

Universidad ORT Uruguay
Facultad de Ingeniería

Segundo obligatorio

Diseño de Aplicaciones II

**Entregado como requisito para la obtención del título de
Licenciatura en Sistemas**

Guillermo Diotti - 285578

Juan Peyrot – 275076

Nicolás Toscano - 220264

Profesores:

Daniel Acevedo

Pablo Geymonat

<https://github.com/IngSoft-DA2/285578-220264-275076>

2024

Descripción general	4
Resumen de las 4 principales mejoras respecto a la entrega anterior	4
Uso de la clase HomifyDateTime	4
Configuración del entorno de Ejecución	5
Extracción de lógica de clases controladoras	5
Mejora en legibilidad con sealed y record.	5
Alcance	7
Independencia de Librerías	9
Persistencia de Datos	10
Control de Versiones	10
Justificación de la estructuración del proyecto	11
Vista de Implementación	11
Diagramas de paquetes	11
Diagrama general de paquetes	11
Diagrama por capas (layers)	11
Diagrama de componentes	12
Vista de Diseño	12
Diagrama de clases	13
Diagrama de clases del paquete BusinessLogic para la entidad Session	15
Diagrama de clases del paquete WebApi para la entidad Session	15
Diagrama de clases del paquete Exceptions	15
Diagrama de clases del paquete DataAccess	15
Diagrama de clases del paquete Utility	15
Diagrama de clases de ValidadorModeloSoloLetras y ValidadorModeloLetrasNumeros paquetes validadores	15
Diagrama de clases de JSONImporter y Importer.Abstractions	16
Vista de Procesos	16
Diagrama de interacción implementado	16
Descripción de jerarquía de herencias utilizadas	16
Modelo de tablas de la estructura de la base de datos	17
Breve descripción de Homify.Angular	17
Justificaciones de diseño	17
Patrones y Principios de diseño	17
Informe de métricas	19
Principios de diseño a nivel de paquetes	22
Principios Cohesionales	22
REP (Equivalencia de Rehuso-Liberación)	22
CCP (Clausura Común)	22
CRP (Rehuso Común)	23
Principios de Acomplamiento	23
ADP (Dependencias Acíclicas)	23
SDP (Dependencias Estables)	23
SAP (Abstracciones Estables)	23
Decisiones de Diseño	24

Administradores y dueños de empresa como dueño de hogar:	24
Agrupar dispositivos por cuarto	25
Eliminado de usuarios con múltiples roles	25
Acciones de los Home Owners	25
Uso del modelo validador	25
Excepciones en repositorios personalizados	25
Funcionamiento de la importación de dispositivos	26
Taxonomía y convenciones de nombre en clases y paquetes	26
Descripción del manejo de excepciones	26
Anexo	27
Informe de cobertura	27
Descripción de pruebas automatizadas	27
Análisis de las Heurísticas de Nielsen	28
1. Entidad HomifyDateTime	29
2. Entorno de ejecución	30
3. Métodos de rehusos de la clase Helpers	30
4. Diagrama general de paquetes	31
5. Diagrama de paquetes por layers	32
6. Diagrama de componentes	33
7. Diagrama completo de clases	34
8. Diagrama de clases de BusinessLogic tomando como ejemplo Session	35
9. Diagrama de clases de WebApi tomando como ejemplo Session	36
10. Diagrama de clases del paquete Exceptions	36
11. Diagrama de clases del paquete DataAccess	37
12. Diagrama de clases del paquete Utility	37
13. Diagrama de clases de los paquetes validadores	38
14. Diagrama de clases de los paquetes JSONImporter y Importer.Abstractions	38
15. Diagrama de Secuencia para la importación de dispositivos	39
16. Modelo de tablas de la estructura de la base de datos	40
17. Lógica de validación en excepciones	41
Cambios respecto a la especificación de la API	42
Interfaz de Usuario	51

Descripción general

Esta documentación tiene como propósito documentar la extensión de requerimientos y nuevas funcionalidades del proyecto **Web Api RESTful** de hogares inteligentes. Recordemos que se trata de una **Web Api** que cumple con el estilo arquitectónico **REST** porque presenta recursos claramente definidos, que son gestionados mediante las operaciones **CRUD** básicas (**POST, PUT, GET, DELETE**). Estos recursos son accedidos a través de **URLs**, siguiendo buenas prácticas en su diseño. Además, cada solicitud es **independiente** (stateless) de las anteriores, lo que significa que incluye toda la información necesaria para ser procesada. Finalmente, las respuestas del servidor contienen los **códigos de estado HTTP** adecuados, indicando si la operación fue exitosa o si ocurrió algún error.

Entre las principales funcionalidades implementadas, se encuentran la incorporación de nuevos dispositivos al sistema, la incorporación de separación de dispositivos en un hogar en distintos cuartos, la posibilidad de que terceros importen dispositivos en el sistema y la implementación de una aplicación cliente. Esta última representa la gran propuesta brindada en esta segunda evaluación. Se dejó de utilizar el cliente Postman y se desarrolló en cambio una aplicación **front-end** con el framework **Angular**.

Resumen de las 4 principales mejoras respecto a la entrega anterior

Antes de abordar los nuevos requerimientos propuestos, nos enfocamos en resolver los inconvenientes detectados y en completar las funcionalidades que aún no estaban implementadas. Si bien la única funcionalidad no implementada en la entrega anterior trataba de un detalle menor como lo es el filtrado por fecha en un determinado endpoint debido a problemas con los tipos **Date** del framework, fue solucionado rápidamente y es totalmente funcional en esta segunda etapa.

Uso de la clase **HomifyDateTime**

Durante esta tarea, notamos que la clase personalizada [HomifyDateTime](#), aunque definida, no estaba siendo utilizada. Para solucionar esto, modificamos las clases que incluyen atributos de fecha de creación, como los usuarios y las notificaciones, cambiando el tipo de dichos atributos de **DateTimeOffset** a string. Esto requirió algunos ajustes en la implementación de la clase **HomifyDateTime**, incluyendo el cambio en el valor de retorno de la función **Parse** a string, y la creación del método **GetActualDate()**. Este último se encarga de llamar a **Parse** para devolver la fecha actual en formato string, que luego se asigna a los objetos mencionados. Es mediante el uso de esta clase que solucionamos la el

filtrado por fecha, que fue un pequeño detalle que no estaba del todo funcional en la anterior entrega.

Configuración del entorno de Ejecución

Detectamos una omisión en la entrega anterior, que no fue debidamente documentada: la configuración del perfil de ejecución en el archivo **launchSettings.json**, que aún utilizaba el perfil por defecto de weatherforecast. Hemos incluido una [imagen](#) para ilustrar cómo quedó el archivo tras realizar los cambios necesarios.

Extracción de lógica de clases controladoras

Para mejorar la estructura de nuestro código y adherirnos a los estándares de **Clean Code**, en esta segunda instancia de evaluación se ha realizado una extracción en gran medida de lógica de los controladores de la Web API, delegando la responsabilidad de las validaciones, cálculos y lógica de negocio a clases dentro del paquete Homify.BusinessLogic. Este refactorio fue realizado en una rama separada. Al culminar este “proceso”, logramos que los controladores se enfoquen casi exclusivamente a las solicitudes y a llamar a distintas clases, mejorando así la claridad y separación de responsabilidades del código.

Sin embargo, aunque se ha quitado grandes cantidades de lógica mal ubicada, es posible que todavía quede algo de lógica restante en los controladores. Reconocemos que eliminar por completo la lógica de negocio de estas clases para dejarlas totalmente perfectas requiere un esfuerzo meticuloso y focalizado.

Para favorecer este procedimiento, hemos implementado, entre otras, un método en la clase [Helpers](#) del paquete **Utility**, llamado **ValidateRequest**. En el dicho método, incorporamos la lógica reutilizada para la validación de las requests en todos, o la mayoría, de los controladores. En la clase descrita se dedicaron otros escasos métodos a arrojar excepciones personalizadas y adicionalmente, **ValidatePaginationOffset**, **ValidatePaginationLimit**, encapsulando las validaciones de los campos pertinentes para las paginaciones

Mejora en legibilidad con sealed y record.

Para brindar una mayor prolijidad y lectura más ordenada del código, se ha procedido a incluir los adornos de sealed y record a las clases de entidades de dominio, a los DTO's y por último a las clases de la lógica de negocio.

Errores Conocidos o Funciones no Implementadas

i) Luego de varias pruebas, encontramos un bug en la aplicación, que ocurre al momento de importar dispositivos desde un archivo cuando la empresa ya tiene un modelo validador seteado, la problemática es que en algunas ocasiones se duplican los dispositivos a importar, es decir, se insertan dispositivos con los mismos valores más de una vez. Si bien esto no afecta negativamente la identificación de los dispositivos ya que por decisión de diseño (como se comentará más abajo) a la hora de importar dispositivos estos no se asumen como únicos por los datos que presentan, es algo que no debería ocurrir.

Intentamos exhaustivamente corregirlo pero no fue posible, logramos identificar que el problema radicaba en la app de Angular, donde se enviaban más solicitudes al backend de las necesarias sin motivo aparente para este caso concreto, lo que generaba datos de más. Sin embargo, probamos con varios clientes HTTP (Postman, Insomnia, ReqBin) enviando los mismos datos al backend y este problema no aparecía, por lo que seguramente esté radicado a algún problema en la implementación del frontend (cosa que nos parece demasiado extraña ya que todos los demás endpoints funcionan correctamente).

ii) Si bien se ha extraído la mayor parte de la lógica de las clases controladoras, se reitera que aún hay un pequeño porcentaje de estas clases que no están del todo “limpias” en este aspecto.

iii) Complementando el punto anterior, algunas de estas consumen más de una lógica de negocio, en lugar de uno solo. El hecho de que una clase controladora consuma únicamente un servicio es una condición ideal, y difícil de llegar, pero no cabe la duda de ser un escenario con severos beneficios.

iv) La solución presenta las clases HomeOwner y Admin vacías, las cuales al inicio del proyecto se usaban para identificar los distintos roles pero por falta de tiempo no pudieron ser eliminadas, lo correcto sería transpolar el comportamiento a un diseño más limpio.

v) Presentamos un problema con las Github Actions con las últimas Pull Requests, donde Build and Test nos daba error ya que no encontraba el path especificado para obtener los validadores en ciertas clases de testing, esto hizo que el código que funcionaba correctamente en desarrollo no pueda pasar esta action, y bajo autorización docente, recibimos la instrucción de desactivar la siguiente opción del repositorio con el fin de poder llevar los cambios finales a las ramas protegidas, lo cual agradecemos.

☐ **Do not allow bypassing the above settings**

The above settings will apply to administrators and custom roles with the "bypass branch protections" permission.

Se trata de un error de diseño nuestro al no considerar que nuestra solución para el requerimiento de los validadores pueda ser traspolada a otros ambientes, lo cual reconocemos y tomamos como un error conocido en el mismo.

Campos de mejora:

i) Consideramos que la interfaz para importar dispositivos puede ser mejorable. Si bien es completamente funcional, solo presenta un input (aparte del dropdown para elegir el tipo de importador) para ingresar datos, lo que puede ser un poco limitado para algunos tipos de fuentes de datos que necesiten separación en los mismos para ingresar varios valores en lugar de solo uno, lo que mejoraría significativamente la extensibilidad y alcance de nuestra solución.

ii) El uso del patrón State en la gestión de dispositivos inteligentes tiene ventajas importantes en cuanto a mantenibilidad y escalabilidad, especialmente para manejar los diferentes estados de conexión y funcionamiento de estos dispositivos. Durante la primera fase del proyecto, aunque se consideró la implementación de este patrón, no se contaba con la estructura adecuada, y el esfuerzo necesario no justificaba los beneficios, ya que actualmente los dispositivos solo tienen unos pocos estados. A futuro, el patrón State sería una gran mejora si se amplían los estados o se añade automatización, pero debido a los plazos y la complejidad, se optó por usar atributos booleanos para los estados. Anteriormente, ocurrió algo similar con el patrón Observer, que también se descartó por su complejidad y tiempo requerido.

Alcance

Se presenta de forma resumida, aquellas principales clases y paquetes afectados/creados/modificados en base a cada cambio implementado considerando los nuevos requerimientos para esta entrega.

Implementación de Interfaz de Usuario (UI)

- **Nuevo paquete:** Homify.Angular para la UI. Sin impacto en back-end.

Asignar Nombre al Hogar

- **Entidad Home en BusinessLogic:** Nuevo atributo Alias.
- **Afectación de servicios:** HomeService, IHomeService, AddHome y CreateHomeArgs.
- **WebApi:** Modificado HomeController con nuevos endpoints Create y UpdateHome y DTOs asociados.

Nombre Custom a Dispositivos

- **Entidad HomeDevice:** Atributo CustomName.
- **Permiso:** Nuevo permiso en PermissionGenerator.
- **Servicios:** Nuevo método en HomeDevice y IHomeDevice.
- **WebApi:** Modificado HomeDeviceController, nuevo endpoint PUT, actualización de DTOs.

Administradores y Dueños de Empresa como Dueños de Hogar

- **Entidad User:** Atributo Role (pasa a ser una relación de agregación para representar múltiples roles).
- **Servicios:** Cambios en CreateUserArgs, IUserService, UserService. Nueva tabla intermedia UserRole en HomeService.
- **DataAccess:** Modificado repositorio custom.
- **WebApi:** Nuevos y modificados endpoints en UserController y RoleController. Actualización de filtros y DTOs.

