Universidad ORT Uruguay

Facultad de Ingeniería

Bernard WAND Polak

Diseño de Aplicaciones 2

Obligatorio 2

Mateo Mazzini Nro. est. 219372

Emiliano Marotta Nro. est. 187884

Link repositorio: https://github.com/IngSoft-DA2-2023-2/219372_187884

Grupo N5A

Docente teórico: Pablo Geymonat

Docentes tecnología: Marco Fiorito / Juan Barrios

Índice

Introducción

Esta entrega tiene como consigna la implementación de nuevos requerimientos para el sistema de gestión de edificios trabajado en la entrega anterior, desarrollando también el Front-end de la aplicación.

Para la segunda instancia se pide desarrollar nuevos requerimientos del back-end trabajado en la primera entrega, y trabajar en un front-end que va a tener que estar conectado con el back para obtener y guardar los datos. Para desarrollar el front-end tuvimos que usar como framework Angular 18, siendo el desafío principal de esta entrega entender cómo crear componentes utilizando buenas prácticas y conectar el back con todas sus funcionalidades, con los componentes creados en el front.

El resultado final nos deja conformes, pero no quita que hubo una gran curva de aprendizaje con la práctica.

Durante el desarrollo de esta segunda entrega nos encontramos con varios desafíos y momentos de desconcierto que fueron resueltos a través de los conocimientos adquiridos a lo largo de esta segunda parte del curso.

La evidencia sobre las diferentes decisiones de diseño del proyecto será desarrollada en este documento, a través de diferentes ejemplos sacados del código entregado para facilitar la visualización de los mismos. Lo mismo con los diagramas que serán dispuestos en sus respectivos sectores.

Decisiones de diseño e implementación

Descripción general del trabajo

Para la realización de la segunda entrega nos centramos inicialmente en el desarrollo de los requerimientos nuevos para el back-end y la corrección y adaptación de los viejos respecto a los nuevos.

Una vez asentados algunos de los nuevos requerimientos, nos dividimos las tareas para empezar a trabajar en el desarrollo del front-end.

Para el front-end empezamos por desarrollar componentes que interactúan con el back de forma limitada, es decir, solo fueran para mostrar elementos en pantalla a través del parseo de los datos obtenidos en las responses de request tipo GET. Estas responses fueron trabajadas con “models”, que son archivos donde definimos los tipos de datos que contiene una response y así podemos parsear los atributos de las responses establecidos en los DTOs utilizados por los controllers, para mostrarlos en el front y trabajar con ellos.

Estas primeras instancias de generación de componentes nos sirvieron para ir entendiendo la dinámica de angular y la interacción y responsabilidades de sus archivos, como la definición de archivos de servicios, de endpoints, de rutas, etc.

Una vez adquiridos otros conocimientos sobre la herramienta, empezamos a priorizar la creación de componentes por la dependencia de unos con otros, es decir, antes de hacer el componente de building, creamos el componente de company, dado que un building no puede existir sin que exista una compañía que lo componga.
De esta manera partimos de un diseño con componentes atómicos que no tuvieran dependencia de otros, hacia componentes más complejos en los cuales se mostraran listas de objetos para la creación de los mismos.

Mientras trabajamos en el desarrollo de los componentes visuales del front-end, también desarrollamos las guardas y servicios de autenticación y autorización que son fundamentales para nuestra solución que contiene usuarios con diferentes roles, capacidades y responsabilidades, pudiendo con estos elementos restringir o permitir acceso a funcionalidades de la aplicación para los diferentes roles.

Decisiones de diseño

Algunas decisiones de diseño a mencionar en este proceso de desarrollo son:

- **Nuevos atributos Company:** Se agregó en la clase de dominio Company el atributo CompanyAdmin, ya que al momento de crear la empresa constructora, es más sencillo obtener el id del usuario.
  En cambio si se agrega Company en CompanyAdmin, habría que, al momento de crear la empresa constructora, editar el Companyadmin poniendo el id de dicha empresa, por lo que en cualquier edifico de CompanyAdmin, ya sea de nombre, apellido etc, habría que pasar la empresa constructora.

- **Nuevos atributos en Session:** En la clase Session se agregó el atributo Role, para poder obtener, a parte del token de la session, el rol del usuario correspondiente a la misma.
  Esta decisión se tomó en base a facilitar la obtención del rol del usuario para poder restringir en base a este, el acceso a funcionalidades y páginas de la aplicación, con el uso de las guardas y otros elementos como *ngIf para ocultar en base a los roles, componentes del front-end.
  Un ejemplo es en la sidenav, donde ocultamos los botones de navegación a ciertas páginas en base al rol:

```
<li *ngIf="getRole() === 'CompanyAdmin'">
    <a routerLink="/company" class="flex items-center p-2 text-base font-bold text-emerald-700 rounded-lg hover:bg-emerald-700 hover
        <svg fill="currentColor" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" class="size-6">
            <path d="M17 10V6l-5 4V6l-5 4V4H2v16h20V6l-5 4zm-8 7H7v-3h2v3zm5 0h-2v-3h2v3zm5 0h-2v-3h2v3z"></path>
        </svg>
        <span class="ml-3">Company</span>
    </a>
</li>
```

- **Supresión de validaciones:** Dado el último formato de importers requerido, preferimos suprimir la validación del algunos datos en la creación de Buildings, flexibilizando el input de datos con formatos diferentes a los que trabajamos inicialmente. Esta supresión de validaciones afecta a la validación del formato de ingreso en las propiedades Address y Location.

- **Cambio en relaciones:** Se borró la relación Staff - Building, ya que un staff puede trabajar en cualquier edificio, por lo que no es necesario mantener la relación. Creemos que no aporta valor

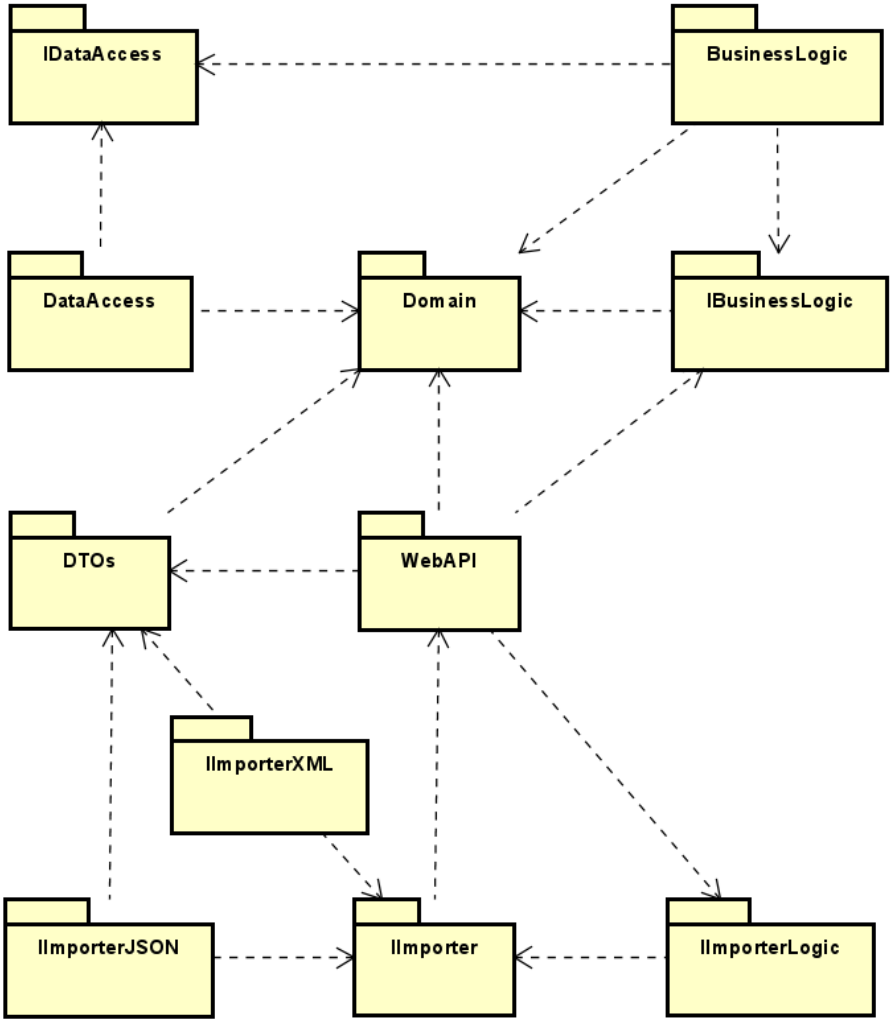
Errores conocidos e implementaciones faltantes

Respecto a la implementación de los requerimientos y funcionalidades core de la entrega, estamos seguros que no estamos faltos en nada. Sin embargo hay algunas cosas que nos hubiera gustado implementar pero por tema de tiempos no fuimos capaces de hacerlos:

- Toasts personalizados: Este punto creemos que no es menos importante, dado que a nivel de UX, siempre es bienvenido para un usuario una validación que establezca que una acción fue completada con éxito. Por el lado de los toasts para mostrar errores, se implementaron para mostrar los estados de error, pero no está controlado en su totalidad, pudiendo mostrar información muy técnica en algunos casos.

