

Universidad ORT Uruguay
Facultad de Ingeniería

Arquitectura de Software
Informe de Obligatorio

https://github.com/IngSoft-AR-2023-2/266628_271568_255981

Manuel Morandi - 271568

Federico Rodriguez - 255981

Santiago Salinas - 266628

Tutores:

Pablo Marcelo Geymonat Roldan

Rafael Jose Alonso Gerla

Gerardo Franklin Quintana Alpuin

2024

1. Introducción	2
1.1 Propósito	2
2. Antecedentes	2
2.1 Propósito del sistema	2
2.2 Requerimientos significativos de Arquitectura	3
2.2.1 Resumen de Requerimientos Funcionales	3
2.2.2 Resumen de Requerimientos de Atributos de Calidad	5
3. Documentación de la arquitectura	7
3.1 Diagrama de contexto	7
3.2 Vistas de Módulos	8
3.2.1 Vista de Descomposición	8
3.2.1.1 Representación primaria	8
3.2.1.2 Catálogo de elementos	8
3.2.1.3 Decisiones de diseño	9
3.2.2 Vista de Uso	9
3.2.2.1 Representación primaria	9
3.2.2.2 Decisiones de diseño	9
3.2.3 Vista de Layers	10
3.2.3.1 Representación primaria	10
3.2.3.2 Catálogo de elementos	11
3.2.3.3 Decisiones de diseño	12
3.2.4 Modelo de datos	13
3.2.5 Catálogo de elementos	13
3.2.6 Interfaces	14
3.2.7 Comportamiento	15
3.3 Vistas de Componentes y conectores	16
3.3.1 Representación primaria	16
3.3.2 Catálogo de elementos	18
3.3.3 Interfaces	19
3.3.4 Comportamiento	20
3.3.5 Relación con elementos lógicos	23
3.3.6 Decisiones de diseño	24
3.4 Vistas de Asignación	25
3.4.1 Vista de Despliegue	25
3.4.1.1 Representación primaria	25
3.4.1.2 Catálogo de elementos	25
3.4.2 Vista de Instalación	27
3.4.2.1 Representación primaria	27
3.4.2.2 Catálogo de elementos	27
3.4.2.3 Decisiones de diseño	28
4. Guía de instalación	29
Primera vez	29
Para futuras corridas	33

1. Introducción

El siguiente trabajo es presentado como entrega de proyecto obligatorio final para la materia Arquitectura de Sistemas. Se busca desarrollar un sistema que cumpla con todos los requerimientos solicitados, así como cumplir ciertos estándares definidos para distintos atributos de calidad.

Este documento busca servir como guía para comprender el funcionamiento de nuestro sistema. Se aportan explicaciones del mismo a través de distintas vistas, para obtener así un panorama general. Las decisiones más específicas y cercanas al código se encuentran en Docs/ADRs en la forma de ADRs.

https://github.com/IngSoft-AR-2023-2/266628_271568_255981/tree/main/Docs/ADRs

1.1 Propósito

El presente documento busca proveer un entendimiento general de la arquitectura del sistema planteado como solución a la problemática presentada, Inmo2.0. Se dividirá en distintas secciones, donde se verá el sistema y su estructura arquitectónica desde distintos puntos de vista. El objetivo es generar una visión clara y extensiva de todas las decisiones de mayor relevancia a la hora de construir y desarrollar nuestra aplicación.

2. Antecedentes

2.1 Propósito del sistema

El trabajo se basa en la ideación y desarrollo de una aplicación de alquiler de inmuebles. Los distintos usuarios pueden administrar propiedades, así como manejar distintas reservas que sobre ellas se pueden hacer. Esto implica una serie de funcionalidades que cada usuario podrá hacer, dependiendo de su rol.

Un elemento diferencial de este sistema, es el soporte de diferentes tipos de sensores. Las propiedades presentan sensores inteligentes que realizan lecturas de distintos aspectos, tales como la temperatura de la casa, y los envían al sistema para que sean almacenados. Además, se presentan alertas personalizadas ante actividad emitida por los sensores mediante funciones.

2.2 Requerimientos significativos de Arquitectura

2.2.1 Resumen de Requerimientos Funcionales

ID Requerimiento	Descripción	Actor
RF1 - Login	Los usuarios tienen la posibilidad de autenticarse para acceder a las funcionalidades de Inmo2.0.	Usuarios web
RF2 - Reserva	Se permite efectuar reservas de las propiedades disponibles.	Inquilino
RF3 - Búsqueda de propiedades	Se puede buscar y filtrar las propiedades según la fecha deseada y otra serie de características. Los resultados de la búsqueda se muestran de forma paginada.	Inquilino
RF4 - Pago	Se permite a los usuarios realizar pagos de manera online al reservar una propiedad o al publicar un nuevo inmueble. Cuando se aprueba un pago, se le notifica al propietario y a los administradores.	Inquilino Propietario
RF5 - Cancelación de reservas por falta de pago	Si pasado un cierto tiempo (modificable) una reserva aún está a la espera de pago, ésta será cancelada.	Sistema

RF6 - Cancelación por parte del inquilino	Se permite a un inquilino cancelar una reserva. Esto tiene que suceder con cierta cantidad de días de anticipación (modificable).	Inquilino
RF7 - Consulta de reserva	Un usuario puede revisar sus reservas a partir de sus códigos.	Inquilino
RF8 - Consulta de reservas generales	Ciertos usuarios pueden acceder a todas las reservas realizadas en el sistema. Además, pueden filtrar por rango de fechas, código de propiedad, email, nombre o apellido del inquilino y/o estado de la reserva.	Administrador Operario
RF9 - Alta de inmuebles	Se pueden agregar propiedades al sistema. Se brindarán todos los datos y fotos necesarias y se validará. Tras hacer esto, el propietario deberá realizar un pago para que el inmueble quedé correctamente publicado y notificar al administrador.	Propietario
RF10 - Agenda	Como propietario, puedo definir las fechas donde mi propiedad estará disponible y cuáles no.	Propietario
RF11 - Alta de sensor	Se pueden agregar nuevos sensores al catálogo de la aplicación.	Administrador
RF12 - Asignación de sensores	Se puede asignar un sensor a una propiedad.	Administrador

RF13 - Monitorización	Se expone una API REST que permite visualizar el estado de los distintos sensores de un inmueble.	Administrador
RF14 - Notificación	Cuando un sensor emite una señal, se procesa y notifica su medida a ciertos usuarios. El medio por el que se notifica puede variar.	Sistema

2.2.2 Resumen de Requerimientos de Atributos de Calidad

ID Requerimiento	ID Requerimiento de Calidad o restricción	Descripción
RF1 - Login	AC1 - Seguridad	Es importante que los datos de inicio sesión, las credenciales y tokens, etc. de los usuarios estén protegidos.
RF1 - Login	AC2 - Disponibilidad	Los usuarios tienen que tener acceso a su cuenta siempre, por lo que la acción de iniciar sesión debe tener alta disponibilidad.
RF2 - Reserva	AC3 - Performance	El alta de reservas debe hacer uso de varios datos, tales como las propiedades o las fechas en la que están ocupadas. Esta necesidad de fetchear muchos datos no debe permitir que esta funcionalidad sea lenta.

RF4 - Pago	AC4 - Resiliencia	Se usará un módulo externo que se ocupará de los pagos. Debido a esto, no se puede garantizar su disponibilidad, por lo que el sistema debe ser resiliente y mantenerse en pie en caso de fallas.
RF5 - Cancelación de reservas por falta de pago	AC5 - Modificabilidad	La cantidad de tiempo que puede transcurrir sin pago puede depender según el país y puede cambiarse. Debido a esto, debe ser para los admins sencillo alterar esta cantidad.
RF6 - Cancelación por parte del inquilino	AC6 - Disponibilidad	El usuario puede querer cancelar su reserva en cualquier momento, y la cantidad de dinero retornada puede variar según cuánto falta hasta el inicio de la reserva, por lo que esta funcionalidad debe estar disponible para no perjudicar a nuestros usuarios.
RF13 - Monitorización	AC7 - Escalabilidad	Se espera el manejo de grandes cantidades de sensores, que realizarán grandes cantidades de medidas. El sistema deberá soportar esta alta demanda.
RF14 - Notificación	AC8 - Interoperabilidad	Los usuarios pueden ser notificados a través de distintos medios (email, WhatsApp, etc.), por lo que el sistema debe permitir que estos cambios de