Nuevos Tipos de Dispositivos

- **Nuevas entidades:** Lamps, MovementSensor con su lógica específica.
- **Inyección de dependencias:** Servicios para los nuevos dispositivos.
- **Endpoints:** Nuevos endpoints y actualización de existentes en WebApi.
- **Modificación de DTOs:** Actualización en CreateDeviceArgs y DTOs pertinentes.

Selección de lógica de validación de modelo

- **Nuevos paquetes:** ValidadorModeloLetrasNumeros y ValidadorModeloSoloLetras.
- **Nueva referencia:** A los módulos afectados por este requerimiento se les agregar una referencia al archivo .dll provisto en aulas. Ahora la clase Company tiene un atributo del tipo IModeloValidador y otro llamado ValidatorType, este último con el objetivo de persistir el modelo validador elegido.

- **Endpoints:** Nuevos endpoints para obtener los validadores existentes, asignarlos a una empresa.

Listado de dispositivos de un hogar

- **Modificación de endpoint:** Endpoint en WebApi actualizado para listar dispositivos de un hogar.
- **DTOs:** Actualización de DTOs en WebApi.
- **Servicios:** Modificación de clases de servicio en BusinessLogic.

Agrupar dispositivos por cuarto:

- **Nueva entidad:** Room, conjuntamente de su lógica de negocio y clase CreateRoomArgs.
- **Inyección de dependencias:** Servicios necesarios para Room.
- **Relación en entidades:**
 - **HomeDevice:** Lista de Room.
 - **Room:** Lista de HomeDevice y atributo Home.
- **WebApi:** Nuevos endpoints para la gestión de cuartos; actualización de los endpoints impactados.

Importar Dispositivos Inteligentes

- **Nuevo paquete InterfacelImporter:**
 - **Interfaz** ImporterInterface: Base para implementaciones de importación de terceros.
 - **DTOs:** Estructuras de datos necesarias para la importación de dispositivos.
- **Implementación de importador JSON:**
 - Nuevo paquete JSONImporter para manejar la importación desde archivos JSON.
- **WebApi:**
 - Nueva clase controladora para la gestión de importación.
- **BusinessLogic:**
 - Lógica de negocio para soportar el proceso de importación de dispositivos.

Independencia de Librerías

En el diseño actual, cada componente lógico está implementado en ensamblados independientes, logrando una estructura modular y desacoplada que permite minimizar la dependencia entre ellos y, en consecuencia, el impacto de posibles cambios. Por ejemplo, el ensamblado **Homify.BusinessLogic** encapsula la lógica de negocio y expone los

servicios del sistema (**IHomeService**, **IUserService**). Actualmente no se utilizan métodos privados para limitar la visibilidad de ciertos elementos, salvo por una muy breve cantidad en la clase `Company`, lo cual podría ser una mejora a considerar para proteger la lógica interna del servicio respectivo. **Homify.WebApi**, por su parte, define los controladores, restringiendo su interacción tras la implementación de distintos filtros. Además, **Homify.Importer.Abstractions** define la interfaz de importación para que las implementaciones específicas (como **JSONImporter**) puedan implementarla sin afectar el resto del sistema. Este diseño modular permite que cada ensamblado se mantenga de forma independiente, aunque se reitera el hecho de que se podría lograr un mayor beneficio con el uso de métodos internos y privados para reforzar aún más el aislamiento y el encapsulamiento en los distintos módulos.

Persistencia de Datos

En el proyecto *Homify.DataAccess* se almacenan las migraciones, contexto (`HomifyDbContext`) y repositorios para el acceso a datos. Con este objetivo, se usó el ORM de Entity Framework Core, conjuntamente de una clase repositorio encargada de manejar las distintas operaciones sobre la capa de persistencia de datos, manejando los tipos `TEntity`. Para cada entidad necesaria, existen repositorios creados con diferentes implementaciones sobre los métodos comunmente compartidos, con el fin de poder extraer de la base de datos propiedades de los objetos cuyo tipo de dato sea no primitivo, ya que por defecto estas vienen con valor nulo.

Control de Versiones

La gestión del trabajo obligatorio se realizó mediante un repositorio en **GitHub**, siguiendo las metodologías planteadas por la consigna del proyecto. En cuanto a las técnicas de flujo, se adoptaron **GitFlow** y **TDD**. Se utilizó una rama para cada funcionalidad, y las pruebas fueron desarrolladas de forma anticipada al código, como marca el enfoque **TDD**. Aunque ambas técnicas fueron implementadas en la mayoría de los casos, es importante destacar que no se aplicaron de manera estricta. En algunas ocasiones, las ramas creadas para una funcionalidad específica afectaron múltiples requerimientos, lo que llevó a la creación de sub-ramas para poder trabajar en paralelo en diferentes tareas. Posteriormente, estas sub-ramas fueron fusionadas mediante el uso de merges. Esta técnica no es la más recomendada, por lo cual se la está documentando adecuadamente para que sea comprendida en su contexto.

En lo que respecta a **TDD**, se aplicó en el desarrollo del backend de la solución, aunque se presentaron algunas situaciones puntuales en las que no se cumplió rigurosamente con esta metodología. Puesto a esto, se documentaron algunas de estas excepciones en github para asegurar que fueran reconocidas y comprendidas.

Cada rama creada fue fusionada con la rama **develop** a través de **PRs**, los cuales fueron acompañados de descripciones del cambio. Además, se incluyeron capturas de pantalla de los endpoints creados en estos **Pull Requests**, lo que facilitó la comprensión del trabajo realizado y mejoró la trazabilidad del código a lo largo del desarrollo.

Justificación de la estructuración del proyecto

En cuanto a la estructuración de la solución propuesta, se continua con la aplicación de distribución en diferentes paquetes, siguiendo una arquitectura vertical separando por funcionalidades en lugar de por tipos. Esto nos permite lograr una alta cohesión y un bajo acoplamiento. Los paquetes implementados en la primer entrega son: **BusinessLogic**, **WebApi**, **Exceptions**, **Tests** y **Utility**. En cuanto a esta segunda evaluación, se crearon paquetes como: **Importer.Abstractions**, **JSONImporter**, **ValidadorModeloLetrasNumeros** y **ValidadorModeloSoloLetras** (evidentemente a todos ellos se le suma el proyecto de frontend *Homify.Angular*).

Es importante destacar que hemos decidido agrupar las entidades de dominio junto con la lógica de la aplicación dentro del mismo paquete, **Homify.BusinessLogic**, ya que ambas conforman el núcleo del negocio. En otras palabras, el dominio no puede existir sin la lógica, y la lógica carece de sentido sin el dominio.

Vista de Implementación

Diagramas de paquetes

Diagrama general de paquetes

[Foto de los paquetes y sus relaciones](#)

Respecto a la solución antecesora, se han creado nuevos paquetes. Estos son **ValidadorSoloLetras** y **ValidadorModeloLetrasNumeros**, encapsullando respectivamente las clases encargadas de la validación pertinente, el paquete **JSONImporter**, especificando dentro la funcionalidad del importador en dicho formato especificado y por último el paquete **Importer.Abstractions**. En esta última solamente se sitúa la interfaz que implementa cada validador específico que se implemente. Al presentar solamente la interfaz allí, ese paquete se resulta con un alto grado de re

Diagrama por capas (layers)

[foto de los paquete separados por layers](#)

En la capa de **Presentación** anteriormente situamos solamente el cliente Postman y se mencionaba que a futuro se situaría allí también la futura aplicación de Angular. Bueno, ahora implementada la aplicación frontenis, la hemos especificado en el diagrama de capas.

La capa de **Negocio de la Aplicación**, es la encargada de administrar y encapsular toda la lógica de la aplicación. La misma también abarca las entidades de **Dominio**.

Por último, la capa de **Persistencia**, es la encargada del manejo de datos y el coleccionamiento de los mismos. Actualmente se usa el motor de base de datos relacional **SQL** y el **framework** de **ef-core** para hacer posible la interacción con esta base de datos.

Los paquetes de los validadores y el del importador json, no sería del todo correcto colocarlos en la layer de presentación o la destinada a la lógica de negocio ya que en teoría no sería código desarrollado por nuestra parte, mas bien emularían código desarrollado por integrantes ajenos al equipo. Es en base a lo dicho que se ha creado una nueva layer llamada "servicios externos" y situado en ella los paquetes mencionados.

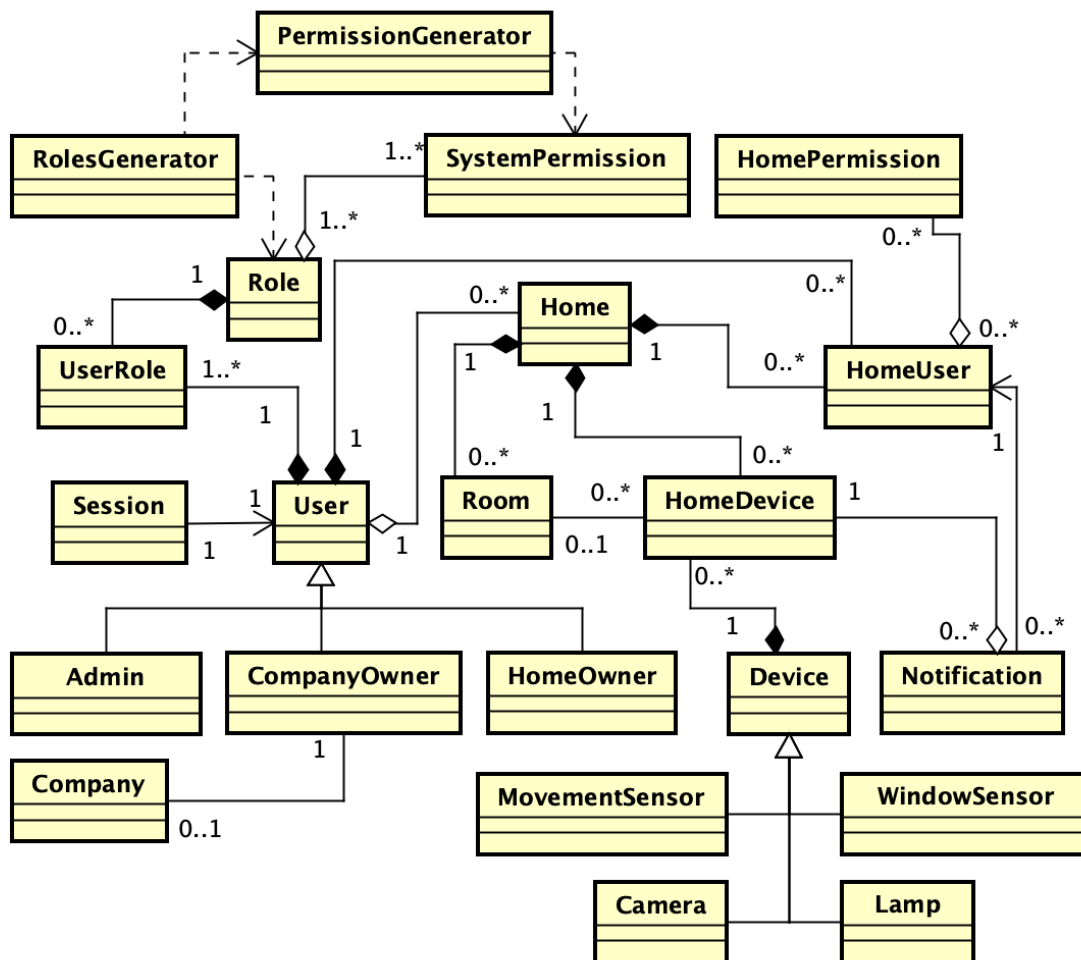
La modularización de la solución en distintas capas, presenta severos beneficios, entre los que se encuentran el cumplimiento de **SoC** (Separation of concerns), donde cada capa se encarga de una responsabilidad única. A su vez, este diseño modularizado favorece el mantenimiento y la escalabilidad a largo plazo, como la facilidad del desarrollo de pruebas unitarias.

Diagrama de componentes

En el diagrama de componentes propuesto, se puede observar los principales componentes de la aplicación. Se utilizó un componente por paquete, remarcando la responsabilidad única de cada uno de estos componentes. A su vez, se hizo uso del estereotipo, señalizando el tipo de componentes, en este caso dll o exe. En el diagrama también se pueden observar las interfaces implementadas, indicando claramente que paquete la implementa o la requiere. Por un lado, se encuentran las interfaces **I<<Entity>>Service**, respectivamente para las entidades que presenten controladores. Estas interfaces funcionan como capa de abstracción entre la clase de comportamiento lógico de la entidad y el controlador que solicite dichos comportamientos. Por otro lado, en el diagrama también se presenta la interfaz **IRepository**, encapsulando todas los métodos pertinentes para comunicarse con la base de datos. Finalmente, y de manera adicional al diagrama de componentes de la documentación previa, se incluye la interfaz **IImporter**, interfaz la cual sería implementada por terceros para definir el formato de archivo para importar dispositivos al sistema. Dato no menor, se resalta que en el diagrama de componentes se puede ver bien claro la dependencia entre los componentes.

