Universidad ORT Uruguay

Facultad de Ingeniería en Sistemas

Diseño y Documentación, Obligatorio 2 - Diseño de Aplicaciones 2

Entregado como requisito del curso de la materia

Sebastian Escuder - 242739

Bruno Odella - 231665

Tutores: Francisco Bouza, Juan Irabedra y Santiago Tonarelli

2024

Link al repositorio de GitHub:https://github.com/IngSoft-DA2-2023-2/242739-231665.git

Índice

1. Descripción general del trabajo (qué hace la solución) y errores conocidos (bugs o funcionalidades no implementadas).

Nuestro proyecto cuenta con dos partes, el frontend y el backend. Sirve para gestionar edificios, ofreciendo todas las funcionalidades pedidas en la letra, pero tiene algunos detalles y puntos a aclarar. Los puntos que son negativos no fueron mejorados por falta de tiempo.

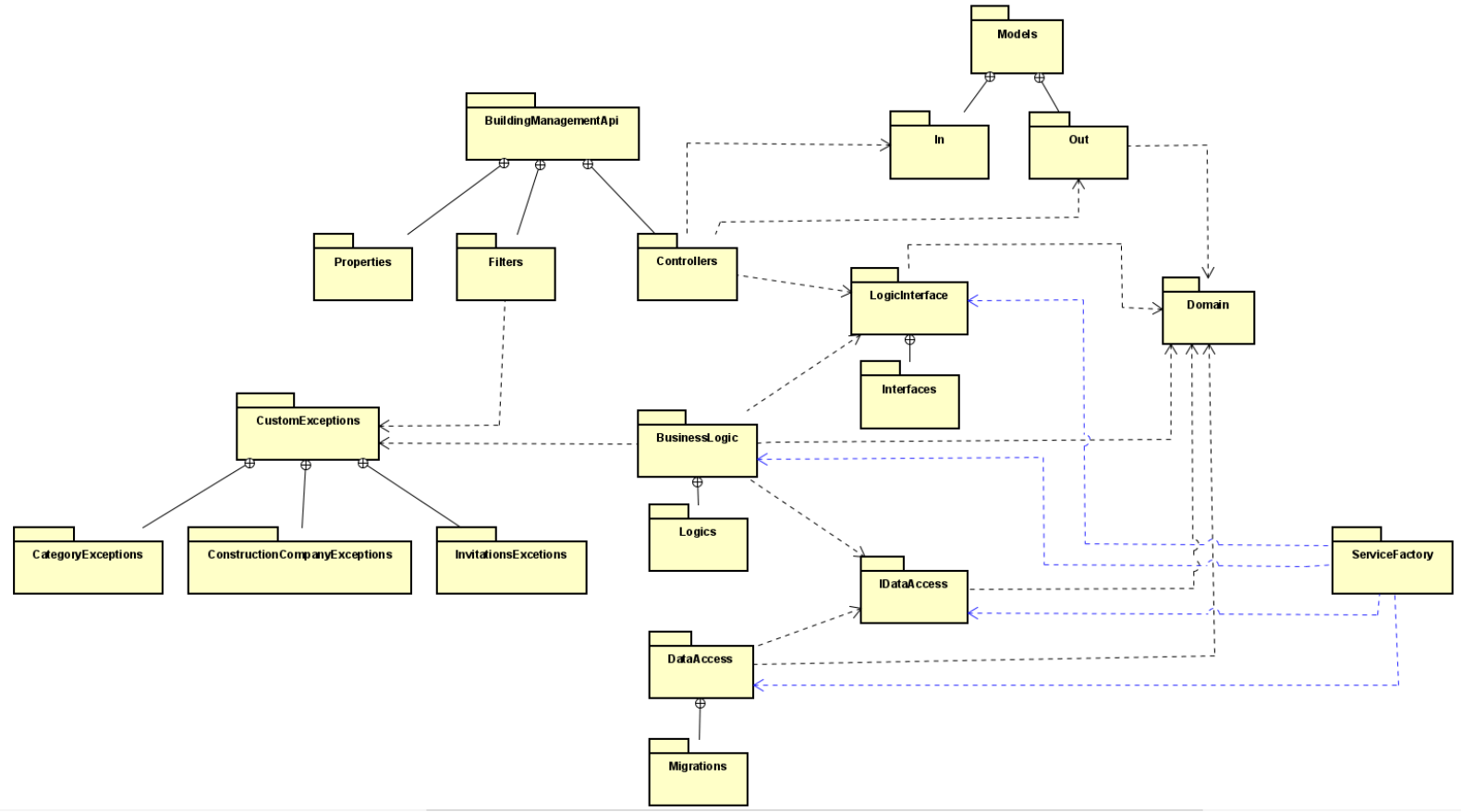
1. En primer lugar, se implementó un filtro de excepciones que filtra gran parte de las excepciones generadas del lado del backend, pero si hay alguna que no hemos implementado, devuelve un 500, unexpected Error. Lo mejor sería que devuelva lo que falló en todos los casos, pero hay casos aislados en los que esto no ocurre.
2. Existen detalles a la hora de crear, o sea hacer POST, de los distintos tipos de usuarios, y otras entidades como edificios. Se incluyeron las validaciones para que no haya repeticiones de los campos en los que no debería(por ejemplo mail, latitud y longitud de un edificio) pero no estamos chequeando del lado del backend, si el string que recibimos como mail no tiene un “@mail.com”. Tampoco estamos revisando si envía estos campos vacíos, lo que permite que si le damos con postman un usuario con los campos vacíos, se cree(solo una vez ya que el mail vacío quedará asignado).
3. En el frontend, se realizan validaciones de los inputs pero no mostramos el mensaje de error debido cuando el usuario las introduce mal. Por ejemplo, si ingreso un mail o contraseña que cuenta con menos de 4 caracteres, el frontend no me deja enviarlo, y no nos dirá la razón.
4. Los managers invitan a personas de mantenimiento, y para que estas personas lo puedan aceptar, le deben enviar el GUID de su invitación. Lo indicado de esta funcionalidad sería que el invitado solo tenga que poner su mail y su nueva contraseña, pero además tiene que dar el GUID.
5. Se podía haber hecho un polimorfismo entre los tipos de usuarios desde una clase padre Usuario, que hubiera facilitado mucho la lógica de las autorizaciones y autenticaciones. Esto fue debido a que había un campo el cual los usuario “Encargado” no poseían, que era el apellido, pero luego nos dimos cuenta que podíamos simplemente dejarlo sin asignar (aunque desperdiciaría espacio, ya que el campo Lastname quedaría como NULL). No se hizo este cambio, debido a que desde la primera entrega ya pusimos los usuarios de esta forma, y cambiarlo implicaría grandes cambios en el modelo de datos, en las pruebas y en las clases.
6. Los managers invitan a personas de mantenimiento y estas personas de mantenimiento serán compartidas por todos los managers del sistema. Aclaramos esto porque no se interpreta si estas personas solo son conocidas por el manager que los invita o por todos.
7. La nueva funcionalidad de poder inyectar el .dll, nos ofrece una interfaz (a nivel de back) a través de la cual el desarrollador puede implementar el método de importación que quiera. Este método es totalmente implementado por el desarrollador, lo cual le da libertad de poder hacerlo desde una API externa, una base de datos, o lo que sea y no solo un .JSON o algún otro formato de texto plano. Por esto mismo, debe estar hardcodeado en el .dll la ruta o archivo desde el cual se

va a importar, y hace que el .dll sea de un solo uso pero esto ofrece la ventaja mencionada anteriormente, de no atarnos a que solo se pueda enviar por ruta el archivo a importar. Lo único que debemos indicar para acceder al endpoint de inyectar el .dll, es la ruta donde se encuentra el mismo.

8. Los apartamentos no tienen el atributo de cantidad de habitaciones. Esta es una mejora potencial.

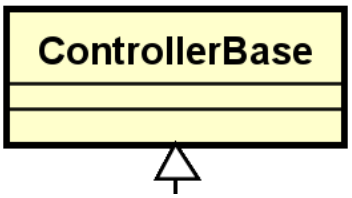
2. Diagrama general de paquetes (namespaces) mostrando los paquetes organizados por capas (layers) y sus dependencias. En caso de que haya paquetes anidados, se debe utilizar el conector de nesting, mostrando la jerarquía de dichos paquetes.

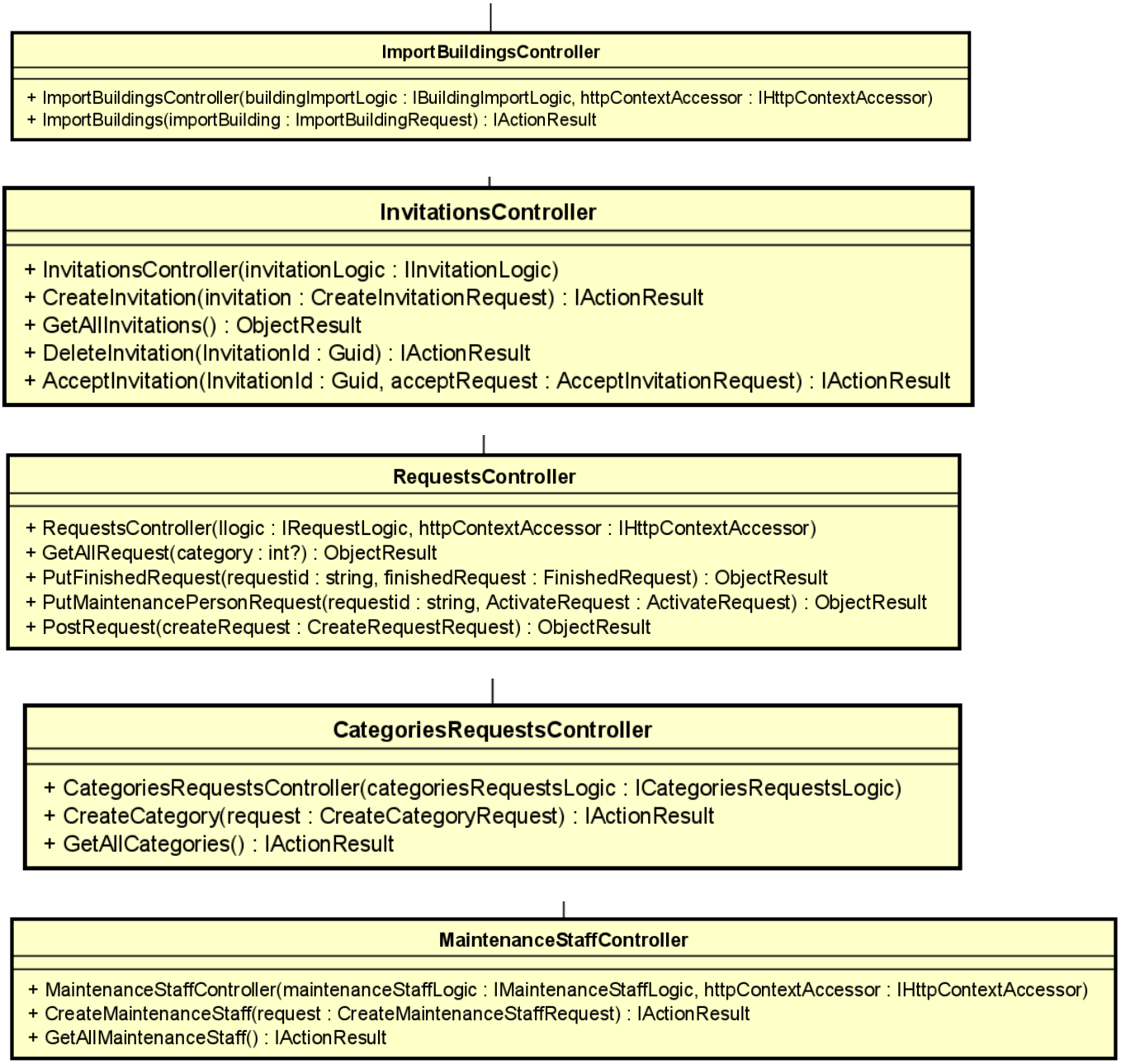
Cada paquete debe tener una breve descripción de responsabilidades y un diagrama de clases asociado.

