



Stella Steinfeld (229398)
Eugenia Matto (227056)
Santiago Vairo(244023)

Diseño de aplicaciones 2
Descripción de Diseño

<https://github.com/IngSoft-DA2/Steinfeld-Vairo-Matto>

Declaración de autoría

Nosotros, Stella Steinfeld, Eugenia Matto y Santiago Vairo, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizamos el obligatorio de Diseño de Aplicaciones 2;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Índice

Declaración de autoría.....	2
Índice.....	3
Decisiones de diseño y estructura de paquetes.....	4
Diagrama UML.....	8
Diagrama de paquetes.....	9
Base de datos.....	11

Decisiones de diseño y estructura de paquetes.

Al comenzar a diseñar nuestra API, decidimos estructurar la aplicación en tres capas: repositorios, servicios y controladores. Esto se alinea con los principios de bajo acoplamiento y alta cohesión, asegurando que cada componente tenga responsabilidades bien definidas. Dividimos la aplicación en paquetes específicos: el paquete businesslogic contiene los servicios y sus interfaces, lo que facilita las pruebas unitarias mediante Test-driven Development (TDD). De manera similar, el paquete data access alberga los repositorios, también con interfaces para permitir su intercambio y pruebas sencillas.

El paquete web api implementa los controladores, mientras que el paquete domain contiene los objetos que se almacenarán en la base de datos, aplicando la Ley de Demeter para limitar las dependencias entre objetos. En el paquete model, se gestionan los objetos de solicitud y respuesta que utilizan los endpoints, con un uso eficiente de enumeraciones en los casos necesarios. Además, se implementa un paquete exception que incluye excepciones personalizadas y se implementa un filtro global para capturar todas las excepciones de la API, garantizando un formato coherente en las respuestas de error.

Para la seguridad, implementamos un filtro de autenticación basado en tokens, y otro filtro adicional para verificar los permisos del usuario en los endpoints que requieren autorización. Las validaciones de negocio se encuentran en el paquete de lógica de negocios, y seguimos el principio de Responsabilidad Única (SRP) al asegurar que los repositorios se centren únicamente en el acceso a la base de datos sin lógica adicional.

En cuanto a la implementación de herencia, creamos un SmartHubBaseController del cual heredan aquellos controladores que requieren autenticación, mientras que los que no necesitan esta validación son independientes. Este diseño permite la reutilización de métodos comunes, como GetUserLogged, que obtiene el usuario autenticado a través del token proporcionado en las solicitudes. Además, se implementan mecanismos para manejar excepciones de forma efectiva; por ejemplo, si el modelo (el objeto que se pasa a través de la solicitud en los métodos POST) no es válido, el controlador puede gestionar esta situación y devolver un mensaje de error estructurado que facilite la identificación del problema.

Además, en el diseño de los dispositivos del sistema, utilizamos herencia para la clase Device como Cámara de Seguridad, en el futuro esto nos facilitaría el añadir más tipos de dispositivos. Este enfoque permite encapsular características comunes de los dispositivos, como propiedades básicas (ID, nombre, estado) y métodos funcionales (encender, apagar, obtener estado). Al extender la clase Device, la Cámara de Seguridad puede heredar estas propiedades y métodos, a la vez que añade funcionalidades específicas. Esto no solo optimiza el código al evitar la duplicación de lógica, sino que también facilita la ampliación del sistema en el futuro al permitir la adición de nuevos tipos de dispositivos que compartan características comunes.

Además, para garantizar una mejor organización y facilitar la expansión futura del sistema, tanto la Cámara de Seguridad como el Sensor de Ventana tienen sus propios controladores dedicados. Esta decisión permite manejar específicamente las notificaciones y la creación de instancias de cada uno de estos dispositivos de manera independiente, lo que a su vez mejora la cohesión del código. Al separar las funcionalidades en controladores individuales, se simplifican las interacciones y se optimizan las operaciones relacionadas con cada tipo de dispositivo. Por ejemplo, el controlador de la Cámara de Seguridad puede gestionar las notificaciones en tiempo real, como alertas de movimiento, o detección de persona, mientras que el controlador del Sensor de Ventana puede enfocarse en la detección de apertura y cierre de ventanas, enviando notificaciones pertinentes al usuario cuando se detectan eventos inusuales. Esta arquitectura modular no solo facilita la comprensión y el mantenimiento del código existente, sino que también permite la integración de nuevos tipos de dispositivos en el futuro con un esfuerzo mínimo. A medida que se añaden más dispositivos, se puede crear un controlador específico para cada uno, asegurando que el sistema permanezca escalable y adaptable a las necesidades cambiantes de los usuarios y del mercado.

