MEMORIA FINAL DE PROYECTO

StudentsRooms

CICLO FORMATIVO DE GRADO SUPERIOR

Desarrollo de Aplicaciones Multiplataforma

AUTOR

Francisco Toribio Respaldo

TUTOR

José Luis Gallego García

Licencia

AGRADECIMIENTOS

Al sistema de enseñanza pública.

INDICE

1 INTRODUCCIÓN

Este documento recoge el trabajo realizado para el **módulo de Proyecto** del CFGS en **Desarrollo de Aplicaciones Multiplataforma.**

Este módulo profesional complementa la formación establecida para el resto de los módulos profesionales que integran el título en las funciones de **análisis** del contexto, **diseño y desarrollo** del proyecto, así como la organización de la **ejecución**.

2 ALCANCE DEL PROYECTO Y ANÁLISIS PREVIO

El propósito principal del proyecto es proponer una aplicación donde los estudiantes puedan encontrar habitaciones en alquiler cerca de sus centros de estudios y, los propietarios de los pisos tengan un sitio donde ofertarlas.

Este proyecto sirve de carta de presentación de las habilidades técnicas adquiridas durante la formación.

El desarrollo de este proyecto se ha llevado a cabo en varias fases: estudio de viabilidad, análisis, diseño, implementación, pruebas, explotación y ejecución. A continuación, se detallan las actividades de cada una de estas fases.

3 ESTUDIO DE VIABILIDAD

El proyecto se realiza con los recursos materiales e inmateriales individuales del único integrante del equipo. El plazo de desarrollo de la aplicación es de dos meses.

Se trata de una iniciativa personal orientada a fortalecer la capacitación en las principales tecnologías utilizadas en el mercado de software actual. Para el desarrollo del proyecto se emplearán los frameworks de mayor demanda y relevancia. Asimismo, se cuenta con un ordenador portátil y conexión a internet, lo que garantiza las condiciones necesarias para su ejecución.

3.1 Estado actual del sistema

El proyecto es de nueva creación, no hay un sistema previo. Tras estudiar varias soluciones alternativas, se deciden las tecnologías, lenguajes y herramientas para la creación de la solución que aporten mayor valor a la demostración de competencias profesionales adquiridas.

3.2 Resumen de requisitos

- ✓ **Arquitectura robusta y escalable** utilizando **Spring Boot** para construir una **API** modular y fácilmente escalable.
- ✓ **Integración sencilla con bases de datos.** Uso de **Spring Data JPA** y soporte nativo para múltiples motores (PostgreSQL, MySQL, Oracle, etc.). Conexión y gestión de datos simple y eficiente para reducir el tiempo de desarrollo y evitar errores comunes en la capa de persistencia.
- ✓ **Seguridad integrada** con **Spring Security** para implementar autenticación y autorización con JWT de forma estándar y confiable. Esto es clave para proteger los datos que consume la aplicación cliente.
- ✓ **Estandarización y buenas prácticas** con **Spring** para seguir patrones reconocidos (REST, MVC, MVVM, IoC, etc.). Garantizar que la API esté alineada con los estándares de la industria y sea más fácil de mantener y documentar.

- ✓ **Compatibilidad multiplataforma.** API REST desarrollada con Spring que pueda ser consumida por Android, aplicaciones web, iOS u otros clientes, lo que amplía el alcance del backend.
- ✓ **Productividad y rapidez en el desarrollo.** Spring Boot ofrece configuraciones automáticas, plantillas y herramientas que aceleran la creación de endpoints y reducen la necesidad de código repetitivo.
- ✓ **Comunidad y soporte.** Spring es uno de los frameworks más usados en el mundo Java, con una comunidad muy activa y abundante documentación, lo que facilita resolver problemas y encontrar ejemplos prácticos.
- ✓ La **aplicación** que se comunica con la **API** debe ser **Android** vía **HTTP/HTTPS** usando **JSON**, para simplificar la integración.
- ✓ **Por su amplia base de usuarios,** Android es el sistema operativo móvil más utilizado en el mundo, lo que garantiza que la API pueda ser consumida por una gran cantidad de dispositivos y usuarios.
- ✓ **Flexibilidad en la integración.** Android ofrece librerías y frameworks (como Retrofit, Volley u OkHttp) que simplifican la comunicación con APIs REST, manejando peticiones HTTP/HTTPS y respuestas JSON de forma eficiente.
- ✓ **Desacoplamiento de la lógica.** La aplicación Android se centra en la experiencia de usuario, mientras que la lógica de negocio y el acceso a datos se gestionan en el backend mediante la API. Esto facilita el mantenimiento y la escalabilidad.
- ✓ **Actualizaciones transparentes.** Al consumir datos desde la API, cualquier cambio en el backend se refleja automáticamente en la aplicación sin necesidad de reinstalarla, siempre que el contrato de la API se mantenga.
- ✓ **Seguridad y autenticación.** Android puede integrar fácilmente mecanismos de autenticación como **JWT, OAuth2 o API Keys**, garantizando un acceso seguro a los datos.

- ✓ **Compatibilidad con múltiples dispositivos** La API puede ser consumida por smartphones, tablets, televisores y otros dispositivos Android, ampliando el alcance de la solución.
- ✓ **Soporte offline y sincronización.** Android permite cachear datos y sincronizarlos con la API cuando hay conexión, mejorando la experiencia del usuario en entornos con conectividad limitada.
- ✓ **Ecosistema de herramientas.** Android Studio y sus librerías ofrecen soporte para depuración, pruebas y monitoreo del consumo de la API, lo que acelera el desarrollo y mejora la calidad del producto.
- ✓ **Spring Boot con PostgreSQL en Docker**. Arquitectura que encaja perfectamente con el objetivo del proyecto.
- ✓ Contenedores para la API y la base de datos para un despliegue reproducible.
- ✓ Tener una **API Spring Boot con PostgreSQL en Docker** ya que Android se convierte en un cliente ideal porque puede consumir endpoints REST de forma nativa.
- ✓ Flexibilidad para escalar hacia otras plataformas (iOS, web) sin cambiar la lógica del servidor.

3.3 Posibles soluciones

Otras posibles herramientas y tecnologías para el desarrollo de la API del proyecto serían:

Tecnologías para desarrollar APIs

Node.js + Express

- Ligero y rápido, ideal para aplicaciones en tiempo real.
- Gran ecosistema de librerías NPM.
- Fácil de integrar con bases de datos NoSQL como MongoDB.

Django REST Framework (Python)

- Ofrece rapidez de desarrollo y seguridad integrada.
- Excelente para proyectos que requieren prototipado rápido.
- Comunidad activa y abundante documentación.

Ruby on Rails (con API mode)

- Productividad muy alta gracias a convenciones sobre configuración.
- Ideal para startups y MVPs (Minimum Viable Product o Producto Mínimo Viable).
- Buen soporte para REST y GraphQL.

ASP.NET Core (C#)

- Rendimiento muy alto en entornos Microsoft.
- Integración nativa con Azure y servicios cloud.
- Seguridad robusta y soporte empresarial.

Go (Golang) con Gin o Echo

- APIs muy rápidas y eficientes en consumo de recursos.
- Ideal para microservicios y sistemas de alta concurrencia.
- Sintaxis simple y compilación directa.

GraphQL (sobre Node.js, Java o Python)

- Alternativa a REST que permite consultas más flexibles.
- Reduce el consumo de datos en móviles al traer solo lo necesario.
- Muy popular en proyectos modernos con múltiples clientes.

Tecnologías móviles para consumir APIs

iOS (Swift/Objective-C)

- Integración nativa con APIs REST y GraphQL.
- Ecosistema sólido y gran experiencia de usuario.

Flutter (Dart)

- Framework multiplataforma (Android e iOS con un solo código).
- Fácil consumo de APIs con librerías como http o dio.
- Gran rendimiento y comunidad creciente.

React Native (JavaScript/TypeScript)

- Permite compartir lógica con aplicaciones web.
- Amplio ecosistema de librerías para consumo de APIs.
- Buena opción para equipos con experiencia en frontend web.

Kotlin Multiplatform Mobile (KMM)

- Reutiliza lógica de negocio en Android e iOS.
- Integración nativa con APIs REST y GraphQL.
- Ideal para trabajar con Kotlin en Android.

3.4 Solución elegida

Spring Boot con Android.

Aunque existen muchas opciones, Spring Boot con Android destaca porque ofrece robustez, seguridad, escalabilidad para la API y soporte empresarial.

Android garantiza alcance masivo y flexibilidad en el consumo de datos.

La combinación es madura, estable y con gran comunidad, lo que facilita soporte y mantenimiento.

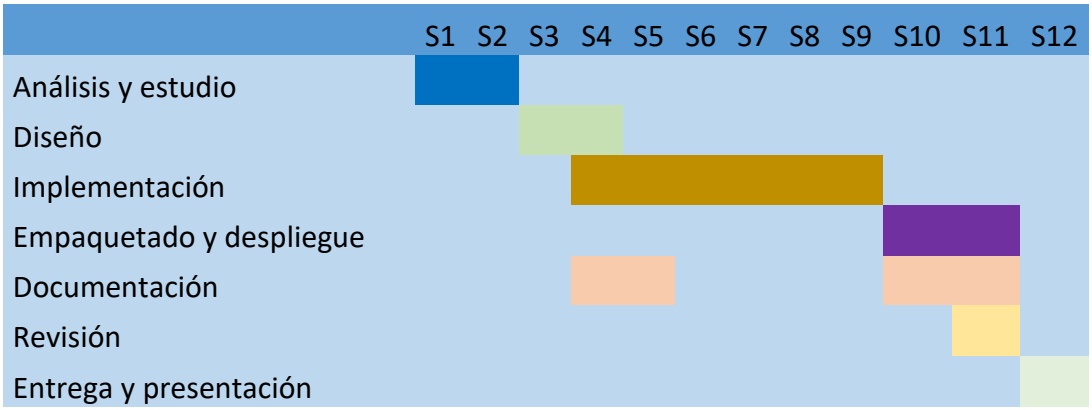
Aquí empieza el desarrollo de un nuevo proyecto ***"StudentsRooms".***

3.5 Planificación temporal de las tareas del proyecto 'StudentsRooms'

El proyecto está dividido en varios bloques de acciones o tareas:

- Búsqueda y estudio del dominio en el que basar el proyecto.
- Análisis y formación en las tecnologías del desarrollo.
- Diseño inicial de la aplicación.
- Realización y entrega del anteproyecto técnico.
- Modelar, codificar, probar y desplegar la aplicación.
- Documentar la aplicación.
- Realizar la memoria del proyecto.
- Preparar la presentación del proyecto ante el tribunal.
- Entrega y presentación final del proyecto.

Cuadro inicial de previsión de las tareas a realizar y la estimación del tiempo invertido dividido en semanas.



La previsión inicial de tareas y el tiempo estimado se ha cumplido. El tiempo utilizado para la implementación se ha ampliado hasta la penúltima semana de desarrollo, solapándose con las tareas de empaquetado, despliegue, documentación y revisión.

3.6 Planificación de los recursos a utilizar

Para solventar los problemas que plantea el proyecto no será necesario contratar personal. Para el desarrollo de '***StudentsRooms'*** bastará con un portátil, una conexión a internet, estudio de la documentación específica y tiempo de dedicación.

- **Recursos humanos**: Al ser un proyecto individual todas las tareas las realiza el autor del proyecto con el asesoramiento y guía del tutor.
- **Recursos materiales**: Para el desarrollo se han usado un ordenador portátil y una conexión a internet.
- **Recursos inmateriales** usados:
    - **Spring**, framework de desarrollo de la API escrita con Java.
    - **Spring JPA**, ORM / biblioteca para el manejo de bases de datos.
    - **PostgreSQL**, sistema de gestión de bases de datos relacionales.
    - **Maven,** herramienta de gestión y construcción de proyectos Java.
    - **Gradle,** herramienta de automatización de construcción de software (build system) que permite compilar, probar y empaquetar proyectos de manera eficiente, especialmente en entornos Java y Android.
    - **Otros lenguajes y tecnologías de programación**, Kotlin, Groovy, HTML, CSS, SCSS, JS, BOOTSTRAP, MD, UML, XML, YAML.
    - **Draw.io**, herramienta de diagramas.
    - **Visual Studio Code**, editor de código.
    - **Android Studio,** IDE de desarrollo móvil.
    - **Windows 11, Power Shell** y **cmd** sistema operativo y terminales.
    - **Git, GitHub**, sistema de control de versiones del código.
    - **ChatGPT, Gemini, Copilot,** inteligencias artificiales**.**
    - **Docker Desktop y Docker,** aplicación de gestión y plataforma de contenedores.
    - **Word, Excel, PowerPoint, Outlook,** herramientas ofimáticas.
    - **Otras herramientas, aplicaciones y conocimientos.**

4 ANÁLISIS

Establecimiento de los requisitos del sistema.

4.1 Diagrama de casos de uso.

Diagrama de casos de uso del administrador (ADMIN).

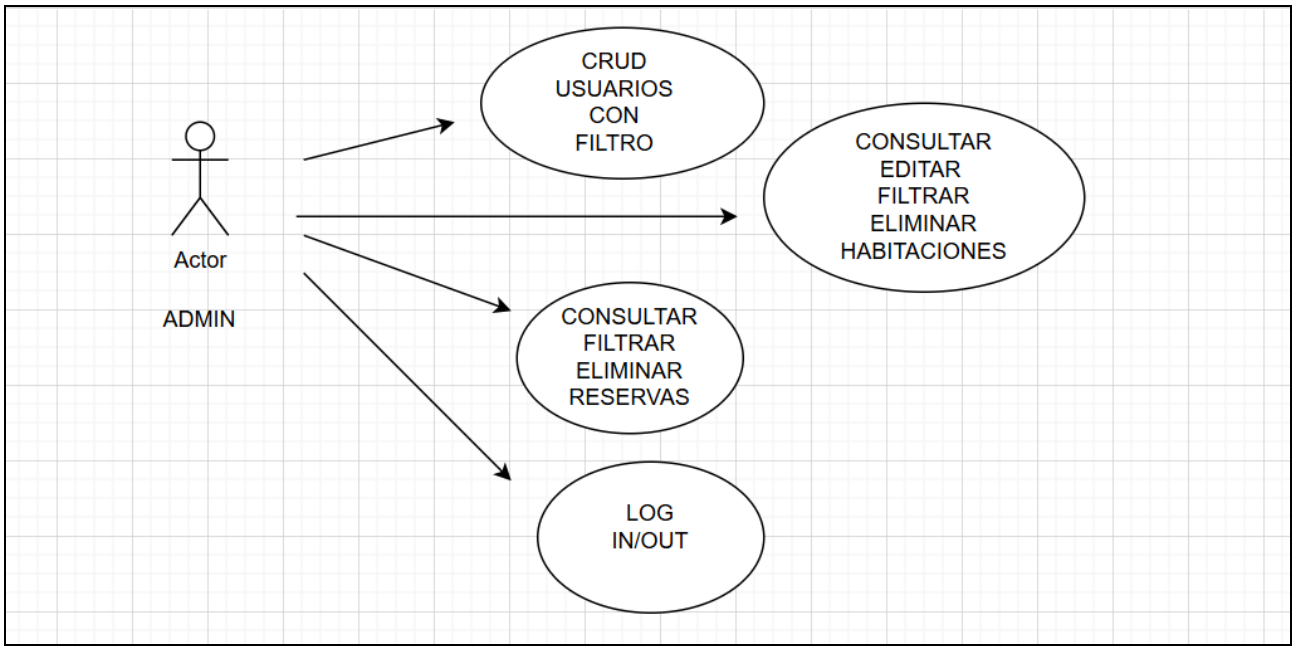


Diagrama de casos de uso del USUARIO.

