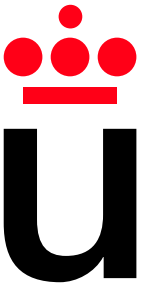


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

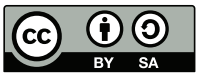
Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

MOAI: UNA APLICACIÓN WEB PARA LA ADMINISTRACIÓN DE COMUNIDADES DE PROPIETARIOS

Autor: Miguel Delgado Amador
Tutor: Michel Maes Bermejo



Agradecimientos

En primer lugar, me gustaría agradecer a mi familia, especialmente a mis padres, y a Marina, por su apoyo y esfuerzo a lo largo de esta aventura, contribuyendo de una forma u otra al desarrollo de este trabajo.

También quisiera agradecer a mi tutor Michel Maes por su colaboración y compromiso a lo largo de este proyecto, aportándome toda la ayuda necesaria desde el primer día.

Por último, me gustaría dedicar este trabajo a mi abuela, que habría estado muy orgullosa de verme llegar hasta el final.

Resumen

Este proyecto consiste en el desarrollo de una aplicación web práctica y robusta diseñada para mejorar y simplificar la gestión de comunidades de propietarios, tanto para los vecinos como para los administradores. La herramienta ofrece funcionalidades que simplifican para el administrador la gestión de elementos tales como inmuebles, vecinos o recibos. Además, facilita a los propietarios la visualización y la actualización de toda la información de su interés.

La aplicación se ha desarrollado siguiendo una arquitectura de tres capas bien definidas: Presentación, Aplicación y Persistencia. Esta estructura modular permite una separación clara de responsabilidades, lo que facilita el mantenimiento del sistema y mejora su rendimiento. La capa de Presentación se encarga de la interacción con el usuario a través de una interfaz web construida con tecnologías como Vue.js, HTML y CSS. Por su parte, la capa de Aplicación implementa la lógica de negocio utilizando Java y el framework Spring Boot, gestionando la coordinación entre los distintos componentes del sistema. Finalmente, la capa de Persistencia se ocupa del acceso a los datos almacenados en una base de datos MySQL, apoyándose en Hibernate y Spring Data JPA para garantizar una comunicación eficiente y segura con el sistema de almacenamiento.

El proyecto incorpora una estrategia de pruebas completa, que abarca tanto pruebas unitarias como pruebas de sistema. Para ello, se han utilizado herramientas como JUnit 5 y REST Assured en la validación de la lógica de negocio, asegurando que los componentes individuales funcionen correctamente. Además, se han empleado pruebas de sistema con Selenium para verificar que las funcionalidades de la aplicación se comportan adecuadamente en escenarios reales.

Para gestionar el desarrollo del proyecto se ha seguido un flujo de trabajo basado en una variante simplificada de la metodología Git Flow. Además, se han integrado prácticas de DevOps mediante la automatización del proceso de integración y entrega continua con GitHub Actions. Este sistema automatizado se encarga de ejecutar pruebas, generar versiones del proyecto con etiquetas temporales y realizar el despliegue en Microsoft Azure, utilizando para ello contenedores Docker desplegados en Instancias de Contenedor de Azure.

Por último, se reflexiona acerca de los resultados en las conclusiones, y se exponen algunas ideas de mejora de cara a posibles trabajos futuros.

Palabras clave:

- Java
- JavaScript
- Vue.js
- Spring Boot
- Maven
- MySQL
- Git
- Docker
- Microsoft Azure

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción

En el presente capítulo se expondrá el contexto del proyecto, su justificación y los objetivos que se persiguen con su realización, junto con la estructura general de este documento.

1.1. Contexto

Vivimos en un mundo en el que tanto los ordenadores como internet han cambiado por completo nuestras vidas, formando parte de nuestro día a día, a veces incluso sin darnos cuenta de ello. Muchas personas, sobre todo las nuevas generaciones, prefieren trabajar de forma digital en lugar de a la vieja usanza, con papel y bolígrafo, enviando las comunicaciones a través del correo tradicional.

Mejorar la eficiencia del trabajo y aprovechar cada minuto de nuestro tiempo es un detalle muy valorado en la sociedad actual, en la que todo se mueve muy rápido, con cambios constantes, queriendo estar siempre a la vanguardia. Las masas de gente con las que se trabaja cada vez son mayores, implicando un mayor volumen de datos que manejar y más problemas que resolver, todo ello en el menor tiempo posible. Cuanto más inmediato, mejor.

Si nos centramos particularmente en el sector de la vivienda, vemos que las obras de nueva construcción son cada vez más avanzadas, con comunidades de vecinos que integran más tecnología y más equipamiento. Por otro lado, los propietarios de estas viviendas suelen tener una vida más ajetreada que hace unos años, pasando bastantes horas fuera de casa, y cuando vuelven lo que buscan es tranquilidad y comodidad, sin tener que enfrentarse a problemas añadidos.

Es por ello que en este contexto surge la necesidad de proporcionar una herramienta digital para que tanto vecinos como administradores puedan resolver rápida y cómodamente los asuntos inherentes a las comunidades de propietarios. Una aplicación a la que poder acceder fácilmente a través de un navegador de internet, sin necesidad de instalar nada en el ordenador, y donde, por un lado, los vecinos puedan consultar de manera sencilla la información relativa a sus inmuebles, así como resolver posibles problemas con los mismos, y, por otro lado, los administradores puedan realizar rápidamente y de forma organizada su trabajo.

1.1.1. Contexto de las comunidades de propietarios

Las comunidades de propietarios constan de múltiples elementos que deben ser tenidos en cuenta para poder gestionarlas correctamente. Los principales son los siguientes:

- **Inmuebles**: un pilar fundamental de los que conforman toda comunidad de propietarios. Los inmuebles pueden ser de distintos tipos, siendo los más comunes viviendas, plazas de garaje y locales comerciales. Pueden convivir distintos tipos en una misma comunidad, como puede ser un bloque de pisos con garaje comunitario y locales comerciales en los bajos, pero también pueden darse comunidades que solo tengan viviendas, o que únicamente posean plazas de garaje porque el garaje comunitario sea gestionado de forma independiente de las viviendas como una comunidad aparte.
- **Propietarios**: otro de los principales pilares de las comunidades de vecinos. Todo inmueble que forme parte de la comunidad tendrá, al menos, un propietario, que será el que tenga que hacerse cargo de los gastos asociados a su inmueble, además de tener que asumir toda la responsabilidad de los posibles problemas que surjan con el mismo, estando por encima de otros posibles usuarios del inmueble, que podrían ser los familiares que vivan con él en una vivienda. Un propietario puede poseer distintos inmuebles en una misma comunidad, como una vivienda y una plaza de garaje, por ejemplo.
- **Administrador**: un elemento muy importante en una comunidad de propietarios. Todas ellas tienen un equipo de gobierno que es el que toma las principales decisiones y se encarga del correcto funcionamiento de la misma, el cual normalmente está compuesto por un presidente, un secretario y un administrador. Sin embargo, es éste último el encargado de llevar a cabo las gestiones necesarias, como puede ser cobrar los recibos o llevar la información de los propietarios de la comunidad.
- **Recibos**: toda comunidad de propietarios, como consecuencia lógica de su funcionamiento natural, generará una serie de recibos para sus inmuebles. El propietario de cada uno de ellos deberá pagar una fracción de los gastos

originados por la comunidad, ya sean gastos ordinarios, como el servicio de limpieza o la electricidad gastada en el portal, o extraordinarios (también conocidos como derramas), como una necesidad puntual de acometer reformas en el edificio. Estos recibos pueden ser cobrados mediante una división equitativa entre todos los inmuebles que conforman la comunidad o según el coeficiente de participación que haya sido asignado para cada inmueble.

- **Zonas comunes y reservas**: todas las comunidades de propietarios poseen una serie de zonas comunes, que pueden ir desde el propio portal de acceso a la comunidad hasta una piscina o jardines. Algunas comunidades más modernas y de mayor renta pueden disponer también de una oferta de ocio superior para sus usuarios, como pueden ser un gimnasio, una pista de tenis o una sala de reuniones. Para estos casos puede ser necesario reservar su uso con antelación, de forma que se pueda llevar a cabo una correcta gestión y organización de estas zonas comunes, garantizando un disfrute equitativo para todos los usuarios.
- **Normativa**: cada comunidad de propietarios elabora su propia normativa, estableciendo un código de conducta para la misma. En ella se establecen aspectos como los horarios de uso de las zonas comunes, cuándo está permitido hacer más ruido para no molestar al resto de vecinos o cómo se puede emplear cada plaza de garaje. Todos los propietarios de la comunidad deben tener acceso a la normativa para poder garantizar su correcto cumplimiento y tener un conocimiento completo de lo que está permitido y lo que no, además de poder acceder a otros documentos de relevancia para ellos, como las actas de reunión de la comunidad.

La comunicación en las comunidades de propietarios típicamente se ha realizado a través de los buzones, repartiendo para cada inmueble los documentos necesarios, como el acta de una reunión, o mediante papeles establecidos en el portal a la vista de todos los propietarios. También, en tiempos más recientes, se ha comenzado a utilizar el correo electrónico para enviar ciertas comunicaciones, pero no de forma centralizada, y cierta información sigue sin estar disponible de una manera simple y cómoda para los propietarios, teniendo que solicitar personalmente al administrador esa información.

1.1.2. Justificación y objetivos del proyecto

Existen múltiples aspectos de importancia en base a lo anteriormente detallado que sugieren la necesidad de desarrollar una herramienta como la que se presenta en este proyecto.

- **Mejora del acceso a la información.**
  Con la creación de una herramienta online que permita su uso tanto a ad-

ministradores como a propietarios se facilita el acceso y se mejora significativamente la rapidez con la que estos últimos pueden ver las más recientes novedades publicadas por el administrador. De esta forma, simplemente abriendo su navegador e iniciando sesión los propietarios pueden ver sus últimos recibos, cambios en la normativa, o los nuevos documentos de interés que hayan sido compartidos.

- **Mayor comodidad para realizar tareas cotidianas.**
  Siendo una aplicación web, los propietarios pueden realizar todas las gestiones necesarias desde cualquier lugar, ya sea desde fuera de casa o simplemente desde ella, pero evitando la necesidad de salir para llevar a cabo las tareas correspondientes. Ya no sería necesario acudir a la oficina del administrador para realizar ciertas gestiones, pudiendo hacerlo de forma remota.

- **Gestión del trabajo más refinada.**
  Utilizando una aplicación centralizada, los administradores pueden tener un acceso rápido y sencillo a toda la información necesaria, como los datos personales de los propietarios o la información de los inmuebles. Así, se eliminan de la ecuación posibles problemas que pueden surgir con el papeleo, como pueden ser incoherencias entre documentos procedentes de diferentes ubicaciones o documentación traspapelada. Con esta herramienta se garantiza que toda la información estará siempre presente y disponible en un mismo lugar, sin diferentes versiones de los datos.

- **Eficiencia superior en la realización de tareas.**
  La velocidad del trabajo aumenta si se puede realizar una misma tarea común para todos los inmuebles al mismo tiempo, como puede ser la creación de nuevos recibos, estableciendo unos criterios comunes y dejando que la aplicación haga de forma automática el resto, en lugar de ir uno por uno. Además, se evita la necesidad de buscar la información de forma tediosa en multitud de papeles y hojas de cálculo, utilizando una aplicación específicamente diseñada para el trabajo que se tiene que realizar.

- **Mayor transparencia y agilidad.**
  Si tanto propietarios como administradores trabajan con una única aplicación, los primeros pueden ver exactamente la misma información que emplea el administrador, conociendo en todo momento cuáles son sus datos personales que están siendo utilizados, por ejemplo. Además, los propietarios pueden actualizar rápidamente estos datos si, como podría ocurrir, cambian de cuenta bancaria o teléfono, teniendo el administrador un acceso inmediato a la nueva información proporcionada para que pueda ser utilizada lo antes posible.

- **Reducción del consumo y residuos.**
  Empleando herramientas digitales se reduce la utilización de papel y otros

consumibles, como tinta de impresora o plásticos para recoger los documentos. De esta forma, se contribuye a un mejor cuidado del medio ambiente por parte de toda la comunidad de propietarios.

1.2. Estructura del documento

La siguiente estructura es la que será seguida a lo largo de este documento:

1. Introducción.
2. Objetivos.
3. Tecnologías, Herramientas y Metodologías.
4. Descripción Informática.
5. Conclusiones y trabajos futuros.
6. Bibliografía.
7. Apéndices.

2

Objetivos

El principal objetivo que se busca con este proyecto es desarrollar una aplicación web funcional y fácil de utilizar para la gestión de comunidades de propietarios. Los objetivos se pueden dividir en dos categorías: objetivos funcionales y objetivos técnicos.

2.1. Objetivos funcionales

1. **Facilitar la gestión de comunidades de propietarios.**
   Aportar características y herramientas directamente dirigidas a esta labor, permitiendo automatizar tareas y crear, actualizar y eliminar todos los elementos constituyentes de una comunidad de propietarios.

2. **Mejorar y simplificar la comunicación entre administradores y propietarios.**
   Implementar funcionalidades que permitan a ambas partes mantenerse informadas con la mayor rapidez posible, evitando la necesidad de realizar trámites a través de otras vías, enfocándose todo lo posible en el uso de internet.

3. **Promover la transparencia y la rapidez.**
   Proporcionar una visibilidad clara y centralizada de toda la información que puedan necesitar tanto un propietario como un administrador, además de permitir una actualización rápida y sencilla de todos estos datos.

2.2. Objetivos técnicos

1. **Garantizar la seguridad de la aplicación.**
   Introducir medidas de seguridad que permitan la protección de todos los datos de las comunidades de propietarios y los separen por roles, cumpliendo con las prácticas adecuadas de seguridad informática.

2. **Desarrollar una aplicación web intuitiva y sencilla.**
   Elaborar una herramienta que resulte fácil de utilizar tanto para los administradores como para los propietarios de una comunidad, proporcionando una interfaz simple, con funcionalidades claras y con valor para los usuarios, sin necesidad de configurar nada por su parte.

3. **Usar metodologías de Integración Continua.**
   Desarrollar la aplicación siguiendo procedimientos de Integración Continua (CI), con un flujo constante en la incorporación de cambios y reduciendo las probabilidades de que ocurran errores durante todo el desarrollo.

4. **Adoptar buenas prácticas durante el desarrollo.**
   Diseñar la aplicación teniendo siempre presentes principios y patrones de arquitectura que permitan una fácil escalabilidad y aumenten la calidad del código, haciendo más sencillo su mantenimiento. Para ello se utilizarán patrones de diseño, sumados a principios de código limpio y sostenible.

3

Tecnologías, Herramientas y Metodologías

En este capítulo se detallarán las tecnologías, herramientas y metodologías que han sido empleadas durante el desarrollo del proyecto.

3.1. Tecnologías

3.1.1. Java

Java [1] es el lenguaje de programación utilizado para desarrollar el backend. Ha sido escogido debido a mi amplia experiencia con él en múltiples proyectos, sumado a que existe una gran cantidad de bibliotecas y recursos para su utilización. En concreto, este proyecto ha sido desarrollado con Java en su versión 17.

3.1.2. Spring

Spring [2] es un framework orientado al desarrollo de aplicaciones web. Simplifica la creación de este tipo de aplicaciones integrando, de forma modular, todo lo que requiera la aplicación en cuestión. Se ha escogido por su amplia popularidad y por la abundancia de recursos disponibles para el mismo.

3.1.3. Spring Boot

Spring Boot [3] es una herramienta que permite el desarrollo de aplicaciones web utilizando el framework Spring de una forma rápida y sencilla, simplificando la configuración y el despliegue de las aplicaciones. Su elección es un paso lógico tras haber adoptado el framework Spring para desarrollar el proyecto, ya que permite utilizarlo reduciendo considerablemente la cantidad de código de configuración necesario. La versión utilizada para este proyecto ha sido la 2.7.4, que ofrece una amplia cantidad de recursos y ha sido bien probada durante años en la industria.

3.1.4. Spring Security

Spring Security [4] es un framework encargado de gestionar todo lo relativo a la seguridad de una aplicación Spring, como puede ser la autenticación y autorización en la misma, que es para lo que ha sido configurado en este proyecto, utilizando un proveedor basado en usuario y contraseña, con BCrypt encargándose del cifrado de las contraseñas.

3.1.5. Maven

Maven [5] es una herramienta ampliamente utilizada para Java que permite gestionar y construir proyectos, y que ha sido empleada para gestionar las dependencias y la configuración de la aplicación, simplificando la integración y el despliegue continuo.

3.1.6. MySQL

MySQL [6] es un sistema gestor de bases de datos relacionales bastante popular y más liviano que otros de la competencia, lo que lo convierte en un firme candidato para garantizar un buen rendimiento en la aplicación.

3.1.7. H2

H2 [7] es un motor de base de datos para Java muy ligero, pensado para utilizarse en memoria, de forma que cuando se cierra la aplicación desaparece toda la información contenida. Ha sido utilizado para los tests del proyecto, ya que ofrece una opción bastante sencilla de utilizar, ideal para evitar emplear la base de datos de producción.

3.1.8. Hibernate

Hibernate [8] es una herramienta que permite realizar un mapeo objeto/-relacional (ORM). Se ha utilizado dada su popularidad en aplicaciones Java, implementando la especificación JPA (Java Persistence API)

3.1.9. Tomcat

Tomcat [9] es un contenedor de servlets empleado para desplegar aplicaciones web. Está directamente integrado con Spring Boot, lo que, sumado a su popularidad, lo convierte en una elección natural para el desarrollo del proyecto.

3.1.10. iText PDF

iText PDF [10] es una biblioteca de código abierto que permite la generación y manipulación de documentos PDF en aplicaciones. En estos documentos se puede personalizar cualquier aspecto deseado, desde la fuente de letra hasta la disposición del texto, pasando por los colores o la creación de tablas. Ha sido utilizada para la generación de los informes de las comunidades de propietarios.

3.1.11. JUnit 5

JUnit 5 [11] es un framework para la realización de tests de aplicaciones, que en este caso ha sido utilizado para las pruebas unitarias del backend. Ofrece una amplia variedad de opciones para crear tests parametrizados de forma sencilla, lo que lo convierte en uno de los más populares.

3.1.12. Selenium

Selenium [12] es un paquete de utilidades y bibliotecas orientadas al desarrollo de pruebas automatizadas para aplicaciones web. Concretamente, se ha hecho uso de Selenium WebDriver en su versión 4.12.0, que permite controlar de forma automática un navegador web, interactuando con la página de la aplicación que se esté probando.

3.1.13. JaCoCo

JaCoCo (Java Code Coverage) [13] es una herramienta para realizar análisis de cobertura del código en Java. Es utilizada para llevar a cabo métricas que indican

qué partes del código se han ejecutado durante la realización de las pruebas, proporcionando información útil acerca de la efectividad de las pruebas unitarias y ayudando a detectar las áreas que no están suficientemente cubiertas por los tests. Se ha incorporado como plugin en su versión 0.8.13.

3.1.14. PDFBox

PDFBox [14] es una biblioteca que permite interactuar con documentos PDF. Al ser capaz de obtener el texto de un documento, resulta útil para realizar pruebas de la generación de PDF en la aplicación. Se ha utilizado en su versión 2.0.29.

3.1.15. REST Assured

REST Assured [15] es una biblioteca para realizar pruebas y validar servicios REST en Java. Aporta la simplicidad de testeo de REST que tienen lenguajes como Ruby o Groovy, pero que Java no tiene. Mediante el JSON obtenido al realizar una operación REST es capaz de llevar a cabo múltiples validaciones de manera simple y rápida.

3.1.16. JavaScript

JavaScript [16] es un lenguaje de programación ligero e interpretado que permite desarrollar aplicaciones web interactivas y dinámicas. Debido a su versatilidad, popularidad y amplio soporte en navegadores de internet es un elemento indispensable para el desarrollo de aplicaciones web modernas. En este proyecto ha sido utilizado como lenguaje de programación para la parte de Frontend.

3.1.17. JSON

JSON (JavaScript Object Notation) [17] es la notación de objetos propia del lenguaje de programación JavaScript. Debido a su simplicidad y facilidad de uso, su utilización se ha extendido más allá del propio lenguaje de programación, convirtiéndose en un estándar muy utilizado en la industria para enviar datos a través de internet entre dos aplicaciones, que pueden ser el Frontend y el Backend de una única aplicación.