Vista de Diseño

Diagrama de clases



Analizando exhaustivamente, en el diagrama se visualizan cuatro clases nuevas. Para empezar, se hallan aquellas provenientes de la herencia de Device, siendo Lamp y MovementSensor, reflejando los nuevos tipos de dispositivos abarcados en la solución. Consiguientemente, la clase Room se encarga de agrupar dispositivos de un hogar en concreto. Finalmente, la creación de UserRole. Esta clase intermedia presenta la mayor funcionalidad en el requerimiento adicional de administradores y dueños de empresa como dueños de hogar. En el apartado de decisiones de diseño, se proveerá una descripción más detallada del motivo que implicó la creación de la clase mencionada.

A continuación, se presenta un diagrama UML de las entidades de dominio del sistema. Este diagrama enfatiza los tipos de relaciones entre las clases, especificando con suma precisión su cardinalidad. Con el fin de simplificar el diagrama y destacar únicamente las relaciones, se han excluido los atributos de las clases.

Para no omitir información relevante, se cuenta con un [diagrama complementario](#) más detallado, reflejando todos los atributos presentes en las clases.

Justificación de las relaciones

Las clases Device, HomeDevice y Home se han relacionado mediante composición. El razonamiento detrás de esto es el siguiente: los HomeDevices representan dispositivos asociados a un hogar específico, por lo que esta clase intermedia contiene referencias tanto a Home como a Device. Si se eliminara un Home o un Device, la existencia de los HomeDevices relacionados se volvería redundante. Por lo tanto, la relación de composición es la más adecuada, ya que sigue el principio de compuesto/componente, donde al eliminar el compuesto (el Home o el Device), también se elimina el componente (el HomeDevice).

Aunque la lógica de eliminación de dispositivos y hogares no es un requerimiento que se haya codificado, la composición es la mejor manera de representar la relación entre estas clases intermedias y sus "padres". Aplicando exactamente esta misma lógica, es que se ha establecido nuevamente la composición entre las entidades User-HomeUser-Home, como también con las entidades User-UserRole-Role

Las relaciones restantes se comprenden de manera intuitiva. La relación Room - Home es de composición, ya que cumple con el principio de compuesto/componente: si se elimina un Home, se eliminan todos los Rooms asociados a dicho Home. Por otro lado, la relación entre Notification y HomeDevice es de agregación, donde una Notification puede estar relacionada con múltiples HomeDevices, pero de una forma menos fuerte que la composición. Esta misma relación de agregación se aplica entre Home y User, así como entre Role y SystemPermission.

Las relaciones de asociación se dan entre Company y CompanyOwner, Session y User, y entre Notification y HomeUser, ya que estas clases contienen un atributo simple de la clase con la que se relacionan.

Finalmente, existen relaciones de dependencia entre clases como PermissionGenerator y RolesGenerator, ya que dependen de la creación y manejo de objetos de la clase relacionada, además de retornar dichos objetos.

Uno de los nuevos requerimientos pertinentes se basa en que un usuario con un rol de Administrador o dueño de empresa, se puede comportar a su vez como dueño de hogar. Anteriormente al cambio mencionado, le hemos diseñado a la clase HomeOwner una lista de **hogares**, y puesto en **Home** un atributo owner de tipo **HomeOwner**. Frente a esta nueva

funcionalidad, se ha cambiado la lista de hogares a la clase **User** y refactorizado el tipo de dueño de hogar en **Home** a User, ya que se cambia el hecho de que únicamente los usuarios con rol de dueño de hogar pueden crear hogares.

Diagrama de clases del paquete BusinessLogic para la entidad Session

Para el siguiente diagrama se tomó como ejemplo la entidad Session, para no generar diagramas repetitivos y tratar de ilustrar la idea lo más sencillamente posible

[Diagrama de paquetes de BusinessLogic](#)

Diagrama de clases del paquete WebApi para la entidad Session

Para el siguiente diagrama se tomó como ejemplo la entidad Session, para no generar diagramas repetitivos y tratar de ilustrar la idea lo más sencillamente posible

[Diagrama de paquetes de WebApi](#)

Diagrama de clases del paquete Exceptions

[Diagrama de paquetes de Exceptions](#)

Se agregaron nuevas excepciones, como por ejemplo InvalidOperationException.

Diagrama de clases del paquete DataAccess

[Diagrama de paquetes de DataAccess](#)

Se agregaron nuevos DbSet y repositorios “personalizados” a la solución para poder manejar de mejor manera los datos.

Diagrama de clases del paquete Utility

[Diagrama de paquetes de Utility](#)

Se añadieron nuevas constantes en el paquete constants para poder abarcar los nuevos tipos de dispositivos incluidos en este obligatorio.

Además, se agregaron a la clase Helpers nuevas funciones para poder validar la paginación, distintos tipos de errores como pueden ser que algo sea nulo o que no lo encuentre, así también como una request.

También en la clase HomifyDateTime se agregó la función para obtener la fecha actual.

Diagrama de clases de ValidadorModeloSoloLetras y ValidadorModeloLetrasNumeros paquetes validadores

[Diagrama de las clases validadoras](#)

ValidadorModeloSoloLetras y ValidadorModeloLetrasNumeros ambas implementan de la interfaz IModeloValidador, ajenas a estos paquetes. La función que imlementan es llamada EsValido(), realizando una seria de operaciones para identificar la correctitud o no del modelo de los dispositivos. Estos modelos pueden ser especificados en la creación de la compañía dueña de los posteriores dispositivos, o asignados posteriormente

Diagrama de clases de JSONImporter y Importer.Abstractions

[Diagrama de clases de JSONImporter y Importer.Abstractions](#)

El paquete Importer.Abstractions presenta una interfaz, IImporter. La interfaz flexibiliza el uso de distintos modelos de importación de dispositivos. En el caso del proyecto brindado, se cuenta con el modelo de importación específico para un formato de JSON. Es a partir de la implementación de IImporter de dónde terceros crearán nuevos modelos de importación

Vista de Procesos

Diagrama de intaracción implementado

[Imagen de diagrama de secuencia de la creacion de dispositivos importados](#)

El diagrama de secuencia correspondiente a la **agregación de dispositivos importados** incluye las sentencias condicionales y muestra claramente a los distintos actores involucrados. Debido a su complejidad, algunas funciones demasiado detalladas se agrupan como bloques, representando abstractamente su flujo de llamadas dentro del método **AddImportedDevices** de la clase **ImporterService**.

Descripción de jerarquía de herencias utilizadas

En los diagramas se refleja la utilización de jerarquías de herencia en el modelo de clases.

Por un lado, las clases **Lamp**, **MovementSensor**, **WindowSensor** y **Camera** heredan de la clase **Device**, lo que permite a cada uno de estos dispositivos especializados heredar propiedades y comportamientos comunes definidos en **Device**. Cada subclase luego define ciertas propiedades en base a la funcionalidad concreta que presentan. Posible aspecto negativo de esta herencia, es que estas clases concretas heredadas no presentan atributos “propios”, aunque se definen el tipo y funcionalidad del dispositivo en ellas. El diseño mencionado es altamente escalable. En la presencia de nuevos dispositivos, basta simplemente don definir su comportamiento específico con las propiedades booleanas y en definir su nuevo tipo.

Por otro lado, se ha mantenido la herencia entre las clases: **User**, clase padre, y las clases hijas **Admin**, **HomeOwner** y **CompanyOwner**.. Al igual que la herencia previa, soluciona el rehuso de propiedades comunes entre todos los usuarios, siendo este diseño también mantenible y fácilmente escalable en la presencia de nuevos tipos de usuarios. Las clases **HomeOwner** y **CompanyOwner** extienden su funcionalidad al definir propiedades propias que presentan. Sin embargo, una gran desventaja de este diseño es el hecho de que la clase **Admin** no presenta distinciones únicas, resultando en una clase totalmente vacía.

Modelo de tablas de la estructura de la base de datos

[Imágen de la estructura de la base de datos](#)

Breve descripción de Homify.Angular

Como se ha mencionado previamente, la aplicación **Homify.Angular** consta de una **Single Page Application (SPA)**.

En las **MPA**, la comunicación **cliente-servidor** es un aspecto constante, frecuente, debido a que cuando un usuario interactúa con la aplicación, el navegador envía la request http al servidor para que la procese y este genere el **HTML** necesario para enviárselo al navegador del cliente. Por lo tanto, para modificar únicamente cierta sección del **HTML**, será necesario renderizar la página **HTML** completa. Entre los efectos que proporciona, los clientes se ven afectados por altas cantidades de tiempo para la carga de las páginas. No hace falta aclarar que hoy en día aquella es una práctica altamente ineficiente. Esto es algo que se soluciona con las **SPA**. Estas aplicaciones se distinguen fundamentalmente por la presencia de una única página **HTML**, mejorando significativamente la performance y el dinamismo.

Homify.Angular se clasifica como una página web **SPA**. Si se dirige al siguiente [anexo](#), se verá una descripción detallada sobre la aplicación de angular junto a sus componentes

Justificaciones de diseño

Inyecciones de dependencias

Se empleó la inyección de dependencias para registrar las implementaciones de los nuevos servicios y repositorios del proyecto *Homify*. Esto contribuye a mejorar la modularidad, la calidad del código y facilita la realización de pruebas unitarias mediante el uso de Mocks gracias a las interfaces. Para este propósito, se utilizó el método `AddScoped`, que establece que el ciclo de vida de los servicios es por cada solicitud HTTP. Esto asegura

que cada request obtenga su propia instancia de los servicios, lo cual es esencial para el funcionamiento correcto de los nuevos requerimientos.

Patrones y Principios de diseño

Principios SOLID

En la implementación de los nuevos requerimientos, se han seguido los principios SOLID. Las consideraciones sobre cada principio son las siguientes:

- **Single Responsibility:** Este principio indica que un módulo debe tener una única razón para cambiar, es decir, una sola responsabilidad. En el código, se puede observar que cada módulo se limita a lo necesario para cumplir con su función. En nuestro caso, siempre intentamos seguir que los controladores tuvieran la responsabilidad de recibir la request y enviar una response, mientras delegaban toda la lógica correspondiente a las clases de servicio, que a su vez delegan la parte del acceso, modificación, creación y eliminación de datos a otras clases pertinentes. Mantener cada clase con una única responsabilidad mejora la cohesión y reduce el acoplamiento, evitando que cambios en un método generen efectos colaterales en otros, lo cual no es deseable.
- **Open/Closed:** Un sistema debe estar abierto a la extensión, pero cerrado a la modificación. Este principio se ve reflejado principalmente en la parte de los importadores, permitiendo que el código se pueda extender (por ejemplo, añadiendo nuevas formas de importar dispositivos, desde otro tipo de archivos u otra metodología afín) sin necesidad de modificar el código existente, ya que existe una interfaz proporcionada para poder manejar los importadores independientemente de su implementación.
- **Liskov Substitution:** Las clases derivadas deben poder sustituir a sus clases base sin alterar el comportamiento del sistema. En *Homify.WebApi*, la herencia utilizada va de *AuthorizationFilterAttribute* a *AuthenticationFilterAttribute*, y se respeta este principio, ya que la clase derivada añade funcionalidad para la verificación de permisos, pero puede seguir empleándose en lugar de la clase base sin modificar la lógica general del sistema. ya que emplea los mismos métodos, y solo difieren en la implementación de los mismos, asegurando así la consistencia en la sustitución.
- **Interface Segregation:** Los módulos cliente no deben depender de métodos que no utilizan. Este principio se respeta en las nuevas interfaces de repositorios y servicios, ya que los clientes solo dependen de los métodos que realmente necesitan. Esto disminuye el acoplamiento y mejora la cohesión.

- **Dependency Inversion:** Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Invertir estas dependencias facilita la extensión y el cambio en las implementaciones. Tal como se expuso en la documentación previa, este principio se cumple, por ejemplo argumentando que las clases de lógica de negocio (paquete *Homify.BussinessLogic*) no dependen de la capa de acceso a datos (paquete *Homify.DataAccess*), sino que hay una abstracción en medio (*IRepository*) que evita que el módulo de alto nivel conozca la implementación, y por lo tanto dependa de un módulo de bajo nivel.

Principios de diseño a nivel de paquetes

Principios Cohesiones

REP (Equivalencia de Rehuso-Liberación)

Establece que un módulo debe tener la suficiente funcionalidad y cohesión para ser útil de forma independiente y reutilizable en diferentes contextos. La idea es que, si algo es suficientemente útil para ser reutilizado en otros proyectos o contextos, entonces merece ser un módulo separado. La evidencia más clara de este principio en **Homify** posiblemente radica en **Homify.Exceptions**, ya que contiene todas las excepciones controladas por el sistema, las cuales no dependen de otras partes del mismo para funcionar. En este caso, todas las excepciones definidas en **Homify.Exceptions** están diseñadas para manejar errores específicos y situaciones excepcionales dentro del sistema, pero no dependen de servicios específicos, controladores, o lógica de negocios, sino que son agnósticos a los mismos y pueden liberarse en conjunto para funcionar en otros contextos.

CCP (Clausura Común)

Indica que las clases que cambian por las mismas razones deberían estar en el mismo módulo. Esto significa que un módulo debe contener clases que son afectadas o modificadas por el mismo tipo de cambios. De esta manera, si hay una actualización o un ajuste necesario, sólo se necesitaría modificar un módulo en lugar de múltiples, facilitando la mantenibilidad del sistema. Un buen ejemplo es el paquete **Homify.BusinessLogic**, donde se conservan las clases de dominio, junto a los servicios e interfaces de los mismos, englobando todas juntas la denominada “lógica de negocio”, por lo que si se necesita hacer un cambio relativo a este campo, solo afectará a este paquete.

