UNIVERSIDAD ORT URUGUAY

Facultad de Ingeniería

Obligatorio 1 Diseño de Aplicaciones 2

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Martín Slepian - 266959

Leopoldo Perez - 257341

Eric Poplawski - 258327

Profesores: Nicolás Fierro, Alexander Wieler, Sofia Duclos

2024

Link al repositorio: https://github.com/IngSoft-DA2/257341-266959-258327

Declaración de autoría

Nosotros, Martín Slepian, Leopoldo Perez y Eric Poplawski declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

1. La obra fue producida en su totalidad mientras cursamos Diseño de Aplicaciones 2.

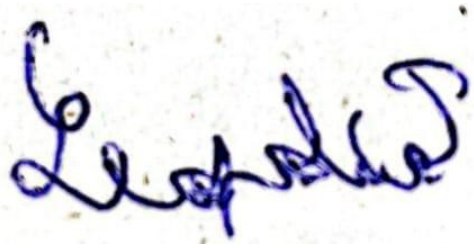
2. Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad.

3. Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra.

4. En la obra, hemos acusado recibo de las ayudas recibidas.

5. Cuando la obra se basa en el trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y que fue construido por nosotros.

6. Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Martín Slepian	Leopoldo Perez	Eric Poplawski
7/10/2024	7/10/2024	7/10/2024

Índice

Abstract

El proyecto "SmartHome" es una solución para la gestión de dispositivos inteligentes en el hogar, diseñada para ofrecer una experiencia centralizada y segura. La plataforma permite a usuarios como administradores, dueños de empresas y dueños de hogares gestionar dispositivos de manera eficiente mediante una API REST. Su objetivo es unificar la administración de dispositivos de distintos fabricantes en un entorno escalable, resolviendo el problema de la fragmentación en la gestión de múltiples aplicaciones.

La arquitectura de capas del sistema facilita la separación de responsabilidades: la capa de presentación (SmartHome.WebApi) gestiona las solicitudes HTTP, la lógica de negocio procesa estas solicitudes y la capa de acceso a datos maneja la persistencia mediante EntityFramework Core. Esta estructura asegura una clara separación y mejora la mantenibilidad del sistema.

La plataforma prioriza la seguridad con filtros personalizados de autenticación y autorización, validando cada solicitud para asegurar que solo usuarios autorizados realicen acciones críticas. Los roles de usuario permiten gestionar permisos específicos para un control de acceso preciso. La autenticación basada en tokens garantiza la seguridad de las transacciones.

El sistema maneja excepciones de manera robusta, proporcionando respuestas HTTP coherentes y mensajes claros. Las pruebas de la API, realizadas con Postman, aseguran una interacción fluida y confiable entre el cliente y el backend.

El desarrollo se basa en principios de diseño SOLID y patrones como repositorio y fachada, garantizando un código modular y extensible. Se han identificado mejoras futuras, como la optimización de la estructura de servicios y la organización de los DTOs para una mejor integración con la capa de presentación.

Descripción general del trabajo

El sistema desarrollado, denominado "SmartHome," es una plataforma integral para la gestión de dispositivos inteligentes en el hogar. La API centraliza la administración de usuarios y dispositivos, ofreciendo distintos niveles de acceso y funcionalidades personalizadas según el rol del usuario, para garantizar una gestión eficiente y segura del entorno doméstico.

¿Qué hace la solución? Características claves, objetivos y alcances.

La solución tiene como objetivo principal unificar la gestión de dispositivos inteligentes de distintos fabricantes en un solo entorno, proporcionando a los usuarios una experiencia fluida y eficiente. Busca resolver el problema de manejar múltiples aplicaciones separadas al consolidarlas en un sistema unificado y escalable. La plataforma permite gestionar usuarios y dispositivos inteligentes de manera integral. Los dueños de empresas pueden registrar y gestionar dispositivos, mientras que los dueños de hogares pueden crear hogares virtuales, asociar dispositivos y compartir la administración de estos con otros miembros del hogar.

La plataforma facilita el registro de dispositivos inteligentes y su gestión avanzada, como detección de movimiento y cambios de estado en sensores, con notificaciones automáticas a los miembros configurados. Los usuarios reciben notificaciones sobre acciones claves de los dispositivos como detección de movimiento o apertura de ventanas, y pueden configurar quién recibe dichas notificaciones. El control de accesos está basado en roles, garantizando que solo usuarios autorizados puedan realizar operaciones críticas, con la posibilidad de agregar miembros a un hogar con permisos específicos.

El alcance de la plataforma incluye la implementación completa del backend mediante una API REST, sin incluir una interfaz de usuario, pero diseñada para una integración sencilla con cualquier cliente HTTP. Para la validación de sus funcionalidades se ha utilizado Postman, asegurando una interacción fluida con la API. La seguridad es un aspecto clave, implementando autenticación basada en tokens y un sistema de permisos granular para asegurar que solo usuarios autorizados puedan acceder y operar los dispositivos y funcionalidades del sistema

que así lo requieran. Filtros de seguridad personalizados validan cada solicitud HTTP, garantizando que las acciones críticas sólo puedan ser ejecutadas por quienes cuentan con los permisos adecuados.

Mejoras a futuro y errores conocidos

1. HomeService tiene atributos a UserRepository y DeviceRepository, para traer a las entidades de las clases del dominio que precisa manipular en sus métodos. Lo mismo ocurre con el IRepository<Resident> en SystemService. Para la próxima entrega se corregirá este error, HomeService tendrá de atributos a las interfaces de ISystemService y IDeviceService, y SystemService tendrá a IHomeService, que ya que poseen los métodos que controlan el acceso a los repositorios de User y Device. De esta forma, mantendremos los mismos mecanismos para todas las partes de la solución.

2. Actualmente los Services e IServices en el paquete de SmartHome.BussinessLogic junto a las clases de UnitTests se encuentran desorganizadas en las carpetas de su paquete. Decidimos no llevar a cabo la organización por posibles conflictos de los namespaces, para la próxima entrega se corregirá.

3. Los DTOs Requests y Responses de los Controladores deberían estar en el paquete de SmartHome.WebApi ya que son elementos pertenecientes a la capa de presentación e utilizados para la comunicación con el cliente.

5. En la mitad del trabajo, se realizó el cambio de nombre de la clase Membership a Resident, en algunos lados no se realizó la corrección. Un ejemplo es en el ApplicationDbContext quedo mal enunciado; dice Memberships pero tendría que ser Residents.

6. Para la próxima entrega, se implementará una validación en el filtro de excepciones que verificará si el sistema está en un entorno de desarrollo o demostración, para determinar qué información de los objetos de respuesta se debe mostrar. En esta entrega, hemos identificado ciertos detalles de los objetos de respuesta que consideramos inapropiados para ser mostrados a cualquier cliente.

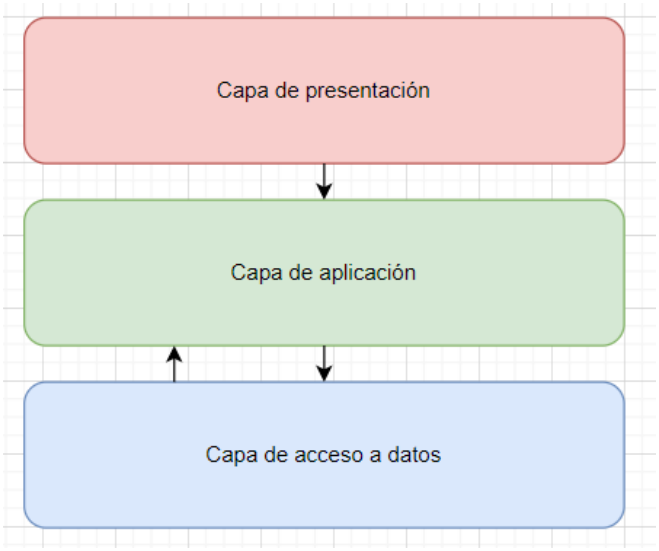
7. Nos faltó tener un poco más de cobertura de líneas de código en HomeServiceTests dado que encontramos un error de último momento y no nos dió el tiempo de modificar los tests.