3.1.18. Jackson

Jackson [18] es la biblioteca que emplea Spring nativamente para parsear los documentos JSON que son utilizados en la aplicación. Permite realizar un manejo más sencillo de este tipo de datos, traduciendo el contenido del documento a objetos de Java.

3.1.19. HTML

HTML [19] es, según las siglas, un lenguaje de marcado de hipertexto, el cual es utilizado para la creación de páginas web. Es el principal estándar de la industria, por lo que no tendría ningún sentido no emplearlo para la maquetación de las páginas web.

3.1.20. CSS

CSS [20] es un lenguaje que permite introducir estilos en la presentación de documentos HTML. Esto permite dar formato y mejorar la apariencia de las páginas web, transmitiendo una mayor calidad al usuario. Dentro de CSS, se ha utlizado la siguiente biblioteca para el proyecto:

- **Font Awesome** [21] es una biblioteca que proporciona una amplia variedad de iconos para su uso en aplicaciones web. Aportan una mayor sensación de calidad a las páginas, permitiendo que cada una pueda desarrollar su propia personalidad.

3.1.21. Vue.js

Vue.js [22] es un framework para JavaScript que simplifica la creación de interfaces web, con una comunidad y una popularidad en constante aumento. Además, proporciona un amplio catálogo de funcionalidades y es más liviano y fácil de utilizar que otros competidores, lo que lo convierte en una gran opción para el desarrollo del Frontend de la aplicación.

3.1.22. Axios

Axios [23] es un cliente HTTP basado en promesas, pensado para utilizarse con JavaScript en un navegador o con Node.js. Permite gestionar las operaciones HTTP (GET, POST, DELETE...) realizadas, convirtiendo automáticamente los

datos en formato JSON. Resulta muy útil para permitir la comunicación entre Frontend y Backend de la aplicación, trabajando concretamente desde la primera de las dos partes.

3.1.23. Node.js

Node.js [24] es un entorno de ejecución de JavaScript multiplataforma que permite ejecutar dicho lenguaje en cualquier lugar, facilitando las labores de desarrollo de una aplicación web.

3.1.24. JSON Web Token

JSON Web Token [25] es un estándar abierto que permite transmitir información de forma segura, compacta y autocontenida en formato JSON. Los tokens pueden ser firmados mediante un secreto o un par de claves pública/privada. Con su encriptación, los JWT resultan muy útiles para gestionar el inicio de sesión en una aplicación web, ya que mantienen tanto usuario como contraseña protegidos y cifrados, pero permiten comprobar rápidamente si una conexión está autenticada o no.

3.2. Herramientas

3.2.1. Spring Tool Suite

Spring Tool Suite [26] es un entorno de desarrollo (IDE) que utiliza como base a Eclipse, orientado al desarrollo de aplicaciones Java con el framework Spring. Permite depurar y lanzar las aplicaciones Spring cómodamente y con todo lo necesario ya incluido de serie, simplificando el proceso.

3.2.2. Visual Studio Code

Visual Studio Code [27] es un editor de código muy liviano, pero a la vez muy versátil, ya que ofrece instalar una elevada cantidad de variados complementos que permiten personalizar la experiencia y funcionalidad a gusto de cada usuario. Es por ello que soporta una gran cantidad de lenguajes de programación, pero en concreto es especialmente útil para el desarrollo de aplicaciones web, específicamente la parte del Frontend, ya que permite trabajar simultáneamente con todos los tipos de archivo necesarios.

3.2.3. Git

Git [28] es una herramienta de control de versiones muy importante para el desarrollo de software y su posterior mantenimiento. Facilita la colaboración entre los miembros de un equipo de desarrollo y permite gestionar de manera más sencilla las diferentes versiones que ha tenido un software a lo largo de su vida, controlando los cambios que se han llevado a cabo y quién los ha realizado.

3.2.4. GitHub

GitHub [29] es una plataforma colaborativa online que permite alojar proyectos de software que utilicen Git para su desarrollo. Es la opción más popular dentro de este segmento, y proporciona funciones como GitHub Actions que permiten realizar tanto Integración Continua como Despliegue Continuo en el proyecto.

3.2.5. Docker

Docker [30] es una herramienta que automatiza el despliegue de aplicaciones dentro de contenedores de software, permitiendo la abstracción y virtualización de las mismas en diversos sistemas operativos. Simplifica el proceso de compilación del código de forma automática, además de permitir combinar Frontend y Backend en una sola aplicación, generando una imagen como resultado.

3.2.6. Docker Hub

Docker Hub [31] es una plataforma que permite almacenar y registrar imágenes de Docker. De esta forma, se pueden conservar imágenes de distintas versiones de una misma aplicación, permitiendo escoger la que interese para desplegarla y trabajar con ella en cualquier momento.

3.2.7. Azure

Azure [32] es una plataforma de servicios en la nube que proporciona gran cantidad de funcionalidades para desarrollar, implementar y gestionar aplicaciones web. Permite desplegar aplicaciones en su plataforma para poder acceder a ellas a través de internet. Para ello, en este proyecto se han utilizado los siguientes servicios:

- **Azure Container Instances** [33] es un servicio que permite ejecutar aplicaciones en contenedores sin necesidad de configurar complejas infraestruc-

turas, ya que proporciona todo lo necesario para simplificar el proceso de configuración.

- **Servidor flexible de Azure Database for MySQL** [34] es un servicio que permite desplegar y gestionar bases de datos MySQL de forma flexible y escalable. Estas bases de datos pueden ser utilizadas por las aplicaciones web para almacenar su información, estén o no desplegadas en Azure.

3.3. Metodologías

Para el desarrollo del proyecto se ha seguido un enfoque que combina distintas metodologías para mejorar la gestión del código, la integración de nuevas funcionalidades y el Despliegue Continuo de nuevas versiones de la aplicación para producción. A continuación se describen todas las metodologías que han sido empleadas.

3.3.1. Git Flow

Durante el desarrollo de la aplicación se ha utilizado el sistema de control de versiones Git [28]. Para estructurar el flujo de trabajo se ha seguido un enfoque basado en un modelo simplificado de Git Flow [35], adecuado para proyectos desarrollados por una única persona.

Se han utilizado dos ramas principales (Figura 3.1):

- **Develop**: rama destinada al desarrollo activo. Todos los cambios y nuevas funcionalidades han sido implementados inicialmente en esta rama.
- **Master**: rama principal para el entorno de producción. Aquí es donde se realizan merge desde develop cuando se quieren trasladar los cambios a producción.

El flujo se ha integrado con un sistema de CI/CD (Integración Continua y Despliegue Continuo) a través de GitHub Actions [36]. Con este sistema se permite que, cada vez que se realiza un merge en la rama Master, se desencadene de forma automática la ejecución de una batería de pruebas, permitiendo la rápida detección de errores. Posteriormente, si el código pasa las pruebas, se realizará el despliegue automático de la aplicación, garantizando que nunca se publique una versión de la aplicación con errores.

Este enfoque ha permitido mantener una clara separación entre el código en desarrollo y el código en producción, asegurando estabilidad en los despliegues sin

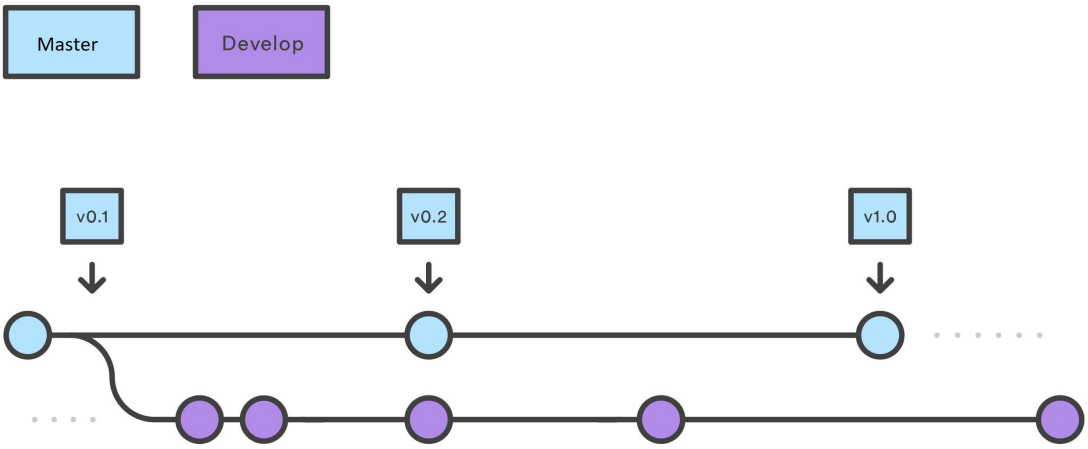


Figura 3.1: Ejemplo de flujo de trabajo de Git Flow simplificado

necesidad de crear ramas adicionales, lo que es apropiado dado que he desarrollado este proyecto de forma individual.

3.3.2. Metodologías Ágiles

Para el desarrollo del proyecto no se ha implementado un marco ágil formal como puede ser Scrum [37], con ciclos de trabajo o sprints definidos, pero la metodología Git Flow sugiere un enfoque ágil implícito. La causa es la propia naturaleza del proyecto, con un desarrollo realizado y guiado de manera individual, sin tener que coordinar un equipo de personas, por lo que ha sido innecesario establecer un marco formal para un desarrollo colaborativo. A pesar de ello, la entrega incremental de nuevas funcionalidades, así como la flexibilidad para adaptar el proyecto a cambios o nuevas necesidades no previstas inicialmente, son aspectos clave de las metodologías ágiles reflejados en el desarrollo de este proyecto.

Gracias a este enfoque ágil se ha podido trabajar de forma continua en mejoras y nuevas funcionalidades, manteniendo un ritmo constante de entregas sin poner en un compromiso a la calidad del desarrollo. Además, durante el desarrollo se han ido sucediendo reuniones con el tutor del proyecto periódicamente para establecer los siguientes objetivos, de manera similar a como se hace con los sprints de Scrum, lo que contribuye a mantener el mencionado enfoque ágil sugerido por la naturaleza de este proyecto.

3.3.3. DevOps

En este proyecto también se adoptan principios fundamentales de DevOps (Development Operations, Operaciones de Desarrollo) [38] mediante la automa-

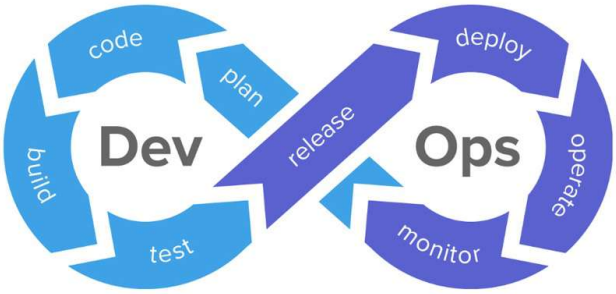


Figura 3.2: Ciclo de DevOps

tización de procesos clave de los que ya se ha hecho mención (Figura 3.2). Ha sido implementado para el proyecto un sistema de Integración Continua (CI) y Despliegue Continuo (CD) mediante GitHub Actions [36].

Cuando se realizan cambios en la aplicación, éstos son subidos a la rama develop, donde se mantiene todo el código en desarrollo. Una vez se ha determinado que interesa llevar estos cambios a producción, se realiza un merge de develop con master, y posteriormente se hace un push de esta última para que la rama esté actualizada en GitHub. Una vez allí, GitHub Actions se encarga de ejecutar todas las pruebas necesarias para asegurar que no se despliega una versión de la aplicación con errores. Si el código pasa las pruebas, entonces se procede a desplegar la aplicación a producción.

El despliegue continuo permite que las nuevas versiones sean lanzadas de forma automática, reduciendo la necesidad de que intervengan personas y asegurando que la aplicación se encuentre siempre al día en el entorno de producción. Además, mediante la generación de imágenes con Docker y el despliegue con Azure Container Instances se garantiza que las nuevas versiones sean consistentes y se puedan replicar fácilmente.

4

Descripción Informática

En este capítulo se realiza una descripción completa del proyecto de software que se ha llevado a cabo, analizando con detalle todos los aspectos fundamentales de su desarrollo, desde la definición de requisitos inicial hasta el despliegue final. El objetivo consiste en proporcionar una visión completa y detallada de la forma en la que se ha realizado el proyecto, empezando por la planificación y diseño y terminando por la implementación y pruebas. A lo largo del capítulo se presentarán los elementos clave y las decisiones técnicas que han establecido el desarrollo del proyecto, aportando una buena base para comprender el resultado final.

4.1. Requisitos

En esta sección se abordarán los requisitos funcionales y no funcionales planteados en la aplicación, tanto los iniciales como los que se han ido planteando a lo largo del desarrollo. Los requisitos se muestran como una lista detallada, proporcionando una buena manera de entender las funcionalidades que debe implementar la aplicación.

4.1.1. Requisitos Funcionales

Los requisitos funcionales especifican tanto la funcionalidad como las características que debe tener el sistema para cumplir con las expectativas de los

usuarios y las necesidades del sector. Con estos requisitos se describen las acciones que cada tipo de usuario puede realizar con la aplicación y las capacidades que se deben ofrecer.

Según el rol del usuario, el sistema debe permitir llevar a cabo las siguientes acciones (organizadas por secciones de la aplicación) para garantizar un buen uso completo y eficaz de la aplicación:

Rol MANAGER (Administrador de la comunidad de propietarios)

- **Gestión de usuarios:**
  - RF1: Los administradores podrán iniciar sesión en el sistema.
  - RF2: Los administradores podrán cerrar sesión en el sistema.
  - RF3: Los administradores podrán visualizar sus datos personales.
  - RF4: Los administradores podrán modificar sus datos personales, a excepción de aquellos que sean invariables.
- **Recibos:**
  - RF5: Los administradores podrán visualizar todos los recibos de la comunidad.
  - RF6: Los administradores podrán crear nuevos recibos de forma individual.
  - RF7: Los administradores podrán crear recibos mensuales para todos los inmuebles a la vez.
  - RF8: Los administradores podrán marcar como pagados los recibos de forma individual.
  - RF9: Los administradores podrán marcar como pagados los recibos de un mes, filtrando por Derrama sí o no.
  - RF10: Los administradores podrán marcar el pago con demora de un recibo de forma individual.
  - RF11: Los administradores podrán eliminar los recibos de forma individual.
- **Movimientos:**
  - RF12: Los administradores podrán visualizar todos los movimientos de la comunidad.
  - RF13: Los administradores podrán crear nuevos movimientos de forma individual.

- RF14: Los administradores podrán incorporar los recibos de un mes para crear un movimiento, filtrando por Derrama sí o no.
- RF15: Los administradores podrán editar los movimientos de forma individual.
- RF16: Los administradores podrán eliminar los movimientos de forma individual.

- **Consultas e informes:**

  - RF17: Los administradores podrán crear informes de recibos de inmuebles.
  - RF18: Los administradores podrán crear informes de consultas de deuda de inmuebles.
  - RF19: Los administradores podrán crear informes de las cuentas anuales de la comunidad.

- **Gestión de vecinos:**

  - RF20: Los administradores podrán visualizar todos los usuarios de la comunidad.
  - RF21: Los administradores podrán crear nuevos usuarios con rol NEIGHBOR (Vecino) de forma individual.
  - RF22: Los administradores podrán editar los vecinos de forma individual.
  - RF23: Los administradores podrán eliminar los vecinos de forma individual.

- **Gestión de inmuebles:**

  - RF24: Los administradores podrán visualizar todos los inmuebles de la comunidad.
  - RF25: Los administradores podrán crear nuevos inmuebles de forma individual.
  - RF26: Los administradores podrán editar los inmuebles de forma individual.
  - RF27: Los administradores podrán eliminar los inmuebles de forma individual.

- **Gestión de zonas comunes:**

  - RF28: Los administradores podrán visualizar todas las zonas comunes de la comunidad.

  - RF29: Los administradores podrán crear nuevas zonas comunes de forma individual.
  - RF30: Los administradores podrán editar las zonas comunes de forma individual.
  - RF31: Los administradores podrán eliminar las zonas comunes de forma individual.

- **Gestión de reservas:**

  - RF32: Los administradores podrán visualizar todas las reservas de zonas comunes de la comunidad.
  - RF33: Los administradores podrán eliminar las reservas de zonas comunes de forma individual.

- **Gestión de la comunidad:**

  - RF34: Los administradores podrán visualizar la información de la comunidad.
  - RF35: Los administradores podrán modificar la información de la comunidad.

- **Incidencias:**

  - RF36: Los administradores podrán visualizar todas las incidencias de la comunidad.
  - RF37: Los administradores podrán eliminar las incidencias de forma individual.

- **Documentos de la comunidad:**

  - RF38: Los administradores podrán visualizar todos los documentos de la comunidad.
  - RF39: Los administradores podrán añadir nuevos documentos para la comunidad de forma individual.
  - RF40: Los administradores podrán descargar los documentos de forma individual.
  - RF41: Los administradores podrán eliminar los documentos de forma individual.

Rol NEIGHBOR (Vecino de la comunidad de propietarios)

- **Gestión de usuarios:**
  - RF42: Los vecinos podrán iniciar sesión en el sistema.
  - RF43: Los vecinos podrán cerrar sesión en el sistema.
  - RF44: Los vecinos podrán visualizar sus datos personales.
  - RF45: Los vecinos podrán modificar sus datos personales, a excepción de aquellos que sean invariables.
- **Mis inmuebles:**
  - RF46: Los vecinos podrán visualizar únicamente sus inmuebles.
- **Reservas:**
  - RF47: Los vecinos podrán visualizar sus propias reservas.
  - RF48: Los vecinos podrán eliminar sus propias reservas.
  - RF49: Los vecinos podrán crear nuevas reservas.
  - RF50: Los vecinos podrán visualizar todas las reservas de la zona común escogida para el día elegido.
- **Mis recibos:**
  - RF51: Los vecinos podrán visualizar todos los recibos correspondientes a sus inmuebles en propiedad.
- **Mis incidencias:**
  - RF52: Los vecinos podrán visualizar todas las incidencias creadas por ellos mismos.
  - RF53: Los vecinos podrán eliminar sus incidencias de forma individual.
  - RF54: Los vecinos podrán crear nuevas incidencias de forma individual.
- **Contactar:**
  - RF55: Los vecinos podrán visualizar los datos de contacto del administrador de la comunidad.
  - RF56: Los vecinos podrán visualizar los nombres de los principales responsables de la comunidad.
- **Documentos de la comunidad:**
  - RF57: Los vecinos podrán visualizar todos los documentos de la comunidad.
  - RF58: Los vecinos podrán descargar los documentos de forma individual.

4.1.2. Requisitos No Funcionales

Los requisitos no funcionales son aquellos que describen los criterios que debe seguir la aplicación para juzgar el funcionamiento del sistema, en lugar de detallar comportamientos específicos.

- RNF1: La aplicación debe ser intuitiva y fácil de usar.
- RNF2: La aplicación debe poder adaptarse a distintos tamaños de pantalla para dispositivos de escritorio.
- RNF3: La aplicación debe ser capaz de trabajar con distintos roles de usuario, especificando las acciones concretas que podrá realizar cada usuario en función de su rol.
- RNF4: La aplicación debe almacenar sus datos en una base de datos relacional que sea segura y robusta.
- RNF5: La aplicación debe tener una serie de pruebas que garanticen el buen funcionamiento del sistema.
- RNF6: La aplicación debe tener sistemas que soliciten confirmación antes de realizar cualquier acción importante.
- RNF7: La aplicación debe permitir trabajar con múltiples sesiones de usuario en distintos equipos de forma simultánea.

4.2. Arquitectura y Análisis

En esta sección se describirá la arquitectura y se realizará un análisis de la aplicación desarrollada. La arquitectura de la misma, como se puede apreciar en la Figura 4.1, ha sido diseñada utilizando el modelo de N-capas, utilizando en concreto una estructura con tres capas: capa de presentación (Frontend), capa de aplicación (Backend) y capa de persistencia (Base de datos). Con esta arquitectura se consigue una separación clara de responsabilidades, lo que facilita el mantenimiento, la escalabilidad y la fiabilidad del sistema, además de ser más sencillo llevar a cabo el desarrollo de forma organizada.