CRP (Rehuso Común)

Sugiere que las clases que suelen ser reutilizadas juntas deberían estar en el mismo módulo, con el objetivo de que si se necesita cambiar algo en ese conjunto de clases, todas las clases relacionadas están en el mismo lugar, minimizando el impacto del cambio. Por

ejemplo, ***Homify.Utility*** contiene clases auxiliares de gran utilidad que están destinadas a convivir en conjunto para ser reutilizadas en distintas partes de la aplicación.

Principios de Acomplamiento

ADP (Dependencias Acíclicas)

El principio de dependencias acíclicas se cumple sí y solo sí no existen relaciones en formas de ciclos entre los paquetes. Fortuitamente, en el diagrama que contamos de las relaciones entre los paquetes, se puede observar como no se halla ningún ciclo. La importancia de este principio es de suma importancia para la mantenibilidad y escalabilidad, al igual que para el correcto uso y ciclo de vida del sistema.

SDP (Dependencias Estables)

Establece que un paquete debe de depender de otro que sea más estable que el mismo. En nuestro caso, hay una situación donde no se cumple, ya que ***Homify.BusinessLogic*** depende de ***Homify.Importer.Abstractions*** que es más inestable, sin embargo, esta dependencia es necesaria ya que ***Homify.Importer.Abstractions*** actúa como intermediario entre la lógica de negocio y el código de terceros, por lo que sería un error no incluirlo en la solución. También se presenta un caso controversial, donde ***Homify.BusinessLogic*** ($I = 0.5$) depende de ***Homify.Utility*** ($I = 0.51$) que es ligeramente más inestable.

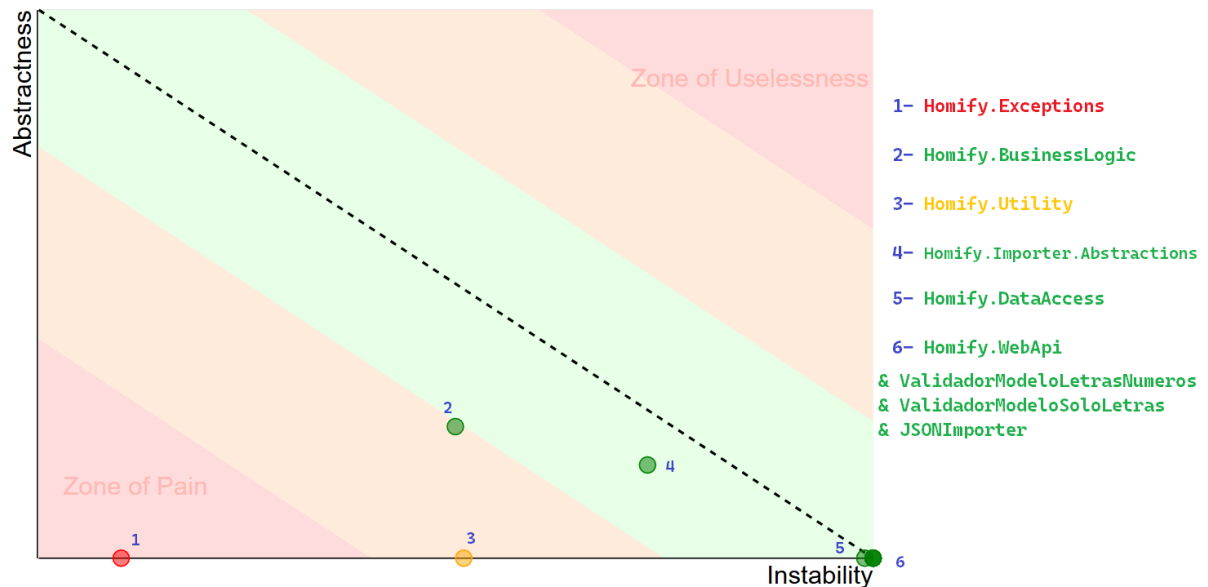
SAP (Abstracciones Estables)

Establece que un paquete debería ser tan abstracto como estable, es decir, que estos valores deberían estar próximos entre sí.

Viendo el diagrama de las métricas, podríamos determinar que los únicos paquete que no cumplen con dicho principio son ***Homify.BusinessLogic***, y ***Homify.Utility***, quienes son medianamente estables pero poco abstractos, lo que no es precisamente lo ideal.

Informe de métricas

Los siguientes gráficos se obtuvieron utilizando la herramienta NDepend.



Projects	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
Homify.Exceptions v1.0.0.0	36	4	0.17	0.1	0	0.64
Homify.Utility v1.0.0.0	19	20	0.25	0.51	0	0.34
Homify.Importer.Abstractions v1.0.0.0	4	11	0.83	0.73	0.17	0.07
Homify.BusinessLogic v1.0.0.0	62	63	3.4	0.5	0.24	0.18
Homify.DataAccess v1.0.0.0	1	115	1.39	0.99	0	0.01
Homify.WebApi v1.0.0.0	0	187	1.28	1	0	0
JSONImporter v1.0.0.0	0	20	1	1	0	0
ValidadorModeloLetrasNumeros v1.0.0.0	0	8	1	1	0	0
ValidadorModeloSoloLetras v1.0.0.0	0	8	1	1	0	0

Homify.Exceptions

- **Cohesión relacional (H):** Valor bastante bajo (0.17), si bien no es lo ideal, este paquete se define para proveer excepciones personalizadas que no se relacionan entre ellas.
- **Abstracción (A):** Es un paquete concreto (0).
- **Inestabilidad (I):** Poca inestabilidad (0.1), por lo que tiene pocas dependencias fuera del paquete, lo cual es esperado.

- **Distancia (D):** Se encuentra en la zona de dolor (0.64), que dado el caso no es tan malo como se plantea, ya que las excepciones están pensadas para no ser modificadas, y en caso de agregar nuevas estas no afectan a las anteriores.

Homify.Utility

- **Cohesión relacional (H):** Valor bajo (0.25), lo cual es esperable ya que ofrece clases de utilidad que no están pensadas para relacionarse entre ellas en este sentido.
- **Abstracción (A):** Es un paquete concreto (0).
- **Inestabilidad (I):** Medianamente estable (0.51).
- **Distancia (D):** Próximo a la zona de dolor (0.34), lo cual es esperable ya que estas clases no están pensadas para ser modificadas a menudo, y en caso de adicionar nuevas, las anteriores no se verían afectadas.

Homify.Importer.Abstractions

- **Cohesión relacional (H):** Valor por debajo del mínimo ideal 1.5 (0.83), sin embargo este paquete se ocupa de definir la interfaz para distintos importadores que puedan agregarse a futuro, por lo que no define comportamiento, lo que explica en parte este valor obtenido.
- **Abstracción (A):** Es un paquete concreto (0.17).
- **Inestabilidad (I):** Valor próximo al máximo (0.73), por lo que es inestable.
- **Distancia (D):** Próximo a la secuencia principal (0.07).

Homify.BusinessLogic

- **Cohesión relacional (H):** Es el paquete con mayor cohesión relacional (3.4), lo cual es esperable ya que aquí se almacena toda la lógica de negocio de la aplicación, la cual tiende a relacionarse entre sí como mostramos en los diagramas.
- **Abstracción (A):** Poco abstracto (0.24), paquete concreto.
- **Inestabilidad (I):** Medianamente estable (0.5).
- **Distancia (D):** Próximo a la secuencia principal (0.18) por lo que se muestra un equilibrio entre estabilidad y abstracción.

Homify.DataAccess

- **Cohesión relacional (H):** Valor próximo a 1.5 pero aún por debajo (1.39), tiene sentido que no sea tan cohesivo dado que su relación con las entidades de dominio solo se justifica para crear la capa de persistencia.
- **Abstracción (A):** Paquete concreto (0).
- **Inestabilidad (I):** Inestable (0.99), sin embargo los cambios aquí no tienen mayor efecto en paquetes externos.
- **Distancia (D):** Muy próximo a la secuencia principal (0.01).

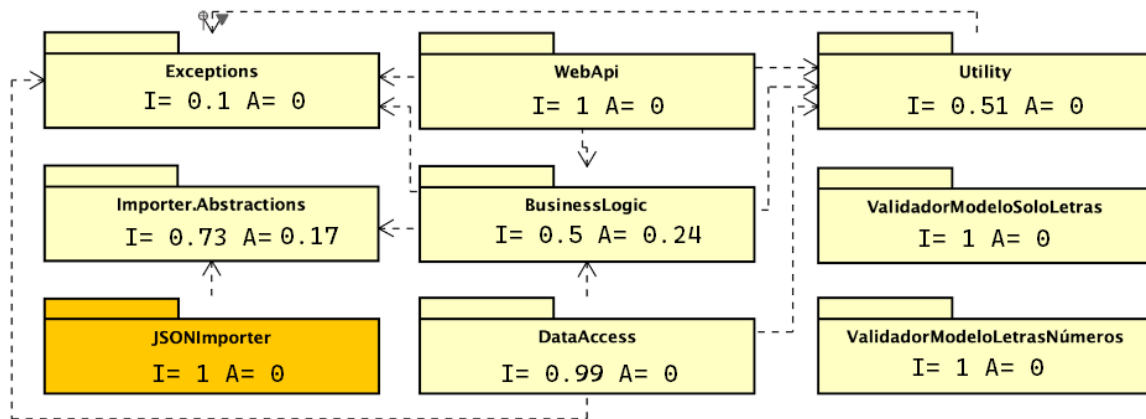
Homify.WebApi

- **Cohesión relacional (H):** Valor próximo a 1.5 pero aún por debajo (1.28), tiene sentido que no sea tan cohesivo dado que las relaciones más importantes de las clases se dan con las entidades de negocio y no entre clases de este mismo paquete.
- **Abstracción (A):** Paquete concreto (0).
- **Inestabilidad (I):** Inestable (1), sin embargo los cambios aquí no tienen mayor efecto en paquetes externos.
- **Distancia (D):** Directamente en la secuencia principal (0).

JSONImporter, ValidadorModeloLetrasNumeros y ValidadorModeloSoloLetras

- **Cohesión relacional (H):** Baja cohesión (1), esperable ya que estan pensados para generar un DLL de un tipo determinado de importador o validador.
- **Abstracción (A):** Paquetes concretos (0).
- **Inestabilidad (I):** Inestables (1), sin embargo los cambios en estos no tienen mayor efecto en paquetes externos.
- **Distancia (D):** Directamente en la secuencia principal (0).

A continuación, a modo de una mejor visualización se presentan los anteriores datos representados en los distintos paquetes de la aplicación.



Conclusión

Al analizar las métricas y los principios de diseño de paquetes, se puede concluir que el sistema presenta un diseño bastante adecuado. La mayoría de los paquetes se sitúan dentro de la secuencia principal, lo que indica una correcta organización y coherencia en la arquitectura. Aunque no todos los principios se cumplen al pie de la letra, lo cual es esperable dado que estos principios suelen describir un escenario ideal, en general se respetan de manera significativa. Además, el diseño se mostró bastante mantenible a la hora de incorporar nuevos requerimientos, dado que las dificultades presentadas estaban asociadas a cosas externas al diseño en su mayoría. Las modificaciones necesarias no resultaron complicadas de implementar y no afectaron de forma sustancial la estructura del sistema, lo que demuestra de alguna forma la flexibilidad del diseño existente.

Decisiones de Diseño

Administradores y dueños de empresa como dueño de hogar:

Dicho requerimiento, resultó ser uno de los que más esfuerzo se tuvo que empeñar, aunque no debido a un mal diseño anterior, sino a problemas surgidos con el manejo de la base de datos.

La solución optada, puede no llegar a ser la más “escalable” ni “prolija”. Esta se basó en cambiar el tipo de atributo “Role” de la clase User a ser en cambio una lista de estos objetos (List<Role>). Por más de haber realizado mucho refactorio, lo más dificultoso fue el momento de mapear la nueva relación en la base de datos. Concretamos el mapeo diseñando una entidad nueva llamada “UserRole” (tabla intermedia para representar la

multiplicidad de roles en un usuario), encargada de relacionar las tablas User y Role, con la idea de que la misma se cargase automáticamente. Sin embargo, por causas que no podemos determinar, eso no sucedió, por lo que las operaciones del registro de usuarios (AddAmin, AddCompanyOwner, AddHomeOwner en UserService) se encargaron de crear una instancia de UserRole al ser invocadas con la ayuda de una función auxiliar.

El cambio vino acompañado con la implementación de un endpoint nuevo con verbo PUT destinado a justamente agregarle el nuevo rol al usuario creando una instancia de UserRole con el id del usuario y el id del rol HomeOwner.

Resumiendo el proceso de creación de usuarios con diversos roles por el lado de la base de datos, al momento de crear un usuario, supongamos con un rol Admin, se crearán tanto una tupla en la tabla Users con los campos como nombre y apellido, como una tupla en la tabla Admins, con el mismo id. A su vez se creará una tupla en la tabla nueva UserRoles. Posteriormente, cuando se realice una petición al endpoint de asignar el usuario como HomeOwner, simplemente se creará una tupla de UserRoles, no creando datos en la tabla de HomeOwner ya que el usuario es un Admin que se comporta como HomeOwner.