Diagrama general de paquetes

La capa de presentación se representa en el paquete de SmartHome.WebApi, el cual aloja a los controladores y filtros. Los controladores procesan y validan las solicitudes del cliente, traduciéndose en llamadas a los servicios de negocio y devolviendo respuestas con datos o mensajes de error. Los filtros de autenticación aseguran que las solicitudes provengan de usuarios autenticados, mientras que los filtros de autorización verifican que los usuarios tengan los permisos adecuados para acceder a recursos sensibles, garantizando la seguridad del sistema y evitando accesos no autorizados.

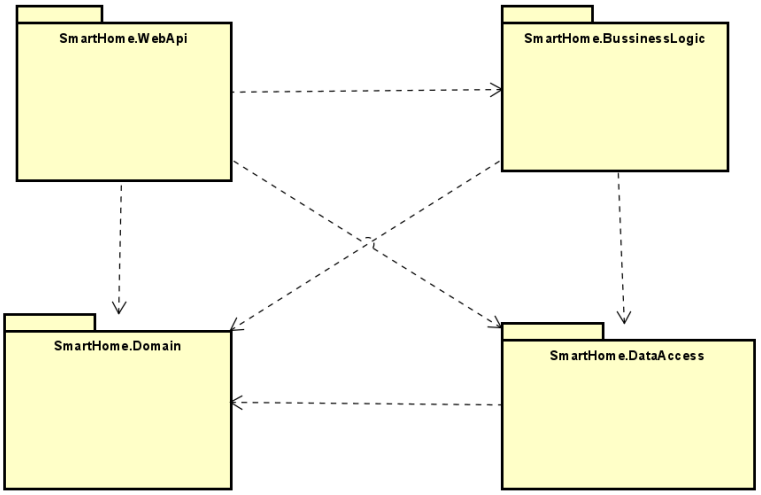
A continuación se encuentran la capa de aplicación, que aloja los servicios, las clases del dominio y los argumentos, por tanto ocupa los paquetes de SmartHome.Domain y SmartHome.BussinessLogic. Los servicios encapsulan la lógica de negocio principal del sistema y coordinan las interacciones entre las entidades del dominio y la capa de acceso a datos. Las clases del dominio, por su parte, representan las entidades principales del sistema y definen las reglas de negocio que rigen su comportamiento, garantizando la coherencia e integridad de los datos a lo largo del sistema. También hemos implementado los Argumentos que se encuentran en los models del paquete de SmartHome.BussinessLogic, que son DTOs de poca complejidad responsables de recibir la información del controlador y validar que no contengan parámetros nulos o vacíos antes de enviarla al servicio correspondiente. Estos elementos se alojan en la capa de aplicación, pero no son ejecutadas por esta capa.

Por último, la capa más baja es la de DataAccess, que contiene la clase Repository y su respectiva interfaz IRepository, además de cualquier otra clase de repositorio que necesite sobrescribir métodos específicos de Repository. Esta capa se encarga de gestionar el acceso a los datos y garantizar la separación entre las capas superiores y la base de datos.



Esta separación de responsabilidades no solo facilita la comprensión y el mantenimiento del código, sino que también mejora la escalabilidad del sistema al permitir que cada componente evolucione de manera independiente. Se respeta el Principio de Inversión de Dependencias, ya que los módulos de alto nivel no dependen de los de bajo nivel. Todos los paquetes dependen de Domain, que es un módulo de alto nivel porque define las entidades y la lógica fundamental del negocio.

En cuanto a las dependencias, WebApi depende de los servicios de la lógica de negocio (BussinessLogic) para ejecutar las operaciones solicitadas por el cliente. BussinessLogic depende de las interfaces de repositorio proporcionadas por la capa de acceso a datos (SmartHome.DataAccess) para acceder y persistir datos. DataAccess, por su parte, se encarga de proveer datos a la capa de lógica de negocio (BussinessLogic) y depende de las definiciones de las entidades que se encuentran en el paquete Domain para gestionar la persistencia de dichos datos.