		medios se den con facilidad.
--	--	---------------------------------

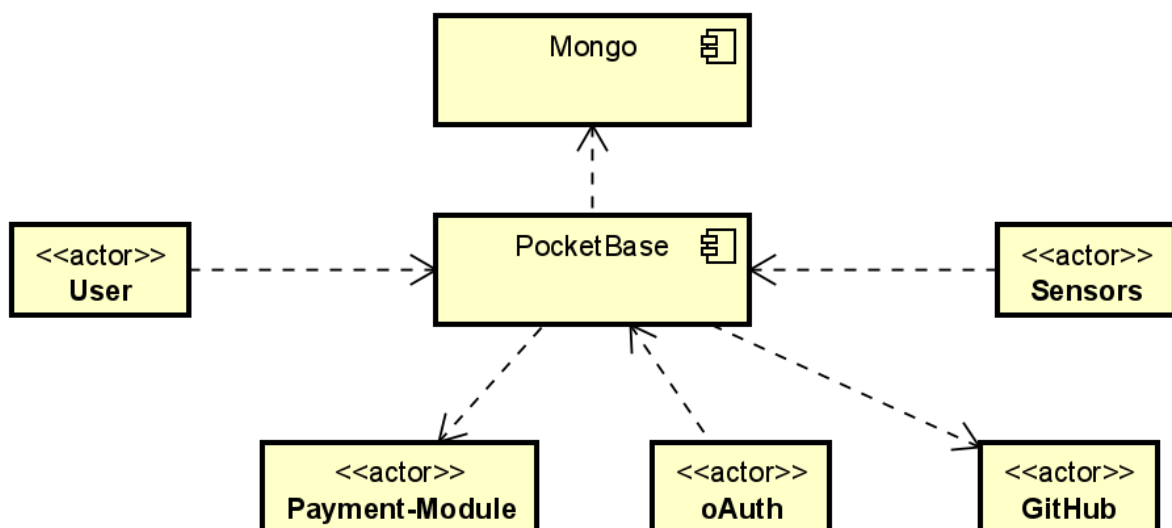
3. Documentación de la arquitectura

3.1 Diagrama de contexto

Es importante entender el contexto de nuestro sistema. Tendremos dos aplicaciones clientes distintas, una para los usuarios y otra para los sensores. Estas se comunicarán con nuestro sistema de manera asíncrona, a través de apis. Luego, el servidor toma los datos y los procesa.

Se trata de un sistema monolítico con estructura cliente-servidor, pero altamente performante. Los clientes realizan peticiones al servidor, que hace uso de los múltiples servicios que contiene para procesar estas requests. El servidor posee múltiples módulos independientes, que se ocupan de llevar a cabo distintos servicios de la lógica de negocio, pero todos estos hacen uso de un sistema de bases de datos único, formado por dos bases de datos diferentes. Por un lado, se maneja PocketBase para almacenar la información de los usuarios, sensores, inmuebles, reservas, etc., mientras que se hace uso de MongoDB para guardar las mediciones de los sensores, que llegarán con mayor frecuencia.

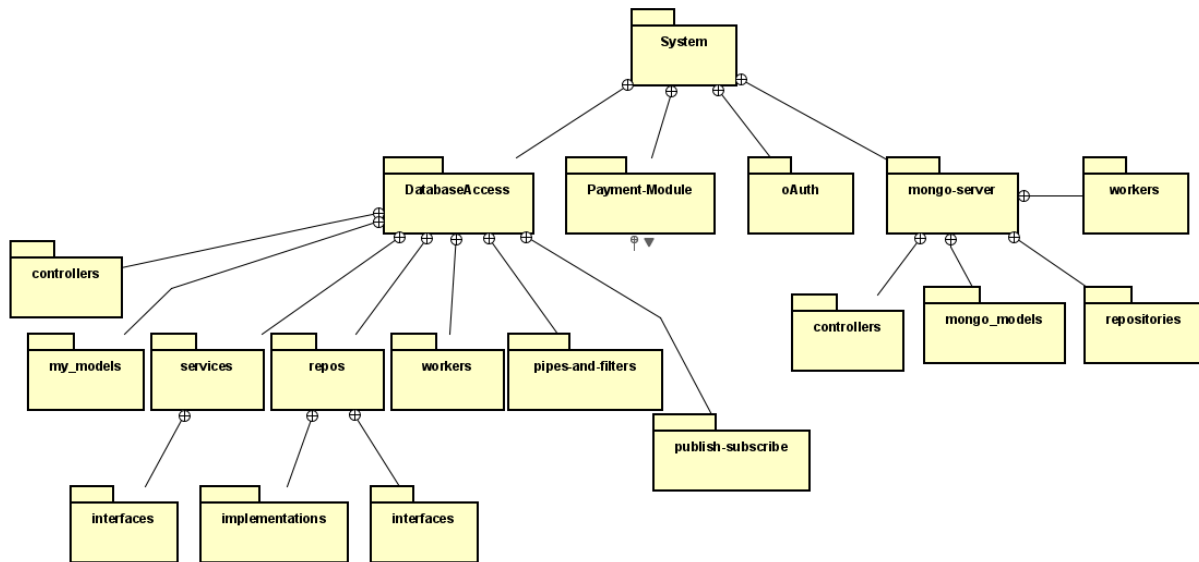
El siguiente diagrama representa el contexto de nuestra solución:



3.2 Vistas de Módulos

3.2.1 Vista de Descomposición

3.2.1.1 Representación primaria



3.2.1.2 Catálogo de elementos

A grandes rasgos, encontramos 4 módulos principales. Por un lado, Payment-Module es el responsable de procesar los distintos pagos que la app puede necesitar hacer. Se busca que este módulo simule ser externo, pese a ser un elemento más de nuestra aplicación.

A continuación encontramos oAuth, responsable del acceso al módulo de autenticación provisto por GitHub. Nuestro sistema usa este proveedor de identidad para ocuparse del login, y este módulo se ocupa de proveer lo necesario para que esto funcione.

Finalmente tenemos los dos servidores principales. El servidor de PocketBase, llamado Database Access, se ocupa de almacenar toda la información relevante para el usuario y exponerla, junto con toda la funcionalidad, a través de una API. Este módulo hace uso de mongo-server, para almacenar en MongoDB los reportes de los sensores, ya que se esperan que estos sean una gran cantidad y se desea poder llevar a cabo esta acción de manera veloz y performante.

3.2.1.3 Decisiones de diseño

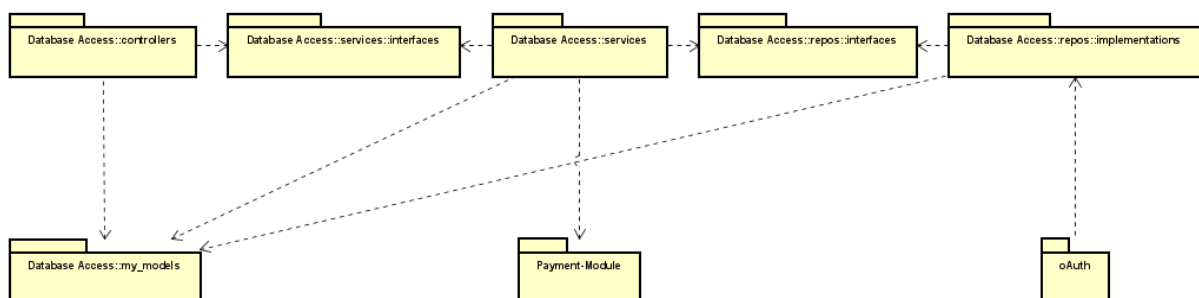
Principalmente, se tomaron 3 decisiones de diseño importantes en relación a esta vista. Primero, como ya se habló, se hace uso de GitHub para realizar las acciones de autenticación de usuarios. Esto nos lleva a la implementación del patrón identidad federada. Los usuarios inician sesión en GitHub y eso les otorga un token, con el que pueden hacer uso de nuestro sistema. Se puede expandir esta explicación con el ADR 007 - *Identidad federada*.

Otras decisiones importantes son las relacionadas con las bases de datos y su uso. Primero, como se explicó en el apartado anterior, se tienen dos bases de datos distintas que cumplen funciones diferentes, teniendo cada una un propósito específico. 004 - *Múltiple bases de datos* abarca lo que implica esto.

Por otro lado, la base de datos de PocketBase tiene también la responsabilidad de cumplir lo propuesto en 006 - *Manejo de variables modificables en run-time con DB*, al exponer tablas para permitir obtener los valores que pueden cambiar, como el porcentaje de devolución. Este enfoque permite que un administrador altere estos valores en tiempo de ejecución, permitiendo adaptar la app a las necesidades del negocio sin que esta deba ser compilada nuevamente.