Agrupar dispositivos por cuarto

La incluimos como decisión de diseño porque cuando encaramos el requerimiento, este no había sido comentado en el foro, por más de haberlo sido posteriormente

Justo coincidió que lo explicado por aquel medio fue lo mismo que se había resuelto por nuestra cuenta. Eso es, en el momento de asociar dispositivos a un hogar, estos se asocian inicialmente a ningún cuarto del mismo, independientemente se presenta un endpoint para crear cuartos en el hogar. Finalmente, es con otro endpoint, que se asignan los dispositivos previamente asociados a cuartos del mismo hogar, excluyendo a dispositivos que ya se presenten en otros cuartos. En este tercer endpoint, los dispositivos se van asignando uno a uno.

Eliminado de usuarios con múltiples roles

Los usuarios con rol de **Administrador** pueden ser eliminados del sistema, exceptuando aquellos que también sean **Dueños de Hogar**. Si se intenta eliminar un **Administrador** con este rol adicional, se le mostrará al usuario un mensaje de error.

Acciones de los Home Owners

Un dueño de hogar puede regular y administrar las acciones pertinentes de los miembros del respectivo hogar, asignándoles o *quitándoles* permisos (listar dispositivos, agregar dispositivos, hacer un usuario notificable dentro de un hogar). El dueño de hogar es

la autoridad máxima en el hogar, es decir, es capaz de realizar todas las acciones posibles en él.

Uso del modelo validador

Imaginemos el siguiente escenario. Se crea una compañía, inicialmente no seteando ningún modelo validador, luego se crea un dispositivo con dicha compañía, y luego posteriormente se le asigna un validador a la compañía. No se usará ese validador para el dispositivo previamente creado, porque el seteo del modelo validador ocurrió posteriormente. Este escenario también enfatiza el hecho de que, el modelo validador no es un campo obligatorio en la creación, el mismo puede ser asignado en otro momento.

Excepciones en repositorios personalizados

Los repositorios personalizados incluyen la lógica de lanzar una excepción en el caso de que la entidad a capturar sea nula. [Ver imagen en anexo](#)

Funcionamiento de la importación de dispositivos

Cuando está seteado el validador se empiezan a importar los dispositivos desde el comienzo hasta encontrar uno que no cumpla con el modelo validador. Es decir, cuando se encuentra un dispositivo que no cumpla, se deja de importar, quedando onbiamente importados los antecesores a ese dispositivo

En el caso de que no se encuentre especificado el modelo validador, se importan todos los dispositivos presentes

Taxonomía y convenciones de nombre en clases y paquetes

Por el lado de la taxonomía, para facilitar la comprensión, se presentana lo largo del proyecto, carpetas como Controllers, Models, Entities, Filters. En cuanto al nombrado de paquetes y clases, se sigue el estándar pascal case. A su vez, en la perspectiva de las interfaces de BusinessLogic, se utiliza un nombrado intuitivo. Imaginemos el ejemplo de estar situados en una interfaz llamada IRoomService. La dicha interfaz presenta dos métodos, uno para agregar y otro para listar. Para ser más prolijos, en vez de llamar a los métodos AddRoom y GetAllRooms, se llaman a los métodos Add y GetAll.









Descripción del manejo de excepciones

A lo largo de las clases, se implementaron bloques try catch para evitar comportamientos surgidos no deseados. Es en la sección de catch dónde llanzamos excepciones en caso de que se llegue hasta allí. Podríamos haber simplemente lanzado

excepciones genéricas, cada una con un mensaje descriptivo, por supuesto, pero no hicimos eso. Fuimos más allá. Desarrollamos clases específicas para cada tipo de excepción que se nos ocurre que se pudiera lanzar, de esa forma encapsulamos los tipos de error. Asimismo, también implementamos un filtro de excepciones, ocupado de atrapar las excepciones lanzadas por el sistema para transformarlas en mensajes y códigos de status HTTP más ilustrativos y convenientes.

Desde la perspectiva de desarrolladores, esto favorece a que, en el momento de estar ejecutando la aplicación, en el caso de ocurrir alguno de estos casos, se agilizaría mucho más el tiempo de detección del error.

Anexo

✓  Total	91%	297/3227
>  Homify.Exceptions	100%	0/21
>  Homify.Utility	100%	0/71
>  JSONImporter	100%	0/44
>  Homify.Importer.Abstractions	100%	0/44
>  Homify.DataAccess	94%	14/221
>  Homify.WebApi	91%	105/1112
>  Homify.BusinessLogic	90%	178/1714

Informe de cobertura

Gracias a la metodología TDD, se logró una cobertura de código muy elevada, lo que implica que la mayoría de las rutas y funcionalidades de la API han sido correctamente testeadas. Si bien una alta cobertura por sí sola no garantiza la calidad del software, sí es una buena señal de que se han validado muchos comportamientos y rutas clave. En nuestro proyecto, se superó el 90% de cobertura en todas las secciones de la aplicación, lo que es un excelente logro. Este resultado se debe al enfoque riguroso que exige TDD, donde cada nueva funcionalidad es acompañada de pruebas en el momento de su desarrollo.

Nuevamente resaltar que la cobertura no habla por sí sola de la calidad o lo a prueba de errores que se encuentra el proyecto, ya que los mismos no se pueden descartar totalmente, pero si es un indicio que la inmensa gran mayoría de casos que podrían ocurrir están bajo control.

Aplicar esta metodología trajo considerables beneficios al desarrollo del backend, ya que podíamos estar seguros que para algunos casos controlados, las nuevas funcionalidades que iban creciendo poco a poco iban respondiendo de la manera adecuada, lo cual nos ayudó a construir una solución robusta paso a paso, agregando complejidad a medida que se comprobaba el correcto funcionamiento previo.

Descripción de pruebas automatizadas

Para validar que las funcionalidades del sistema tengan un correcto funcionamiento, se han realizado pruebas unitarias. Estas fueron hechas con las tecnologías MSTest y Moq. Mediante a estas pruebas unitarias, es que se puede identificar fácilmente los errores y capturarlos de manera sencilla.

Análisis de las Heurísticas de Nielsen

1. Visibilidad del estado del sistema:

El sistema informa constantemente al usuario lo que está sucediendo, otorgándole retroalimentación en informando en los casos de error.



Formulario de registro con los siguientes campos:

- Email: mail@mail.com
- Password: ...
- Name: 123
- LastName: 123
- Profile Picture: 123

Mensaje de error (rojo): La contraseña debe tener al menos 6 caracteres.

Ejemplo: Error de la contraseña al registrarse en el sistema

2. Relación entre el sistema y el mundo real:

El sistema usa lenguaje y términos del mundo real, proporcionándole al usuario una experiencia cómoda e intuitiva

3. Control y libertad del usuario:

Los usuarios tienen el control de las acciones que realicen en el sistema. Se permite de manera sencilla que corrijan errores cometidos al informar de estos mediante un cartel en rojo

4. Consistencia y estándares:

Se sigue un estándar, por ejemplo, para el color de los mensajes de éxito con verde y los de error con rojo. También cada formulario sigue en grandes aspectos el mismo estilado

5. Prevención de errores:

Se previenen los errores al nuevamente mostrar error de ellos y no proceder con la ejecución del proceso dado, por ejemplo, cuando un formulario se rellena incorrectamente, al momento de enviarlo, este comunica su invalidez y rechaza el envío

6. Reconocimiento antes que recuerdo:

Los usuarios no deben recordar ningún funcionamiento. La interfaz es intuitiva y el flujo de acciones por parte del usuario ocurre de manera natural

7. Flexibilidad y eficiencia de uso:

La interfaz no provee atajos para aquellos usuarios más avanzados

8. Estética y diseño minimalista:

La interfaz presenta un diseño estético y minimalista debido a un estilado agradable

por parte del usuario y no presenta elementos que lo distraigan, deteriorando su experiencia, salvo los carteles de éxito u error que buscan exactamente esa respuesta en los usuarios

9. **Ayudar a los usuarios a reconocer, diagnosticar y recuperarse de errores:**

Los mensajes de error son claros, descriptivos, orientando a la corrección del problema

10. **Ayuda y documentación:**

En caso de que usuarios presenten un error más allá de su alcance o un desconocimiento del manejo de la aplicación, no se presenta con una guía de ayuda o similar, aunque, nuevamente mencionando, se espera a que el mismo no lo vea necesario por el fácil y sencillo manejo de la aplicación.

1. Entidad HomifyDateTime

```
public static class HomifyDateTime
{
    2 usages Guillermo Dotti
    public static string Parse(string date)
    {
        try
        {
            var isParsed = DateTimeOffset.TryParseExact(
                input: date,
                format: "dd/MM/yyyy",
                CultureInfo.InvariantCulture,
                DateTimeStyles.None,
                out DateTimeOffset dateParsed);

            if (!isParsed)
            {
                throw new ArgumentException("Invalid date format");
            }

            return dateParsed.ToString(format: "dd/MM/yyyy");
        }
        catch (ArgumentException ex)
        {
            return ex.Message;
        }
    }

    6 usages Guillermo Dotti
    public static string GetActualDate()
    {
        DateTimeOffset actual = DateTimeOffset.Now;
        var fecha:string = Parse(date: actual.ToString(format: "dd/MM/yyyy"));
        return fecha;
    }
}
```

2. Entorno de ejecución



The screenshot shows a code editor with a file named `launchSettings.json`. The file contains a JSON configuration for a development profile. The configuration is as follows:

```
{
  "profiles": {
    "Homify.WebApi.Development": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5003",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

3. Métodos de rechazo de la clase Helpers

```
public static class Helpers
{
    public static string GetUserFullName(string name, string lastName)
    {
        return $"{name} {lastName}";
    }

    public static void ValidateRequest(object? obj)
    {
        if (obj == null)
        {
            throw new NullRequestException("Request cannot be null");
        }
    }

    public static void ValidateNotFound(string? actor, object? obj)
    {
        if (obj == null)
        {
            throw new NotFoundException($"{actor} not found");
        }
    }

    public static void ValidateArgsNull(string? actor, object? obj)
    {
        if (obj == null)
        {
            throw new ArgsNullException($"{actor} cannot be null");
        }
    }
}
```

```

public static int ValidatePaginationLimit(string? limit)
{
    var pageSize = 10;

    if (!string.IsNullOrEmpty(limit) && int.TryParse(limit, out var
parsedLimit))
    {
        pageSize = parsedLimit > 0 ? parsedLimit : pageSize;
    }

    return pageSize;
}

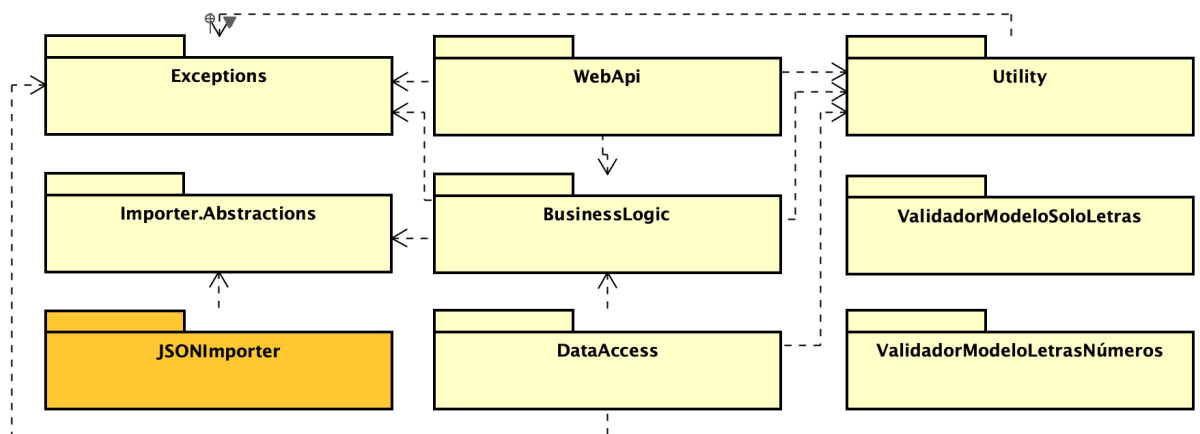
public static int ValidatePaginatioOffset(string? offset)
{
    var pageOffset = 0;

    if (!string.IsNullOrEmpty(offset) && int.TryParse(offset, out
var parsedOffset))
    {
        pageOffset = parsedOffset >= 0 ? parsedOffset : pageOffset;
    }