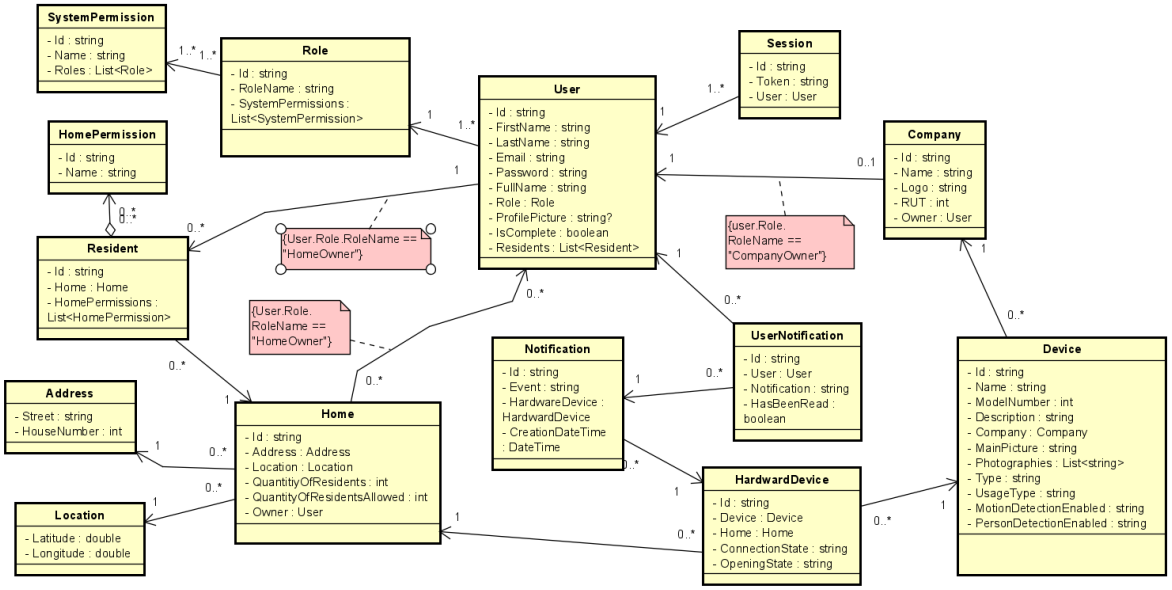


Descripción de los paquetes

SmartHome.Domain

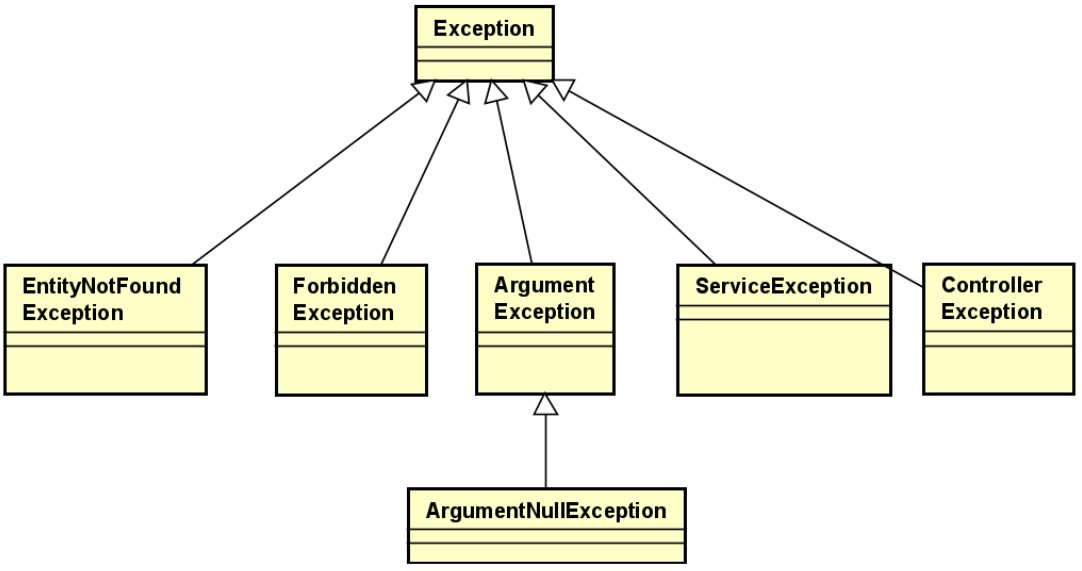
Responsabilidad: Contiene las entidades del sistema que representan los objetos del mundo real (usuarios, dispositivos, empresas, hogares, etc.) y las relaciones entre ellos. Esta capa es esencialmente el núcleo de la aplicación y define las estructuras de datos que se manipulan a través de las capas superiores.

Domain



Exceptions

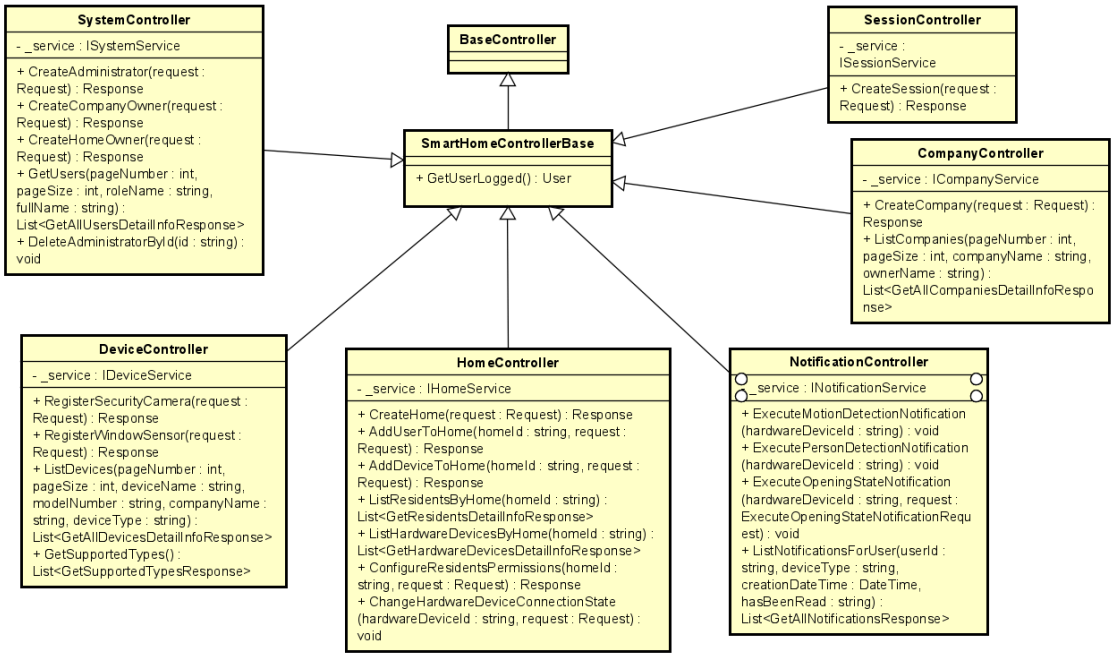
Se gestiona el manejo de errores específicos de la aplicación a través de una jerarquía de excepciones personalizadas. Proporciona excepciones como EntityNotFoundException para recursos no encontrados, ForbiddenException para accesos no autorizados y ControllerException para solicitudes inválidas, entre otras. Este diseño permite un manejo de errores preciso, con respuestas HTTP adecuadas y una experiencia de usuario coherente, facilitando la depuración y mantenimiento del sistema.



SmartHome.WebApi

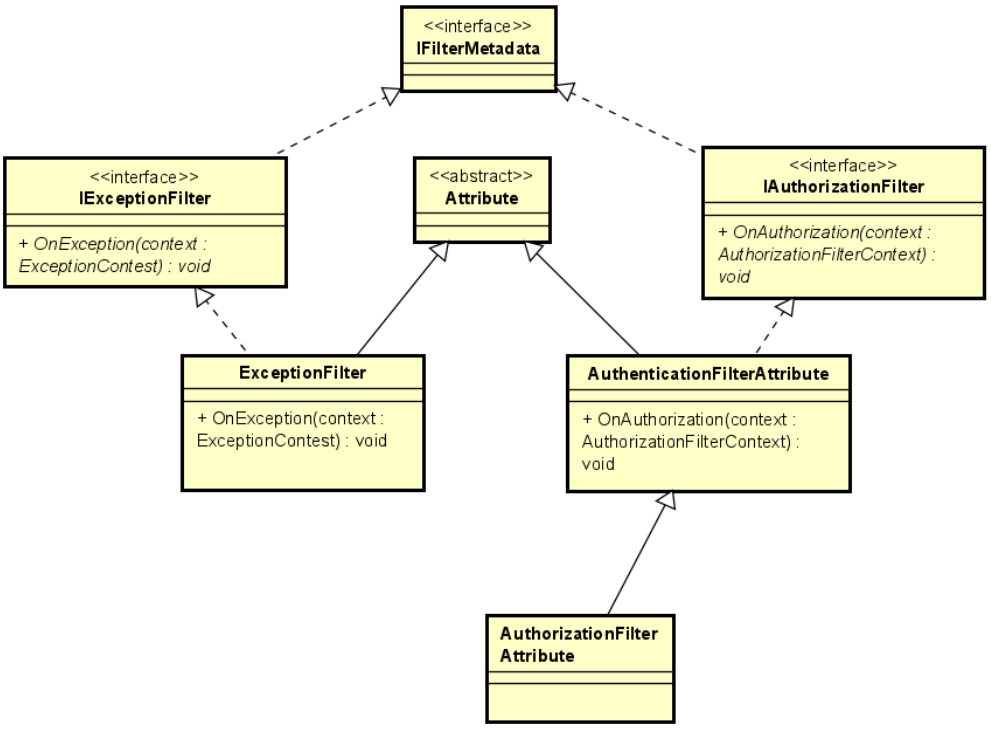
Responsabilidad: Este paquete contiene los controladores de la API REST, que son los puntos de entrada para todas las interacciones externas con el sistema. Define los endpoints que permiten gestionar dispositivos, usuarios, notificaciones y hogares, sirviendo de intermediario entre las solicitudes HTTP entrantes y la lógica de negocio. Además, incluye filtros como AuthenticationFilter y AuthorizationFilter, los cuales se encargan de autenticar las solicitudes y verificar que los usuarios tengan los permisos necesarios para acceder a los recursos, garantizando la seguridad y control de acceso en cada operación.

Controllers



Este diagrama como se puede apreciar existen 6 controladores que tienen el acceso a los Services de los recursos de la API. Estos 6 apuntan/dependen/etc a una clase SmartHomeControllerBase que posee el método GetUserLogged() para obtener el usuario que está realizando la request. Además, hereda de BaseController de ASP.NET Core, proporcionando funcionalidades estándar de controladores, como el manejo de respuestas HTTP y validación.