Diagrama de paquetes

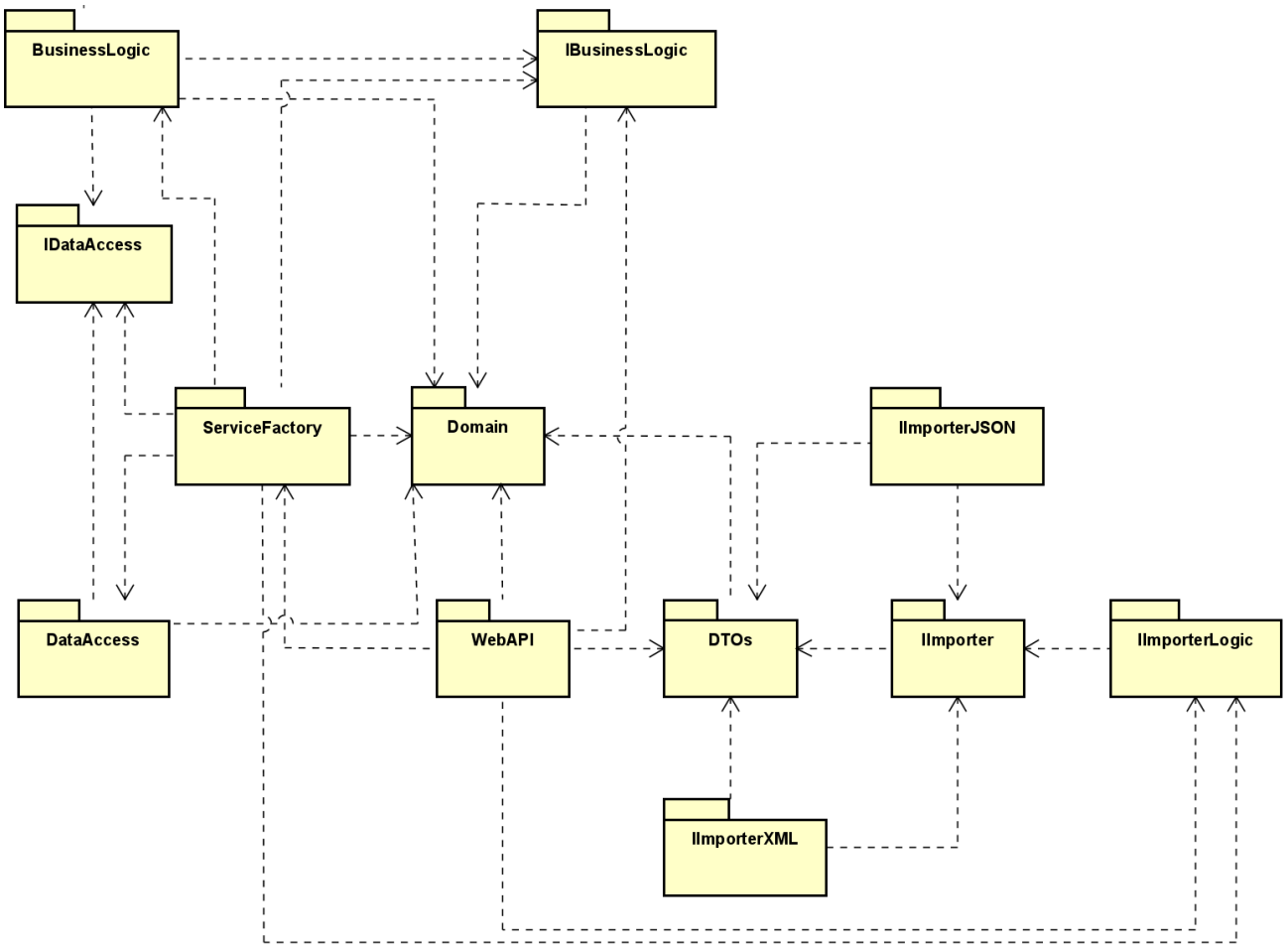
Solución básica



Este es el diagrama de paquetes principal, sin contar con los proyectos de Test ni de Service Factory para la inyección de dependencias. Al no mostrar dichos paquetes, se puede observar en un nivel más general las asociaciones que existen entre los principales paquetes de la solución, haciendo énfasis en el desconocimiento entre los más específicos, en este caso DataAccess, BusinessLogic y WebAPI.

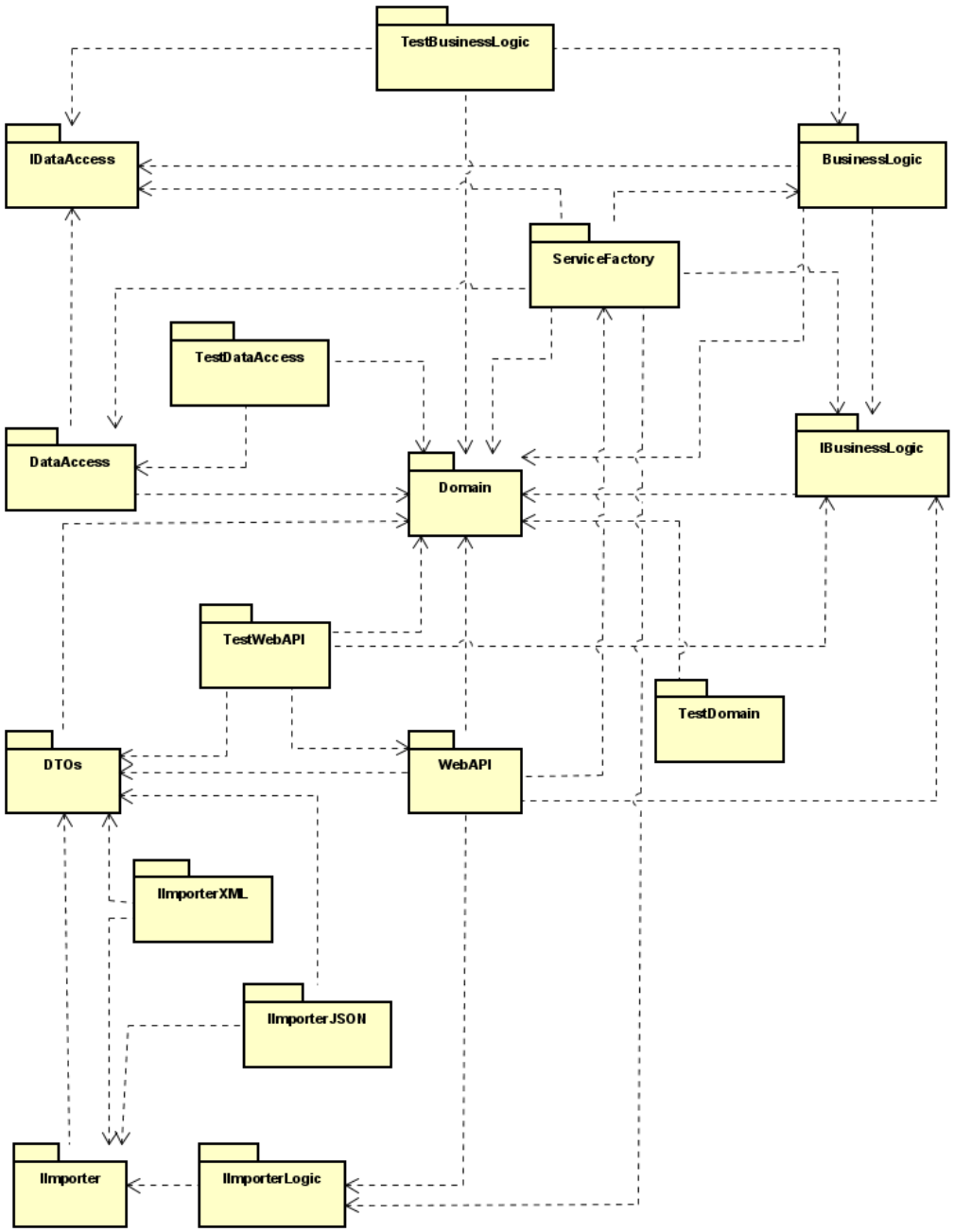
Este tipo de interacción lo que permite es tener una solución desacoplada a través del uso de interfaces, es decir, las interfaces se encuentran a un nivel más general y los paquetes más específicos las utilizan tanto para implementarlas como para tener una idea del tipo de elemento con el que van a trabajar, pero sin la dependencia de paquetes específicos entre sí.

