Universidad ORT Uruguay

Facultad de Ingeniería

https://github.com/IngSoft-DA2-2023-2/276280-243218

1er Obligatorio Diseño de Aplicaciones

Descripción del diseño

Gonzalo Loureiro – 243218

Santiago Alfonso – 276280

02 de Mayo de 2024

Índice

Descripción General del Proyecto

El proyecto se centra en el desarrollo de una aplicación dedicada a la administración de edificios, permitiendo a los usuarios ejercer diferentes roles que corresponden a sus responsabilidades y privilegios dentro del sistema. La aplicación está diseñada para garantizar que cada funcionalidad sea accesible únicamente por los usuarios autorizados.

Roles y Funcionalidades

Administradores

Los administradores son el eje central de la gestión de la plataforma. Tienen acceso exclusivo a funciones críticas que permiten configurar y supervisar todas las operaciones importantes del sistema. Estos incluyen la capacidad de manejar las categorías de los servicios de mantenimiento y de enviar invitaciones personalizadas para nuevos usuarios que asumirán el rol de encargados.

Encargados (Managers)

Los encargados o managers tienen la tarea de coordinar el mantenimiento de los edificios. Utilizan el recurso de solicitudes del encargado para crear, asignar y supervisar las solicitudes de mantenimiento necesarias. Este rol es clave para garantizar la eficiencia operativa y la adecuada atención de las necesidades de cada apartamento dentro del edificio el cual manejan.

Personal de Mantenimiento (Maintenance)

El personal de mantenimiento se encarga de ejecutar las tareas de reparación y mejoras en los edificios. Este grupo tiene sus propios perfiles dentro del sistema, que pueden ser gestionados por los encargados, y son asignados a las solicitudes creadas por los mismos para llevar a cabo las labores necesarias.

Recursos Principales de la Aplicación

Edificios y Apartamentos

- **Edificios**: Se administra toda la información relacionada con los edificios, desde su ubicación y datos del constructor hasta los gastos comunes y la asignación de un encargado.
- **Apartamentos**: Se controla toda la información detallada de cada unidad, incluyendo su distribución, propietarios y características específicas.

Propietarios de Apartamentos

Este recurso maneja la información de los dueños de los apartamentos, facilitando el registro, la actualización y la eliminación de datos para mantener una gestión organizada de la propiedad.

Registro y Gestión de Sesiones

El recurso de gestión de sesiones es esencial para controlar el acceso a la aplicación. A través de este, los usuarios pueden autenticarse y recibir autorización para interactuar con los recursos adecuados a su rol, asegurando un entorno estructurado.

Invitaciones

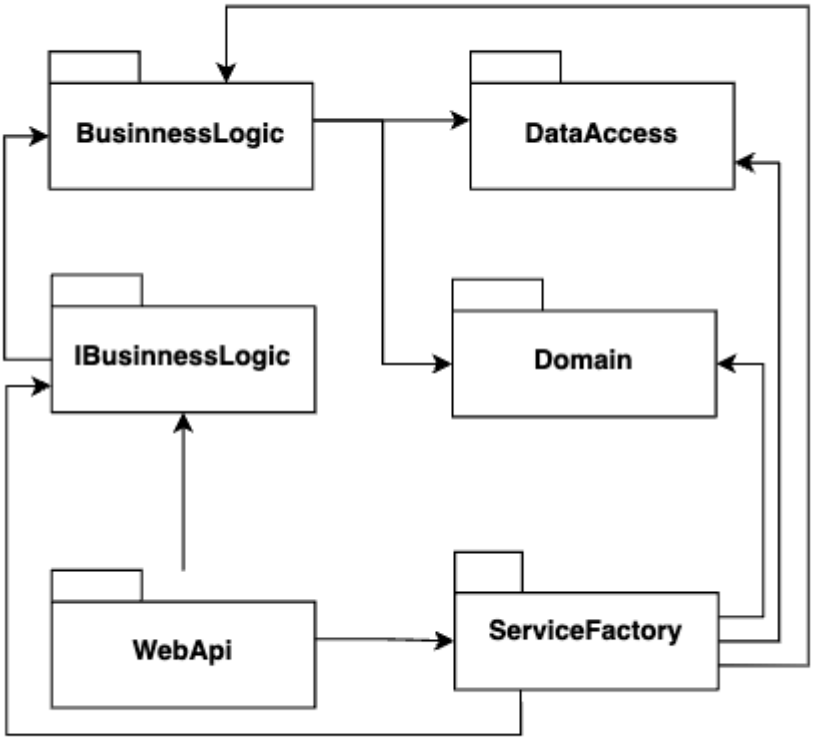
Este recurso permite a los administradores enviar invitaciones a potenciales nuevos encargados, siendo necesaria la aceptación de estas para formalizar su creación y acceso al sistema.