Adicionalmente, para realizar el despliegue de la aplicación se ha contado con la ayuda de tecnologías como Azure Container Instances, Azure Database for MySQL y Docker. A continuación, se detalla con mayor precisión cada una de las tres capas que componen la aplicación.

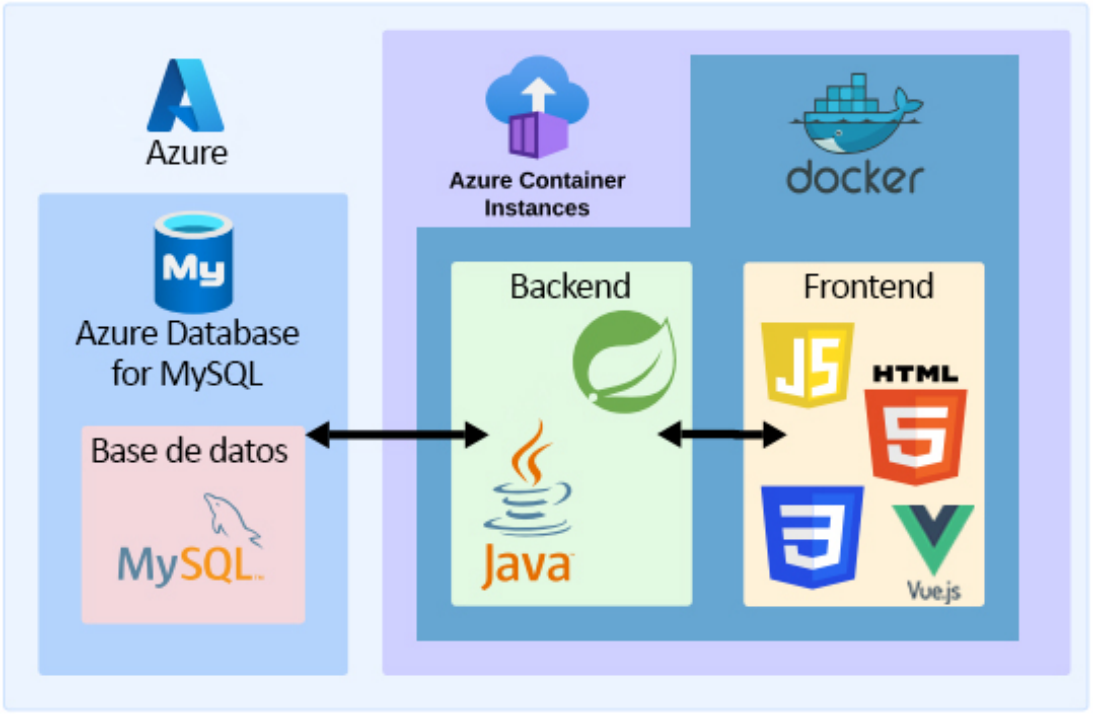


Figura 4.1: Arquitectura general de Moai

4.2.1. Capa de presentación (Frontend)

La capa de presentación es la que se encarga de la interacción del usuario con la aplicación. También conocida popularmente como Frontend, su principal objetivo es gestionar la entrada y salida de los datos, proporcionando al usuario una interfaz intuitiva y visualmente atractiva que facilite la interacción con el sistema, simplificando lo máximo posible el trabajo del usuario y ocultando todos los detalles técnicos.

Principales funciones de la capa de presentación:

- **Interacción del usuario**
  La capa de presentación se encarga de recibir las acciones del usuario mediante elementos de la interfaz gráfica como formularios, botones, menús y otros componentes interactivos.

  Incluye paneles de control adaptados según el rol del usuario, mostrando información relevante y específica para sus funciones. Esto permite ofrecer una experiencia personalizada y orientada a facilitar sus tareas.

- **Generación dinámica de contenido**
  En la capa de presentación, el uso del framework Vue.js (Figura 4.2) permite construir interfaces dinámicas y reactivas directamente en el navegador.

Figura 4.2: Framework Vue.js utilizado para el desarrollo del Frontend

Gracias al sistema de enlace de datos y a su estructura basada en componentes, la interfaz se actualiza automáticamente cuando cambian los datos subyacentes. Esto permite ofrecer una experiencia de usuario fluida y coherente.

Además, Vue.js proporciona directivas como v-for, v-if o v-model, que facilitan la gestión de contenido dinámico, listas, formularios y estructuras condicionales de manera declarativa. Esto simplifica el desarrollo y la adaptación de las vistas a diferentes estados de la aplicación.

- **Visualización y gestión de datos**
  Esta capa presenta datos complejos de forma comprensible, utilizando componentes visuales como tablas y gráficos interactivos. Estas herramientas permiten aplicar funciones como ordenamiento, filtrado y paginación, facilitando el manejo de grandes cantidades de información.

- **Actualización en tiempo real**
  La aplicación utiliza técnicas de comunicación asíncrona, como las proporcionadas por la biblioteca Axios, para intercambiar datos con el backend sin necesidad de recargar la página. Combinadas con el sistema reactivo de Vue.js, estas llamadas permiten actualizar partes concretas de la interfaz tan pronto como se reciben nuevas respuestas del servidor.

  Este enfoque mejora notablemente la eficiencia y la fluidez de la interacción del usuario, ya que las operaciones como guardar cambios o cargar información se ejecutan sin interrupciones visuales.

- **Diseño adaptable**
  El diseño adaptable garantiza que la interfaz se ajuste correctamente a distintas resoluciones y tamaños de pantalla, asegurando una experiencia de usuario coherente y accesible desde diferentes dispositivos.

4.2.2. Capa de aplicación (Backend)

La capa de aplicación, conocida popularmente como Backend, es la encargada de la gestión de la lógica de negocio, del procesamiento de las solicitudes de los usuarios y de la interacción con la base de datos, además de garantizar el correcto funcionamiento del sistema. Esta capa es la intermediaria entre la interfaz de usuario (capa de presentación) y los datos almacenados (capa de persistencia), aportando características fundamentales para la operatividad de la aplicación.

Principales funciones de la capa de aplicación:

- **Procesamiento de solicitudes**
  El Backend de la aplicación es responsable de recibir y atender las peticiones que llegan desde el Frontend. Estas solicitudes son procesadas aplicando la lógica de negocio correspondiente, y se responde con los datos adecuados. Para estructurar este comportamiento, se ha empleado el patrón arquitectónico Spring MVC[39] (Modelo-Vista-Controlador), que permite organizar de forma clara la interacción entre los controladores, los servicios y los modelos de datos.

- **Interacción con la base de datos**
  Toda la gestión de datos se realiza mediante operaciones CRUD (crear, leer, actualizar y eliminar). Estas acciones se llevan a cabo mediante Spring Data JPA y Hibernate, que permiten mapear las entidades Java con las tablas de la base de datos de manera eficiente y transparente, simplificando las tareas de persistencia gracias al uso de anotaciones y repositorios.

- **Gestión de la seguridad**
  La aplicación implementa un sistema de seguridad basado en Spring Security, que se encarga de autenticar a los usuarios y autorizar el acceso a los distintos recursos del sistema en función de los roles definidos. Esta configuración asegura que solo los usuarios con los permisos adecuados puedan realizar determinadas acciones dentro del sistema.

- **Implementación de la lógica de negocio**
  La lógica que define el comportamiento del sistema se implementa en Java (Figura 4.3), y se organiza en servicios y controladores dentro del Backend. La capa de aplicación es la encargada de aplicar las reglas de negocio, validar la información recibida y coordinar las respuestas adecuadas al Frontend, actuando como intermediaria entre los datos persistidos y la interfaz de usuario.

Figura 4.3: Lenguaje Java utilizado para el desarrollo del Backend

4.2.3. Capa de persistencia (Base de datos)

La capa de persistencia es la que se dedica al almacenamiento y recuperación de datos de forma eficiente y segura. Se manejan distintas entidades y sus relaciones para garantizar la integridad y la consistencia de los datos almacenados durante el uso de la aplicación.

Principales funciones de la capa de persistencia:

- **Gestión del almacenamiento de datos**
  La capa de persistencia se encarga principalmente del almacenamiento estructurado de la información. Para ello, se apoya en un sistema de gestión de bases de datos relacional como MySQL (Figura 4.4), donde se definen y administran tablas que contienen los datos fundamentales de la aplicación, como usuarios, roles, inmuebles, incidencias o recibos, entre otros.

- **Acceso y recuperación eficiente de datos**
  Proporciona los mecanismos necesarios para acceder a los datos cuando se requieren, permitiendo realizar consultas complejas que abarcan múltiples tablas y relaciones. Esto garantiza que la información relevante esté disponible para el procesamiento posterior en la capa de aplicación.

- **Soporte de operaciones CRUD**
  Esta capa implementa las operaciones básicas de creación, lectura, actualización y eliminación (CRUD), esenciales para mantener la integridad de los datos y reflejar correctamente las modificaciones que se producen en la lógica de negocio.

- **Mapeo Objeto-Relacional (ORM)**
  Se emplea Spring Data JPA para mapear clases Java a tablas en la base de datos, lo que evita la necesidad de escribir SQL de forma manual. Este enfoque proporciona una abstracción de alto nivel que facilita la gestión de las entidades y sus relaciones. Además, permite definir consultas derivadas

Figura 4.4: Sistema de gestión de base de datos MySQL utilizado en Moai

automáticamente a partir de los nombres de los métodos en los repositorios, así como consultas personalizadas mediante anotaciones. También se encarga de gestionar las transacciones de forma automática, mejorando la eficiencia y robustez de las operaciones sobre la base de datos.

4.2.4. Comunicación entre capas

Para que las capas de una aplicación que sigue la arquitectura N-capas funcionen correctamente, es fundamental que exista una buena comunicación entre ellas. Los datos deben fluir desde la capa de persistencia hasta la capa de presentación, pasando por la de aplicación. Para ello, cada capa interactúa con la siguiente mediante interfaces bien definidas y utilizando todos los protocolos de comunicación necesarios. A continuación, se explica con detalle el sistema de comunicación entre capas, que se puede visualizar de forma gráfica en la Figura 4.5:

- **Comunicación entre la capa de presentación y la capa de aplicación**

  La interacción entre el Frontend (capa de presentación) y el Backend (capa de aplicación) se lleva a cabo principalmente mediante solicitudes HTTP. Estas solicitudes siguen los métodos estándar (también llamados verbos) del protocolo, como GET, POST, PUT y DELETE, en función del tipo de operación que se desee ejecutar.

  En el caso de esta aplicación, el Frontend realiza peticiones al Backend utilizando la librería Axios para Vue.js. Esta herramienta permite enviar solicitudes de forma asíncrona, lo cual favorece una experiencia de usuario fluida y dinámica sin necesidad de recargar la página.

  El Backend responde a estas solicitudes devolviendo datos en formato JSON, que es ampliamente compatible y fácil de manipular desde el lado del cliente. Este formato permite que los datos puedan ser consumidos directamente

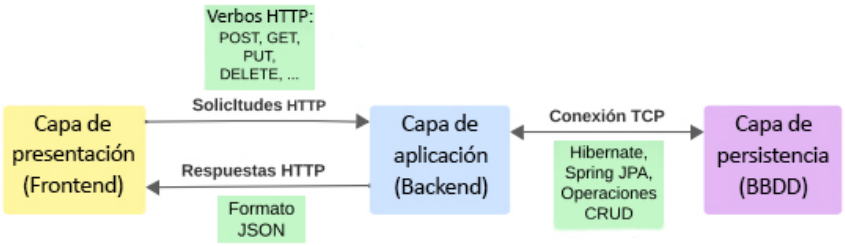


Figura 4.5: Esquema de la comunicación entre capas de Moai

por los componentes del Frontend y presentados al usuario de forma clara y eficiente.

- **Comunicación entre la capa de aplicación y la capa de persistencia**

  La interacción entre la capa de aplicación (Backend) y la capa de persistencia (Base de datos) se establece mediante conexiones TCP, a través de las cuales se intercambian datos utilizando Hibernate como framework de mapeo objeto-relacional (ORM).

  En esta arquitectura, el Backend configura y mantiene una conexión TCP con el servidor de base de datos MySQL, definida en los archivos de propiedades de Spring Boot. Esta conexión permite ejecutar operaciones de almacenamiento y recuperación de datos durante el ciclo de vida de la aplicación.

  Hibernate, junto con Spring Data JPA, se encarga de mapear automáticamente las clases Java a las tablas correspondientes de la base de datos. Las operaciones sobre los objetos en Java se traducen internamente a sentencias SQL que se ejecutan mediante dicha conexión, facilitando así la persistencia de datos sin necesidad de escribir consultas SQL manualmente.

4.3. Diseño e Implementación

En esta sección se describirá con mayor detalle cómo se ha llevado a cabo la implementación y el diseño de la aplicación en cada parte de la misma.

4.3.1. Backend

Diseño

Para realizar el diseño de la aplicación se han seguido distintos estándares de la programación orientada a objetos que aseguran la mantenibilidad, la extensibilidad y la testabilidad del código. Los principales son:

- **Modelo-Vista-Controlador (MVC) y Principio de responsabilidad única**

  El diseño de la aplicación sigue el patrón arquitectónico Modelo-Vista-Controlador (MVC) mostrado en el Código 4.1, respetando el principio de responsabilidad única, que establece que cada clase debe encargarse exclusivamente de una tarea concreta.

  Esta separación de responsabilidades se refleja claramente en la estructura del Backend. Los controladores reciben y gestionan las peticiones HTTP provenientes del Frontend y actúan como puente entre las vistas (interfaz de usuario) y el modelo de datos. Por su parte, los servicios encapsulan la lógica de negocio, permitiendo una mayor reutilización y mantenibilidad del código. Finalmente, los repositorios son responsables de todas las operaciones relacionadas con la persistencia de datos, utilizando Spring Data JPA para interactuar con la base de datos. Esta organización promueve una arquitectura limpia, modular y escalable.

- **Inversión de Control (IoC)**

  La Inversión de Control, aplicada mediante el mecanismo de inyección de dependencias, refuerza la arquitectura modular de la aplicación. En lugar de que las clases gestionen directamente la creación de sus dependencias, es el contenedor de Spring quien se encarga de suministrarlas automáticamente. Esto se logra principalmente a través de la anotación **@Autowired**, como se muestra en el Código 4.1. Gracias a esta estrategia, se reduce significativamente el acoplamiento entre componentes, lo que facilita su desarrollo, prueba y mantenimiento de forma independiente. La IoC contribuye así a una estructura más limpia, flexible y escalable.

Código 4.1: Ejemplo de Responsabilidad única e Inversión de Control para el tratamiento de los inmuebles

```
@RestController
@RequestMapping("/api/property")
public class PropertyController {

    @Autowired
        private PropertyService propertyService;

    @Autowired
    private CommunityService communityService;

    @Autowired
    private UserService userService;

    @GetMapping("/")
    @Transactional
    public ResponseEntity<List<Property>> getProperties() {
            return ResponseEntity.ok(propertyService.findAll());
    }
```

- **Reutilización de código**

  La incorporación de clases utilitarias ha sido clave para mejorar la organización del código y evitar su duplicación. Estas clases permiten centralizar funcionalidades comunes, lo que facilita tanto el mantenimiento como la evolución del sistema. Un caso concreto se muestra en el Código 4.2, donde se presenta la clase **ErrorResponse.java**, que establece las respuestas en caso de error compartidas por distintos módulos de la aplicación, garantizando coherencia y reduciendo la cantidad de código del proyecto.

Código 4.2: Ejemplo de reutilización de código en la clase ErrorResponse.java

```
public class ErrorResponse {

    private String errorMessage = null;

    public ErrorResponse(String errorMessage) {
            this.errorMessage = errorMessage;
    }

    public String getErrorMessage() {
            return errorMessage;
    }

    public void setErrorMessage(String errorMessage) {
            this.errorMessage = errorMessage;
    }

}
```

Implementación

- **Controladores**

  Para optimizar el funcionamiento de la aplicación, se ha creado un controlador dedicado para cada función específica, mejorando la escalabilidad del sistema y la limpieza del código.

  1. **CommonAreaController**

     El controlador **CommonAreaController** se encarga de gestionar las zonas comunes de la comunidad, permitiendo su creación, actualización, visualización y eliminación. Se realizan todas las verificaciones necesarias para garantizar que los datos son utilizados de forma correcta y sin errores. Tiene inyectados los servicios **CommonAreaService**, **ReservationService** y **CommunityService** para poder realizar todas las gestiones necesarias.

  2. **CommunityController**

     El controlador **CommunityController** es el encargado de llevar a cabo las acciones respectivas a la propia comunidad de vecinos. Permite crear una nueva comunidad para la primera inicialización de la aplicación y para dar soporte a posibles ampliaciones del proyecto. También permite su visualización, edición y eliminación.

     Además, añade algunas funciones específicas que permiten obtener directamente el usuario administrador de la comunidad, obtener únicamente la imagen personalizada de la comunidad o eliminar esta imagen. Con estas funciones se permite ejecutar un código simplificado que realiza exclusivamente la tarea necesaria, en lugar de tener que pasar por procesos más complejos e innecesarios. Para poder realizar todas las acciones necesarias se ha inyectado el servicio **CommunityService**.

  3. **DocumentController**

     Mediante el controlador **DocumentController** se realiza toda la gestión respectiva a los documentos de la comunidad. Estos documentos pueden ser los PDF que se comparten para los propietarios o la imagen personalizada de la comunidad que se utiliza como fondo de la aplicación. Por lo tanto, se permite visualizar, crear, actualizar o eliminar los documentos. Para cumplir sus funciones, este controlador tiene inyectados los servicios **DocumentService** y **CommunityService**.

4. **FrontendController**

   El controlador **FrontendController** cumple una única y simple función, pero que es muy importante para el correcto funcionamiento del sistema. Se trata de filtrar las solicitudes que se reciben desde el navegador para poder diferenciar entre una solicitud de contenido, como puede ser el icono mostrado por el navegador para la página (conocido como "favicon"), o solicitudes de tipo REST para realizar acciones con alguna entidad, como puede ser actualizar la información de un inmueble. Las solicitudes de tipo REST no continúan a través del filtro, por lo que es el Backend, mediante el controlador apropiado, el encargado de gestionarlas. Sin embargo, las solicitudes de contenido sí pasan a través del filtro y son redirigidas para que sea el Frontend el encargado de gestionarlas correctamente.

5. **IssueController**

   El controlador **IssueController** es el encargado de gestionar correctamente todo lo relativo a las incidencias. Para poder trabajar con ellas, se permite visualizarlas, crearlas, editarlas y eliminarlas. Además, permite obtener exclusivamente las incidencias de un usuario en concreto, facilitando el trabajo para el Frontend a la hora de mostrar dicha información. Los servicios que se han inyectado son **IssueService**, **CommunityService** y **UserService**.

6. **LoginController**

   Con **LoginController** se gestiona el inicio de sesión y autenticación de los usuarios en el sistema. Mediante la inyección de los servicios **UserService**, **CommunityService**, **AuthenticationManager** y **JwtTokenManager** se comprueba que el DNI y la contraseña recibidos son válidos y coinciden con los de algún usuario del sistema y, si es así, se procede a crear un token JWT que contiene toda la información que necesita el cliente para poder realizar futuras solicitudes al sistema, como puede ser el DNI o el rol del usuario. Este token solo permite su uso en el navegador desde el que se quiere iniciar sesión, y es válido durante toda la sesión del usuario en la aplicación, caducando una vez que termine la misma.

   Adicionalmente, este controlador posee la funcionalidad necesaria para realizar el primer inicio del sistema, ya que, si un usuario se encuentra con la pantalla de inicio de sesión e intenta iniciarla sin que haya ningún usuario registrado en el sistema, el controlador procederá a crear en la base de datos todos los datos necesarios para comenzar a utilizar la aplicación. Se creará el usuario administrador de la comunidad, así como una nueva comunidad de propietarios, para tener el punto inicial

desde el que poder personalizar el sistema al gusto del cliente.

7. **MovementController**

   El controlador **MovementController** es el encargado de gestionar todo lo relacionado con los movimientos utilizados en la comunidad. Estos movimientos se pueden visualizar, crear, editar y eliminar con la ayuda de la inyección de los servicios **MovementService**, **CommunityService** y **ReceiptService**. Además, el controlador incorpora algunas funciones adicionales que mejoran la experiencia utilizando los movimientos. Una de ellas es la obtención de los nombres de los conceptos permitidos para los movimientos, agilizando el trabajo del Frontend y proporcionando una mayor rapidez en el sistema. También permite validar si un concepto recibido está soportado o no por la aplicación.