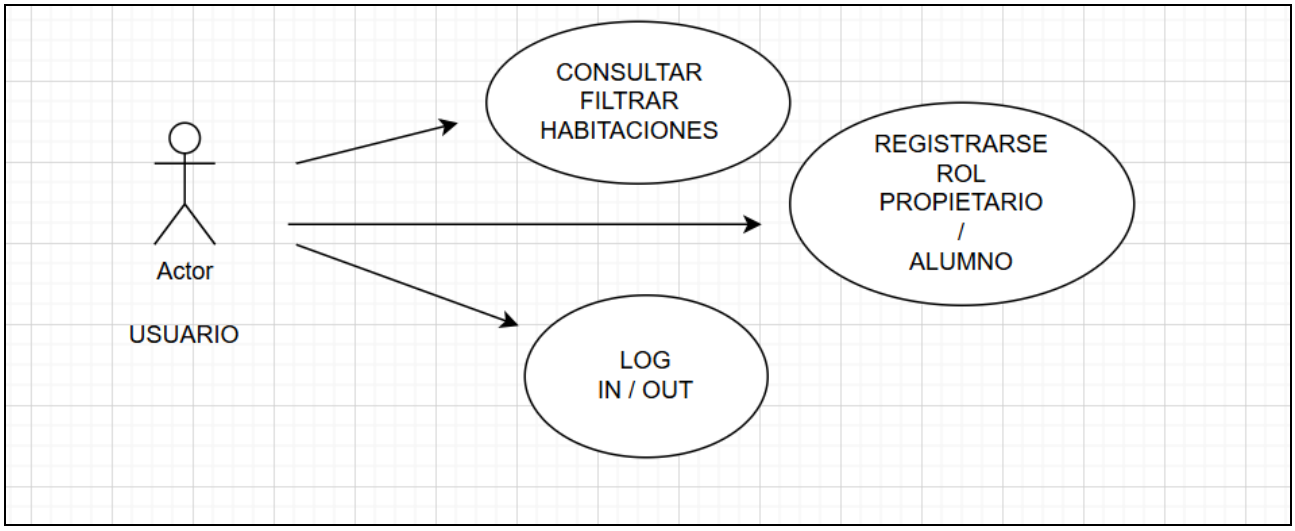


Diagrama de casos de uso del PROPIETARIO.

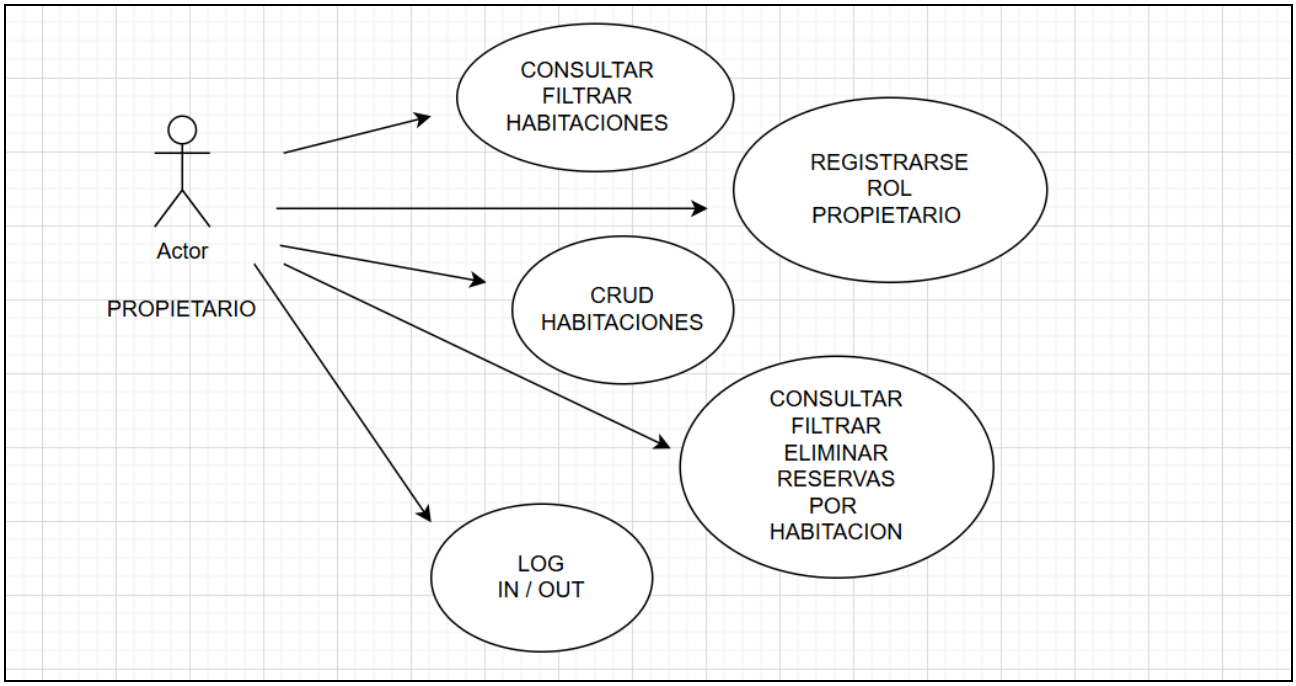
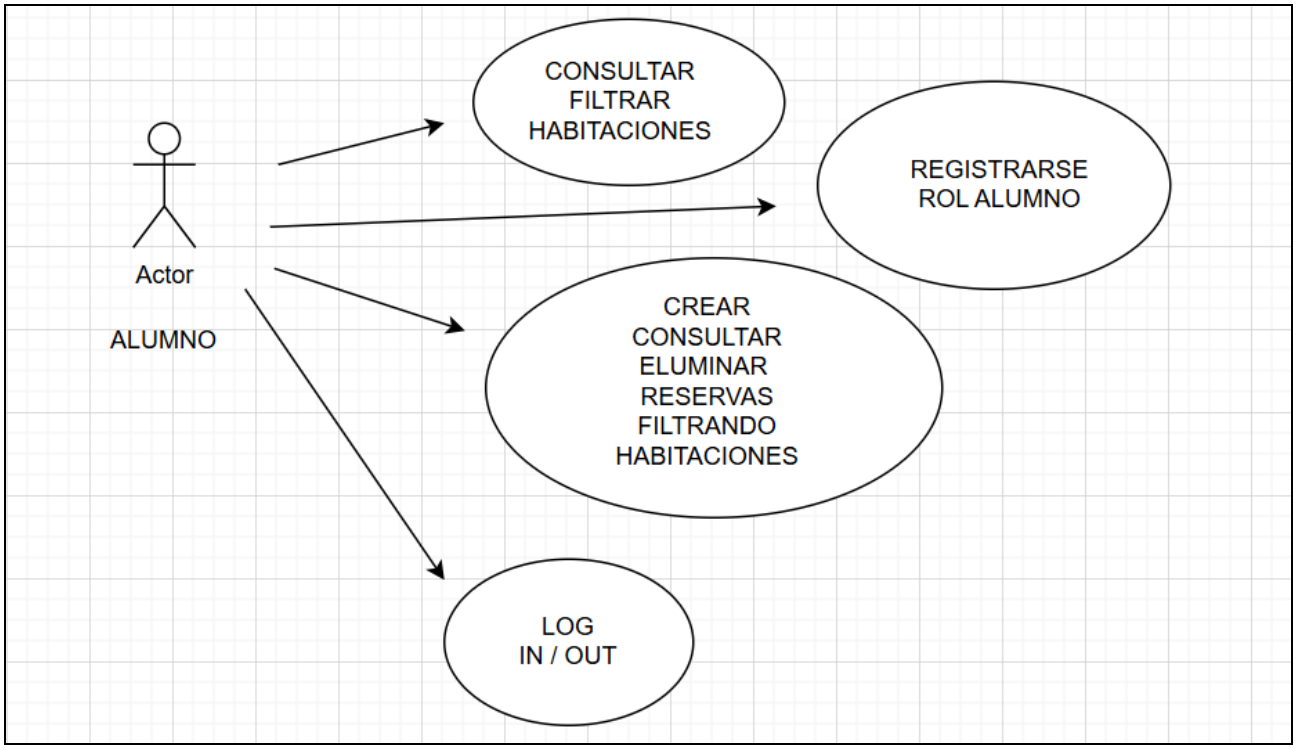
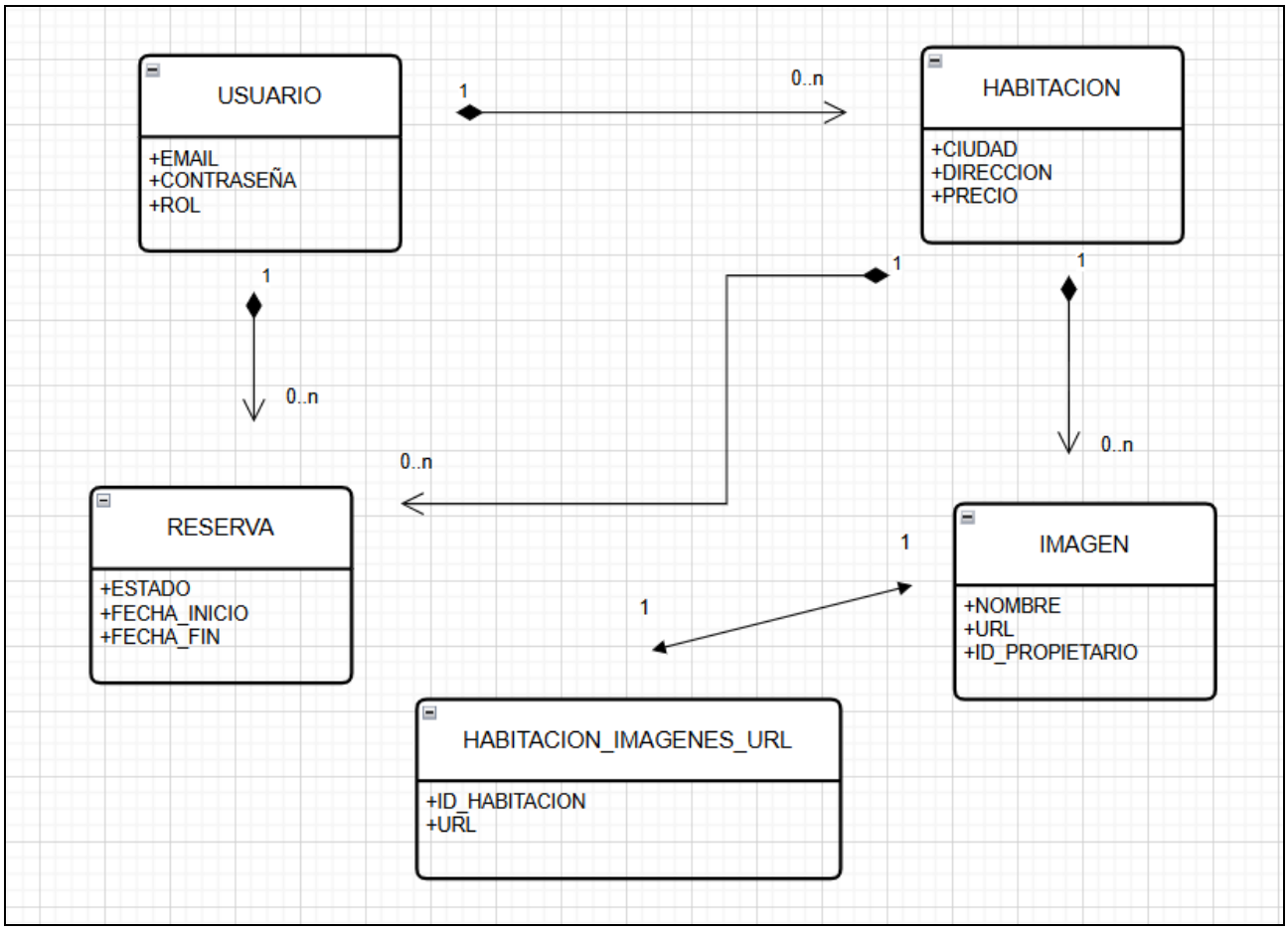


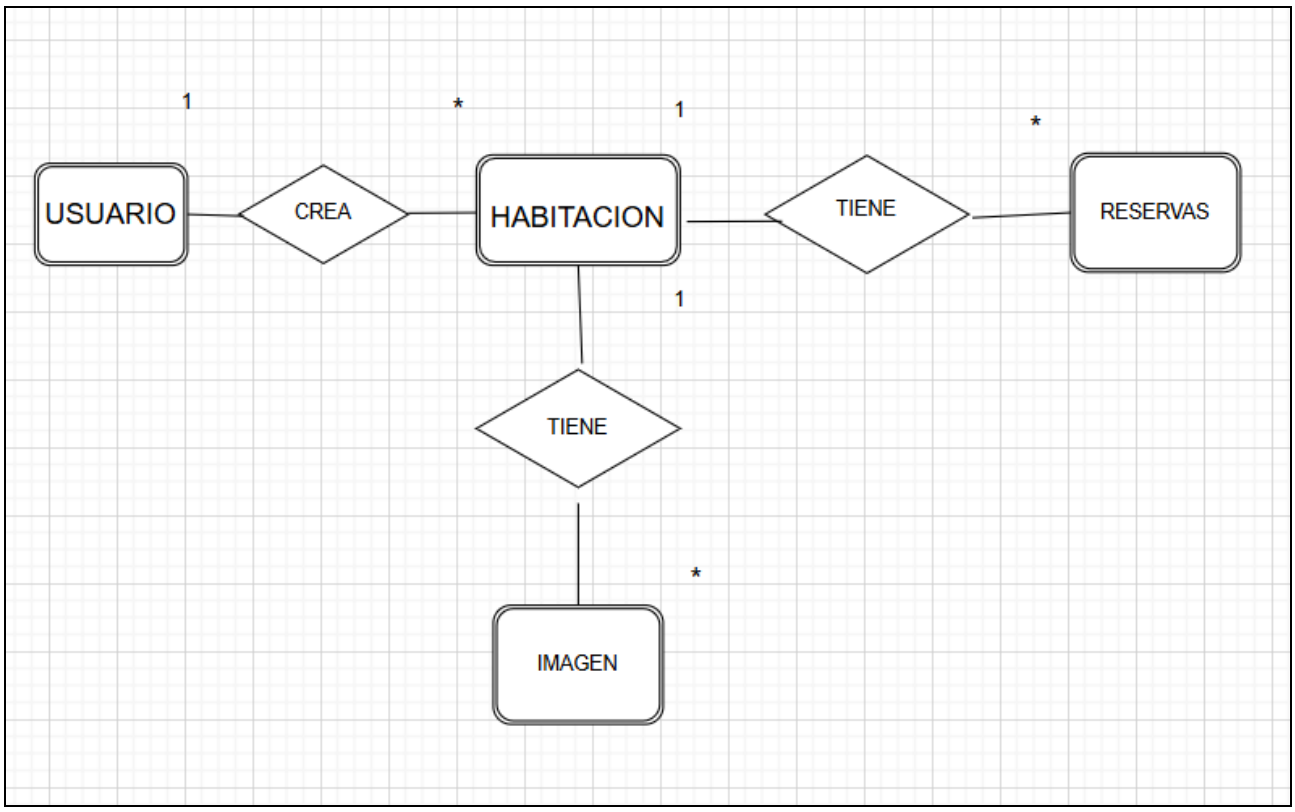
Diagrama de casos de uso del ALUMNO.