Reportes

Los encargados tienen la capacidad de generar informes detallados sobre las actividades de mantenimiento y otros aspectos relevantes de la administración de los edificios.

UML

Paquetes

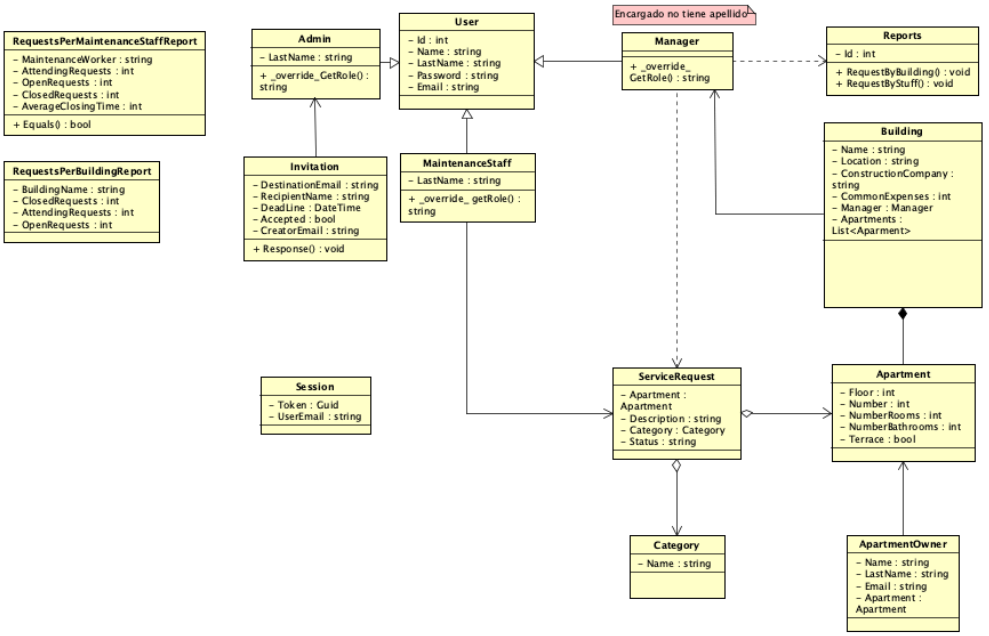


Clases

A continuación se presentan los diagramas de clase de cada paquete y como estas se relacionan entre ellas.