Solución con Inyección de Dependencias a través de ServiceFactory



En este diagrama agregamos el paquete que incluye la clase ServiceFactory, a través de la cual realizamos la inyección de dependencias que es utilizada para desacoplar la solución.
A simple vista el modelado se vuelve más complejo dada la interacción entre la nueva clase de ServiceFactory para con los demás paquetes que necesita conocer, pero de esta manera evitamos que la lógica de negocio conozca el acceso de datos.
De igual manera que en el diagrama anterior se ve que el paquete BusinessLogic no interactúa directamente con ninguna clase concreta del acceso a datos, sino que con una interfaz, evitando tener que crear una instancia de cualquier clase de DataAccess.

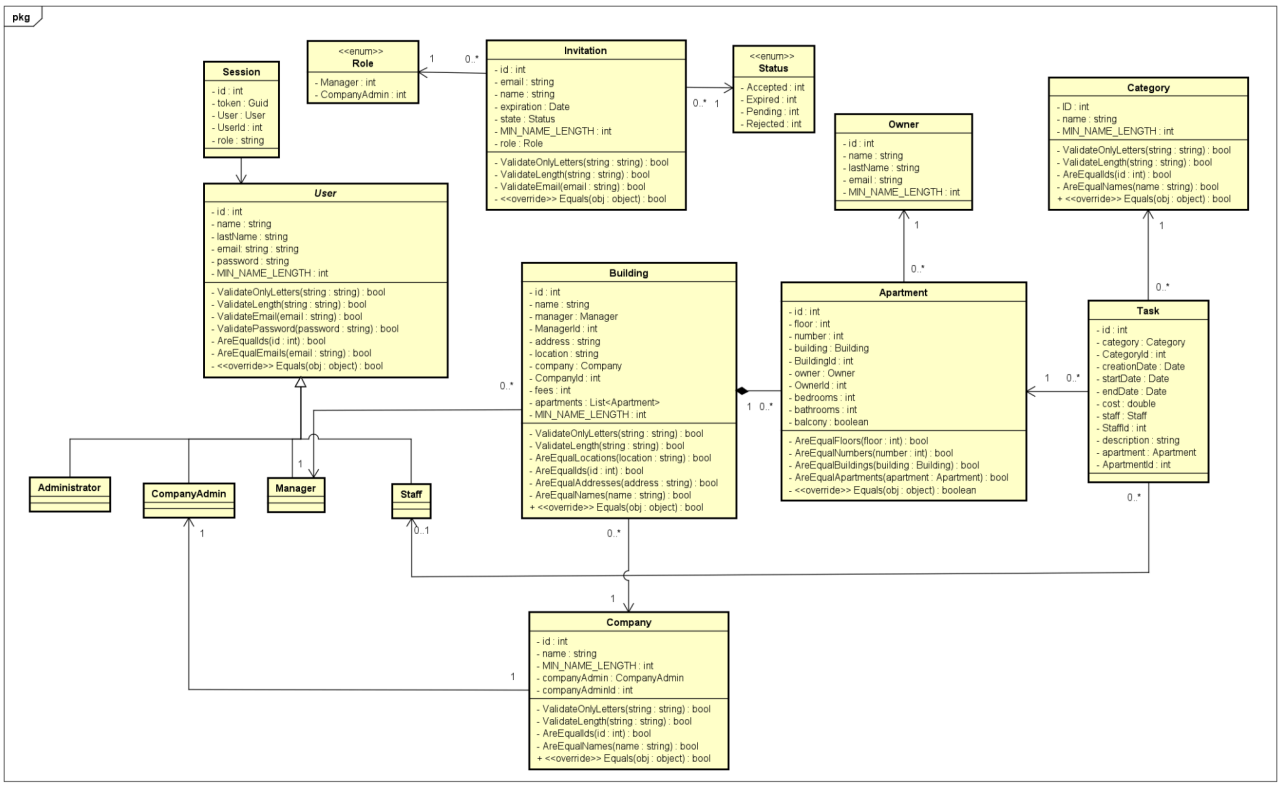
Solución completa



Como podemos ver en el diagrama de paquetes completo, las dependencias son reducidas al dominio e interfaces.