4.2 Modelo de datos

Diagramas de clases y de entidad relación (E/R).





Base de datos, claves foráneas referencias y restricciones.

TABLAS

```
mydatabase=# \dt
                List of tables
 Schema |          Name           | Type  | Owner
--------+-------------------------+-------+--------
 public | habitacion              | table | myuser
 public | habitacion_imagenes_url | table | myuser
 public | imagen                  | table | myuser
 public | reserva                 | table | myuser
 public | usuario                 | table | myuser
(5 rows)
```

USUARIO:

```
mydatabase=# \d usuario
                    Table "public.usuario"
   Column   |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 id         | uuid                   |           | not null |
 contraseña | character varying(255) |           |          |
 email      | character varying(255) |           |          |
 nombre     | character varying(255) |           |          |
 rol        | character varying(255) |           |          |
Indexes:
    "usuario_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "usuario_rol_check" CHECK (rol::text = ANY (ARRAY['ALUMNO'::character varying, 'PROPIETARIO'::character varying, 'ADMIN'::character varying]::
text[]))
Referenced by:
    TABLE "habitacion" CONSTRAINT "fk68vdor41uae3i4aq51j56cokn" FOREIGN KEY (propietario_id) REFERENCES usuario(id)
    TABLE "imagen" CONSTRAINT "fkd19pxifrmy9lnwi9s3hb83d6c" FOREIGN KEY (usuario_id) REFERENCES usuario(id)
    TABLE "reserva" CONSTRAINT "fkjar9wrcvgdxhp3o1pv7ge5x7d" FOREIGN KEY (propietario_id) REFERENCES usuario(id)
    TABLE "reserva" CONSTRAINT "fkpbab1y8d8vb54dk9wgx5w6pe6" FOREIGN KEY (alumno_id) REFERENCES usuario(id)
```

HABITACION:

```
mydatabase=# \d habitacion
                      Table "public.habitacion"
     Column     |          Type          | Collation | Nullable | Default
----------------+------------------------+-----------+----------+---------
 id             | uuid                   |           | not null |
 ciudad         | character varying(255) |           |          |
 descripcion    | character varying(255) |           |          |
 direccion      | character varying(255) |           |          |
 precio_mensual | numeric(38,2)          |           |          |
 titulo         | character varying(255) |           |          |
 propietario_id | uuid                   |           | not null |
Indexes:
    "habitacion_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fk68vdor41uae3i4aq51j56cokn" FOREIGN KEY (propietario_id) REFERENCES usuario(id)
Referenced by:
    TABLE "habitacion_imagenes_url" CONSTRAINT "fkk3hbofe0b80s7w5vyaqyr0boe" FOREIGN KEY (habitacion_id) REFERENCES habitacion(id)
    TABLE "reserva" CONSTRAINT "fktr5bg864m3dseko7gif2bl239" FOREIGN KEY (habitacion_id) REFERENCES habitacion(id)
```

RESERVAS:

```
mydatabase=# \d reserva
                      Table "public.reserva"
     Column     |          Type          | Collation | Nullable | Default
----------------+------------------------+-----------+----------+---------
 id             | uuid                   |           | not null |
 estado_reserva | character varying(255) |           |          |
 fecha_fin      | date                   |           |          |
 fecha_inicio   | date                   |           |          |
 alumno_id      | uuid                   |           |          |
 habitacion_id  | uuid                   |           |          |
 propietario_id | uuid                   |           |          |
Indexes:
    "reserva_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "reserva_estado_reserva_check" CHECK (estado_reserva::text = ANY (ARRAY['PENDIENTE'::character varying, 'CONFIRMADA'::character varying, 'CANC
ELADA'::character varying]::text[]))
Foreign-key constraints:
    "fkjar9wrcvgdxhp3o1pv7ge5x7d" FOREIGN KEY (propietario_id) REFERENCES usuario(id)
    "fkpbab1y8d8vb54dk9wgx5w6pe6" FOREIGN KEY (alumno_id) REFERENCES usuario(id)
    "fktr5bg864m3dseko7gif2bl239" FOREIGN KEY (habitacion_id) REFERENCES habitacion(id)
```

IMAGEN:

```
mydatabase=# \d imagen
                      Table "public.imagen"
   Column   |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 id         | uuid                   |           | not null |
 filename   | character varying(255) |           |          |
 url        | character varying(255) |           |          |
 usuario_id | uuid                   |           |          |
Indexes:
    "imagen_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "fkd19pxifrmy9lnwi9s3hb83d6c" FOREIGN KEY (usuario_id) REFERENCES usuario(id)
```

HABITACION_IMAGENES_URL

```
mydatabase=# \d habitacion_imagenes_url
               Table "public.habitacion_imagenes_url"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 habitacion_id | uuid                   |           | not null |
 imagenes_url  | character varying(255) |           |          |
Foreign-key constraints:
    "fkk3hbofe0b80s7w5vyaqyr0boe" FOREIGN KEY (habitacion_id) REFERENCES habitacion(id)
```

4.3 Requisitos funcionales

Determinan qué tareas tiene que hacer el sistema:

- ✓ Proporcionar un sistema de autorización y autenticación para todos los usuarios con JWT.
- ✓ Gestionar las entidades (CRUD completo: crear, leer, actualizar, eliminar).
- ✓ Exposición de endpoints RESTful para consumo desde clientes móviles.
- ✓ Guardar las contraseñas encriptadas en la base de datos.
- ✓ Persistencia de datos almacenada en una base de datos PostgreSQL con JPA/Hibernate.
- ✓ Proporcionar un sistema de búsqueda y filtros.
- ✓ Proporcionar al administrador un sistema de creación, lectura, modificación y borrado de entidades.
- ✓ Validar datos de entrada en peticiones HTTP.
- ✓ Registro de actividad y errores mediante logs.
- ✓ Configuración de CORS para permitir acceso desde Android.
- ✓ Pantalla de login y registro conectado a la API.
- ✓ Consumo de endpoints REST usando Retrofit, OkHttp o Volley.
- ✓ Visualización y manipulación de datos obtenidos del backend.
- ✓ Gestión de sesiones y tokens JWT para mantener autenticación.
- ✓ Interfaz adaptativa y responsiva para distintos tamaños de pantalla.
- ✓ Manejo de errores y mensajes de estado (conexión, validación, etc.).
- ✓ Sincronización de datos en segundo plano si es necesario.

4.4 Requisitos no funcionales

- ✓ La aplicación debe cargarse de forma fluida bajo condiciones normales de uso.
- ✓ La aplicación debe ser capaz de manejar múltiples usuarios concurrentes sin degradar el rendimiento.
- ✓ Escalabilidad: capacidad para manejar aumento de usuarios y peticiones.

- ✓ Seguridad: protección contra ataques comunes (XSS, CSRF, inyecciones SQL).
- ✓ Disponibilidad: alta disponibilidad con despliegue en contenedores Docker.
- ✓ Mantenibilidad: código modular, bien documentado y con buenas prácticas.
- ✓ Rendimiento: tiempos de respuesta óptimos (<500ms en operaciones comunes).
- ✓ La aplicación debe implementar autenticación y autorización seguras para los usuarios.
- ✓ La aplicación debe usar el protocolo HTTPS para todas las comunicaciones en producción.
- ✓ La aplicación debe cumplir con las normativas de protección de datos asegurando que la información personal de los usuarios esté protegida.
- ✓ El código debe seguir las mejores prácticas de Spring y Android incluyendo el uso de estándares de codificación y patrones de diseño.
- ✓ La aplicación debe incluir una suite de pruebas automatizadas para facilitar la detección y corrección de errores.
- ✓ La aplicación debe ser fácilmente desplegable en diferentes entornos (desarrollo, prueba y producción) utilizando herramientas de automatización como Docker.
- ✓ La interfaz de usuario debe ser intuitiva y accesible cumpliendo con las pautas de accesibilidad de WCAG 2.1.
- ✓ La aplicación en su versión web debe ser compatible con los navegadores más utilizados (Chrome, Safari, Edge) y adaptarse a diferentes tamaños de pantalla (responsive design).
- ✓ La aplicación debe tener un diseño atractivo que facilite su usabilidad.
- ✓ La aplicación debe estar abierta a futuras mejoras y a su ampliación.
- ✓ Portabilidad: despliegue en distintos entornos (local, cloud, CI/CD).
- ✓ Trazabilidad: logs estructurados y trazas para auditoría.
- ✓ Compatibilidad: soporte para múltiples versiones de Android.
- ✓ Seguridad: almacenamiento seguro de credenciales y tokens.

5 DISEÑO

Diseño tecnológico de la solución, documento técnico de desarrollo de requisitos para API Spring Boot y aplicación Android. Introducción a la arquitectura del sistema:

El sistema está compuesto por una aplicación móvil Android que consume una API REST desarrollada con Spring Boot. La API se conecta a una base de datos PostgreSQL desplegada en un contenedor Docker. La comunicación entre cliente y servidor se realiza mediante HTTP/HTTPS, utilizando el formato JSON. La arquitectura sigue un modelo cliente-servidor desacoplado, lo que permite escalabilidad, mantenibilidad y seguridad.

Desarrollo de requisitos funcionales para el Backend (API Spring Boot).

Autenticación y autorización implementada mediante JWT. Se configura un filtro personalizado (OncePerRequestFilter) y se define la política de seguridad en SecurityConfig.

CRUD de entidades utilizando Spring Data JPA para definir repositorios, servicios y controladores REST. Las operaciones se exponen mediante anotaciones como @GetMapping, @PostMapping, etc.

Exposición de endpoints RESTful definiendo rutas claras y documentadas. Se utiliza @RestController para exponer los servicios.

Validación de datos aplicando anotaciones como @NotNull, @Size, @Email en los DTOs, y se valida con @Valid en los controladores.

Persistencia en base de datos PostgreSQL configurando application.properties para conectar con PostgreSQL. Las entidades JPA se relacionan mediante @OneToMany, @ManyToOne, etc.

Registro de actividad y errores. Se integran SLF4J o Logback para generar logs estructurados. Se configuran niveles de log.

Configuración de CORS habilitada en WebMvcConfigurer para permitir el acceso desde la aplicación Android.

Frontend (Aplicación Android)

Pantalla de login y registro desarrollando interfaces con Text, Button y lógica en ViewModel. Se consume la API con Retrofit.

Consumo de endpoints REST. Se utiliza Retrofit + Gson para realizar peticiones HTTP. Las respuestas se gestionan con Flow.

Visualización y manipulación de datos empleando CardView y ViewModel para mostrar datos dinámicos.

Gestión de sesiones y tokens JWT almacenando los tokens en SharedPreferences o EncryptedSharedPreferences.

Interfaz adaptativa y responsiva usando Material Design para adaptar la UI a distintos dispositivos.

Manejo de errores y mensajes de estado capturando excepciones en Retrofit y mostrando mensajes con Toast, Snackbar o Dialog.

Sincronización de datos para tareas en segundo plano y sincronización periódica.

Desarrollo de requisitos no funcionales en el backend (API Spring Boot).

Escalabilidad separando las capas (Controller, Service, Repository) y despliegue con Docker para modularidad.

Seguridad configurando HTTPS, JWT, validación de entrada y protección contra CSRF si se expone a web.

Disponibilidad utilizando Docker Compose con reinicio automático. Se puede monitorizar con Prometheus/Grafana.

Mantenibilidad aplicando principios SOLID, uso de DTOs,, MAPPERS y documentación con README.

Rendimiento implementando paginación (Pageable), caché con @Cacheable, y profiling para detectar cuellos de botella.

Portabilidad dockerizando la API y la base de datos. Se usan archivos .env para variables de entorno.

Frontend (Aplicación Android)

Usabilidad aplicando Material Design y navegación intuitiva con Navigation Component.

Rendimiento evitando el uso del hilo principal para operaciones pesadas. Se usan Coroutines y Flows.

Compatibilidad definiendo minSdkVersion y se prueba en emuladores y dispositivos reales.

Seguridad encriptando datos sensibles, usando HTTPS y validando certificados.

Confiabilidad implementando try-catch.

Actualizable separando la lógica en ViewModel y Repository para facilitar cambios y mantenimiento.

5.1 Estructura de la aplicación

Arquitectura de red

- **Cliente móvil (Android)** → envía peticiones HTTP/HTTPS (JSON).
- **Servidor backend (Spring Boot)** → recibe, procesa y responde a las solicitudes.
- **Base de datos (PostgreSQL en Docker)** → almacena y gestiona la información.
- Comunicación segura mediante **HTTP/HTTPS + JWT** para autenticación.

Backend (Spring Boot)

Organizado en **capas** siguiendo buenas prácticas:

- **Controller Layer**
  - Expone endpoints REST (@RestController).
  - Define rutas (/api/habitaciones, /api/propietarios, etc.).
  - Maneja solicitudes y respuestas en formato JSON.
- **Service Layer**
  - Contiene la lógica de negocio.
  - Aplica validaciones, reglas y procesos antes de interactuar con la base de datos.
  - Se implementa con @Service.
- **Repository Layer**
  - Acceso a la base de datos mediante **Spring Data JPA**.
  - Define interfaces que extienden JpaRepository.

