLamp`, con el atributo `IsOn` y `WindowSensor`, con el atributo `IsOpen` que

heredaban de Device. Pero después nos dimos cuenta que nos iba a pasar lo mismo que con el nombre de los dispositivos, si el atributo `IsOn` estaba en `true` a todos los `homeDevices` de ese tipo les iba a aparecer en `true`. Para corregir esto, decidimos eliminar esa herencia y en `HomeDevice` agregar los atributos nulleables `IsOn` e `IsOpen` y dependiendo el tipo de dispositivo se asignaban los atributos o no.

Agrupar dispositivos por cuarto

En este requerimiento al principio quisimos hacer un atributo nooleable `room` de tipo `string` en `HomeDevice` y si se asignaba un `room` a cierto `HomeDevice` se modificaba el atributo pero nos dimos cuenta que esto no iba a servirnos ya que no íbamos a poder filtrar por cuarto y no era bueno tener el `room` de tipo `string` para el manejo. Entonces decidimos hacer una clase `Room` que tiene como atributos `Home`, `Name` y `HomeDevices` que es la lista de los `HomeDevice` asociados a ese cuarto. Esta clase `Room` tiene una relación de agregación con `HomeDevice` y una relación de composición con `Home`.



Importar dispositivos inteligentes

Para importar los dispositivos inteligentes, agregamos dos proyectos nuevos a la solución. Al primero le llamamos `JSONImporter`, dentro de este hay una carpeta con las clases `Dispositivo`, `Foto` y `Root` que representan la información tal cual como viene del json. Después definimos otra clase llamada `IDeserializer` que tiene un método `Deserialize` que sirve para tomar datos de un archivo o una cadena de texto que están en un formato específico (como JSON, XML o binario) y convertirlos en un objeto de C#. Hicimos la interfaz por si en un futuro se quiere pasar un archivo XML, la clase define la interfaz para el método `Deserialize` de XML. Después definimos la clase `JSONDeserializer` que implementa la interfaz y tiene el método `Deserialize` para json. Por último, tenemos la clase `JSONImporterClass` que la vamos a explicar después de explicar el otro proyecto para que tenga contexto el uso de esta.

El otro proyecto que definimos es `SmartHome.ImporterCommon`, éste hace de "contrato" el cual especifica que debe implementar el proyecto el cual se quiera agregar como compilado en la carpeta `ImporterDLLs`. También cumple la función de proveer al `businessLogic` la interfaz por la cual va a acceder al compilado. En este proyecto tenemos una clase `DTODevice`, `DTOPhoto` y la interfaz `IDeviceImporter` que cuenta con el método `ImportDevicesFromFile`. La clase `JsonImporterClass` implementa esta interfaz. En este

método se llama al `Deserialize` para poder leer el json y después se llama al método `MapDispositivosToDTODevice` que recibe la data del json deserealizada y una lista vacía de `DTODevice` y agrega a esa lista los dispositivos convertidos en DTOs.

Después en la lógica del negocio creamos una carpeta llamada `DeviceImporter` la cual está compuesta por 3 carpetas, la primera es `DeviceFiles` y es donde se tiene que guardar el json con los dispositivos, la segunda es `ImporterDLLs` y tiene el dll del `JSONImporter` que hablamos previamente y un dll llamado `Newtonsoft.dll` que lo tuvimos que agregar para que funcione el `deserialize` del `JSONImporter`. Y la tercera es `TypeMap` que contiene una clase con variables constantes de tipo string que representan como recibimos los tipos de los dispositivos en el json.

Como se puede ver en la descripción de los cambios, como el modelo que hicimos en la primera entrega era mantenible no tuvimos que hacer grandes cambios y ninguno requirió

Responsabilidades por paquete

Domain

Este paquete contiene las clases que representan el dominio del sistema. El [diagrama de clases](#) se encuentra en el anexo. Estas clases están diseñadas principalmente como modelos de datos, por lo que, en su mayoría, no incluyen métodos ni lógica compleja. La responsabilidad de gestionar las interacciones y relaciones entre estas clases recae en el paquete `Services`, lo que permite mantener una clara separación de responsabilidades y una arquitectura más modular.

Constants

El paquete `constants` encapsula las constantes que se manejan en la lógica. Esto lo hicimos para evitar el uso de magic strings. Las constantes que se encuentran en este paquete son `User` y `DeviceTypesStatic` que contiene los diferentes tipos de dispositivos disponibles en la aplicación.

CustomExceptions

En el paquete `CustomExceptions` se encuentran las excepciones personalizadas que diseñamos específicamente para manejar situaciones excepcionales en nuestra aplicación. Decidimos dividir las en siete tipos principales, cada una relacionada con una responsabilidad específica en la aplicación: `DeviceException`, `HomeArgumentException`, `HomeDeviceException`, `HomeException`, `RoleException`, `SessionException` y `UserException`. Para esta segunda entrega agregamos `BusinessException`, `DeviceImporterException`, `RoomException` y `ValidatorException`.

El uso de excepciones personalizadas nos permite tener un manejo de errores más claro y específico, lo que facilita la identificación y solución de problemas en distintas partes del sistema. Cada excepción está relacionada con un contexto particular, lo que mejora la escalabilidad y flexibilidad del código, permitiendo añadir nuevas excepciones sin afectar el manejo existente. Además, este enfoque promueve un manejo de errores consistente, ofreciendo mensajes claros y detallados tanto para los usuarios como para los

desarrolladores. Finalmente, facilita el mantenimiento del sistema, ya que los errores están claramente separados por áreas, lo que hace que el código sea más organizado y fácil de gestionar.

También contamos con un paquete de CustomExceptions en DataAccess que es para manejar las excepciones de la base de datos.

DeviceImporter

Este paquete tiene la responsabilidad de manejar la importación de los dispositivos para posteriormente pasar la información a la lógica y que la transforme en entidades del sistema. Está compuesto por tres subpaquetes: DevicesFiles, ImporterDLLs y TypeMap. En DevicesFiles se alojan los dll de los dispositivos a importar. En ImporterDLLs se aloja el dll del importador del formato json, también tuvimos que incluir el archivo Newtonsoft.Json.dll ya que sin él no nos compilaba el JsonImporter. Por último está el paquete TypeMap que contiene la clase JsonTypeMap, esta se encarga de mapear los tipos de dispositivos que recibimos en el json al formato de tipos de dispositivos que usamos en el sistema.

DTOs

En ciertas situaciones por la estructura de nuestro dominio requerimos crear Data Transfer Objects para poder comunicarle a la WebApi la información pertinente. Esta situación se presentó tres veces. La primera fue cuando empezamos a trabajar sobre el front-end y nos dimos cuenta que necesitábamos devolver información adicional en el log-in además del token de autenticación. Esta información es los permisos que son utilizados por las guardas de angular para autorizar el acceso a ciertas páginas. Y como nuestra clase Session del dominio solo almacena el token y el id del usuario recurrimos a crear este DTOSessionAndSystemPermissions. El segundo caso es en Notification. Nosotros modelamos el dominio de las notificaciones de manera de que solo se genere una notificación en la tabla notification y por cada usuario que debía recibirla como un registro en la tabla HomeMemberNotification que almacena si fue leída o no. Esto lo hicimos para no repetir innecesariamente la información de la notificación por cada usuario que debía asignarse la notificación. Pero al momento de proveer al usuario su notificación correspondiente se debe poder distinguir si esta había sido leída o no por lo que creamos el DTONotification para pasarsela. Por último creamos el DTOValidator. Esto lo hicimos porque buscamos el validator por el nombre porque es fijo mientras el id se genera cada vez que se inserta en la base de datos y no devolvemos dominio al front.