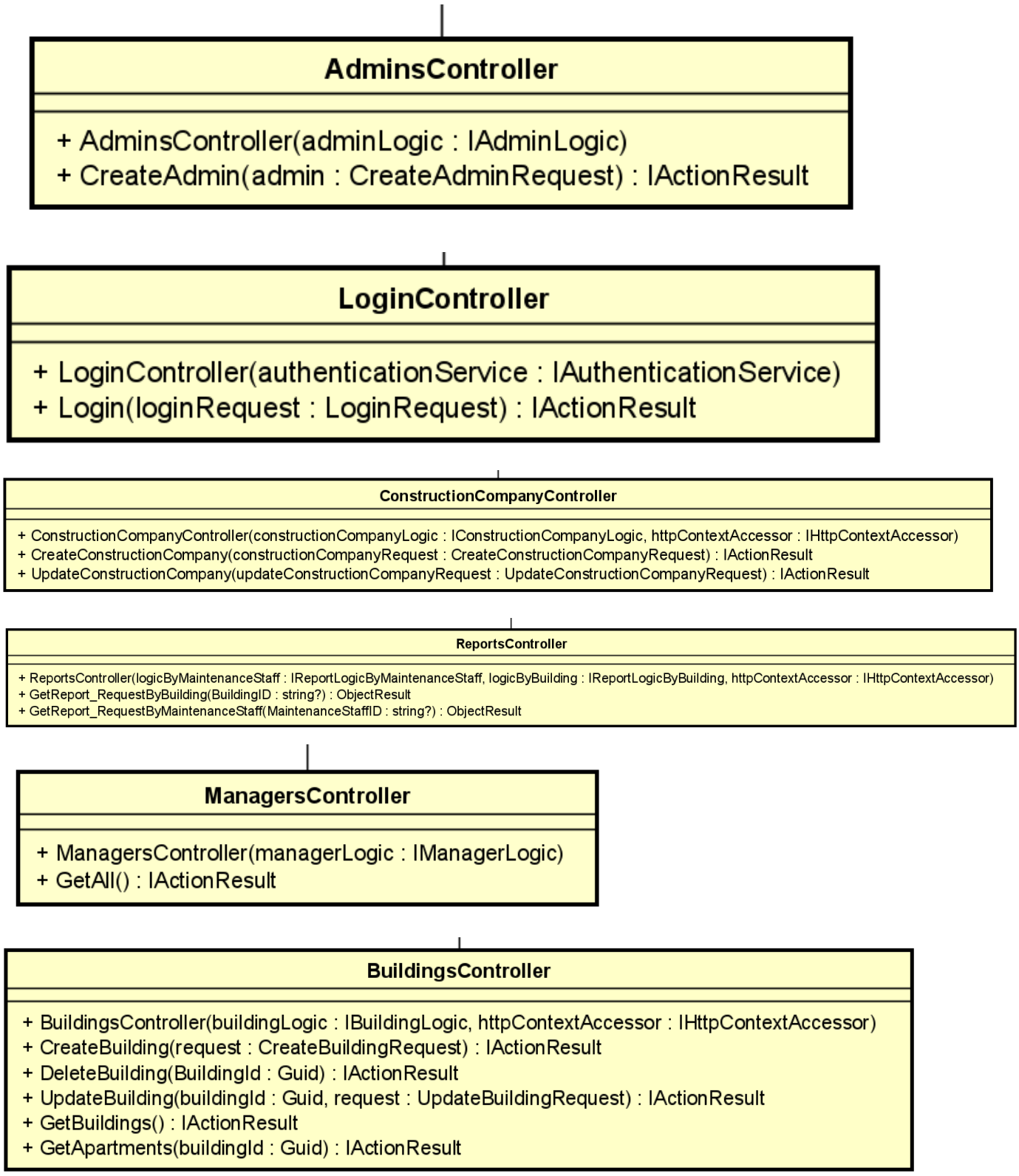


2.1 Diagrama de clases del paquete BuildingManagementApi.Controller:

En primer lugar, todas estas clases heredan de ControllerBase.
La línea que sale de todos ellos, hace referencia a que está heredando de este. Es responsable de recibir todas las llamadas a la API, y actuar como una fachada del backend. Una vez recibe las llamadas, las delega a la lógica para que esta realice lo que este endpoint de la API está destinado a hacer. Contiene una clase por cada controlador, donde tiene los métodos asociados con las URIs que los llaman.



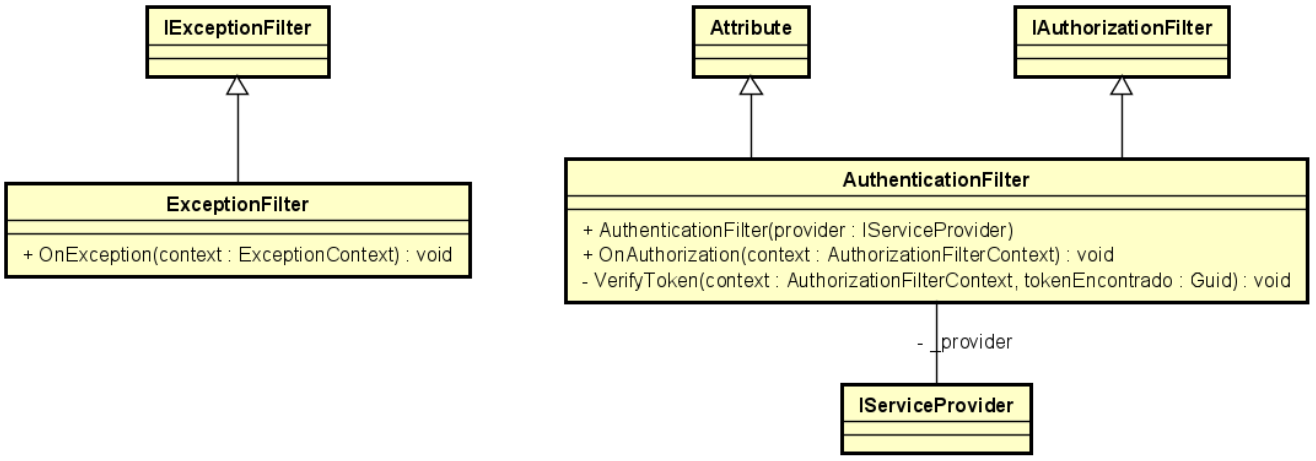




2.2 Diagrama de clases del paquete BuildingManagementApi.Filters:

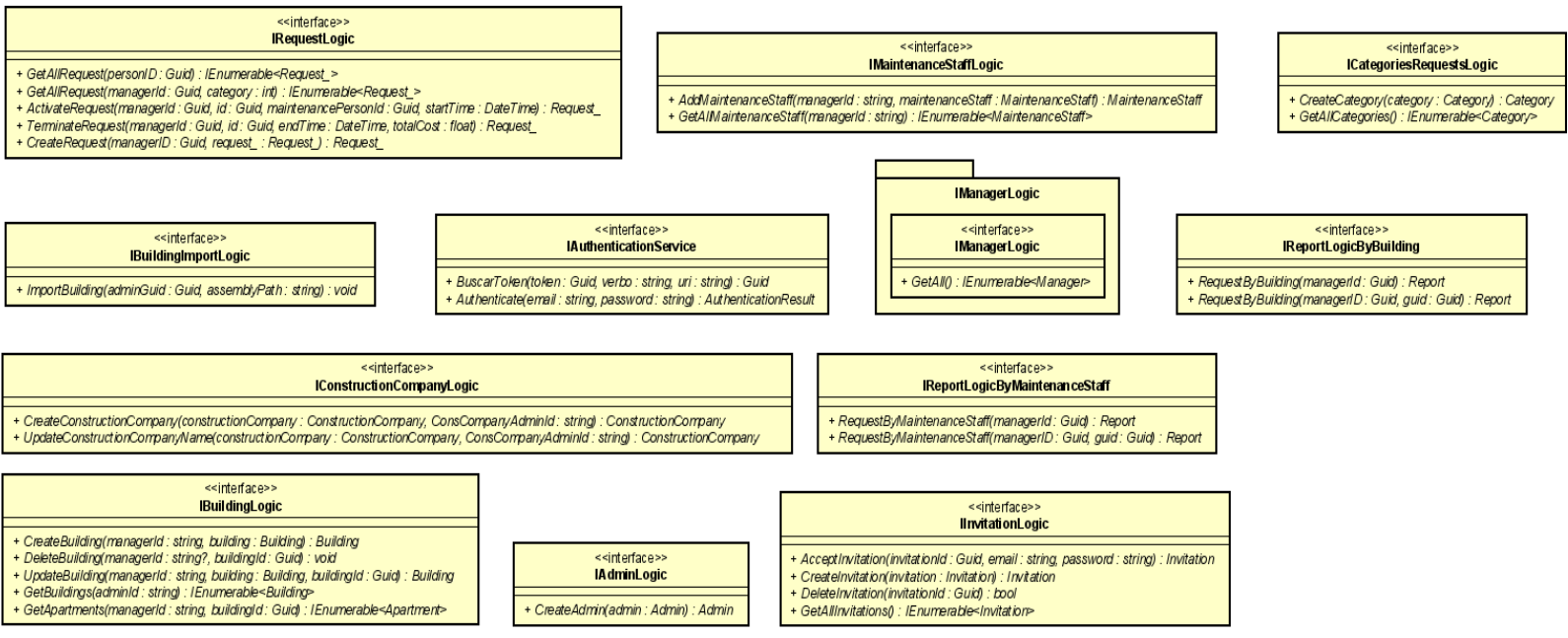
El paquete de filtros contiene dos clases principales: el filtro de autenticación y el filtro de excepciones.

El **filtro de autenticación** se encarga de verificar que el usuario esté autenticado correctamente utilizando sus credenciales. Además, este filtro distingue entre los diferentes tipos de usuarios, controlando el acceso a los endpoints específicos según los permisos correspondientes a cada tipo de usuario. Esto garantiza que solo los usuarios autorizados puedan acceder a determinadas funcionalidades de la API.
Por otro lado, el **filtro de excepciones** actúa como un intermediario que gestiona las excepciones generadas por la lógica de negocio invocada por los controladores. Este filtro captura cualquier excepción que ocurra y la transforma en una respuesta HTTP con un código de estado y un mensaje de error definidos por nosotros, los desarrolladores. Estas excepciones personalizadas se encuentran en el paquete CustomExceptions. Este mecanismo mejora la experiencia del usuario al proporcionar respuestas claras y consistentes sobre los errores, y asegura que la API maneje las excepciones de manera estructurada y controlada.



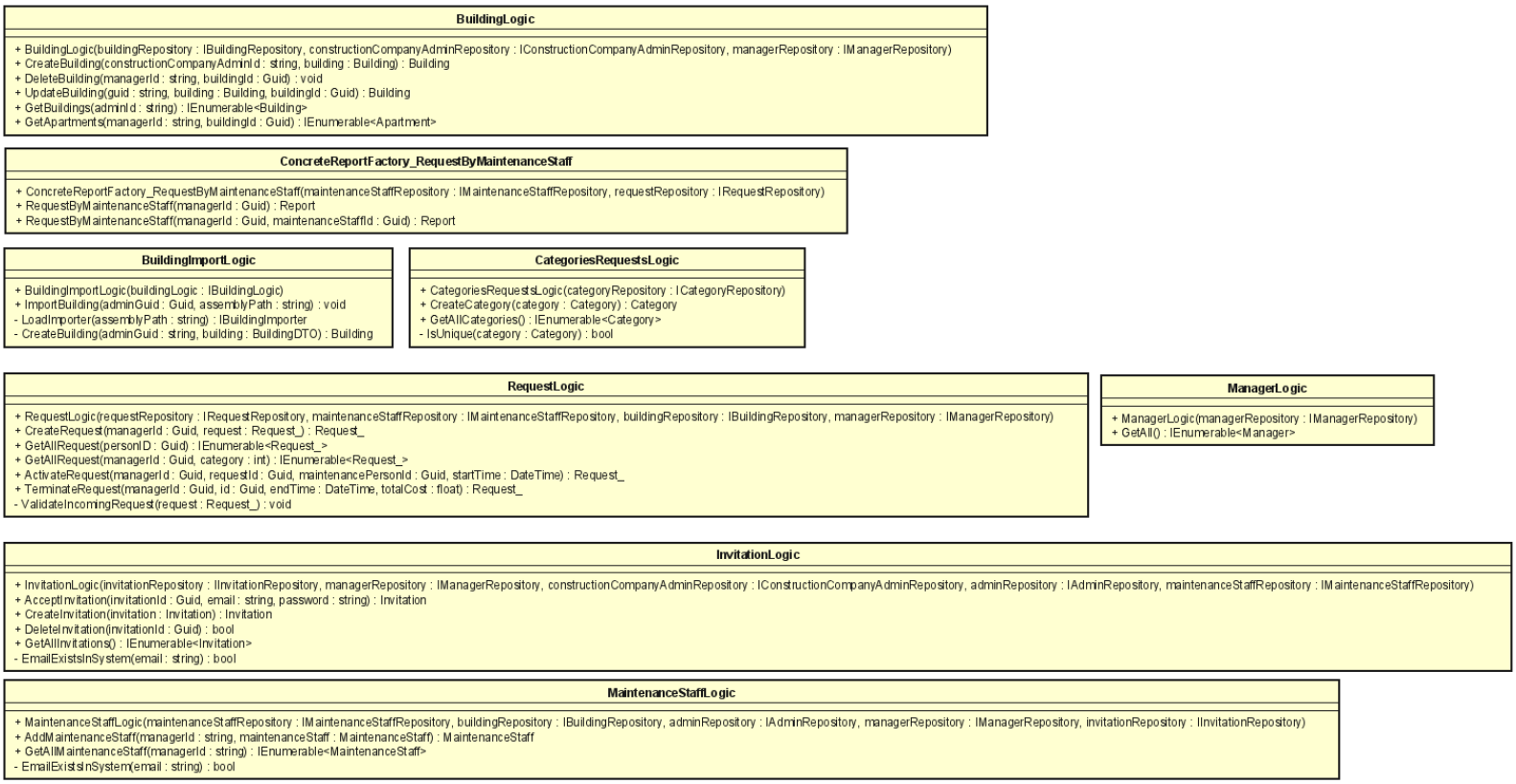
2.3 Diagrama de clases del paquete LogicInterface.Interfaces:

Este paquete contiene las interfaces de la capa de lógica de negocio (BusinessLogic) conocidas por los controladores (Controllers). Todas estas interfaces definen únicamente los métodos que serán invocados por los controladores, permitiendo así que la implementación concreta de la lógica de negocio permanezca oculta en tiempo de compilación.