   Aparte de las funciones para los conceptos, también se incluye otra función destinada exclusivamente a añadir los recibos mensuales. A partir de un mes y un estado de derrama, el sistema selecciona todos los recibos coincidentes y crea un nuevo movimiento con la suma de todos los importes, automatizando una tarea que de otra forma resultaría mucho más costosa para el administrador de la comunidad.

8. **PropertyController**

   El encargado de la gestión de los inmuebles es el controlador **PropertyController**. A través de él se permite visualizarlos, actualizarlos, crearlos y eliminarlos. Adicionalmente, el controlador permite obtener los tipos de inmueble que admite el sistema, además de poder proporcionar de forma directa todos los inmuebles correspondientes a un determinado propietario. Para esto, se inyectan los servicios **PropertyService**, **CommunityService** y **UserService**.

9. **ReceiptController**

   Mediante el controlador **ReceiptController** se gestionan correctamente los recibos de la comunidad. Se pueden visualizar, actualizar, crear y eliminar, pero la funcionalidad proporcionada por este controlador va mucho más allá, realizando la inyección de los servicios **ReceiptService**, **PropertyService** y **UserService**.

   El controlador permite obtener exclusivamente los recibos correspondientes a un determinado usuario. Puesto que los recibos son asignados a los inmuebles, de esta forma se permite obtener rápidamente y de una sola vez todos los recibos de los inmuebles que el usuario tiene en propiedad. Además, el controlador también permite marcar los reci-

bos como pagados. A través de esta funcionalidad, se pueden marcar múltiples recibos como pagados de forma simultánea, escogiendo para ello un mes y un estado de derrama.

Otra funcionalidad proporcionada por el controlador es la que permite la creación de lotes de recibos, que serán los correspondientes a determinado mes para todos los inmuebles de la comunidad y según si son derrama o no. Como opción, este lote de recibos también se puede crear utilizando el coeficiente de participación de cada inmueble, en lugar de establecer un importe fijo para todos los inmuebles por igual. De esta manera, se simplifica el trabajo que tiene que realizar mensualmente el administrador de la comunidad.

Para llevar a cabo estas acciones, se codificó un método privado en el controlador que se encarga de calcular el importe correspondiente a un recibo, dada la cantidad de dinero total a recaudar entre todos los inmuebles y el coeficiente de participación del inmueble concreto. Este método se puede observar en el Código 4.3.

Código 4.3: Método utilizado para el cálculo del importe correspondiente a un recibo

```
1  // Metodo que calcula el importe correspondiente a un recibo, dada una cantidad
       total a recaudar y un coeficiente de participacion de la propiedad
2  private String calculateReceiptQuantity(String totalQuantity, String
       ownershipShare) throws ParseException {
3      try {
4          // Para formatear los decimales desde String
5          DecimalFormat df = new DecimalFormat();
6
7          // Se parsea la cantidad total
8          Number totalQuantityNumber = df.parse(totalQuantity);
9          Float totalQuantityParsed = totalQuantityNumber.floatValue();
10
11         // Se parsea el coeficiente de participacion
12         Number ownershipShareNumber = df.parse(ownershipShare);
13         Float ownershipShareParsed = ownershipShareNumber.floatValue();
14
15         Float receiptQuantity = ((ownershipShareParsed / 100) *
               totalQuantityParsed);
16
17         // Se parsean los decimales a String utilizando la coma y manteniendo
               unicamente dos decimales en caso de que sean distintos a cero
18         df = new DecimalFormat("#,##0.##");
19         String receiptQuantityFinal = df.format(receiptQuantity);
20
21         return receiptQuantityFinal;
22     } catch (ParseException e) {
23         throw e;
24     }
25 }
```

10. **ReportController**

    El controlador **ReportController** es el encargado de gestionar los informes de la comunidad. Los informes pueden ser creados utilizando la información almacenada en la base de datos, y proporcionan un documento PDF como resultado, que el administrador puede descargar e imprimir libremente. Para lograrlo, se inyectan los servicios **CommunityService**, **MovementService**, **PropertyService**, **UserService** y **PdfService**.

    El controlador permite crear hasta tres tipos diferentes de informes. El primero de ellos es el informe de recibo, que genera un documento con toda la información correspondiente a un recibo concreto. El segundo tipo es el informe de deuda, que proporciona un documento en el que se declara si un determinado inmueble tiene alguna deuda en su haber, junto con su importe, o si, por el contrario, se encuentra libre de cargas. Por último, el tercer tipo de informe es el de cuentas anuales de la comunidad, que genera un documento en el que figuran de forma resumida y agrupados por conceptos todos los movimientos que se han realizado durante un determinado año en la comunidad, indicando el saldo restante (o negativo) en la cuenta de la misma.

    Además, el controlador posee un par de métodos adicionales que ayudan a crear estos informes. El primero de ellos se encarga de cambiar el formato de una fecha, y el segundo es el encargado de calcular la deuda total de un inmueble en base al importe de sus recibos y si están pagados o no.

11. **ReservationController**

    Con el controlador **ReservationController** se realiza la gestión de las reservas de zonas comunes. Se permite visualizarlas, crearlas y eliminarlas, llevando a cabo siempre que sea necesario las verificaciones correspondientes. El controlador también proporciona adicionalmente un método que permite obtener todas las reservas de un usuario en concreto. Para el buen funcionamiento del controlador se inyectan los servicios **ReservationService**, **UserService** y **CommonAreaService**.

12. **UserController**

    El controlador **UserController** gestiona todo lo respectivo a los usuarios de la aplicación, independientemente de su rol. Para ello, permite visualizarlos, crearlos, editarlos y eliminarlos. También se incorporan todas las comprobaciones necesarias para garantizar la integridad de

los datos en la aplicación. Cabe destacar que, en caso de eliminar un usuario, todos sus inmuebles serán asignados automáticamente al administrador de la comunidad para que sea él el que posteriormente los gestione a su gusto. En este controlador, han sido inyectados los servicios **UserService**, **PropertyService**, **ReservationService**, **CommunityService** e **IssueService**.

- **Servicios**

  Al igual que se ha hecho con los controladores, se ha creado un servicio dedicado a cada función de la aplicación, cada uno de los cuales se detalla a continuación.

  1. **CommonAreaService**

     Este servicio es el encargado de todas las operaciones relacionadas con las zonas comunes de la comunidad. Proporciona todas las funcionalidades necesarias para que **CommonAreaController** y **ReservationController** puedan realizar sus tareas, como pueden ser crear, visualizar, actualizar o eliminar las zonas comunes.

  2. **CommunityService**

     Este servicio ha sido diseñado para manejar todas las operaciones relacionadas con la comunidad. Permite a **CommunityController** y otros muchos controladores realizar correctamente sus tareas, ya sean crear, visualizar, editar o eliminar una comunidad.

  3. **CustomUserDetailsService**

     Este servicio es fundamental para realizar la autenticación de usuarios en la aplicación, ya que es utilizado por el filtro de seguridad **JwtAuthenticationFilter** dentro del contexto de Spring Security. Implementa la interfaz **UserDetailsService** proporcionada por el propio Spring Security, y su función principal consiste en cargar los detalles del usuario a partir de su DNI, verificando si existe en el repositorio y concediendo los roles y permisos necesarios para garantizar un correcto control de acceso en el sistema.

  4. **DocumentService**

     Este servicio es el encargado de todas las operaciones relacionadas con los documentos de la comunidad. Aporta toda la funcionalidad que puede necesitar **DocumentController** para llevar a cabo sus tareas, que pueden ser permitir la creación de nuevos documentos, visualizar los ya existentes, realizar modificaciones en alguno de ellos o directamente eliminarlos de la comunidad.

5. **IssueService**

   Este es el servicio encargado de manejar las operaciones correspondientes a las incidencias de la comunidad. Proporciona toda la funcionalidad que necesitan **IssueController** y **UserController** para gestionar las incidencias, pudiendo mostrarlas, crearlas, actualizarlas y eliminarlas.

6. **MovementService**

   Mediante este servicio se gestionan las operaciones respectivas a los movimientos de cuentas de la comunidad. Es utilizado por **MovementController** y por **ReportController**. Para ello, se proporciona funcionalidad que permite visualizarlos, editarlos, crearlos y eliminarlos. Además, se proporciona funcionalidad específica que permite buscar todos los movimientos de un año concreto, y también se permite buscar todos los movimientos que tengan un mismo concepto dado.

7. **PdfService**

   Este servicio se encarga de la generación de los documentos PDF para la aplicación. Es utilizado por **ReportController** para los documentos de los informes, y tiene funcionalidad dedicada a la generación de documentos PDF para los informes de recibo de inmuebles, los informes de deuda de inmuebles y los informes de cuentas anuales de la comunidad. En cada caso se generará un documento PDF totalmente distinto y orientado exclusivamente a esa finalidad. Se han creado algunos métodos privados en el servicio que permiten la reutilización de código más genérico, como puede ser la creación de celdas en las tablas que se muestren en los documentos, el filtrado de movimientos por concepto o la obtención de la fecha actual del sistema.

8. **PropertyService**

   Con este servicio se manejan las operaciones necesarias para los inmuebles de la comunidad. Éstos se pueden crear, visualizar, actualizar y eliminar, y el servicio es empleado por una variedad de controladores, entre ellos **PropertyController**. Como funcionalidad adicional se proporciona una herramienta que permite buscar un inmueble a partir de su dirección, considerando los distintos campos que la conforman.

9. **ReceiptService**

   Este es el servicio que gestiona las operaciones realizadas para los recibos de la comunidad. Se pueden crear nuevos recibos, modificar los existentes, visualizarlos o eliminarlos. **ReceiptController** y **MovementController** son los que hacen uso de este servicio para sus

tareas. Se incluye funcionalidad adicional que permite buscar un recibo por fecha o por pago atrasado.

10. **ReservationService**

    Mediante este servicio se controlan las operaciones necesarias para las reservas de zonas comunes de la comunidad. Las reservas se pueden mostrar, crear, editar y eliminar. Como extra, el servicio pone a disposición de los controladores **ReservationController**, **UserController** y **CommonAreaController** una funcionalidad que permite, a partir de una zona común, una fecha, una hora de inicio y una hora de finalización, buscar si ya existe alguna reserva que se solape, lo que puede ser útil para impedir la creación de una nueva reserva con esas condiciones.

11. **UserService**

    Este servicio maneja las operaciones correspondientes a los usuarios de la aplicación. Se pueden crear nuevos usuarios, actualizar su información, visualizarlos y eliminarlos. También se permite encontrar un usuario a partir de su DNI, que es su identificador principal. Este servicio es utilizado por muchos controladores, entre ellos **UserController**, ya que para realizar operaciones con otras entidades también es necesario en ocasiones el manejo de usuarios.

- **Repositorios**

  La interacción con la base de datos en la aplicación se realiza a través de repositorios, responsables de manejar las operaciones CRUD para cada entidad. Por ello, se cuenta con un repositorio por cada entidad de la aplicación. Estos son: **CommonAreaRepository**, **CommunityRepository**, **DocumentRepository**, **IssueRepository**, **MovementRepository**, **PropertyRepository**, **ReceiptRepository**, **ReservationRepository** y **UserRepository**.

- **Entidades y modelo**

  Las entidades representan los objetos fundamentales del dominio de la aplicación, es decir, los conceptos clave sobre los que se organiza y gestiona la información. Cada entidad se corresponde con una tabla en la base de datos y define sus atributos y relaciones con otras entidades. Estas clases forman el modelo de datos de la aplicación y permiten mapear de forma directa los objetos Java al esquema relacional, facilitando la persistencia y recuperación de datos. Las entidades que han sido definidas en Moai son: **CommonArea**, **Community**, **Document**, **Issue**, **Movement**, **Property**, **Receipt**, **Reservation** y **User**.

Además, el modelo también comprende una serie de clases denominadas Data Transfer Objects (DTO). Estas clases, incluidas dentro del paquete **moai.dto**, definen estructuras que facilitan la transferencia de datos entre el Backend y el Frontend de la aplicación. Las clases en cuestión son: **DocumentResponse**, **ErrorResponse**, **LoginRequest**, **LoginResponse**, **MovementRequest**, **ReceiptRequest** y **ReportRequest**.

- **Otras clases**

  En el proyecto se puede encontrar el paquete **moai.security**, que contiene tres clases destinadas a la seguridad de la aplicación. La primera de ellas es **JwtAuthenticationFilter**, encargada de autenticar a los usuarios y de filtrar los tokens recibidos. La segunda es **JwtTokenManager**, que proporciona todas las herramientas necesarias para manejar los tokens JWT, desde generar nuevos tokens hasta realizar su validación, pasando por el proceso de interpretarlos. Por último, se encuentra la clase **SecurityConfig**, que hace uso del mencionado **CustomUserDetailsService** y del **JwtAuthenticationFilter** para aplicar filtros de seguridad y determinar a qué sitios puede acceder un usuario en función de su rol y de si está autenticado en el sistema.

4.3.2. Frontend

Diseño

El diseño del lado Frontend del proyecto ha sido estructurado siguiendo estándares y buenas prácticas que, al igual que en el lado Backend, garantizan la mantenibilidad, la extensibilidad y la usabilidad del código.

- **Organización de archivos**

  Para el desarrollo del Frontend se creó un proyecto independiente del proyecto del Backend, ya que cada parte utiliza sus propias tecnologías y funcionaban como dos aplicaciones independientes. Estos dos proyectos fueron unificados posteriormente de cara a llevar la aplicación a producción, y para ello se compiló el proyecto Frontend y se introdujo en el otro proyecto, formando parte de las carpetas contenidas dentro de **src/main/resources/static**, que es donde los proyectos de Java almacenan los ficheros relativos a la interfaz de la aplicación, organizados en las carpetas **css** (estilos visuales), **img** (imágenes) y **js** (código JavaScript).

  Aparte de ello, el proyecto Frontend sigue su propia distribución de carpetas. En la raíz del proyecto se encuentran ciertos ficheros de configuración

utilizados por Vue.js. Dentro, existe una carpeta **nodemodules** que contiene los módulos utilizados por Node.js para permitir la ejecución del proyecto de forma local. También existe una carpeta **public** con el icono de la aplicación, **favicon.ico**, y el fichero principal de la interfaz, **index.html**. Sin embargo, la carpeta más importante es **src**, que contiene más ficheros de configuración de Vue.js y Axios, así como las carpetas **assets** (imágenes para la aplicación), **components** (vistas de la interfaz) y **store** (almacenamiento de información en ejecución).

- **Estructura de vistas**

  En Vue.js, las vistas representan los componentes visuales que forman la interfaz de usuario de la aplicación. Cada vista corresponde generalmente a una página o sección específica del Frontend y se asocia con una ruta definida en el sistema de enrutamiento. Estas vistas actúan como contenedores que integran y coordinan distintos componentes reutilizables, gestionan el estado visual y se comunican con el Backend a través de solicitudes HTTP. Gracias a esta estructura modular, se facilita tanto el desarrollo como el mantenimiento del código, promoviendo una organización clara y escalable de la interfaz.

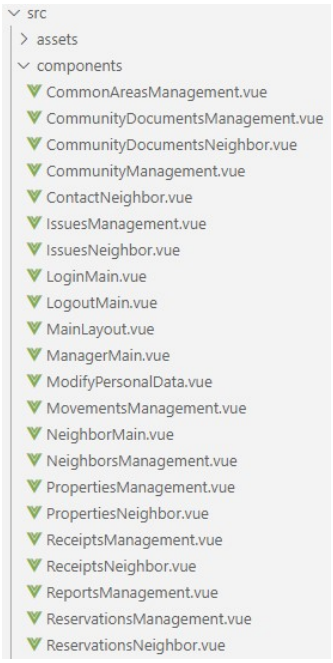


Figura 4.6: Vistas de Vue.js para la interfaz

Como se ha mencionado previamente, las vistas se encuentran ubicadas en **src/components**, y existen algunas vistas que son únicas y otras que tienen una versión para los vecinos y otra para el administrador, en base a la

funcionalidad que interese para cada rol. En la Figura 4.6 se pueden observar todos los ficheros que componen las vistas de la aplicación, anotados con la terminación Management aquellos destinados al administrador y con la terminación Neighbor los destinados a los vecinos.

Cabe destacar que cada vista esconde un gran trabajo en su interior, conformado por una parte HTML, otra parte CSS y una última parte JavaScript que es la más compleja de todas, puesto que consta de un código elaborado meticulosamente para satisfacer las necesidades que debe resolver cada vista en concreto. Este código puede llegar a ser aún más complejo que las líneas que componen el Backend, ya que solventa múltiples desafíos presentados a la hora de realizar verificaciones, estructurar la información, actualizar la página en tiempo real cuando se añaden nuevos elementos, filtrar las direcciones de los inmuebles cuando se va seleccionando cada parte de la dirección...

- **Componentes reutilizables**

  Para reducir la cantidad de código y mejorar la eficiencia de la aplicación se han intentado reutilizar aquellos componentes que lo permitían. Por ejemplo, en el fichero **App.vue**, que es donde se concentra la configuración principal de visualización, se han añadido todos los estilos CSS que son compartidos por múltiples vistas, ahorrando la necesidad de incluirlos individualmente en cada una de ellas. También se han reutilizado aquellas vistas que lo permitiesen tanto para administrador como para vecinos, como es el caso de **ModifyPersonalData.vue**.

Implementación

En este apartado se revisarán las vistas que componen el Frontend, que suman un total de 22 vistas. También se analizarán los diversos ficheros de configuración que ayudan al buen funcionamiento de la aplicación.

La vista **LoginMain.vue** (Figura 4.7) es la que presenta la primera página que se encuentra al acceder a la aplicación, que es la que permite iniciar sesión en el sistema. Se muestra una interfaz sencilla e intuitiva, donde el usuario puede introducir su DNI y contraseña para autenticarse en el sistema. Por otro lado, la vista **LogoutMain.vue** será la encargada de gestionar el cierre de sesión en el sistema al presionar el icono de color rojo situado en la parte superior derecha de la interfaz (Figura 4.8), el cual se mostrará en todas las vistas de la aplicación, a excepción de la pantalla de inicio de sesión. Además, existe una vista llamada **MainLayout.vue** que gestiona la visualización del resto de vistas, puesto que establece **LogoutMain.vue** para que siempre sea visible y determina que

el contenido específico de cada página será el que se contemple a través del enrutamiento de vistas con router-view, que hace uso del fichero de configuración **router.js**.

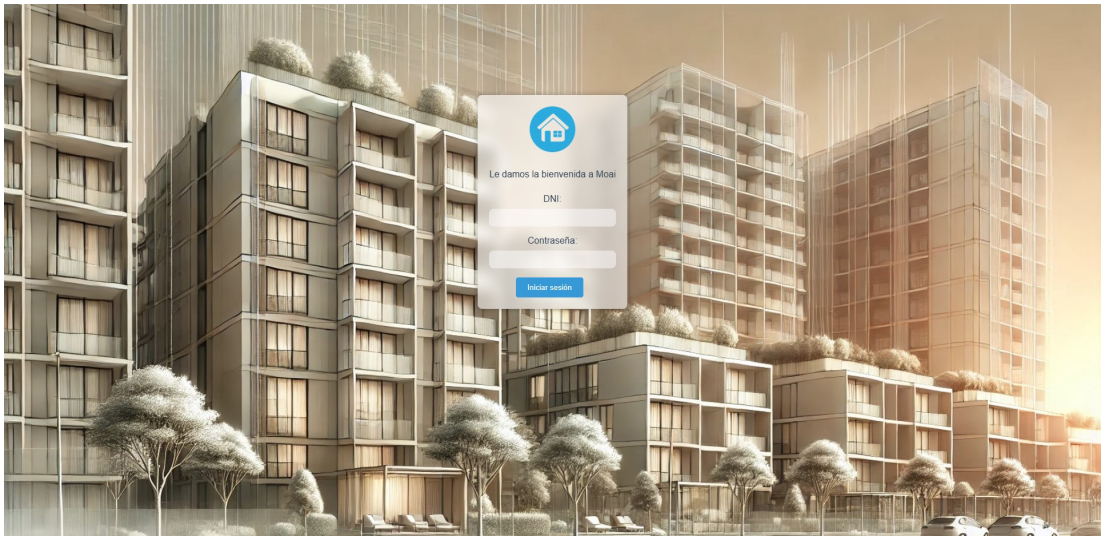


Figura 4.7: Vista LoginMain.vue

Tras iniciar sesión, el usuario se encontrará con la página principal de la aplicación. Esta página no será idéntica para todos los usuarios, ya que el administrador obtendrá una página generada con la vista **ManagerMain.vue** (Figura 4.8), mientras que los vecinos usarán la vista **NeighborMain.vue** (Figura 4.9), mostrando cada una de ellas las funciones específicas para cada tipo de usuario.