3.2.2 Vista de Uso

3.2.2.1 Representación primaria



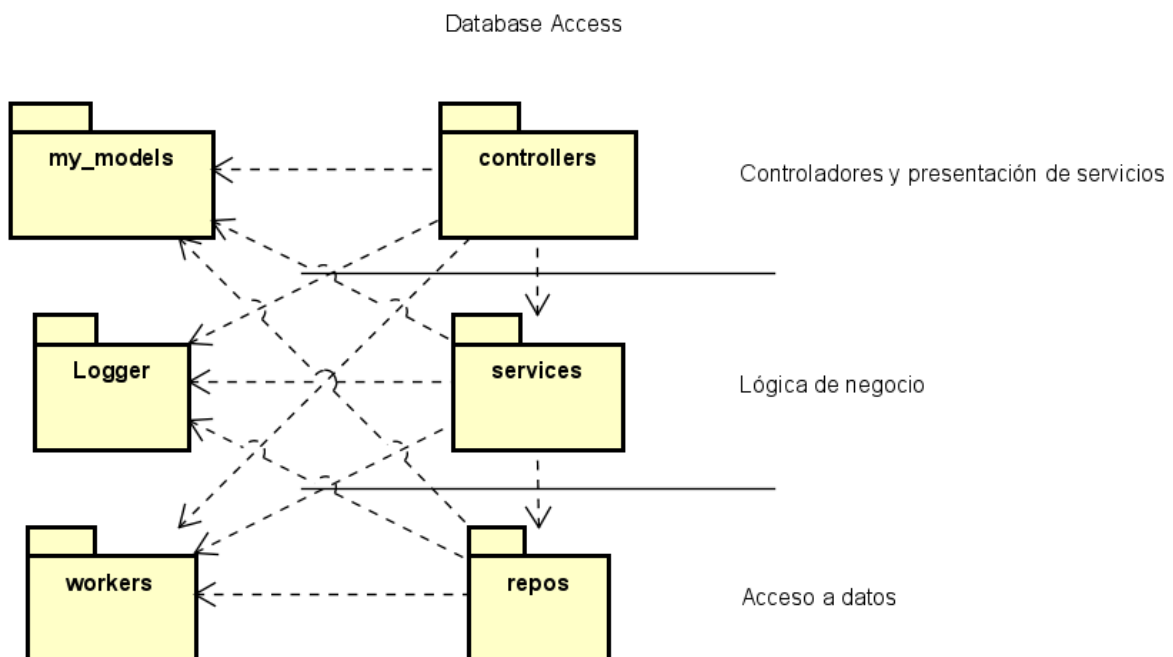
3.2.2.2 Decisiones de diseño

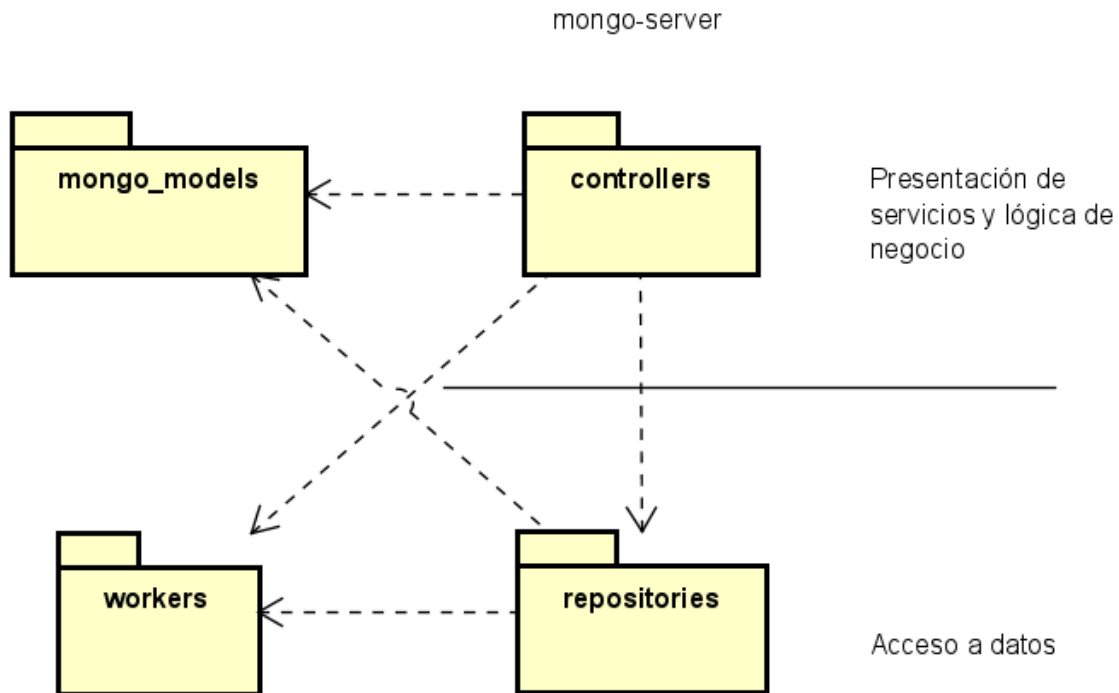
En general, las decisiones de diseño tomadas en este apartado se asemejan bastante a las presentadas en la vista de descomposición. Se hace uso de módulos externos (Payment-Module y OAuth), pero se intenta encapsular lo más posible, disminuyendo la cantidad de módulos que de estos dependen. Esto hace que, en caso de un eventual cambio en estos módulos, nuestro sistema no se vea afectado.

Esta vista también nos permite ver claramente la división en capas de nuestro sistema, pero esto es algo que se tratará en el siguiente punto. Otro punto importante es el plasmado en *ADR 016 - Uso de abstracciones*, donde, para fomentar la modificabilidad del sistema y protegernos ante posibles cambios, se establecen interfaces que definen los métodos que serán requeridos por los servicios. Los distintos repositorios luego implementan estas interfaces para así poder llevar a cabo las funcionalidades necesarias.

3.2.3 Vista de Layers

3.2.3.1 Representación primaria





3.2.3.2 Catálogo de elementos

Los módulos que es interesante analizar por capas son nuestros servidores. Ambos fueron codificados en go y tienen paquetes compartidos y utilizados por todas las capas, tales como los modelos y los workers.

Ahora, si vamos a cosas más específicas, ambos módulos están formados por capas, pero divididas de manera distinta. Database Access tiene 3 capas: una para los controladores, una intermedia para los servicios y una final con los repositorios. Este enfoque tiene sentido debido a la complejidad de este módulo, ya que se ocupa de una diversidad de tareas y una mayor granularidad permite distribuir de manera más ordenada y comprensible las responsabilidades entre los elementos que forman este módulo.

En contrapartida, mongo-server no tiene un abanico de funcionalidades tan amplio, por lo que realizar esa división es contraproducente. Se agregaría más complejidad de lo necesario. Este servidor cuenta únicamente con dos capas, suprimiendo la capa intermedia de servicios. Este enfoque genera un módulo más sencillo, donde las responsabilidades se distribuyen de manera simple y lógica.

3.2.3.3 Decisiones de diseño

De manera más específica, podemos destacar que nuestro sistema se construirá en capas, cosa que ya se adelantó al explicar que se trata de una arquitectura cliente-servidor. Cada capa se ocupará de un nivel de abstracción distinto, teniendo diferentes responsabilidades y objetivos.

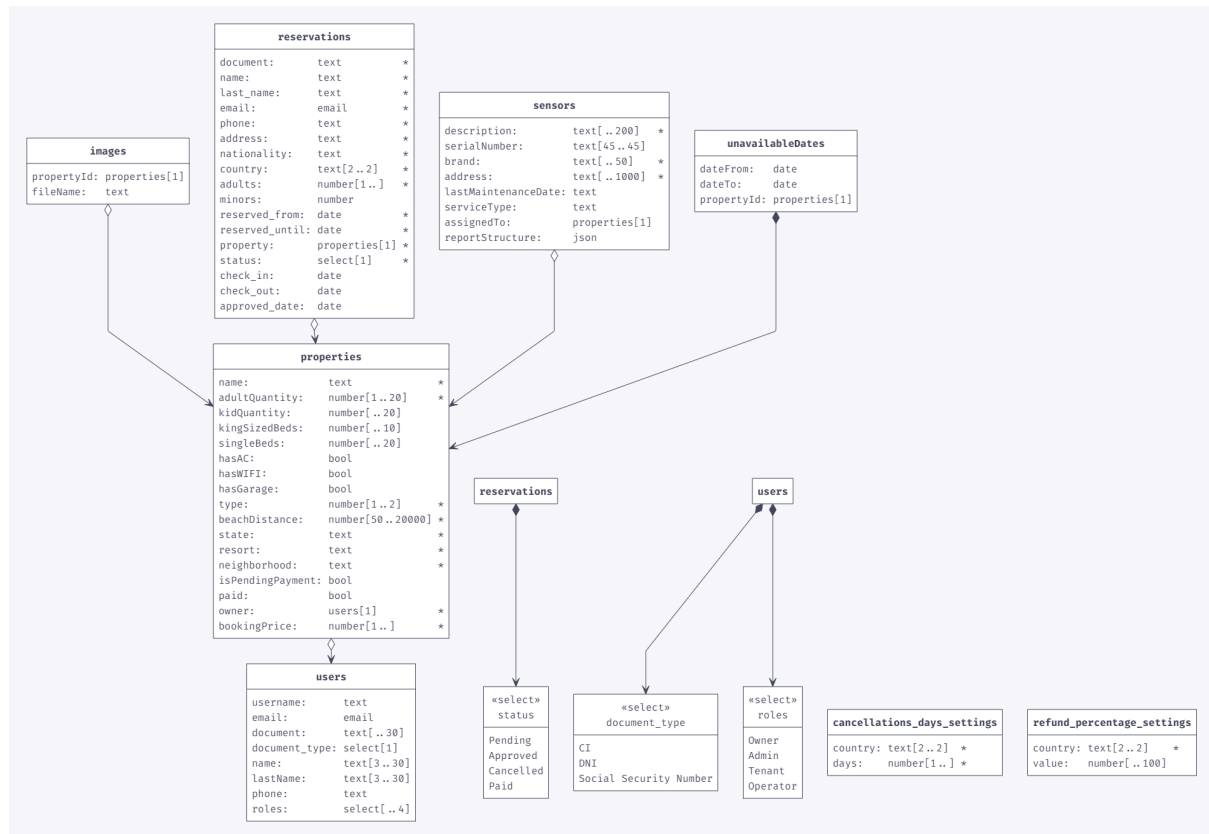
Se poseerán los sistemas clientes, que se comunicarán con apis, definidas en la capa de controladores. Estos controladores harán uso de los servicios específicos definidos en la siguiente capa. A su vez, esta capa tendrá como herramienta principal los modelos y la lógica básica de negocio definido en la penúltima capa. Finalmente, se encuentra la capa de acceso a datos, que es el intermediario entre el servidor y la base de datos.