Esta arquitectura orientada a objetos y el uso de herencia proporcionan una base sólida para el diseño de la aplicación, asegurando que los cambios y mejoras se realicen de manera eficiente. Por ejemplo, si decidimos introducir un nuevo tipo de dispositivo, simplemente podemos crear una nueva clase que herede de Device, minimizando así el esfuerzo de implementación y manteniendo la cohesión en el diseño del sistema. Al adoptar esta metodología, también fomentamos la implementación de patrones de diseño, como el Factory Method, para facilitar la creación de instancias de diferentes tipos de dispositivos, permitiendo que la lógica de creación esté centralizada y controlada.

Implementamos principios SOLID para mantener un diseño limpio y mantenible, siguiendo la fabricación pura en la creación de objetos y aplicando indirección para disminuir el acoplamiento entre componentes. Utilizamos patrones de diseño como el Patrón Facade, que se utilizó para simplificar las interacciones complejas con el sistema. A través de este patrón, se crearon clases que actúan como interfaces simplificadas para las operaciones más complicadas que involucran múltiples subsistemas. Por ejemplo, al implementar la gestión de usuarios y hogares, el Facade proporciona una interfaz clara y sencilla que oculta la complejidad del sistema subyacente, facilitando su uso por parte de nosotros los desarrolladores y mejorando la legibilidad del código.

Además, se aplicó el patrón de comportamiento Strategy. Se utilizó para encapsular algoritmos relacionados con la gestión de permisos de usuario. Este enfoque permite intercambiar diferentes estrategias de autorización según el contexto de la solicitud, lo que proporciona flexibilidad al sistema. Por ejemplo, se podrían definir diferentes estrategias de autorización para usuarios regulares y administradores, facilitando la adaptación a futuros requisitos.

El diseño del sistema de permisos implementa un enfoque basado en roles y permisos individuales, utilizando patrones de diseño que aseguran flexibilidad y escalabilidad. Un ejemplo clave de este diseño es la aplicación del patrón Template Method para la creación de cuentas de diferentes tipos de usuarios (administradores, propietarios de hogares y propietarios de empresas) dentro de los servicios `AdministratorService`, `HomeOwnerService` y `CompanyOwnerService`. Este patrón permite que cada tipo de cuenta defina pasos específicos (como los permisos asociados), pero manteniendo una estructura general común que simplifica la creación y asignación de roles y permisos.

Técnicamente, el repositorio de permisos (`_permissionRepository`) actúa como un Singleton y administra la asignación y verificación de permisos, asegurando que las reglas de negocio sobre quién puede realizar qué acción estén centralizadas y se mantengan consistentes en todo el sistema. Cada vez que se crea una cuenta, se asignan permisos predeterminados y específicos utilizando el método `AddNeutralPermissions`, lo cual implementa el principio de Single Responsibility al delegar la gestión de permisos a su propio repositorio, separando esta lógica del manejo de usuarios.

El sistema también garantiza que las relaciones entre los usuarios y los permisos se mantengan claras y estructuradas. Por ejemplo, en `HomeOwnerService`, los permisos como

"newhome-home" y "adddevice-home" están explícitamente asignados al usuario, controlando qué acciones puede realizar sobre un hogar específico. De manera similar, el `CompanyOwnerService` asigna permisos como "createcompany-companies" para delimitar las capacidades del usuario dentro del contexto empresarial. Con la implementación del patrón `Repository`, se encapsula la lógica para agregar, eliminar y verificar los permisos en el `IPermissionRepository`, lo que facilita la extensibilidad y el mantenimiento del código.

El Patrón `Template Method` también se aplica en el contexto de la gestión de dispositivos, específicamente al momento de guardar objetos de tipo `Device` en la base de datos a través de un único método en el `DeviceRepository`. Este patrón permite definir la estructura general del algoritmo para el almacenamiento de un dispositivo, mientras que ciertos pasos específicos se delegan a las subclases o se adaptan según el tipo de dispositivo que se está gestionando. En este caso, se utiliza un método común, que se encarga de las operaciones básicas de almacenamiento, como la validación del objeto y la interacción con la base de datos.