Figura 4.8: Vista ManagerMain.vue

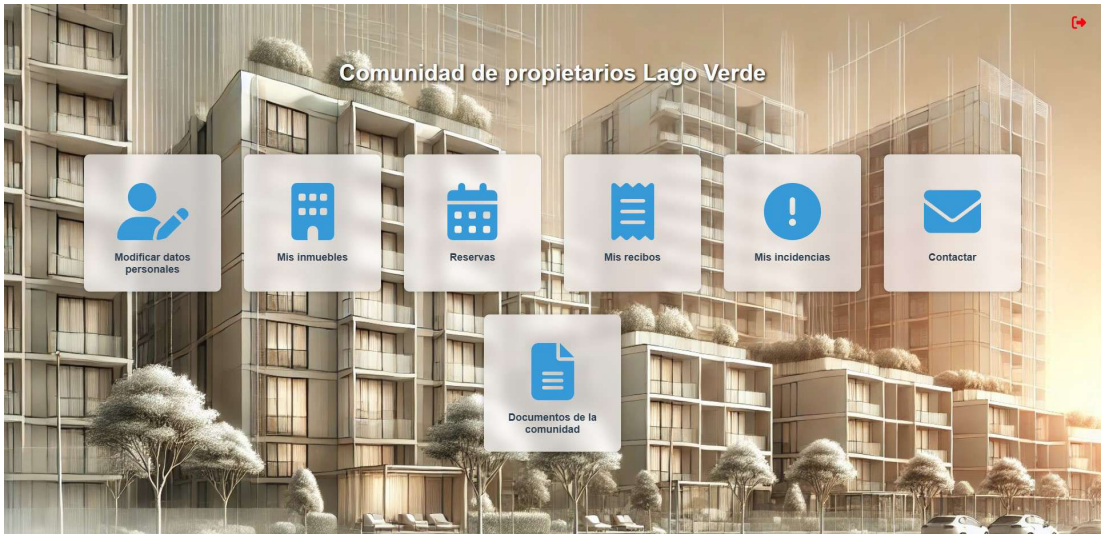


Figura 4.9: Vista NeighborMain.vue

Todos los usuarios, ya sean administradores o vecinos, tendrán acceso a una función llamada *Modificar datos personales*, generada con la vista común **ModifyPersonalData.vue** (Figura 4.10). Desde aquí, los usuarios podrán actualizar todos sus datos personales a excepción de los datos invariables, que son nombre, apellidos y DNI. La vista notificará al usuario en tiempo real si la información que está introduciendo no cumple las condiciones para ese campo.

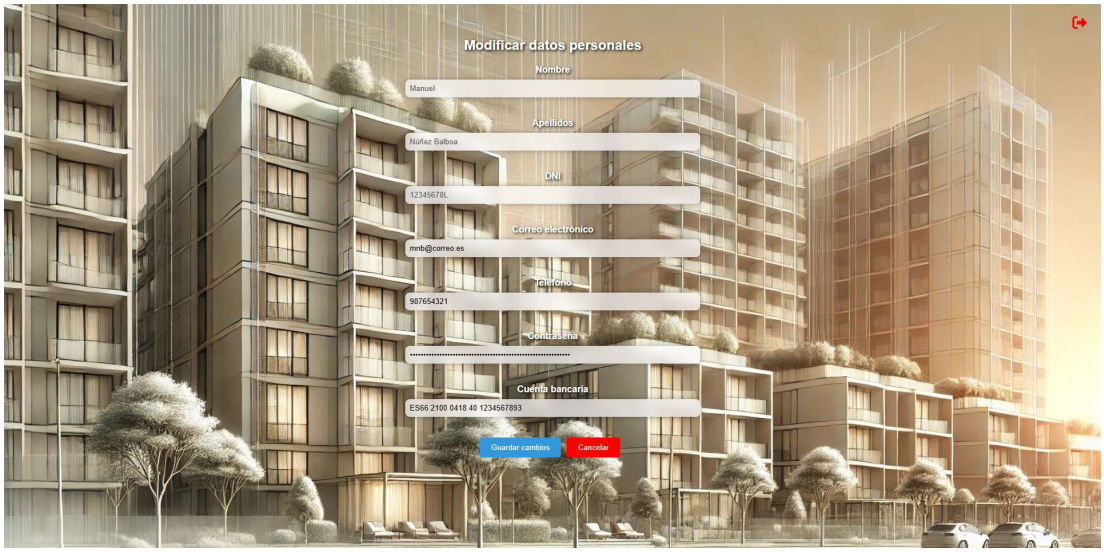


Figura 4.10: Vista ModifyPersonalData.vue

Otra función similar que encontrarán los usuarios es la relativa a los recibos. El administrador tendrá acceso a la página *Recibos* proporcionada por la vista **ReceiptsManagement.vue** (Figura 4.11), mientras que los vecinos podrán

acceder a la página *Mis recibos* a través de la vista **ReceiptsNeighbor.vue** (Figura 4.12). La página ofrecida al administrador le permite visualizar todos los recibos, además de poder crear nuevos y gestionar los ya existentes. Sin embargo, la funcionalidad para los vecinos estará limitada a visualizar los recibos correspondientes a los inmuebles de su propiedad, permitiéndoles llevar un control de los cobros requeridos por la comunidad.

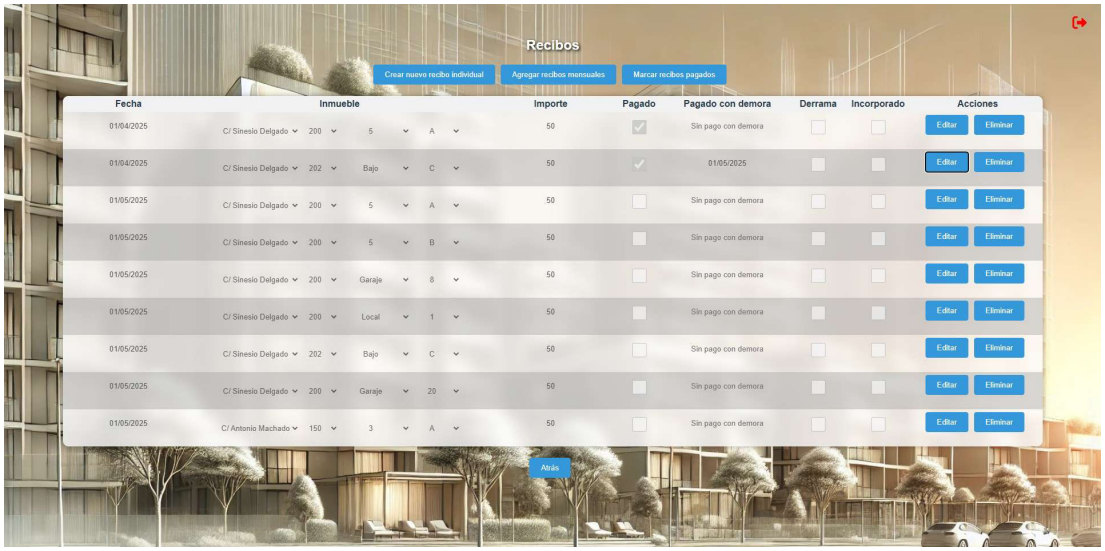


Figura 4.11: Vista ReceiptsManagement.vue

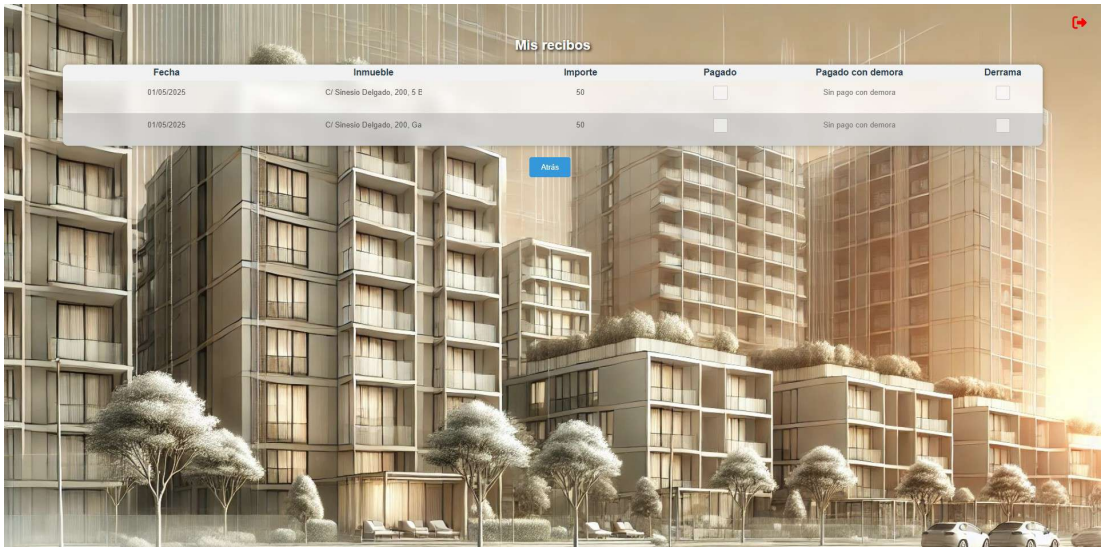


Figura 4.12: Vista ReceiptsNeighbor.vue

Para asegurar el buen funcionamiento del sistema y evitar errores, la aplicación incorpora numerosos modales para el usuario, que actúan en caso de tener que

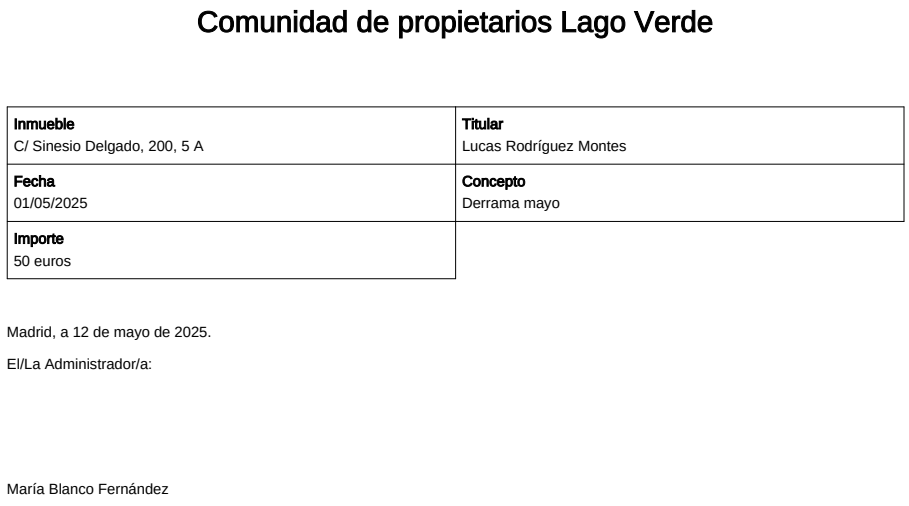
notificar un error en el sistema, un error en los datos introducidos, o para solicitar confirmación antes de realizar una acción importante, como puede ser eliminar un recibo (Figura 4.13). También se incluyen modales que permiten al usuario introducir los datos necesarios o visualizar cierta información (Figura 4.14).

Figura 4.13: Modal de confirmación para eliminar un recibo en la vista ReceiptsManagement.vue

Figura 4.14: Modal para crear un nuevo lote de recibos mensuales en la vista ReceiptsManagement.vue

Para la gestión de los movimientos producidos en la cuenta de la comunidad dentro de la función *Movimientos* del administrador, la vista **MovementsManagement.vue** será la encargada de ello. Otra función exclusiva del administrador es *Consultas e informes*, cuya página es generada por **ReportsManagement.vue** (Figura 4.15) y permite al usuario obtener un informe en formato PDF para la opción seleccionada (Figura 4.16).

Figura 4.15: Vista ReportsManagement.vue con la opción Informe de recibo de inmueble seleccionada



Comunidad de propietarios Lago Verde

Inmueble C/ Sinesio Delgado, 200, 5 A	**Titular** Lucas Rodríguez Montes
Fecha 01/05/2025	**Concepto** Derrama mayo
Importe 50 euros	

Madrid, a 12 de mayo de 2025.

El/La Administrador/a:

María Blanco Fernández

Figura 4.16: Documento generado mediante la opción Informe de recibo de inmueble

Las funciones *Gestión de vecinos*, *Gestión de inmuebles* y *Gestión de zonas comunes* son similares entre sí, pero cada una gestiona concretamente el elemento correspondiente de la comunidad. Están generadas por las vistas **NeighborsManagement.vue**, **PropertiesManagement.vue** y **CommonAreasManagement.vue** (Figura 4.17), respectivamente.

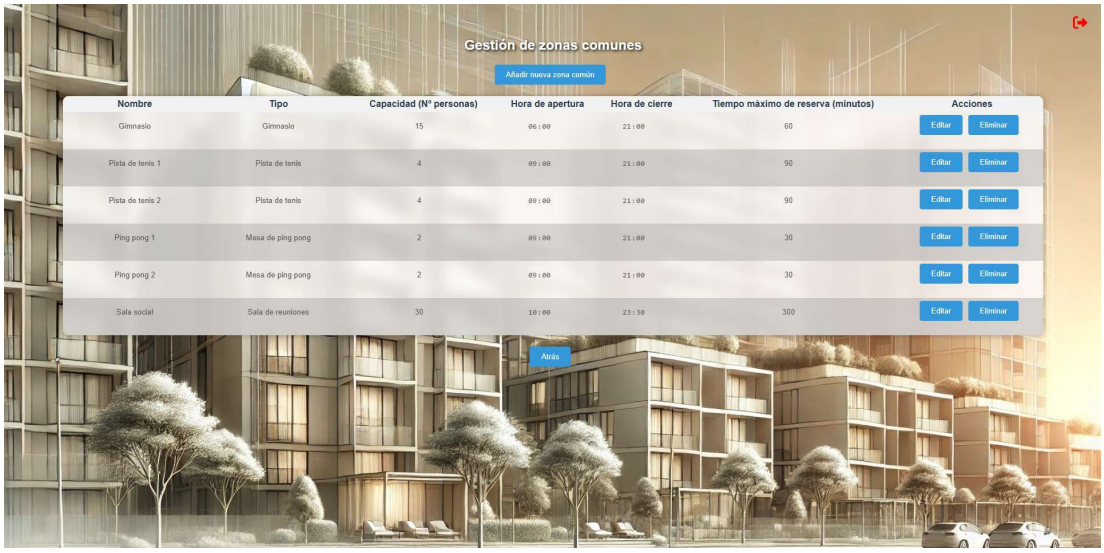


Figura 4.17: Vista CommonAreasManagement.vue

Otras funciones que son similares entre sí son *Gestión de reservas* e *Incidencias*, gestionadas por las vistas **ReservationsManagement.vue** e **IssuesManagement.vue**, respectivamente. Aquí el administrador solo puede visualizar y eliminar, ya que no tendría sentido la creación y la modificación de ninguno de estos dos elementos de la comunidad por su parte. Sin embargo, en *Incidencias* se permite hacer clic sobre una de ellas para mostrar información más detallada de la misma y así poder ver más cómodamente la descripción del caso (Figura 4.18).

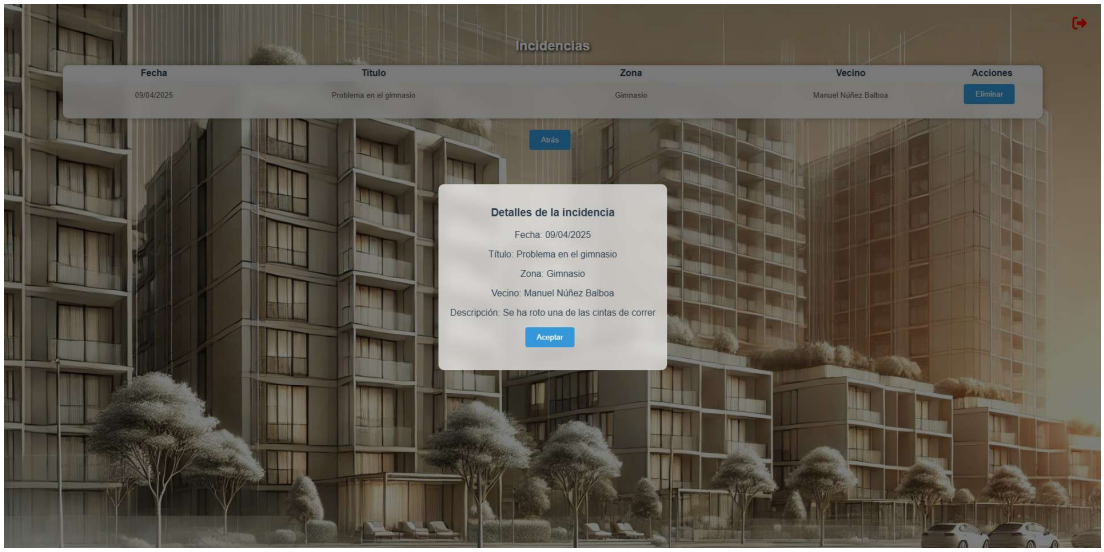


Figura 4.18: Vista IssuesManagement.vue con el modal mostrado al hacer clic en una incidencia

La vista **CommunityManagement.vue** es la encargada de generar la página correspondiente a la función *Gestión de la comunidad* (Figura 4.19). Es una página similar en formato a la de *Modificar datos personales*, pero en este caso permite que el administrador ajuste todos los detalles relativos a la propia comunidad de propietarios, incluida la imagen que se utiliza como fondo de la aplicación.

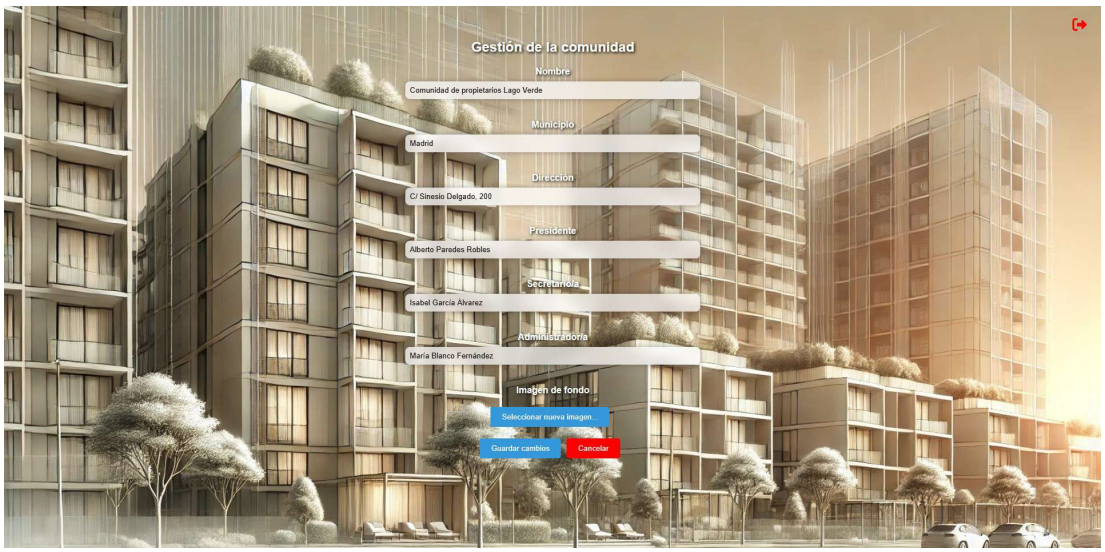


Figura 4.19: Vista CommunityManagement.vue

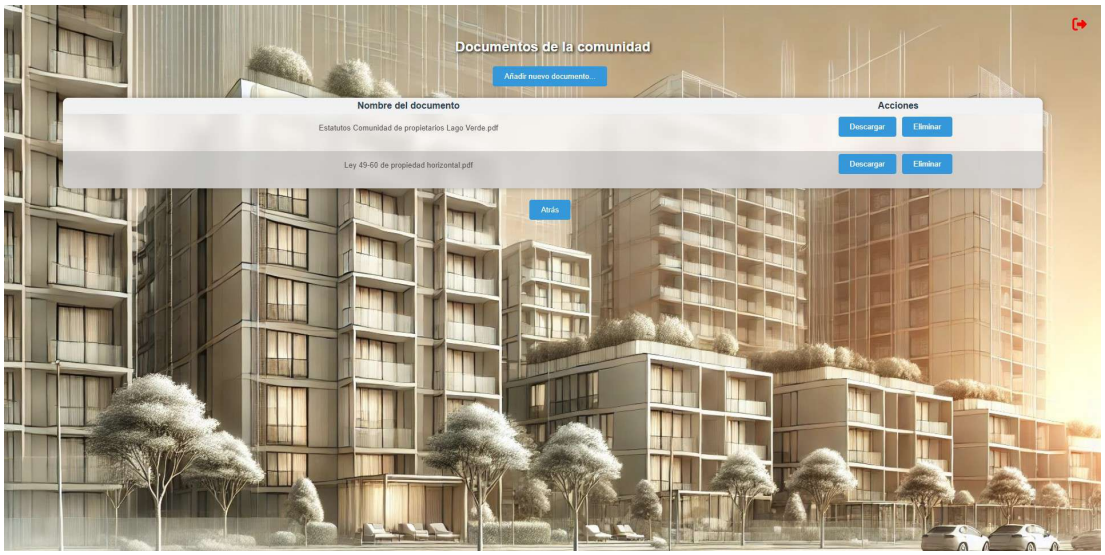


Figura 4.20: Vista CommunityDocumentsManagement.vue

La última función que encontrará el administrador en su panel es *Documentos de la comunidad*, de la cual se encarga la vista **CommunityDocumentsManagement.vue** (Figura 4.20). A través de ella, el administrador puede gestionar

todo lo relacionado con los documentos que se mostrarán a los vecinos, quienes pueden visualizarlos a través de una función homónima a ésta generada por la vista **CommunityDocumentsNeighbor.vue**, que restringe la funcionalidad y solo permite la visualización y descarga de los documentos.