Una capa interactúa con la siguiente a través de interfaces. De esta manera, los servicios no dependen directamente de la implementación del repositorio, ya que únicamente requieren una interfaz que el repositorio provee. Esto está dentro de las directivas del *ADR 016 - Uso de abstracciones*.

Esta arquitectura disminuye el acoplamiento y define un orden lógico de cada elemento de nuestro sistema. Esto se traduce a una mayor modificabilidad, con el costo de bajar la performance dado a la cantidad de intermediarios existentes.

Podemos notar también la presencia de un logger en Database Access. Según lo planteado en *ADR 012 - Logger*, se implementó este módulo para poder registrar cualquier evento que ocurre en nuestro sistema. Al inicializar el servidor, se genera un archivo de texto donde se escribe en tiempo real cualquier fallo, imprevisto, advertencia o información relevante que se da en el uso del sistema. De esta manera, se puede tener un mejor monitoreo y favorecer la disponibilidad al poder recuperarnos ante fallas de manera más sencilla, ya que si el sistema cae tendremos una bitácora de eventos que llevaron a su caída.

3.2.4 Modelo de datos



3.2.5 Catálogo de elementos

Elemento	Responsabilidades
main	Inicializa todos los objetos necesarios para el correcto funcionamiento del sistema, como el cliente de Redis o el logger
controllers	Presenta la funcionalidad a través de métodos que serán servidos como endpoints
logger	Introduce una clase que se ocupa de registrar en disco cualquier evento relevante que se da en el sistema, como errores o warnings
my_models	Incluye todos los modelos que se usan en el sistema
pipes_and_filters	Se establece una pipeline que recibe y procesa los reportes de enviados por los sensores

pubsub	Se ocupa de la implementación de una cola pubsub, utilizada para la notificación a través de distintos medios
repositories	Es la última capa antes de la base de datos, realizando todo el procesamiento necesario y llevando a cabo el acceso a datos
repointerfaces	Abstrae la implementación de los repositorios
services	Expone cierta lógica de negocio y sirve de intermediario entre los controladores y los repositorios
workers	Gestiona conexiones y operaciones de trabajadores con RabbitMQ, permitiendo configurar, escuchar, enviar mensajes y añadir middleware en colas de mensajes
interfaces	Guarda las interfaces que deben cumplir los servicios y los repositorios. A su vez, los controladores conocen solo conocen interfaces y no implementaciones directas.

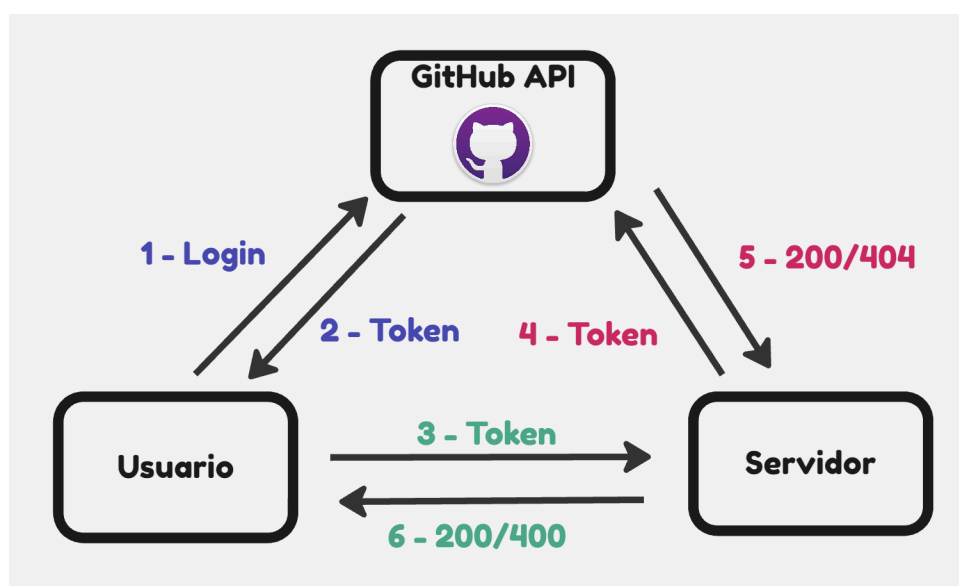
3.2.6 Interfaces

Interfaz:	Worker
Paquete que la implementa:	Mongo-server y Database Access
Servicio	Descripción
ReportsController	Es un servicio dentro de DatabaseAccess que utiliza una interfaz Worker, implementada mediante RabbitMQ para insertar los reportes válidos en una cola de mensajes.
Mongo-server	En su código de inicio, instancia un Worker implementado mediante RabbitMQ para consumir los mensajes de las colas que fueron popular por el DatabaseAccess

Interfaz:	API Rest
Paquete que la implementa:	controllers dentro de DatabaseAccess
Servicio	Descripción
Todos los controllers del módulo Database Access.	<p>Mediante Pocketbase definimos y exponemos endpoints de una API Rest, y luego manejamos y respondemos a las peticiones entrantes.</p> <p>Con la información entrante, la procesamos y hacemos operaciones con nuestra base de datos para dar las respuestas esperadas.</p>

3.2.7 Comportamiento

El proceso de autenticación y autorización para los usuarios del sistema se hace mediante el uso de Github como un ente intermedio de confianza. En el siguiente diagrama se muestra el proceso de intercambios peticiones y respuestas entre las distintas partes:



1. Primero el usuario debe acceder al endpoint establecido para iniciar sesión para nuestro sistema mediante su cuenta de Github. Esto se logra gracias a registrar nuestro sistema como una OAuth app de github, por lo que podemos utilizar su API para mostrar una página de login con github.
2. Completar el login correctamente resulta en que la API le devolverá al usuario su token de identificación. El usuario debe guardarse dicho token para autenticarse con nuestra API al hacer peticiones http.
3. Cuando el usuario desee realizar una operación que requiera de autenticación, debe enviar su token en un header "auth".
4. Al recibir una petición, el servidor tomará el token del header para utilizarlo en una petición a la API de github. Este intercambio termina antes de comenzar cualquier operación para no realizar acciones innecesarias en el caso de que falle la autenticación.
5. El endpoint de autenticación de github responderá con un un body que incluye el nombre de usuario que corresponde al token, o con un error si el token enviado no es válido.
6. Finalmente, dependiendo de la respuesta de Github, el servidor buscaría los roles (permisos) en su base de datos que están asociados al nombre de usuario obtenido. Así determina si proseguir con la operación o responder al cliente con un error.