ExtraRepositoryInterfaces

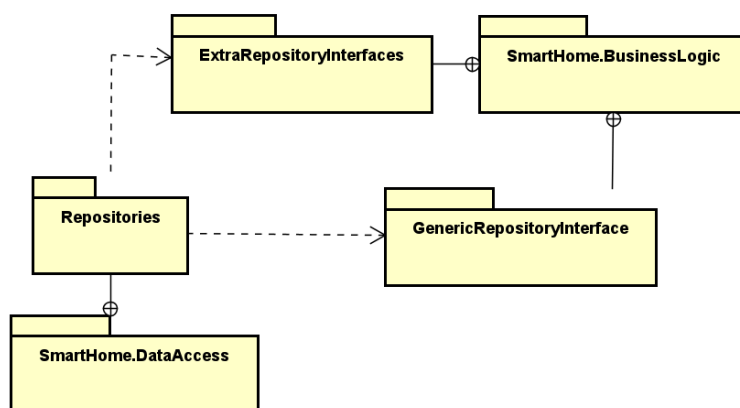
En ciertos casos, los métodos que necesitábamos no estaban contemplados en la interfaz genérica. Específicamente, para los métodos GetAllDeviceTypes y GetAllHomesByUserId, fue necesario crear interfaces adicionales. Esto lo hicimos porque estos métodos son específicos de ciertos repositorios y no tenían sentido en la interfaz genérica. Para poder acceder a estos métodos desde los servicios en BusinessLogic, creamos un paquete separado llamado ExtraRepositoryInterfaces.

La principal ventaja de esta implementación es que la capa BusinessLogic no depende de las implementaciones concretas de los repositorios en DataAccess, sino de las interfaces. Esto facilita la sustitución de implementaciones (por ejemplo, cambiar de una

base de datos SQL a una NoSQL) sin afectar la lógica de negocio, haciendo el sistema más flexible y mantenible. A su vez, la interfaz genérica facilita la reutilización de código en diferentes repositorios. Los métodos estándar de acceso a la base de datos se pueden implementar de manera uniforme en varios repositorios, lo que reduce la duplicación de código y mejora la mantenibilidad.

GenericRepositoryInterface

Para la creación de los repositorios, decidimos implementar una interfaz genérica que define métodos universales para el acceso a la base de datos, los cuales deben ser implementados por todos los repositorios. Optamos por definir esta interfaz dentro del paquete BusinessLogic, para respetar el principio de inversión de dependencias (DIP). Este principio establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino de abstracciones (interfaces). De este modo, el paquete DataAccess (que contiene los repositorios) depende de las abstracciones definidas en BusinessLogic, lo que asegura que DataAccess, como módulo de nivel inferior, dependa de BusinessLogic, que es un módulo de nivel superior.



InitialSeedData

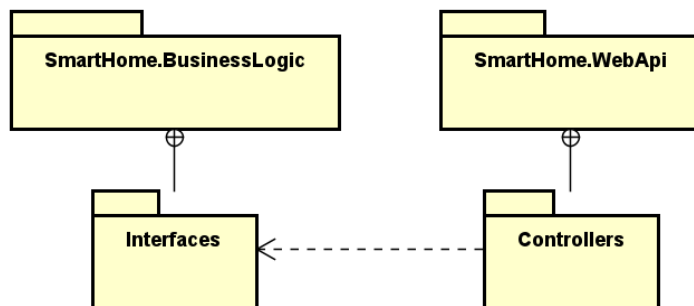
En el paquete InitialSeedData definimos todos los datos esenciales que el sistema necesita para funcionar correctamente. Estos datos incluyen los identificadores de roles, permisos y el administrador inicial, que son fundamentales para establecer la estructura de acceso en la aplicación. Implementamos este paquete dentro de BusinessLogic, ya que estos identificadores son utilizados en los servicios para asignar roles y permisos a los usuarios.

En el paquete Contexts dentro de DataAccess, tenemos el método ConfigSeedData, que se encarga de crear y configurar el administrador, los roles y los permisos a partir de los identificadores definidos en SeedDataConstants. Este método se llama dentro de la función OnModelCreating, que es parte del ciclo de vida de la creación del esquema de la base de datos.

Decidimos hardcodear los identificadores en el archivo SeedDataConstants para garantizar la consistencia de los datos en la base de datos. Si no lo hiciéramos de esta forma, cada vez que se creara una nueva instancia del contexto de base de datos, se generarían nuevos identificadores para los permisos y roles, lo que provocaría una

inconsistencia con los datos que ya están relacionados con los identificadores anteriores. Al mantener los IDs fijos, aseguramos que las relaciones entre entidades (roles, permisos, usuarios) sean consistentes y válidas en todo momento.

Controllers e Interfaces



En este diagrama se puede ver la relación entre los controllers y la lógica, como se ve los controllers dependen de las interfaces lo que refleja el desacoplamiento entre la capa de presentación y la implementación de la lógica. En lugar de interactuar directamente con la implementación de la lógica utiliza la interfaz. Gracias a la inyección de dependencias el controller no tiene que instanciar la lógica.

LoadAssembly y Model Validators

La responsabilidad de estos paquetes las mencionamos cuando explicamos cómo implementamos los cambios

Services

El paquete **Services** contiene las implementaciones de la lógica de negocio, donde se definen los métodos y operaciones que gestionan las principales funcionalidades del sistema. En la primera entrega este paquete consistía de seis clases principales: **BusinessService**, **DeviceService**, **HomeService**, **RoleService**, **SessionService** y **UserService**. Para esta segunda entrega agregamos **DeviceImportService** y **ValidatorService**. Cada una de estas clases implementan las interfaces correspondientes, que están definidas en el paquete **Interfaces**, siguiendo un enfoque basado en la responsabilidad del dominio, más que en los recursos. Esto nos permite agrupar las operaciones por áreas de responsabilidad dentro del sistema, en lugar de tener una clase por cada entidad.

Para manejar la persistencia de entidades, los services hacen uso de los paquetes **GenericRepositoryInterfaces** y **ExtraRepositoryInterfaces**.

Repositories

El paquete **Repository** es el que se comunica con la base de datos, en la primera entrega, en este paquete se encontraban las clases **BusinessRepository**, **DeviceRepository**,

HomeDeviceRepository, HomeMemberRepository, HomePermissionRepository, HomeRepository, RoleRepository, SessionRepository y UserRepository. Para la segunda entrega tuvimos que agregar RoomRepository, SystemPermissionRepository y ValidatorRepository. Todos heredan de la interfaz IGenericRepository, a diferencia de DeviceRepository que también hereda de IDeviceTypeRepository y de HomeRepository que también hereda de IHomesFromUserRepository.

Vista de procesos

Las tres funcionalidades que elegimos mostrar en los diagramas de interacción fueron, son la de creación de una [notificación de detección de persona](#), [borrado de un administrador](#) y la [creación de un cuarto](#).

Métricas de diseño

Una métrica es una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado. Es una forma de medición y escala definidas para realizar mediciones de uno a varios atributos.