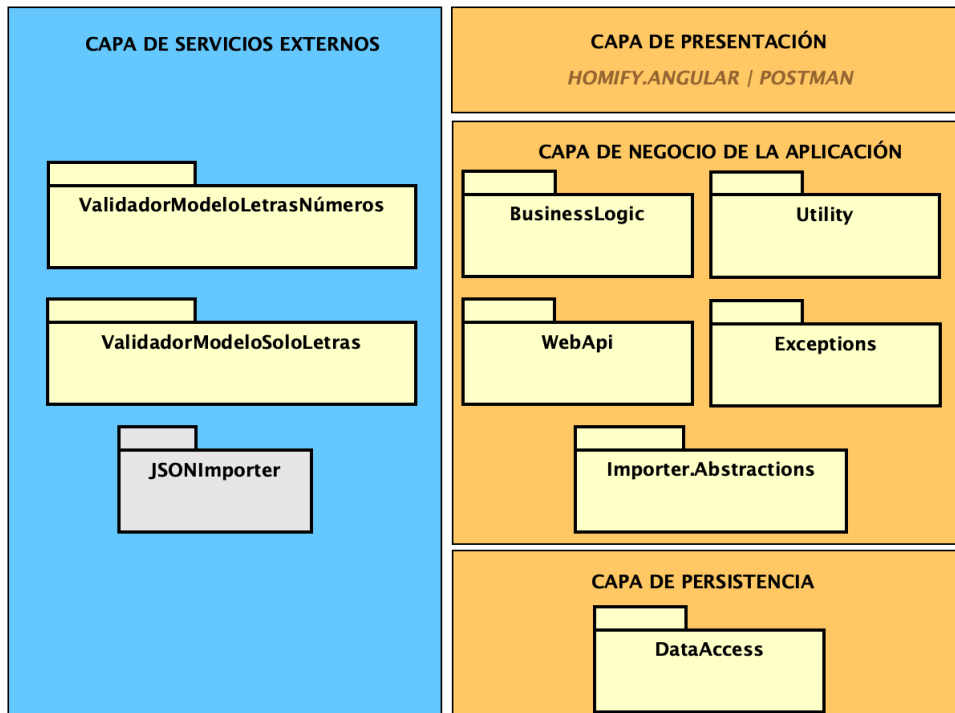
    return pageOffset;
}
}

```

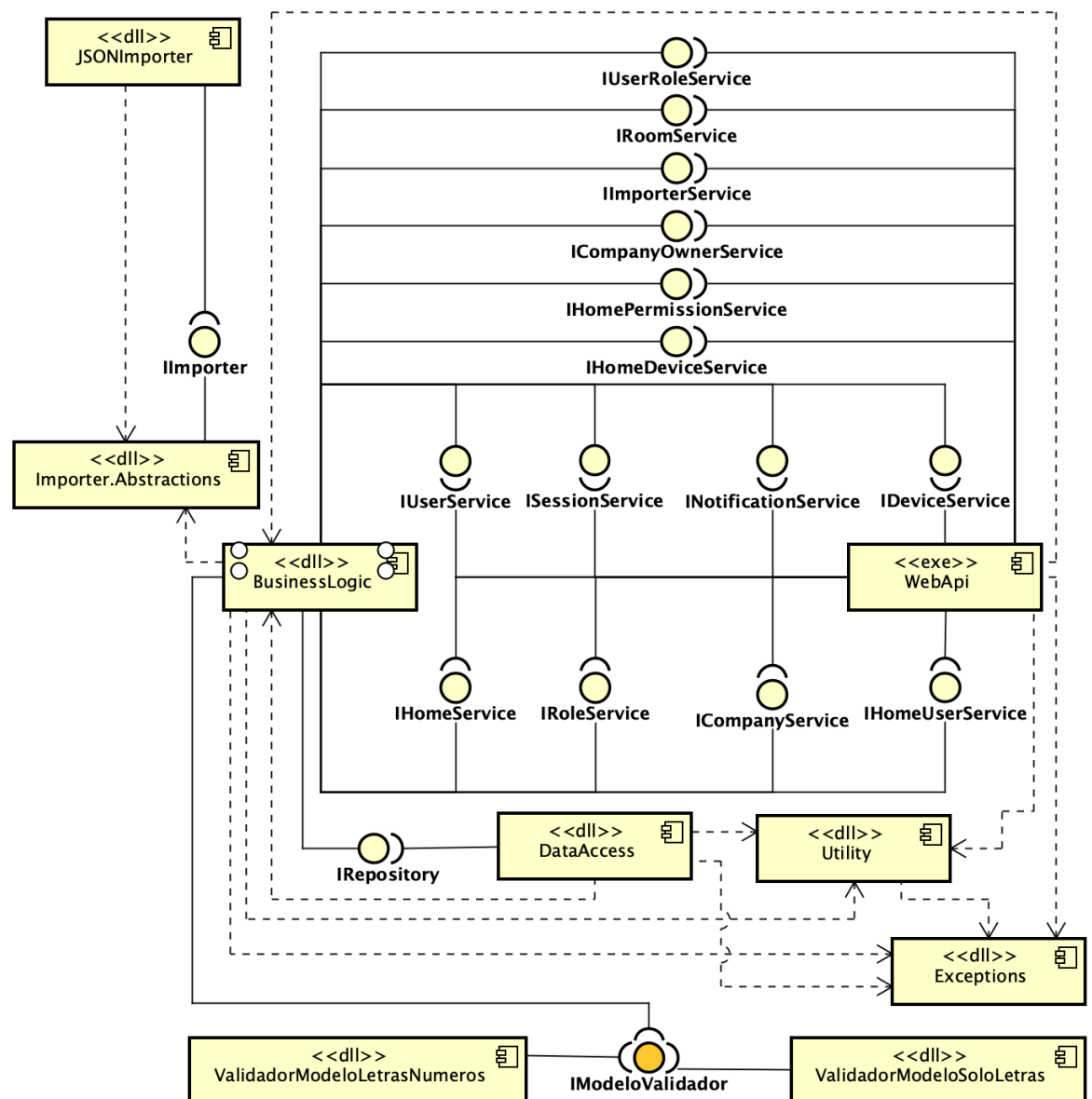
4. Diagrama general de paquetes



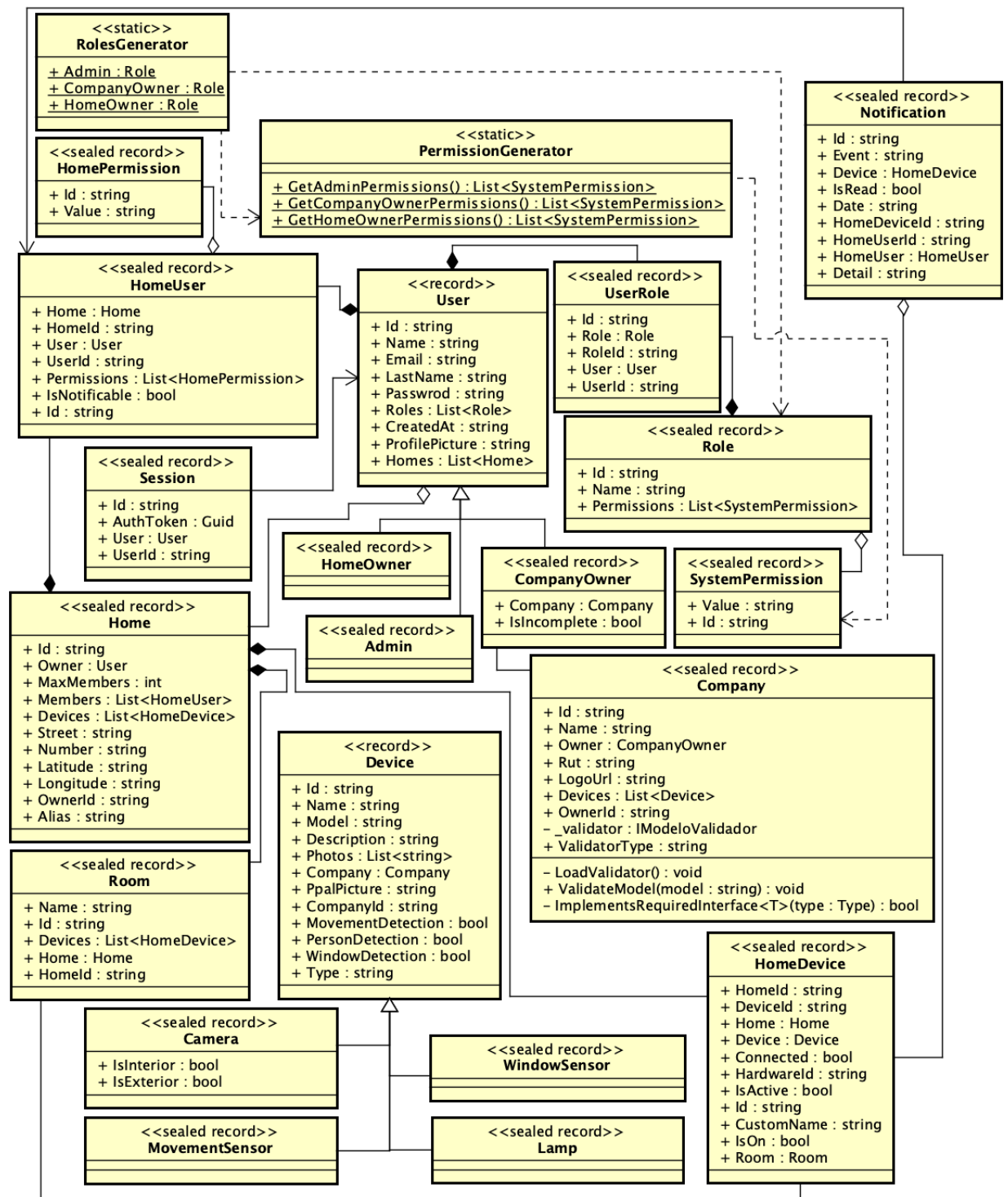
5. Diagrama de paquetes por layers



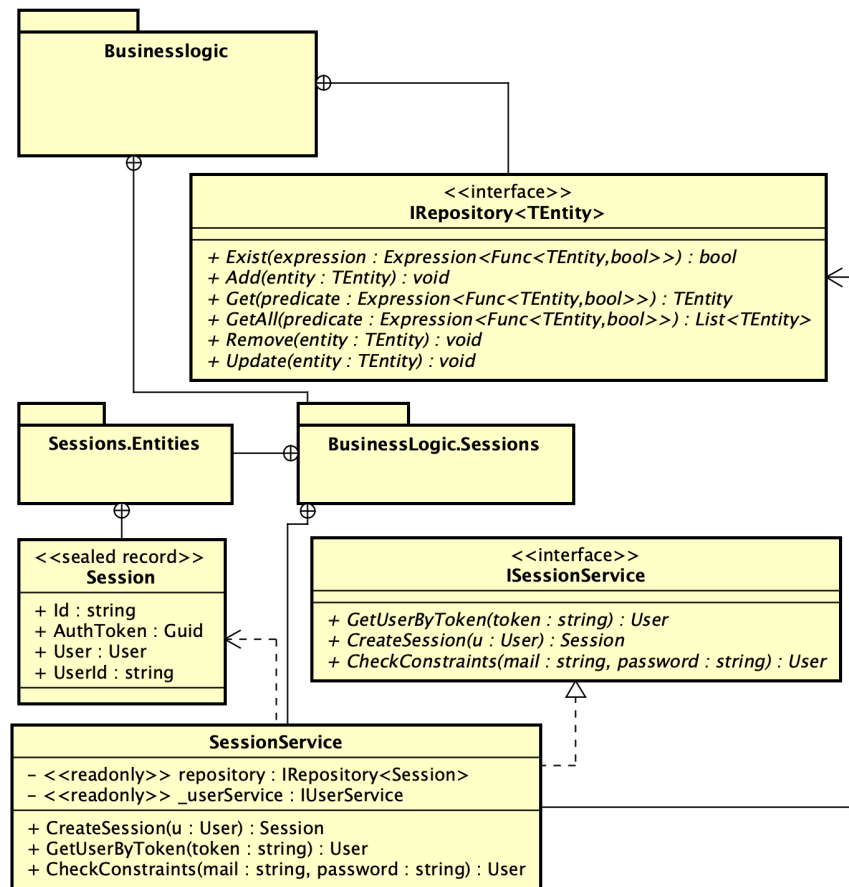
6. Diagrama de componentes



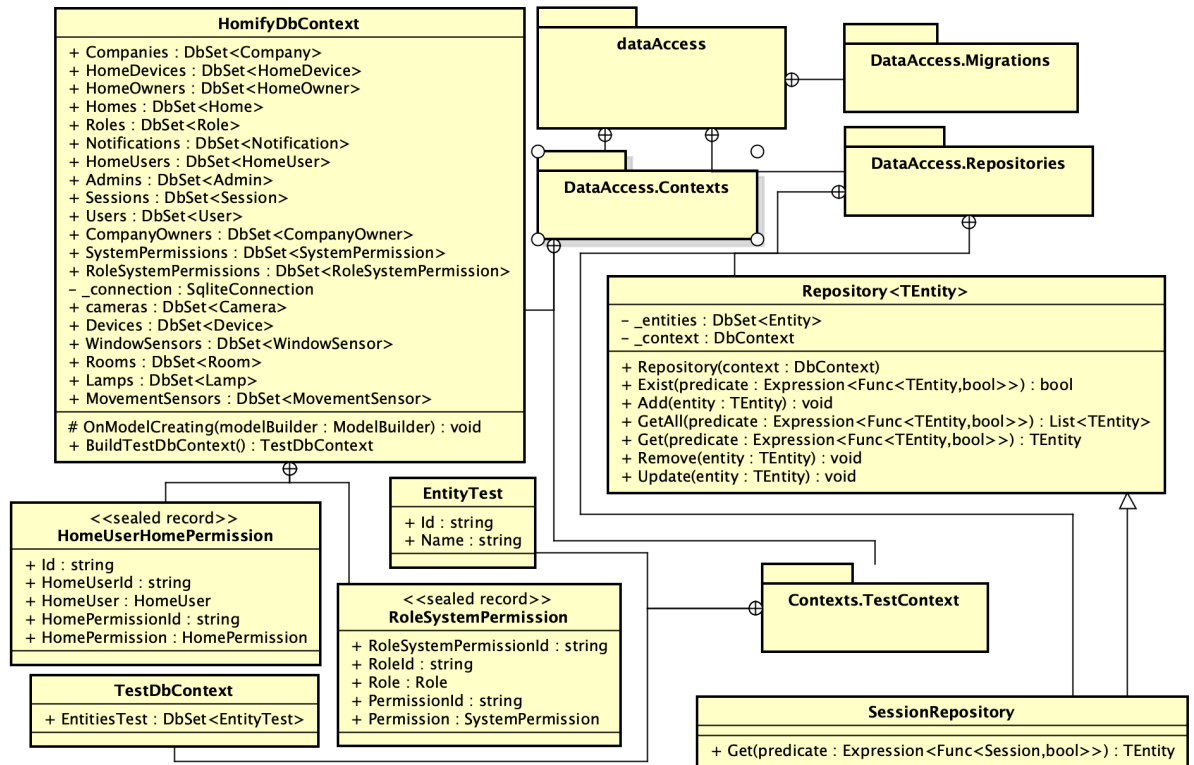
7. Diagrama completo de clases



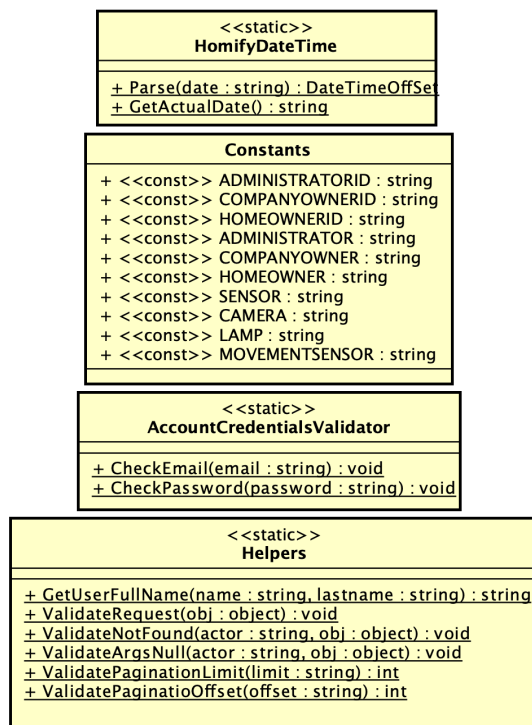
8. Diagrama de clases de BusinessLogic tomando como ejemplo Session



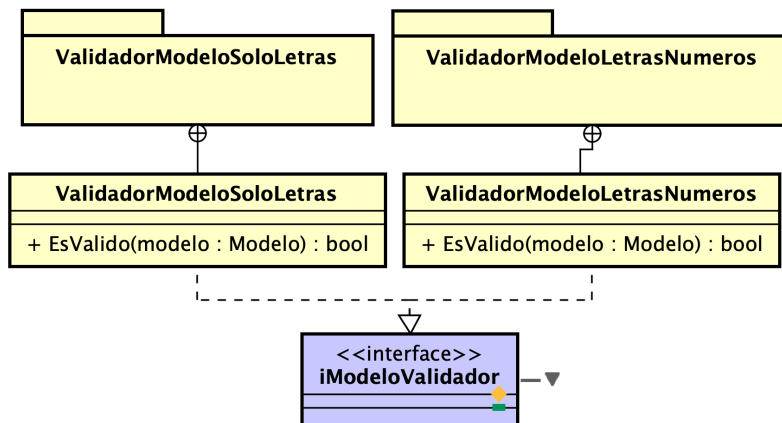