    - Encapsula consultas SQL/JPQL.

- **Entity Layer**
    - Modelos de datos (@Entity) que representan tablas en PostgreSQL.
    - Relaciones (@OneToMany, @ManyToOne, etc).
- **Security Layer**
    - Configuración de **Spring Security**.
    - Filtros JWT para autenticación y autorización.
- **Infrastructure Layer**
    - Configuración de Docker, variables de entorno, y application.yml.
    - Integración para documentación.

Frontend (Android)

Organizado en **capas MVVM (Model-View-ViewModel)**:

- **View (UI)**
    - Pantallas (Activity).
    - Componentes visuales (Material Design).
    - Interacción con el usuario.
- **ViewModel**
    - Contiene la lógica de presentación.
    - Gestiona estados y datos con StateFlow.
    - Se comunica con el Repository para obtener información.

- Repository
    - Encapsula la lógica de acceso a datos.
    - Consume la API mediante Retrofit/OkHttp.
- **Model**
    - Clases de datos (data class) que representan las respuestas JSON de la API.
    - Conversión automática con Gson/Moshi.

Componentes del sistema

- **Android App**
    - UI adaptativa, gestión de sesión JWT.
- **Spring Boot API**
    - Endpoints REST, seguridad, validación, logs.
- **PostgreSQL en Docker**
    - Base de datos persistente, aislada en contenedor.
- **Docker Compose**
    - Orquestación de API + DB.

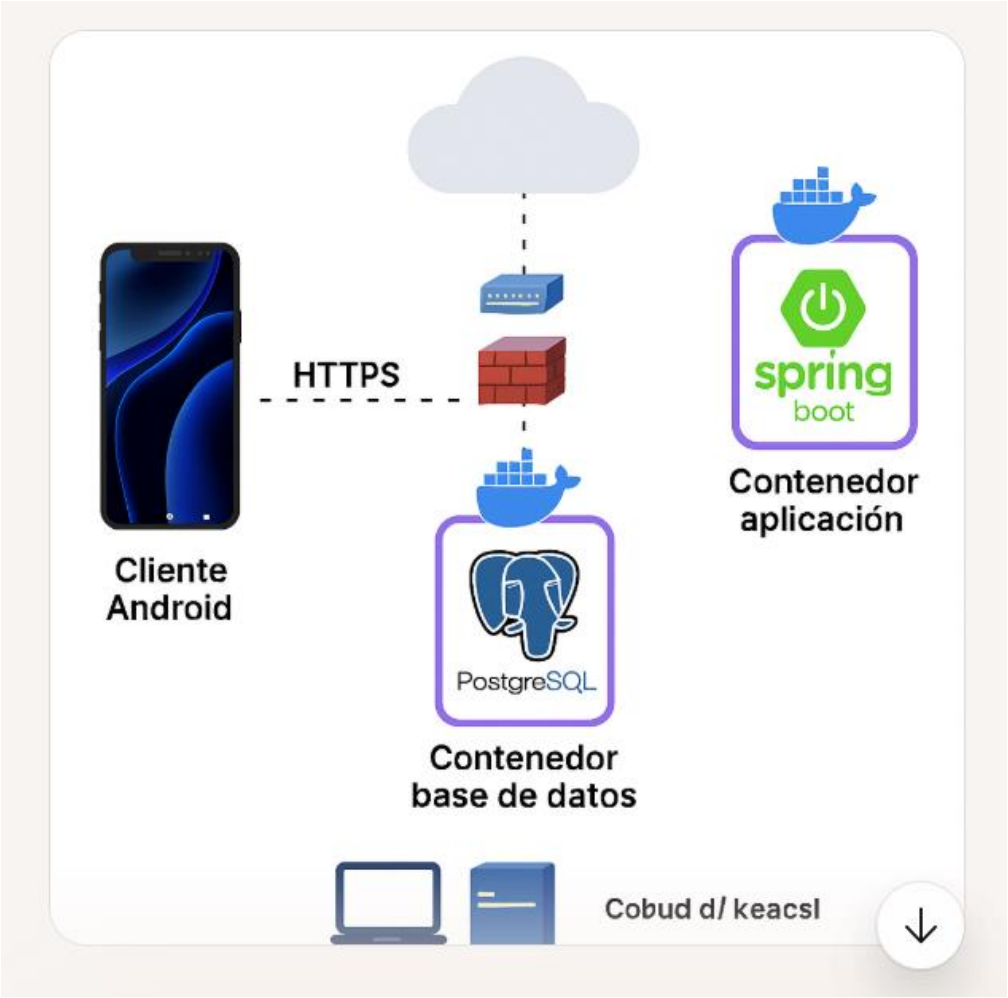
5.2 Componentes del sistema / arquitectura de red

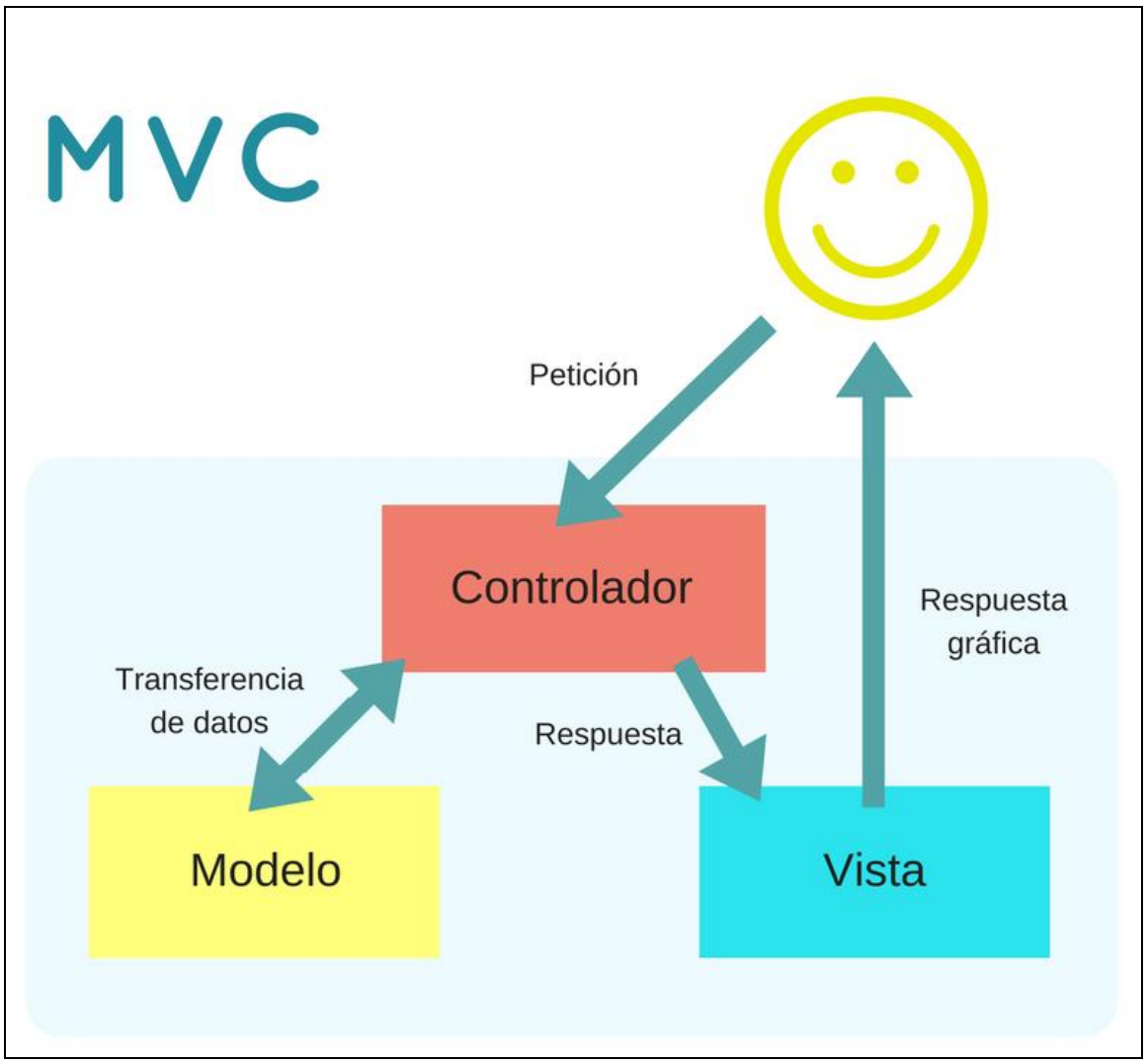
Dispositivos de red

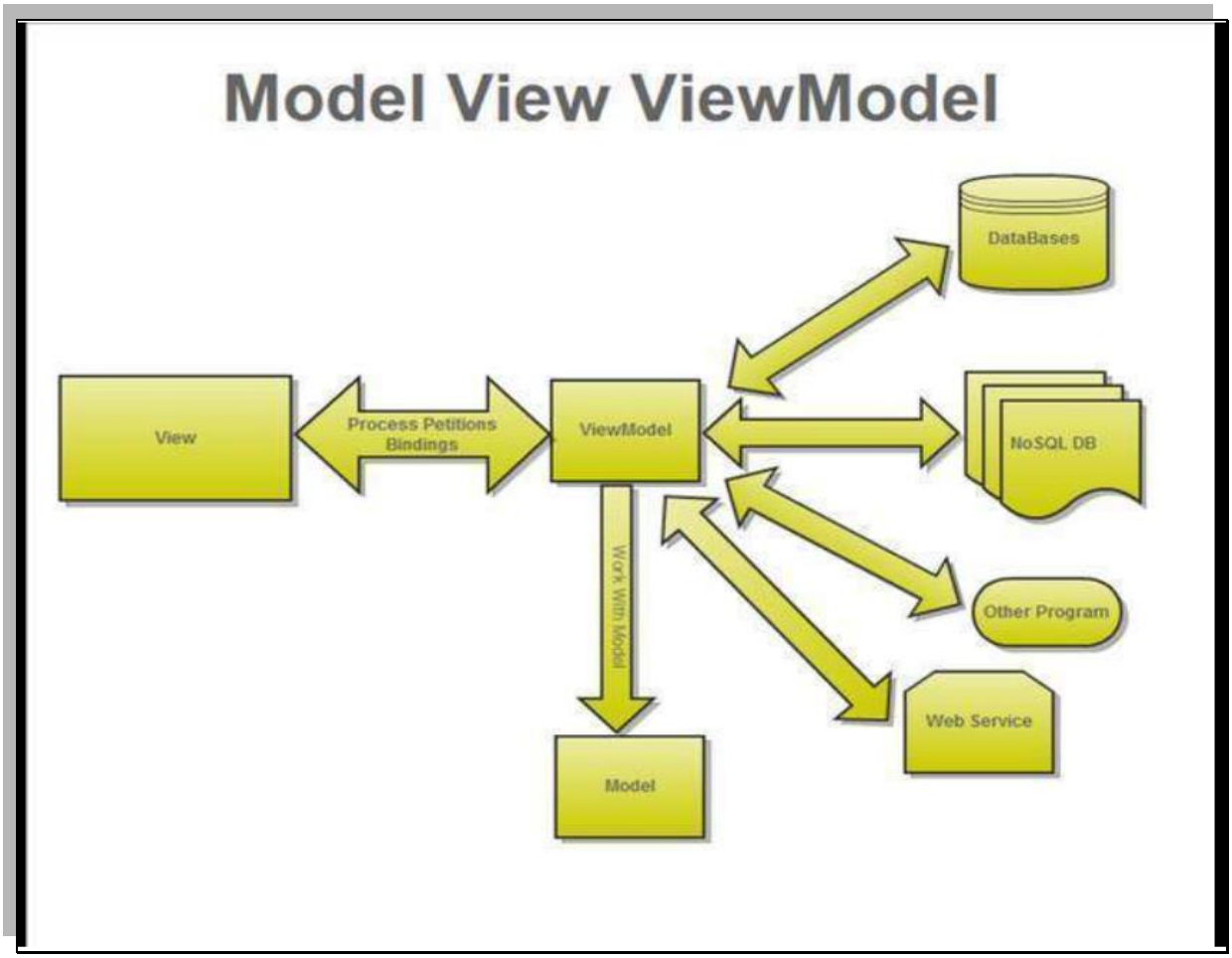
Router / Switch: Para conexión entre cliente Android y servidor backend.

Firewall: Para proteger el acceso externo al servidor.

Certificados SSL/TLS: Para asegurar la comunicación HTTPS.




MVC
Petición
Controlador
Respuesta
gráfica
Transferencia
de datos
Respuesta
Modelo
Vista


Model View ViewModel
DataBases
View
Process Petitions
Bindings
ViewModel
NoSQL DB
Work With Model
Other Program
Web Service
Model

5.3 Herramientas y teconologías utilizadas.

Categoría	Tecnología / Herramienta	Descripción / Uso
Plataformas	Android OS	Sistema operativo móvil para la app cliente
	Docker	Contenedores para backend y base de datos
	Docker Compose	Orquestación de servicios (API + DB)
	PostgreSQL	Motor de base de datos relacional
	GitHub / GitLab	Control de versiones y colaboración
Lenguajes	Java	Backend con Spring Boot
	Kotlin	Desarrollo de la app Android
	SQL	Consultas en PostgreSQL
	YAML	Configuración de Spring Boot y Docker Compose
	JSON	Intercambio de datos entre cliente y servidor
Servidores	Apache Tomcat (embebido)	Servidor web integrado en Spring Boot
	Docker host / Cloud VM	Despliegue local o en producción

Categoría	Tecnología / Herramienta	Descripción / Uso
Frameworks Backend	Spring Boot	Framework principal para la API REST
	Spring Data JPA	Acceso a datos con PostgreSQL
	Spring Security	Autenticación y autorización con JWT
	Hibernate	ORM para mapear entidades Java a tablas SQL
Frameworks Android	Android Jetpack	Arquitectura MVVM y navegación
	Retrofit + OkHttp	Consumo de API REST desde Android
	Gson / Moshi	Serialización/deserialización de datos JSON
	Material Design Components	UI moderna y responsiva
	Room / DataStore	Persistencia local y sincronización offline

Categoría	Tecnología / Herramienta	Descripción / Uso
Entornos de desarrollo	Android Studio	IDE oficial para desarrollo Android
	IntelliJ IDEA / VS Code	IDE para desarrollo backend con Spring Boot
	Postman / Insomnia	Pruebas y depuración de endpoints REST
	Docker CLI / Docker Desktop	Gestión de contenedores y redes
	Git + GitHub/GitLab CLI	Control de versiones y trabajo colaborativo

6 IMPLEMENTACIÓN

Enlaces a los repositorios con el código de la aplicación:

API SPRING

https://github.com/frantoribio/demo

APP ANDROID

https://github.com/frantoribio/alquilerapp-compose

Partiendo del diseño, en esta fase se construye el sistema.