3.3 Vistas de Componentes y conectores

3.3.1 Representación primaria

Figura 3.3.1.1

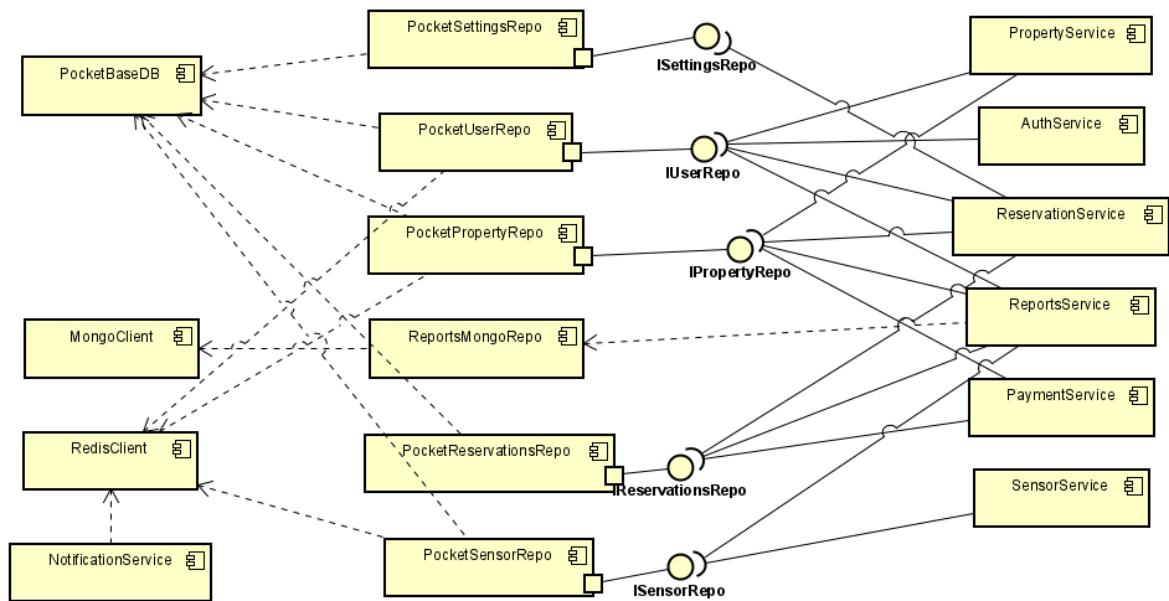


Figura 3.3.1.2

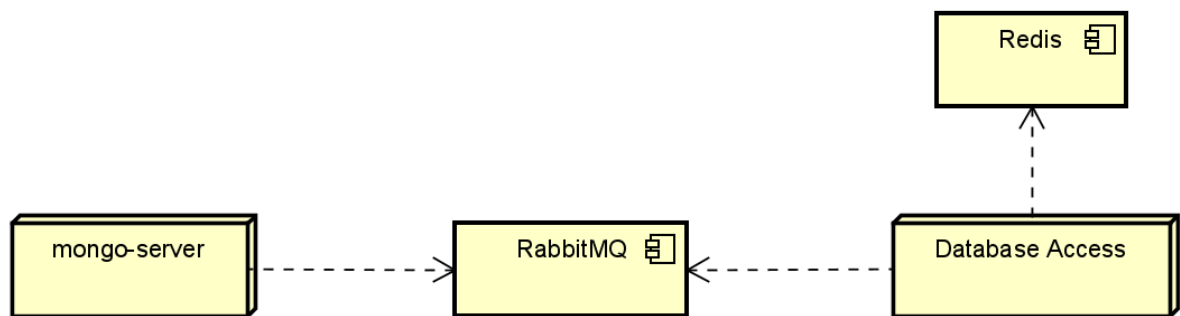
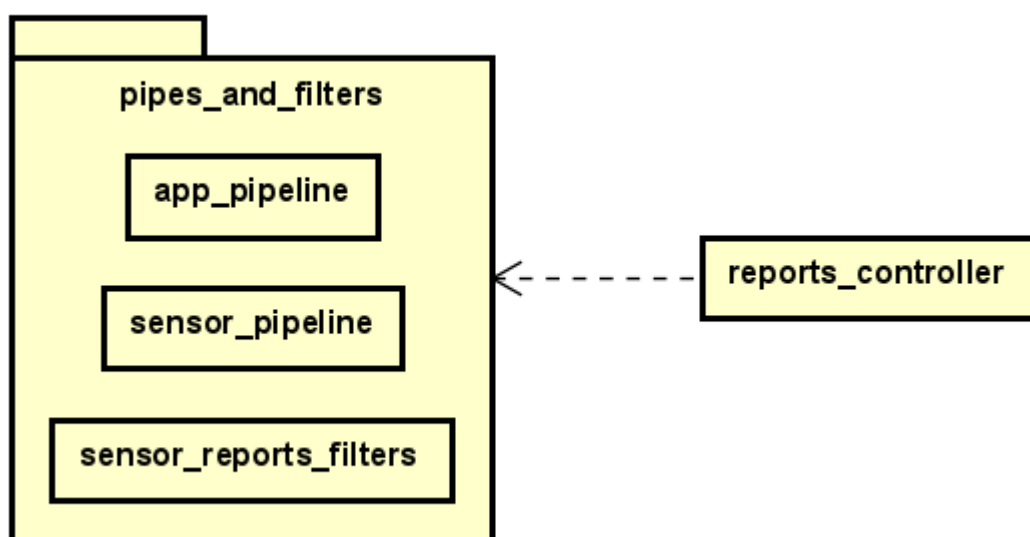


Figura 3.3.1.3



3.3.2 Catálogo de elementos

Componente/conector	Tipo	Descripción
PocketBaseDB	Base de datos	Base de datos principal del sistema.
MongoClient	Cliente de BD	Cliente para interactuar con la base de datos MongoDB.
RedisClient	Cliente de BD en memoria	Cliente para interactuar con la base de datos en caché Redis.
NotificationService	Servicio	Servicio encargado de gestionar las notificaciones.
PocketSettingsRepo	Repositorio	Repositorio para gestionar las configuraciones, tales como la cantidad de días de anticipación con los que se puede cancelar con retorno total del dinero.
PocketUserRepo	Repositorio	Repositorio para gestionar los usuarios.

PocketPropertyRepo	Repositorio	Repositorio para gestionar los inmuebles.
ReportsMongoRepo	Repositorio	Repositorio para gestionar los informes almacenados en MongoDB.
PocketReservationsRepo	Repositorio	Repositorio para gestionar las reservas.
PocketSensorRepo	Repositorio	Repositorio para gestionar los datos de los sensores.
PropertyService	Servicio	Servicio para operaciones relacionadas con las propiedades.
AuthService	Servicio	Servicio para la autenticación de usuarios.
ReservationService	Servicio	Servicio para gestionar reservas.
ReportsService	Servicio	Servicio para generar y gestionar informes.
PaymentService	Servicio	Servicio para gestionar los pagos.
SensorService	Servicio	Servicio para gestionar la información de los sensores.

3.3.3 Interfaces

Interfaz	Componente que la provee	Servicio	Descripción
ISettingsRepo	PocketSettingsRepo	ReservationService	Provee los métodos necesarios para acceder a los valores configurables del negocio.

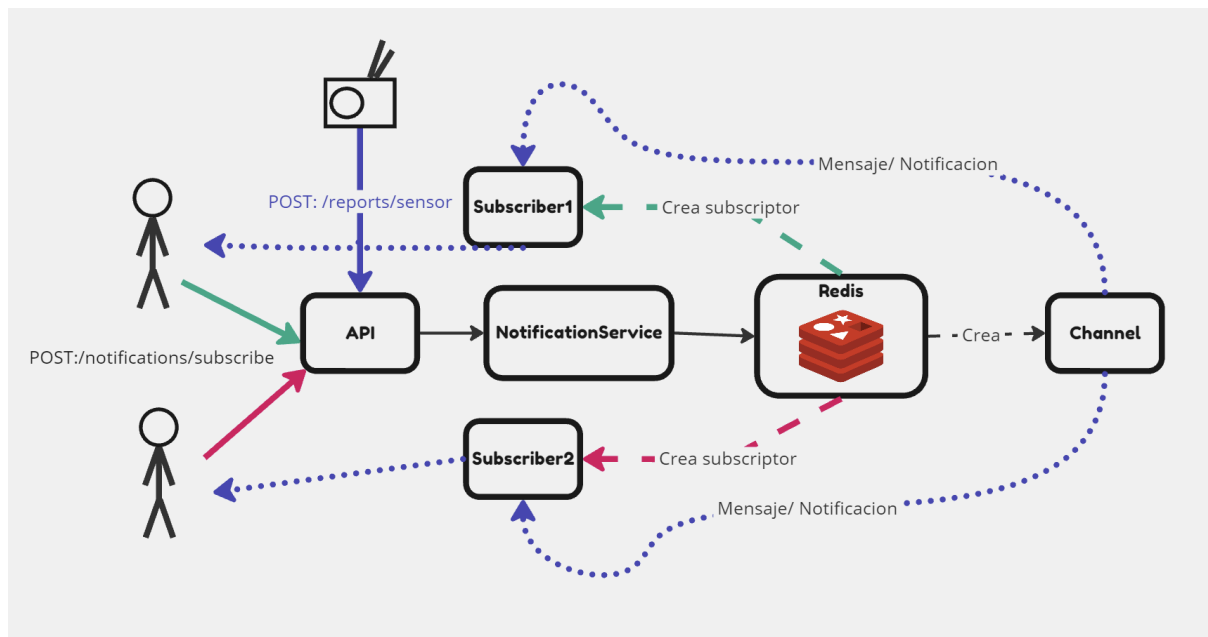
IUserRepo	PocketUserRepo	PropertyService, AuthService, ReservationService, ReportsService	Proporciona acceso a la gestión de usuarios.
IPropertyRepo	PocketPropertyRepo	PropertyService, ReservationService, PaymentService, ReportsService	Expone las funciones de manejo de inmuebles.
IReservations Repo	PocketReservations Repo	ReservationService, PaymentService, ReportsService	Permite la gestión de reservas.
ISensorRepo	PocketSensorRepo	SensorService, ReportsService	Expone todos aquellos métodos relevantes para la gestión y uso de sensores.

3.3.4 Comportamiento

El módulo de notificaciones de nuestro sistema se centra alrededor de la implementación de un patrón Pub-Sub, implementado con la ayuda de la tecnología Redis. Tenemos dos clases: RedisSubscriptionChannel y Publisher, que son utilizadas por la clase NotificationService para administrar canales de información a los cuales clientes pueden suscribirse para obtener notificaciones en base a los reportes recibidos por el servidor.

Este servicio de notificaciones simula a los suscriptores de los canales mediante funciones que dejan en los logs registros de haber recibido un reporte del canal al que se suscribieron. Se implementó de manera tal que está preparado para utilizar otros métodos de notificación, ya sea enviando un email o un mensaje de whatsapp por ejemplo.

Mediante un POST a los endpoints /notifications/subscribe o /notifications/unsubscribe, y llenando el body con el mail del suscriptor y el canal con el que quiere interactuar, podemos modificar los suscriptores de los canales en tiempo de ejecución.