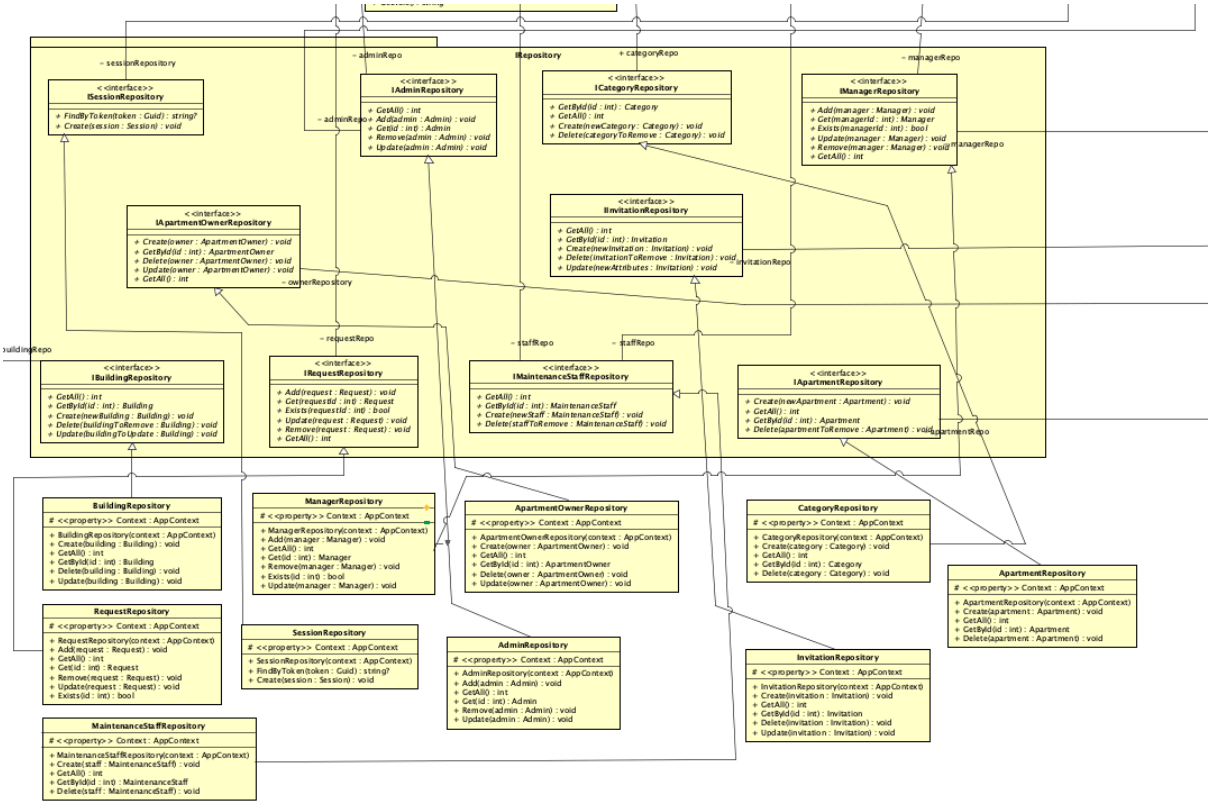
Dominio

En esta sección se presenta un diagrama de cómo las clases del Dominio se relacionan entre ellas y con aquellas clases en otros paquetes.



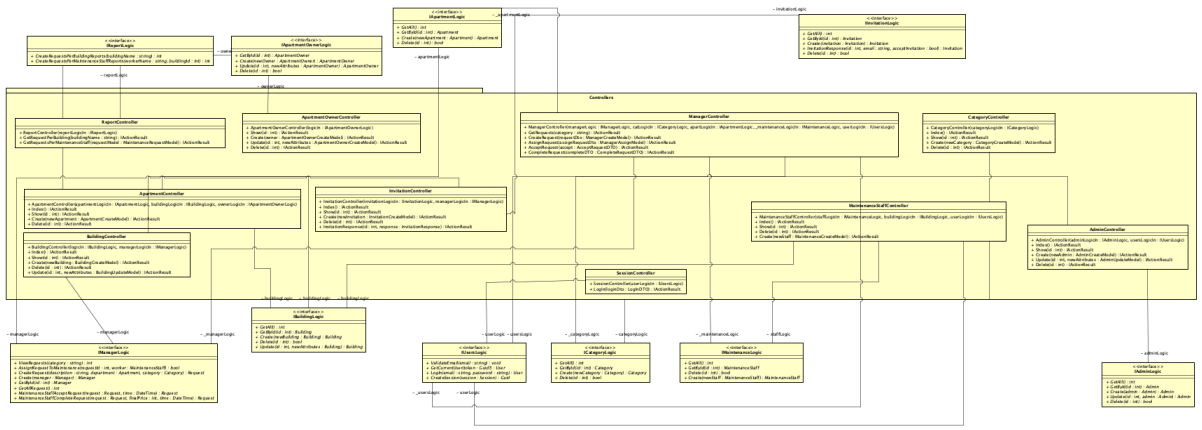
DataAccess

En esta sección se presenta un diagrama de cómo se relacionan los Repositories con los IRepositories, luego en BusinessLogic se muestran las relaciones entre IRepositories y BusinessLogic