6.1 Implementación del modelo de datos

El desarrollo del código está en dos repositorios públicos en GitHub.

Se puede descargar el código ejecutando comandos de git, clonando los repositorios o directamente descargando los archivos.

El **backend** también viene empaquetado en un **.jar** creando y arrancando la imagen de la base de datos desde un script .bat.

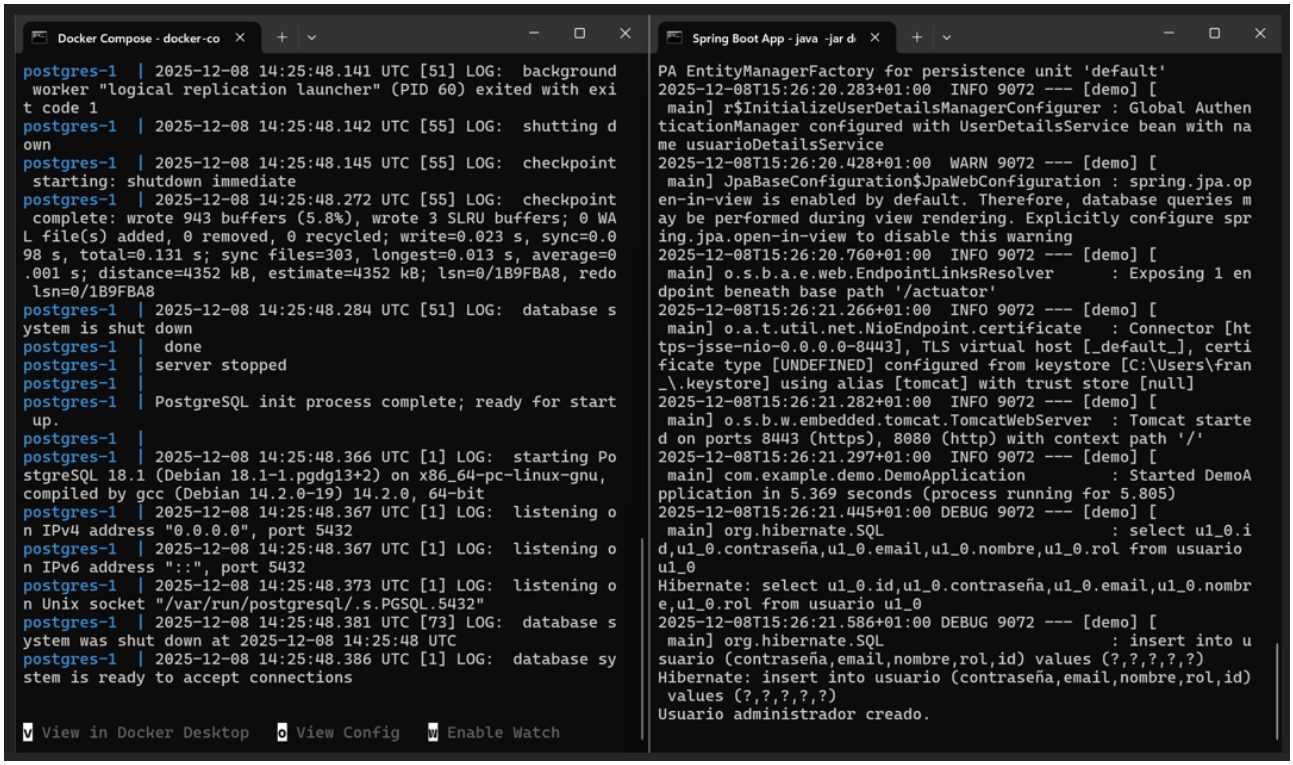
6.2 Carga de datos

Los datos de la aplicación los crean y cargan los usuarios. Al arrancar el sistema se crea automáticamente un usuario Administrador con una clave conocida por el propietario.

6.3 Configuraciones realizadas en el sistema

Es necesario la instalación de Docker.

El **backend** de la aplicación está orquestado en contenedores Docker independientes que se comunican entre ellos. La API Spring expone los **endpoints**, puede estar en una imagen de un servidor Linux corriendo en Docker o en un servidor local ejecutando en consola el **.jar** con todo lo necesario para su uso. La base de datos con la persistencia de la aplicación corre en otro contenedor **Docker** desde una imagen de **PostgresQl**, todo ello orquestado con un archivo **DockerCompose**.



6.4 Implementaciones de código realizadas

Se desarrolla en Java los modelos y entidades. La API se ha desarrollado siguiendo una **arquitectura en capas** basada en el patrón **MVC (Modelo–Vista–Controlador)**. Las capas principales son:

- **Entidad (Modelo):** Representa las tablas de la base de datos mediante clases Java anotadas con JPA.
- **Repositorio:** Gestiona el acceso a datos utilizando Spring Data JPA.
- **Servicio:** Contiene la lógica de negocio y validaciones.
- **Controlador:** Expone los endpoints REST para ser consumidos por la aplicación Android.

Este diseño favorece la separación de responsabilidades, la escalabilidad y el mantenimiento del sistema.

Las entidades representan los objetos principales del sistema: **Usuario, Habitación y Reserva**. Ejemplo de implementación:

```java
package com.example.demo.model;

import jakarta.persistence.*;
import java.util.*;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonManagedReference;

/**
 * Clase que representa un usuario en el sistema.
 */
@Entity
public class Usuario implements UserDetails{

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private UUID id;

    private String nombre;
    private String email;
    private String contraseña;

    @Enumerated(EnumType.STRING)
    private Rol rol;

    @OneToMany(mappedBy = "propietario", cascade = CascadeType.REMOVE, orphanRemoval = true)
    @JsonManagedReference
    @JsonIgnore
```

@Entity y **@Table** definen la clase como tabla en la base de datos.

@Id y **@GeneratedValue** gestionan la clave primaria.

@ManyToOne @OneToMany establecen las relaciones entre entidades.

Se han implementado interfaces que extienden **JpaRepository**, lo que permite realizar operaciones CRUD sin necesidad de código adicional.

```java
package com.example.demo.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.model.Habitacion;

import java.util.List;
import java.util.UUID;

/**
 * Repositorio para la entidad Habitacion.
 */
public interface HabitacionRepository extends JpaRepository<Habitacion, UUID> {

    List<Habitacion> findByPropietarioId(UUID propietarioId);
```

Esto facilita consultas personalizadas. Un ejemplo sería obtener habitaciones filtradas por precio máximo o ciudad.

La capa de servicio centraliza la lógica de negocio, evitando que los controladores gestionen directamente la persistencia.

```java
package com.example.demo.service;

import com.example.demo.dto.CrearHabitacionDto;
import com.example.demo.dto.HabitacionDTO;
import com.example.demo.model.Habitacion;
import com.example.demo.model.Usuario;
import com.example.demo.repository.HabitacionRepository;
import com.example.demo.repository.UsuarioRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;
import java.util.UUID;

/**
 * Servicio para gestionar operaciones relacionadas con Habitacion.
 * Proporciona métodos para CRUD y utiliza HabitacionMapper para conversiones DTO.
 */
@Service
public class HabitacionService {

    @Autowired
    private HabitacionRepository habitacionRepository;

    @Autowired
    private HabitacionMapper habitacionMapper;

    @Autowired
    private UsuarioRepository usuarioRepository;

    public HabitacionService(HabitacionRepository habitacionRepository, UsuarioRepository usuari
        this.habitacionRepository = habitacionRepository;
```

Aquí se encapsula la lógica de búsqueda de habitaciones.

Controladores (API REST)

Los controladores exponen los **endpoints** que la aplicación Android consume mediante peticiones **HTTP/HTTPS**.

```java
package com.example.demo.controller;

import java.util.*;
import java.util.logging.Logger;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.core.context.SecurityContextHolder;
import com.example.demo.model.Habitacion;
import com.example.demo.model.Usuario;
import com.example.demo.service.HabitacionService;
import com.example.demo.dto.CrearHabitacionRequest;
import com.example.demo.dto.HabitacionDTO;

/**
 * Controlador REST para gestionar habitaciones.
 */
@RestController
@RequestMapping("/api/habitaciones")
public class HabitacionController {

    @Autowired
    private HabitacionService habitacionService;
```

Seguridad y Autenticación

Se ha configurado **Spring Security** con **JWT (JSON Web Tokens)** para garantizar la autenticación y autorización:

- Los usuarios deben iniciar sesión para obtener un token.
- El token se incluye en cada petición para validar permisos.
- Se han definido roles: **Alumno, Propietario y Administrador**.

La aplicación Android se ha desarrollado siguiendo el patrón **MVVM (Model–View–ViewModel)**, que facilita la separación de responsabilidades y mejora la mantenibilidad del código. Las capas principales son:

Vista (Activity con funciones Screen): Interfaz gráfica que interactúa con el usuario.

ViewModel: Gestiona el estado de la interfaz y coordina la comunicación con el modelo.

Modelo: Contiene las clases de datos y la lógica de acceso a la API mediante Retrofit.

Este patrón asegura que la lógica de negocio no esté acoplada directamente a la interfaz.

Estos modelos se utilizan para mapear automáticamente las respuestas de la API a objetos Java.

Consumo de la API (Retrofit)

Se ha implementado **Retrofit** como cliente HTTP para consumir los endpoints REST de la API en Spring.

@GET define el tipo de petición.

@Query permite enviar parámetros en la URL.

Call<List<T>> gestiona la respuesta en forma de lista de objetos.

Repositorios

Los repositorios encapsulan la lógica de acceso a datos desde la API

ViewModel

El **ViewModel** coordina la comunicación entre la vista y el repositorio, manteniendo el estado de la interfaz.

Seguridad y Autenticación

La aplicación gestiona la autenticación mediante tokens **JWT**:

El usuario inicia sesión y recibe un token.

El token se almacena en SharedPreferences.

Retrofit añade el token en cada petición mediante un Interceptor.

7 PRUEBAS

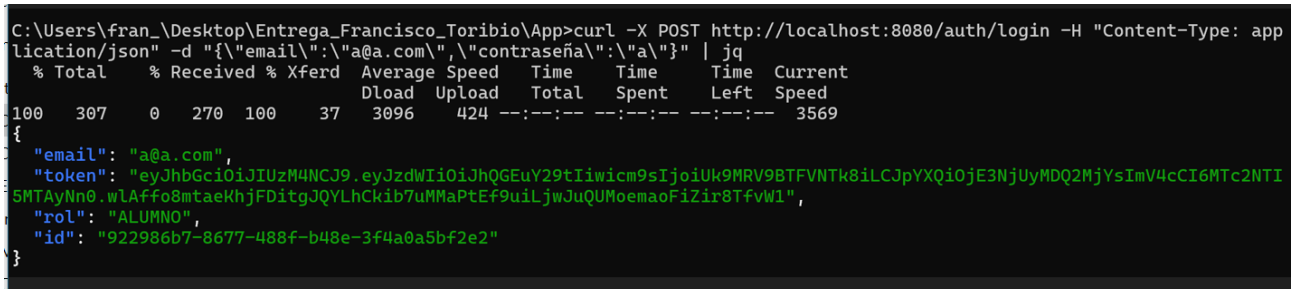
Son muchas las pruebas que pueden realizarse en un proyecto para eliminar los posibles errores y garantizar su correcto funcionamiento. Los casos de prueba establecen las condiciones/variables que permitirán determinar si los requisitos establecidos se cumplen o no.

A continuación, se detallan algunos de los casos de prueba que se ejecutarán para comprobar la correcta construcción de este proyecto.

Se realizaron pruebas unitarias e integración con **JUnit y Mock** para verificar la correcta funcionalidad de los endpoints.

Se realizaron pruebas de interfaz funcionales testando las pantallas y pruebas unitarias con **JUnit** para validar la correcta interacción entre vistas y ViewModels.

También se han probado los endpoints con curl.



src > test > java > com > example > demo > service > HabitacionServiceTest.java > ...

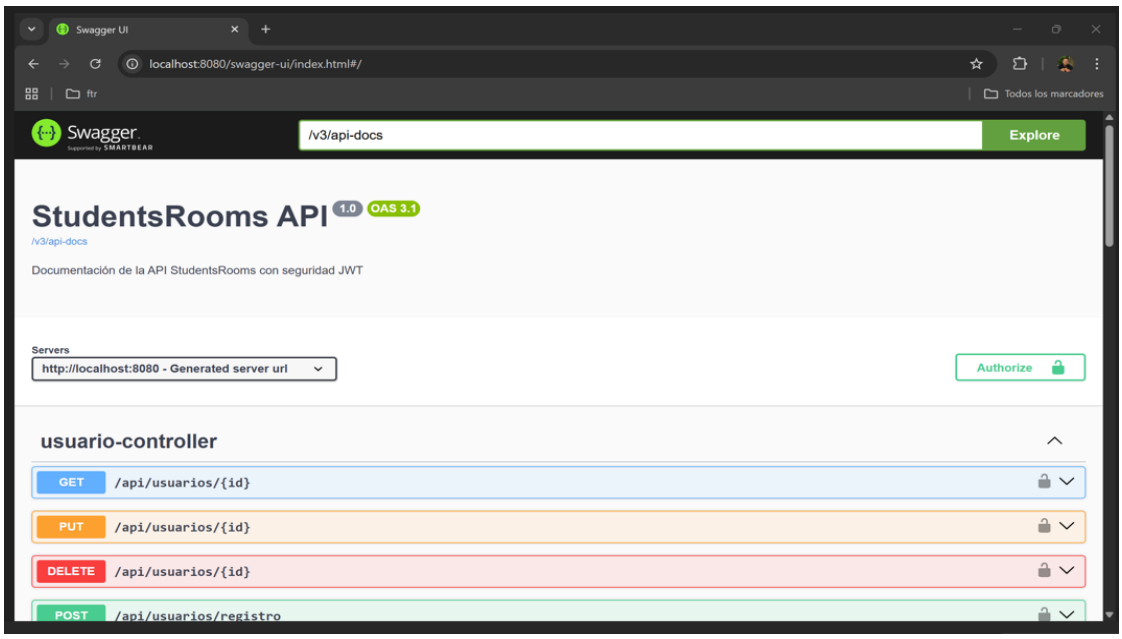
```java
package com.example.demo.service;

import com.example.demo.dto.CrearHabitacionDto;
import com.example.demo.model.Habitacion;
import com.example.demo.model.Usuario;
import com.example.demo.repository.HabitacionRepository;
import com.example.demo.repository.UsuarioRepository;
import org.junit.jupiter.api.Test;
import java.math.BigDecimal;
import java.util.Optional;
import java.util.UUID;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class HabitacionServiceTest {

    @Test
    void testCrearHabitacionAsignaPropietario() {
        // Arrange
        HabitacionRepository habitacionRepository = mock(HabitacionRepository.class);
        UsuarioRepository usuarioRepository = mock(UsuarioRepository.class);

        HabitacionService service = new HabitacionService(habitacionRepository, usuarioRepository)

        UUID propietarioId = UUID.randomUUID();
        Usuario propietario = new Usuario();
        propietario.setId(propietarioId);
        propietario.setEmail(email: "test@example.com");

        when(usuarioRepository.findById(propietarioId)).thenReturn(Optional.of(propietario));