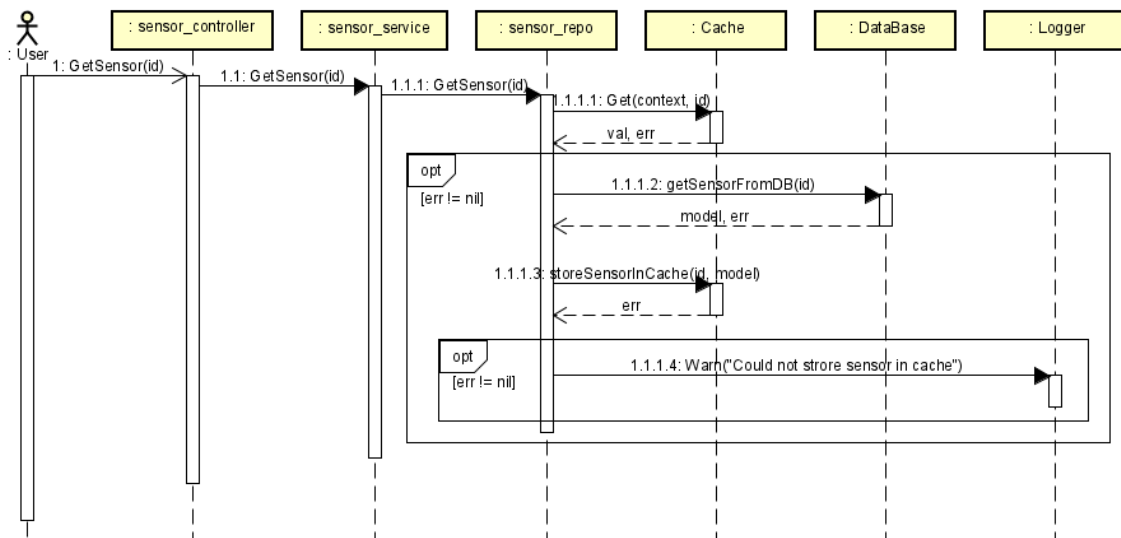
En el diagrama anterior se muestran tres flujos: El verde y el rojo representan el flujo que ocurre cuando un ente se suscribe a un canal. En este caso, los dos usuarios se suscriben al mismo canal, creado por el notificationService a través de Redis. Ambos de estos flujos resultan en la creación de las entidades Subscriber1 y Subscriber2, creados también por Redis. Finalmente, el flujo azul representa un sensor emitiendo un reporte, que luego es publicado en el canal de Redis, a través del cual llega a cada subscriber, que tienen cada uno un handler para enviar la notificación correspondiente a los usuarios.

Otro comportamiento interesante es el del uso del caché. Para ejemplificar, veamos como funciona al hacer el fetch de un sensor. El usuario solicita el sensor al controlador, y ahí comienza el pasamano hasta que llega al repositorio. En ese punto, el repositorio solicita el sensor al cliente de Redis y, en caso de tenerlo, lo retorna.

Ahora, si no se encuentra en la memoria caché o se dió algún fallo, se solicita el sensor a la base de datos. Posteriormente, se intenta almacenar el sensor en caché. En caso de que algo falle en este paso y no se pueda guardar en caché, no se hará y el logger escribirá un warning avisando que no se pudo llevar a cabo esta acción.

Algo importante a destacar es que todo esto sucede únicamente si el cliente Redis se encuentra disponible. Si no fue inicializado o por alguna razón no está funcionando en ese momento, se saltea todo este proceso y se accede directamente a la base de datos. El logger se ocupará de notificar si cualquier eventualidad impide que transcurra el proceso esperado.

El siguiente diagrama sirve para ejemplificar esto.

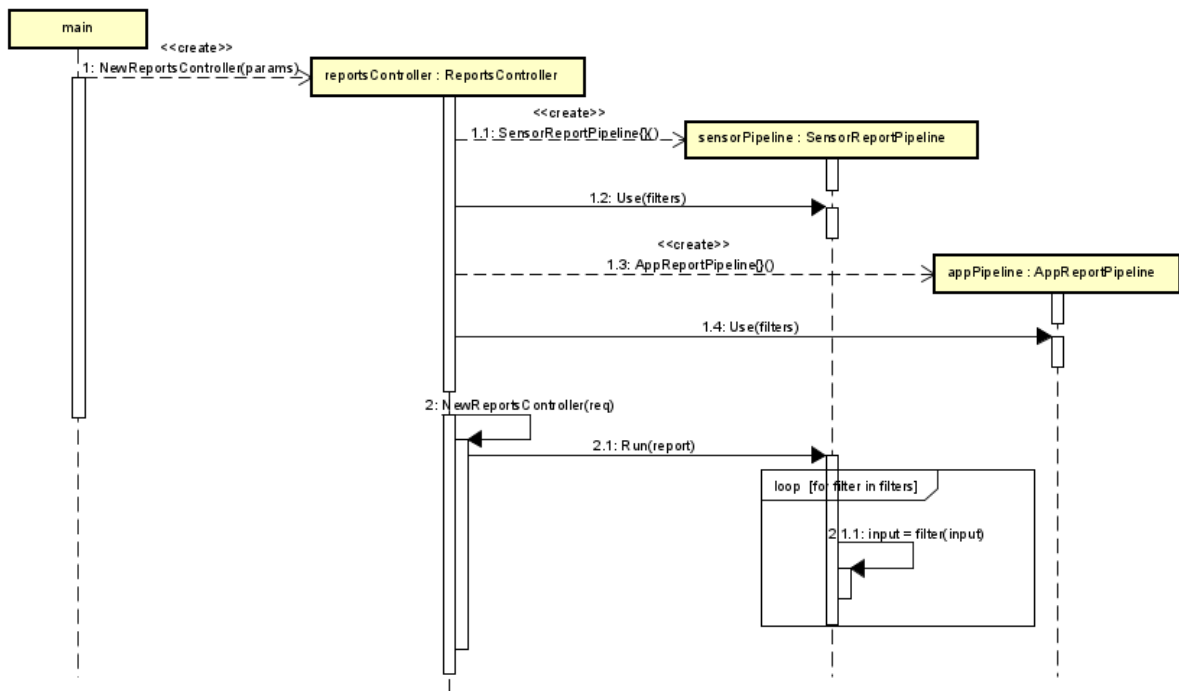


La vista 3.3.1.3 nos ofrece el microsistema generado por el uso de pipes and filters. Este patrón se aplicó para favorecer la modificabilidad y disponibilidad del controlador de reportes.

Al inicializar el controlador, este creará las pipelines con los pasos necesarios (como validación de los datos y publicación de los reportes). Frente al POST de algún reporte por un sensor o desde la app, el controlador inserta dicho reporte en el pipeline para que pase por todos los filtros establecidos.

En el caso de fallar en algún paso, existe la posibilidad de intentarlo un número determinado de veces, pero si aun así falla, se corta su camino por el pipeline. Esto significa que, buscando la disponibilidad de nuestro controlador, si algo no funciona correctamente se aplica la táctica de retry, hasta que eventualmente se abandone la intención de llevar a cabo esa operación que falla.

El siguiente diagrama muestra la creación de ambas pipelines, y el posterior uso de una de ellas:



3.3.5 Relación con elementos lógicos

Componente	Paquetes
PocketSettingsRepo	PropertyService
PocketUserRepo	AuthService
PocketPropertyRepo	PropertyService, ReservationService, PaymentService
ReportsMongoRepo	ReportsService
PocketReservationsRepo	ReservationService, PaymentService
PocketSensorRepo	SensorsService
PocketBaseDB	PropertyService, AuthService, PropertyService, ReservationService, PaymentService, SensorsService

MongoClient	ReportsService
RedisClient	NotificationService

3.3.6 Decisiones de diseño

Esta vista nos ofrece la oportunidad de ver algunas decisiones tomadas bastante interesantes. Por ejemplo, comencemos por la redactada en el *ADR 004 - Múltiple bases de datos*. Como podemos ver en la Figura 3.3.1.2, nuestro sistema posee dos servidores distintos, Database Access (PocketBase) y mongo-server (MongoDB). Es evidente, entonces, que se hace uso de dos bases de datos distintas, PocketBase para almacenar la mayoría de datos con los que se trabaja y Mongo para aquellos datos insertados con gran frecuencia. Estos dos servidores se comunican entre ellos a través de colas de mensajes, tal como se explica en el *ADR 002 - Colas de mensajes*.

También es destacable que Database Access utiliza Redis para poder definir una memoria caché, como se planteó en el *ADR 003 - Múltiple copias de datos (caché)*. Ciertos datos que se piden con mayor frecuencia, como los inmuebles, son cacheados para agilizar el proceso al disminuir los accesos a la base de datos.

En la Figura 3.3.1.1 podemos notar claramente el *ADR 016 - Uso de abstracciones*, ya que los servicios no interactúan directamente con los repositorios. Se crea una interfaz para cada repositorio, que es requerida por los servicios. Luego cada repositorio implementa su interfaz, lo que nos permite abstraer el servicio del repo. Si en algún momento se decide cambiar la base de datos o utilizar una nueva implementación de algún repositorio, solo deberemos modificar el resto sin que el servicio siquiera se entere del cambio.