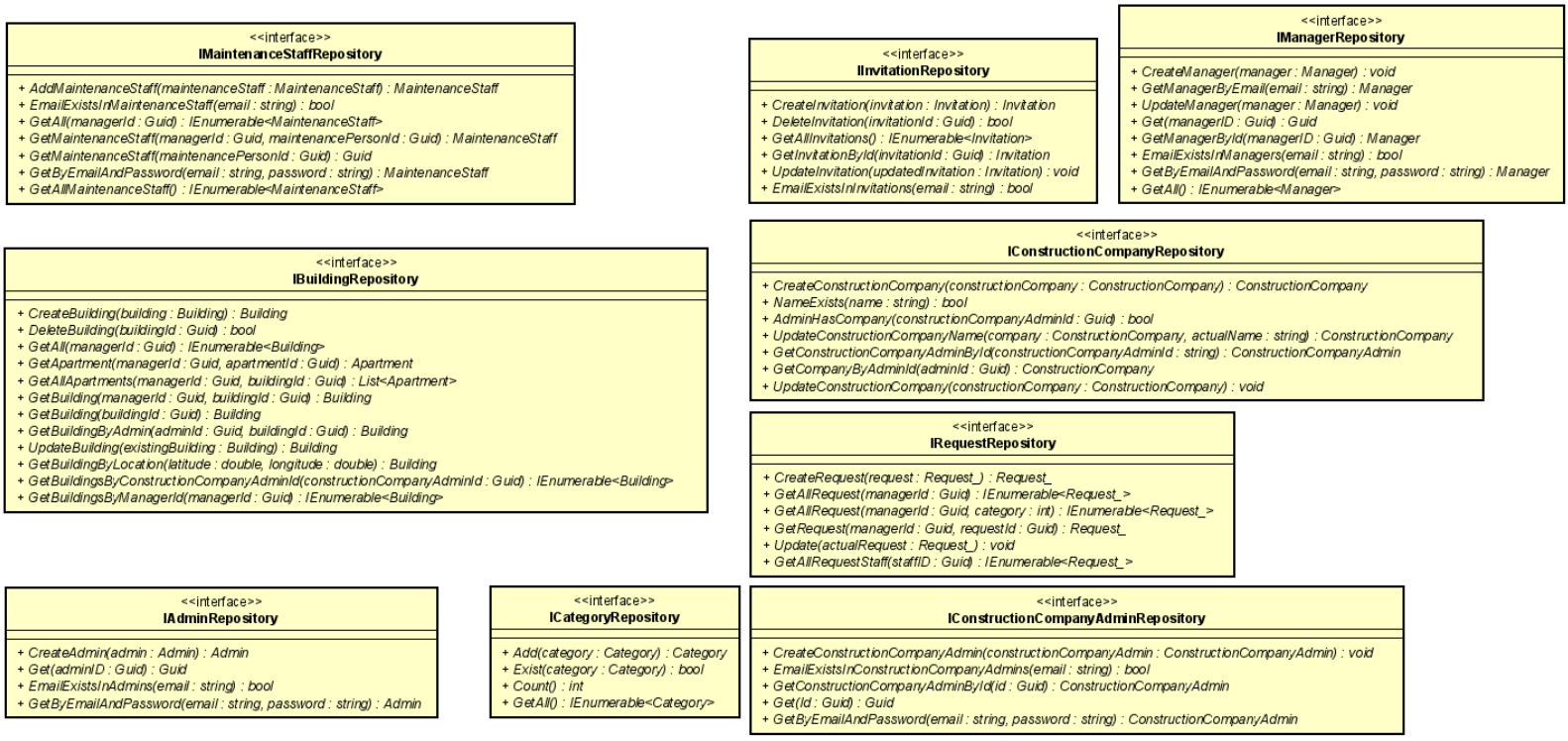
2.4 Diagrama de clases del paquete BussinessLogic.Logics :

Este paquete contiene las clases que implementan las interfaces de LogicInterface. Estas clases, además de contener todos los métodos definidos en su interfaz correspondiente, poseen métodos internos utilizados por la lógica de negocio para cumplir con sus responsabilidades. Estos métodos internos no se exponen a través de la interfaz, manteniendo así la encapsulación de las implementaciones.



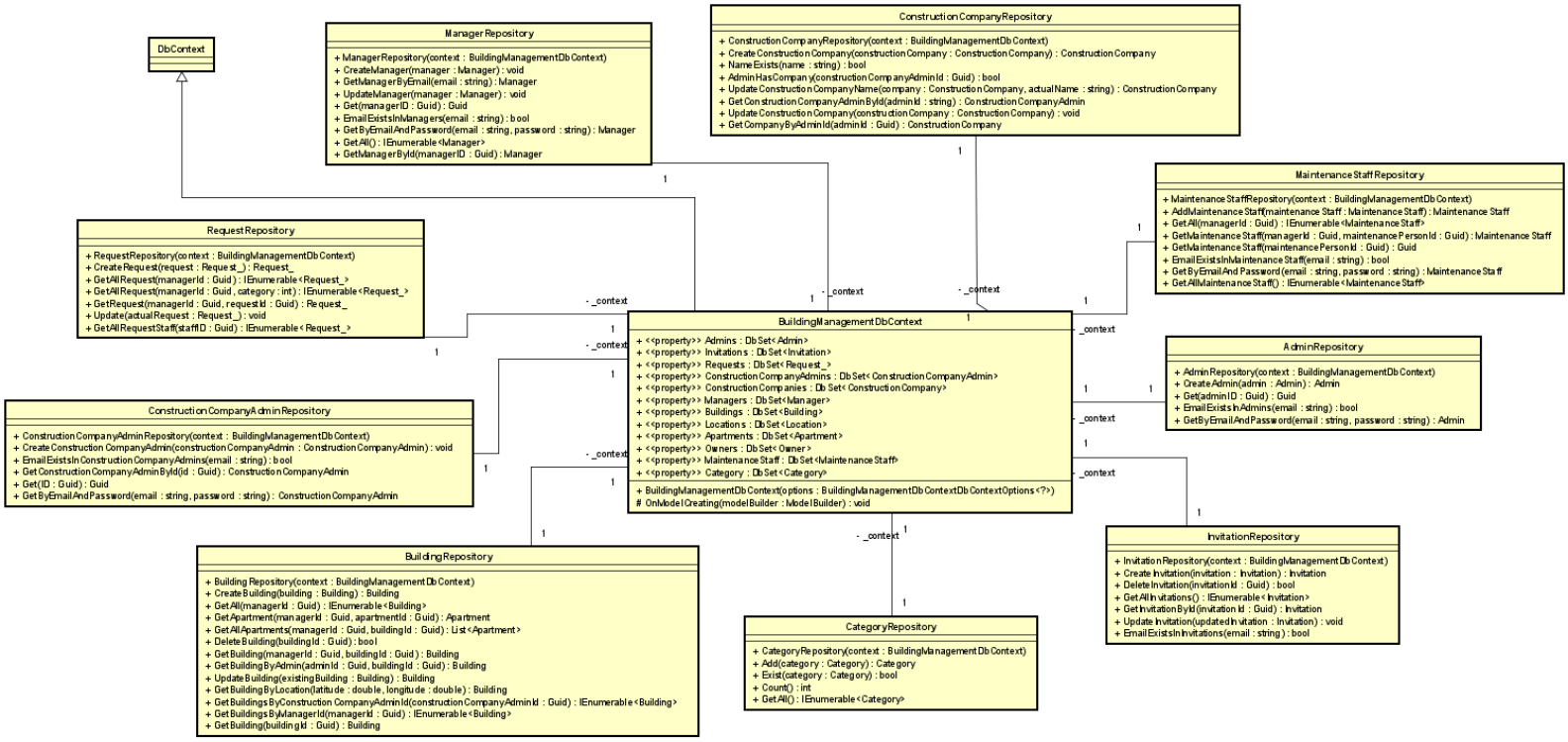
2.5 Diagrama de clases del paquete IDataAccess:

Este paquete contiene las interfaces de acceso a datos (DataAccess) utilizadas por la capa de lógica de negocio. Estas interfaces definen únicamente los métodos que serán invocados por las clases de la lógica de negocio, proporcionando una abstracción sobre las operaciones de acceso a datos. De este modo, se permite que la implementación concreta del acceso a datos permanezca oculta, garantizando un bajo acoplamiento y facilitando la flexibilidad y la capacidad de mantenimiento del sistema. La lógica de negocio puede interactuar con los datos sin conocer los detalles específicos de cómo se implementa este acceso, asegurando así una separación clara de responsabilidades.



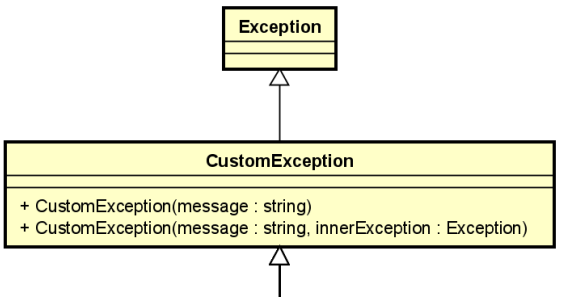
2.6 Diagrama de clases del paquete DataAccess:

Este paquete contiene las implementaciones de las interfaces de acceso a datos (IDataAccess). Las clases dentro de este paquete proporcionan las implementaciones concretas de los métodos definidos en las interfaces de acceso a datos. Al encapsular la lógica específica de acceso a datos, estas implementaciones permiten a la capa de lógica de negocio interactuar con los datos de manera abstracta, sin necesidad de conocer los detalles de las operaciones subyacentes. Esta estructura promueve un diseño modular, facilitando la mantenibilidad y la extensibilidad del sistema, al permitir cambios en la implementación del acceso a datos sin afectar a la lógica de negocio.

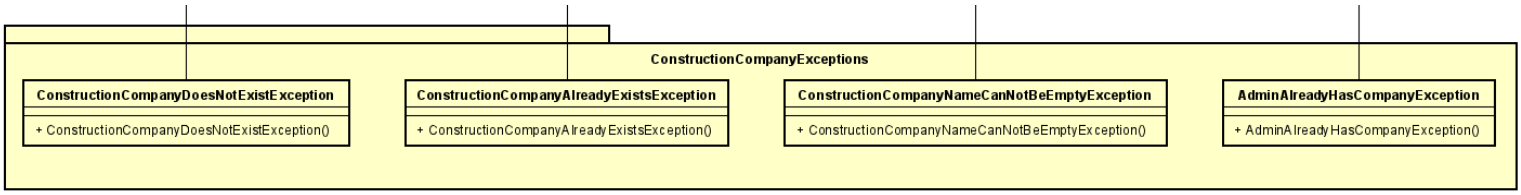


La carpeta Migrations, contiene la información del modelo para que EFCore pueda crear la base de datos en nuestra PC al usar Update-Database

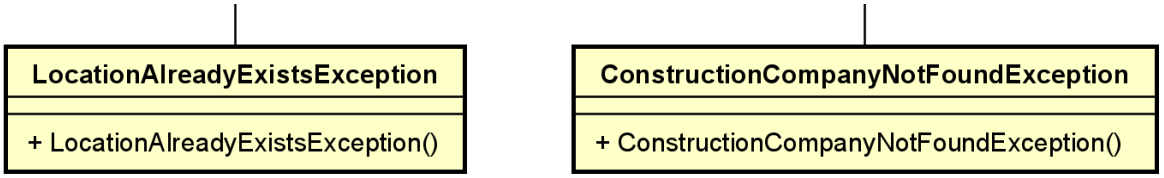
2.7 Diagrama de clases del paquete CustomExceptions:



Todas las clases de este paquete heredan de la clase CustomExceptions, la cual a su vez extiende la clase base Exception de .NET. Estas clases están diseñadas para gestionar las excepciones específicas de nuestro proyecto. Por esta razón, hemos decidido agruparlas todas dentro del mismo espacio de nombres (namespace), facilitando su organización y administración. Este enfoque centralizado asegura que el manejo de excepciones sea coherente y uniforme a lo largo de toda la aplicación, mejorando la claridad y la eficiencia en la gestión de errores.