11. Diagrama de clases del paquete DataAccess



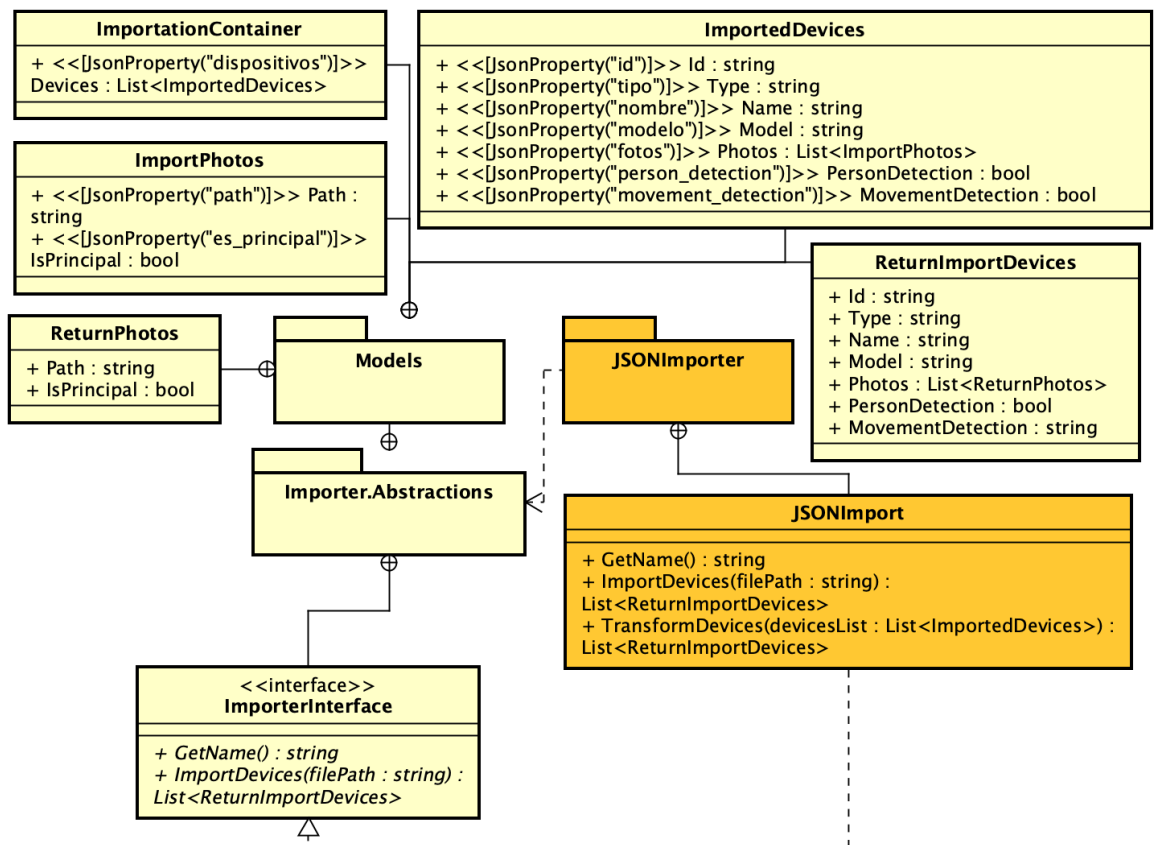
12. Diagrama de clases del paquete Utility



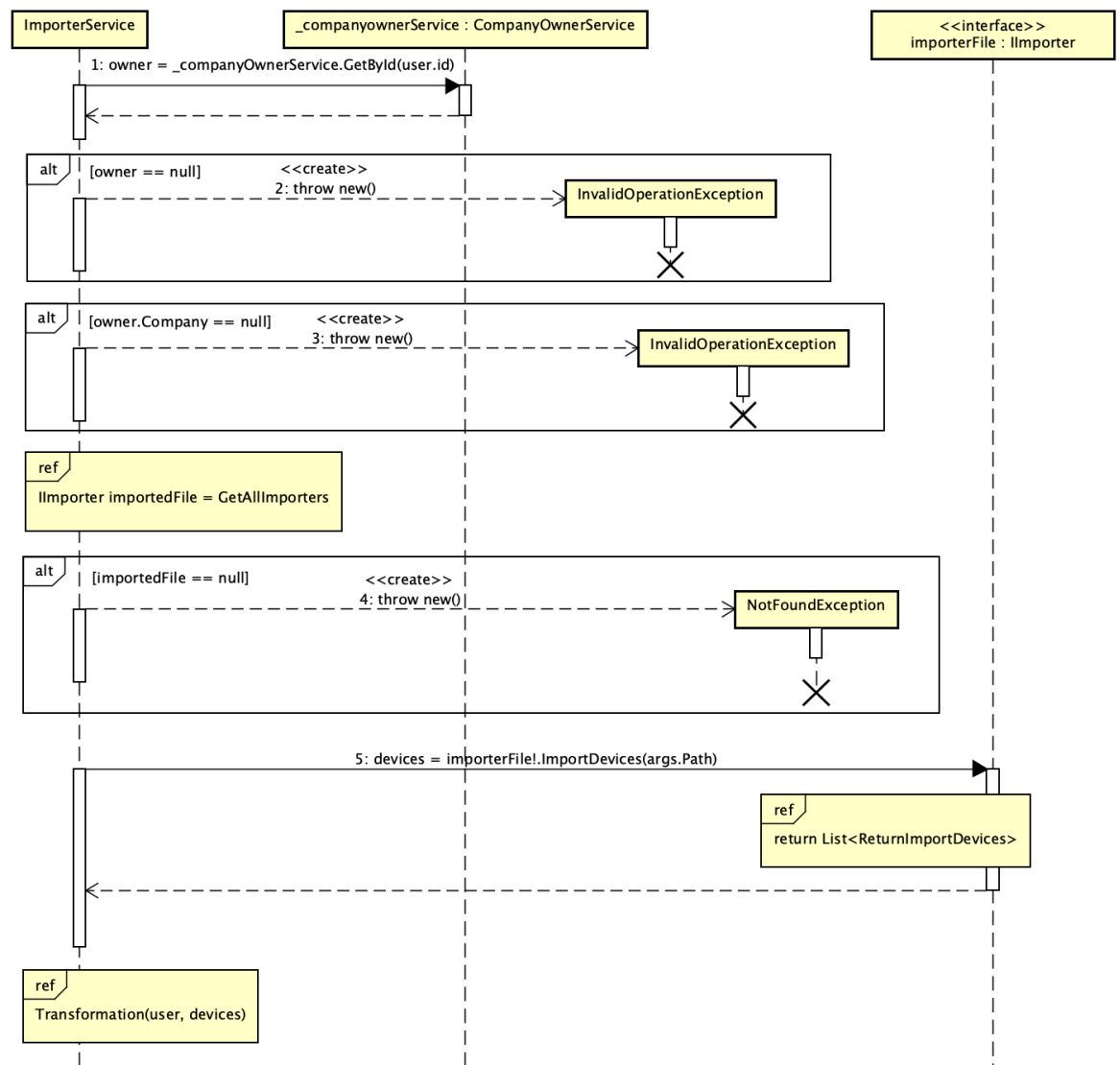
13. Diagrama de clases de los paquetes validadores



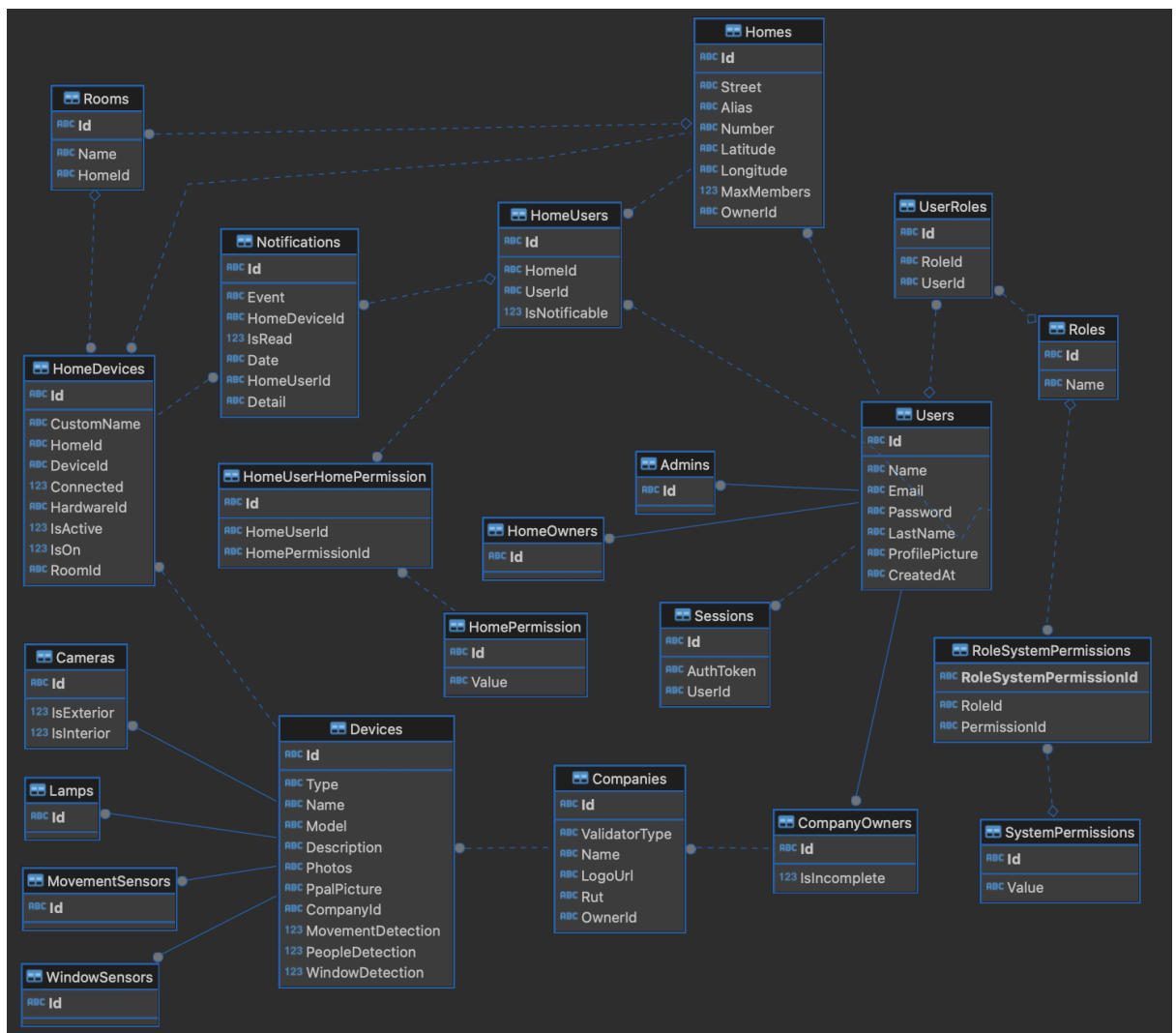
14. Diagrama de clases de los paquetes JSONImporter y Importer.Abstractions



15. Diagrama de Secuencia para la importación de dispositivos



16. Modelo de tablas de la estructura de la base de datos



17. Lógica de validación en excepciones