En cuanto a los vecinos, en su página principal se encontrarán con la función *Mis inmuebles*, la cual, mediante la vista **PropertiesNeighbor.vue**, les permite visualizar todos los inmuebles en su propiedad con su información concreta. Otra función más completa que también encontrarán es *Reservas*, creada a partir de la vista **ReservationsNeighbor.vue** (Figura 4.21). Aquí podrán crear nuevas reservas de zonas comunes para un horario concreto, además de poder visualizar y eliminar sus propias reservas con el botón Mis reservas.

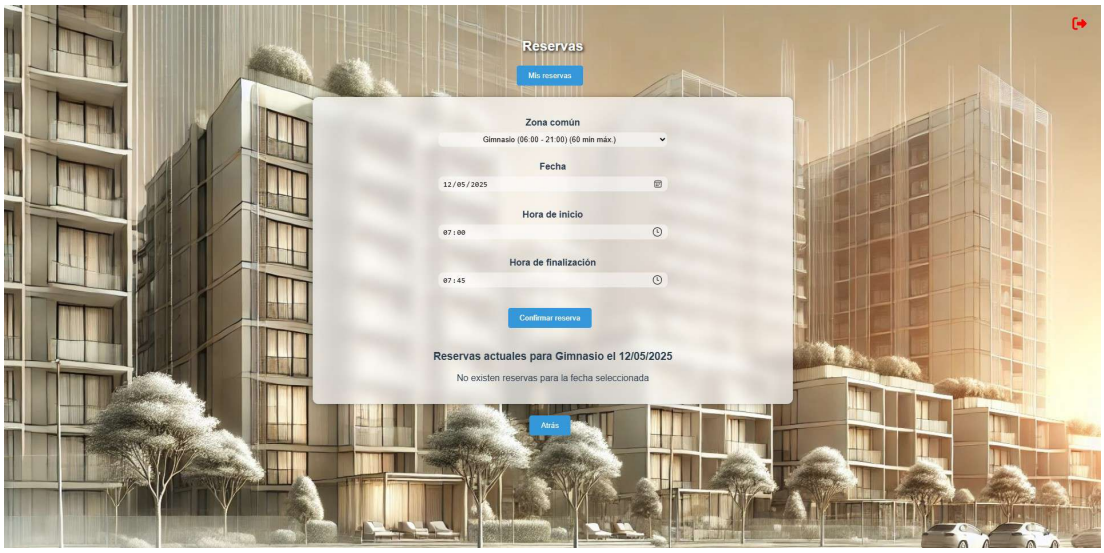


Figura 4.21: Vista ReservationsNeighbor.vue

Las últimas dos vistas de la aplicación son las que generan las funciones para los vecinos *Mis incidencias* y *Contactar*, que son **IssuesNeighbor.vue** y **ContactNeighbor.vue**, respectivamente. La primera de ellas permite a los vecinos ver y eliminar sus incidencias, así como crear unas nuevas (Figura 4.22), mientras que la segunda les permite ver datos de contacto de los responsables de la comunidad.

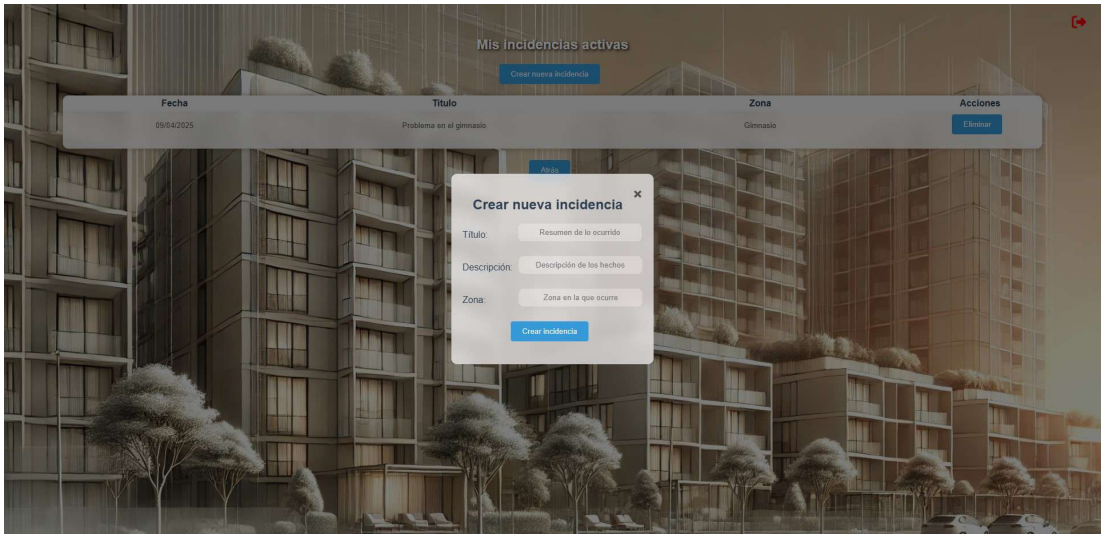


Figura 4.22: Vista IssuesNeighbor.vue con el modal de creación de nueva incidencia

Para que todas las vistas anteriores puedan funcionar correctamente son necesarios determinados ficheros de configuración que establecen los parámetros a seguir por el Frontend. Los más importantes son estos:

- **App.vue**

  Este se puede considerar el fichero principal de toda la aplicación Vue.js. En él se han determinado tareas como obtener y mostrar la imagen de fondo de la comunidad desde el principio o los estilos CSS comunes que serán utilizados en diversas vistas.

- **axios.js**

  Este fichero establece la configuración que necesita la biblioteca Axios para funcionar como se desea. Se configura la URL base que se utilizará para las peticiones REST o el manejo de los tokens para gestionar la autenticación de los usuarios, que deben ser incorporados en todas las peticiones para que el Backend pueda mantener una correcta seguridad en el sistema.

- **router.js**

  Aquí se configuran las URL a través de las cuales se accederá a cada una de las vistas del Frontend, partiendo de la URL base en la que se esté ejecutando la aplicación. Para ello, se utiliza el sistema de enrutamiento de Vue.js, llamado *vue-router*, que también se encarga de comprobar si el usuario está autenticado antes de permitirle acceder al contenido de la aplicación y, si

no es así, se le redirige automáticamente a la página de inicio de sesión, impidiendo accesos no autorizados al sistema.

- **main.js**

  Con este fichero se establecen los componentes que serán utilizados en la aplicación, como puede ser *store* para el almacenamiento de contenido durante la ejecución o *router*, que hace uso de la configuración establecida en **router.js**. También permite añadir bibliotecas a la aplicación, como es Font Awesome, de la que se pueden cargar específicamente los iconos que van a ser empleados.

- **index.js**

  Dentro de la carpeta *store* se encuentra este fichero que permite configurar el almacén que será utilizado durante la ejecución de la aplicación. Aquí se ha establecido un almacén con estados en el que se guarda el estado de autenticación del usuario, lo que permite comprobar rápidamente si se encuentra autorizado o no.

La implementación de las vistas ha sido la parte más compleja del desarrollo, ya que se ha tenido que elaborar código JavaScript específico para cada una de ellas que resuelva los desafíos encontrados. Se han elaborado algoritmos que filtran las opciones ofrecidas en los selectores según se va completando la dirección de un inmueble, limitando las direcciones posibles únicamente a direcciones existentes. Estos códigos han sido utilizados, por ejemplo, en **ReportsManagement.vue**.

También ha habido que resolver en la vista **ReservationsNeighbor.vue** los problemas presentados al cambiar el formato de fecha y hora entre el soportado por el Backend y un formato legible que cumpla los estándares utilizados en España. Además, había que permitir al código comprobar en tiempo real, sin llegar a interactuar con el Backend, si los valores establecidos para una reserva son válidos o no, es decir, si cumplen las condiciones de duración, la hora de apertura de la zona común, etc.

Otras vistas, que son **CommunityDocumentsManagement.vue**, **CommunityDocumentsNeighbor.vue** y **ReportsManagement.vue**, necesitan trabajar con documentos PDF e intercambiar dicho flujo de información con el Backend, lo que también ha supuesto una serie de conversiones para garantizar que los documentos llegan a ambas partes como se espera.

Existen códigos más comunes que se utilizan con diversas variantes en múltiples vistas, como los que permiten utilizar un mismo botón para Editar y Guardar un elemento de la comunidad, que puede ser un recibo, una zona común, un inmueble...

4.3.3. Base de datos

Diseño

El diseño de la base de datos de la aplicación sigue un enfoque relacional, en el que se emplea JPA (Java Persistence API) junto con Hibernate para mapear las tablas a entidades Java. Esta estrategia permite que la lógica del Backend interactúe con la base de datos de forma eficiente y estructurada. Gracias al uso de este mapeo objeto-relacional (ORM), no es necesario escribir consultas SQL manuales para las operaciones básicas CRUD (Crear, Leer, Actualizar y Eliminar), ya que éstas son gestionadas automáticamente por Spring Data JPA a través de los repositorios. A continuación, se presentan los aspectos clave de la implementación de la base de datos basada en entidades, que constituyen el núcleo del modelo de datos de la aplicación.

- **Mapeo de entidades**

  Cada entidad representa una tabla en la base de datos, y sus atributos se corresponden con las columnas de dicha tabla. Por ejemplo, la entidad **Property**, que representa una de las tablas principales del sistema como es **PROPERTY**, contiene atributos como **id**, **owner**, **type** o **lane**, los cuales se mapean directamente a las columnas de la base de datos mediante anotaciones como **@Id** o **@JoinColumn**.

- **Relaciones entre entidades**

  El sistema define distintos tipos de relaciones entre entidades que reflejan la estructura relacional de la base de datos. Por ejemplo, una entidad **Property** pertenece a una única **Community** mediante una relación **@ManyToOne**, y una **Community** puede tener varias **Properties**, lo cual se modela con una relación **@OneToMany** empleando una lista.

- **Generación automática de claves primarias**

  En la entidad **Property**, el campo **id** actúa como identificador principal y está configurado con las anotaciones **@Id** y **@GeneratedValue(strategy = GenerationType.AUTO)**. Gracias a esta configuración, la base de datos se encarga de asignar automáticamente un valor único a cada nuevo

registro, evitando la necesidad de controlar manualmente las claves primarias.

- **Gestión de mapeos**

  Algunos atributos de las entidades requieren configuraciones específicas para garantizar la compatibilidad con la base de datos. Por ejemplo, en la entidad **Reservation**, los campos **startTime** y **endTime** se mapean a columnas de tipo **TIME**, y en Java se gestionan con el tipo **java.time.LocalTime**, lo que permite una correcta interacción con el motor SQL.

  Por otro lado, cuando es necesario almacenar datos más grandes, se emplea la anotación **@Lob** (Large Object). Un caso concreto se da en la entidad **Document**, donde el atributo **data** de tipo **byte**[] está anotado con **@Lob** y vinculado a una columna de tipo **LONGBLOB**. Esta configuración permite almacenar los flujos de bytes que componen los documentos, ya sean imágenes o PDF.

  Se ha decidido almacenar en la base de datos estos dos tipos de documentos, que son los únicos presentes en la aplicación, debido a que la cantidad de ellos que estarán presentes en una comunidad (una única imagen y algún posible PDF) es demasiado reducida como para que compense almacenarlos en otro tipo de almacenamiento separado, como puede ser Amazon S3, ya que implicaría un mayor coste de desarrollo y utilización para el sistema que no se vería debidamente aprovechado.

Implementación

La aplicación organiza sus componentes principales mediante clases interrelacionadas que representan la estructura lógica del sistema y la asignación de responsabilidades. En la Figura 4.23 se presenta un diagrama entidad-relación que ilustra las relaciones existentes entre las entidades definidas en Moai, las cuales se detallan a continuación.

La clase **Community** representa a la comunidad de propietarios y es la clase principal de la aplicación, estando conectada con entidades fundamentales como **User**, **Property** o **CommonArea**. La clase **User**, que representa a los usuarios de la comunidad, está a su vez asociada con **Property**, ya que los usuarios tienen uno o varios inmuebles en su propiedad. Por su parte, **Property** está conectada con **Receipt**, reflejando los recibos que se crean para cada inmueble.

Cada **CommonArea**, además de a **Community**, está asociada a una o varias **Reservation**, indicando que la comunidad posee diversas zonas comunes y que cada una de ellas puede tener distintas reservas creadas. Estas reservas, además, están asociadas a un **User** que las ha creado. Las incidencias, representadas por

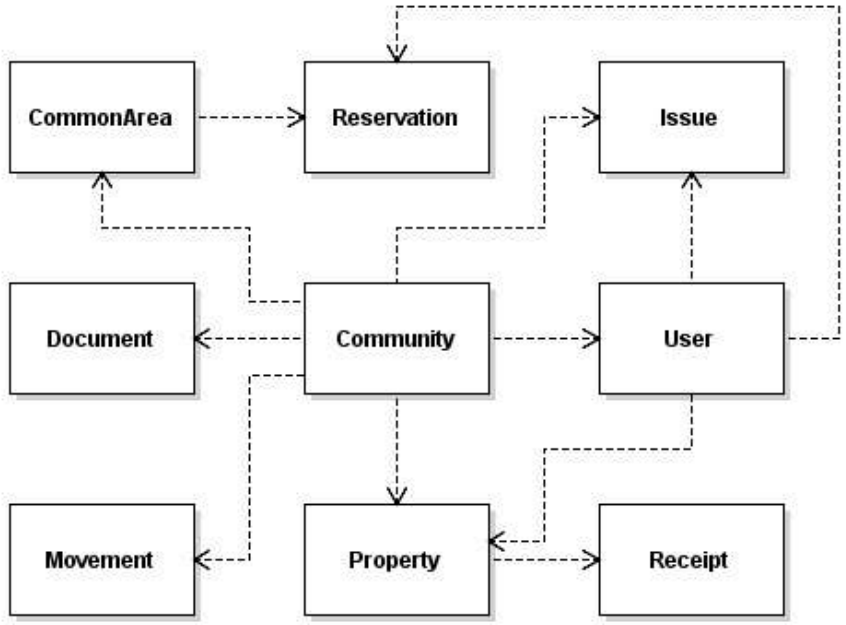


Figura 4.23: Diagrama de Entidad - Relación de Moai

la entidad **Issue**, también están conectadas al **User** que las ha creado, además de unirse todas a la **Community**.

Las entidades **Document** y **Movement** solo están relacionadas con **Community**, ya que es la comunidad la que agrupa todos los documentos subidos y todos los movimientos creados en su cuenta.

Mediante esta estructura se consiguen asignar de forma clara las responsabilidades que debe asumir cada entidad de la aplicación.

Una complicación que surgió durante el desarrollo de la aplicación fue la relacionada con la serialización de entidades en las respuestas JSON. Algunas relaciones entre entidades en Moai son bidireccionales (por ejemplo, una propiedad pertenece a una comunidad, y esta a su vez contiene un listado de propiedades), lo que puede provocar bucles infinitos al serializar los datos en formato JSON, generando errores en la API. Para evitar estos problemas, se utilizó la anotación **@JsonIgnoreProperties** en aquellas propiedades de las entidades que no debían ser serializadas en las respuestas JSON, impidiendo así la serialización circular.

4.4. Pruebas

En el desarrollo de software, las pruebas desempeñan un papel fundamental para asegurar la calidad, la estabilidad y el buen funcionamiento del sistema. En

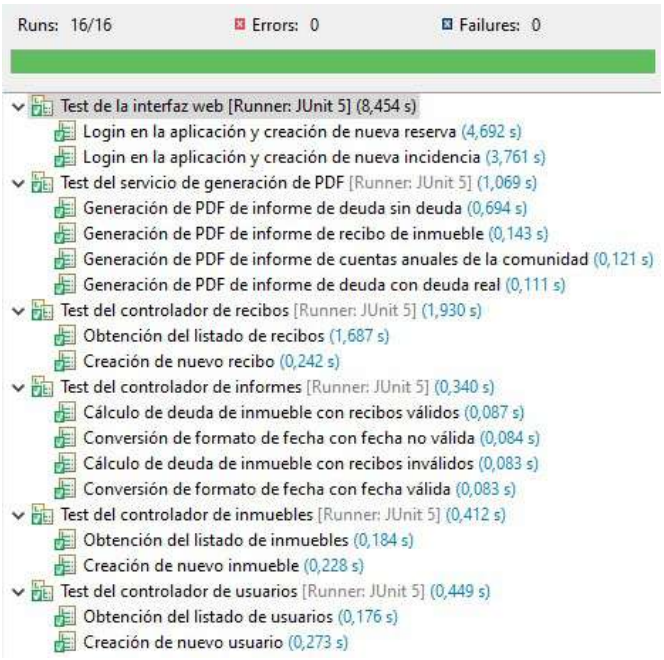


Figura 4.24: Pruebas unitarias y de sistema de Moai

el caso de Moai, se han implementado pruebas unitarias y pruebas de sistema. La Figura 4.24 muestra un resumen de las pruebas realizadas, que se detallan a continuación.

4.4.1. Pruebas unitarias

Las pruebas unitarias implementadas en este proyecto permiten validar el comportamiento de distintos métodos y clases, con el fin de garantizar que su lógica sea correcta. Estas pruebas se centran principalmente en los servicios, los controladores, las entidades y los componentes modales de la aplicación.

Para su ejecución se ha utilizado JUnit 5, un framework robusto y flexible que hace más sencilla la creación de las pruebas, así como su posterior ejecución. Por su parte, Spring Test proporciona las herramientas necesarias para que el contexto de Spring pueda ser integrado en las pruebas, garantizando el correcto funcionamiento de aquellos componentes que dependen del framework.