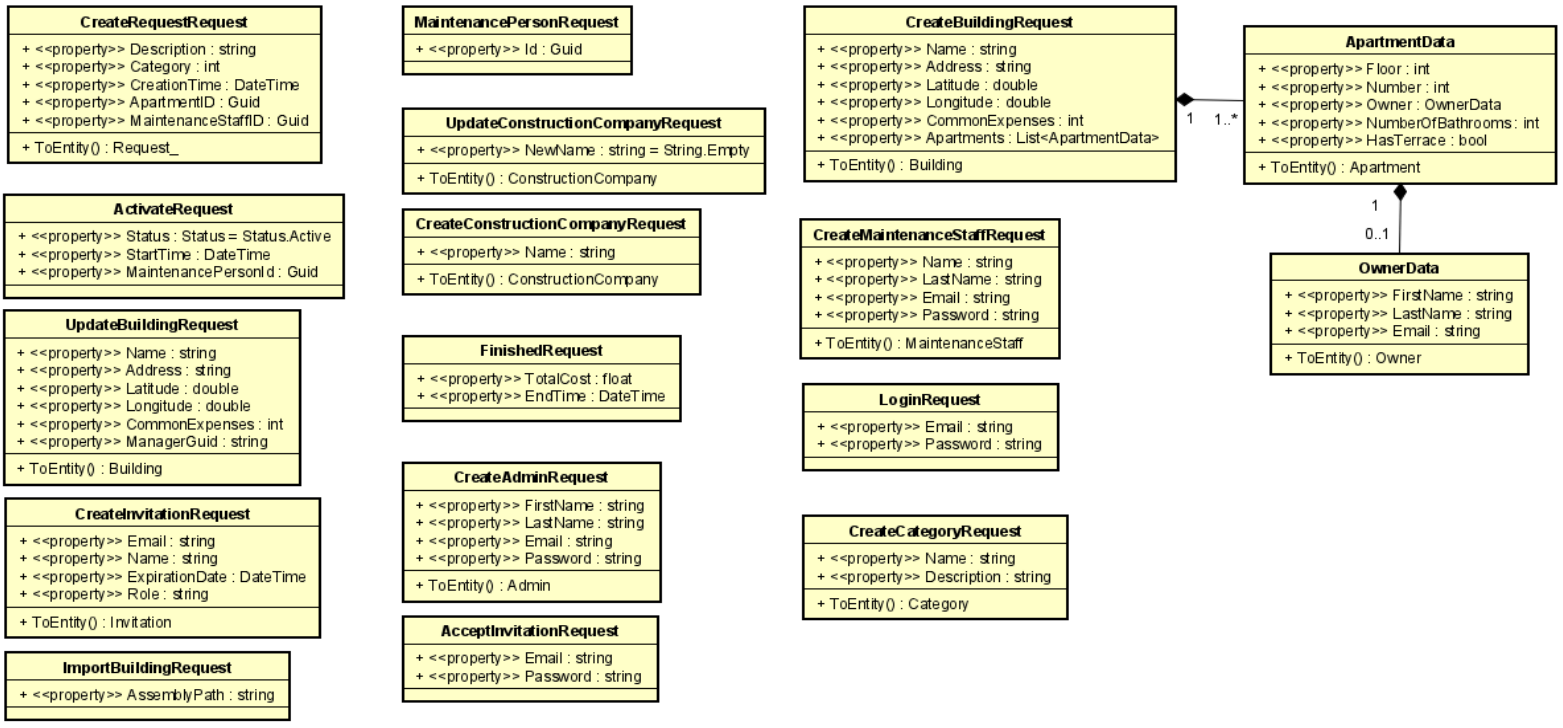


Los demás paquetes de excepciones son análogos a este. Las excepciones que son generales o que no justifican su inclusión en un paquete específico, se han colocado directamente dentro del paquete CustomExceptions. Por ejemplo, este paquete contiene excepciones (ver siguiente imagen) que son ampliamente utilizadas en distintas partes del proyecto o que no tienen una categoría específica que justifique su agrupación en un paquete separado.



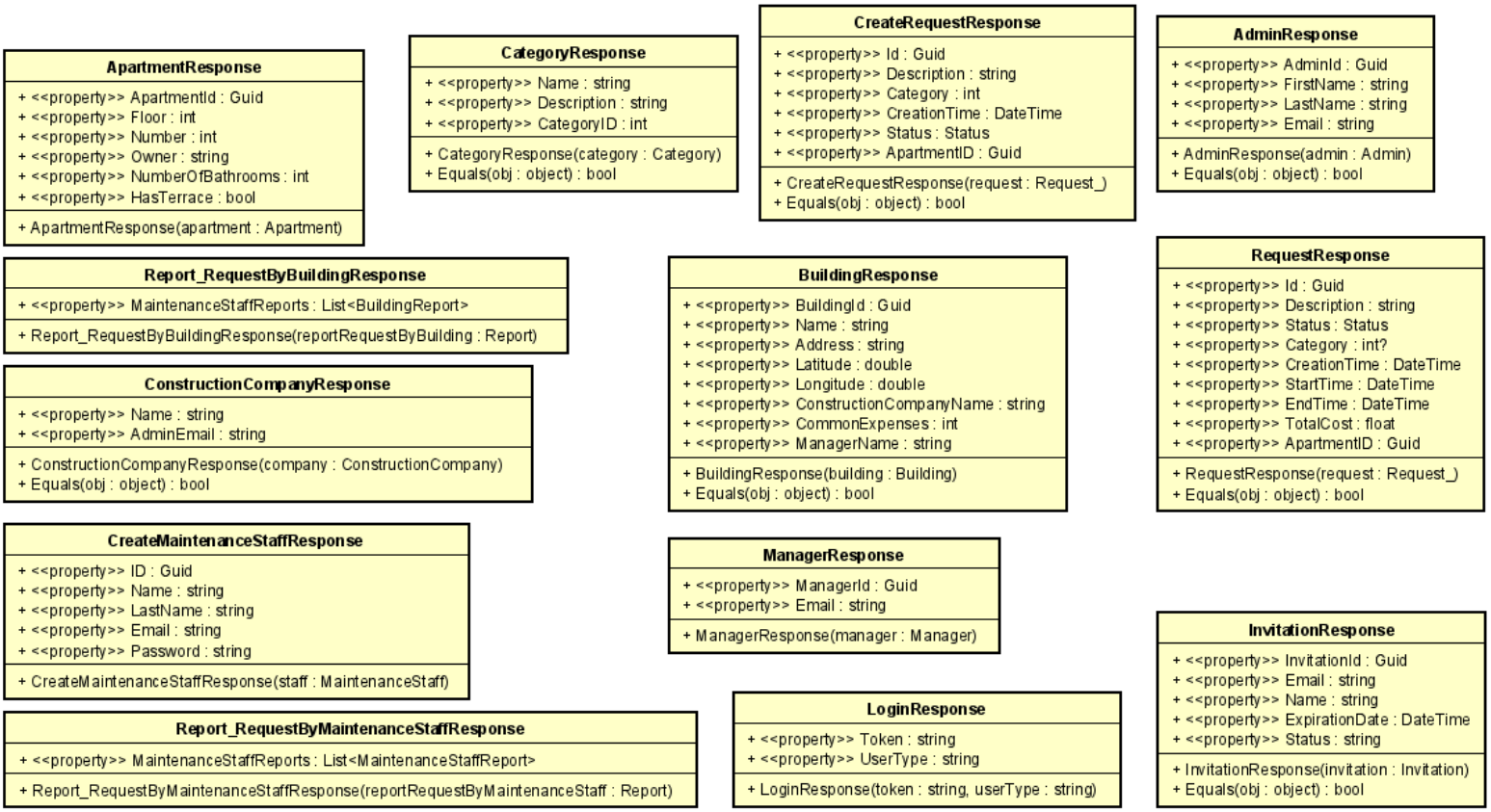
2.8 Diagrama de clases del paquete Models.In:

Este paquete contiene las clases que representan los modelos de entrada de la API, actuando como intermediarios para que el cliente no interactúe directamente con las clases del dominio. Cada solicitud proveniente de un cliente debe incluir un JSON que cumpla con las estructuras definidas en estas clases, ya que estas estructuras constituyen los parámetros de entrada de las funciones invocadas en los controladores. Este diseño asegura que los datos recibidos sean validados y transformados adecuadamente antes de ser procesados por la lógica de negocio, manteniendo así la integridad y la seguridad de la aplicación.



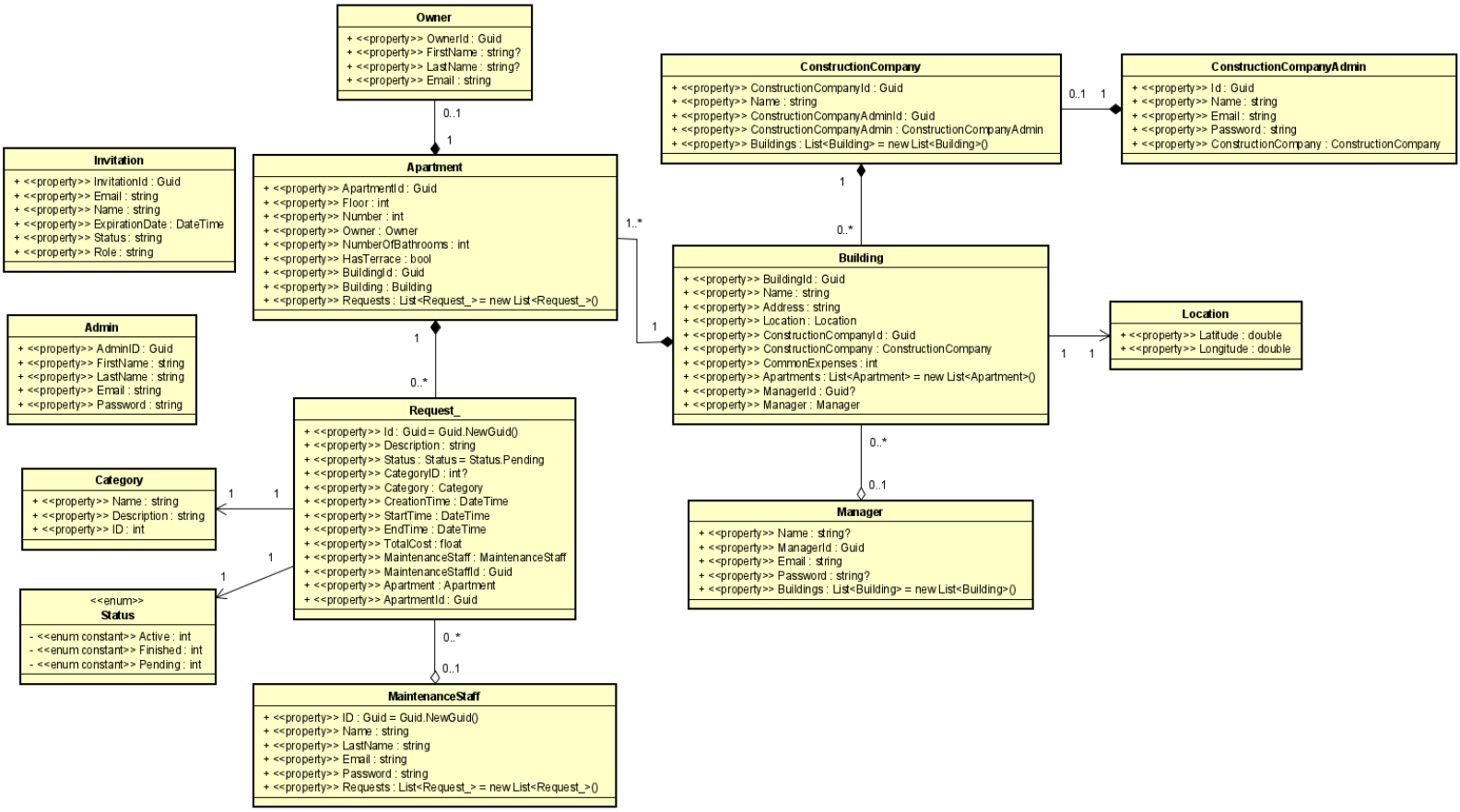
2.9 Diagrama de clases del paquete Models.Out:

Este paquete contiene las clases que representan los modelos de salida de la API, actuando como intermediarios para que el cliente no interactúe directamente con las clases del dominio. Cada respuesta enviada a un cliente está estructurada en formato JSON conforme a las clases definidas en este paquete, ya que estas estructuras constituyen los datos de salida de las funciones invocadas en los controladores. Este diseño asegura que los datos devueltos sean formateados y estructurados adecuadamente antes de ser enviados al cliente, manteniendo así la coherencia, integridad y seguridad de la aplicación.



