



ugr | Universidad
de **Granada**

TRABAJO FIN DE GRADO
GRADO EN INGENIERIA INFORMATICA

Sistema multiplataforma para la búsqueda y gestión de pisos compartidos

Autor
Alonso Doña Martínez

Director
Luis López Escudero



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

—
Granada, Septiembre de 2025



ugr | Universidad
de **Granada**

Sistema multiplataforma para la búsqueda y gestión de pisos compartidos

Autor

Alonso Doña Martínez

Director

Luis López Escudero

Sistema multiplataforma para la búsqueda y gestión de pisos compartidos

Alonso Doña Martínez

Palabras clave: *Software libre, Tecnología aplicada a la vivienda, Co-Housing, Gestión de pisos compartidos, Desarrollo FullStack*

Resumen

Actualmente, uno de los principales problemas de la sociedad es la constante escalada de los precios en el ámbito de la vivienda, un desafío que afecta tanto a jóvenes como a adultos.

Una de las soluciones que está ganando terreno para mitigar los costes es la búsqueda de viviendas compartidas. Sin embargo, a pesar del contexto tecnológico actual, donde existen múltiples plataformas y portales para la búsqueda de viviendas y la organización de tareas, todos presentan limitaciones al no centrarse en ofrecer soluciones eficientes que pongan al usuario en el centro. En lugar de tratar la vivienda solo como un espacio, estas plataformas deberían considerar también los lazos y relaciones entre las personas que la habitan, brindándoles herramientas para la gestión y organización de tareas y promoviendo una convivencia agradable.

Esta *plataforma*¹ nace con el objetivo de cambiar el enfoque tradicional de los pisos compartidos, creando un entorno agradable que cubra las necesidades e intereses de todos los miembros de la vivienda.

¹El repositorio del proyecto está disponible en: [Repositorio en GitHub](#)

Multiplatform system for the search and management of shared housing

Alonso Doña Martínez

Keywords: *Free software, Technology applied to housing, Co-Housing, Shared housing management, FullStack Development*

Abstract

Currently, one of the main problems in society is the constant rise in housing prices, a challenge that affects both young people and adults.

One of the solutions that is gaining traction to reduce costs is the search for shared housing. However, despite the current technological landscape, where multiple platforms and portals exist for finding housing and organizing tasks, they all have limitations as they do not focus on providing efficient and convenient solutions that put the user at the center. Instead of treating housing merely as a space, these platforms should also consider the bonds and relationships between the people who live there, offering them tools for task management and organization while promoting a pleasant coexistence.

This *platform*² was created with the goal of changing the traditional approach to shared housing, fostering an environment that aligns with the needs and interests of all household members.

²The repository for this project is available on GitHub: [Github Repository](#)

Yo, **Alonso Doña Martínez**, alumno de la titulación Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 77555566A, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Alonso Doña Martínez

Granada a 2 de Septiembre de 2025 .

D. **Luis López Escudero**, Profesor del Área de XXXX del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Sistema multiplataforma para la búsqueda y gestión de pisos compartidos***, ha sido realizado bajo su supervisión por **Alonso Doña Martínez**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 2 de Septiembre de 2025 .

El director:

Luis López Escudero

Agradecimientos

En primer lugar me gustaría expresar mi agradecimiento a mi tutor de TFG, Luis López Escudero, por su ayuda, compromiso y guía durante los meses de desarrollo de este proyecto.

Agradecer al personal docente de la Universidad de Granada por la dedicación y enseñanza durante estos años, claves en mi formación tanto profesional como personal.

A mi familia, por su apoyo y respaldo constante, por estar conmigo y ayudarme siempre.

A mi hermano Rafa, por ser una inspiración constante durante estos años de carrera.

A mis amigos y compañeros, por realizar este camino junto a mí, por estar en los momentos más duros, por las noches de estudio y por todas las risas.

A Carlos, por su compañerismo y amistad tanto en los momentos difíciles como bonitos de esta etapa.

Finalmente, me gustaría agradecer a todas las personas que han permitido que el desarrollo de este proyecto haya sido posible.

Índice general

1. Introducción	12
1.1. Contexto	12
1.2. Motivación	12
1.3. Objetivos	13
1.4. Estructura de la memoria	14
2. Estado del arte	15
2.1. Análisis de las posibles soluciones al problema propuesto	15
2.2. Revisión del estado del arte	15
2.2.1. Búsqueda de pisos	15
2.2.2. Gestión de tareas en pisos compartidos	21
2.3. Tecnologías en el mercado	23
2.3.1. Back-End	24
2.3.2. Front-End	25
2.3.3. Base de Datos	27
2.4. Conclusión tras estudiar el mercado	27
3. Especificación de Requisitos	29
3.1. Propósito	29
3.2. Alcance	29
3.3. Definiciones, siglas y abreviaturas	29
3.4. Perspectiva del producto	30
3.5. Restricciones	31
3.6. Funciones del producto	32
3.7. Redacción de requisitos	32
3.8. Requisitos funcionales	33
3.8.1. Bloque Gestión de Usuarios y Autenticación	33
3.8.2. Bloque Gestión de Comunidades	33
3.8.3. Bloque Gestión de Tareas	34
3.8.4. Bloque Recomendación de Comunidades	34