```

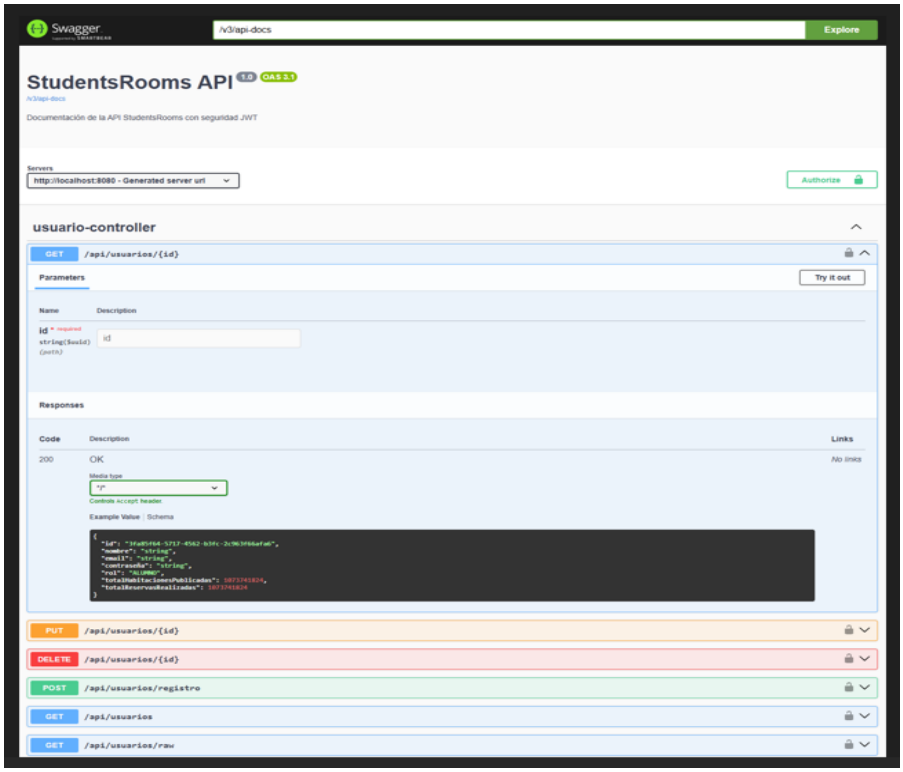
7.1 Casos de prueba

La aplicación tiene una suite de pruebas unitarias para probar las principales funciones de los servicios y los controladores.

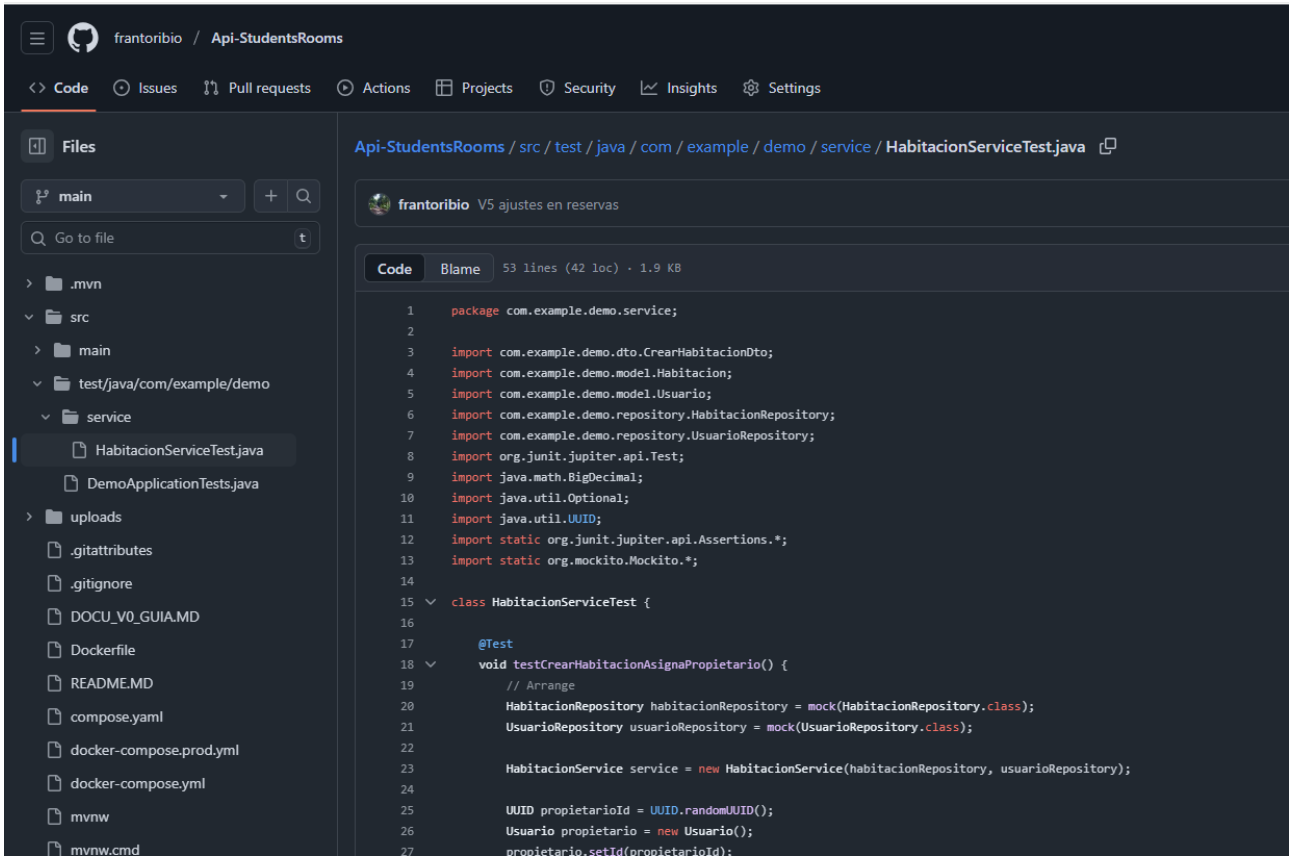
Se ha realizado un simulacro exhaustivo de todas las pruebas de funcionalidad con la interfaz de usuario. Los errores que han ido apareciendo se han depurado y corregido.

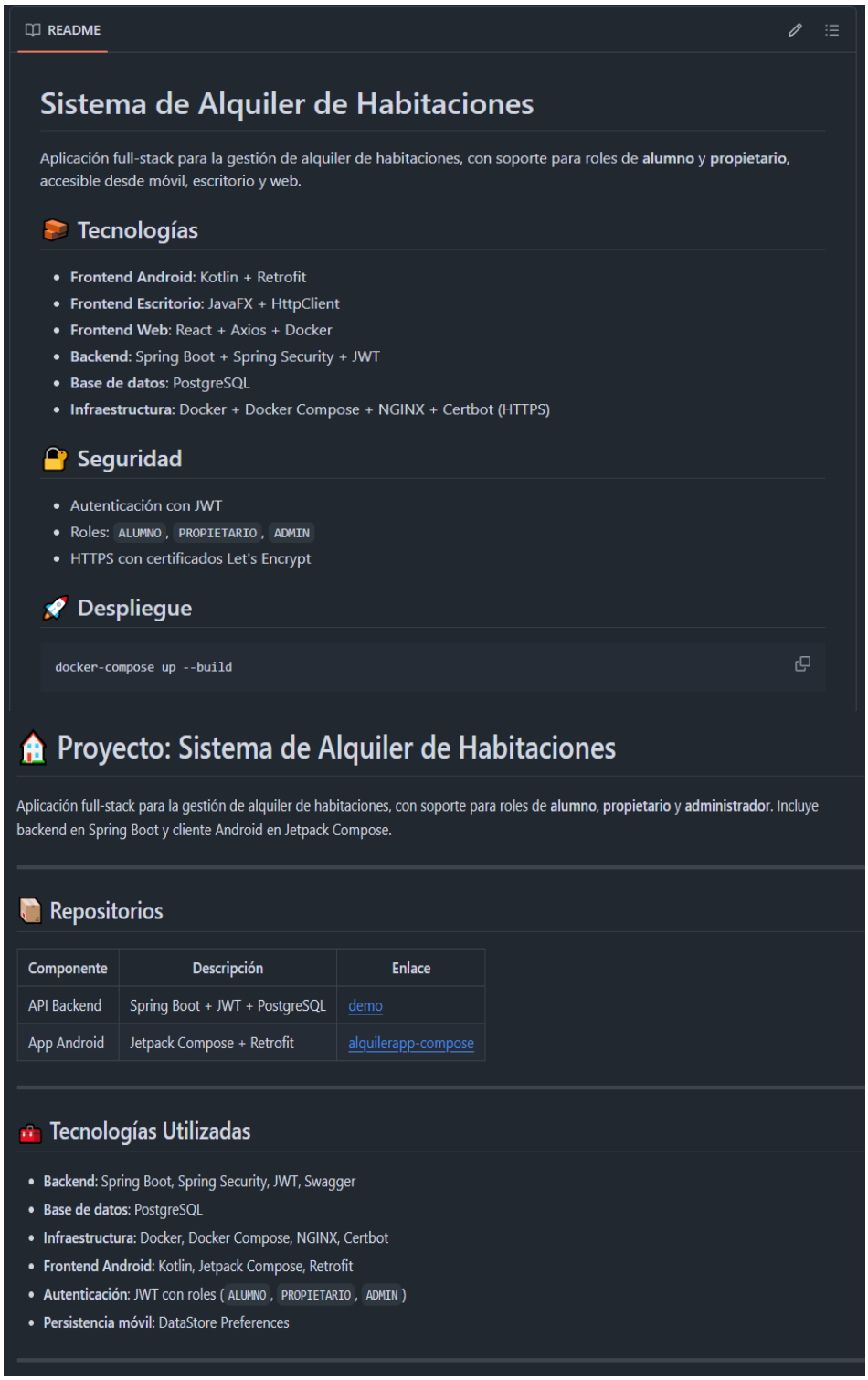
La aplicación cuenta con Swagger para documentar y probar todos los endpoints de la Api gracias a que todas las clases y funciones están comentadas.





En los enlaces a los repositorios se encuentran las instruciones y el código de los test.


frantoribio / Api-StudentsRooms
Code
Issues
Pull requests
Actions
Projects
Security
Insights
Settings
Files
main
Go to file
.mvn
src
main
test/java/com/example/demo
service
HabitacionServiceTest.java
DemoApplicationTests.java
uploads
.gitattributes
.gitignore
DOCU_V0_GUIA.MD
Dockerfile
README.MD
compose.yaml
docker-compose.prod.yml
docker-compose.yml
mvnw
mvnw.cmd
Api-StudentsRooms / src / test / java / com / example / demo / service / HabitacionServiceTest.java
frantoribio V5 ajustes en reservas
Code
Blame
53 lines (42 loc) · 1.9 KB
package com.example.demo.service;
import com.example.demo.dto.CrearHabitacionDto;
import com.example.demo.model.Habitacion;
import com.example.demo.model.Usuario;
import com.example.demo.repository.HabitacionRepository;
import com.example.demo.repository.UsuarioRepository;
import org.junit.jupiter.api.Test;
import java.math.BigDecimal;
import java.util.Optional;
import java.util.UUID;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;
class HabitacionServiceTest {
@Test
void testCrearHabitacionAsignaPropietario() {
// Arrange
HabitacionRepository habitacionRepository = mock(HabitacionRepository.class);
UsuarioRepository usuarioRepository = mock(UsuarioRepository.class);
HabitacionService service = new HabitacionService(habitacionRepository, usuarioRepository);
UUID propietarioId = UUID.randomUUID();
Usuario propietario = new Usuario();



README

Sistema de Alquiler de Habitaciones

Aplicación full-stack para la gestión de alquiler de habitaciones, con soporte para roles de **alumno** y **propietario**, accesible desde móvil, escritorio y web.

Tecnologías

- **Frontend Android**: Kotlin + Retrofit
- **Frontend Escritorio**: JavaFX + HttpClient
- **Frontend Web**: React + Axios + Docker
- **Backend**: Spring Boot + Spring Security + JWT
- **Base de datos**: PostgreSQL
- **Infraestructura**: Docker + Docker Compose + NGINX + Certbot (HTTPS)

Seguridad

- Autenticación con JWT
- Roles: `ALUMNO`, `PROPIETARIO`, `ADMIN`
- HTTPS con certificados Let's Encrypt

Despliegue

```
docker-compose up --build
```

Proyecto: Sistema de Alquiler de Habitaciones

Aplicación full-stack para la gestión de alquiler de habitaciones, con soporte para roles de **alumno**, **propietario** y **administrador**. Incluye backend en Spring Boot y cliente Android en Jetpack Compose.

Repositorios

Componente	Descripción	Enlace
API Backend	Spring Boot + JWT + PostgreSQL	demo
App Android	Jetpack Compose + Retrofit	alquilerapp-compose

Tecnologías Utilizadas

- **Backend**: Spring Boot, Spring Security, JWT, Swagger
- **Base de datos**: PostgreSQL
- **Infraestructura**: Docker, Docker Compose, NGINX, Certbot
- **Frontend Android**: Kotlin, Jetpack Compose, Retrofit
- **Autenticación**: JWT con roles (`ALUMNO`, `PROPIETARIO`, `ADMIN`)
- **Persistencia móvil**: DataStore Preferences

🚀 Guía de Instalación

1. Backend Spring Boot

Requisitos

- Java 17+
- Maven
- Docker + Docker Compose
- PostgreSQL

Pasos

```
# Clona el repositorio
git clone https://github.com/frantoribio/demo
cd demo
```

🛠️ Construcción y arranque del proyecto

Para iniciar el backend, abre el proyecto `demo` en **Visual Studio Code** y asegúrate de tener la extensión de **Docker** instalada y activa.