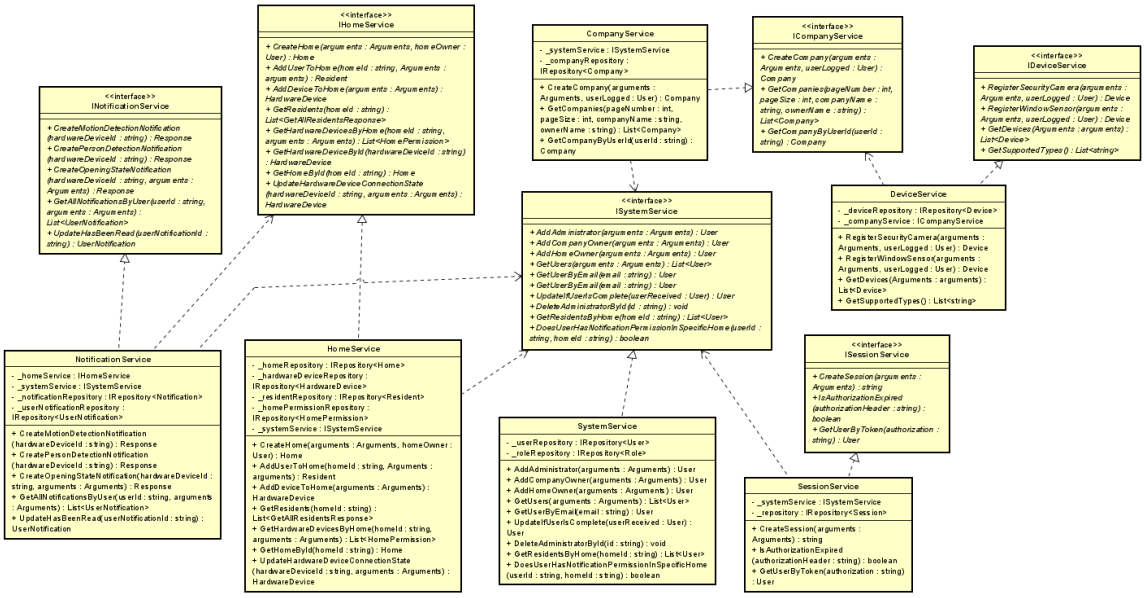
Filters



Implementa AuthenticationFilterAttribute y AuthorizationFilterAttribute, los cuales aseguran que las solicitudes sean autenticadas y que los usuarios cuenten con los permisos necesarios. También implementa un ExceptionFilter para capturar y gestionar excepciones no controladas, proporcionando respuestas HTTP adecuadas en caso de errores, lo que garantiza una mayor robustez y consistencia en la respuesta del sistema ante fallos inesperados.

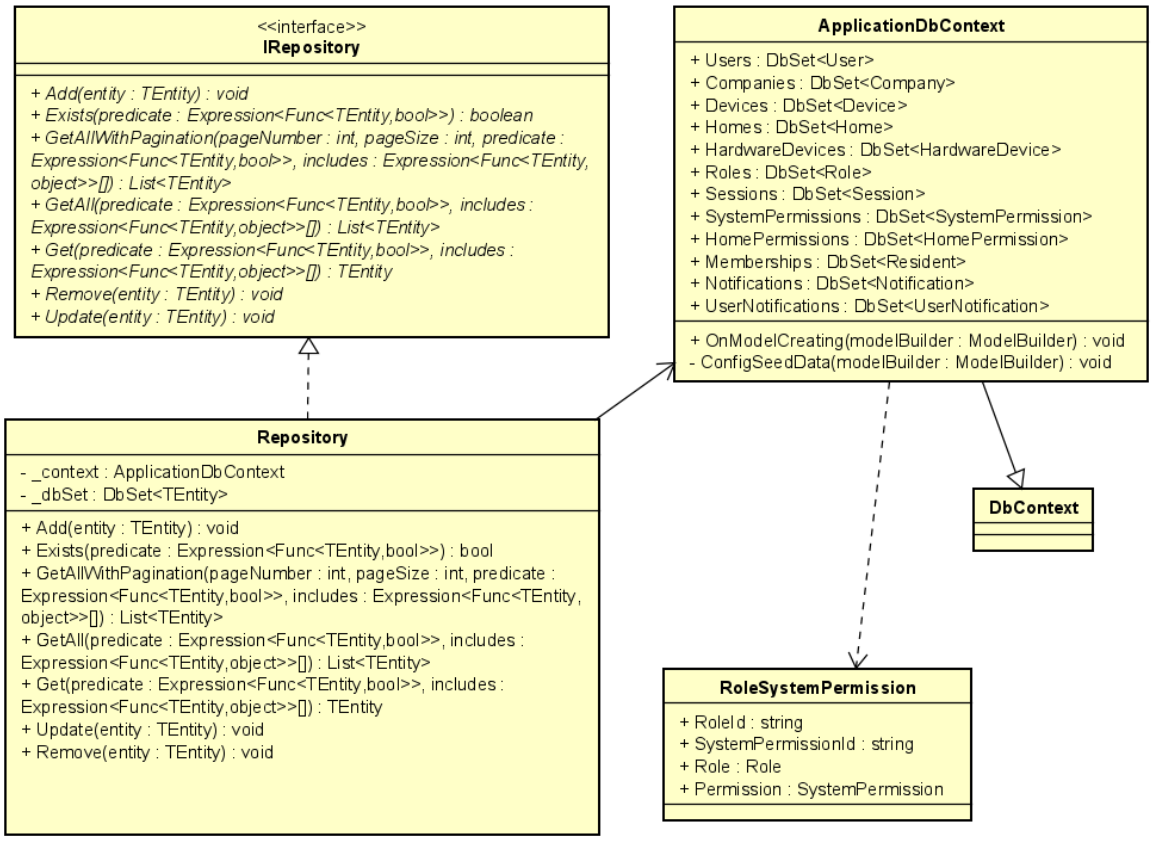
SmartHome.BussinessLogic

Responsabilidad: Los servicios tienen la responsabilidad principal de recibir la información de las solicitudes desde los controladores, procesar la lógica de negocio, comunicarse con la base de datos y devolver una respuesta a los controladores. Cada servicio gestiona el acceso a los repositorios de las entidades sobre las que actúa y, cuando necesita una o varias entidades que no controla, posee como atributo el servicio que gestiona dichas entidades para invocar uno de sus métodos. Esta separación de responsabilidades es fundamental para mantener un diseño modular y escalable, facilitando el mantenimiento y la extensión del sistema sin afectar otras partes del código. Además, la separación mejora la reutilización de la lógica de negocio y permite realizar cambios en la persistencia de datos de manera más aislada.