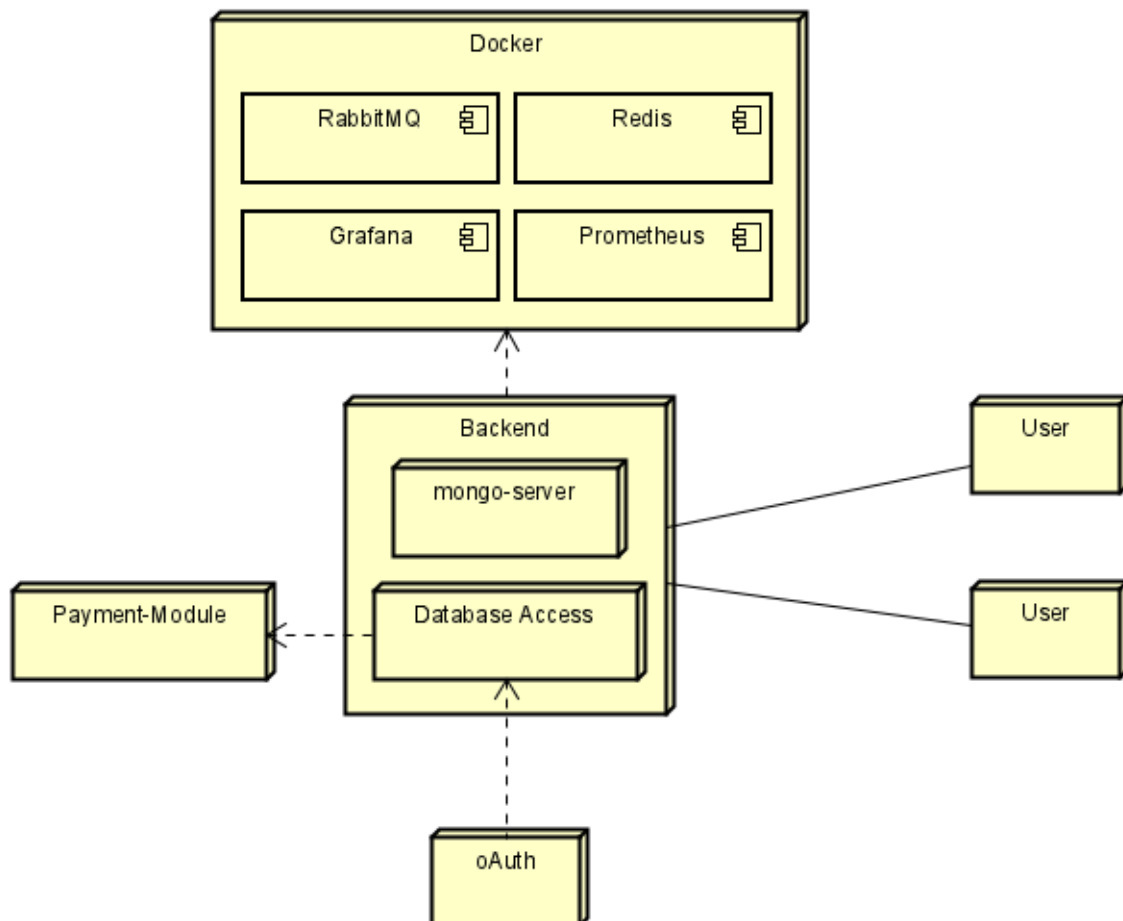
En la figura 3.3.1.3 se nota el uso de pipes and filters para ejecutar las validaciones y acciones necesarias en el proceso de creación de un reporte. Este patrón se aplicó tal como se definió en el *ADR 017 - Pipes and Filters*, y su comportamiento fue explicado en el punto 3.3.4 del presente informe.

Finalmente, tal como se explicó en la sección de comportamiento, y siguiendo lo establecido en *ADR 009 - Suscripción a distintos medios de notificaciones* y *ADR 018 - PubSub*, se hizo uso del patrón publicador-suscriptor para poder notificar a los distintos usuarios a través de distintos medios. Cada usuario se suscribe a ciertos medios, y, cuando debe ser notificado, se le notificará a todos los medios a los que esté suscripto.

3.4 Vistas de Asignación

3.4.1 Vista de Despliegue

3.4.1.1 Representación primaria



3.4.1.2 Catálogo de elementos

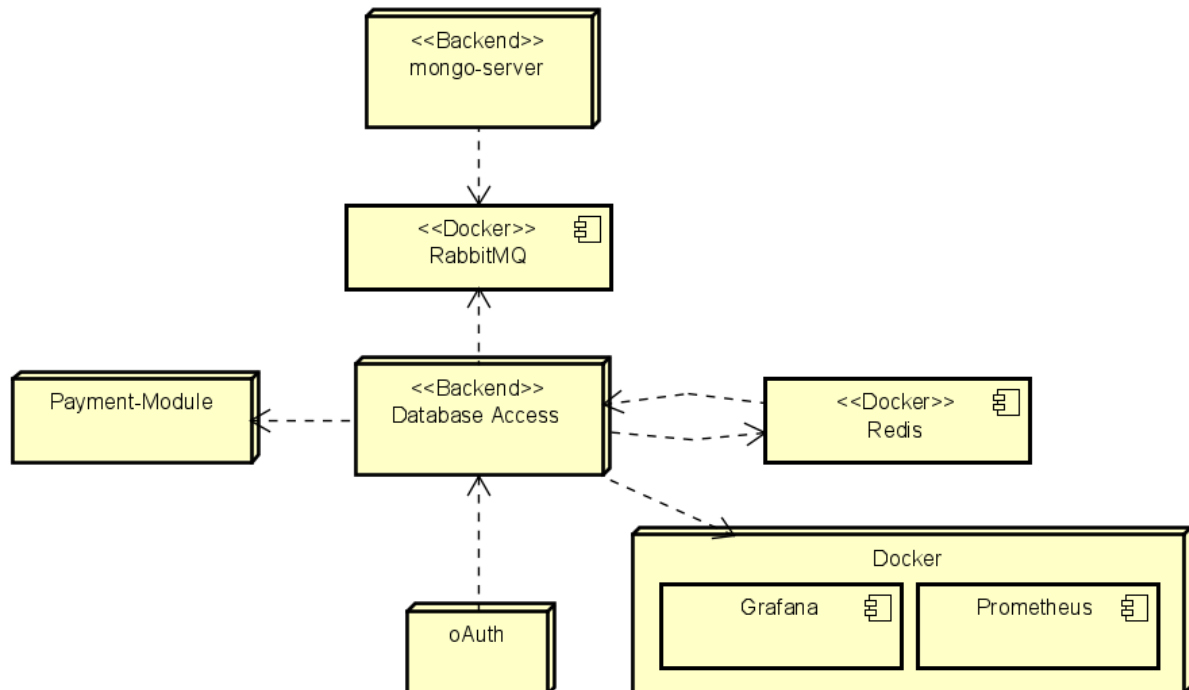
Nodo	Características (velocidad, memoria, etc.)	Descripción
User	Depende de la máquina del usuario	Es el dispositivo utilizado por el usuario para hacer uso de los endpoints expuestos por nuestro sistema

Database Access	<p>Nuestra máquina con la que corremos nuestro sistema contiene a todos los componentes:</p> <p>i5-10400F</p> <p>Total Cores 6</p> <p>Total Threads 12</p> <p>16 gb RAM</p>	Son los puntos de acceso a nuestro sistema. Expone múltiples endpoints para iniciar todas las operaciones para los inquilinos, operadores, admins, y propietario
Payment-Module		Componente que simula la interacción de nuestro servidor con un componente para realización de pagos expuesto por un tercero.
Docker		Contiene las múltiples tecnologías que complementan a nuestro sistema. Grafana, Redis, Prometheus, RabbitMQ
oAuth		Simple componente que expone una url para poder iniciar sesión a la OAuth app asociada a nuestro sistema

Conector	Características (velocidad, etc.)	Descripción
RabbitMQ	Ambos son ejecutados mediante contenedores de Docker en la máquina previamente descrita	RabbitMQ es una cola de mensajes utilizada por el componente DatabaseAccess para comunicar al componente mongoServer todos los reportes de los sensores, tras ser validados.
Redis		Redis actúa como un conector para los suscriptores y el notificationsService dentro del dataAccess, así como un intermedio para acceder al caché antes de ir directo a la base de datos cuando se hace un GET.

3.4.2 Vista de Instalación

3.4.2.1 Representación primaria



3.4.2.2 Catálogo de elementos

Nodo	Descripción
DataAccess	Es el componente principal del servidor. Donde se maneja la lógica del negocio y el acceso a la base de datos. Expone una API Rest con la cual los usuarios interactúan para ejecutar operaciones. También envía mensajes al mongo-server con los reportes que recibe.
Oauth	Componente simple que expone un endpoint para permitir a usuarios iniciar sesión en la OAuth app de github asociada a nuestro servidor.
Mongo-server	Componente simple que consume reportes de una cola de mensajes para guardarlos en una base de datos implementada con MongoDB.

Payment Module	Componente simple utilizado directamente por DatabaseAccess para simular un sistema de pagos expuesto por un tercero.
Grafana	Solución para mostrar en un Dashboard las requests por minuto que llegan a nuestro servidor, la cantidad de pagos completados correctamente y aquellos rechazados. Está desplegado en un volumen de Docker.
Prometheus	Herramienta de monitoreo de eventos. Se utiliza para generar una base de datos con la información que es consumida por Grafana. Está desplegado en un volumen de Docker

Conector	Descripción
RabbitMQ	Cola de mensajes que actúa como intermediario entre DatabaseAccess y el mongo-server, permitiendo al segundo consumir a su ritmo el constante flujo de reportes de los sensores despues de ser validados por DatabaseAccess. La instancia de RabbitMQ se encuentra desplegada mediante un volumen de Docker.
Redis	Tecnología utilizada no solo como caché por DatabaseAccess, sino que también mantiene canales de informacion a los cuales publica mensajes, y a los que los usuarios pueden suscribirse. Lo consideramos un conector dado su rol de mantener de publicar mensajes proveídos por y para DatabaseAccess. La instancia de Redis también se encuentra desplegada mediante un volumen de Docker.