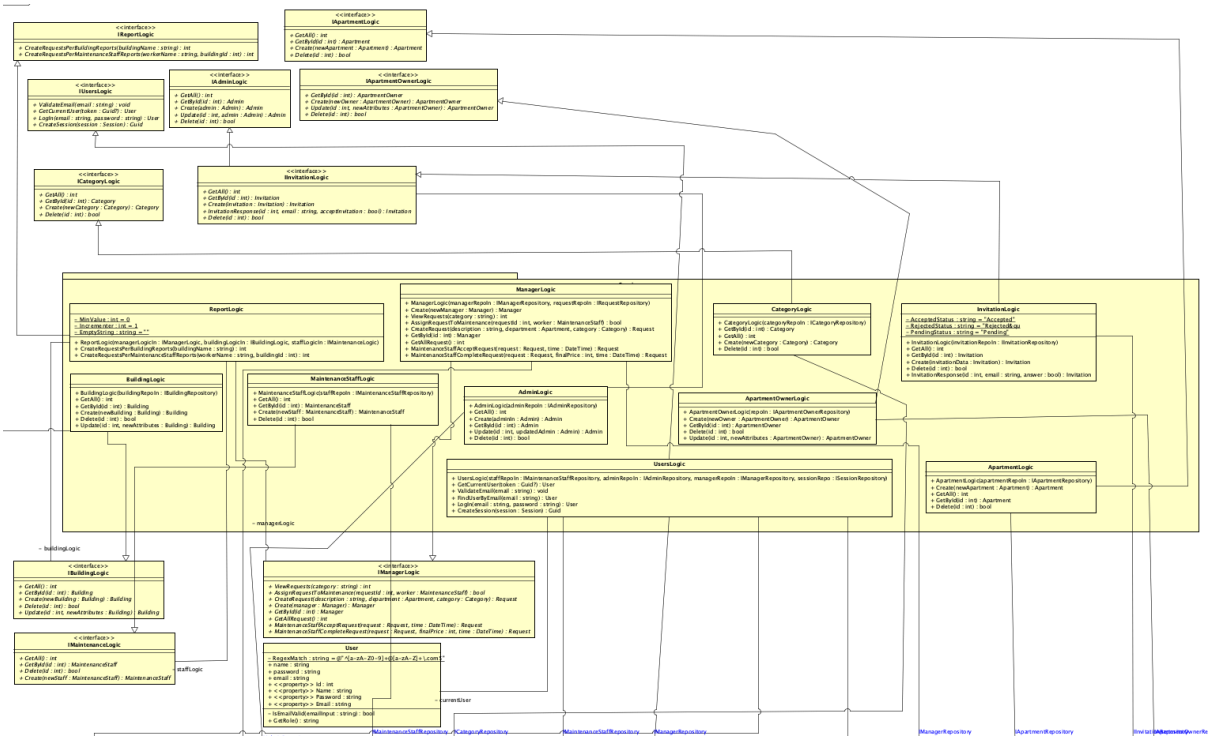
WebApi

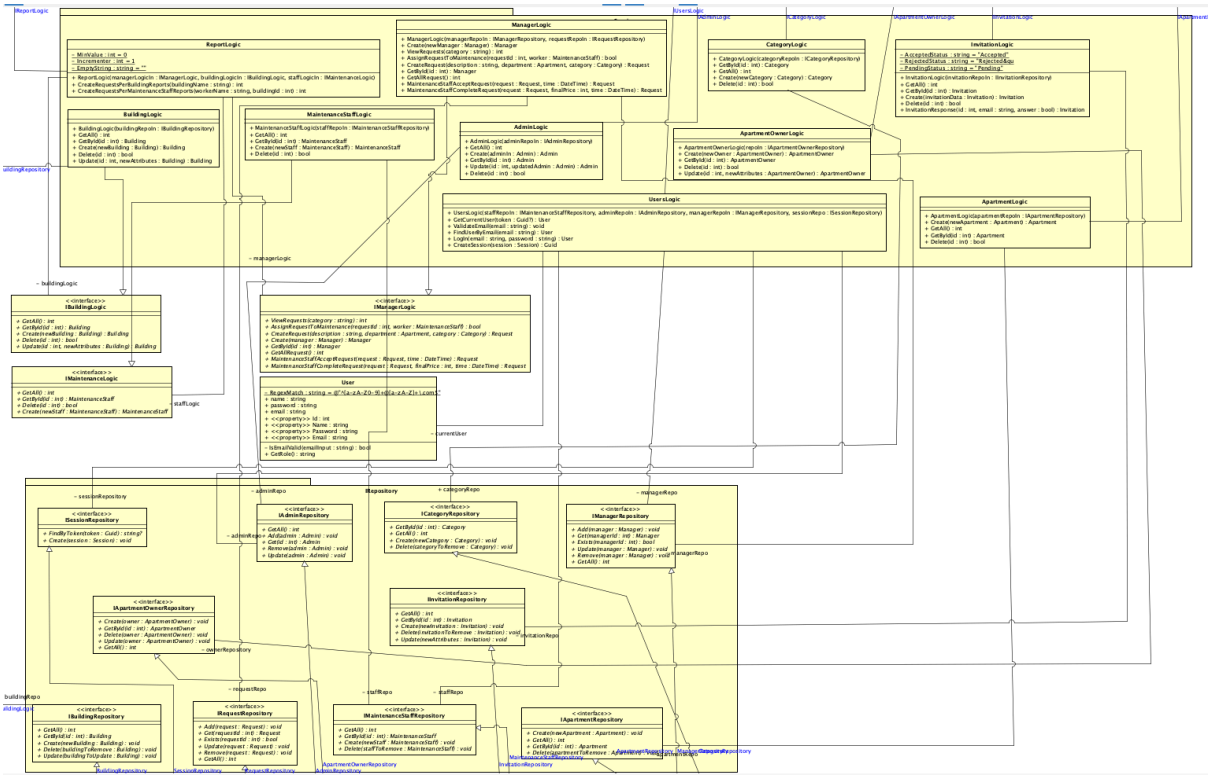
En esta sección se presenta un diagrama de cómo las clases de WebApi (Controladores) se relacionan con aquellas clases en otros paquetes.