SmartHome.DataAccess

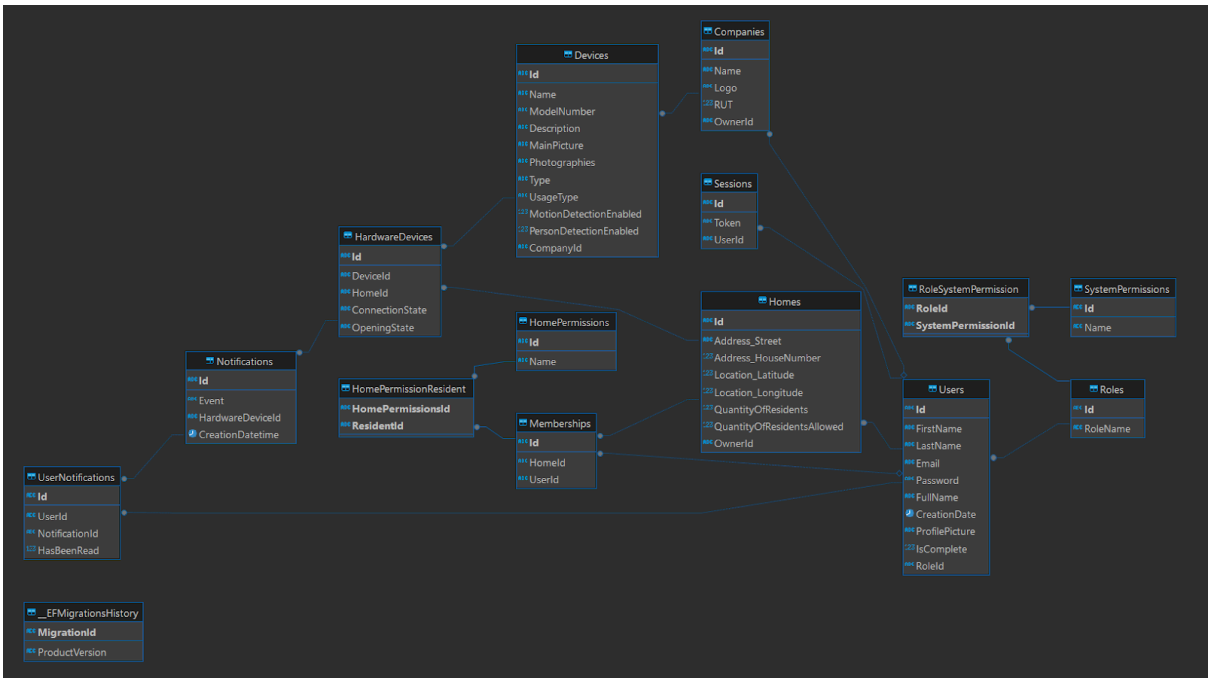
Responsabilidad: Esta capa se encarga de la persistencia y acceso a los datos utilizando Entity Framework Core como ORM. Proporciona los repositorios para acceder a las entidades del sistema y abstraer la interacción directa con la base de datos. Además, maneja la configuración del DbContext que gestiona las transacciones con la base de datos.



Se encuentran los “Repositories” que define los repositorios genéricos para cada entidad, como Repository<User>, Repository<Company>, Repository<Device>, etc.

También se encuentra el “ApplicationDbContext” que configura el contexto de la base de datos y gestiona las conexiones, mapeos y transacciones de las entidades a la base de datos SQL Server.

Modelo de tablas



Uno de los puntos claves es la creación de la clase en RoleSystemPermission hecha específicamente para mapear a la base de datos la relación entre los Roles y los System Permissions, este hecho es vital para la API porque le permite a los usuarios interactuar con las acciones que pueden y que no pueden ejecutar.
Los roles, los permisos y un usuario administrador ya vienen precargados en el SeedData de la clase ApplicationDbContext en el paquete de DataAccess, ya que estos elementos son necesarios para que la aplicación pueda ser puesta en funcionamiento. Otro punto clave es que decidimos mapear los atributos de las clases como Address y Location, en la misma clase que Home.

Visualización de los datos

El flujo de datos en el sistema sigue un ciclo definido por la interacción entre las capas. Un ejemplo típico de cómo los datos fluyen a través del sistema es el siguiente:

1. Solicitud HTTP: Un cliente externo, como Postman o una aplicación móvil, envía una solicitud HTTP a uno de los controladores en la capa smart home.WebApi. Esta solicitud puede ser, por ejemplo, para registrar un nuevo dispositivo en el sistema.

2. Controlador: El controlador correspondiente en la capa smarthome.WebApi procesa la solicitud. Si la solicitud es válida, el controlador extrae los datos del cuerpo de la solicitud y llama al servicio correspondiente en smarthome.BussinessLogic, pasando estos datos como argumentos.

3. Lógica de Negocio: El servicio en la capa smarthome.BussinessLogic recibe los datos y aplica las reglas de negocio correspondientes, como validar si el dispositivo ya existe o si el usuario tiene permisos para registrarlo. Si es necesario, el servicio interactúa con la capa smarthome.DataAccess para almacenar o recuperar datos.

4. Acceso a Datos: La capa smarthome.DataAccess interactúa con la base de datos mediante Entity Framework Core, recuperando o persistiendo datos según sea necesario. Utiliza el contexto de la base de datos (ApplicationDbContext) para ejecutar consultas y transacciones.

5. Respuesta: Una vez que la operación se completa, el servicio en smarthome.BussinessLogic devuelve los resultados al controlador en smarthome.WebApi, que finalmente los empaqueta en una respuesta HTTP y los envía de vuelta al cliente.

Este flujo asegura una separación clara de responsabilidades y facilita el mantenimiento, ya que cada capa se ocupa de una parte específica de la lógica.

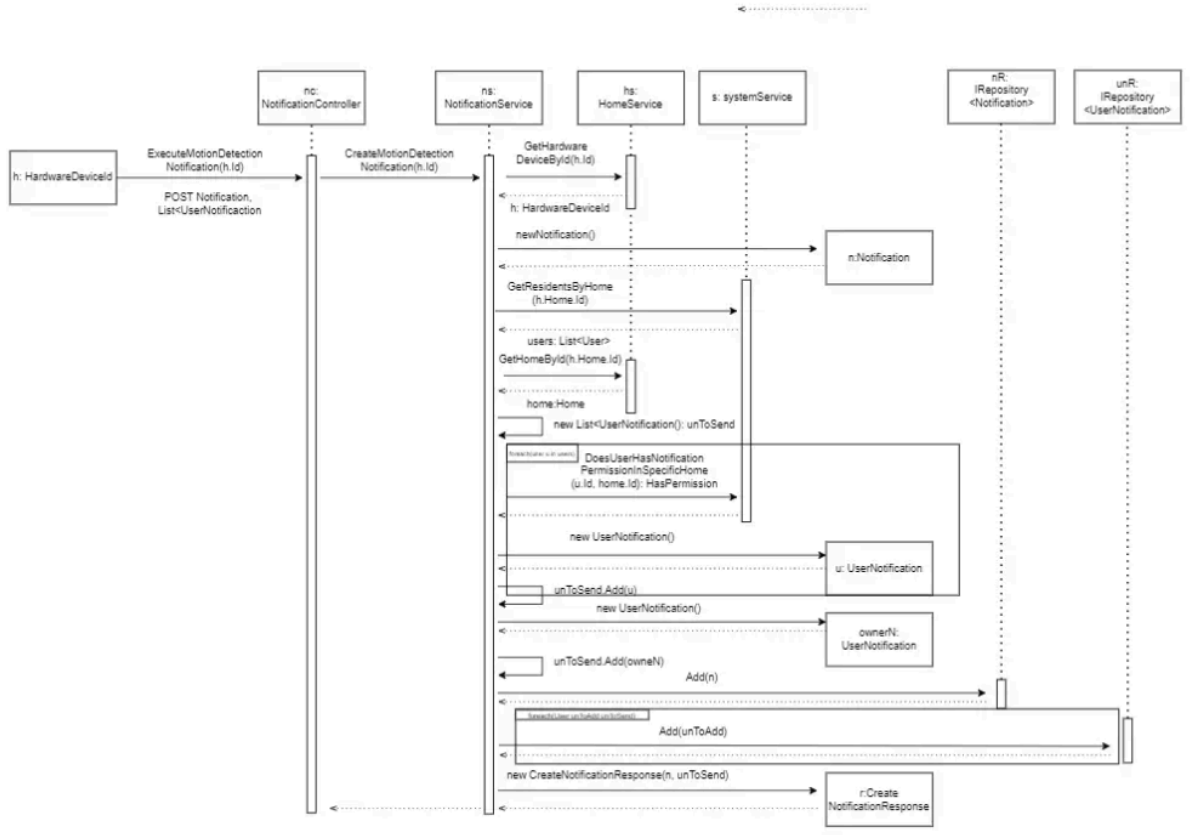
Diagrama de Interacción

Se utilizó la herramienta Draw.IO para la realización del diagrama de interacción.