La `Unified Modeling Language (UML)` se utilizó para diagramar la estructura y comportamiento del sistema, facilitando la comunicación entre los miembros del equipo. Para el diseño de nuestras APIs, seguimos las mejores prácticas de `REST`, asegurando que los recursos estén bien definidos y que las interacciones sean intuitivas. Implementamos enlaces en tiempo de ejecución y utilizamos `reflection` e inyección de dependencias para mejorar la flexibilidad y testabilidad del código.

Referencias como "Web API Design, Crafting Interfaces that Developers Love" de B. Mulloy nos guiaron en la aplicación de estos conceptos y principios, garantizando que la API sea robusta y fácil de mantener. En resumen, la organización y el diseño de nuestra aplicación se fundamentan en principios sólidos que aseguran su escalabilidad y facilidad de uso, tanto para desarrolladores como para usuarios finales.

A continuación se presenta el diagrama UML del proyecto, que ilustra la estructura general del sistema y las relaciones entre los diferentes componentes que lo conforman. Este diagrama muestra la organización en capas de la aplicación, dividiéndose en repositorios, servicios y controladores, además de las entidades del dominio y sus interacciones. Cabe destacar que, para facilitar un entendimiento más completo, no se modelaron las interfaces en este diagrama, aunque su implementación en el código es clave para garantizar la testabilidad y la intercambiabilidad de componentes, siguiendo principios `SOLID` y buenas prácticas de diseño.

Diagrama UML.

Se podrá observar en mejor calidad en el repositorio de GitHub.