BusinessLogic

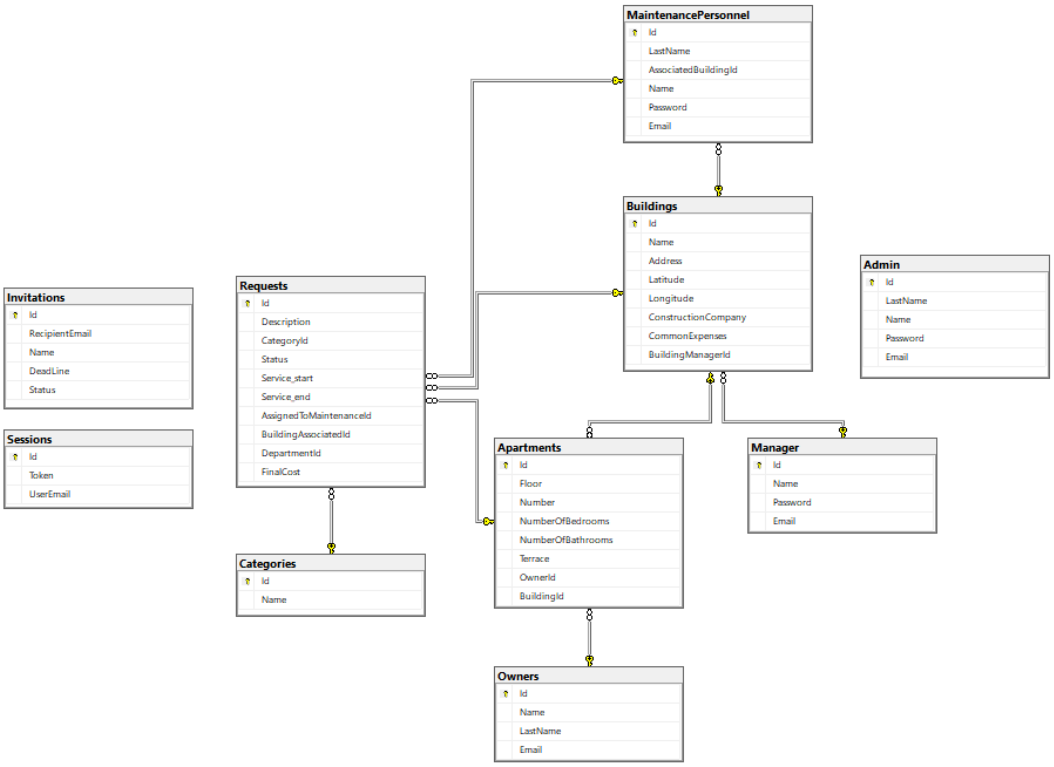
Se muestran las relaciones entre DataAccess y BussinesLogic y IBusinessLogic





Descripción de Tablas

Para poder persistir tanto los departamentos, edificios, propietarios como todos los tipos de usuarios (Encargados, Admin, Mantenimiento) se creó una base de datos en SQL Server utilizando la potencia de Entity Framework. Esto nos permite mapear las clases y sus atributos a tablas en el servidor SQL.



1. Invitations (Invitaciones)
    - Almacena información sobre las invitaciones enviadas a los usuarios, incluyendo el correo electrónico del destinatario, el nombre de la invitación y su estado.
2. Sessions (Sesiones)
    - Registra las sesiones de los usuarios con detalles como el ID de la sesión y el correo electrónico del usuario asociado con la sesión.
3. Requests (Solicitudes)
    - Contiene datos relacionados con solicitudes de servicio o mantenimiento, como la descripción, estado, inicio y fin de los servicios, y los costos asociados.
4. Categories (Categorías)

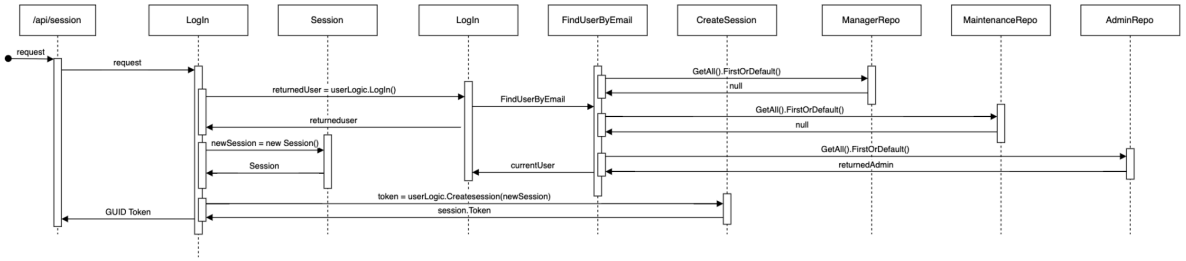
- Una lista simple de diferentes categorías que podrían relacionarse con tipos de solicitudes, edificios u otras clasificaciones.

5. Maintenance Personnel (Personal de Mantenimiento)
    - Detalles sobre el personal de mantenimiento, incluyendo su apellido, edificio asociado, credenciales de acceso y detalles de contacto.
6. Buildings (Edificios)
    - Información sobre los edificios, incluyendo nombre, dirección, coordenadas geográficas, empresa constructora asociada, gastos comunes y el ID del gerente del edificio.
7. Apartments (Apartamentos)
    - Detalles de los apartamentos individuales dentro de un edificio, tales como piso, número, cantidad de dormitorios, baños y el propietario asociado.
8. Owners (Propietarios)
    - Información sobre los propietarios de los apartamentos incluyendo sus nombres y direcciones de correo electrónico.
9. Manager (Gerente)
    - Contiene datos sobre los gerentes de edificios, similar al personal de mantenimiento, con campos para nombre, contraseña y correo electrónico.
10. Admin (Administrador)
    - Tabla para los usuarios administradores que gestionan el sistema, incluyendo campos para nombre, apellido, contraseña y correo electrónico.