2.10 Diagrama de clases del paquete Domain:

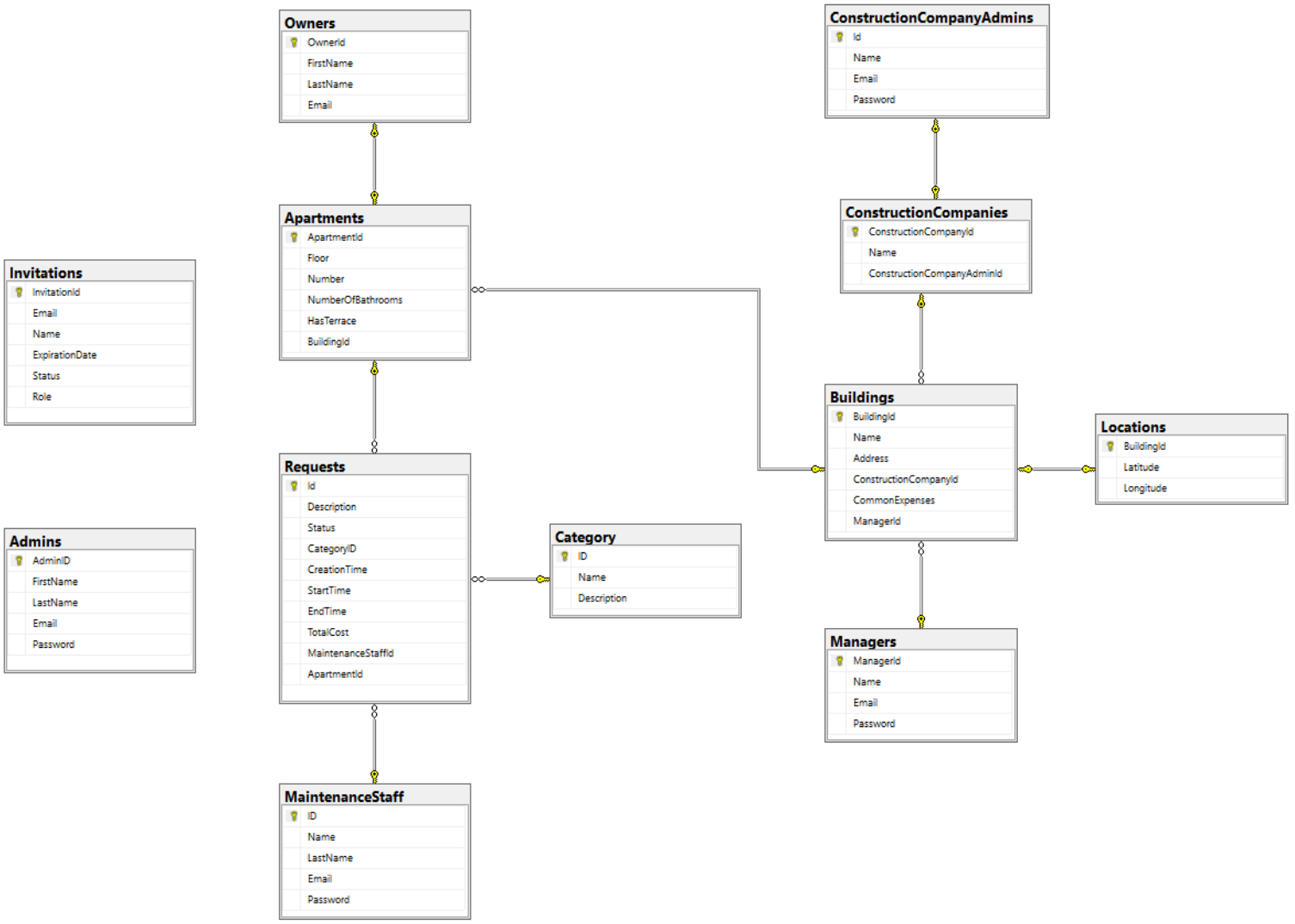
Este paquete contiene las clases del dominio, que representan los conceptos fundamentales y las reglas de negocio de la API. Las clases del dominio encapsulan la lógica central del sistema y son una carpeta independiente en el sistema. Por esto mismo, lo podemos definir como un sidecar domain ya que se mantiene separado de la carpeta de la lógica y permitiendo que todos los packages de la API lo conozcan si necesariamente conocer el package de la BL.



4. Descripción de jerarquías de herencia utilizadas (en caso de que así haya sido)

A pesar de la posibilidad de crear una jerarquía de herencia con una clase padre Usuario, de la cual todos heredarían debido a los atributos en común, no la implementamos. Esta idea de cambio no se llevó a cabo en esta entrega porque se consideró que sería muy complicado hacer ese cambio, ya que tenemos todo el sistema armado de otra manera, sin usar este polimorfismo de usuarios. No obstante, es una posibilidad de mejora importante. Luego se usaron herencias que ya nos otorga .NET, como la de el *ControllerBase* o la del Exception, de la cual creamos una herencia *CustomExceptions*.

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



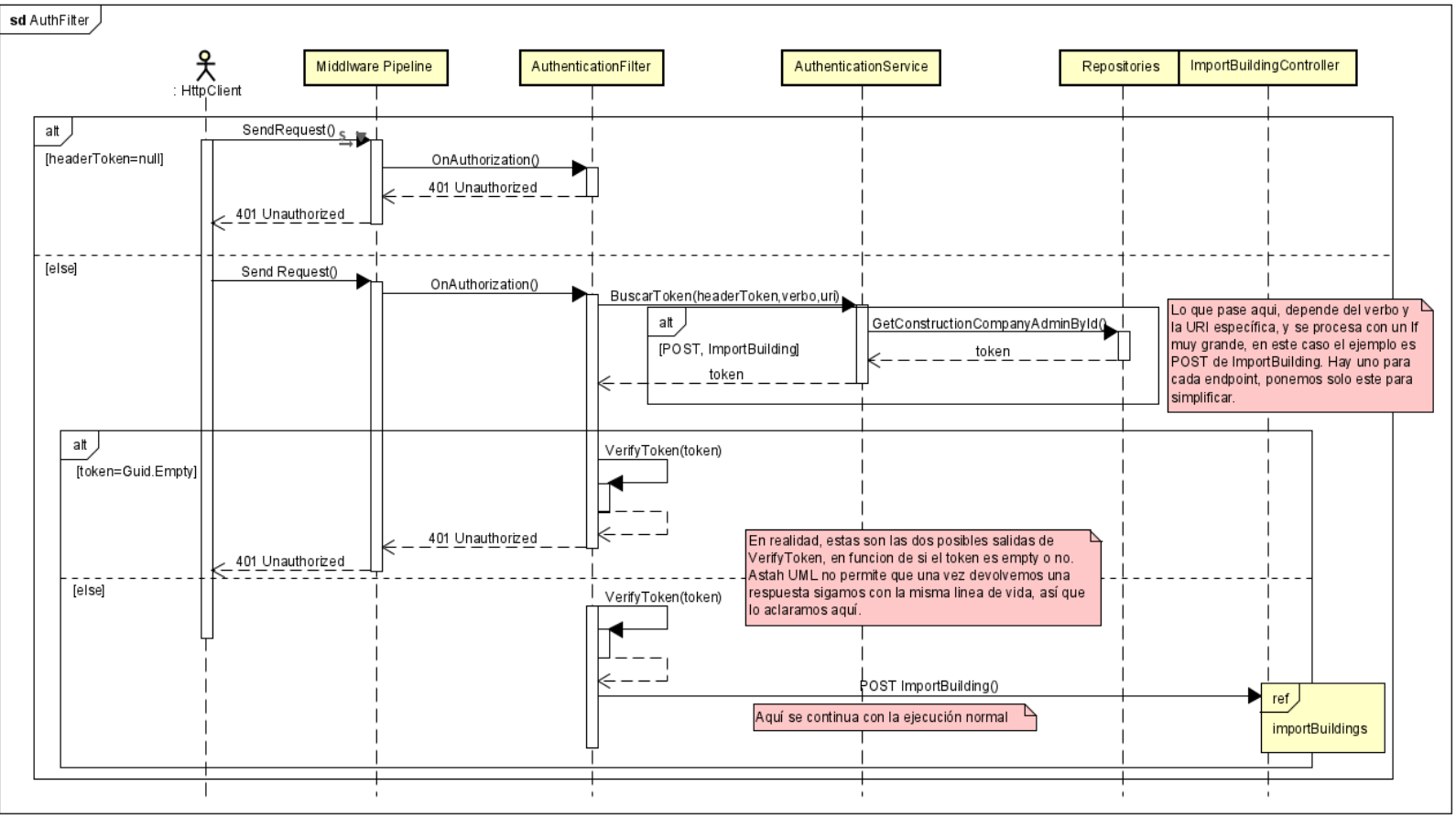
6. Para aquellas funcionalidades que el equipo entienda como relevantes:

o Diagramas de interacción que muestran las clases involucradas en estas funcionalidades y las interacciones para lograr la funcionalidad.
o Estas interacciones a mostrar pueden ser del flujo a alto nivel, de un cierto algoritmo específico de su obligatorio, etc.

Diagrama de secuencia del AuthenticationFilter:

A continuación, el diagrama de como funciona el filtro de excepciones. Empieza cuando el cliente mande una request, en este caso un POST del recurso *ImportBuildings* y termina una vez el filtro de autenticación ha realizado su función. El *HttpClient* envía una solicitud a la APIm por lo que primero debe pasar por el *Middleware Pipeline*, que la pasa al AuthenticationFilter. Si el *headerToken* es nulo, se devuelve un *401 Unauthorized*. Si el

headerToken no es nulo, el *AuthenticationFilter* llama al *AuthenticationService* para buscar el *token*. Dependiendo del *token*, si es vacío *(Guid.Empty)*, se devuelve un *401 Unauthorized*. Si no es vacío, se verifica el *token* y, si es válido, se procede con la solicitud y



se llama al *ImportBuildingController* para continuar con la ejecución normal.

Diagrama de secuencia de cómo se procesa el Post de de ImportBuidingLogic

User envía una solicitud *POST /ImportBuildings()* al *authFilter*. Luego, el *ImportBuildingController* recibe la solicitud y llama al *BuildingImportLogic* para cargar el importador *(LoadImporter(AssemblyPath))*. Dentro de un bucle, el *BuildingImportLogic* verifica cada tipo en el ensamblado; si el tipo es válido, crea una instancia del importador *(Activator.CreateInstance(type))*, de lo contrario, lanza una excepción de operación inválida. Posteriormente, en otro bucle, por cada *buildingDto* en la solicitud, se llama al *BuildingLogic* para crear el edificio *(CreateBuilding(adminGuid.ToString(), buildingDto))*. Finalmente, la respuesta *201 Created()* se devuelve al *User*, indicando que la importación se ha completado con éxito.

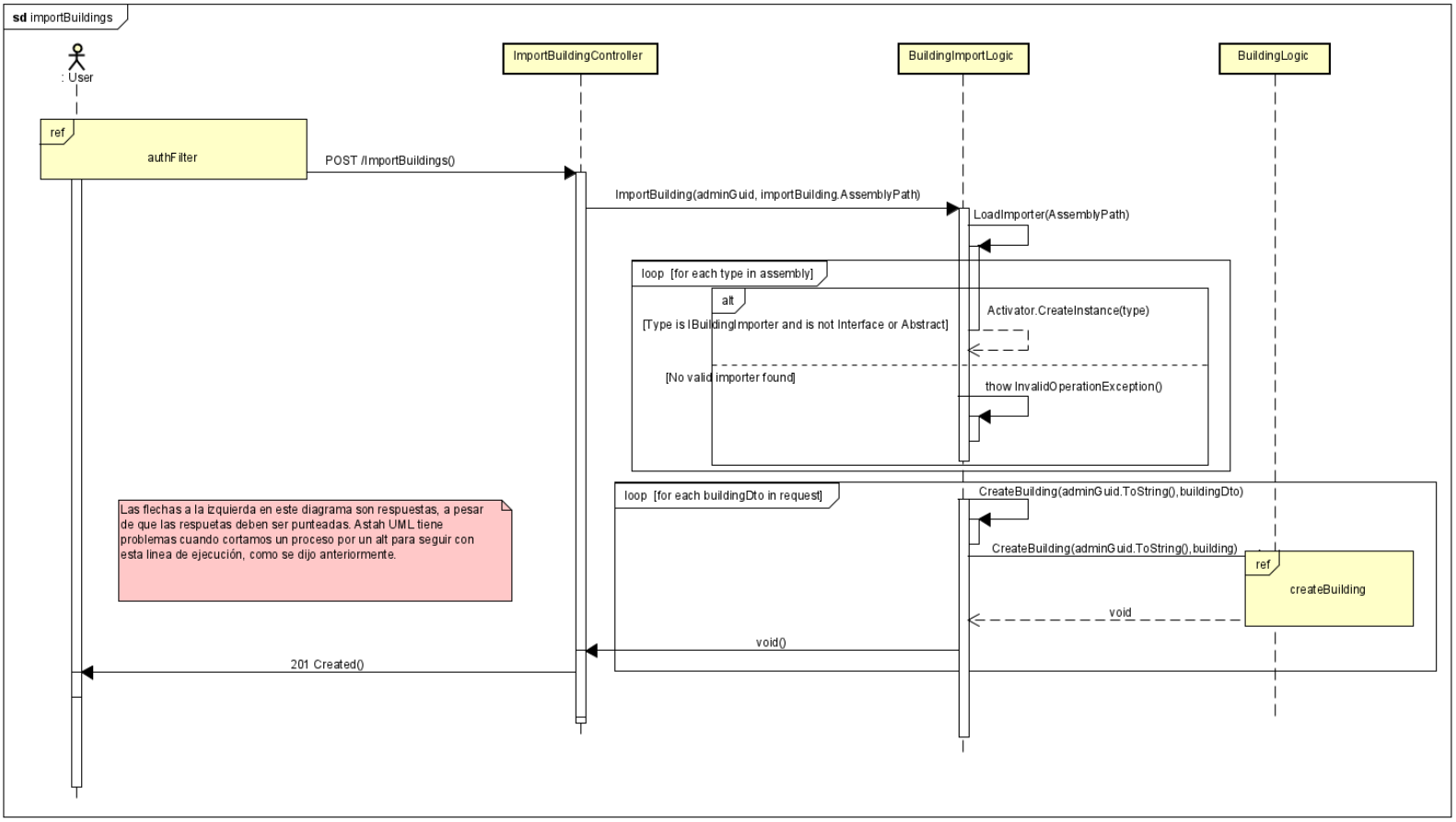
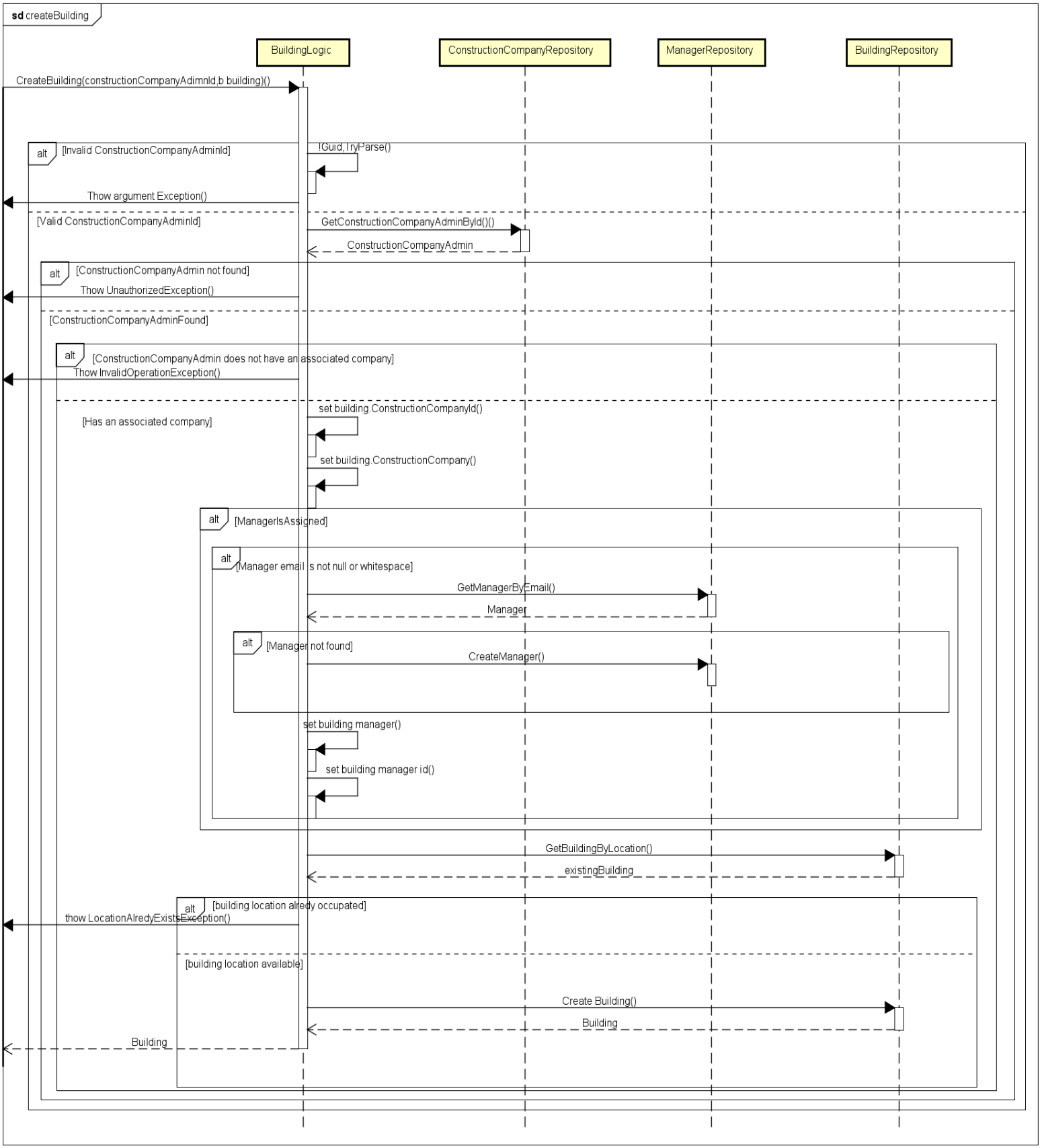