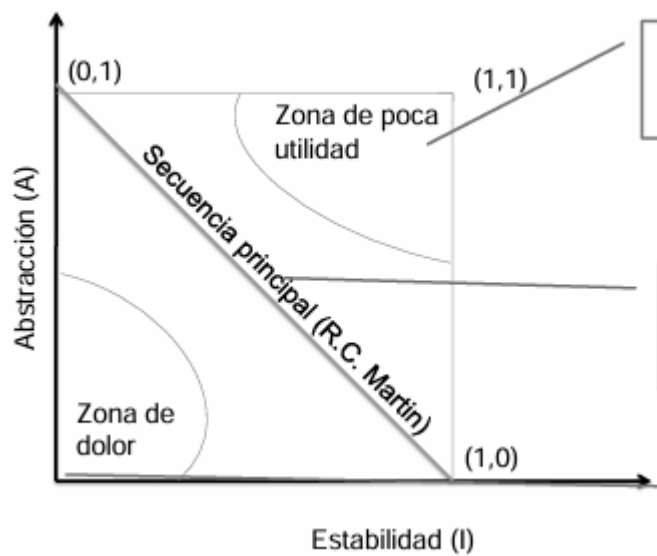
En el curso nos centramos en las métricas de inestabilidad (I), abstracción (A), cohesión relacional (H) y la distancia (D o distancia normalizada D').

La inestabilidad se encuentra como métrica dentro de los principios de dependencias de paquetes referente al Stable Dependencies Principle (SDP) y está relacionada con la cantidad de trabajo requerido para realizar un cambio. Se calcula a partir de las relaciones aferentes (clases fuera del paquete que dependen de alguna clase dentro del paquete) y eferentes (clases fuera del paquete que son dependidas por clases dentro del paquete). Su ecuación es $I = Ce / (Ca + Ce)$ y se encuentra en el rango de 0 a 1 siendo 1 lo más inestable posible.

La abstracción también se encuentra dentro de los principios de dependencias de paquetes referente a Stable Abstraction Principle (SAP) que dice que los paquetes estables deberían ser abstractos. Se calcula a partir de las clases abstractas o interfaces en el paquete (N_a) y las totales (N_c). Su ecuación es $A = N_a / N_c$ y también varía entre 0 y 1 siendo 1 lo más abstracta posible.

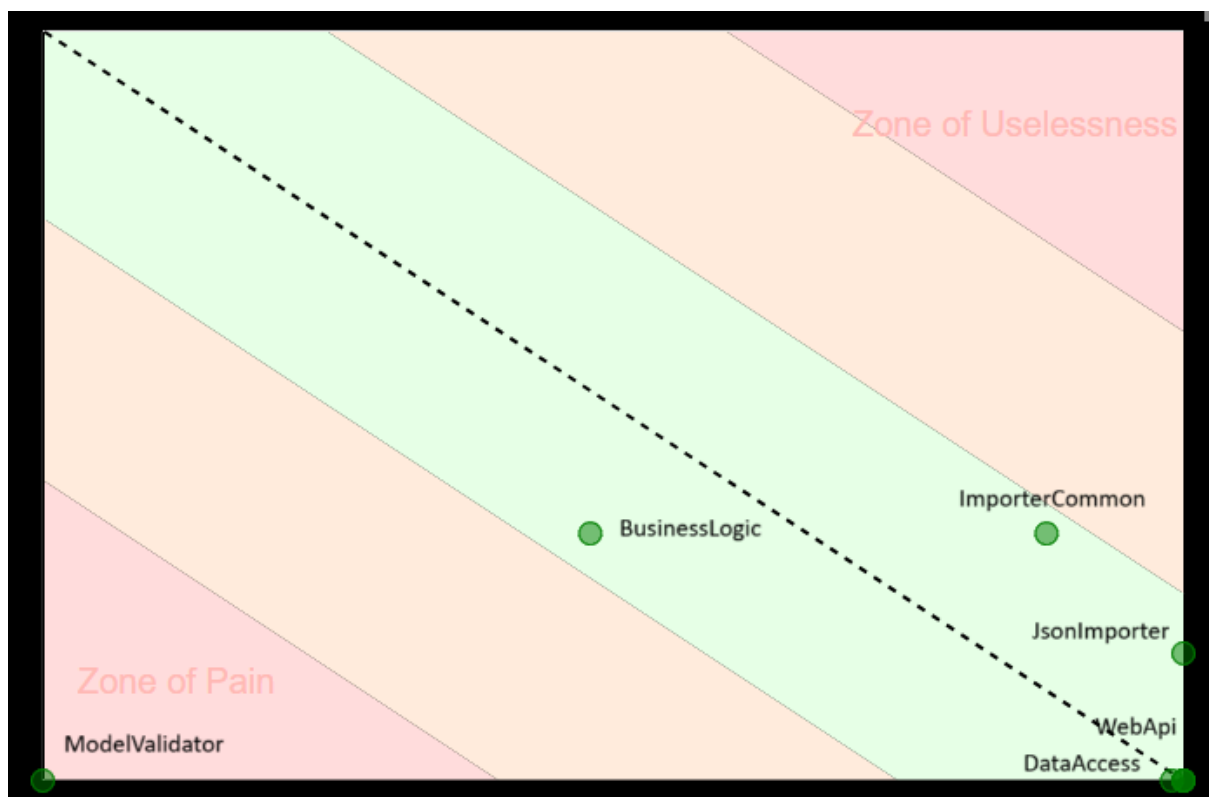
La tercera métrica se trata de la Cohesión relacional (H) que mide la relación entre las clases de un paquete. Para calcularla se necesitan la cantidad de relaciones internas (R) y cantidad de clases e interfaces dentro del paquete (N). Su ecuación es la siguiente $H = (R + 1) / N$. Es considerada buena cohesión relacional cuando se encuentra entre 1,5 y 4.

A partir de las dos primeras métricas vistas podemos calcular la distancia. La distancia se trata de una métrica que relaciona la inestabilidad con la abstracción de los paquetes. Se calcula de dos maneras $D = |A + I - 1| \div \sqrt{2}$ esta varía entre 0 y 0,707 y la normalizada $D' = |A + I - 1|$ que varía entre 0 y 1, siendo en ambas lo más ideal lo más cercano a 0 posible. Una vez obtenidas las distancias podemos realizar la gráfica a continuación.



Los puntos ideales son las dos esquinas en la zona de la secuencia principal y los peores son la zona de poca utilidad que no le encuentra sentido al paquete y la zona de dolor que significa que dicho paquete se encuentra muy acoplado.

Inestabilidad y cohesion a nivel de componentes



8 assemblies	Abstractness	Instability	Relational cohesio
8 assemblies matched			
SmartHome.ImporterCommon	0.33	0.88	1
SmartHome.BusinessLogic	0.33	0.48	2.18
SmartHome.DataAccess	0	0.99	1.59
SmartHome.WebApi	0	1	1.68
SmartHome.JSONImporter	0.17	1	1.5
SmartHome.ModelValidator	0	0	0
SmartHome.ThreeLettersThreeNumbersVal	0	1	1
SmartHome.SixLettersValidator	0	1	1

Nosotros seguimos el criterio de Clean Architecture para crear los componentes del sistema. Este enfoque promueve la separación de responsabilidades, organizando la aplicación en capas independientes que interactúan entre sí a través de interfaces. De esta manera, se reduce el acoplamiento entre los distintos niveles del sistema, se mejora la modularidad y se facilita tanto la extensibilidad como las pruebas unitarias.

En la tabla, podemos observar que el componente BusinessLogic es notablemente más estable en comparación con otros componentes del sistema. Esto es lógico, ya que esta capa encapsula las reglas de negocio fundamentales y, como resultado, no depende en gran medida de otros componentes, manteniendo un bajo nivel de inestabilidad (0.48). Este diseño se apoya en un enfoque centrado en la inyección de dependencias, lo cual incrementa la modularidad y facilita las pruebas unitarias. Se puede ver que se cumple el principio de dependencias estables ya que WebApi depende de BusinessLogic y DataAccess que son más estables que él y DataAccess depende de BusinessLogic que es el más estable.