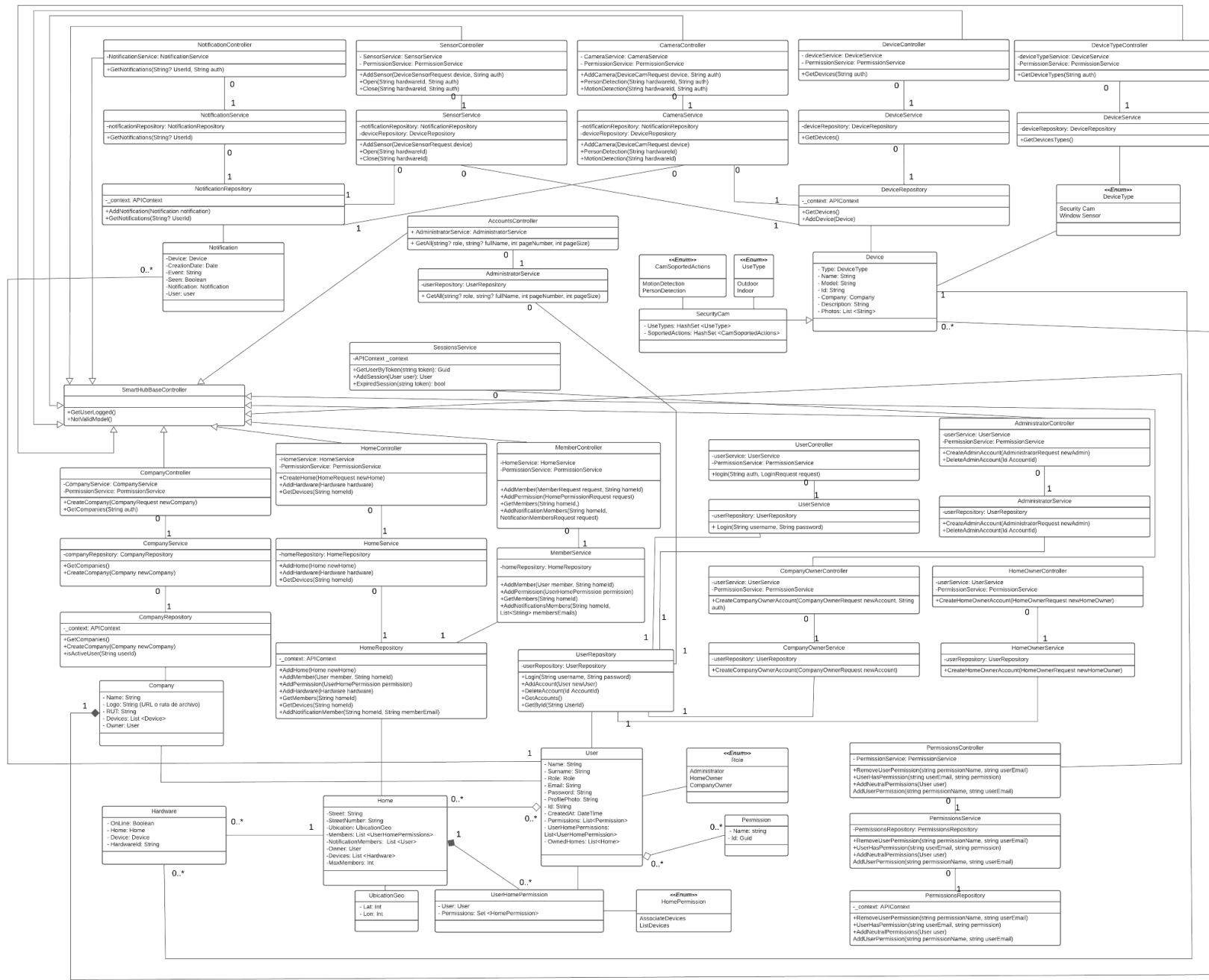
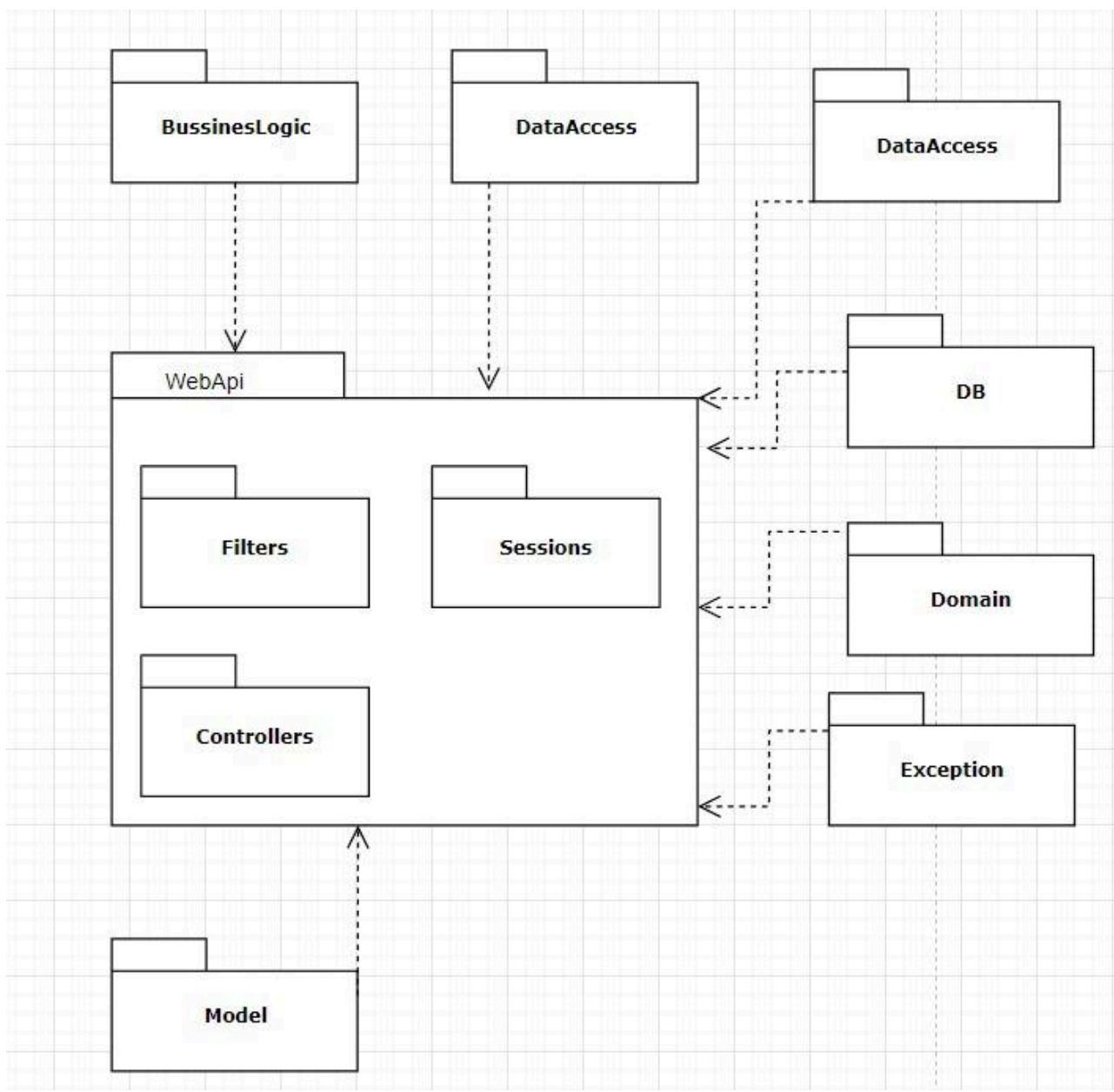