Índice general

3.9. Requisitos No Funcionales	35
3.9.1. Requisitos de Seguridad	35
3.9.2. Requisitos de Interfaces Externas	36
4. Planificación del Proyecto	39
4.1. Propósito	39
4.2. Cálculo de la Velocidad de Desarrollo	40
4.3. Listado de historias de usuario	41
4.4. Descripción de las iteraciones	44
4.4.1. Duración y Esfuerzo	44
4.4.2. Distribución de Funcionalidades	44
4.5. Diagrama de Gantt	45
4.6. Estimación de costes	48
4.6.1. Costes de Herramientas Utilizadas	48
4.6.2. Coste de Desarrollo	49
4.6.3. Costes de Infraestructura	49
4.6.4. Coste de Mantenimiento	49
4.6.5. Resumen de Costes	50
5. Análisis	51
5.1. Objetivo	51
5.2. Product backlog	51
5.2.1. Tarjetas de historia de usuario	55
5.3. Diagrama de clases de análisis	87
6. Diseño	89
6.1. Arquitectura del sistema	89
6.2. Arquitectura del Backend	90
6.2.1. Estructura de los Microservicios	90
6.2.2. Estructura Interna de los Microservicios	90
6.3. Arquitectura del Frontend	91
6.3.1. Scream Architecture	91
6.3.2. Estructura de la Carpeta Feature	92
6.4. Descripción de la API de cada microservicio	92
6.4.1. Microservicio Gestión de Usuarios	93
6.4.2. Microservicio Gestión de Comunidades	94
6.4.3. Recomendación de Comunidades	97
6.4.4. Solicitudes	97
6.5. Diagrama Entidad-Relación	98
6.6. Diagrama de Clases	100
6.7. Diagramas de Secuencia	100
6.8. Diseño de la interfaz de usuario	105
6.8.1. Landing	105
6.8.2. Login	106

6.8.3. Registro	107
6.8.4. Home	109
6.8.5. Home de la Comunidad	110
6.8.6. Tareas	112
6.8.7. Tarea Específica	113
6.8.8. Buscador de comunidades	114
6.8.9. Perfil del usuario	116
6.8.10. Solicitudes	117
6.9. Diagrama de navegabilidad	118
7. Implementación	120
7.1. Herramientas	120
7.1.1. VSCode IDE	120
7.1.2. GitHub	121
7.1.3. Contenerización con Docker	123
7.1.4. Orquestación con Docker Compose	124
7.2. Funcionalidades clave del sistema	124
7.2.1. Inicio de sesión y Registro	125
7.2.2. Comunicación Asíncrona entre Microservicios	127
7.2.3. Recomendación de Comunidades	128
7.2.4. Proceso de Unión a una Comunidad	133
7.2.5. Gestión de comunidades	135
7.2.6. Administración de Tareas por Parte de los Usuarios	137
8. Pruebas	140
8.1. Microservicio de Gestión de Usuarios	140
8.1.1. Pruebas Unitarias	140
8.1.2. Pruebas End to End	141
8.2. Microservicio de Gestión de Comunidades	142
8.2.1. Pruebas Unitarias	142
8.2.2. Pruebas End to End	143
8.3. Microservicio Recomendador	143
8.3.1. Pruebas Unitarias	144
8.3.2. Pruebas End to End	144
8.4. Microservicio API Gateway	145
8.4.1. Pruebas Unitarias	145
8.4.2. Pruebas End to End	145
8.5. Microservicio de Solicitudes	145
8.5.1. Pruebas Unitarias	145
8.5.2. Pruebas End to End	146
8.6. Frontend	147
8.6.1. Pruebas de Componentes	147
8.7. Ejecución y Cobertura de Pruebas	148

Índice general

9. Despliegue	149
9.1. Despliegue del frontend	149
9.2. Despliegue del backend	150
10. Conclusiones	154
10.1. Balance general	154
10.1.1. Objetivos conseguidos	154
10.2. Valoración final	155
10.3. Trabajos Futuros	156

Índice de figuras

2.1. Interfaz de búsqueda de Idealista.	16
2.2. Configuración del perfil de usuario	18
2.3. Configuración del perfil de la vivienda	19
2.4. Sección de búsqueda de vivienda por afinidad.	20
2.5. Calendario de la aplicación.	21
2.6. Sección para crear y editar una tarea.	22
2.7. Lenguajes de programación y tecnologías presentes en el mercado.	23
2.8. Arquitectura NodeJS.	24
2.9. Starters de Spring Boot.	25
2.10. Explicación del concepto JSX.	26
2.11. Explicación del concepto de TypeScript.	26
2.12. Logo PostgreSQL.	27
2.13. Logo MongoDB.	27
3.1. Diagrama interacción entre componentes.	31
3.2. Boceto "Landing Page".	37
3.3. Boceto Página Buscador.	38
4.1. Metodología ágil Scrum.	39
4.2. Diagrama de Gantt iteración 0.	46
4.3. Diagrama de Gantt iteración 1.	46
4.4. Diagrama de Gantt iteración 2.	47
4.5. Diagrama de Gantt iteración 3.	47
4.6. Diagrama de Gantt iteración 4.	48
5.1. Diagrama UML de clases de análisis.	88
6.1. Diagrama de la arquitectura del sistema.	89
6.2. Diagrama de la arquitectura del backend	91
6.3. Explicación arquitectura screaming. Fuente: [25]	92
6.4. Diagrama de Entidad-Relación del sistema.	99