Por otro lado, los componentes DataAccess y WebApi tienen valores de abstracción bajos (0), lo que indica que están compuestos por clases altamente concretas y específicas. Esto es coherente con su propósito: el primero gestiona operaciones relacionadas con la persistencia de datos, mientras que el segundo se enfoca en la comunicación con las interfaces de usuario y otros sistemas. Estos componentes están diseñados para interactuar directamente con tecnologías específicas, lo que explica su falta de abstracción. Acá se puede ver que se cumple el principio de abstracciones estables ya que son paquetes inestables y concretos.

Finalmente, la abstracción intermedia de BusinessLogic (0.33) refleja un equilibrio entre elementos concretos y abstractos. Esto se debe a que, si bien implementa directamente ciertas reglas de negocio, también utiliza interfaces y patrones orientados a abstraer dependencias externas, favoreciendo la extensibilidad y la mantenibilidad.

En cuanto a Relational Cohesion, los valores rondan entre 1.5 y 2.18 en todos los componentes menos en ModelValidator esto indica que las clases dentro de cada ensamblado tienen una cohesión adecuada, trabajando juntas de manera efectiva para lograr sus objetivos funcionales.

En resumen, el diseño mostrado en la tabla resalta un enfoque arquitectónico que promueve estabilidad en la lógica del negocio, manteniendo componentes concretos en las

capas tecnológicas para un acoplamiento claro con herramientas específicas y alta cohesión interna.

ModelValidator está en la zone of pain ya que en este paquete lo único que se aloja es la dll "ModeloValidador.Abstracciones.dll". Las implementaciones del método EsValido en los validadores las incluimos en la entrega para mostrar cómo formamos las dll pero podrían no estar.

Los validadores no los incluimos en el análisis ya que los dejamos para mostrar cómo formamos las dlls.

Inestabilidad y cohesion a nivel de namespaces

Paquete	Inestabilidad	Abstracción	Cohesión Relacional	Distancia
Constants	0	0	0	1
CustomExceptions	0	0	0	1
DeviceImporter	0	0	0	1
Domain	0	0	1,72	1
DTOs	0,16	0	0	0,84
ExtraRepository Interfaces	0,2	1	0	0,2
GenericRepositoryInterface	0	1	0	0
InitialSeedData	0	0	0	1
Interfaces	0.3	1	0	0,3
LoadAssembly	1	0	0	0
ModelValidators	0	0	0	1
Services	0,95	0	0,5	0,05
Filters	0,9	0	0	0,1
WebModels	0,9	0	0	0,1
Controllers	1	0	0	0
Repositories	0,95	0	0	0,05

En la tabla, podemos observar que el único namespace que presenta valores válidos de cohesión relacional es Domain. Esto se debe a que el criterio que utilizamos para dividir los namespaces fue en función de tipos, es decir, estructuras genéricas como Services, Domain, Interfaces, etc., y no en función de flujos o funcionalidades específicas.

Esta decisión se tomó al inicio del proyecto, cuando no conocíamos estas métricas y su impacto en la arquitectura. Para cuando las descubrimos, ya era tarde para reestructurar toda la arquitectura sin comprometer significativamente el desarrollo.

Un enfoque que podría mejorar la cohesión es adoptar una arquitectura vertical, que organiza el sistema en capas orientadas a características o módulos funcionales en lugar de componentes genéricos. Por ejemplo, podríamos haber implementado un paquete llamado Device, que encapsule clases como Device, DeviceService, DeviceInterfaces y otros elementos relacionados directamente con esta funcionalidad. Este diseño no solo mejora la cohesión al agrupar clases relacionadas en un contexto funcional, sino que también facilita el mantenimiento y la escalabilidad del sistema.

Este cambio de enfoque hubiera permitido que las relaciones entre las clases dentro de un namespace fueran más relevantes y coherentes, optimizando las métricas de diseño en especial la cohesión relacional.

Podemos ver que los únicos que se acercan al 0 en la distancia (valor ideal) son Repositories, Controllers, WebModels, Filters, LoadAssembly y ExtraRepositoryInterfaces estos valores tienen sentido por lo que explicamos previamente.

Patrones

En nuestra aplicación, implementamos el patrón de diseño Facade a través del paquete WebApi, que funciona como una fachada para la interacción con el sistema. Este paquete centraliza el acceso a las funcionalidades internas, sirviendo como punto único de entrada y salida. De esta forma, encapsula la complejidad de las capas internas, exponiendo una interfaz simplificada para los clientes externos, mientras delega las solicitudes hacia la lógica de negocio y la capa de persistencia de datos.

Otro patrón que utilizamos fue el observar en el frontend. Un ejemplo en el que lo utilizamos fue en la lista de los homeMembers. En este caso específico, el patrón Observable se implementa en una arquitectura de capas donde el flujo de datos comienza cuando HomeMembersListComponent (Subscriber/Cliente) solicita los miembros de un hogar a través de HomeService (intermediario). El HomeService delega la petición HTTP al HomeApiRepositoryService (Publisher principal), que realiza la llamada HTTP real y crea un Observable usando RxJS. Este Observable encapsula la respuesta HTTP futura. Cuando los datos llegan del servidor, el Observable emite estos datos, que fluyen de vuelta a través de HomeService hasta llegar al componente, donde el método subscribe() maneja la respuesta en sus callbacks next (para datos exitosos) y error (para errores). Este flujo permite que el componente reaccione de manera asíncrona a los datos que llegan del servidor, actualizando su propiedad status.homeMembers cuando recibe nuevos datos, lo cual desencadena una actualización en la vista.

Principios de diseño

SOLID

Se diseñó la aplicación de manera que las responsabilidades estén claramente definidas y bien distribuidas, cumpliendo con el principio de responsabilidad única (Single Responsibility Principle, SRP). Como mencionamos anteriormente, cada capa del sistema tiene su propia responsabilidad específica: la capa de presentación maneja las solicitudes HTTP, la capa de lógica de negocio contiene las reglas y operaciones clave del sistema, y la capa de persistencia interactúa con la base de datos. Esto asegura que cada componente del sistema tenga una única razón para cambiar.

Además, el sistema está diseñado para que los cambios impliquen la mínima modificación de código posible. Por ejemplo, agregar nuevos permisos, roles o incluso nuevos tipos de dispositivos no requiere cambios en la estructura básica del sistema, solo extensiones de las funcionalidades existentes. Esto respeta el principio de abierto/cerrado (Open/Closed Principle, OCP), que establece que las clases deben estar abiertas para extensión pero cerradas para modificación, permitiendo que el sistema evolucione de manera flexible sin afectar el código existente. Se vio evidenciado en esta segunda entrega que tuvimos que crear nuevos roles y permisos y no tuvimos que cambiar ninguna clase.

El principio de sustitución de Liskov (Liskov Substitution Principle, LSP) también se cumple, ya que las interfaces se utilizan de forma que las clases que las implementan pueden ser intercambiadas sin romper la funcionalidad del sistema. Las implementaciones de las interfaces se pueden modificar o cambiar por completo sin que esto afecte a los consumidores de esas interfaces. Esto es fundamental para mantener la estabilidad del sistema cuando se realizan cambios o se agregan nuevas funcionalidades.

Finalmente, como las diferentes capas de la aplicación se comunican a través de interfaces, se cumple el principio de inversión de dependencias (Dependency Inversion Principle, DIP). Los módulos de alto nivel (como los controladores y servicios de negocio) no dependen de módulos de bajo nivel (como los repositorios), sino que dependen de abstracciones (interfaces). Esto facilita la sustitución de implementaciones de bajo nivel sin impactar el funcionamiento de las capas superiores, mejorando la flexibilidad y escalabilidad del sistema.