Diagrama de secuencia de cómo se procesa el Create de la BuildingLogic

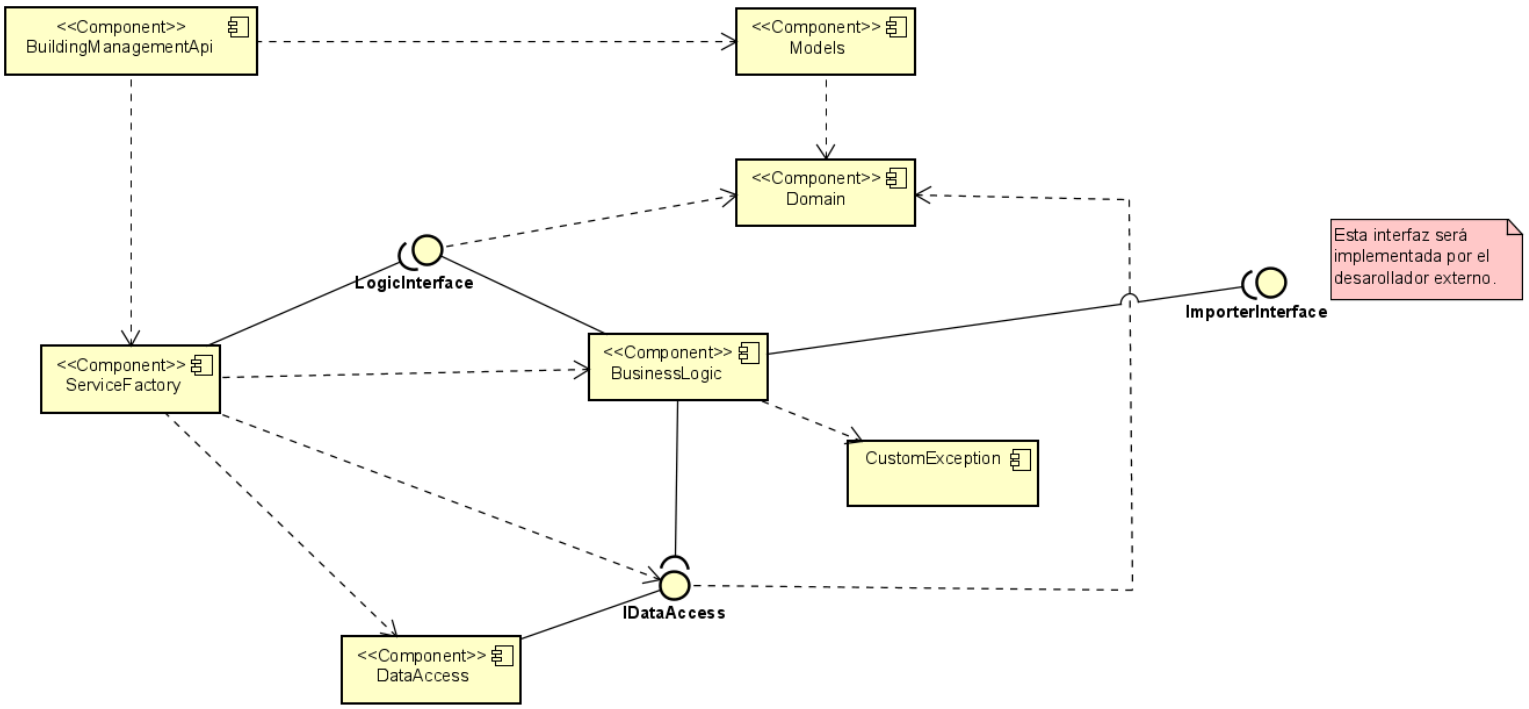
En esta accion, *BuildingLogic* inicia *CreateBuilding(constructionCompanyAdminId, buildingDto)*. Primero, válida *constructionCompanyAdminId* y obtiene la compañía de construcción correspondiente. Si la compañía es válida y el *manager* tiene una empresa asociada, se configura la compañía en el edificio. Luego, verifica si el *manager* está asignado y obtiene o crea un *manager* según sea necesario. Finalmente, verifica si la ubicación del edificio ya está ocupada; si no, crea el edificio y devuelve el objeto *Building*.
No se incluyeron todos los parámetros de entrada de las funciones para evitar mayor complejidad.


sd createBuilding
BuildingLogic
ConstructionCompanyRepository
ManagerRepository
BuildingRepository
CreateBuilding(constructionCompanyAdimnId,b building)()
alt
[Invalid ConstructionCompanyAdminId]
!Guid.TryParse()
Thow argument Exception()
[Valid ConstructionCompanyAdminId]
GetConstructionCompanyAdminById()()
ConstructionCompanyAdmin
alt
[ConstructionCompanyAdmin not found]
Thow UnauthorizedException()
[ConstructionCompanyAdminFound]
alt
[ConstructionCompanyAdmin does not have an associated company]
Thow InvalidOperationException()
[Has an associated company]
set building.ConstructionCompanyId()
set building.ConstructionCompany()
alt
[ManagerIsAssigned]
alt
[Manager email is not null or whitespace]
GetManagerByEmail()
Manager
alt
[Manager not found]
CreateManager()
set building manager()
set building manager id()
GetBuildingByLocation()
existingBuilding
alt
[building location already occupated]
thow LocationAlredyExistsException()
[building location available]
Create Building()
Building
Building

7. Diagrama de implementación (componentes) mostrando las dependencias entre los mismos. Justificar el motivo por el cual se dividió la solución en dichos componentes.

Motivo de la División de la Solución en Componentes

Hemos dividido nuestra solución en estos componentes específicos para mantener nuestro



sistema modular, flexible y fácil de mantener. Aquí les explico cada componente y la razón detrás de su existencia:

- ***BuildingManagementApi***:
    - **Motivo**: Este componente contiene todos los controladores de la API que permiten la comunicación con sistemas externos y usuarios. Esto nos permite modificar nuestra lógica interna sin afectar las interacciones externas.
- ***ServiceFactory***:
    - **Motivo**: Centralizamos la creación de servicios aquí para gestionar la instanciación de servicios de manera consistente. Esto facilita la inyección de dependencias y nos permite cambiar la implementación de los servicios de manera más sencilla.
- ***Models* y *Domain***:
    - **Motivo**: Separar los modelos de entrada y salida de los controladores(Models) y la lógica del dominio (Domain) es crucial para mantener un diseño limpio. Los modelos representan las entidades que se reciben y se envían con el exterior de la API, mientras que el dominio contiene los objetos de la lógica del negocio, que son los que se guardan en la base de datos. Esto hace que el que use la API no conozca directamente

como son las clases de la verdadera lógica, brindando seguridad y separación de capas.

- ***BusinessLogic***:
    - **Motivo**: Este componente contiene la lógica de negocio principal. Mantener la lógica de negocio separada nos permite escalar y mantener el sistema de forma más efectiva. Si necesitamos cambiar alguna regla de negocio, lo hacemos aquí sin afectar a otras partes del sistema.
- ***DataAccess***:
    - **Motivo**: La interacción con la base de datos está completamente encapsulada en este componente. Esto significa que si decidimos cambiar nuestra base de datos o la forma en que accedemos a los datos, podemos hacerlo sin afectar la lógica de negocio.
- ***LogicInterface* e *IDataAccess***:
    - **Motivo**: Estas interfaces nos permiten definir contratos claros para nuestras operaciones de lógica de negocio y acceso a datos. Así, si necesitamos cambiar la implementación, solo debemos asegurarnos de que la nueva implementación cumpla con la interfaz definida. Esto nos da la seguridad de que la capa de las capas más altas no conozcan la implementación de capas más bajas, brindando flexibilidad.
- ***ImporterInterface***:
    - **Motivo**: Definimos esta interfaz para que se pueda inyectar en tiempo de ejecución una implementación de un método de importar edificios, que será implementada por un desarrollador externo. Al tener una interfaz clara, garantizamos que cualquier implementación de importación cumpla con nuestras expectativas sin necesidad de cambiar el núcleo del sistema.
- ***CustomException***:
    - **Motivo**: Definimos excepciones personalizadas para manejar errores específicos de nuestra aplicación de manera controlada. Esto nos permite tener un manejo de errores consistente y predecible en todo el sistema, que luego serán capturadas por el filtro de excepciones.

8. Justificación y explicación del diseño en base al uso de principios de diseño, patrones de diseño y métricas.

8.1 Principios SOLID Aplicados:

Principios de Diseño Aplicados

1. **Principio de Inversión de Dependencias (DIP)**:
    - **Justificación**: Hemos asegurado que nuestros componentes dependan de abstracciones (interfaces) en lugar de implementaciones concretas. Esto promueve la flexibilidad y la facilidad para realizar cambios. Por ejemplo, *BusinessLogic* depende de *IDataAccess* en lugar de una implementación específica de acceso a datos, lo que permite cambiar la implementación de acceso a datos sin afectar la lógica de negocio.
2. **Principio de Segregación de Interfaces (ISP)**:

    - **Justificación**: Diseñamos interfaces específicas para la necesidad de cada componente. Esto evita que los componentes dependan de interfaces que contienen métodos que no utilizan, lo que mantiene el sistema limpio y manejable. *LogicInterface* y *IDataAccess* son interfaces claras y específicas para sus propósitos.
3. **Principio de Sustitución de Liskov (LSP)**:
    - **Justificación**: Los componentes e interfaces están diseñados de tal manera que las clases derivadas pueden sustituir a sus clases base sin alterar el correcto funcionamiento del sistema. Por ejemplo, cualquier implementación de *IDataAccess* debería poder reemplazar a otra sin necesidad de cambiar el comportamiento de *BusinessLogic*, siempre y cuando el resultado de la operación no se afecte. Además, como comentamos antes, al no hacer el polimorfismo de Usuarios (a pesar de que es una mejora) nos ahorramos tener problemas con este principio.
4. **Principio de Responsabilidad Única (SRP)**:
    - **Justificación**: Cada componente tiene una única responsabilidad claramente definida. Por ejemplo, *DataAccess* se encarga exclusivamente del acceso a datos y *BusinessLogic* maneja únicamente la lógica de negocio. Esto facilita la mantenibilidad y la prueba de cada componente por separado.