Para el proyecto se han realizado un total de 14 pruebas unitarias que abarcan las secciones más críticas de la aplicación, incluyendo tanto servicios como controladores. Debido a que el proyecto ha sido desarrollado únicamente por mí, se ha optado por probar manualmente otras secciones menos críticas de la aplicación, ya que los cambios han sido introducidos de manera controlada y escalada en el sistema y realizando pruebas inmediatamente después de incorporarlos, reduciendo al mínimo la posibilidad de dañar otras partes de la aplicación. Los ejemplos

incluyen:

- **Pruebas de integración de controladores**

  Para verificar el correcto funcionamiento de los endpoints se ha utilizado **RestAssured**, una herramienta que permite la realización de peticiones HTTP simuladas. También son validadas las respuestas recibidas, así como el comportamiento esperado del controlador. Por ejemplo, con el test **testPostProperties** situado en la clase **PropertyControllerTest.java** (Código 4.4) se realiza una comprobación de acceso al endpoint **/api/property**, que crea un nuevo inmueble en la comunidad, y también se verifica su integración con los servicios relacionados.

  Código 4.4: Sección Given - When - Then del test unitario testPostProperties

```
given()
    .port(port) // Se usa el puerto aleatorio asignado
    .contentType("application/json")
    .header("Authorization", "Bearer " + token) // Se introduce el token de
        autenticacion en la cabecera
    .body(newPropertyJson)
.when()
    .post("/api/property/")
.then() // Se verifica que el codigo de la respuesta y los datos devueltos
    coinciden con los esperados
    .statusCode(201)
    .body("type", equalTo(property3.getType().toString()))
    .body("lane", equalTo(property3.getLane()))
    .body("number", equalTo(property3.getNumber()))
    .body("floor", equalTo(property3.getFloor()))
    .body("letter", equalTo(property3.getLetter()))
    .body("storageRoom", equalTo(property3.getStorageRoom()))
    .body("ownershipShare", equalTo(property3.getOwnershipShare()));
```

  En este test, se crea un entorno simulado que incluye objetos como **Property** o **User**, necesarios para la creación de un nuevo inmueble. Se elabora un JSON que incluye todos los elementos esperados por el endpoint para poder atender a la petición, rellenando la información necesaria con los objetos creados en la simulación. Previamente a la ejecución del test se genera un token para poder autenticarse en la aplicación, aprovechando la situación para probar también esta parte del código. Finalmente, tras enviar el JSON simulado junto al token, se comprueba que la respuesta del endpoint es la esperada y que todo funciona correctamente.

- **Pruebas de integración de servicios**

  Las pruebas de integración de servicios se centran en validar la lógica de negocio implementada en **PdfService**, que es el servicio más complejo de toda la aplicación y el que más funcionalidad propensa a posibles fallos posee.

  La clase que recoge todos estos tests es **PdfServiceTest.java**, donde se realizan diversos tipos de pruebas para validar toda la funcionalidad provista

por el servicio. Un ejemplo es **testGenerateDebtReportPdf_WithoutDebt** (Código 4.5), donde se evalúa el método **generateDebtReportPdf**, encargado de la generación de los documentos PDF de informe de deuda de un inmueble, en este caso probando con un inmueble que no tiene deudas.

Código 4.5: Test unitario testGenerateDebtReportPdf_WithoutDebt para el servicio PdfService

```
1  @Test
2  @DisplayName("Generacion de PDF de informe de deuda sin deuda")
3  public void testGenerateDebtReportPdf_WithoutDebt() {
4
5      // Declaracion de variables de prueba
6      float debt = 0;
7
8      try {
9          // Se llama al metodo correspondiente del servicio de generacion de PDF
10         byte[] pdfBytes = pdfService.generateDebtReportPdf(community, property,
               owner, debt);
11
12         // Verificaciones basicas
13         assertNotNull(pdfBytes, "El PDF generado no debe ser nulo");
14         assertTrue(pdfBytes.length > 0, "El PDF generado debe contener datos");
15
16         // Se verifica si el PDF tiene contenido valido
17         try (PDDocument document = PDDocument.load(pdfBytes)) {
18             assertTrue(document.getPages().getCount() == 1, "El PDF debe tener
                   una pagina");
19
20             // Se extrae el texto del PDF
21             PDFTextStripper textStripper = new PDFTextStripper();
22             String pdfText = textStripper.getText(document);
23
24             // Se verifica que el texto contiene el mensaje de no deuda
25             String normalizedPdfText = pdfText.replace("\r\n", "\n");
26             String expectedAddress = property.getLane() + ", " +
                   property.getNumber() + ", " + property.getFloor() + " " +
                   property.getLetter();
27             String expectedNoDebtMsg = "Que el inmueble sito en " +
                   expectedAddress + " se encuentra al corriente de pago de las
                   \ncuotas de la comunidad a la fecha de este certificado.";
28             assertTrue(normalizedPdfText.contains(expectedNoDebtMsg), "El PDF
                   deberia contener el mensaje de no deuda: " + expectedNoDebtMsg);
29         }
30     } catch (Exception e) {
31         fail("La generacion del PDF fallo con excepcion: " + e.getMessage());
32     }
33 }
```

- **Pruebas unitarias de lógica**

  Estas pruebas verifican el correcto funcionamiento de los algoritmos que conforman la lógica de negocio de la aplicación. En la clase **ReportControllerTest.java** se pueden encontrar distintas pruebas que garantizan que los algoritmos presentes en **ReportController** funcionan como se espera. Una de estas pruebas es **testCalculateDebt_InvalidReceiptFormat** (Código 4.6), que verifica el funcionamiento del método **calculateDebt** cuando se le pasa un recibo con un formato no válido. Este método es el que se utiliza para calcular la deuda de un inmueble cuando se crea un informe de deuda.

  Código 4.6: Test unitario testCalculateDebt_InvalidReceiptFormat para el controlador ReportController

```
1  @Test
2  @DisplayName("Calculo de deuda de inmueble con recibos invalidos")
3  public void testCalculateDebt_InvalidReceiptFormat() {
4
5      // Preparacion de datos simulados
6      List<Receipt> receipts = new ArrayList<>();
7
8      Receipt invalidReceipt = new Receipt(this.property, false, "ABC123", "",
           "", false, false); // Formato invalido
9
10     receipts.add(invalidReceipt);
11     property.setReceipts(receipts);
12
13     // Se ejecuta el metodo y se verifica la excepcion
14     assertThrows(IllegalArgumentException.class, () -> {
15         ReportController.calculateDebt(property);
16     }, "Se debe lanzar IllegalArgumentException si el formato es invalido");
17 }
```

4.4.2. Pruebas de sistema (E2E)

En este proyecto, se han desarrollado pruebas de sistema para verificar que los distintos componentes de la aplicación funcionen correctamente cuando se integran entre sí. Estas pruebas aseguran que el sistema completo responda adecuadamente a los escenarios previstos por el usuario.

Para ello se ha empleado Selenium WebDriver junto con JUnit 5, permitiendo simular interacciones reales en el navegador y validar el comportamiento de la interfaz de usuario. JUnit 5 coordina la ejecución de las pruebas y Selenium automatiza las acciones sobre la interfaz, facilitando la detección de errores que no podrían detectarse con pruebas unitarias.

La preparación del entorno antes de cada prueba automatizada es fundamental para garantizar una correcta ejecución, y se realiza mediante el método **setUp** (Código 4.7), encargado de inicializar el contexto necesario antes de iniciar las pruebas.

Código 4.7: Método setUp de la clase WebTest.java

```
1  @BeforeEach
2  public void setUp() {
3      String driverPath = Paths.get("drivers",
           "msedgedriver.exe").toAbsolutePath().toString(); // Se obtiene la ruta
           relativa del driver para Edge
4      System.setProperty("webdriver.edge.driver", driverPath);
5
6      EdgeOptions options = new EdgeOptions(); // Se configuran opciones para
           iniciar el navegador con ellas
7      options.addArguments("--start-maximized"); // Se inicia el navegador
           maximizado
8      options.addArguments("--remote-allow-origins=*"); // Se permite la conexion
           de origenes remotos
9
10     driver = new EdgeDriver(options); // Se setean las opciones para el
           navegador
11
12     wait = new WebDriverWait(driver, Duration.ofSeconds(5)); // Se crea un
           objeto WebDriverWait que permitira realizar esperas durante la
           ejecucion de los tests
13 }
```

También existe un método **setUpAll** (Código 4.8) que se ejecuta una única vez antes de realizar todas las pruebas contenidas en la clase **WebTest.java**, que es donde se reúnen todas las pruebas de la interfaz web. Este método realiza las configuraciones que sean comunes a todos los tests para ahorrar código repetido en cada uno de ellos, como puede ser la creación de algún usuario o zona común para la comunidad.

Código 4.8: Método setUpAll de la clase WebTest.java

```
1  @BeforeAll
2  public void setUpAll() {
3      this.community1 = new Community("Comunidad de Propietarios Dunas Arenosas",
           "Jose Manuel Perez Garcia", "Maria Gomez Lopez", "Pedro Garcia Lopez",
           "Madrid", "Paseo de la Castellana, 10");
4      communityService.save(this.community1);
5
6      this.dni = "12345678M";
7      this.password = "Contrasena";
8
9      this.user = new User("Luis", "Ramirez Lopez", "luis.rl@gmail.com",
           "123456789", dni, password, community1, "ES66 2100 0418 40 1234567892",
           Role.NEIGHBOR);
10     userService.save(this.user);
11
12     this.commonArea = new CommonArea("Pista de tenis 1", "Pista de tenis", 4,
           "09:00", "21:00", 120, community1);
13     commonAreaService.save(commonArea);
14 }
```

Dentro de **WebTest.java**, las pruebas de sistema que se encuentran son las siguientes:

- **testLoginAndCreateIssue**

  Este test verifica la funcionalidad de inicio de sesión en el sistema con unas

credenciales válidas de usuario tipo vecino. Tras esto, se accede a la sección de Mis incidencias y se procede a crear una nueva incidencia introduciendo datos válidos en el modal destinado a ello (Código 4.9). Una vez se ha comprobado que la incidencia se ha creado correctamente, ésta es eliminada con su botón correspondiente. También se realiza una verificación para garantizar que la incidencia ha podido ser eliminada, tras lo cual se procede a volver a la página principal y a cerrar sesión en el sistema, comprobando que se redirige automáticamente a la página de inicio de sesión. De esta forma, el test abarca un completo escenario que podría ocurrir realmente en producción.

Código 4.9: Fragmento de creación de nueva incidencia en el test testLoginAndCreateIssue de WebTest.java

```
1  // Se hace clic en la tarjeta de Mis incidencias y se comprueba que se ha
       accedido
2  driver.findElement(By.cssSelector(".card:nth-child(5)")).click();
3  wait.until(ExpectedConditions.titleContains("Mis incidencias"));
4
5  // Se crea una nueva incidencia y se comprueba que se ha creado esperando a la
       aparicion del boton Eliminar
6  driver.findElement(By.xpath("//button[text()='Crear nueva
       incidencia']")).click();
7  driver.findElement(By.id("title")).click();
8  driver.findElement(By.id("title")).sendKeys("Gotera");
9  driver.findElement(By.id("description")).click();
10 driver.findElement(By.id("description")).sendKeys("Hay una gotera en el techo");
11 driver.findElement(By.id("area")).click();
12 driver.findElement(By.id("area")).sendKeys("Portal 2");
13 driver.findElement(By.xpath("//button[text()='Crear incidencia']")).click();
14 wait.until(ExpectedConditions.presenceOfElementLocated(
15     By.xpath("//button[text()='Eliminar']")));
```

- **testLoginAndCreateReservation**

  Este test sigue los pasos del anterior, pero en lugar de crear una nueva incidencia se crea una nueva reserva, también con un usuario vecino. Tras iniciar sesión, se introducen datos válidos en el modal para la creación de reservas y se comprueba que ha sido creada correctamente (Código 4.10). Finalmente, se cierra la sesión en el sistema y se comprueba la redirección a la página de inicio de sesión, poniendo a prueba otro caso que podría darse frecuentemente en producción.

Código 4.10: Fragmento de creación de nueva reserva en el test testLoginAndCreateReservation de WebTest.java

```
1  // Se hace clic en la tarjeta de Reservas y se comprueba que se ha accedido
2  driver.findElement(By.cssSelector(".card:nth-child(3)")).click();
3  wait.until(ExpectedConditions.titleContains("Reservas"));
4
5  // Se crea una nueva reserva y se comprueba que se ha creado esperando a la
       aparicion de la etiqueta que lo confirma
6  driver.findElement(By.id("date")).click();
7  driver.findElement(By.id("date")).sendKeys("01012030");
8  driver.findElement(By.id("startTime")).click();
9  driver.findElement(By.id("startTime")).sendKeys("10:00");
10 driver.findElement(By.id("endTime")).click();
11 driver.findElement(By.id("endTime")).sendKeys("11:00");
12 driver.findElement(By.id("confirmReservation")).click();
13 wait.until(ExpectedConditions.textToBe(By.id("successMessage"), "Reserva creada
       con exito"));
```

4.4.3. Cobertura

La cobertura de pruebas es una métrica clave que indica qué porcentaje del código ha sido ejecutado durante la realización de las pruebas. Esta información permite evaluar el alcance de las pruebas implementadas y comprobar si el comportamiento del sistema ha sido verificado de forma adecuada en las distintas partes del código fuente.

El proyecto cuenta con un total de 16 tests que cubren el 40 por ciento de la lógica de la aplicación (Figura 4.25). Para obtener estas métricas se ha utilizado la herramienta JaCoCo (Java Code Coverage), que genera informes detallados de cobertura de código. Con JaCoCo se analiza el código ejecutado durante las pruebas e informa de qué líneas, ramas y métodos han sido cubiertos. Con estos informes se facilita la tarea de identificar las áreas del código que no se han probado adecuadamente.

Como se puede comprobar en la mencionada Figura 4.25, los mayores esfuerzos de las pruebas se han dedicado a la verificación del buen funcionamiento de los servicios, de la seguridad de la aplicación y de las entidades que conforman el modelo. Son estos tres paquetes donde se sitúa la mayor funcionalidad crítica de la aplicación. Los DTO (Data Transfer Objects) han sido revisados en menor medida, ya que son bastante simples y poco propensos a errores. Por último, se comprueba que dentro de los controladores solo han sido probados los más críticos, complementando dichas pruebas con verificaciones manuales realizadas por mí.

Element	Missed Instructions	Cov.
moai		37 %
moai.controller		16 %
moai.dto		25 %
moai.model		86 %
moai.security		83 %
moai.service		76 %
Total	5.497 of 9.222	40 %

Figura 4.25: Cobertura de los tests de Moai

4.5. Distribución y despliegue

En esta sección se explicará cómo se ha llevado a cabo el empaquetado de la aplicación en Docker mediante GitHub Actions y el posterior despliegue de la misma en Microsoft Azure.

4.5.1. Docker

Docker es una plataforma de código abierto que permite automatizar el despliegue de aplicaciones dentro de contenedores, los cuales proporcionan un entorno ligero y aislado que incluye tanto la aplicación como todas sus dependencias. Esto asegura que el software se ejecute de forma consistente en distintos entornos, sin importar las configuraciones específicas del sistema anfitrión.

En este proyecto, Docker se emplea para generar una imagen que encapsula toda la aplicación, facilitando así su distribución y despliegue. La imagen generada se publica en el repositorio de Docker Hub **migueld7/moai** e incluye tanto el Backend como el Frontend, junto con todos los componentes necesarios para su correcto funcionamiento.

Nombre		Tipo	Ubicación	Grupo de recursos
moai-mysql	···	Servidor flexible de Azure Database for MySQL	West Europe	moai-group
moai	···	Instancias de contenedor	West Europe	moai-group
moai-group	···	Grupo de recursos	West Europe	moai-group
GitHubUser	···	Identidad administrada	West Europe	moai-group

Figura 4.26: Recursos creados para el despliegue de Moai en Microsoft Azure

4.5.2. Microsoft Azure

Azure es la plataforma en la nube de Microsoft que proporciona una variedad de servicios destinados al desarrollo, despliegue y gestión de aplicaciones. Su principal ventaja radica en su flexibilidad, escalabilidad y facilidad de integración con herramientas de automatización y DevOps.

En este proyecto, Azure se utiliza como infraestructura para el despliegue de la aplicación, ofreciendo un entorno estable y económico para aplicaciones web. Como paso inicial, se llevó a cabo la configuración básica del entorno en la nube, que incluyó la creación de un grupo de recursos y una identidad administrada, tal como se ilustra en la Figura 4.26.

Grupo de recursos e Identidad administrada

En Azure, un grupo de recursos actúa como contenedor lógico para reunir todos los elementos relacionados con un proyecto. En este caso, se creó un grupo de recursos llamado **moai-group**, que incluye todos los componentes necesarios para la ejecución de la aplicación, como los contenedores, la base de datos y otros servicios implicados. Esta organización permite gestionar los recursos de forma centralizada y simplifica tanto su supervisión como su mantenimiento.

Además, se configuró una identidad administrada, que permite establecer autenticación entre servicios sin necesidad de almacenar o manejar credenciales manualmente. Gracias a esta identidad, la aplicación desplegada en contenedores puede comunicarse de forma segura con la base de datos u otros servicios necesarios, sin exponer información sensible en el proceso.

Instancias de contenedor de Azure

Instancias de contenedor de Azure es un servicio que permite ejecutar contenedores Docker directamente en la nube sin necesidad de gestionar infraestructura subyacente, como máquinas virtuales o clústeres de Kubernetes. Esta solución reduce significativamente la complejidad del despliegue y permite ejecutar aplicaciones basadas en contenedores con mayor rapidez y eficiencia.

En este proyecto, Instancias de contenedor de Azure se utiliza para desplegar

la aplicación contenida en una imagen Docker almacenada en Docker Hub. Durante el despliegue, se definen variables de entorno necesarias para la integración con otros servicios del sistema, como es la base de datos.

Servidor flexible de Azure Database for MySQL

Servidor flexible de Azure Database for MySQL es un servicio de base de datos relacional totalmente gestionado que ofrece una solución segura, escalable y eficiente para entornos que requieren MySQL. En este proyecto, se emplea para almacenar la información persistente de la aplicación, incluyendo usuarios, inmuebles y recibos.

El servidor MySQL se crea dentro del mismo grupo de recursos que el resto de componentes, lo que facilita la administración conjunta del entorno. Además, la autenticación entre la aplicación en contenedores y la base de datos se realiza mediante la identidad administrada configurada previamente, lo que elimina la necesidad de gestionar credenciales manuales y mejora la seguridad de la infraestructura.

4.5.3. GitHub Actions

Para el despliegue y distribución del proyecto, se ha adoptado una metodología ágil basada en un flujo de trabajo Git Flow simplificado. Este enfoque permite organizar el desarrollo de forma estructurada mediante el uso de distintas ramas: la rama master representa el estado de producción, mientras que develop actúa como rama de integración para desarrollo y pruebas.

Cada nueva funcionalidad se desarrolla en develop, que se fusiona posteriormente con master una vez completada y probada. La rama develop agrupa los desarrollos en curso, y cuando una versión está lista para ser liberada, se fusiona develop con master, desencadenando automáticamente el proceso de pruebas y el despliegue en producción.

El sistema de automatización está implementado con GitHub Actions mediante un workflow **Main** que orquesta las tareas clave del ciclo de vida del software. Este workflow está diseñado para adaptarse al flujo Git establecido y asegurar una entrega continua y sin errores.

El workflow Main es el encargado de gestionar de forma completa el proceso de integración y despliegue continuo de la aplicación. Este flujo se activa automáticamente cada vez que se realiza un push sobre la rama master, o bien de forma manual mediante ejecución por workflow_dispatch, garantizando flexibilidad para realizar despliegues controlados.

Este workflow se divide en dos fases principales: **build** y **deploy**, estructuradas como jobs independientes. Cada fase realiza tareas específicas que garantizan

la calidad del código, la correcta generación de la imagen Docker y su despliegue seguro en la nube de Azure.

Fase 1: Build

Esta fase tiene como objetivo verificar la calidad del código y generar la imagen de contenedor que se desplegará posteriormente:

1. **Pruebas automatizadas**

   Se ejecutan todos los tests unitarios y de integración del Backend, organizados por paquetes (service y controller) con el perfil test (Código 4.11), para asegurar que la lógica de negocio y los endpoints HTTP funcionen correctamente.

Código 4.11: Ejecución automática de pruebas en el workflow Main

```
- name: Run service tests # Se ejecutan todos los tests de servicios del backend
  run: |
    cd moai
    mvn test -Dtest=moai.service.** -Dspring.profiles.active=test

- name: Run controller tests # Se ejecutan todos los tests de controladores del
    backend
  run: |
    cd moai
    mvn test -Dtest=moai.controller.** -Dspring.profiles.active=test
```

2. **Generación de versión única**

   Se crea una etiqueta de versión basada en la fecha y hora del despliegue, que se utiliza como tag para versionar la imagen Docker (Código 4.12).