Dominio

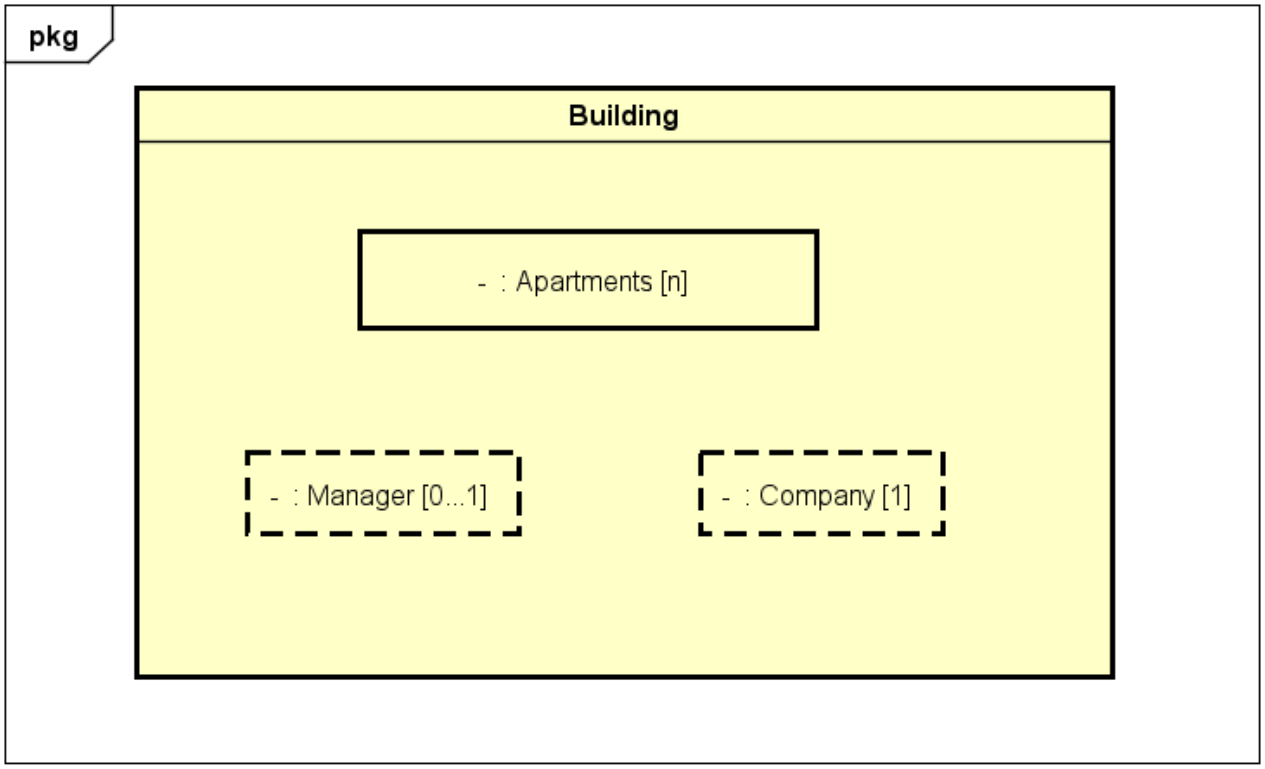
Diagrama de clases



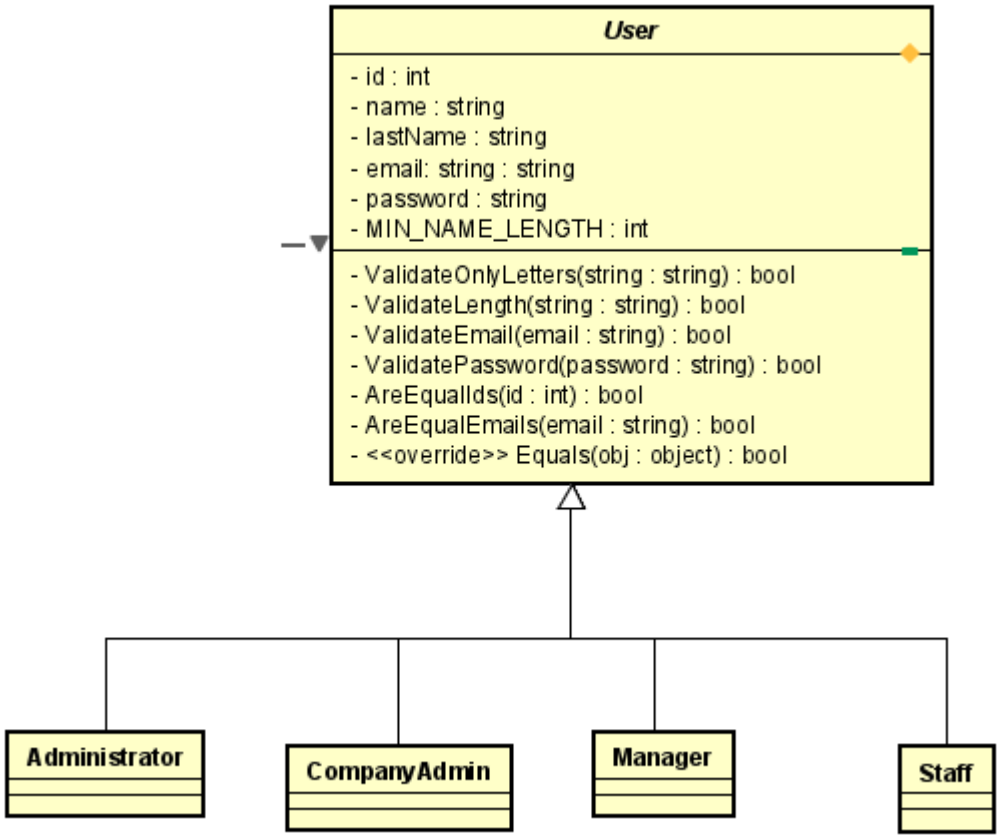
Este es el UML final del dominio, sobre el cual trabajamos durante todo el proyecto.
En el mismo se pueden observar las relaciones entre las clases y sus respectivas cardinalidades.

Respecto a la entrega anterior, los cambios observables son la adición de un nuevo tipo de usuario, el CompanyAdmin. Por otro lado se puede observar que el stff ya no cuenta con buildingId ni building, dado que preferimos que se relacione con el mismo a través de la task que contiene un apartamento perteneciente a un building. Por último, cambiamos la cardinalidad de Building y Manager, siendo que ahora no es necesario un Manager para crear un Building.

En el siguiente diagrama de estructura compuesta, se puede ver cómo está compuesto un Building, teniendo como partes la lista de apartamentos, y cómo properties, a diferencia del diagrama de la primera entrega que tenía sí o sí un Manager y una Company, ahora puede no tener un Manager.



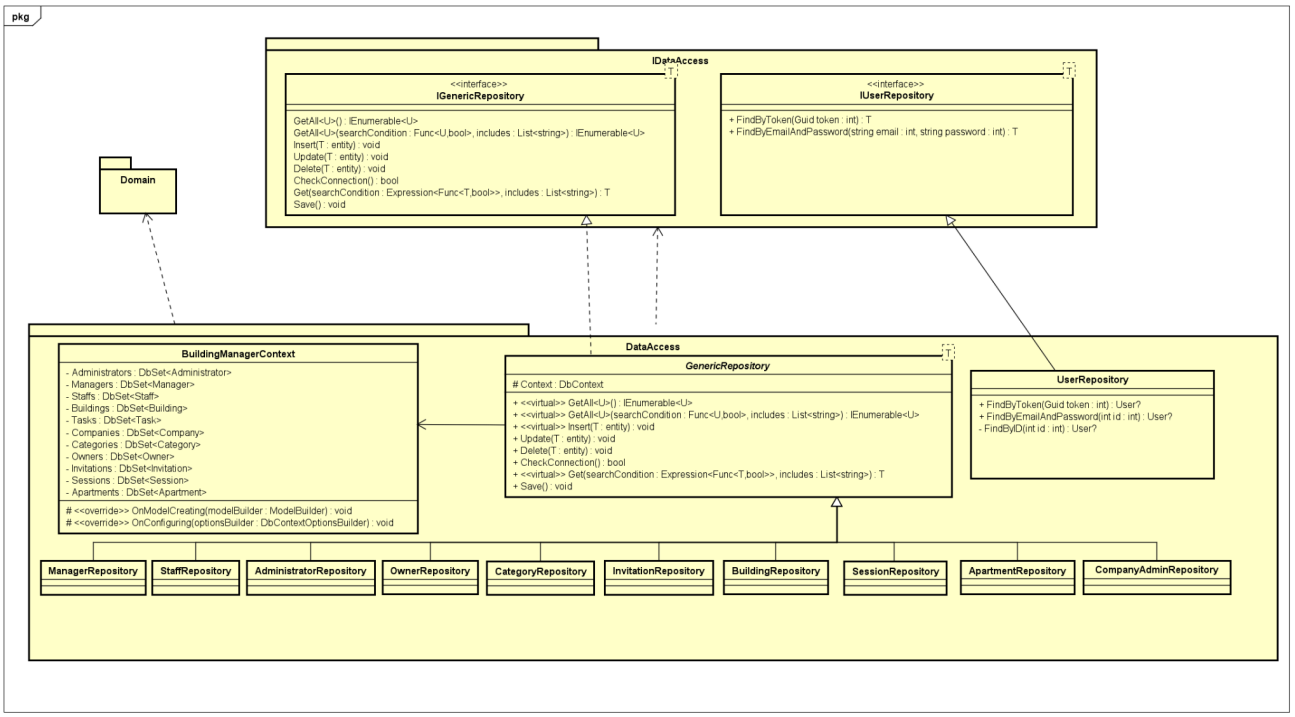
Herencia



A destacar en este diseño tenemos la herencia que planteamos para los tipos de usuario, fue una decisión de diseño que nos costó en un principio pero sabíamos que queríamos hacerlo así dada la cantidad de atributos repetidos entre estas tres clases.
A lo largo de la solución muchas funciones, principalmente en estas clases, se puede ver que comparten métodos similares, por lo que justamente se sigue viendo este patrón que nos invita a utilizar la herencia.

Data access

Diagrama de clases



Este diagrama UML muestra el paquete DataAccess, que es responsable de manejar el acceso a los datos del sistema y su interacción con los paquetes de IDataAccess y Domain. Las clases y componentes principales son:

IGenericRepository: Esta es una interfaz genérica que define los métodos básicos para interactuar con un repositorio de datos, como obtener todos los elementos, obtener elementos por condición, insertar, actualizar, eliminar, verificar la conexión, obtener la condición de búsqueda y guardar los cambios.
Esta interfaz se encuentra en el paquete IDataAccess, con el cual se comunica el paquete DataAccess, al implementar la interfaz a través de la clase GenericRepository.

GenericRepository: Esta clase implementa la interfaz IGenericRepository. Contiene métodos virtuales que pueden ser sobrescritos por clases derivadas. En nuestro código decidimos acoplarnos a las implementaciones ya dadas por el GenericRepository, por lo que las clases que heredan del mismo no tienen métodos propios más que los heredados.

BuildingManagerContext: Esta clase representa el contexto de la base de datos del sistema. Contiene propiedades DbSet para acceder a las colecciones de entidades principales, como Administradores, Gerentes, Personal, Edificios, Tareas, Compañías, Categorías, Propietarios e Invitaciones.