```
public override Company? Get(Expression<Func<Company, bool>> predicate)
{
    var query:IQueryable<Company> =
        _entities.Include( navigationPropertyPath: u:Company => u.Owner) // Include
        .Include( navigationPropertyPath: u:Company => u.Devices) // Include
        .Where(predicate);

    var user:Company? = query.FirstOrDefault();

    if (user == null)
    {
        throw new NotFoundException($"User not found");
    }

    return user;
}
```

Cambios respecto a la especificación de la API

Recurso	Acción	URI	Headers	Body	Response	Status Code	Justificación
devices	POST	/devices/window-sensors	Authorization: token	{ "name": string, "model": string, "description": string, "photos": string[], }	{ "id": string, }	201 Created 400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	Se renombró el endpoint (anteriormente /devices/sensors) dado que se agregó un nuevo tipo de sensor, para evitar ambigüedad.
devices	POST	/devices/movement-sensors	Authorization: token	{ "name": string, "model": string, "description": string, "photos": string[], "ppalPicture": string }	{ "id": string }	201 Created 400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	Un dueño de empresa puede crear sensores de movimiento.
devices	POST	/devices/lamps	Authorization: token		{ "id": string }	201 Created 400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	Un dueño de empresa puede crear lámparas inteligentes.
homedevices	PUT	/home-devices/{har	Authorization:	-	{ "id": string, }	200 Ok	Permite marcar

		dwareId}/deactivate	token		"isActive": boolean}	400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	un dispositivo como inactivo. Mientras un dispositivo de un hogar esté inactivo no podrá generar notificaciones.
homedevices	PUT	/home-devices/{hardwareId}/lampOn	Authorization: token	-	{"id": string, "isOn": boolean}	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Cuando una lámpara está activa, puede encenderse, lo que genera una notificación para los usuarios configurados del hogar y cambia el estado de la lámpara.
homedevices	PUT	/home-devices/{hardwareId}/lampOff	Authorization: token	-	{"id": string, "isOn": boolean}	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Cuando una lámpara está activa, puede apagarse, lo que genera una notificación para los usuarios configurados del hogar y cambia el

							estado de la lámpara.
homedevices	PUT	/home-devices/{hardwareId}/windowOpen	Authorization: token	-	{“id”: string, “isOn”: boolean}	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Cuando un sensor de ventana está activo, puede detectar cuando una ventana se abre, lo que genera una notificación para los usuarios configurados del hogar y cambia el estado del sensor.
homedevices	PUT	/home-devices/{hardwareId}/windowClose	Authorization: token	-	{“id”: string, “isOn”: boolean}	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Cuando un sensor de ventana está activo, puede detectar cuando una ventana se cierra, lo que genera una notificación para los usuarios configurados

							del hogar y cambia el estado del sensor.
homes	GET	/homes/by-owner	Authorization: token	-	{ "id": string, "street": string, "alias": string, "number": string, "latitude": string, "longitude": string, "maxMembers": number }	200 Ok 401 Unauthorized 403 Forbidden 500 Internal Server Error	Obtener todos los hogares donde un determinado usuario es dueño. Necesario para poder seleccionar hogares desde el frontend y poder realizar las operaciones pertinentes con ellos.
homes	GET	/homes/by-member	Authorization: token	-	{ "id": string, "street": string, "alias": string, "number": string, "latitude": string, "longitude": string, "maxMembers": number }	200 Ok 401 Unauthorized 403 Forbidden 500 Internal Server Error	Obtener todos los hogares donde un determinado usuario es miembro. Necesario para poder seleccionar hogares desde el frontend y poder realizar

					}}		las operaciones pertinentes con ellos.
homes	GET	/homes/{homeId}/devices?room={roomName}	Authorization: token	-	{ { "name": string, "customName": string, "model": string, "mainPhoto": string, "isConnected": boolean, "deviceId": string, "hardwareId": string, "id": string, "isActive": string, "room": string } }	200 Ok 401 Unauthorized 404 Not Found 403 Forbidden 500 Internal Server Error	El dueño del hogar o cualquier miembro con el permiso adecuado puede ver todos los dispositivos instalados en su hogar. Se agregó filtrado por cuarto y brinda más información sobre los dispositivos al usuario.
homes	PUT	/homes/{homeId}/rename	Authorization: token	{ { "alias": string } }	{ { "id": string, "alias": string } }	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Los usuarios ahora pueden asignar un alias a su hogar para identificarlo más fácilmente.
sessions	POST	/sessions	-	{	{"token": string,	201 Created	Anteriormente

				<pre> "email": string, "password": string } </pre>	<pre> "roles": string[], "name": string } </pre>	400 Bad Request 500 Internal Server Error	solo se retornaba al cliente el token, ahora también se le adicionan el nombre y roles del usuario para poder implementar Guards y mostrar al usuario su nombre.
rooms	POST	/rooms	Authorization: token	<pre> { "name": string, "homeld": string } </pre>	<pre> { "id": string } </pre>	201 Created 400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	Crear cuartos para un determinado hogar.
rooms	GET	/rooms/{homeld}	Authorization: token	-	<pre> [["name": string, "homeld": string]] </pre>	200 Ok 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Obtener todos los cuartos pertenecientes a un hogar. Necesario para que del lado del cliente se pueda seleccionar a qué cuarto

							asignar un dispositivo y realizar filtrado por cuarto.
rooms	PUT	/rooms/{roomId}/{homeDeviceId}	Authorization: token	-	{ "id": string, "name": string, "homeId": string }	200 Ok 400 Bad Request 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Asignar un dispositivo previamente asignado al hogar, a un cuarto específico dentro de este.
roles	PUT	/roles	Authorization: token	-	{ "user": string, "roles": string }	200 Ok 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Admins y dueños de empresa pueden usar sus cuentas para operar como dueños de hogar si lo desean.
homeowners	PUT	/homeowners/profile	Authorization: token	{ "profilePicture": string }	{ "user": string, "profilePicture": string }	200 Ok 404 Not Found 401 Unauthorized 403 Forbidden 500 Internal Server Error	Actualizar la foto de perfil de un usuario. Especialmente útil cuando un usuario agrega un nuevo rol a su cuenta y desea agregar una foto de

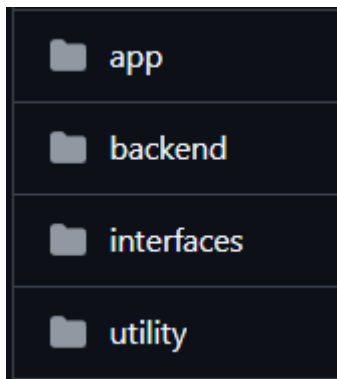
							perfil.
importers	POST	/importers	Authorization: token	{ "importerSelected": string, "filePath": string }	-	201 Created 400 Bad Request 401 Unauthorized 403 Forbidden 500 Internal Server Error	El sistema permite que dueños de empresa puedan importar nuevos dispositivos a partir de un archivo y un tipo de importador.
importers	GET	/importers	Authorization: token	-	string[]	200 Ok 401 Unauthorized 403 Forbidden 500 Internal Server Error	Consultar por todos los importadores disponibles en el sistema.
importers	GET	/importers/validators	Authorization: token	-	string[]	200 Ok 401 Unauthorized 403 Forbidden 500 Internal Server Error	Consultar por todos los modelos validadores disponibles en el sistema.
companies	PUT	/companies/validators	Authorization: token	{ "model": string }	string[]	200 Ok 400 Bad Request 401 Unauthorized 403 Forbidden	Asignarle a una empresa un modelo validador determinado para que

						500 Internal Server Error	ejecute validaciones a la hora de crear nuevos dispositivos.
--	--	--	--	--	--	------------------------------	--

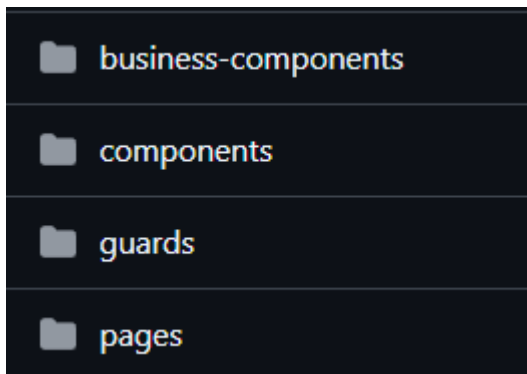
Interfaz de Usuario

La interfaz de usuario se compone de una Single Page Application realizada en su totalidad con Angular. Se utilizaron los lenguajes Typescript, HTML y CSS para su desarrollo dentro del framework, al igual que las librerías **Ng-Zorro** (<https://ng.ant.design/docs/introduce/en>) y **Angular Material** (<https://material.angular.io/>) para una mejor satisfacción de los usuarios y agilizar los tiempos de desarrollo, estas se utilizaron respectivamente para crear el sidebar de la app (mediante el cual se pueden acceder a todas las rutas) y permitir el uso de modals/pop-ups para mejorar la interacción de los clientes con ciertas funcionalidades del sistema.

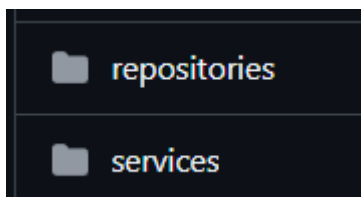
En cuanto a la arquitectura, se presenta de la siguiente manera:



El directorio app se compone de:



El directorio backend se compone de:



business-components: Contiene componentes standalone que encapsulan una pequeña porción del negocio. Estos componentes son reutilizables en distintas partes de la aplicación y tienen lógica específica relacionada con el negocio.

components: Almacena componentes standalone base o genéricos que contienen únicamente el HTML esencial para la construcción de la aplicación. Estos componentes son los bloques fundamentales sobre los que se basa la misma.

guards: Contiene los guardias de ruta en Angular, que son clases especiales utilizadas para controlar el acceso a diferentes rutas de la aplicación. Los guards permiten proteger las rutas asegurando que ciertas condiciones se cumplan antes de que un usuario pueda acceder a una página.

pages: Contiene los componentes que representan las vistas completas o pantallas de la aplicación. Estos componentes son los responsables de representar páginas enteras, y son las rutas base de la aplicación, que a su vez, están encapsuladas dentro de módulos específicos. Esto se hace para que cada sección de la aplicación tenga su propio módulo, lo que facilita la gestión, la carga perezosa (lazy loading), y la reutilización de componentes y servicios.

backend: Contiene la lógica relacionada con la obtención y manipulación de datos.

- **services:** Aquí están los servicios que utilizan los componentes en *business-components*, ofrecen una capa de abstracción entre los componentes de negocio y los repositorios, para que ninguno tenga que verse obligado a conocer la implementación del otro para un correcto funcionamiento.
- **repositories:** Servicios responsables de la manipulación de datos. Aquí se maneja la comunicación con la API de Homify mediante consultas HTTP.

interfaces: Contiene estructuras de tipos que son reutilizadas constantemente en la aplicación, como por ejemplo la estructura de los errores mandados desde la API para poder mostrarle al usuario el mensaje de error en caso que la solicitud no se haya completado correctamente.

La idea general de esta organización es mantener una separación clara entre la lógica del negocio, la construcción visual, y la obtención de datos. Al igual que permitir el reúso de los componentes base como los de negocio, quienes pueden estar en distintas vistas de la aplicación si es necesario.

Conclusión general

Comenzando por el lado del backend, consideramos que hicimos un buen trabajo en líneas generales, quitando las complicaciones mencionadas al principio del informe, no hubo mayores dificultades a la hora de ir desarrollando el proyecto, dado que apoyados por una metodología como TDD, solucionando los problemas fuera del código en primera instancia antes de llevar las soluciones a cabo, y un intercambio constante con el cuerpo docente, notamos una agradable experiencia en el desarrollo que nos deja conformes con el trabajo realizado, aunque estamos de acuerdo en que es bastante mejorable a pesar de ser totalmente funcional.

La creación de los nuevos requerimientos no se sintió “antinatural” dado el buen diseño y extensibilidad de la etapa predecesora, por lo que en todo momento la sensación de “mejorar” el sistema se antepone al pensamiento de “tener que cambiar lo ya hecho”, por lo que fue una experiencia gratificante que nos puso a prueba con desafíos laboriosos.

Por el lado del frontend, fue un trabajo muy arduo que concluyó en la implementación de todas las funcionalidades necesarias, pero consideramos que en temas de estilado y responsividad es muy mejorable a futuro, sin embargo estos campos no fueron prioritarios al no ser considerados de tanta exigencia en el alcance de esta entrega.