1. Abre el archivo `compose.yaml` ubicado en la raíz del proyecto.
2. Desde la pestaña **Docker** en VS Code, selecciona el servicio correspondiente y haz clic en **"Compose Up"**.
3. Espera a que los contenedores se levanten correctamente.

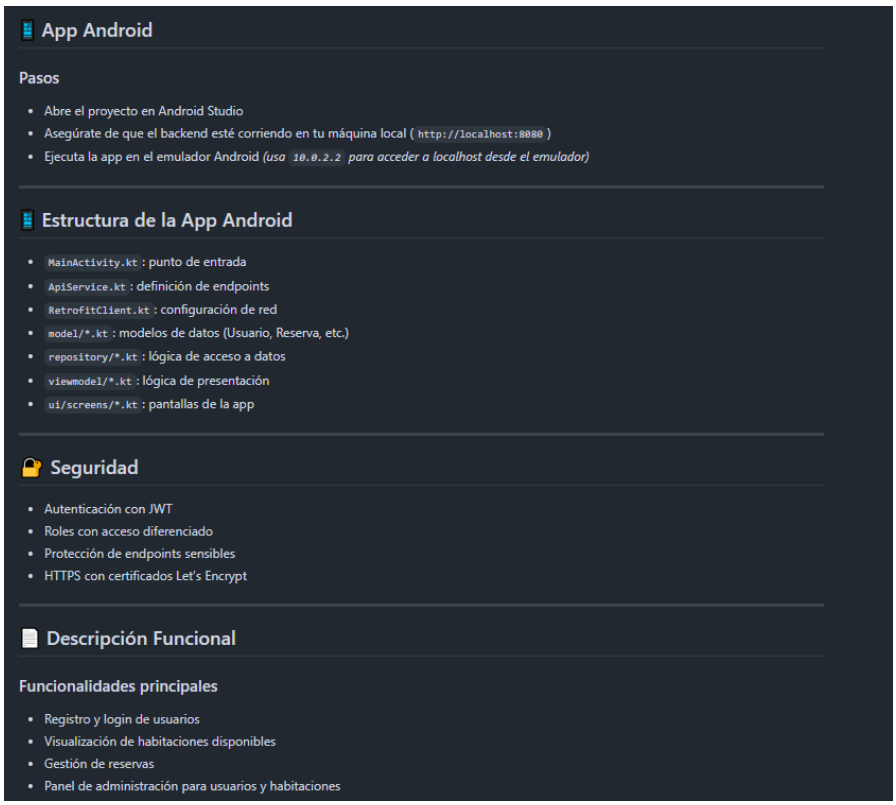
Esto iniciará:

- La API en `http://localhost:8080`
- PostgreSQL con datos persistentes
- NGINX con HTTPS si configuras Certbot

> 💡 Nota: No es necesario usar el comando `docker-compose up --build` manualmente, ya que Visual Studio Code gestiona el proceso con el archivo `compose.yaml`. También se pueden omitir estos pasos ejecutando el proyecto desde el archivo `DemoApplication.java`.

🚀 Esto iniciará

- API en `http://localhost:8080`
- PostgreSQL con datos persistentes
- NGINX con HTTPS si configuras Certbot



📱 App Android

Pasos

- Abre el proyecto en Android Studio
- Asegúrate de que el backend esté corriendo en tu máquina local (`http://localhost:8080`)
- Ejecuta la app en el emulador Android *(usa `10.0.2.2` para acceder a localhost desde el emulador)*

📱 Estructura de la App Android

- `MainActivity.kt` : punto de entrada
- `ApiService.kt` : definición de endpoints
- `RetrofitClient.kt` : configuración de red
- `model/*.kt` : modelos de datos (Usuario, Reserva, etc.)
- `repository/*.kt` : lógica de acceso a datos
- `viewmodel/*.kt` : lógica de presentación
- `ui/screens/*.kt` : pantallas de la app

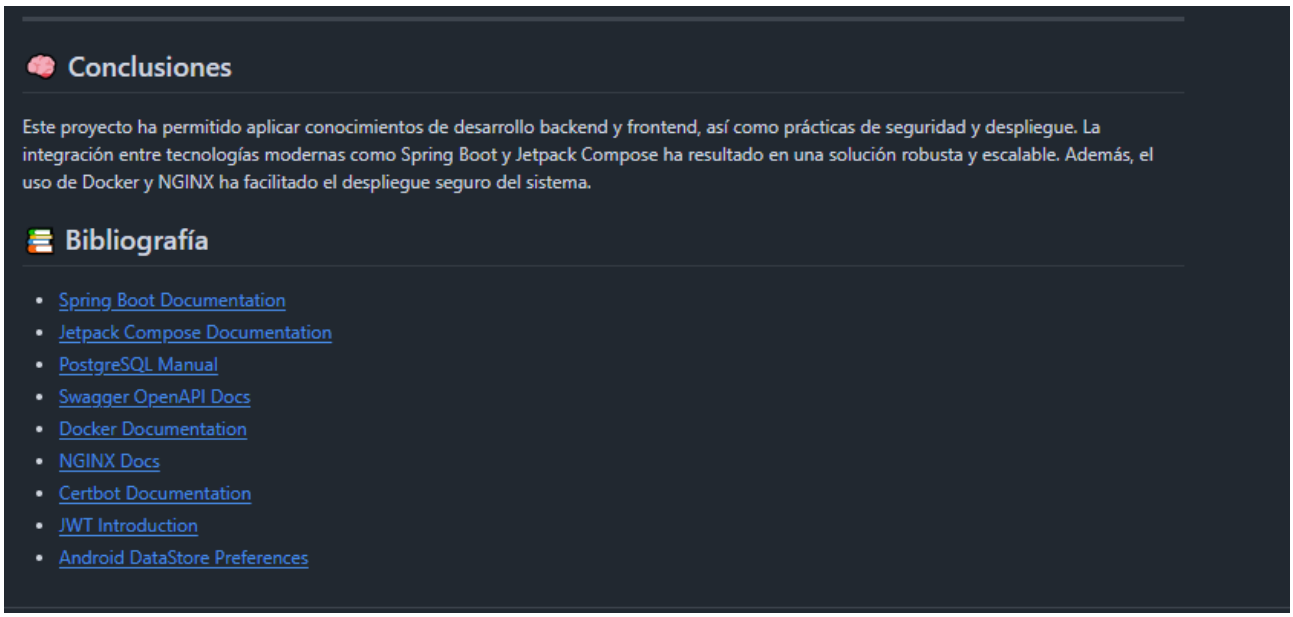
🔐 Seguridad

- Autenticación con JWT
- Roles con acceso diferenciado
- Protección de endpoints sensibles
- HTTPS con certificados Let's Encrypt

📄 Descripción Funcional

Funcionalidades principales

- Registro y login de usuarios
- Visualización de habitaciones disponibles
- Gestión de reservas
- Panel de administración para usuarios y habitaciones



🧠 Conclusiones

Este proyecto ha permitido aplicar conocimientos de desarrollo backend y frontend, así como prácticas de seguridad y despliegue. La integración entre tecnologías modernas como Spring Boot y Jetpack Compose ha resultado en una solución robusta y escalable. Además, el uso de Docker y NGINX ha facilitado el despliegue seguro del sistema.

📚 Bibliografía

- Spring Boot Documentation
- Jetpack Compose Documentation
- PostgreSQL Manual
- Swagger OpenAPI Docs
- Docker Documentation
- NGINX Docs
- Certbot Documentation
- JWT Introduction
- Android DataStore Preferences

8 EXPLOTACIÓN

La implantación es la fase más crítica del proyecto ya que el sistema entra en producción, es decir opera en un entorno real, con usuarios reales.

El sistema queda preparado para su puesta en producción sin ser el propósito principal del proyecto. Se explota con despliegue en servidor local.

8.1 Planificación

Para la puesta en producción sólo es necesario comprar un dominio y contratar el hosting donde se aloje la API. Basta con modificar los properties de Dev a Prod y cambiar las **URLs base** o **root paths** de los endpoints para desplegar en producción, esa raíz cambia según el **dominio o dirección del servidor** donde esté alojada la aplicación

8.2 Preparación para el cambio

No se requiere de preparación previa, basta con la decisión de comenzar la explotación del sistema.

8.3 Manual de usuario

El proyecto se entrega en una carpeta comprimida que contiene:

- Las instrucciones de instalación.

- La memoria.

- Una carpeta con:

·El archivo .apk instalable en terminales Android.

·El application.properties, necesario para la configuración inicial en Spring de la base de datos

·Un archivo docker-compose.yaml para crear la imagen de postgres en Docker, configurar el entorno del contenedor y el puerto de la base de datos.

·El archivo .jar que empaqueta la API Spring del backend.

·Un script .bat que automatiza el proceso

Instrucciones para cargar la aplicación en un entorno local

Abrir la carpeta y ejecutar start_app si tienes Docker Desktop abierto, si no, hay que seguir estos pasos:

1º Tener Docker Desktop instalado y corriendo. Ya se podría ejecutar start.app.

Si no quieres ejecutar el start_app.bat:

2º Abrir un terminal en el directorio donde está el docker-compose y escribir docker-compose up --build

3º Abrir otro terminal en el directorio donde está demo-0.0.1-SNAPSHOT.jar y escribir java -jar .\demo-0.0.1-SNAPSHOT.jar

Cuando esté corriendo Docker con la base de datos y la API:

4º Abrir un emulador en Android Studio y arrastrar el app-debug.apk

5º Descargar imágenes desde internet con tamaño inferior a 1Mb en el emulador para poder cargarlas en la aplicación Android https://www.pexels.com/es-es/buscar/habitaci%C3%B3n/ .

6º Probar la aplicación creando usuarios.

7º Logarse como administrador: Usuario > admin@alquilerapp.com Contraseña > admin123

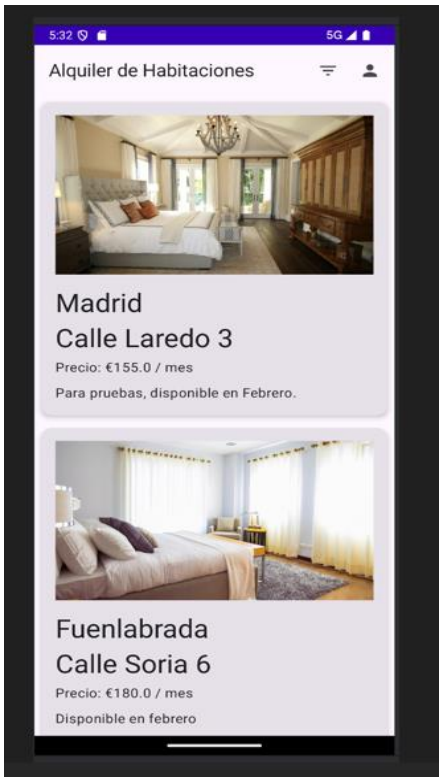
También se puede clonar el código desde los repositorios:

API SPRING: https://github.com/frantoribio/demo

APP ANDROID: https://github.com/frantoribio/alquilerapp-compose

Instrucciones de uso

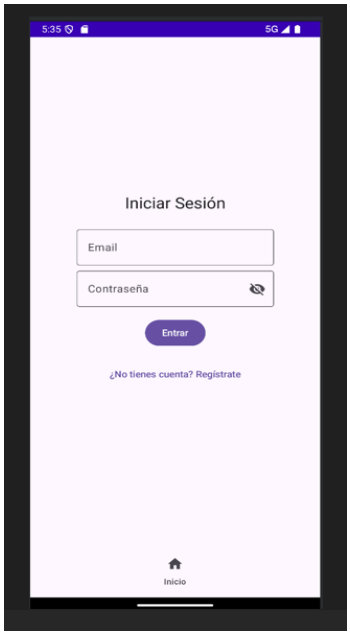
Corriendo el servidor e instalando el .apk en un dispositivo Android o en un emulador en Android Studio la aplicación arranca con una pantalla de **landing** donde se muestran todas las habitaciones disponibles presentadas en **Cards** con un **scrolling** vertical infinito.

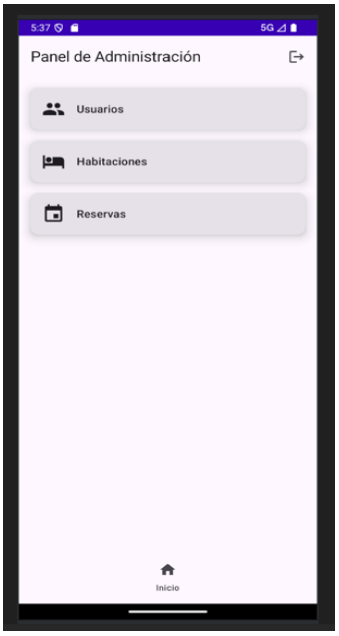


Aprovecho esta sección para la representación del diseño y resultado final de las Interfaces de usuarios y no cargar con más datos repetitivos la memoria.

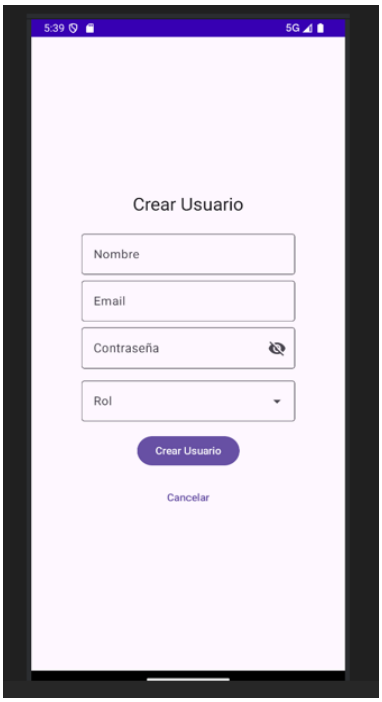
Como se aprecia en la imagen, tiene un menú superior con un Icono **People** por el que se navega a la pantalla de login o registro.

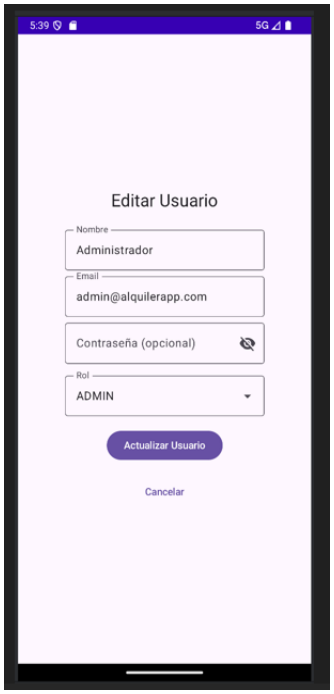
Después del registro eligiendo nuestro rol, el usuario se loga y la aplicación le lleva a la pantalla que le corresponde.