Además, la clase BuildingManagerContext tiene dos métodos:

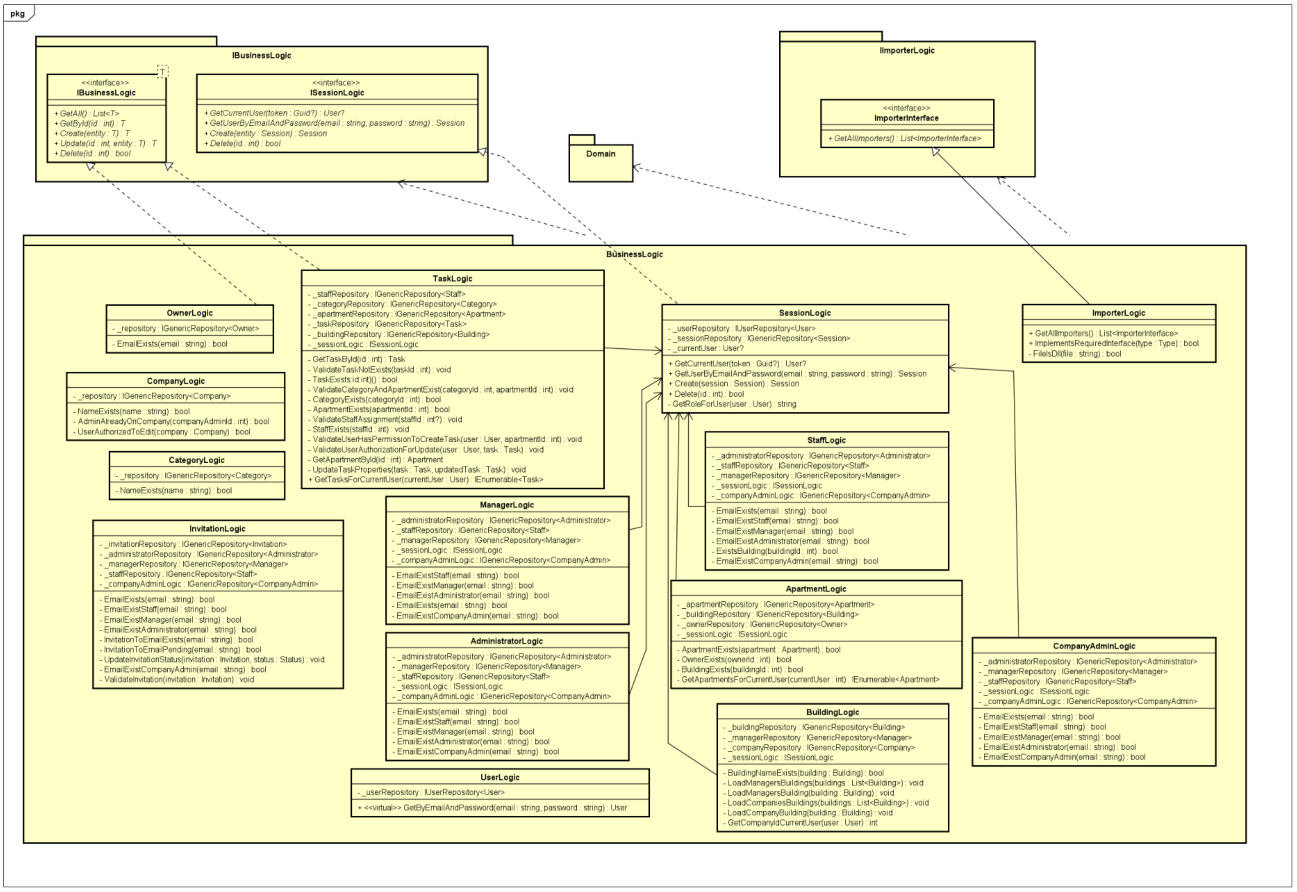
- OnModelCreating: Permite configurar el modelo de datos utilizando un ModelBuilder.
- OnConfiguring: Permite configurar las opciones del contexto utilizando un DbContextOptionsBuilder.

Repositorios específicos: El diagrama muestra una serie de repositorios concretos derivados de GenericRepository, como ManagerRepository, StaffRepository, AdministratorRepository, CompanyAdminRepository, OwnerRepository, CategoryRepository, InvitationRepository, BuildingRepository y ApartmentRepository. Estos repositorios heredan los métodos definidos .

Hablando de patrones de diseño, paquete DataAccess implementa el patrón de diseño Repositorio, donde IGenericRepository define un contrato genérico para el acceso a datos, mientras que GenericRepository y los repositorios específicos proporcionan implementaciones concretas. BuildingManagerContext actúa como el contexto de la base de datos, conteniendo las colecciones de entidades principales y permitiendo la configuración del modelo y las opciones del contexto.

Business Logic

Diagrama de clases



Este diagrama UML muestra el paquete BusinessLogic, que contiene las clases de la lógica del negocio, las cuales se encargan de las funcionalidades principales del proyecto, y su interacción con los paquetes de IBusinessLogic e Domain. Las clases y componentes principales son:

Interfaz IBusinessLogic: Define el contrato para la lógica de negocio, que incluye métodos para obtener todas las entidades, obtener una entidad por ID, crear una entidad, actualizar una entidad y eliminar una entidad.
Pertenece al paquete IBusinessLogic.

Interfaz ImporterInterface: Define el contrato para la lógica de importers, que incluye un metodo para obtener el listado de importers disponibles.

Pertenece al paquete IImporterLogic.

Interfaz ISessionLogic: Define el contrato para la lógica de sesiones, que incluye métodos para obtener el usuario actual, obtener o crear una sesión de usuario, y eliminar una sesión.
Tal como la interfaz IBusinessLogic, también pertenece al paquete IBusinessLogic. La agregamos en este paquete debido a que su uso es similar al de la otra interfaz, teniendo la definición de los métodos que utiliza una clase del paquete BusinessLogic.

Clases de lógica específicas: En general, el paquete BusinessLogic contiene clases que encapsulan la lógica de negocio para interactuar con las entidades del dominio a través de los repositorios correspondientes. Estas clases implementan la interfaz IBusinessLogic y, en algunos casos, también tienen una asociación con la clase SessionLogic para manejar la lógica de sesiones.

En el diagrama se ve que no todas tienen una flecha que simboliza la implementación de la interfaz, pero decidimos no ponerlas porque no encontramos el espacio sin que quede demasiado repleto el diagrama. Para explicar, todas las clases dentro del paquete BusinessLogic implementan los métodos de la interfaz IBusinessLogic, menos la clase SessionLogic que implementa los métodos de la interfaz ISessionLogic.

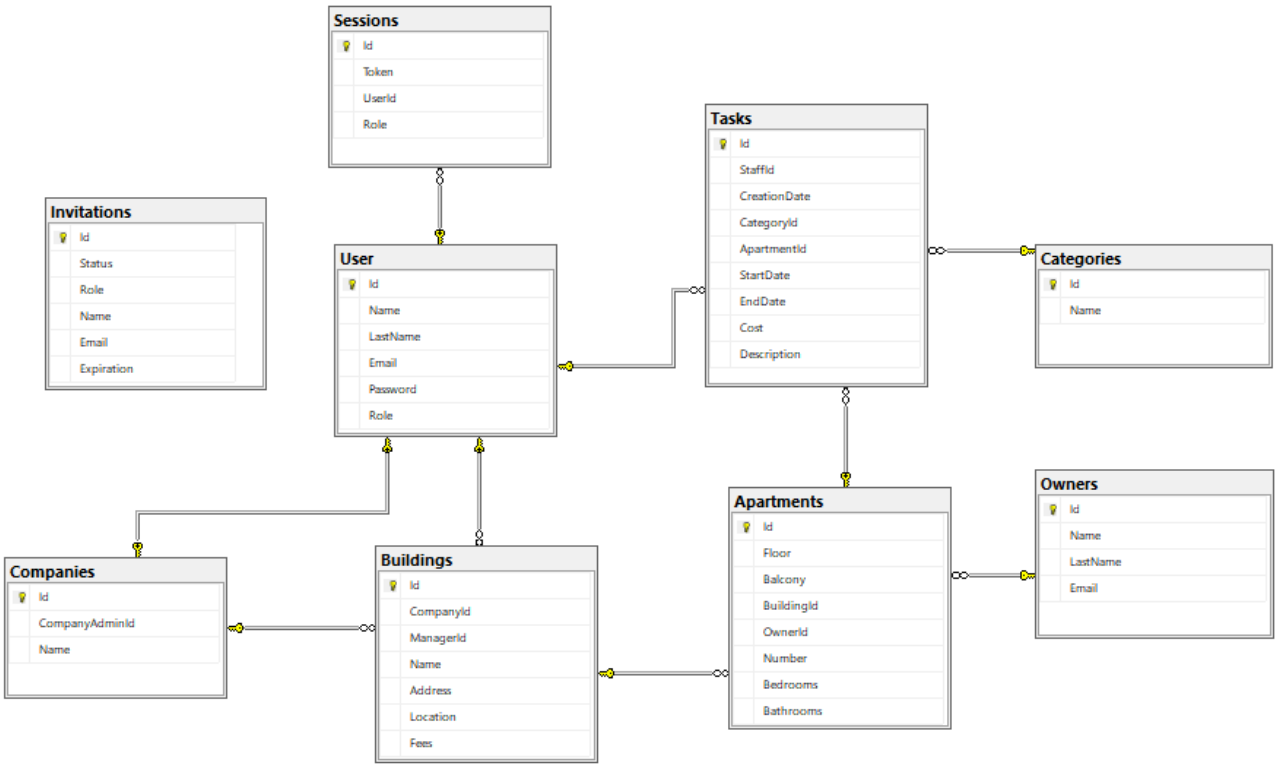
Manejo de Excepciones

Para el manejo de excepciones, en cada proyecto que las implementamos creamos una carpeta de excepciones donde pusimos todas las custom exceptions que creamos.
De esta forma evitamos generar dependencias entre proyectos que tuvieran una excepción que queremos usar en otros proyectos. También hay excepciones que consideramos que no se corresponden con algunos proyectos por lo que no nos parece bien que las conozcan, de esta forma también logramos una separación por contextos específicos. Y otro de los beneficios no menos importante de haber diseñado las excepciones de esta forma es poder tener mensajes default diferentes por proyecto en algunas excepciones.