Endpoint: Acción por detección de movimiento lanza notificación.

Inicia cuando el controlador NotificationController recibe una solicitud POST para generar una notificación de detección de movimiento relacionada con un dispositivo inteligente. El controlador llama al método CreateMotionDetectionNotification del servicio NotificationService, el cual, a su vez, interactúa con HomeService para obtener información del dispositivo correspondiente mediante GetHardwareDeviceById. Una vez que tiene el dispositivo, el servicio crea una nueva instancia de la entidad Notification, que representa el evento de detección de movimiento.

El servicio también interactúa con System Service para llamar a GetResidentsByHome para obtener la lista de usuarios residentes en el hogar asociado al dispositivo, y a GetHomeById para obtener el dueño del hogar. A continuación, verifica si cada usuario tiene los permisos necesarios para recibir la notificación mediante DoesUserHasNotificationPermissionInSpecificHome. Se crean entidades UserNotification tanto para los usuarios con permisos como para el propietario del hogar, y todas estas notificaciones se agregan al repositorio de notificaciones correspondiente. Finalmente, se crea y envía una respuesta CreateNotificationResponse, que contiene la notificación y las notificaciones de usuario.



https://drive.google.com/file/d/1MVz23AT0WqcvcEodVT3r6UYplFd-fcGg/view?usp=sharing

Justificaciones de Diseño

Manejo de excepciones

La API se encarga de devolver un "ObjectResult" con código 200 Ok para indicar que una solicitud se ha procesado correctamente. Sin embargo, para aquellos casos donde las solicitudes no puedan ser procesadas debido a diversas situaciones, se ha organizado un manejo de excepciones para gestionar cómo estas se envían al cliente. La API posee un filtro de excepciones que captura cualquier excepción lanzada internamente y envía la respuesta adecuada al cliente. Dependiendo del tipo de excepción, el filtro utiliza un "Object Result" específico que proporciona información de manera clara y sofisticada, asegurando que el cliente reciba toda la información necesaria sobre el error ocurrido al procesar la solicitud. Como ya se menciona a lo largo del documento, se utilizan filtros de autenticación y autorización para garantizar que solo los usuarios autorizados accedan a ciertos recursos y operaciones. Si se detecta una falta de autenticación o autorización, se lanza un “Object Result” de código de estado 401 o 403 respectivamente.

Se ha decidido utilizar la clase Guard, parte de la extensión "CQ.Utility", que provee varios métodos para validar excepciones, aplicándolos a ciertas áreas de la API. Los métodos de Guard se utilizan principalmente para las excepciones de tipo ArgumentNullException, las cuales se lanzan en los argumentos. De esta manera, se establece una primera línea de defensa en la validación de los datos de entrada, garantizando que se cumplan las reglas básicas de negocio desde el principio, como asegurar que los parámetros no sean nulos ni vacíos cuando llegan al servicio. Además, los métodos de la clase Guard también se utilizan para lanzar excepciones de tipo ArgumentException en casos específicos, como la validación adecuada del ingreso de correos electrónicos. Estas excepciones están relacionadas con la lógica de negocio y, por lo tanto, son lanzadas dentro del servicio.

Para las excepciones genéricas restantes, se ha optado por clasificarlas según el lugar donde son lanzadas, ya sea en el Controlador o en el Servicio. Así, se ha implementado la excepción ControllerException, que se utiliza en casos como cuando la solicitud es nula, y la excepción ServiceException, que se aplica en situaciones como cuando el atributo UsageType recibe un valor que no sea "Interior"

o "Exterior". Para estos cuatro tipos de excepciones, el filtro de excepciones lanza un objeto de resultado que proporciona información sobre la naturaleza de la excepción, el mensaje específico y un código de estado 401 Bad Request, dejando claro que se trató de una solicitud incorrecta.

Además, se ha implementado la excepción ForbiddenException para manejar situaciones en las que un usuario no tiene el permiso del hogar necesario para ejecutar la solicitud. Estas excepciones llegan al filtro, generando un objeto que aclara que se trata de un error con código de estado 403 Forbidden y la información adicional correspondiente. También se manejan excepciones de tipo EntityNotFoundException, que surgen cuando se intenta buscar una entidad por su ID en la base de datos y esta no existe. En tales casos, se proporciona al usuario una aclaración de lo sucedido, lanzando un objeto de resultado con el código 404 Not Found. Por último, el filtro de excepciones actúa como último recurso para proteger la API en caso de que se reciba una excepción de un tipo no registrado. En este caso, se envía un objeto indicando que ha ocurrido un “InternalError”, junto con el código 500 Internal Server Error.

Mecanismos de acceso a datos

La solución utiliza Entity Framework Core como mecanismo de acceso a datos. EF Core es un ORM para .NET que facilita la interacción con bases de datos relacionales al permitir trabajar con datos utilizando clases de C# en lugar de consultas SQL directas.

Descripción del Enfoque de Acceso a Datos:

1. **Uso de Repositorios**: La capa de acceso a datos de la aplicación sigue el patrón de repositorio, lo cual encapsula la lógica de acceso a datos y provee una API simplificada para interactuar con la base de datos. Cada entidad tiene su propio repositorio, como IRepository<TEntity>, lo que permite realizar operaciones comunes como Add, Remove, Update y consultas específicas a través de métodos como Get y GetAll. Este enfoque hace que el código de acceso a datos sea más limpio y facilita su testeo y mantenimiento.

2. **DbContext como Punto Central**: Se utiliza la clase ApplicationDbContext para definir las entidades y su relación con la base de datos. Esta clase hereda de DbContext y es la responsable de mapear las entidades del dominio a las tablas de la base de datos. En OnModelCreating, se configuran las relaciones entre las entidades, como llaves foráneas y propiedades de navegación, lo cual asegura que las relaciones entre las tablas estén correctamente establecidas en la base de datos.

3. **Consultas con LINQ**: Las operaciones de consulta se realizan utilizando LINQ. Esto mejora la legibilidad del código y reduce la posibilidad de errores típicos de las consultas SQL.

4. **Inyección de Dependencias:** Se utiliza la inyección de dependencias para gestionar las instancias de ApplicationDbContext y los repositorios, lo que facilita la configuración de las dependencias en los controladores y servicios.

Inyección de Dependencias

La inyección de dependencias se utiliza ampliamente en la solución para gestionar las instancias de servicios, repositorios y el ApplicationDbContext. Esto se logra configurando los servicios en el contenedor de inyección de dependencias de .NET, lo que permite que sean inyectados automáticamente en los controladores y otros servicios.

Este enfoque ofrece varios beneficios. En primer lugar, facilita el desacoplamiento entre las clases, ya que estas no son responsables de instanciar sus propias dependencias, lo que reduce la dependencia entre componentes. Además, mejora la legibilidad del código, lo que a su vez facilita su mantenimiento. La inyección de dependencias también simplifica la creación de mocks para pruebas unitarias, lo que hace que el código sea más fácil de testear. Finalmente, fomenta el cumplimiento de principios SOLID, en particular el Principio de Abierto/Cerrado (OCP) y el Principio de Responsabilidad Única (SRP).