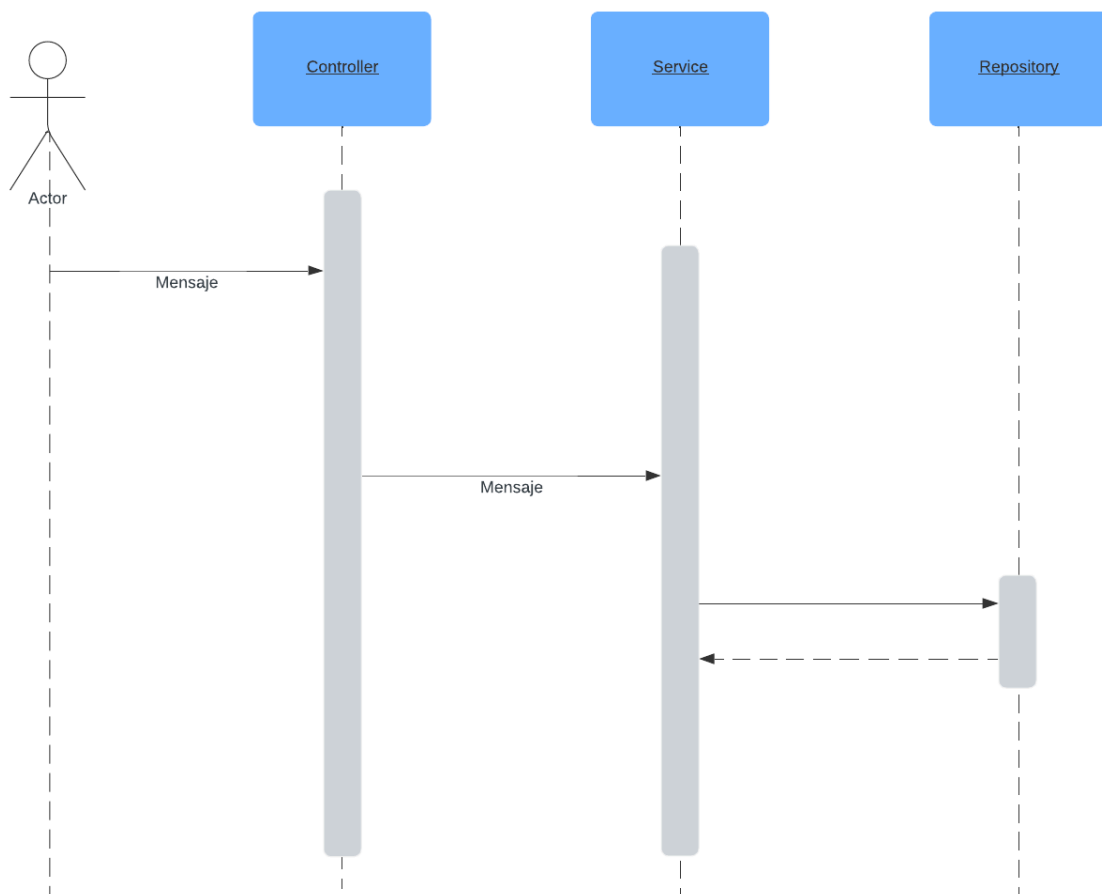