GRASP

EXPERT

Para cumplir con el patrón expert creamos los services BusinessService, DeviceImportService, DeviceService, HomeService, RoleService, SessionService, UserService y ValidatorService con el objetivo de que el servicio correspondiente a la clase con la información necesaria para realizar una acción sea él quien la ejecute (El experto).

CREATOR

En la mayoría de los casos el creador de instancias en nuestro proyecto son los request models mediante las funciones ToEntity() definidas en ellos. Optamos por ir por este camino ya que "A tiene los datos necesarios para inicializar a B" que son los datos que provee el usuario mediante los formularios del front-end. Si no se creaban todas las instancias necesarias para la creación completa del objeto la responsabilidad de crearlos se delega al Service correspondiente ya que "A agrega a B y A contiene a B,".

CONTROLLER

Para manejar los eventos del sistema tenemos la capa WebApi que es la encargada de recibir las solicitudes y hacer los llamados correspondientes a la capa de BusinessLogic.

POLYMORPHISM

En el único caso que tenemos una variación de comportamiento basado en tipos es con los dispositivos que uno de los tipos "Security Camera" tiene comportamientos distintos. En estos casos optamos hacer o switches o ifs que evalúen el Type del Device.

INDIRECTION

Para reducir el acoplamiento directo entre componentes utilizamos inyección de dependencias. Esto trata de crear una interfaz por la cual el componente A que utiliza el componente B tiene la firma que el componente B tiene que implementar. En base a esto si hay un cambio en el componente B el componente A no tiene por que enterarse porque ya sabe que es los parámetros que debe mandar y que es lo que va a recibir. Gracias a esto tenemos un sistema más flexible, escalable y menos acoplado.

HIGH COHESION

Quisimos crear solo Lógicas que sean necesarias para que las que si creamos estén enfocadas y relacionadas para tener alta cohesión y tener el código pertinente encapsulado.

LOW COUPLING

Intentamos minimizar la dependencia entre clases. En la BusinessLogic tenemos pocas dependencias entre lógicas. Como resultado tenemos un sistema más mantenible.

PURE FABRICATION

Aplicamos Pure Fabrication dos veces. La primera son los DTOs que creamos tanto en la WebApi para recibir y enviar datos y en la businessLogic para pasar datos entre capas. La segunda es en nuestro dominio que creamos clases como HomeMemberNotification y RoleSystemPermission cuyo propósito es no repetir información y representar relaciones entre clases.

PROTECTED VARIATIONS

Este patrón lo aplicamos en la SeedData y en las Constants ya que queríamos asegurarnos de que cada vez que se creara la base de datos se cree con los mismos datos.

Jerarquía de herencias

En nuestro proyecto utilizamos herencia en lugares: primero, la clase SecurityCamera hereda de Device, añadiendo atributos como Outdoor, Indoor, MovementDetection y PersonDetection, que son exclusivos de este tipo de dispositivo. No creamos clases específicas para otros dispositivos porque, al no tener atributos propios, resultarían vacías y no tenía sentido la herencia. En el caso que en el futuro se busque agregarle atributos propios a esas clases ahí si se crearía la herencia.

Segundo, en el paquete BusinessLogic, utilizamos herencia en CustomExceptions, donde las excepciones personalizadas, nombradas según la clase que las lanza, heredan de Exception, lo que facilita un manejo de errores más claro.

Por último, aplicamos un enfoque similar en las excepciones personalizadas de DataAccess, asegurando consistencia y eficiencia en la gestión de errores a lo largo del proyecto.

Mecanismo para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio

Para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio se utilizó la técnica de reflection para poder agregar dlls con importadores que permitan importar dispositivos y también dlls con validadores para poder agregar validadores.

Para el caso de los validadores, implementamos la clase LoadAssemblyClass que tiene el metodo GetImplementations que carga las dll que se van agregando y así se van guardando los nuevos tipos de validadores.

Para importar dispositivos, implementamos una interfaz, ImportDevicesFromFile que deben implementar los importadores que quieran importar dispositivos. En la parte de nuevos requerimientos está explicado en profundidad cómo se aplica.

Instructivo para importar dispositivos

1. Agregar el archivo de los dispositivos a importar en la carpeta "DevicesFiles" dentro de la carpeta "DeviceImporter"
 2. Seleccionar el botón "Import multiple devices" en la ventana de register device
 3. Ingresar el file name del json a importar
 4. Ingresar el tipo de importer (en este caso es solo json)
- En la pagina tambien estan detallados los pasos

Device Importer

Instructions:

1. Paste the importer file in the
./SmartHome.BusinessLogic/DeviceImporter/DeviceFiles
directory
3. Input file name (with the extension)
4. Select an importer
5. Click on the "Import Devices" button

File name

Importer

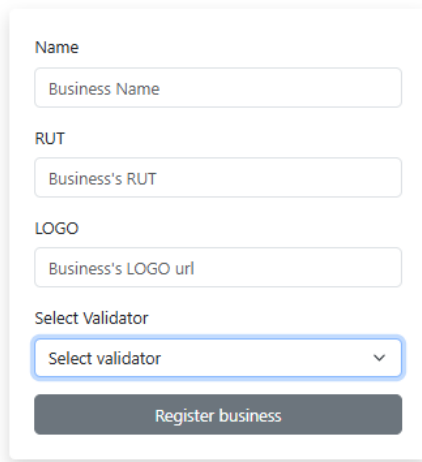
Select an Importer ▼

Import Devices

Instructivo para seleccionar un nuevo modelo validador

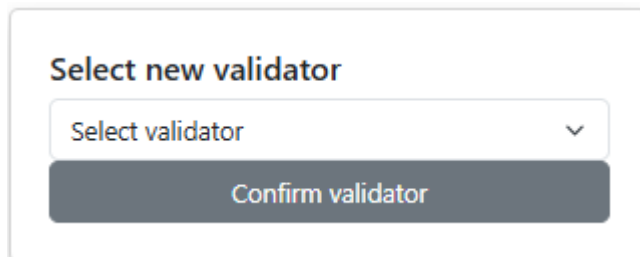
1. Agregar en la carpeta ModelValidators la dll que contenga el validador. Este tiene que implementar la interfaz IModeloValidador
2. En la ventana de business owner, si aún no se ha creado la empresa va a aparecer la opción select validator y ahí van a aparecer los validadores.

Register Business



A form titled "Register Business" with the following fields: "Name" with a sub-label "Business Name", "RUT" with a sub-label "Business's RUT", "LOGO" with a sub-label "Business's LOGO url", and a "Select Validator" dropdown menu with the text "Select validator" and a downward arrow. Below these fields is a dark grey button labeled "Register business".

3. En el caso que ya esté creada se selecciona el botón "select validator" en la esquina superior izquierda y se puede elegir la opción.



A modal titled "Select new validator" containing a dropdown menu with the text "Select validator" and a downward arrow, and a dark grey button labeled "Confirm validator" below it.

Resumen de las mejoras de diseño

La implementación de las nuevas funcionalidades que se pidieron para esta entrega no requirió hacer muchos cambios de diseño en el modelo que ya teníamos.

Los principales cambios fueron dos. El primero que hicimos fue el de agregar la técnica de reflection a nuestro proyecto. Esto permitió agregar extensibilidad a la aplicación ya que el usuario puede agregar nuevos tipos de validadores y nuevos dispositivos.

El segundo cambio que hicimos fue el de las imágenes de los dispositivos como mencionamos anteriormente ya que ahora se permite que los dispositivos puedan tener varias imágenes asociadas.