Patrones y principios de diseño

Patrones de diseño

En primera instancia se ha implementado el **patrón repositorio** para encapsular la lógica de acceso a datos y proporcionar una API limpia y coherente para interactuar con la base de datos. Cada entidad relevante tiene un repositorio asociado que facilita la realización de operaciones CRUD. Al encapsular la lógica de acceso a datos, cualquier cambio en la implementación de la persistencia (como un cambio de base de datos) solo afecta a los repositorios, no a la lógica de negocio, favoreciendo a la mantenibilidad. Por otra parte, los métodos genéricos de IRepository<>, como GetAll, Add y Update, permiten reutilizar la lógica de acceso a datos para distintas entidades, reduciendo la repetición de código.

El **patrón Facade** en nuestra entrega se aplica a través de los controladores que simplifican la interacción del cliente con el sistema al exponer una API clara. Los controladores actúan como una interfaz que oculta la complejidad de la lógica de negocio y la interacción con los servicios subyacentes. Por ejemplo, para registrar un dispositivo, el cliente interactúa solo con el DeviceController, que delega las tareas al DeviceService. Esto desacopla la lógica de presentación de la lógica de negocio y del acceso a datos, facilitando el mantenimiento y la evolución del sistema sin afectar la experiencia del cliente.

Principios SOLID

Principio de Responsabilidad Única (**SRP**): Cada clase tiene una única responsabilidad, lo que significa que debe tener un solo motivo para cambiar. Un ejemplo puede ser que DeviceService solo gestiona dispositivos, sin encargarse de lógica relacionada con usuarios o empresas.

Principio de Abierto/Cerrado (**OCP**): Las clases deben estar abiertas para la extensión, pero cerradas para la modificación. La implementación de los servicios (como DeviceService y HomeService) permite agregar nuevos métodos sin modificar la estructura existente de la clase. Este principio reduce el riesgo de introducir errores al modificar clases existentes, promoviendo una mayor estabilidad del sistema a medida que se agregan nuevas funcionalidades.

Principio de Abierto/Cerrado (**OCP**): Las clases deben estar abiertas para la extensión, pero cerradas para la modificación. La implementación de los servicios (como DeviceService y HomeService) permite agregar nuevos métodos sin

modificar la estructura existente de la clase. Este principio reduce el riesgo de introducir errores al modificar clases existentes, promoviendo una mayor estabilidad del sistema a medida que se agregan nuevas funcionalidades.

Principio de Sustitución de Liskov (**LSP**): Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento esperado del programa. Los servicios y repositorios que implementan interfaces como IHomeService pueden ser sustituidos por otras implementaciones sin alterar la lógica que depende de ellos. Facilita la reutilización de código y mejora la capacidad de pruebas y sustitución de componentes.

Principio de Segregación de Interfaces (**ISP**): Las interfaces deben ser específicas y enfocadas en un propósito concreto, en lugar de interfaces genéricas que obliguen a las clases a implementar métodos que no necesitan. Interfaces como IHomeService, INotificationService y IRepository<> están diseñadas para cumplir con responsabilidades específicas de cada contexto. Esto permite que cada servicio tenga una dependencia sólo de las interfaces que realmente utiliza, evitando la implementación de métodos innecesarios. Facilita la mantenibilidad y mejora la claridad del diseño al definir contratos claros y específicos para cada servicio.

Principio de Inversión de Dependencias (**DIP**): Las clases deben depender de abstracciones (interfaces), no de implementaciones concretas. Los servicios como HomeService dependen de su interfaz, en este caso IHomeService. Mejora la flexibilidad del sistema y facilita la introducción de nuevas implementaciones sin afectar a la lógica de alto nivel.

Patrones GRASP

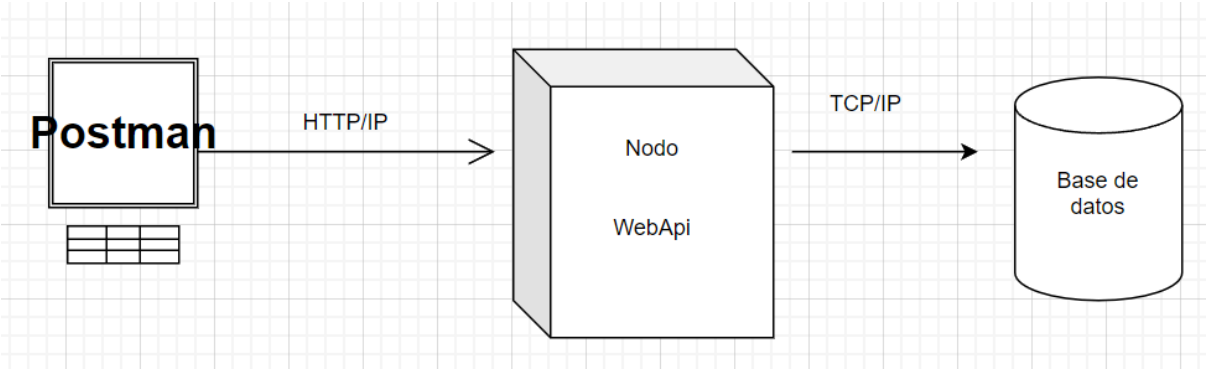
Controlador: Asigna la responsabilidad de manejar eventos de entrada a un objeto controlador que delega la lógica a otros objetos. Los controladores de la API, como DeviceController, actúan como intermediarios entre la capa de presentación y la lógica de negocio, delegando las solicitudes a los servicios correspondientes. DeviceController maneja la solicitud POST para registrar una cámara de seguridad, valida la entrada y delega la lógica a DeviceService a través de su método RegisterSecurityCamera.

Creador: Asigna la responsabilidad de crear instancias de una clase a la clase que tiene la información suficiente para hacerlo. DeviceService crea instancias de dispositivos cuando se registran en el sistema, ya que tiene toda la información necesaria para validar y construir un dispositivo. Mejora la cohesión al agrupar la lógica de creación y los datos necesarios en un solo lugar. DeviceService es responsable de crear una nueva instancia de Device al registrar una cámara de seguridad. Este servicio tiene la información necesaria para validar y construir el dispositivo, siguiendo el patrón Creador.

Alta Cohesión: Asigna responsabilidades de manera que se mantenga un nivel alto de cohesión, lo que significa que las clases tienen un propósito claro y pocas responsabilidades. NotificacionService es altamente cohesivo al gestionar todas las operaciones relacionadas con las notificaciones. Facilita la reutilización y el mantenimiento de las clases, ya que sus responsabilidades están bien definidas y son fáciles de comprender. NotificationService tiene una alta cohesión porque se encarga de toda la lógica relacionada con las notificaciones. Aquí, gestiona la creación de una notificación de detección de movimiento, lo cual facilita el mantenimiento al mantener la lógica de notificaciones centralizada en una clase.