Diagrama de paquetes

El diagrama de paquetes del proyecto ilustra la organización de los diferentes componentes de la aplicación, distribuidos en capas y módulos específicos. Cada paquete contiene elementos que cumplen con responsabilidades claras, como los repositorios, servicios, controladores y objetos de dominio, permitiendo así una estructura ordenada y fácil de mantener. La segmentación del sistema en estos paquetes facilita la escalabilidad y promueve una arquitectura limpia. Aunque el diagrama no incluye las interfaces para lograr una representación más simplificada, estas son esenciales dentro del diseño, garantizando la intercambiabilidad y facilidad de prueba del código.



En este punto, se presentará un diagrama de secuencia que ilustra el comportamiento general de las interacciones entre el controlador, el servicio y el repositorio en el sistema. Este diagrama representa el flujo de operaciones y cómo se comunican estos componentes para llevar a cabo una acción específica. A través de este esquema, se pueden observar las llamadas a métodos, la transferencia de datos y la gestión de respuestas a lo largo de la cadena de invocación. El objetivo de este diagrama es ofrecer una comprensión clara de la dinámica entre los diferentes niveles de la arquitectura, destacando cómo cada componente contribuye al funcionamiento general de la aplicación y al cumplimiento de los requisitos del sistema.



Base de datos.

En este proyecto, la persistencia de los datos se gestiona mediante Entity Framework Core, que nos permite mapear nuestras entidades del dominio a tablas en la base de datos. Dentro del contexto de la aplicación, representado por la clase `APIContext`, definimos una serie de `DbSet` que almacenan las principales entidades del sistema, como `Device`, `User`, `Permission`, `Home`, `Company` y `Notification`, entre otras. Cada una de estas entidades se relaciona entre sí mediante claves foráneas, lo que permite una estructura relacional clara y eficiente en la base de datos.























Uno de los aspectos clave de este diseño es el manejo de propiedades que requieren el almacenamiento de listas o colecciones. Para estos casos, utilizamos un enfoque en el que las listas se guardan como strings en formato JSON dentro de la base de datos. Esto es especialmente útil para atributos como las fotos de los dispositivos, o las acciones soportadas por una cámara de seguridad (`SecurityCam`). Almacenando estos datos en formato JSON en columnas tipo `nvarchar(max)`, logramos una mayor flexibilidad y simplicidad en el acceso a estos datos dinámicos sin necesidad de crear tablas adicionales. En la clase `SecurityCam`, por ejemplo, las propiedades `UseType` y `CamSupportedActions` se convierten en JSON para su almacenamiento, y un mapper en la clase se encarga de traducirlos de vuelta a listas cuando se recuperan desde la base de datos.

Las relaciones entre las entidades también están bien definidas. Por ejemplo, un hogar (`Home`) puede tener varios dispositivos asociados a él, lo que se refleja en una relación uno-a-muchos entre `Home` y `Device`. Asimismo, una compañía (`Company`) puede gestionar varios dispositivos, y esta relación también se modela como uno-a-muchos entre `Company` y `Device`. Cada dispositivo, por lo tanto, puede estar vinculado tanto a muchos hogares y a una compañía, dependiendo de su contexto de uso. Además, las notificaciones están asociadas directamente a un hardware, que sería un device asociado a un hogar, a través de una relación uno-a-muchos entre `Notification` y `Hardware`, lo que permite rastrear eventos importantes relacionados con ese dispositivo en particular.

En cuanto a la gestión de permisos, se han definido relaciones claras mediante las entidades `UserHomePermission` y `Permission`, que manejan los permisos que los usuarios tienen en diferentes hogares. La entidad `UserHomePermission` establece una relación muchos-a-uno

tanto con Home como con User, lo que permite que un usuario pueda tener permisos en varios hogares, y cada hogar puede tener múltiples usuarios con diferentes permisos. La entidad Permission, por su parte, se relaciona con usuarios y hogares para definir los tipos de acceso y acciones que cada usuario puede realizar dentro de un hogar específico. Esta estructura de relaciones asegura una correcta representación de la administración de accesos y la propiedad de dispositivos y recursos dentro del sistema.

En la siguiente imagen se muestran las tablas que componen la base de datos del proyecto. Cada tabla ha sido diseñada para reflejar las entidades fundamentales del sistema y sus relaciones, asegurando así la integridad de los datos y facilitando el acceso a la información necesaria para las operaciones de la aplicación. Estas tablas son el núcleo de la estructura de datos, permitiendo el almacenamiento eficiente y la recuperación de información relevante para las funcionalidades que se implementan en el sistema. A través de esta presentación, se busca proporcionar una visión clara de cómo se organiza la información dentro de la base de datos.

-   **dbo.Companies**
-   **dbo.Devices**
-   **dbo.Hardwares**
-   **dbo.Homes**
-   **dbo.NotificationMembers**
-   **dbo.Notifications**
-   **dbo.Permission**
-   **dbo.Sessions**
-   **dbo.User**
-   **dbo.UserHomePermissions**
-   **dbo.UserPermissions**