Manejo de excepciones

Implementamos un `ExceptionHandler` para manejar errores de forma centralizada. Este filtro se encarga de interceptar cualquier excepción no manejada que se propague durante la ejecución de las solicitudes. Al activarse, modifica el contexto de la solicitud, identifica el

tipo de excepción lanzada y devuelve el código de respuesta HTTP correspondiente junto con el mensaje asociado a la excepción.

Dado que el `ExceptionHandler` es un mecanismo transversal a todos los controladores, decidimos instanciarlo directamente en el archivo `Program`. Esto nos permite evitar la repetición de código en cada controlador y centralizar su configuración.

Especificación de la API

Para esta segunda entrega un cambio que hicimos en todos los endpoints fue cambiar el v1 por v2 ya que esta vendría a ser la segunda versión de la aplicación. Los [nuevos endpoints](#) que agregamos se encuentran en el anexo.

En algunos endpoints que recibían ids de tipo `Guid` o un `string` nos daba problemas al hacer el traspaso desde angular al backend porque nos daba un error que el body no era del tipo esperado y la solución que encontramos fue hacer un request body que tenga como atributo lo que estábamos pasando.

Un ejemplo del antes y después de crear el request model para darle este uso es el del endpoint `AddDeviceToHome`:

Antes:

```
[HttpPost("{homeId}/homeDevices")]
```

```
public IActionResult AddDeviceToHome([FromRoute] Guid homeId, [FromBody] Guid deviceId)
```

```
{
```

```
    _homeLogic.AddDeviceToHome(homeId, deviceId);
```

```
    return NoContent();
```

```
}
```

Después

```
[AuthorizationFilter(SeedDataConstants.HOME_RELATED_PERMISSION_ID)]
[HomeAuthorizationFilter(SeedDataConstants.ADD_DEVICES_TO_HOME_PERMISSION_ID)]
[HttpPost("{homeId}/homeDevices")]
1 reference | 1/1 passing | Pedro Azambuja, 3 days ago | 1 author, 1 change
public IActionResult AddDeviceToHome([FromRoute] Guid homeId, [FromBody] AddDeviceToHomeRequestModel request)
{
    _homeLogic.AddDeviceToHome(homeId, request.DeviceId);
    return NoContent();
}
```

New Request Model

```
namespace SmartHome.WebApi.WebModels.HomeModels.In;
2 references | Pedro Azambuja, 3 days ago | 1 author, 1 change
public class AddDeviceToHomeRequestModel
{
    2 references | 1/1 passing | Pedro Azambuja, 3 days ago | 1 author, 1 change
    public Guid DeviceId { get; set; }
}
```


- AddUserToHomeRequestModel
- ValidatorIdRequestModel
- DeviceImportRequestModel
- AddDeviceToHomeRequestModel
- IntelligentLampRequestModel
- MovementSensorRequestModel
- RoomRequestModel
- HomeDeviceIdRequestModel
- UpdateNameRequestModel
- WindowSensorRequestModel

[illegible]

Análisis de la cobertura de los tests

Para esta segunda parte del proyecto tuvimos que implementar alrededor de 200 tests, siendo el evento más importante el pasar de cubrir 0% a casi 89% de los Repositories en DataAccess. Donde podríamos haber cubierto el 100% de no ser por el tipo abstracto de SQLException implementado por nosotros en los repositorios lo cual se dificulta cubrir.

smarthome.dataaccess.dll	31.81%	2085	4470
{ } SmartHome.DataAccess.Repositories	88.87%	1453	182
{ } SmartHome.DataAccess.Migrations	0.00%	0	4286
{ } SmartHome.DataAccess.CustomExceptions	100.00%	2	0
{ } SmartHome.DataAccess.Contexts	99.68%	630	2

Por otra parte, en la WebApi, los controllers quedaron 100% cubiertos, mientras que los filters están cubiertos en un 88,4%:

{ } SmartHome.WebApi.Filters	89.40%	312	37
{ } SmartHome.WebApi.Controllers	100.00%	483	0

Para el JSONImporter creamos y precisamos únicamente de un test para cubrirlo en su totalidad:

smarthome.jsonimporter.dll	100.00%	74	0
{ } SmartHome.JSONImporter	100.00%	54	0
{ } SmartHome.JSONImporter.ImportingData	100.00%	20	0

Por último, BusinessLogic queda cubierto en un 83%:

smarthome.businesslogic.dll	83.10%	1741	354
{ } SmartHome.BusinessLogic.Services	82.72%	1417	296
{ } SmartHome.BusinessLogic.LoadAssembly	86.42%	70	11
{ } SmartHome.BusinessLogic.DTOs	100.00%	12	0
{ } SmartHome.BusinessLogic.Domain	82.40%	220	47
{ } SmartHome.BusinessLogic.CustomExceptions	100.00%	22	0

pero yendo específicamente a los services, el promedio aumenta considerablemente a un 94% de los services:

smarthome.businesslogic.dll	83.10%	1741	354
{ } SmartHome.BusinessLogic.Services	82.72%	1417	296
BusinessService	98.90%	180	2
DeviceImportService	85.00%	102	18
DeviceService	96.30%	130	5
HomeService	88.62%	475	61
RoleService	100.00%	66	0
SessionService	94.23%	49	3
UserService	98.49%	196	3
ValidatorService	90.91%	60	6

La cobertura ascendió considerablemente, con respecto al 74% de la primera entrega del proyecto, nos dedicamos a cubrir todo lo nuevo implementado y corregir lo no cubierto anteriormente, haciendo énfasis en las correcciones sobre clean code e implementando TDD. Los tests que directamente pasaron a fase GREEN son aquellos que

se implementaron para cubrir código ya implementado. Por último, los que comenzaron en fase RED son aquellos tests con los cuales pudimos implementar las nuevas funcionalidades.

También hicimos correcciones con respecto a clean code: hicimos énfasis en organizar el código, quitamos las combinaciones `Assert.IsTrue(...Equals())` e implementamos los nuevos tests utilizando `FluentAssertions (Should().Be()...`, `Should().HaveCount()...`, etc) quitamos los comentarios innecesarios, los `using` que no eran utilizados, y codificamos en Vertical Programming.