Diagramas de Interacción

Autenticación

El diagrama presentado muestra cómo sería la autenticación de un usuario de tipo Admin.

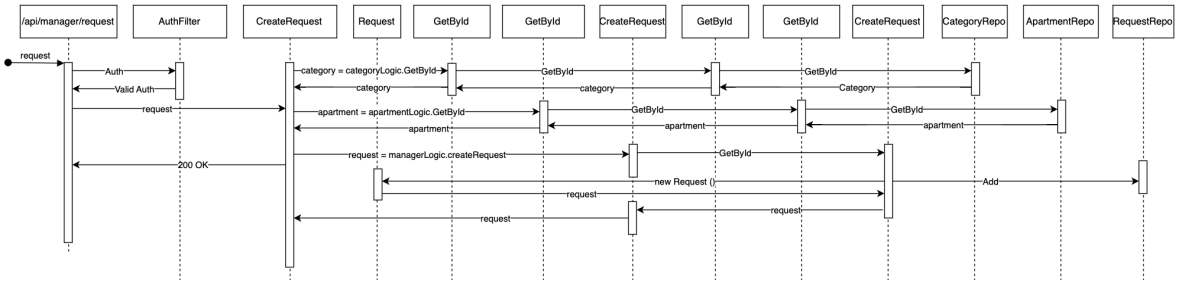


1. **/api/session** - Este es el punto final de la API para la gestión de sesiones, manejando inicios de sesión de usuarios segun roles.
2. **Login y Session** - Estos componentes gestionan la autenticación y el estado de la sesión.
3. **FindUserByEmail** - Un método utilizado para recuperar datos del usuario basado en su correo electrónico, durante el inicio de sesión.
4. **CreateSession** - Después de un inicio de sesión exitoso, este método maneja la creación de una nueva sesión para el usuario.
5. **ManagerRepo, MaintenanceRepo, AdminRepo** - Estos repositorios son revisados para verificar que el usuario que quiere crear la sesión exist.
6. **GUID Token** - El resultado final, un token (GUID), que el cliente utiliza para autenticar solicitudes subsiguientes.

En este proceso, el usuario inicia sesión (autenticado por correo electrónico), se crea una sesión, y se genera un token para la gestión de la sesión, facilitando interacciones seguras del usuario con la API.

Creación de Request

El diagrama presentado muestra cómo sería la la creación de una Request por parte de Encargado.



1. **/api/manager/request** - Este es el punto final de la API que inicia el proceso de creación de la solicitud.
2. **Auth Filter** - Filtro encargado de autenticar al usuario, en caso de que el token pasado en el header *"Authorization"* sea valido (Token de Manager) se permite crear la request.
3. **CreateRequest** - La acción inicial donde se crea la solicitud.
4. **Request** - Este componente maneja el procesamiento y la logística de una solicitud.
5. **GetById** - Métodos dentro de varios componentes (como Category y Apartment) que recuperan datos basados en un identificador.
6. **CreateRequest en ManagerLogic** - Responsable de la creación real de una solicitud después de recopilar los datos necesarios.
7. **CategoryRepo, ApartmentRepo, RequestRepo** - Repositorios donde se almacenan y recuperan datos, involucrando operaciones como obtener datos por ID y añadir nuevos registros.
8. **200 OK** - La respuesta final devuelta al cliente, indicando que el procesamiento fue exitoso.

El flujo implica consultar datos existentes de repositorios (Category y Apartment) para compilar los detalles necesarios para crear una solicitud, que luego se almacena en el repositorio de solicitudes.

Justificación de Diseño

Estructuramos el sistema en cuatro paquetes principales: WebApi, BusinessLogic, Domain y DataAccces. El dominio es el núcleo de nuestra aplicación y consta de clases esenciales que personifican las funcionalidades del software, tales como Admin, Building, Category, entre otras. Además, designamos servicios específicos para cada una de estas clases, los cuales se encuentran en el paquete BusinessLogic. Estos servicios tienen la responsabilidad de realizar validaciones que no son inherentes a las clases, interactuar con la base de datos y ser capaces de ejecutar la lógica de negocio que solicita el cliente.

El paquete WebApi es contenedor de los controllers de la aplicación, los cuales manejan la menor cantidad posible de lógica, siendo este último el trabajo asignado al paquete BusinessLogic y sus clases adheridas.