8.2 Patrones de Diseño Aplicados

1. **Facade**:
    - ***BuildingManagementApi*** actúa como una fachada, proporcionando una interfaz simplificada a un conjunto más complejo de interfaces en el sistema. Esto oculta la complejidad del subsistema y proporciona un punto de entrada único para los clientes externos.
2. **Factory**:
    - ***ServiceFactory*** centraliza la creación de servicios, permitiendo una gestión consistente y controlada de la instanciación de servicios. Este patrón es útil para la inyección de dependencias y la creación de instancias bajo demanda.
3. **Adapter**:
    - ***ImporterInterface*** sirve como un adaptador, permitiendo que una interfaz existente se adapte a una nueva interfaz. Esto facilita la integración con componentes externos sin necesidad de modificar el código existente.

8.3 Patrones GRASP Aplicados:

1. **Controller**:
    - ***BusinessLogic*** actúa como un controlador, centralizando la lógica de negocio y manejando las solicitudes del sistema. Este principio asigna la responsabilidad de manejar eventos del sistema a una clase específica, promoviendo un diseño cohesivo.
2. **Alta Cohesión**:
    - **Justificación**: Cada componente tiene responsabilidades claramente definidas y limitadas, asegurando que todas sus responsabilidades estén estrechamente relacionadas. Esto facilita el mantenimiento y la comprensión del sistema.

3. **Bajo Acoplamiento**:
    - **Justificación**: Hemos minimizado las dependencias entre componentes, asegurando que los cambios en un componente no afecten a otros. Por ejemplo, *BusinessLogic* depende de *IDataAccess* en lugar de una implementación concreta, permitiendo reemplazar componentes fácilmente.

9. Se debe discutir claramente los mecanismos utilizados para permitir la extensibilidad solicitada en la funcionalidad de este nuevo obligatorio.

Para extender las funcionalidades pedidas en este nuevo obligatorio, se realizó lo siguiente (dividiendo cada funcionalidad en un punto):

1. **Implementación de interfaz de usuario (UI):.**
    - **Explicación:** Se utilizó Angular CLI: 16.2.14 junto con Node: 18.20.3 (versiones compatibles) para crear un proyecto de frontend que consiste en una SPA(Single Page Application), donde utilizamos componentes, interceptores, guardias y servicios para implementar el mismo. El interceptor permite que se mande siempre por header el GUID del usuario de la sesión, y las guardias y servicio de *auth*, permiten que el usuario de este GUID solo pueda acceder a las páginas (es SPA, en realidad nos referimos a cada pantalla) de su tipo de usuario, hasta que la sesión del navegador finalice o se haga el logout. Cada una de estas páginas o pantallas, es un componente.
2. **Invitar a futuros admins de empresas constructoras:**
    - **Explicación:** Se creó un nuevo objeto de dominio, llamado *"ConstructionCompanyAdmin"* para poder soportar este nuevo rol. La invitación ahora tiene un nuevo campo llamado "*Role*", que indica si se está invitando un *ConstructionCompanyAdmin* o un *Manager.*
3. **Creación de empresa constructora y posibilidad de cambiarle el nombre:**
    - **Explicación:** Creamos una nueva clase del dominio llamada *"ConstructionCompany"* y un endpoint entero (controlador, lógica y acceso a datos) para poder acceder a la creación y modificación de empresas constructoras. Esta empresa constructora se añadió como un atributo del *ConstructionCompanyAdmin*, y ambos tienen *FKs* para navegar de uno al otro.
4. **CRUD de los edificios ahora es de ConstructionCompanyAdmin y posibilidad de asignar encargados a los mismos:**
    - **Explicación:** Para realizar esto, tuvimos que cambiar el modelo de la base de datos, y como estamos trabajando con *EFCore*, por consiguiente se modificó el dominio. La lista de edificios antiguamente pertenecía a un *Manager*, pero ahora pertenece a una *ConstructionCompany* y por tanto a su *ConstructionCompanyAdmin* asociado. Ahora el *Manager*, solo tiene una *FK* a un edificio, que es el que el administra. Los edificios, ahora tienen una *FK* a su *ConstructionCompany* y a un *Manager*(pero este último ahora puede ser nulo). Este enfoque de que la *ConstructionCompany* y el *ConstructionCompanyAdmin* sean 1 a 1, permite que no se tenga que guardar en los edificios quién es su *ConstructionCompanyAdmin* (se sabe

por transitiva). Además, se tuvo que modificar la lógica de negocio de los edificios, ya que antes todas estas acciones (CRUD) estaban pensadas para el usuario *Manager*.Además, se tuvo que implementar un GET (lo que llevó a crear un controlador) de los *Managers* para poder luego asignarlos a un *Building*.

5. **Los encargados ya no pueden dar de alta edificios, pero si continúan realizando el resto de las operaciones:**
    - **Explicación:** Ahora los Managers no son responsables del CRUD, pero pueden dar de alta personas de mantenimiento, y crear solicitudes para asignarle a estos. Este punto conllevo la modificación mencionada anteriormente en la parte de la lógica, donde la autenticación y autorización cambio para que estos campos CRUD ahora solo sean accesibles por la el *ConstructionCompanyAdmin*.
6. **Ahora las personas de mantenimiento pueden atender múltiples edificios:**
    - **Explicación:** Antes, las *MaintenanceStaff* (personal de mantenimiento) pertenecían a un *Building*, por lo que el *Building* llevaba una lista de los mismos, y solo eran asignables para las solicitudes de mantenimiento(*Requests*) pertenecientes a los apartamentos del mismo. Para hacer el cambio, se borro esta lista de *MaintenanceStaff* del *Building*, y se borro la FK que tenía cada *MaintenanceStaff* a su building. Además se quitó en la *BusinessLogic* la validación de que para crear la *Request*, este tenga que pertenecer a ese *Building*. Esto conllevo también cambios en el modelo de de la base de datos. Esto permite que cada persona de mantenimiento de la plataforma se pueda asignar a cualquier edificio. Es como un operador independiente. Podría ser una mejora que esta persona solo sea asignable por el encargado que la agregó a la plataforma.

Importar Edificios: Puesto que esta funcionalidad necesita detallar cómo se utiliza, le brindamos al desarrollador interno como se implementó con Reflection y cómo puede crear su propio método de importación de edificios para inyectar en tiempo de ejecución:

Descripción General

El ImporterInterface es un conjunto de clases DTO (Data Transfer Object) y una interfaz que permite la importación de información de edificios a tu aplicación. Esta interfaz está diseñada para ser extendida por cualquier método de importación que se necesite integrar.

Archivos del Proyecto

1. **BuildingDTO.cs**: Contiene las definiciones de clases para transferir datos de los edificios, apartamentos, localización, manager y propietarios.
2. **IImporter.cs**: Define una interfaz para los métodos de importación de edificios.

BuildingDTO.cs

Este archivo define varias clases utilizadas para representar la información necesaria de un edificio y sus componentes.

- **BuildingDTO**: Representa un edificio, incluyendo detalles como nombre, dirección, localización, gastos comunes, apartamentos, y su administrador.
- **LocationDTO**: Detalles de la ubicación geográfica del edificio.
- **ApartmentDTO**: Detalles de cada apartamento dentro del edificio.
- **ManagerDTO**: Información del administrador del edificio.
- **OwnerDTO**: Información del propietario de un apartamento.

Ejemplo de Uso de `BuildingDTO`

```
var building = new BuildingDTO

{

    Name = "Edificio Central",

    Address = "123 Calle Principal",

    Location = new LocationDTO { Latitude = -34.603722, Longitude = -58.381592 },

    CommonExpenses = 5000,

    Apartments = new List<ApartmentDTO>

    {

        new ApartmentDTO

        {

            Floor = 5,

            Number = 101,

            HasTerrace = true,

            NumberOfBathrooms = 2,

            NumberOfRooms = 3,

            Owner = new OwnerDTO { Email = "owner@example.com" }

        }

    },

    Manager = new ManagerDTO { Email = "manager@example.com" }

};
```

IImporter.cs

IBuildingImporter

Esta interfaz define un método que cualquier clase que implemente esta interfaz debe cumplir para importar edificios.

- **ImportBuilding()**: Método que debe ser implementado para devolver una lista de BuildingDTO importados.

Ejemplo de Implementación de IBuildingImporter

```
public class ExcelBuildingImporter : IBuildingImporter

{

    public List<BuildingDTO> ImportBuilding()

    {

        // Implementación para importar edificios desde un archivo Excel

        // Retorna una lista de BuildingDTO

    }

}
```

Consideraciones Importantes

Puntos a Favor

- **Flexibilidad en Métodos de Importación**: La interfaz IBuildingImporter simplifica la conexión, permitiendo al desarrollador crear una conexión a su base de datos, realizar llamadas API, o simplemente especificar una ruta de archivo, abriendo un gran abanico de posibilidades para métodos de importación.

Puntos en Contra

- **Gestión de Dependencias**: El desarrollador debe manejar las dependencias del archivo .dll cada vez que desee implementar una nueva forma de importación o reutilizar una existente. Esto puede aumentar la complejidad del manejo de las importaciones, especialmente cuando se cambian o actualizan las rutas de los archivos.

Notas para Desarrolladores

- Implementa la interfaz IBuildingImporter para crear nuevos métodos de importación.
- Utiliza las clases DTO para manejar datos consistentemente en toda la aplicación.
- Cualquier nuevo tipo de importación debe ser capaz de construir y devolver objetos BuildingDTO.

10. Se debe analizar la calidad del diseño discutiendo la calidad de este en base a las métricas de diseño y contrastándolas respecto a la aplicación de principios.

Assemblies	# Types	# Abstract Types	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
Domain v1.0.0.0	17	0	76	17	1.12	0.18	0	0.58
LogicInterface v1.0.0.0	12	12	25	20	0.08	0.44	1	0.31
IDataAccess v1.0.0.0	9	9	21	19	0.11	0.47	1	0.34
CustomExceptions v1.0.0.0	15	0	7	5	1	0.42	0	0.41
ImporterInterface v1.0.0.0	6	1	3	11	1	0.79	0.17	0.03
BusinessLogic v1.0.0.0	12	0	1	90	0.08	0.99	0	0.01
DataAccess v1.0.0.0	31	0	1	116	1	0.99	0	0.01
ServiceFactory v1.0.0.0	1	0	1	59	1	0.98	0	0.01
Models v1.0.0.0	31	0	11	35	0.1	0.76	0	0.17
BuildingManagementApi v1.0.0.0	17	0	0	146	0.88	1	0	0
JsonImporter v1.0.0.0	6	0	0	31	1.67	1	0	0
CsvImporter v1.0.0.0	3	0	0	41	1	1	0	0

En nuestro proyecto .NET, hemos realizado un análisis utilizando NDepend para evaluar el cumplimiento de varios principios de diseño de software. Nos enfocamos en los principios de Clausura Común (CCP), Reuso Común (CRP), Abstracciones Estables (SAP) y Dependencias Estables (SDP). Además, revisamos la métrica de distancia para comprender la relación entre abstracción e inestabilidad en nuestros módulos.

Principio de Clausura Común (CCP)

El CCP sugiere que las clases que cambian juntas deberían estar agrupadas en el mismo módulo. Este principio busca minimizar el número de paquetes que se cambian en cada ciclo de lanzamiento. Para medirlo, podemos tomar que el CA(Afferent Coupling) debería ser alto, y el CE(Efferent Coupling) moderado o bajo en relación a este. De esta forma, podemos ver que la única clase que parece cumplir con esto es *Domain*. Tiene sentido, ya que las clases del dominio se relacionan entre sí y no dependen de ninguna otra.

Principio de Reuso Común (CRP)

El CRP sugiere que las clases que no se reutilizan juntas no deberían estar agrupadas en el mismo módulo. Esto evita que los cambios en clases no relacionadas obliguen a

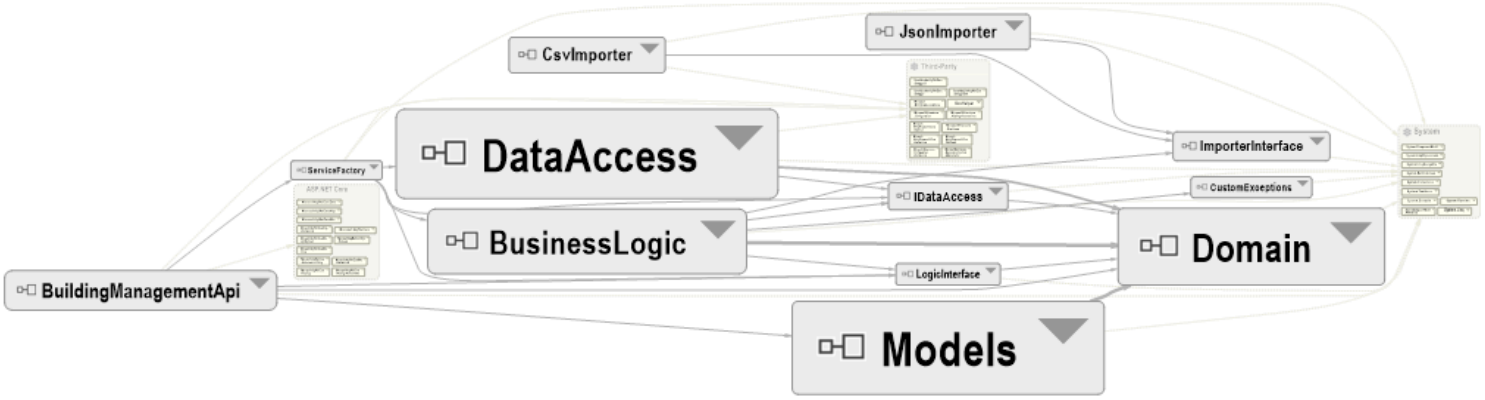
actualizaciones innecesarias en otros módulos. Para medir esto, podemos usar como referencia un CE bajo, ya que las clases del mismo módulo no deberían depender de clases externas, sino de dentro del mismo. Si analizamos la tabla obtenida, podemos concluir que nuevamente *Domain* es la más se amolda a este principio, seguida por las interfaces. Nuevamente tiene sentido, ya que las clases del dominio son todas reutilizadas a la vez por los distintos módulos, ya que cada uno contiene la lógica y el acceso a datos por ejemplo de todos, por lo que usa todas las clases del dominio. También las interfaces, que son llamadas por el mismo módulo siempre, hacen sentido en estar correctas.