Base de datos

Como parte de los requerimientos, se modeló la persistencia de datos para la solución utilizando Entity Framework realizando un enfoque de Code First.

En la siguiente imagen se muestra el modelo de base de datos generado en Microsoft SQL Server Management.



Para modelar la herencia entre User, Staff, Manager y Administrator, se eligió el diseño TPH (Table por Herencia), creando el atributo discriminador “Role”, con el rol de cada uno de los usuarios.

Utilización de Librerías

Language Integrated Query (LINQ)

Se utilizó Linq para realizar operaciones sobre listas, permitiendo que el código sea más legible y mantenible. Utilizando la misma sintaxis, se puede obtener los datos de forma más sencilla que iterando listas por ejemplo con un for.

Moq Framework

Para realizar pruebas unitarias en el proyecto, se utilizó Moq, un framework de simulación para .NET. Moq permite crear mocks de interfaces y clases con facilidad, lo que resulta especialmente útil para simular el comportamiento de dependencias externas durante las pruebas. Aprovechando las capacidades de Moq, se pudo diseñar y ejecutar pruebas unitarias de manera efectiva, garantizando la calidad y el correcto funcionamiento del código.

Regex

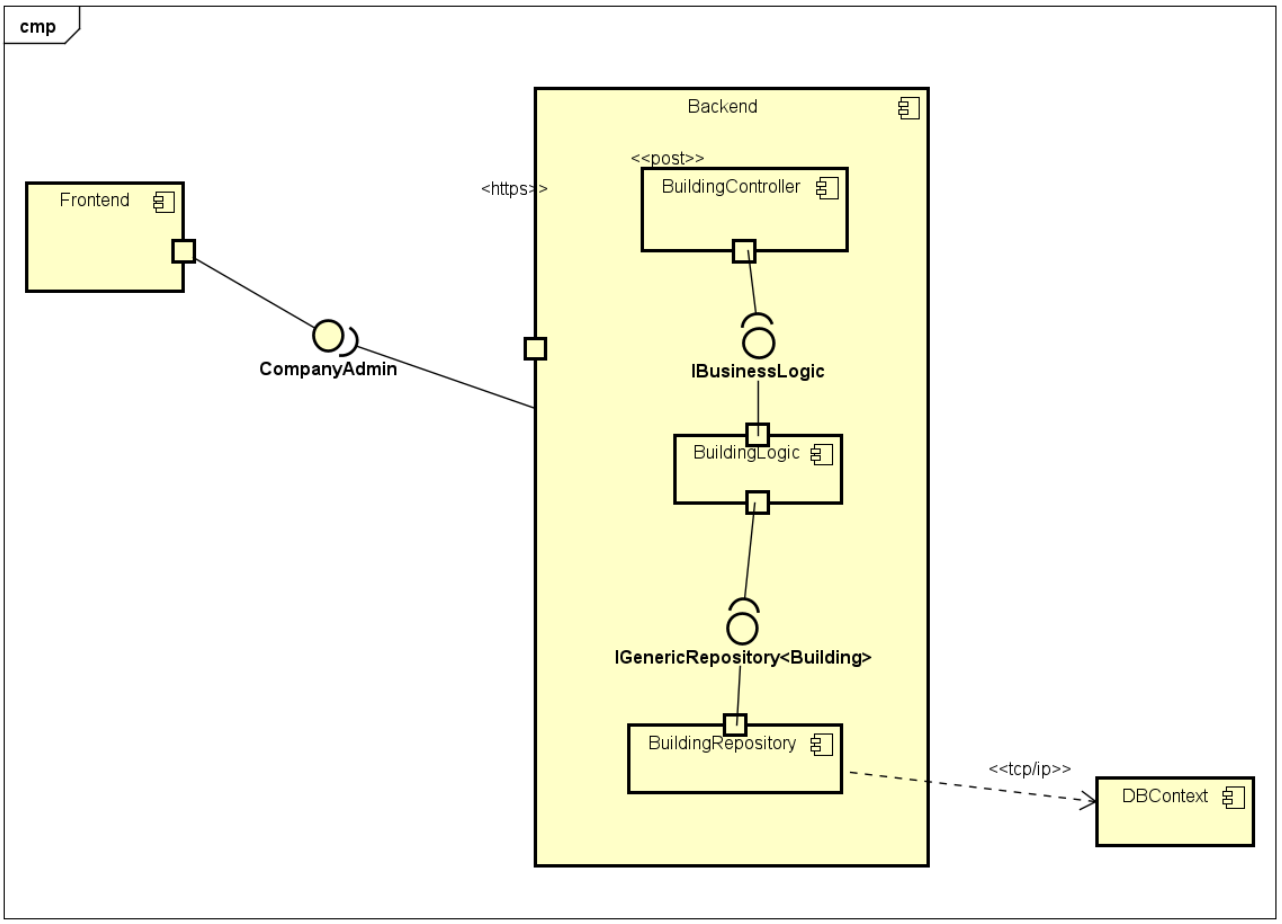
Para validar cadenas de texto en el proyecto, se empleó Regex, una herramienta que permite realizar búsquedas y manipulaciones avanzadas en strings. Utilizando expresiones regulares, se definieron patrones que permitieron validar y manipular eficazmente diferentes tipos de cadenas, como direcciones de correo electrónico, que un nombre contenga solo letras, entre otros.

Tailwind css

Esta librería nos permite usar clases en el html que tienen formatos de css aplicados a las mismas, facilitando la implementación de diseños a través de las mismas para los selectores que tratamos dentro de los componentes.

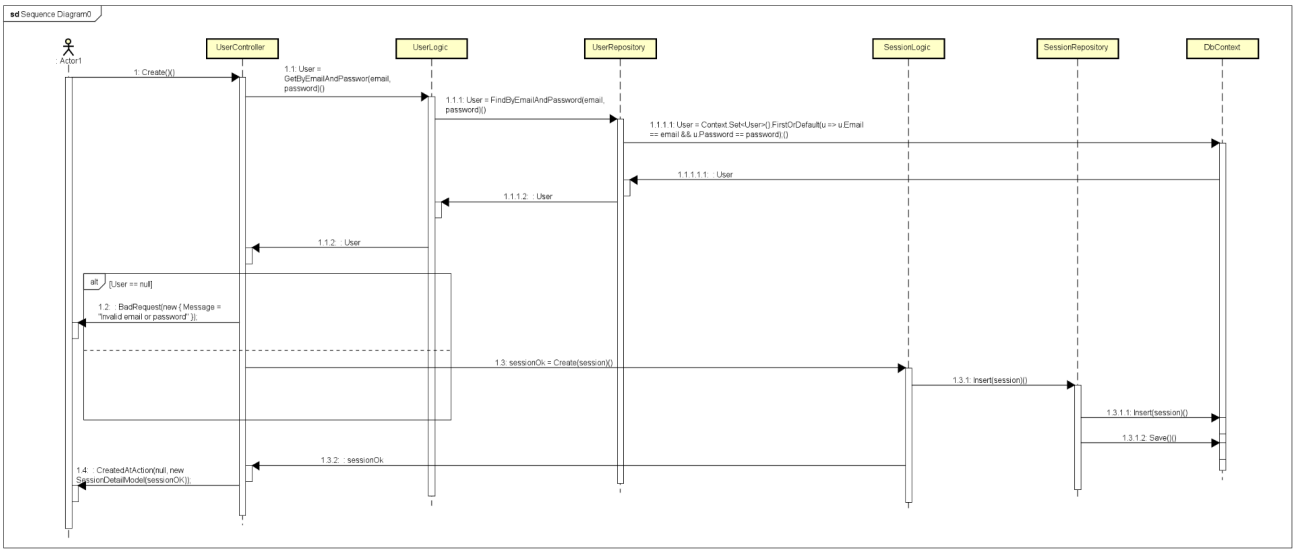
Diagramas de componentes

El siguiente diagrama de componentes muestra cómo es la interacción entre componentes del proyecto para crear un edificio desde el frontend:

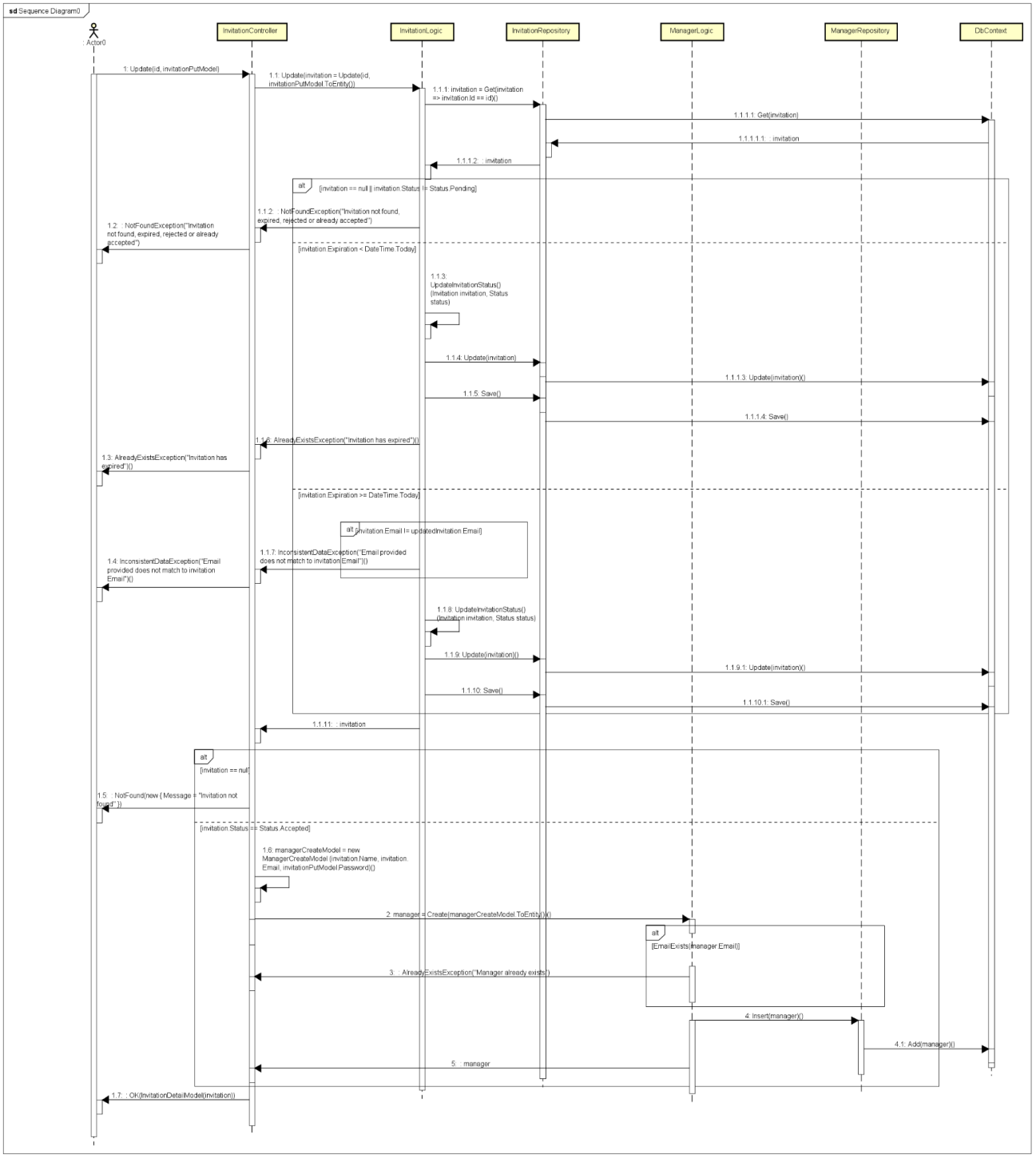


Diagramas de interacción

El siguiente diagrama muestra cómo es la interacción entre capas de los distintos paquetes del proyecto para el proceso de crear una Sesión de usuario:



En este otro diagrama se muestra como es la interacción para actualizar una invitación enviada a un email.



En este diagrama se omitió la función para validar que un email exista en la clase ManagerLogic, ya que creemos que no aportaba al ejemplo diagramado.

Diseño

Patrones

Dependency injection

Para la inyección de dependencias utilizamos la clase ServiceFactory que se encuentra en un paquete con su nombre.
Como no queremos que nuestra capa de lógica de negocio (BusinessLogic) y la de acceso de datos (DataAccess) estén fuertemente acopladas, es decir no hayan dependencias entre sí, utilizamos este patrón para invertir las dependencias, rompiendo con el acoplamiento.

En la clase mencionada de ServiceFactory registramos servicios para las diferentes clases con sus tipos concretos. Para esto utilizamos el método AddTransient en todos los servicios que registramos dado que es el que más se adecúa a nuestra solución, ya que es stateless.
Luego desde el archivo Program.cs que se encuentra en el proyecto de la web API, se hacen las llamadas a los métodos de ServiceFactory que se encargan de registrar los servicios esenciales para la lógica de negocio y el acceso a los datos de la aplicación.

De esta manera centralizamos el registro de los servicios en una sola clase, pudiendo instanciarla y ampliarla añadiendo simplemente las líneas de los nuevos servicios que queramos agregar y diciéndoles el tipo de comportamiento que van a tener.

Factory

A través de la implementación del patrón de inyección de dependencias con la clase ServiceFactory, estamos a su vez implementando el patrón Factory, donde a través de la clase mencionada y sus métodos creamos las instancias de los servicios para las clases de lógica de negocios y acceso a datos de la solución, por un tiempo determinado.

Repository

El patrón repositorio se cumple a través de la segregación de responsabilidad de el acceso a los datos, siendo únicamente el paquete de DataAccess el que contiene dicha responsabilidad, conteniendo las clases de repositorios de cada tipo y el DbContext.

Observer

A nivel de frontend en la solución lo utilizamos para suscribirnos a las respuestas que hacemos hacia los servicios HTTP.

SOLID

SRP

El principio de responsabilidad única se respeta a lo largo del proyecto al tener la solución separada por paquetes con responsabilidades únicas. El paquete de BusinessLogic se encarga de establecer la lógica del negocio para cada clase; el paquete de DataAccess se encarga de proporcionar el modelo de persistencia y acceso a los datos a través de los diferentes repositorios; y así ocurre con los diferentes paquetes que quedan. A su vez, cada clase tiene un único motivo de cambio, por ejemplo, la clases clases relacionadas a Building (BuildingLogic, BuildingRepository y BuildingController), se encargan de todas las operaciones relacionadas a la clase de Dominio “Building”). Esto se mantiene para el resto de las clases de nuestro dominio.

OCP

A través del uso de interfaces como IBusinessLogic e IDataAccess estamos abordando este principio ya que permitimos la extensión de nuestro código con nuevas clases que las implementan sin necesidad de modificar las que los implementan ya existentes.

LSP

Aplica en la herencia de usuarios. Cualquiera de los usuarios puede sustituir la clase padre dado que todos tienen los mismos atributos. Por esta razón se cumple LSP.

ISP

Este principio se cumple a través del uso de las interfaces ya mencionadas anteriormente, donde se definen métodos que son coherentes a lo que van a implementar las clases consumidoras de las mismas, además de que cada clase implementa los métodos que va a utilizar, por lo que hay segregación de interfaces.

DIP

Como tenemos clases que implementan los métodos definidos por interfaces podemos decir que en nuestra solución las capas de alto nivel dependen de abstracciones en lugar de implementaciones concretas, facilitando la flexibilidad y la extensibilidad del sistema.