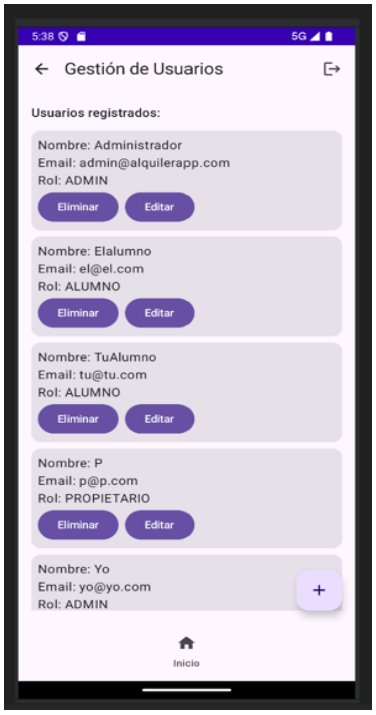

5:35
5G
Iniciar Sesión
Email
Contraseña
Entrar
¿No tienes cuenta? Regístrate
Inicio

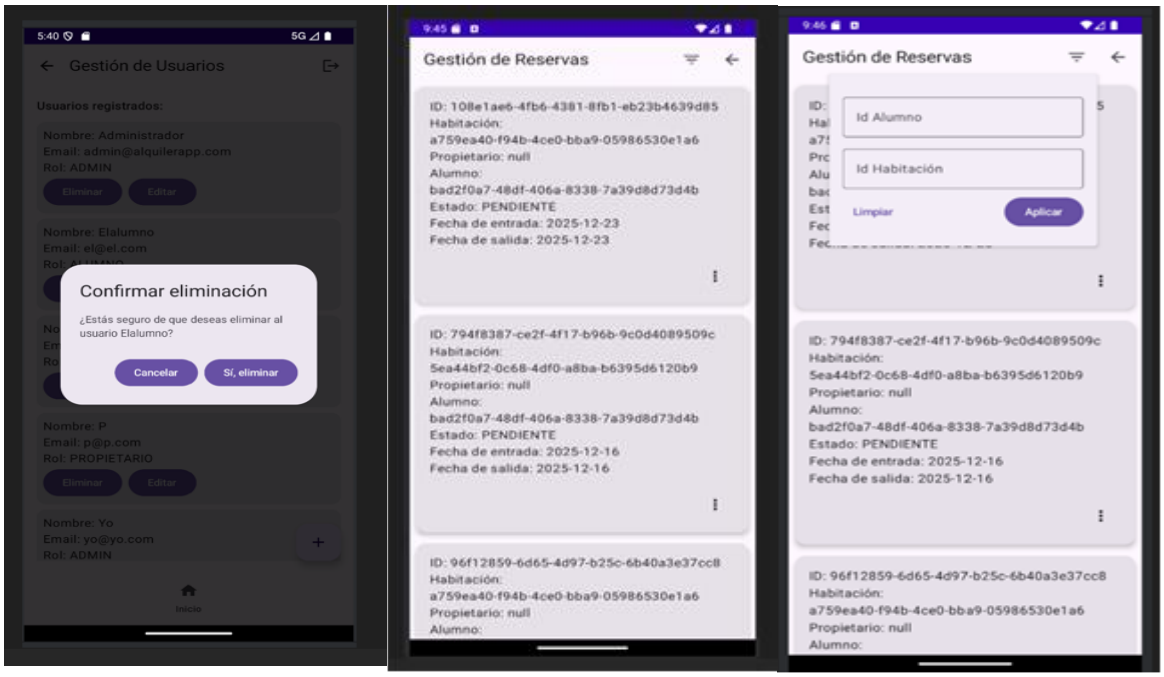

5:37
5G
Panel de Administración
Usuarios
Habitaciones
Reservas
Inicio


5:36
5G
Registro de Usuario
Nombre
Email
Contraseña
Rol
Registrar
¿Ya tienes cuenta? Inicia sesión
Inicio


5:39
5G
Crear Usuario
Nombre
Email
Contraseña
Rol
Crear Usuario
Cancelar


5:39
5G
Editar Usuario
Nombre
Administrador
Email
admin@alquilerapp.com
Contraseña (opcional)
Rol
ADMIN
Actualizar Usuario
Cancelar

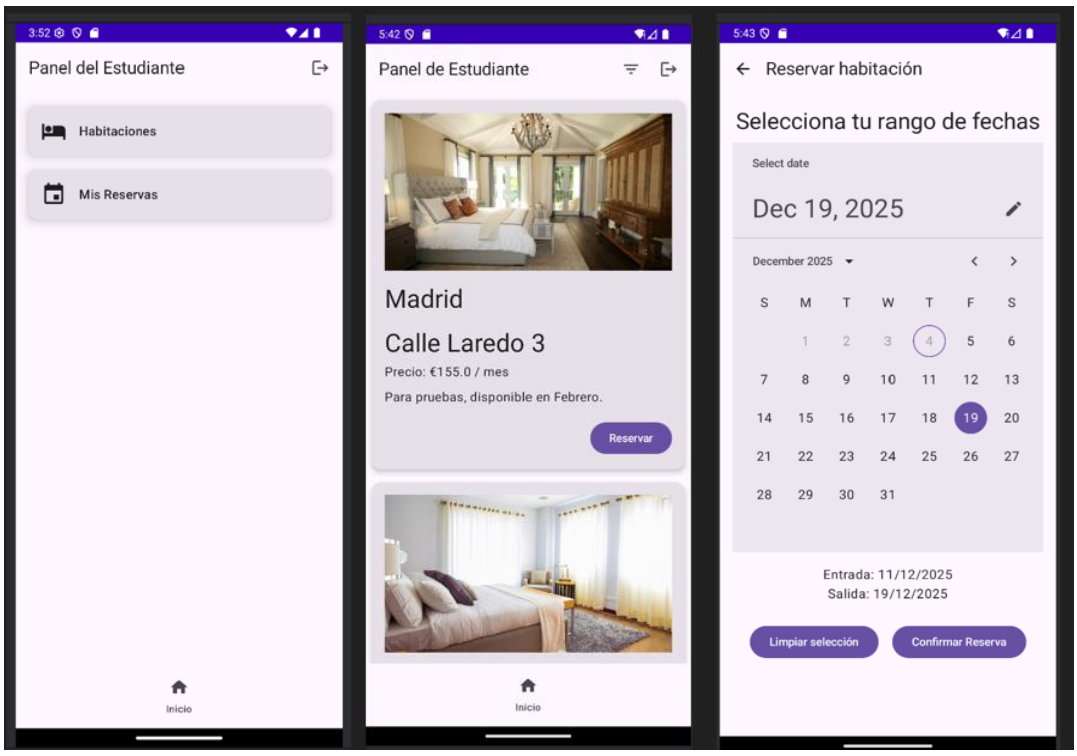

5:38
5G
Gestión de Usuarios
Usuarios registrados:
Nombre: Administrador
Email: admin@alquilerapp.com
Rol: ADMIN
Eliminar
Editar
Nombre: Elalumno
Email: el@el.com
Rol: ALUMNO
Eliminar
Editar
Nombre: TuAlumno
Email: tu@tu.com
Rol: ALUMNO
Eliminar
Editar
Nombre: P
Email: p@p.com
Rol: PROPIETARIO
Eliminar
Editar
Nombre: Yo
Email: yo@yo.com
Rol: ADMIN
Inicio

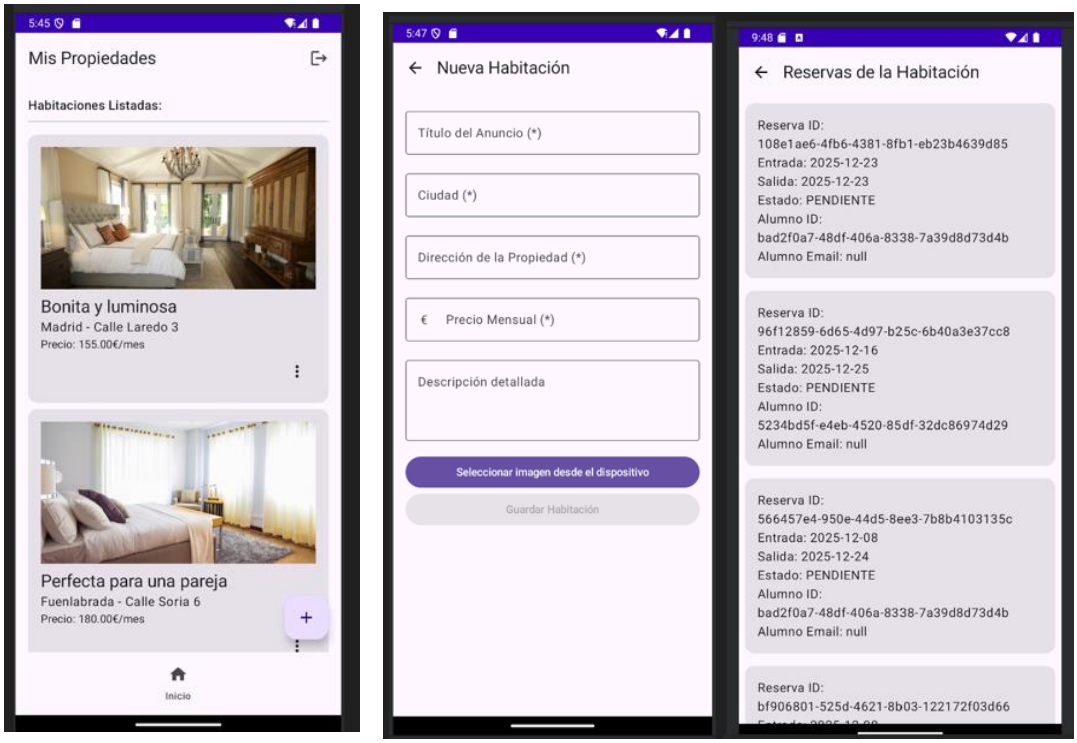


En la barra superior hay un icono para el filtrado de entidades y una flecha izquierda para volver hacia atrás en la aplicación . En la pantalla principal del administrador, en la parte superior derecha está el icono para deslogarse de la aplicación . Este icono también se encuentra en el panel principal de alumno y propietario. En el panel de gestión de usuarios del administrador, al pulsar el floating boton se navega a la pantalla de creación de usuarios. Con estos botones el administrador edita o elimina el usuario de la **card** .

En la pantalla propietario también encontramos el floating boton para la creación de nuevas habitaciones.

En la pantalla del alumno, en cada **card** de habitación, hay un botón que te lleva a un calendario donde elegir las fechas de la reserva.





8.4 Implantación propiamente dicha

La aplicación está lista para probarla en local y realizar todas las pruebas funcionales de sus casos de uso.

8.5 Pruebas de implantación

En el supuesto de despliegue a producción se haría un seguimiento y monitorización de la aplicación durante la primera semana para evitar posibles imprevistos.

9 DEFINICIÓN DE PROCEDIMIENTOS DE CONTROL Y EVALUACIÓN

A lo largo del ciclo de vida del proyecto se producirán cambios e incidencias que deberán controlarse y registrarse.

Se requiere el uso de herramientas específicas que permitan hacer un seguimiento para evaluar las **incidencias** que puedan presentarse durante la realización de las actividades, su posible solución y registro.

Para esta tarea se utiliza **Git** y su sistema de almacenamiento en la nube **GitHub**, herramientas específicas de control de versiones, que permiten hacer un seguimiento de los posibles **cambios** en los recursos y en las actividades, incluyendo el sistema de registro de estos.

La aplicación cuenta con un sistema configurable de **logs** que permite su trazabilidad para la corrección de errores y monitorizado.

10 CONCLUSIONES

Integración exitosa entre cliente y servidor La implementación de la API en Spring y su consumo desde la aplicación Android ha permitido establecer una comunicación fluida mediante servicios REST, garantizando la correcta gestión de datos de usuarios, habitaciones y reservas.

Arquitectura modular y escalable El uso de patrones de diseño como **MVC** en la API y **MVVM** en la aplicación Android ha favorecido la separación de responsabilidades, facilitando la mantenibilidad, escalabilidad y futuras ampliaciones del sistema.

Persistencia y gestión de datos eficiente Gracias a **Spring Data JPA**, se simplificó el acceso a la base de datos, reduciendo la necesidad de código repetitivo y asegurando consultas optimizadas. En el cliente, el uso de **Retrofit** permitió transformar respuestas JSON en objetos Java de manera sencilla y robusta.

Seguridad y control de acceso La incorporación de **Spring Security con JWT** garantizó la autenticación y autorización de usuarios, estableciendo roles diferenciados (estudiante, propietario, administrador) y reforzando la protección de la información.

Interfaz intuitiva y experiencia de usuario La aplicación Android ofrece una interfaz clara y dinámica, apoyada en componentes como **Cards**, que facilita la visualización de habitaciones y reservas. Esto contribuye a una experiencia de usuario satisfactoria y adaptada al público objetivo.

Pruebas y validación del sistema Se realizaron pruebas unitarias e integración tanto en la API como en la aplicación Android, asegurando la fiabilidad de los **endpoints** y la correcta interacción entre vistas y lógica de negocio.

Aprendizaje y aplicación de buenas prácticas El desarrollo del proyecto permitió aplicar buenas prácticas de programación, como la inyección de dependencias, el uso de **Flows** en Android y la separación de capas, consolidando conocimientos en desarrollo de software moderno.

Posibilidades de mejora y ampliación El sistema se encuentra preparado para futuras mejoras, como la integración de pasarelas de pago, notificaciones push en la aplicación móvil o la incorporación de analítica para optimizar la gestión de reservas.

11 FUENTES

11.1 Legislación

DAM

Enseñanzas mínimas**: Real Decreto 450/2010, de 16 de abril (BOE 20/05/2010)**

https://www.boe.es/boe/dias/2010/05/20/pdfs/BOE-A-2010-8067.pdf

Currículo: **D. 3/2011, de 13 de enero (BOCM 31/01/2011)**

https://www.comunidad.madrid/sites/default/files/doc/educacion/fp/FP-Ensenanza-IFCS02-LOE-Curriculo-D20110003.pdf

11.2 Enlaces

https://github.com/frantoribio/Api-StudentsRooms

https://github.com/frantoribio/Alquilerapp-compose-StudentsRooms

12 ANEXOS

12.1 Bibliografía

https://spring.io/projects/spring-boot

https://developer.android.com/compose

https://www.postgresql.org/docs/

https://docs.docker.com/

https://www.jwt.io/introduction

https://swagger.io/docs/