3.4.2.3 Decisiones de diseño

Como se puede notar en esta vista, se usa RabbitMQ como intermediario, interfaz entre nuestros servidores. También se usa Redis para llevar a cabo la

implementación de un modelo PubSub bajo el cual enviar las distintas notificaciones. No obstante, ya se habló de estos puntos y sus ADRs correspondientes en anteriores tramos del informe.

Como novedad, encontramos que nos hemos decidido por utilizar Prometheus y Grafana para la recolección de datos y su análisis en dashboard para poder llevar un control del funcionamiento del sistema, ya que entre las opciones consideradas era la de menor costo de instalación, además de que son herramientas reconocidas en la industria. El hecho de que database access exponga un endpoint con métricas y que solo consuma recursos al ser consultado por Prometheus y Grafana, hace además que mientras no se esté utilizando no se estén enviando cantidades excesivas de datos a través de la red. Todo esto sigue lo planteado en el *ADR 010 - Monitoreo del estado del sistema*.

4. Guía de instalación

Primera vez

Instalar en el ordenador Go

```
https://go.dev/dl/
```

Nosotros estuvimos utilizando

```
go version go1.22.1 windows/amd64
```

Instalar en el ordenador CompassDB con MongoDB

```
https://www.mongodb.com/try/download/compass
```

Instalar en el ordenador Docker

```
https://docs.docker.com/desktop/install/windows-install/
```

Instalar en el ordenador k6

```
winget install k6 --source winget
```

Clonar el repositorio

```
git clone https://github.com/IngSoft-AR-2023-2/266628\_271568\_255981.git
```

Buscar mi dirección en la red con

```
ipconfig
```

```
Santiago ~ 0ms ipconfig

Windows IP Configuration

Ethernet adapter vEthernet (WSL):

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : 
    IPv4 Address. . . . . : 
    Subnet Mask . . . . . : 
    Default Gateway . . . . . : 

Ethernet adapter Ethernet:

    Connection-specific DNS Suffix  . : 
    IPv6 Address. . . . . : 
    Temporary IPv6 Address. . . . . : 
    Link-local IPv6 Address . . . . . : 
    IPv4 Address. . . . . : 192.168.1.12
    Subnet Mask . . . . . : 
    Default Gateway . . . . . :
```

Ir al archivo code/Docker/prometheus.yml y escribir la dirección ipv4 con el puerto 8181





















```
18 # Here it's Prometheus itself.
19 scrape_configs:
20   - job_name:
21       sample-project
22       # metrics_path defaults to '/metrics'
23       # scheme defaults to 'http'.
24
25   static_configs:
26     - targets: ["192.168.1.12:8181"]
27
```

Parados sobre code/Docker ejecutar en la terminal

```
cd code/Docker
docker network create metrics
```

y luego

```
cd code/Docker
docker-compose up
```

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	 docker		Running (4/4)	0%		17 seconds ago	  
<input type="checkbox"/>	 redis	redis:alpine	Running	0%	6379:6379	18 seconds ago	  
<input type="checkbox"/>	 prometheus	prom/prometheus:v2.45.3	Running	0%	9090:9090	18 seconds ago	  
<input type="checkbox"/>	 rabbitmq	rabbitmq:3-management-alpine	Running	0%	15672:15672 Show all ports (2)	18 seconds ago	  
<input type="checkbox"/>	 grafana	grafana/grafana:10.2.4	Running	0%	3000:3000	17 seconds ago	  

Luego se debe hacer en

```
cd code/Payment-Module
go run main.go serve
```

Luego se debe hacer en

```
cd code/mongo-server
go run main.go serve
```

Si se desea obtener credenciales de auth de github iniciar en

```
cd code/oAuth
go run main-git.go serve
```

Finalmente se debe prender el sistema yendo a

```
cd "code/Database Access"
go run main.go serve
```

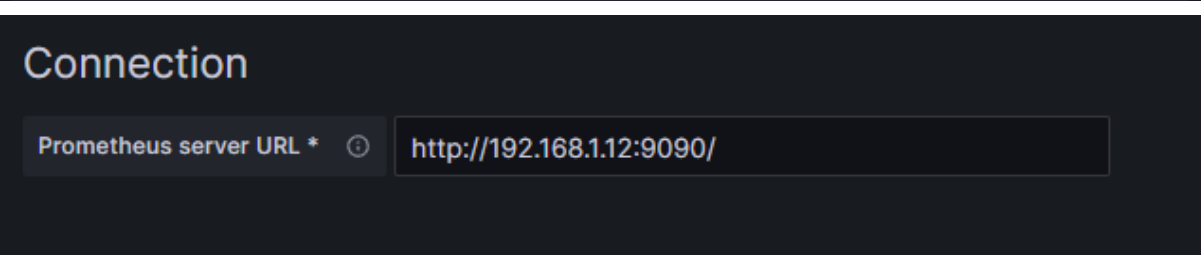
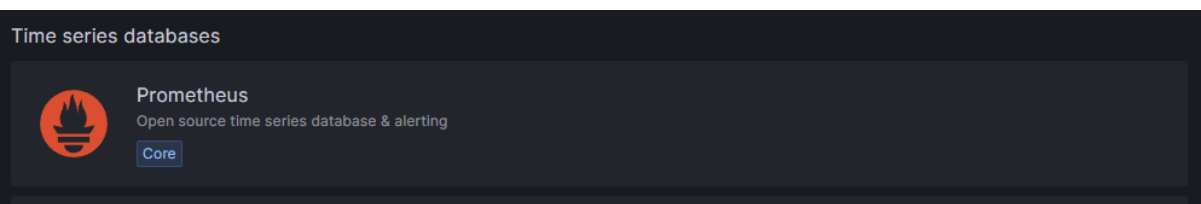
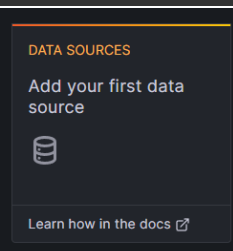
En la ruta que obtuvimos ipv4 + puerto 8181

<http://192.168.1.12:8181/metrics>

Debería obtener una respuesta con información

Entrar en grafana con admin admin y skipper cambiar la contraseña


<http://localhost:3000/login>



✓ Successfully queried the Prometheus API.

DASHBOARDS

Create your first dashboard



[Learn how in the docs](#)

Start your new dashboard by adding a visualization

Select a data source and then query and visualize your data with charts, stats and tables or create lists, markdowns and other widgets.

[+ Add visualization](#)

Add a library panel

Add visualizations that are shared with other dashboards.

[+ Add library panel](#)


Import a dashboard

Import dashboard from file or grafana.com.

[📁 Import dashboard](#)

Import dashboard

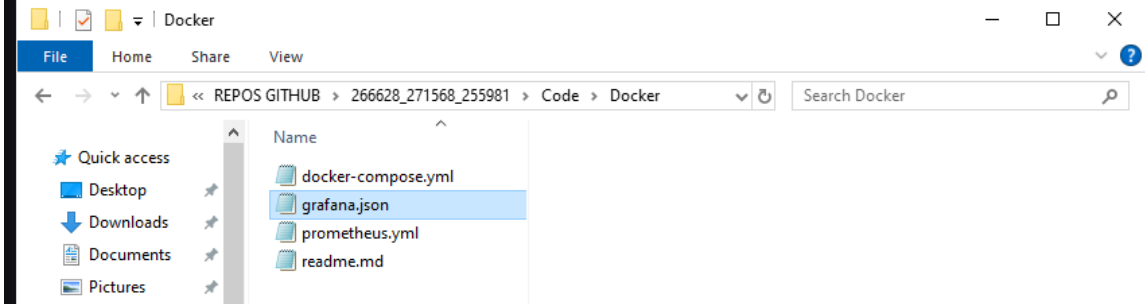
Import dashboard from file or Grafana.com



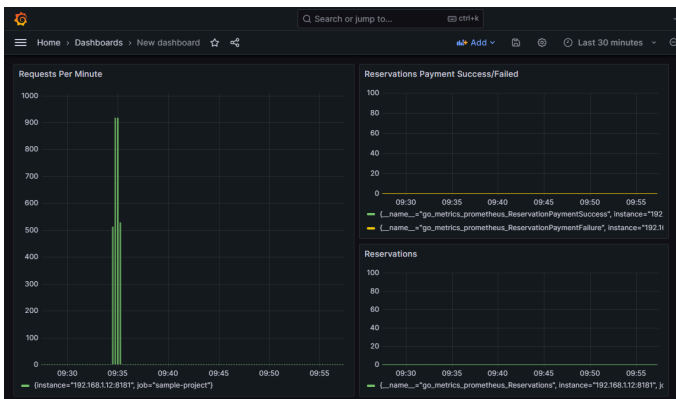
Upload dashboard JSON file

Drag and drop here or click to browse

Accepted file types: .json, .txt



Y listo, ya estaría todo en funcionamiento



Para futuras corridas

```
cd code/Docker  
docker-compose up
```

Luego se debe hacer en

```
cd code/Payment-Module  
go run main.go serve
```

Luego se debe hacer en

```
cd code/mongo-server  
go run main.go serve
```

Si se desea obtener credenciales de auth de github iniciar en

```
cd code/oAuth  
go run main-git.go serve
```

Finalmente se debe prender el sistema yendo a

```
cd "code/Database Access"  
go run main.go serve
```

Se adjunta una guía de como realizar algunas de las funciones del sistema en Docs/flujo.md.