Métricas

Assemblies	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
WebAPI v1.0.0.0	0	154	1.43	1	0	0
DataAccess v1.0.0.0	2	114	0.91	0.98	0.03	0.01
BusinessLogic v1.0.0.0	2	71	0.07	0.97	0	0.02
ServicesFactory v1.0.0.0	1	65	1	0.98	0	0.01
DTOs v1.0.0.0	24	34	0.12	0.59	0	0.29
Domain v1.0.0.0	80	19	1.56	0.19	0.06	0.53
ImporterXML v1.0.0.0	0	19	1	1	0	0
ImporterJSON v1.0.0.0	0	15	1	1	0	0
IBusinessLogic v1.0.0.0	35	13	0.17	0.27	0.33	0.28
IDataAccess v1.0.0.0	36	11	0.33	0.23	0.67	0.07
IImporter v1.0.0.0	6	7	0.5	0.54	0.5	0.03
IImporterLogic v1.0.0.0	3	3	1	0.5	1	0.35

Cohesión relacional

Según lo visto en clase, el resultado óptimo de esta métrica debería oscilar entre 1.5 a 4.0.
En nuestra tabla se puede ver que únicamente Domain se encuentra apenas entre estos valores, mientras que todas las demás están por debajo del estándar.
Deberíamos replantearnos las relaciones entre los paquetes, para poder cumplir con este rango.

Abstracción

Una abstracción cercana a 0, indica un paquete concreto, mientras que un valor cercano a uno indica un paquete abstracto, por lo que esta métrica tiene sentido, ya que los paquetes que tienden 1 son los que exponen interfaces, mientras que los que tienen como valor 0, o cercano a 0 las implementan.

Inestabilidad (SDP)

Valor cercano a 0 - máxima estabilidad porque no se depende de otros paquetes. Se debe intentar extender haciéndolo abstracto

Valor cercano a 1 - máxima inestabilidad porque depende de otros paquetes. Este paquete es fácil de cambiar debido a que impacta en pocos paquetes.

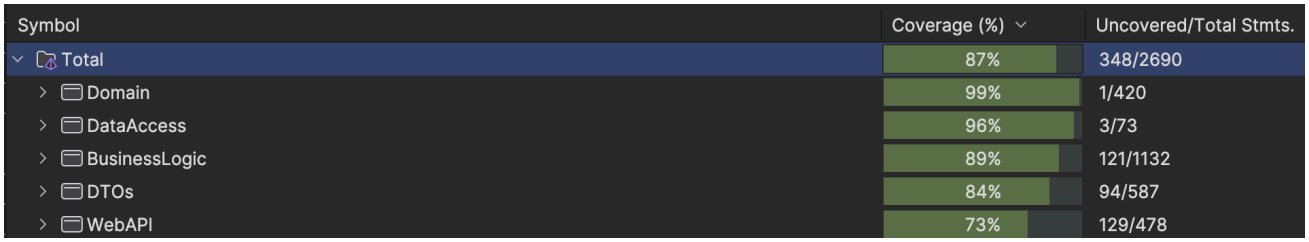
Una regla importante es que los paquetes no deben depender de paquetes más inestables que ellos. En base a esto podemos observar que los únicos casos en los que sucede es en el de IImporter que es más estable que DTOs y sin embargo depende de este; el otro caso es uno límite dado que tienen el mismo nivel de inestabilidad, ServiceFactory depende de DataAccess.

Se puede concluir que no se cumple SDP del todo, dado que tenemos el caso de IImporter (inestabilidad = 0.54) dependiendo de DTOs (inestabilidad = 0.59).

ADP

En nuestro caso siempre se cumple este principio dado que el IDE Visual Studio nos advierte, al intentar compilar la solución, de dependencias circulares existentes.

Cobertura de pruebas automáticas



Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	87%	348/2690
Domain	99%	1/420
DataAccess	96%	3/73
BusinessLogic	89%	121/1132
DTOs	84%	94/587
WebAPI	73%	129/478

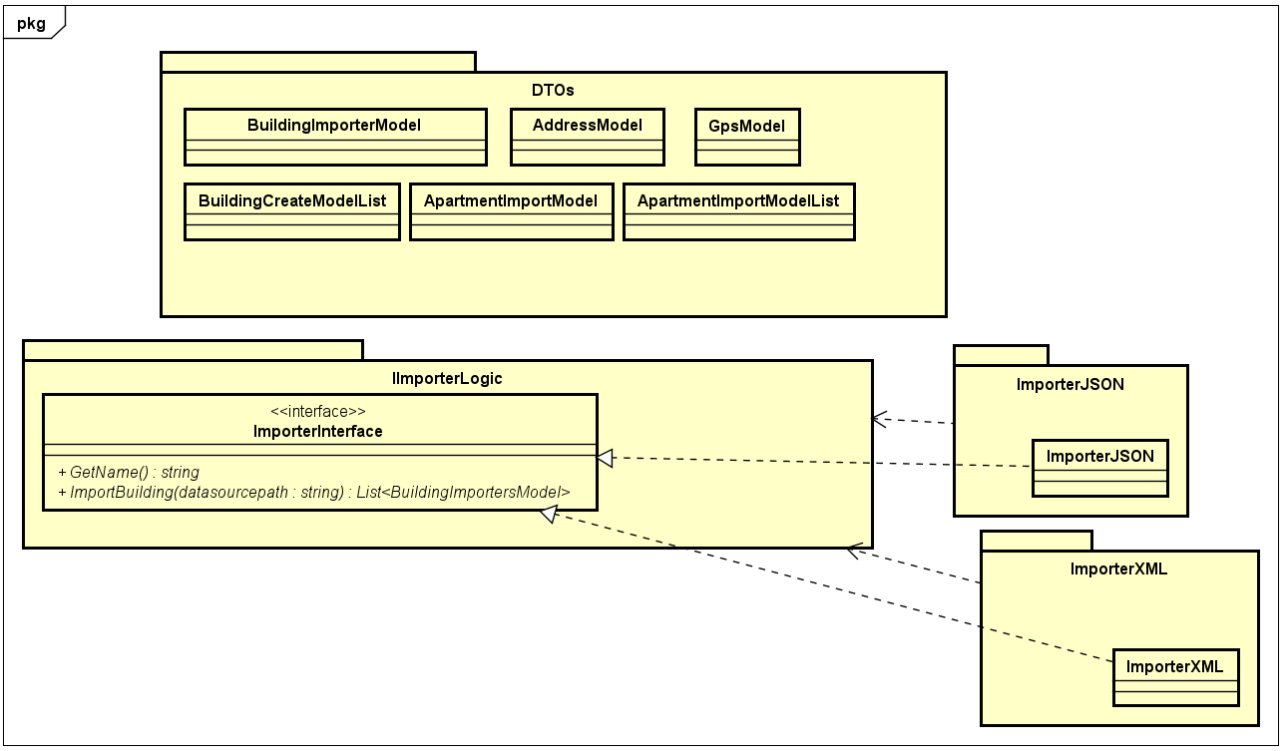
Respecto a la primera entrega, se puede observar una disminución en el porcentaje de cobertura de las pruebas automatizadas.
Esto se debe a que no nos basamos en TDD para la implementación de las nuevas funcionalidades del backend. Por lo que pueden haber quedado líneas de código sin cubrir.
La decisión de no seguir esta práctica se debe a que a medida que desarrollamos fuimos probando lo que hacíamos, resultando en una obtención de resultados más veloz para el tiempo acotado que manejamos para la implementación de los nuevos requerimientos y la modificación de antiguas funcionalidades.

Somos conscientes de que es una buena práctica para el desarrollo TDD, dada su alta cobertura de código, que proporciona una solución más estable. Pero por motivos de tiempo no fuimos capaces de aplicarlo.

Importación de edificios

Para la importación de edificios creamos el siguiente paquete IImporterLogic, el cual cuenta con una interfaz llamada ImporterInterface la cual define el contrato para que las dlls externas se acoplen a ella.

A su vez, por cada formato que se quiere importar, por ejemplo JSON y XML, se creó un nuevo paquete, que implementa la interfaz mencionada anteriormente.



Para soportar el formato de aulas, se crearon los DTOs que se ven en el diagrama.

En el controller de importar, se hace un pasaje de BuildingImporterModel, a BuildingCreateModel, que es el formato utilizado en el endpoint de crear edificios.

Especificación de la API

Agregamos los endpoints correspondientes para la implementación del Importer.

La misma está dentro de la carpeta de documentación bajo el nombre de:
Especificación API.xlsx

Creación de usuario Administrador

Para ingresar un usuario Administrador para comenzar a usar el sistema, utilizando SQL Server Management Studio, se debe ejecutar la siguiente consulta:

```
INSERT INTO dbo.[User] (Name, Lastname, Email, Password, Role)

VALUES ('Semilla', 'Semillita',
'elsemillero@semillardosemillero.com', 'Test.1234',
'Administrator');
```