Código 4.12: Creación de etiqueta para la imagen Docker en el workflow Main

```
# Generacion de imagenes
- name: Generate tag for date
  run: echo "TIMESTAMP=$(date +'%Y%m%d.%H%M%S')" >> $GITHUB_ENV

- name: Export TIMESTAMP # Se exporta la variable para permitir su uso en otros
    jobs
  run: echo "TIMESTAMP=${{ env.TIMESTAMP }}" >> $GITHUB_OUTPUT
  id: timestamp
```

Código 4.13: Generación de una imagen Docker en el workflow Main

```
- name: Generate Docker image
  run: |
    cd $GITHUB_WORKSPACE
    docker build -t ${{ secrets.DOCKERHUB_USERNAME }}/moai:${{ env.TIMESTAMP }}
        -f Dockerfile .
```

3. **Construcción de la imagen Docker**

   Se genera una imagen a partir del Dockerfile del proyecto, etiquetada con la marca de tiempo anteriormente definida (Código 4.13). El Dockerfile contiene las instrucciones necesarias para compilar y unificar en una sola aplicación Backend y Frontend (Código 4.14), y está dividido en tres partes: la construcción del Frontend, la construcción del Backend y la construcción de la imagen final unificada.

   Código 4.14: Fichero Dockerfile utilizado para la generación de la imagen Docker en el workflow Main

```
# Etapa 1: Construccion del frontend
FROM node:18 AS frontend-builder
WORKDIR /app
COPY vue/moai/package.json vue/moai/package-lock.json ./
RUN npm install
COPY vue/moai/ .
RUN npm run build

# Etapa 2: Construccion del backend
FROM maven:3.8.6-eclipse-temurin-17 AS backend-builder
WORKDIR /app
COPY moai/ /app
COPY --from=frontend-builder /app/dist/ src/main/resources/static/
RUN mvn clean package -DskipTests

# Etapa 3: Construccion de la imagen final
FROM eclipse-temurin:17-jdk AS final
WORKDIR /app
COPY --from=backend-builder /app/target/*.jar app.jar
EXPOSE 8090
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

4. **Subida a Docker Hub**

   Una vez construida la imagen, se realiza la autenticación en Docker Hub para publicarla en el repositorio del proyecto, dejándola disponible para el entorno de producción, como se puede ver en el Código 4.15.

   Código 4.15: Subida de la imagen Docker a Docker Hub en el workflow Main

```
# Subida de la imagen a DockerHub
- name: Login to DockerHub
  run: docker login -u "${{ secrets.DOCKERHUB_USERNAME }}" -p "${{
      secrets.DOCKERHUB_TOKEN }}"

- name: Push image to DockerHub
  run: docker push ${{ secrets.DOCKERHUB_USERNAME }}/moai:${{ env.TIMESTAMP }}
```

Fase 2: Deploy

Tras finalizar correctamente la fase de build, comienza la fase de despliegue automático en Azure:

1. **Inicio de sesión en Azure**

   Se establece una conexión segura mediante credenciales federadas configuradas con la identidad administrada de Azure, sin necesidad de gestionar claves manualmente (Código 4.16).

   Código 4.16: Inicio de sesión en Azure realizado en el workflow Main

```
- name: Login to Azure
  uses: azure/login@v2
  with:
    client-id: ${{ secrets.AZURE_CLIENT_ID }}
    tenant-id: ${{ secrets.AZURE_TENANT_ID }}
    subscription-id: ${{ secrets.AZURE_SUBSCRIPTION_ID }}
```

2. **Despliegue en Instancias de Contenedor de Azure**

   Se lanza una nueva instancia de contenedor utilizando la imagen recién publicada en Docker Hub. Durante el proceso se definen los recursos necesarios (CPU, memoria, puertos, sistema operativo...) y se inyectan las variables de entorno requeridas para que la aplicación se conecte de forma segura a la base de datos del Servidor flexible de Azure Database for MySQL. Todo este proceso se puede observar en el Código 4.17.

   Código 4.17: Despliegue de la aplicación en Azure en el workflow Main

```
- name: Deploy container in Azure
  run: |
    az container create \
    --resource-group ${{ secrets.AZURE_GROUP }} \
    --name moai \
    --image ${{ secrets.DOCKERHUB_USERNAME }}/moai:${{
        needs.build.outputs.TIMESTAMP }} \
    --dns-name-label moai-comunidades \
    --registry-login-server index.docker.io \
    --registry-username ${{ secrets.DOCKERHUB_USERNAME }} \
    --registry-password ${{ secrets.DOCKERHUB_TOKEN }} \
    --ports 8090 \
    --os-type Linux \
    --cpu 1 \
    --memory 1.5 \
    --environment-variables \
        SPRING_DATASOURCE_URL=
            "jdbc:mysql://${{ secrets.AZURE_MYSQL_HOST }}:3306/
            ${{ secrets.AZURE_MYSQL_DATABASE }}?
            serverTimezone=UTC&useSSL=true" \
        SPRING_DATASOURCE_USERNAME="${{ secrets.AZURE_MYSQL_USER }}" \
        SPRING_DATASOURCE_PASSWORD="${{ secrets.AZURE_MYSQL_PASSWORD }}"
```

5

Conclusiones y trabajos futuros

En el presente capítulo se detallan las conclusiones extraídas de este proyecto y las propuestas para trabajos futuros.

5.1. Conclusión

El desarrollo de esta aplicación web para gestionar comunidades de propietarios ha resultado en una experiencia positiva de cara a mis conocimientos, ya que me ha permitido profundizar en múltiples áreas del desarrollo de software, especialmente aquellas en las que no tenía tantas competencias. Durante el proceso, he podido aplicar conocimientos técnicos que había adquirido a lo largo de mi formación, además de desarrollar nuevas habilidades que han sido necesarias para poder completar el proyecto.

Para la elaboración de Moai, he tenido la oportunidad de aplicar mis conocimientos de Java y Spring, dos tecnologías que, si bien ya había usado con anterioridad, me permitieron sentar una buena base para comenzar con el desarrollo de la aplicación. Esto hizo que la implementación de la parte Backend del proyecto resultase más sencilla y se pudiese cumplir con todos los objetivos previstos.

Por el lado opuesto, el desarrollo de la parte Frontend supuso la utilización de nuevas tecnologías que aún no había empleado, como es el framework Vue.js. Esto implicó adquirir nuevos conocimientos al respecto que me permitiesen crear

la aplicación como tenía previsto. Sin embargo, según fui integrando estas nuevas herramientas en el proyecto, pude comprobar su enorme utilidad a la hora de simplificar el desarrollo de la aplicación, además de permitir elaborar una interfaz con un mejor diseño y que proporciona una experiencia de usuario superior. Durante mi progreso en el desarrollo, mis habilidades con estas nuevas tecnologías fueron aumentando significativamente, reduciendo los tiempos de implementación.

Uno de los retos más relevantes a lo largo del proyecto fue el despliegue de la aplicación en Azure, una plataforma con la que no había tenido contacto previo durante mi formación académica. En particular, uno de los desafíos más complejos fue la configuración del servicio de base de datos MySQL en la nube. La falta de experiencia en este tipo de entornos me llevó a investigar de manera exhaustiva cómo realizar una correcta implementación y conexión de MySQL dentro del ecosistema de Azure. Este proceso implicó familiarizarme con aspectos técnicos clave, como la configuración de redes para garantizar la conectividad, la aplicación de medidas de seguridad para proteger la información almacenada y la correcta gestión de las credenciales de acceso. Todo ello supuso una adaptación minuciosa para cumplir con los requisitos propios de la plataforma. Pese a los problemas del comienzo, esta etapa me permitió adquirir competencias valiosas en la gestión de bases de datos en la nube, ampliando así mis conocimientos sobre el despliegue de aplicaciones en entornos reales de producción.

Para que un proyecto de gran calibre pueda salir adelante, es necesario establecer y seguir una buena metodología clara y correctamente definida. Si no se hace, es probable que los tiempos de desarrollo aumenten considerablemente. Ciertas tareas, como puede ser la redacción de los mensajes de commit en Git, pueden suponer conflictos en el equipo a lo largo del tiempo. En mi caso, para esta tarea en concreto establecí un esquema a seguir que garantizase una rápida comprensión de los cambios introducidos en un commit. Este esquema consiste en indicar, al comienzo del mensaje, el tipo de operación que se ha llevado a cabo en la aplicación, que puede ser ADD si se han introducido nuevas funcionalidades, UPD si se han actualizado algunas ya existentes, o DEL si se ha eliminado una funcionalidad o fichero. Tras esto, se escribe un breve y conciso mensaje que ayude a entender la modificación realizada de forma resumida. Este es un ejemplo de las labores realizadas para garantizar el cumplimiento de los objetivos establecidos para el proyecto.

Si bien en esta ocasión no se ha realizado un trabajo en equipo y todo se ha desarrollado de forma individual, ha sido una buena oportunidad para poner en práctica determinados conceptos de las metodologías ágiles. El hecho de que yo haya sido la única persona en el proyecto ha permitido aplicar dichos conceptos de una manera más sencilla y con menos conflictos, tolerando una simplificación del proceso que se realizaría en un equipo grande. Esto me aporta una mayor seguridad a la hora de contribuir con estos conocimientos al éxito del equipo, cuando me encuentre desarrollando proyectos colaborativos futuros.

5.2. Trabajos futuros

Para este proyecto encuentro determinadas áreas que podrían ser más expandidas y potenciadas para aportar un valor superior a la aplicación.

En primer lugar, veo opciones de mejora en el área de seguridad de la aplicación. Si bien la seguridad que ha sido implementada es adecuada y robusta, estaría bien una mayor comprobación de las contraseñas, exigiendo el uso de claves más fuertes y con variedad de caracteres, así como integrar el uso del doble factor de autenticación mediante el teléfono del usuario. También se podría implementar una opción para restablecer la contraseña de forma automática, enviando un correo electrónico de validación al propietario de la cuenta.

Otro campo en el que se podrían introducir mejoras es en el manejo de errores, ya que, si bien la aplicación es capaz de responder correctamente en caso de error, hay ciertos eventos que se podrían notificar de manera más rápida al usuario, o utilizando otros sistemas más integrados en lugar del modal de errores, aportando una mejor experiencia de uso final.

Respecto a las mejoras relacionadas con la funcionalidad proporcionada por Moai, cabe destacar una mayor integración de la mensajería entre los vecinos y el administrador. De esta forma, se podría prescindir completamente del uso de canales externos como el correo electrónico o el buzón tradicional. Con ello, los usuarios recibirían una notificación al instante para hacerlos conocedores de que tienen novedades en sus comunicaciones.

Asociada a lo anterior se encuentra una posible opción de mejora en lo respectivo a las incidencias, donde los usuarios pudiesen actualizar sus estados y ser visualizados por los demás en tiempo real, permitiendo una resolución más rápida de los conflictos abiertos.

Otra posible mejora de interés estaría relacionada con los recibos, implementando la generación de remesas SEPA (Single Euro Payments Area). Estas remesas permitirían realizar los pagos y cobros de los recibos, dando la posibilidad de domiciliarlos, lo que aportaría un extra de automatización a la aplicación, liberando al administrador de la necesidad de realizar estas tareas.

Por último, se podría considerar un cambio en el planteamiento de negocio de la aplicación. Actualmente Moai está configurada para ser desplegada en un servidor y atender exclusivamente las necesidades de una única comunidad, siendo necesario desplegar la aplicación en más servidores para dar servicio a más comunidades de propietarios. De esta forma, cada comunidad se encarga de su propia configuración y de los gastos que conlleve, aportando flexibilidad a la hora de utilizar el sistema. Sin embargo, internamente la aplicación está bastante preparada para que, en caso de quererse hacer en un futuro, se pudiera desplegar en un único servidor y atender a todas las comunidades que sea necesario desde allí, teniendo

cada una su propia configuración personalizada, pero compartiendo todas una misma plataforma. Esto supondría un cambio en el modelo de negocio planteado actualmente por Moai, pero que sería totalmente válido si las necesidades de la situación lo aconsejan.

Bibliografía

[1] Oracle, "Java software," Online, n.d., [Online]. Disponible: https://www.oracle.com/java/.

[2] Broadcom, "Spring framework," Online, n.d., [Online]. Disponible: https://spring.io/projects/spring-framework.

[3] ——, "Spring boot," Online, n.d., [Online]. Disponible: https://spring.io/projects/spring-boot.

[4] ——, "Spring security," Online, n.d., [Online]. Disponible: https://spring.io/projects/spring-security.

[5] Apache Software Foundation, "Apache maven," Online, n.d., [Online]. Disponible: https://maven.apache.org/.

[6] Oracle, "Mysql," Online, n.d., [Online]. Disponible: https://www.mysql.com/.

[7] T. Mueller, "H2 database engine," Online, n.d., [Online]. Disponible: https://www.h2database.com/html/main.html.

[8] R. Hat, "Hibernate orm," Online, n.d., [Online]. Disponible: https://hibernate.org/orm/.

[9] Apache Software Foundation, "Apache tomcat," Online, n.d., [Online]. Disponible: https://tomcat.apache.org/.

[10] Apryse, "Discover itext pdf," Online, n.d., [Online]. Disponible: https://itextpdf.com/.

[11] The JUnit Team, "Junit 5," Online, n.d., [Online]. Disponible: https://junit.org/junit5/.

[12] Software Freedom Conservancy, "Selenium," Online, n.d., [Online]. Disponible: https://www.selenium.dev/.

[13] Mountainminds GmbH Co., "Jacoco - documentation," Online, n.d., [Online]. Disponible: https://www.jacoco.org/jacoco/trunk/doc/.

[14] Apache Software Foundation, "Apache pdfbox - a java pdf library," Online, n.d., [Online]. Disponible: https://pdfbox.apache.org/.

[15] Johan Haleby, "Rest assured," Online, n.d., [Online]. Disponible: https://rest-assured.io/.

[16] Mozilla Corporation, "Javascript," Online, n.d., [Online]. Disponible: https://developer.mozilla.org/es/docs/Web/JavaScript.

[17] json.org, "Json," Online, n.d., [Online]. Disponible: https://www.json.org/json-es.html.

[18] FasterXML, LLC, "Jackson," Online, n.d., [Online]. Disponible: https://github.com/FasterXML/jackson.

[19] Web Hypertext Application Technology Working Group (WHATWG), "Html," Online, n.d., [Online]. Disponible: https://html.spec.whatwg.org/multipage/.

[20] World Wide Web Consortium, "Cascading style sheets," Online, n.d., [Online]. Disponible: https://www.w3.org/Style/CSS/.

[21] FontIcons, Inc., "Font awesome," Online, n.d., [Online]. Disponible: https://fontawesome.com/.

[22] Evan You, "Vue.js - the progressive javascript framework," Online, n.d., [Online]. Disponible: https://vuejs.org/.

[23] axios-http, "Axios," Online, n.d., [Online]. Disponible: https://axios-http.com/es/docs/intro.

[24] OpenJS Foundation, "Node.js - ejecuta javascript en cualquier parte," Online, n.d., [Online]. Disponible: https://nodejs.org/es.

[25] Okta, "Introduction to json web tokens," Online, n.d., [Online]. Disponible: https://jwt.io/introduction.

[26] Broadcom, "Spring tools," Online, n.d., [Online]. Disponible: https://spring.io/tools.

[27] Microsoft Corporation, "Visual studio code - code editing. redefined," Online, n.d., [Online]. Disponible: https://code.visualstudio.com/.

[28] Software Freedom Conservancy, "Git," Online, n.d., [Online]. Disponible: https://git-scm.com/.

[29] GitHub, Inc., "Github - build and ship software on a single, collaborative platform," Online, n.d., [Online]. Disponible: https://github.com/.

[30] Docker Inc., "Docker: Accelerated container application development," Online, n.d., [Online]. Disponible: https://www.docker.com/.

[31] ——, "Docker hub container image library," Online, n.d., [Online]. Disponible: https://hub.docker.com/.

[32] Microsoft Corporation, "Microsoft azure," Online, n.d., [Online]. Disponible: https://azure.microsoft.com/es-es/.

[33] ——, "Documentación de azure container instances," Online, n.d., [Online]. Disponible: https://learn.microsoft.com/es-es/azure/container-instances/.

[34] ——, "¿qué es azure database for mysql: servidor flexible?" Online, n.d., [Online]. Disponible: https://learn.microsoft.com/es-es/azure/mysql/flexible-server/overview.

[35] Atlassian, "Flujo de trabajo de gitflow," Online, n.d., [Online]. Disponible: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow.

[36] GitHub, Inc., "Documentación de github actions," Online, n.d., [Online]. Disponible: https://docs.github.com/es/actions.

[37] ScrumGuides.org, "Scrum guides," Online, n.d., [Online]. Disponible: https://scrumguides.org/.

[38] Microsoft Corporation, "¿qué es devops?" Online, n.d., [Online]. Disponible: https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-devops#:~:text=Definici%C3%B3n%20de%20DevOps,los%20clientes%20de%20forma%20constante.

[39] Broadcom, "Spring web mvc," Online, n.d., [Online]. Disponible: https://docs.spring.io/spring-framework/reference/web/webmvc.html.

Apéndices

A

Lanzamiento de la aplicación en local

En este apéndice se explicarán los pasos detallados que son necesarios para ejecutar Moai localmente.

A.1. Prerrequisitos

Es necesario tener instalado el software que ha sido descrito en la memoria y que emplea la aplicación para su funcionamiento. Principalmente, este software consiste en **Java 17** para ejecutar el Backend, junto con **Maven** para realizar la compilación y las dependencias necesarias. El Frontend requiere de **Node.js**, que incluye **npm** para lanzar la aplicación, además de las dependencias requeridas.

A.2. Lanzamiento

Los pasos necesarios para ejecutar tanto Backend como Frontend se explicarán a continuación.

A.2.1. Clonar el repositorio

El repositorio de Moai en GitHub puede clonarse con el siguiente comando, abriendo una consola en el directorio deseado:

```
1 git clone https://github.com/migueld7/gestor_de_comunidades.git
```

A.2.2. Preparación del Frontend

Para compilar la aplicación Vue.js correspondiente al Frontend, primero habrá que acceder a **vue/moai** a partir de la raíz del repositorio clonado. Una vez ahí, será necesario ejecutar el siguiente comando:

```
1  npm run build
```

Con esto, se creará una nueva carpeta **dist** dentro de **vue/moai**. Será necesario copiar todo el contenido de esta carpeta dentro de **moai/src/main/resources/static** para que el Backend pueda ejecutar también la aplicación Frontend, unificando la ejecución de ambas aplicaciones.

Sin embargo, este paso no es completamente necesario, puesto que, al clonar el repositorio, ya se habrán descargado también los ficheros compilados, si bien es posible que no se encuentren en su última versión y carezcan de ciertos cambios, por lo que se recomienda realizar el proceso previamente mencionado.

A.2.3. Lanzamiento del Backend junto al Frontend

Para ejecutar la aplicación Java correspondiente al Backend, que incluye también la ejecución del Frontend, primero es necesario entrar en la carpeta **moai**, que se encuentra en la raíz del repositorio clonado. Una vez dentro, habrá que ejecutar en una consola los siguientes comandos:

```
1  mvn clean install
2  mvn spring-boot:run
```

Una vez hecho, tras unos segundos, la aplicación ya se encontrará accesible introduciendo en un navegador la URL **http://localhost:8090/login**. Sin embargo, es posible que sea necesario ajustar antes de la ejecución en el fichero **application.properties**, que se encuentra en **moai/src/main/resources**, los datos relativos a la base de datos que se va a utilizar. Por defecto está establecida la configuración que necesita GitHub Actions para desplegar la aplicación en Azure, pero se pueden introducir otros datos para utilizar una base de datos persistente o una que se ejecute en memoria y solo funcione durante la ejecución de Moai, como es H2.

A.2.4. Ejecución de pruebas

Para lanzar la batería de pruebas que incorpora la aplicación será necesario ejecutar el siguiente comando:

```
1  mvn test -Dspring.profiles.active=test
```

Con esto, no solo se ejecutan las pruebas, sino que se indica que se haga utilizando el perfil **test**, configurado específicamente para una correcta ejecución de las mismas.