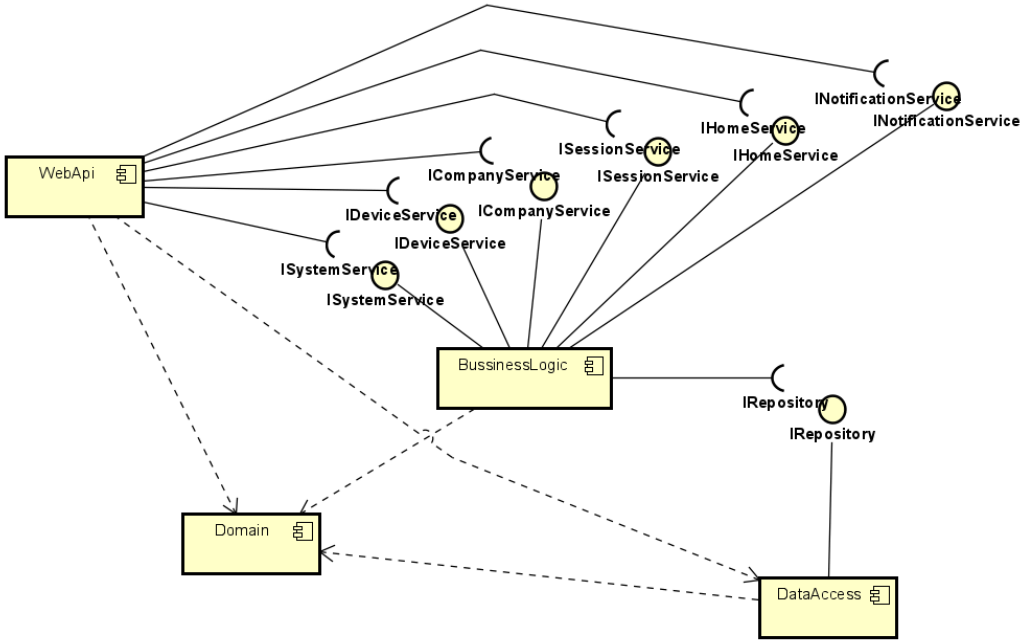
Experto: Asigna la responsabilidad de una tarea al objeto que tiene la información necesaria para cumplir con esa responsabilidad. IRepository es responsable de las operaciones CRUD porque tiene acceso directo a las entidades y sus datos. Minimiza la necesidad de transferir información entre objetos, lo que simplifica el diseño y mejora la eficiencia. Repository sigue el patrón de Experto en Información porque tiene el conocimiento necesario para realizar operaciones CRUD sobre las entidades (Get, Add, Remove, etc.). Al estar directamente conectado con la base de datos, sabe cómo obtener y manipular la información de las entidades, lo que simplifica el acceso a los datos desde los servicios.

Diagrama de despliegue



El diagrama de despliegue muestra cómo una solicitud HTTP es enviada desde Postman a través del protocolo HTTP/IP hacia un nodo que ejecuta la Web API. Este nodo procesa la solicitud y, mediante el protocolo TCP/IP, se comunica con una base de datos para leer o escribir información. El flujo de datos entre los componentes refleja cómo las operaciones de la API son gestionadas por la base de datos mediante conexiones de red.

Diagrama de componentes



Este diagrama de componentes muestra las dependencias y relaciones entre los distintos paquetes del sistema SmartHome. El paquete WebApi depende de

múltiples servicios definidos en la capa de lógica de negocio (BusinessLogic). Estos servicios, a su vez, se implementan dentro del paquete BusinessLogic, que interactúa con la capa de acceso a datos (DataAccess) a través de la interfaz IRepository para gestionar la persistencia y recuperación de datos.

El paquete Domain define las entidades fundamentales del negocio y está dependido tanto por BusinessLogic como por DataAccess. La capa de lógica de negocio utiliza las entidades del dominio para procesar las operaciones, mientras que DataAccess necesita las definiciones del dominio para mapear las entidades a la base de datos. En resumen, el diagrama refleja cómo la capa WebApi solicita acciones a los servicios de negocio, los cuales a su vez dependen de la capa de acceso a datos y las entidades del dominio para llevar a cabo sus operaciones.

Decisiones de diseño particulares

Cada usuario del sistema tiene asignado un rol, que puede ser uno de tres tipos: Administrator, CompanyOwner o HomeOwner. Se decidió llevar a cabo la clase Usuario y que esta sea concreta, en buena parte para que lleve una menor complejidad a la hora de mapearla a la base de datos. Por tanto, para poder implementar los requisitos es fundamental que según el rol que tengan los usuarios, puedan poseer diferentes atributos. Los usuarios con rol de HomeOwner son los únicos que tendrán una lista de Residentes, el resto de los usuarios tendrán esta lista nula. La misma situación sucede con la “ProfilePicture”.

Por otra parte los únicos usuarios que tienen el atributo booleano “IsComplete” son los usuarios con rol de CompanyOwner, ya que este atributo indica que ya poseen son dueños de una compañía. Este atributo “cambia” a true, cuando en el endpoint de Crear empresa, se le asocia el usuario logueado a la empresa y se llama al método “UpdateIfUserIsComplete” del SystemService para ejecutar este cambio. Este hecho es presentado en el diagrama UML mediante las constraints, dejando en claro que únicamente los usuarios con este rol pueden ser justamente los dueños de una empresa. Similarmente sucede y también está demarcado por una constraint, solamente los usuarios con rol Home Owner pueden ser los Owners de un Home.

En cuanto a la gestión de los roles y de los permisos, se lleva a cabo de la siguiente forma. Los roles cuentan con una lista de permisos que definen qué acciones pueden llevar a cabo, pero para los usuarios con rol Home Owner existen otras barreras que definen muchos de los métodos que pueden llevar a cabo. En cuanto a los “métodos del hogar”, para los métodos de “Listar dispositivos por hogar”, “Agregar dispositivo a un hogar” y “Listar notificaciones del hogar” el usuario con este rol precisa validar que sea el dueño del hogar o que contenga el permiso en este hogar para validar que puede consumir el endpoint. Estas validaciones se realizan, primero consultando si para el hogar referido el usuario logueado es el mismo que el Owner del Home, mediante la comparación de los Id de los usuarios . La segunda validación se lleva a cabo chequeando que sí el usuario tiene en su lista de residentes, una entidad asociada al hogar que se busca ejecutar la acción y chequeando en la lista de HomePermissions del residente sí es que posee la acción necesaria.

Otro punto clave en la API es el manejo de los Devices, la cual también es una entidad concreta por las mismas razones que el Usuario. Esta clase representa el dispositivo lógico, la cual puede haber 2 tipos “securityCamera” y “windowSensor”, lo cual estos últimos poseen 3 de los atributos nulos que son el MotionDetectionEnabled, PersonDetectionEnabled y UsageType (que puede ser “Exterior” o “Interior”), ya que estos están creados específicamente para las cámaras de seguridad. Esta clase tiene una relación de asociación con Company tal como indica el UML, porque se asocia la empresa del usuario que realiza la request a la nueva entidad de Device. Por otro lado, al agregar un dispositivo a un hogar se genera el dispositivo físico -HardwardDevice-, que tiene relaciones de asociación con el Device lógico y el Home al cual se añadió. Sí el Device asociado es de tipo “securityCamera” poseerá el atributo de OpeningState nulo, ya que este está creado para el uso de los “windowSensor”.

En cuanto a las empresas, se valida que el RUT y Name sean únicos, que el logo no posea la misma dirección en string que otro logo guardado en la base de datos. En cuanto a los hogares, para la creación de estos se puede ingresar una cantidad de residentes del hogar pero que estos no van a ser “manipulados” por el sistema y se valida que la cantidad de residentes ingresada no sea mayor a la cantidad de

residentes permitidos. Puede ser que se ingresen 2 usuarios como residentes al hogar al momento de la creación pero el sistema no los tomará en cuenta, como por ejemplo al momento de ejecutar el endpoint de listar residentes de un hogar solamente listará a los usuarios que fueron añadidos como residentes al hogar mediante el endpoint de agregar usuarios a un hogar.

La gestión de las notificaciones se lleva a cabo de la siguiente forma, cuando se ejecuta una acción de los dispositivos, se genera una entidad de la clase Notificación, que lleva el contenido específico de esta. Manteniendo una relación de asociación con Hardware Device, manteniendo de atributo al dispositivo físico que lanzó la notificación. A su vez, se generan las entidades de las clases NotificationUser que tienen la clase Notification asociada y el User al cual debe “enviarse” esa notificación, por ende posee relaciones de asociación con esas 2 clases.