Estos dos primeros principios se contraponen bastante, así que lo más óptimo es buscar un equilibrio entre ellos. Además, por las características de nuestro proyecto en ser layered y con DIP, claramente se pierde la objetividad del CCP y el CRP.

Cohesión Relacional: La cohesión relacional mide como los elementos dentro de un módulo están relacionados entre sí y trabajan juntos para cumplir una única tarea o propósito. Esta métrica se calcula como (Número de relaciones internas del paquete + 1)/ Número de clases e interfaces dentro del paquete. El estándar es que este valor sea mayor a 1,5 o menor a 4,0.

En nuestro proyecto, vemos que se rompe, pero esto es debido a cumplir con la D(DIP) de SOLID. La aplicación del Principio de Inversión de Dependencias requiere la introducción de abstracciones como interfaces o clases abstractas, que fragmentan las relaciones directas entre los componentes del módulo. Estas abstracciones dispersan la responsabilidad y el conocimiento contextual, lo que puede reducir la cohesión relacional, ya que los elementos que solían trabajar juntos de manera cohesiva ahora interactúan a través de múltiples capas de abstracción. Aunque esto puede disminuir la cohesión relacional, es crucial para lograr un diseño más flexible, mantenible y desacoplado, facilitando futuras modificaciones y extensiones del sistema.

Principio de Dependencias Estables (SDP).



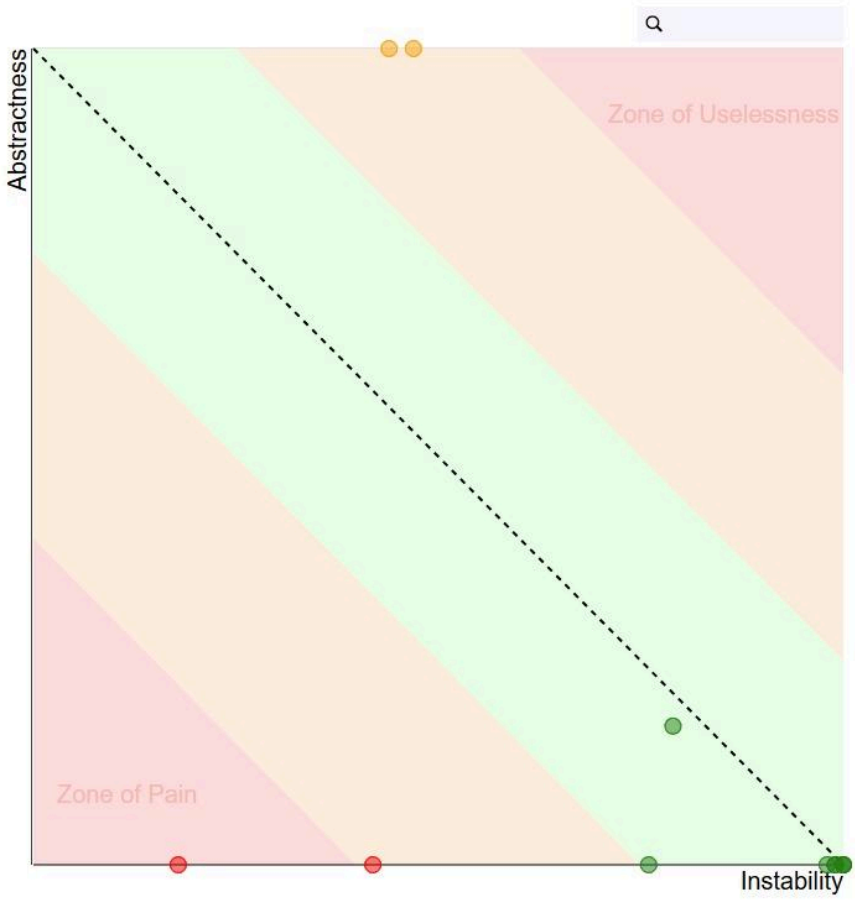
Este principio establece que un paquete debe siempre depender de uno igual o más estable que este. En el diagrama generado por NDepend, todas las flechas van a la derecha, así que analicemos en base a la tabla y este diagrama. *Domain* es el más estable, no depende de nadie y todos dependen de él, por lo que cumple con SDP. *ImporterInterface* y *CustomExceptions* no dependen de nadie, por lo que cumplen SDP. *LogicInterface*, *IDataAccessy Models* solo dependen de *Domain*, por lo que también cumplen. *DataAccess*

y *BusinessLogic* tienen múltiples dependencias a sus interfaces, dominio y excepciones, pero todos estos son más estables que ellos, así que lo cumplen. Siguiendo con esto todos los paquetes cumplen.

Sin embargo, el que hace que violemos el SDP es el ServiceFactory, ya que depende de todos en el proyecto y en concreto, depende de BusinessLogic y DataAccess que son más inestables que él (0,98 vs 0,99 de ambos). La solución a esto, podría ser agregar más lógicas e interfaces, o sea inyectar más dependencias para mapear en la ServiceFactory, para que de esta forma el CE suba y así nos de mayor al del DataAccess y BusinessLogic.

Métrica de Distancia y SAP (D)

La métrica de distancia mide la desviación de un módulo de la línea ideal de abstracción vs estabilidad lo que está directamente relacionado con el SAP. Decimos que esta métrica es correcta en un módulo cuando es menor a 0,3. Si la distancia es mayor a 0,3, se viola el SAP. Introducimos a continuación el gráfico de la métrica distancia:



Como podemos ver, todos los paquetes del proyecto lo cumplen, exceptuando los que se encuentran cerca de la Zone Of Pain, que son el *Domain* (está dentro de Zone Of Pain) y *Exceptions*; y los los que se encuentran cerca de la Zone of Uselessness: *IDataAccess* y *LogicInterface*. Esto se debe a que el Dominio y las excepciones no tienen nada abstracto y no dependen de nadie, pero todos dependen de ellos (más en concreto del dominio), por lo que cualquier cambio en el dominio afectaría a todos los que dependen del mismo, por

ejemplo. Por otro lado, las interfaces que se encuentran cerca de la Zone Of Uselessness ,están ahí debido a que son totalmente abstractas, dependen únicamente de las clases del dominio y son dependidas solamente por las clases del paquete que las utiliza. En conclusión, hace sentido que están donde están, pero se podría mejorar. Por ejemplo el dominio se podría mejorar haciendo una interfaz del mismo IDomain, del cual todos dependan y haciendo que Domain la implemente, retirándolo de la Zone Of Pain.

Anexo:

- Cambios respecto la versión anterior:
    - Se actualizaron las URI para que incluyan la versión actual de la API (/v2).
    - También se movió el endpoint respecto a la creación de Maintenance Staff. Anteriormente estos se creaban junto al edificio, ahora tienen un endpoint específico ajeno a los edificios.
    - Se creo el controller para el LogIn, para la importacion de edificios, para las companias de construccion.
    - Se modifico el AuthenticationFilter para incluir los nuevos cambios y agregados de los controladores nombrados
- Informe de cobertura

Introducción

El presente informe detalla la cobertura de código de las principales componentes del proyecto: `BuildingManagementApi.Controllers`, `BusinessLogic.Logics` y `DataAccess`. La cobertura de código es una métrica importante que mide la cantidad de código que se prueba mediante pruebas automatizadas, proporcionando una indicación de la calidad y la estabilidad del código.

Cobertura de Código por Componentes

1. **BuildingManagementApi.Controllers**: Este componente alcanza una cobertura perfecta del 100% con un total de 28 pruebas. Esta cobertura completa asegura que cada línea de código ha sido evaluada, minimizando el riesgo de errores en la ejecución en producción.
2. **BusinessLogic.Logics**: La cobertura de código es del 90% con un total de 141 pruebas. Las principales clases dentro de esta componente muestran una alta cobertura, con la excepción de:
    - `BuildingImportLogic`, que tiene una cobertura del 0% debido a dificultades técnicas para realizar mocks sobre la clase `Assembly`, necesaria para testear la función de lectura de assemblies.
    - Las lógicas de generación de reportes, cada una con un 97% de cobertura debido a un switch que incluye un caso `default`. Este caso se mantiene como una salvaguarda para manejar futuros estados en el enum sin afectar la solución actual.
3. **DataAccess**: Presenta una cobertura de 96% con un total de 76 pruebas, lo que refleja un alto nivel de fiabilidad en las operaciones de acceso a datos.

Análisis de BusinessLogic.Logics

Excluyendo la clase `BuildingImportLogic`, la cobertura de `BusinessLogic.Logics` estaría cerca del 100%. Esto se debe a que de las demás clases analizadas, 9 alcanzan una cobertura del

100% y 2 un 97%. Esta métrica sugiere que la mayoría de las funcionalidades están completamente validadas por pruebas, fortaleciendo la confiabilidad del componente.

Beneficios de la Alta Cobertura de Código

Mantener una alta cobertura de código, como se observa en los componentes descritos, ofrece múltiples beneficios:

- **Reducción de errores**: Una alta cobertura ayuda a identificar y corregir errores antes de que el código se implemente en producción, reduciendo el riesgo de fallas operativas.
- **Mantenimiento simplificado**: El código bien probado facilita la refactorización y la actualización, ya que los cambios pueden introducirse con la seguridad de que no afectarán inadvertidamente otras partes del sistema.
- **Confianza durante la integración**: Al integrar nuevas funcionalidades o componentes, una alta cobertura de pruebas garantiza que las interacciones entre componentes sean previsibles y estables.

Conclusión

La cobertura de código es un indicador clave de la calidad del software. A pesar de ciertos desafíos técnicos específicos, el proyecto mantiene una cobertura excepcionalmente alta, reflejando un compromiso continuo con la calidad y la estabilidad del sistema. La inversión en pruebas y cobertura de código se traduce en un software más robusto y confiable.

Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Blocks)
sebae_DESKTOP_SEBA_2024-06-10.15_29_40.coverage	12076	1586	7003	30	2353	88.39%
buildingmanagementapi.dll	294	154	185	0	153	65.63%
{ } BuildingManagementApi.Controllers	294	0	185	0	0	100.00%
{ } BuildingManagementApi.Filters	0	104	0	0	105	0.00%
{ } Global Classes	0	50	0	0	48	0.00%
buildingmanagementapitest.dll	1333	2	770	1	0	99.85%
businesslogic.dll	946	101	639	2	63	90.35%
{ } BusinessLogic.Logics	946	101	639	2	63	90.35%
AdminLogic	21	0	23	0	0	100.00%
AuthenticationService	208	0	124	0	0	100.00%
BuildingImportLogic	0	79	0	0	51	0.00%
BuildingImportLogic.<>c	0	17	0	0	8	0.00%
BuildingLogic	160	0	112	0	0	100.00%
CategoriesRequestsLogic	15	0	16	0	0	100.00%
ConcreteReportFactory_RequestByBuilding	86	2	52	0	2	97.73%
ConcreteReportFactory_RequestByMaintenanceStaff	91	2	64	0	2	97.85%
ConstructionCompanyLogic	46	0	39	0	0	100.00%
InvitationLogic	84	0	66	0	0	100.00%
MaintenanceStaffLogic	47	0	40	0	0	100.00%
ManagerLogic	9	0	10	0	0	100.00%
RequestLogic	179	1	93	2	0	99.44%
businesslogictest.dll	5114	118	2603	3	112	97.74%
customexceptions.dll	28	4	36	0	5	87.50%
dataaccess.dll	1315	1101	427	1	1931	54.43%
{ } DataAccess	1315	45	427	1	13	96.69%
AdminRepository	61	0	21	0	0	100.00%
BuildingManagementDbContext	211	2	70	0	2	99.06%
BuildingRepository	382	1	93	1	0	99.74%
CategoryRepository	34	0	19	0	0	100.00%
ConstructionCompanyAdminRepository	80	1	25	0	2	98.77%
ConstructionCompanyRepository	84	0	35	0	0	100.00%
ConstructionCompanyRepository.<>c__DisplayClass7_0	4	0	1	0	0	100.00%
InvitationRepository	74	0	38	0	0	100.00%
MaintenanceStaffRepository	70	41	27	0	9	63.06%
ManagerRepository	109	0	39	0	0	100.00%
RequestRepository	206	0	59	0	0	100.00%
{ } DataAccess.Migrations	0	1056	0	0	1918	0.00%
dataaccesstest.dll	2356	1	1824	0	1	99.96%
domain.dll	178	15	169	0	15	92.23%
models.dll	512	90	350	23	73	85.05%

Especificacion Formal de la API:

Tabla con los distintos endpoints y la información correspondiente en el archivo especifiacion_formal_api.xlsx
Se encuentra en otro archivo ya que al incorporar la tabla en el documento su lectura era imposible.