Anexo

Diagrama de clases Domain

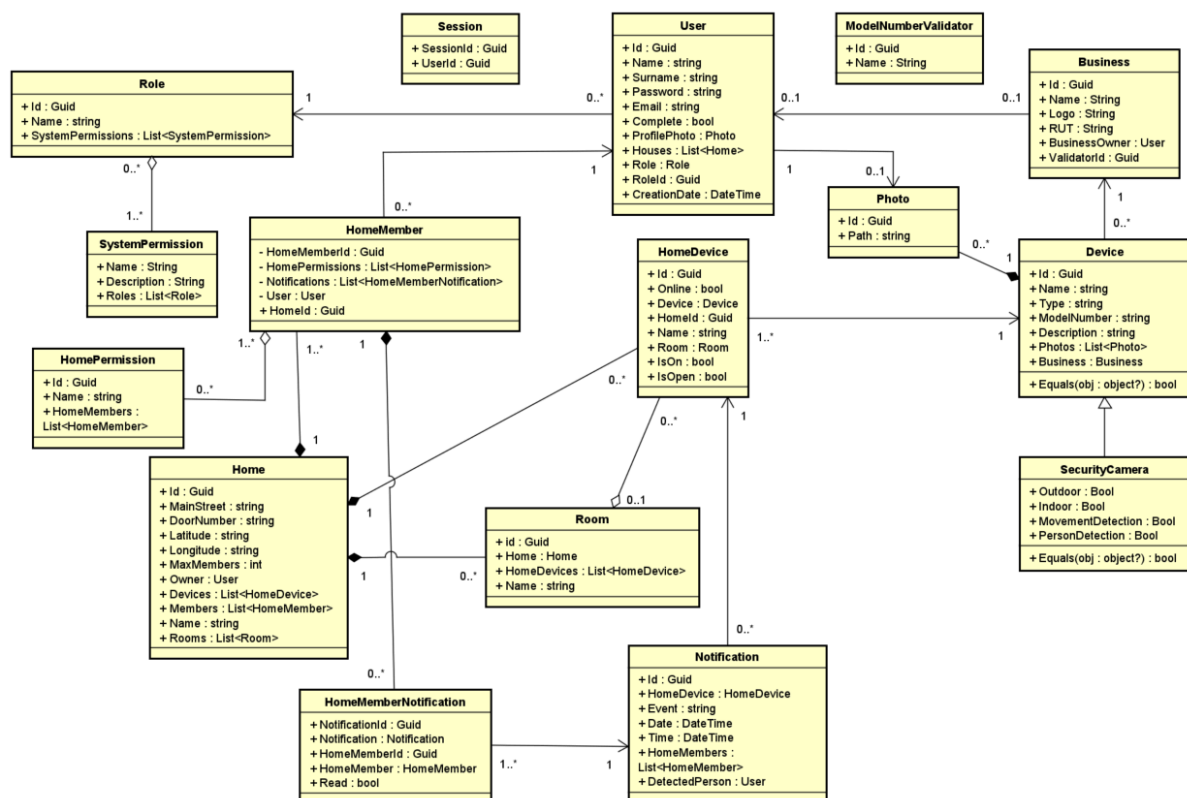


Diagrama de secuencia creación de una notificación de detección de persona

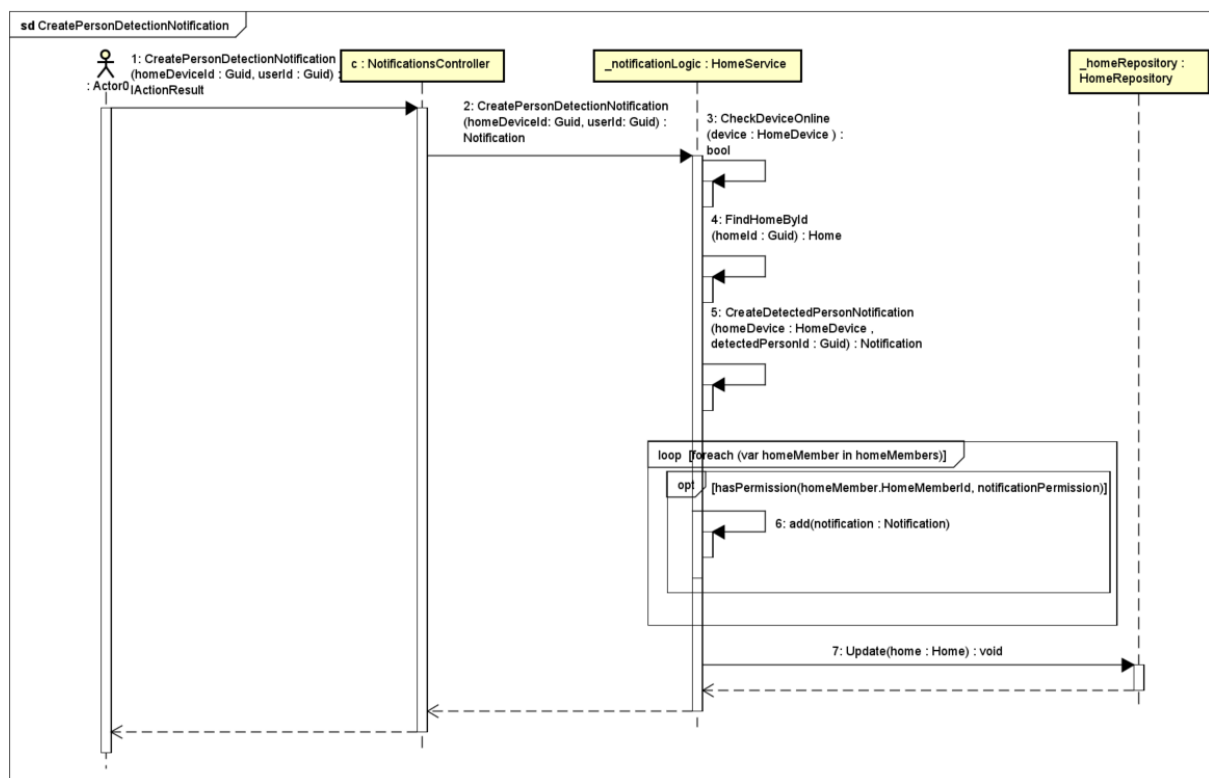


Diagrama de secuencia borrado de un administrador

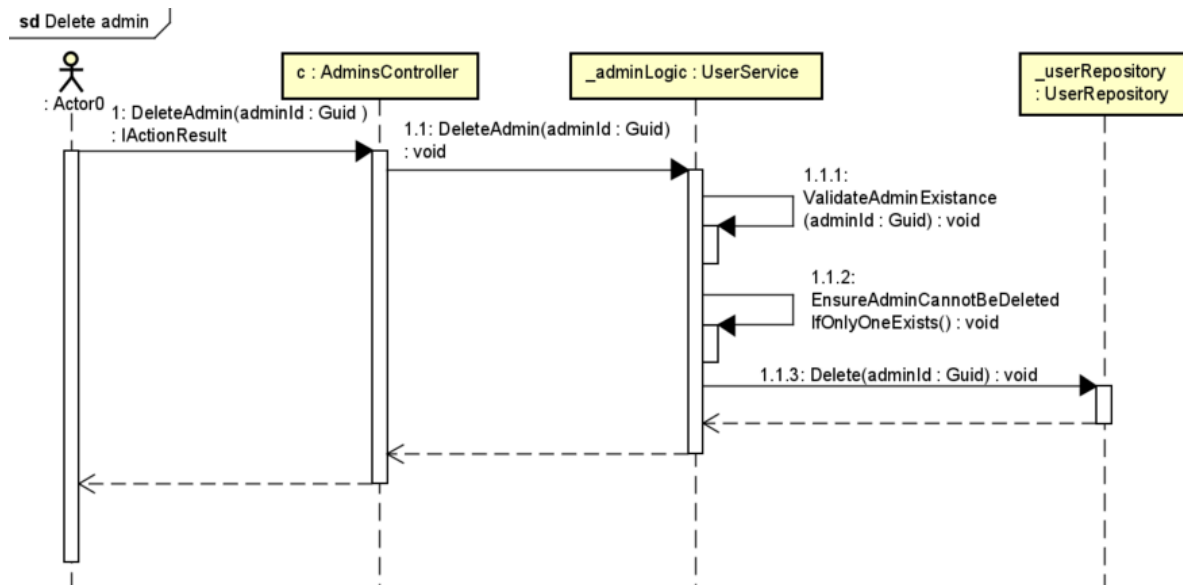
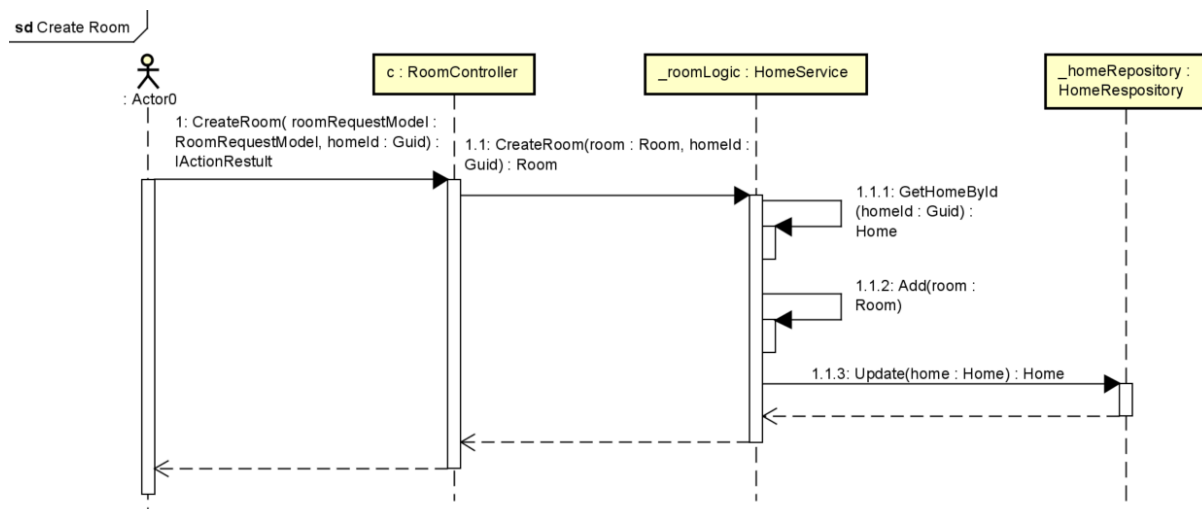