Creamos interfaces con el objetivo de acceder a la memoria e interactuar con ella, mediante estas interfaces los servicios logran comunicarse con los repositorios de la base de datos y realizar las funciones necesarias. Dentro del paquete de acceso a datos tenemos los repositorios que se encargan de realizar las operaciones de manejo a memoria, la cual se guarda en el AppContext que es nuestra base de datos. Para optimizar el manejo de datos y mejorar la legibilidad y mantenimiento de código se introdujeron estructuras auxiliares en la lógica de negocio, como lo son las estructuras(structs) y enumeraciones (enums). Por último, hemos creado clases de excepciones específicas con el objetivo de gestionar eficientemente los errores.

En la lógica de negocio, los servicios se encargan de las operaciones esenciales, como “crear”, “modificar”, “listar” y “eliminar” para cada clase a través de las interfaces que comunican a la lógica con la base de datos. Además, se encargan de efectuar las validaciones adecuadas, excluyendo aquellas que están relacionadas con la correcta estructura de los parámetros ingresados por el usuario, tarea que recae sobre los propios objetos. Todas las operaciones que tengan que ver con la memoria se llevan a cabo a través de estos servicios por lo que son utilizados frecuentemente. Cada servicio se instala un atributo referente a la interfaz que

corresponde a la clase que maneja, que hace referencia a los repositorios de la base de datos.

Paquete WebApi: En este decidimos crear un controlador por cada servicio, de forma de que cada controlador tuviera que usar la menor cantidad de servicios posibles, para no sobrecargar el mismo, generando una cohesión baja. En general la mayoría cumplio con este requerimiento, cumpliendo en gran parte con el principio de responsabilidad única, excepto el ManagerController, el cual por una mala comunicación entre los integrantes terminó con más trabajo del que debía. La mejor solución hubiera sido crear un controlador para aceptar, completar y visualizar las solicitudes de servicio, de esta forma ManagerController y el nuevo controller cumplirían con el principio de responsabilidad única.

Los controladores se relacionan con interfaces de la lógica de negocio, estas interfaces se encuentran dentro de un paquete llamado IBusinessLogic, el cual lo creamos con la idea de romper con la dependencia de “WebApi usa BusinessLogic”, ya que los módulos de alto nivel no deben de depender de los módulos de bajo nivel, entonces aplicando el principio de inversión de dependencias, logramos que esto no suceda, provocando que WebApi dependa de una abstracción y los servicios implementen dicha abstracción.

También implementamos DTO en los controllers para mejorar la seguridad de la aplicación al limitar el acceso a los datos y la lógica de la aplicación. Esto permite restringir el acceso a datos sensibles y limitar la capacidad de un usuario malintencionado para modificar el comportamiento de la aplicación. Creamos filtros para las excepciones y para la autenticación, de esta forma evitamos el código repetido, generando buen clean code y evitando la baja cohesión.

Paquete BusinessLogic: En este paquete utilizamos lo que viene siendo el patrón controlador, generando un controlador de eventos(servicio) por cada caso de uso, manejando los eventos relacionados a ese caso de uso. Todos los servicios implementan una interfaz, la cual WebApi usa. Esto nos ayuda ya que mediante polimorfismo ocultamos los servicios, y podemos modificarlos cómodamente sin

tener que impactar directamente en el paquete de los controladores. Los servicios se encargan también de crear,modificar y eliminar los elementos del sistema, ya que contiene la información necesaria para hacerlo, así como la comunicación directa con los repositorios. Siguiendo el patrón Creator de GRASP, esto sería lo correcto, de otra forma podríamos hacerlo en los repositorios, pero sería un pasamanos de datos que no sería muy útil.

También un dato importante es que BusinessLogic no recibe en ningún momento DTOs ya que esto generaría una dependencia con WebApi innecesaria, entonces creamos una función llamada ToEntity(), el cual en general, convierte los DTOs en modelos del sistema que son manejables por la lógica de negocio.

De manera similar al paquete de los controladores, los servicios utilizan interfaces de los repositorios, los cuales se encargan de interactuar con la base de datos. Esto por lo ya hablado del principio de inversión de dependencias. En este caso nos pareció muy útil, ya que si se cambia la base de datos, la lógica de negocio no se debería de ver afectada, haciendo que nuestro código sea modificable y extensible, cumpliendo el principio de responsabilidad única y el principio Open closed.

Agregamos un servicio para la creación de los reportes, ya que si lo hubiéramos creado en el servicio del manager, la clase tendrá una caída total de cohesión y una subida de acoplamiento. Para solucionar esto, pensando en el patrón de fabricación pura, creamos un servicio de reportes, el cual con toda la información requerida crea los mismos.

Paquete DataAccess: Las clases pertenecientes a la misma, sólo realizan operaciones CRUD sobre la base de datos, intentamos que en ningún momento tuvieran que realizar lógica sobre el manejo de datos. Podríamos haber creado un repositorio genérico, pero aun así tendriamos que crear repositorios prácticamente vacíos que implementen una interfaz única, o sea tendria mas funcionalidades de las que necesito. Nuestra solución fue crear un repositorio por cada modelo que queríamos persistir, teniendo cada uno sus funcionalidades. El lado negativo de esta solución es que hay código que podría ser “ahorrado”, ya que son muy similares.