Diagrama de secuencia crear un cuarto



Nuevos endpoints

UpdateAdminRole

Verbo HTTP: PATCH

Recurso: HomeOwnerPermission

Headers: Authorization

Ubicación: /api/v2/admins/homeOwnerPermissions

Body: {"addMemberPermission":"bool",
"addDevicesPermission":"bool","listDevicesPermission":"Bool","notificationsPermission":"bool"
}

Respuestas: 204/401/500

Descripción: Actualiza los permisos de un usuario a los de admin

GetBusinesses

Verbo HTTP: GET

Recurso: Business

Headers: Authorization

Ubicación: /api/v2/businesses

Body: -

Respuestas: 200/204/401/500

Descripción: Permite listar los businesses, pudiendo filtrarlos y paginarlos también

GetAllValidators

Verbo HTTP: GET

Recurso: Validators

Headers: Authorization

Ubicación: /api/v2/businesses/validators

Body: -

Respuestas: 200/401/500

Descripción: Permite listar los validators asociados a los businesses

AddValidatorToBusiness

Verbo HTTP: PATCH

Recurso: Validators

Headers: Authorization

Ubicación: /api/v2/businesses/validators

Body: { "validatorId": "string" }

Respuestas: 200/400/401/500

Descripción: Permite agregar un validator a un business

UpdateBusinessOwnerRole

Verbo HTTP: PATCH

Recurso: HomeOwnerPermissions

Headers: Authorization

Ubicación: /api/v2/businessOwners/homeOwnerPermissions

Body: -

Respuestas: 200/401/500

Descripción: Actualiza los permisos del businessOwner

ImportDevice

Verbo HTTP: POST

Recurso: DeviceImport

Headers: Authorization

Ubicación: /api/v2/deviceImport

Body: { "DIName" : "string", "FileName" : "string" }

Respuestas: 201/401/500

Descripción: Importa un dispositivo a partir del archivo especificado por el usuario autenticado

UpdateHomeMemberPermissions

Verbo HTTP: PUT

Recurso: HomeMemberPermission

Headers: Authorization

Ubicación: /api/v2/homeMembers/{homeMemberId}/permissions

Body: { "permissions" : [HomePermission] }

Respuestas: 204/401/403/500

Descripción: Actualiza la lista de permisos del home member especificado

GetAllHomePermissions

Verbo HTTP: GET

Recurso: HomePermissions

Headers: Authorization

Ubicación: /api/v2/homeMembers/homePermissions

Body: -

Respuestas: 200/401/500

Descripción: Obtiene una lista con todos los permisos disponibles para los miembros del hogar

UnRelatedHomeOwners

Verbo HTTP: GET

Recurso: HomeOwner

Headers: Authorization

Ubicación: /api/v2/homes/{homeId}/unRelatedHomeOwners

Body: -

Respuestas: 200/401/403/500

Descripción: Devuelve una lista de los Home Owners no relacionados al Home

UpdateHomeDeviceName

Verbo HTTP: PATCH

Recurso: HomeDevice

Headers: Authorization

Ubicación: /api/v2/homes/{homeDeviceId}/homeDeviceName

Body: { "name" : "string" }

Respuestas: 200/401/403/500

Descripción: Actualiza el nombre de un HomeDevice

UpdateHomeName

Verbo HTTP: PATCH

Recurso: Home

Headers: Authorization

Ubicación: /api/v2/homes/{homeld}/homeName

Body: { "name" : "string" }

Respuestas: 200/401/403/500

Descripción: Actualiza el nombre del hogar

CreateIntelligentLamp

Verbo HTTP: POST

Recurso: IntelligentLamps

Headers: Authorization

Ubicación: /api/v2/intelligentLamps

Body: { "name" : "string", "brightness" : "int", "color" : "string", "location" : "string" }

Respuestas: 201/401/403/500

Descripción: Crea una nueva lámpara inteligente asociada al usuario autenticado

CreateMovementSensor

Verbo HTTP: POST

Recurso: MovementSensors

Headers: Authorization

Ubicación: /api/v2/movementSensors

Body: { "name" : "string", "sensitivity" : "int", "range" : "int", "location" : "string" }

Respuestas: 201/401/403/500

Descripción: Crea un nuevo sensor de movimiento asociado al usuario autenticado

CreateRoom

Verbo HTTP: POST

Recurso: Room

Headers: Authorization

Ubicación: /api/v2/rooms/{homeld}

Body: { "name" : "string", "description" : "string" }

Respuestas: 201/403/500

Descripción: Crea una nueva habitación en el hogar

AddDevicesToRoom

Verbo HTTP: POST

Recurso: HomeDevices
Headers: Authorization
Ubicación: /api/v2/rooms/{roomId}/homeDevices
Body: { "id" : "guid" }
Respuestas: 201/403/500
Descripción: Agrega un dispositivo a la habitación

GetAllRoomsFromHome

Verbo HTTP: GET
Recurso: Rooms
Headers: Authorization
Ubicación: /api/v2/rooms/{homeId}
Body: -
Respuestas: 200/401/403/500
Descripción: Devuelve una lista de habitaciones asociadas a un hogar

GetHomeById

Verbo HTTP: GET
Recurso: Homes
Headers: Authorization
Ubicación: /api/v2/homes/{homeId}
Body: -
Respuestas: 200/401/403/500
Descripción: Devuelve el hogar asociado al homeId

GetBusinessByUser

Verbo HTTP: GET
Recurso: Business
Headers: Authorization
Ubicación: /api/v2/businesses/myBusinesses
Body: -
Respuestas: 200/401/403/500
Descripción: Devuelve un negocio asociado a un usuario que es su businessOwner

Endpoints modificados

AddDeviceToHome

Verbo HTTP: POST
Recurso: HomeDevice
Headers: Authorization
Ubicación: /api/v2/homes/{homeId}/homeDevices
Body: {id: Guid}
Respuestas: 201/401/400/500
Descripción: Agregar un nuevo dispositivo al hogar

AddHomeMemberToHome

Verbo HTTP: POST

Recurso: HomeMember

Headers: Authorization

Ubicación: /api/v2/homes/{homeId}/members

Body: {"id": Guid}}

Respuestas: 201/401/500

Descripción: Agregar un miembro al hogar