En cuanto a creación de tablas nos pareció que la mejor opción sería, la utilización de TPC para la herencia en usuario.

Ventaja: No genera valores con null, no se necesita joinear una tabla intermedia como en TPT, permite separar las entidades y es más parecido al diseño en sí(tabla por cada clase que hereda).

Desventaja: Acá se repetirían columnas, en caso de que las entidades tengan atributos en común, esto podría llegar a ser un problema. Además dos elementos de tipos de usuario diferentes podrían llegar a tener mismo ID, lo cual es perjudicial si necesitas traer todos los tipos de usuario en una sola lista. Este modelo se aleja de lo que pensamos originalmente, pero fue el que terminamos implementando.

Recuperación de datos: En esta parte consideramos que lo mejor sería utilizar eager, ya que casi siempre utilizamos los datos vinculados a cierto modelo para realizar consultas posteriores.

Paquete Domain: En este están ubicadas las excepciones, las cuales creamos solo algunas. En un principio creamos muchas para cada clase en especifico, pero entendimos que esto no tenía sentido ya que contemplabamos el mismo caso pero en clases distintas, por lo tanto era mejor hacer por casos, NotFound, DuplicatedEntry, etc.

Un punto importante fue hacer herencia con una clase usuario que herede a la clase Administrador, encargado y mantenimiento.

Ventajas: Es extensible y reutilizable, la clase usuario puede contener las autenticaciones respecto a atributos los cuales son comunes a todos los anteriores. Un caso posible es que no hereden de usuario y se realice una modificación en alguno de los atributos compartidos, en este caso habría que modificar el mismo código en las 3 clases, además de la adición de funcionalidades extras que podrían estar sujetas al perfil de un usuario en sí, como pedidos y pagos,mensajeria,eventos,entre otros.

Desventajas: Genera mayor acoplamiento, un cambio en la clase usuario afecta a todas las clases hijas, además hay pocas funcionalidades en común(se desaprovecha la idea de polimorfismo).

Una justificación y prueba clara es que LastName lo tiene admin y las personas de Staff, pero no los encargados(especificado por el cliente mediante consultas). Esto da un indicio de que hay que separar en clases o en entidades diferentes y no podemos dejar un usuario con un atributo “rol” común para todos.

Por otro lado separamos en clases los elementos mencionados en la especificación de la aplicación pasada por el cliente,teniendo edificios, apartamentos,entre otros.

A continuación se detallan puntos importantes de decisión:

-ApartmentOwner: Decidimos hacer una clase aparte para los dueños de edificios,esto ya que de forma contraria las solicitudes tendrán todos los datos de sus dueños, los cuales podrían cambiar o aumentar sus datos o lógica.

-Category: Podríamos haber dejado un string en solicitudes de servicio, pero nos pareció mejor para el tema de la comprobación si hacíamos una clase category y la persistimos. En caso de no persistir la categoría, tendríamos que modificar el código de la request, y cada vez que agregaramos una request con una categoría, comprobando si esa categoría es una válida para el sistema, ya sea con una lista de string o un Enum, lo cual no cumple con el principio OpenClosed.

-Compañía de construcción: Decidimos que solo fuera un string, esto ya que la letra específica que se pueden agregar existentes o nuevos para el sistema, por lo tanto no es un elemento a persistir y comprobar si existe o no.

-Request: Esta clase contiene un status para el estado de la misma, nos pareció buena idea ya que no hay muchos estados y es probable que no se agreguen muchos más, esto nos da un indicio de que con representarlo con un Enum bastaba. También pensamos asignar un tipo string, pero tendríamos que comprobar que fuera

correctamente ingresado, por lo tanto terminamos yendo por la opción del enum por su simplicidad. Algo que no nos gusta es que algunos valores quedan en null, ya que una solicitud recién creada no tiene un trabajador de mantenimiento adherido, o un coste final. Pensamos en la idea de separar por clases que hereden de una superclase Request, siendo implementada por OpenRequest, ClosedRequest y AttendingRequest. Esta opción prevenía valores en null pero aumentando la complejidad y el acoplamiento del sistema, por lo que la descartamos.

-Session: Esta clase fue hecha para poder validar que el usuario a solicitar un servicio tuviera los permisos necesarios. Le asignamos un email de usuario para saber quien es el que inició sesión con el token enviado. No pudimos asignarle un usuario ya que al separar las clases con TPC, entity framework no entendía la relación, por lo tanto le asignamos un email el cual era único y usado por todos los tipos de usuario. A los usuarios le asignamos una función para obtener su rol en formato string y hacer más fácil las autenticaciones.