Índice de figuras

6.5. Diagrama UML de entidades del sistema.	100
6.6. Diagrama de secuencia del registro.	101
6.7. Diagrama de secuencia del login.	101
6.8. Diagrama de secuencia unión a una comunidad.	102
6.9. Diagrama de secuencia búsqueda de comunidades.	102
6.10. Diagrama de secuencia consultar tareas.	103
6.11. Diagrama de secuencia repartir tareas.	103
6.12. Diagrama de secuencia crear tarea.	104
6.13. Boceto de la Landing Page.	105
6.14. Boceto del login.	106
6.15. Boceto del registro.	107
6.16. Boceto del registro selección de preferencias.	108
6.17. Boceto del home.	109
6.18. Boceto del home de la comunidad.	110
6.19. Boceto de la lista de tareas del usuario.	112
6.20. Boceto de la tarea.	113
6.21. Boceto de la sección de búsqueda.	114
6.22. Boceto del perfil del usuario.	116
6.23. Boceto de la sección de Solicitudes.	117
6.24. Diagrama de navegabilidad del sistema.	119
7.1. Logo IDE VSCode	120
7.2. Logo GitHub	121
7.3. Flujo de desarrollo github actions	122
7.4. Imagen tablero GitHub Project	123
7.5. Logo docker	123
7.6. Flujo de autenticación a través del API Gateway.	125
7.7. Pantalla del login en el front-end	127
7.8. Flujo de eventos y comunicación entre microservicios usando RabbitMQ	128
7.9. Explicación del sistema de recomendación para usuarios	132
7.10. Pantalla de recomendaciones en el front-end	133
7.11. Esquema del flujo para la unión de un usuario a una comunidad	134
7.12. Vista del usuario para la administración de tareas dinámicamente	139
8.1. Muestra del resultado de los test del microservicio Gestión de Usuarios	142
8.2. Muestra del informe de cobertura por Codecov	148
9.1. Logo Github Pages	149
9.2. Logo Railway	150
9.3. Muestra del tablero de despliegue del primer proyecto en Railway	151
9.4. Muestra del tablero de despliegue del segundo proyecto en Railway	152

Capítulo **1**

Introducción

1.1. Contexto

En la actualidad, el acceso a la vivienda se ha convertido en un desafío tanto para jóvenes como adultos, especialmente en grandes ciudades donde los precios de compra y alquiler siguen aumentando drásticamente.

En este contexto, ha surgido una nueva alternativa de convivencia, el *co-housing* [34], una solución innovadora que combina el acceso a una vivienda digna con el aprovechamiento de recursos y espacios compartidos, fomentando la colaboración y la gestión conjunta. Sin embargo, este modelo suele aplicarse en infraestructuras diseñadas específicamente para ello, lo que implica un alto coste inicial y una planificación a largo plazo.

Dado que los pisos son la forma de vivienda más extendida en las ciudades, adaptar los principios del *co-housing* —como la búsqueda de personas afines y la vida en comunidad— a esta modalidad permitirá ofrecer una solución que mejore la calidad de vida de los individuos. Además, permitirá optimizar la organización de tareas y mejorar la convivencia entre los integrantes.

A pesar de sus evidentes beneficios, la gestión de pisos compartidos siguiendo estos principios presenta ciertos desafíos, como la organización y reparto de tareas o la búsqueda en función de la afinidad. En este sentido, la tecnología puede desempeñar un papel crucial en la resolución de estos problemas.

Este trabajo tiene como objetivo desarrollar una plataforma de creación y gestión de comunidades en pisos compartidos, permitiendo a los usuarios administrar estas viviendas de forma sencilla y moderna, fomentando un modelo de convivencia colaborativo.

1.2. Motivación

El crecimiento exponencial de las tendencias de convivencia en comunidad y el incremento del precio de la vivienda ha generado la necesidad de una platafor-

ma tecnológica que facilite la creación, organización y administración de estos entornos en la vivienda más común: los pisos.

A pesar de la existencia de plataformas que ayudan en la búsqueda de viviendas o en la organización de tareas diarias, muchas de ellas carecen de toda la funcionalidad necesaria para aplicar los principios del *co-housing*. En general, estas plataformas suelen abordar solo uno de los principios fundamentales de este tipo de convivencia, sin proporcionar una solución completa.

Teniendo esto en cuenta, la motivación de este proyecto radica en las siguientes cuestiones:

- **Falta de plataformas especializadas:** Las plataformas actuales no están diseñadas para gestionar de manera completa la convivencia en pisos compartidos bajo los principios del *co-housing*. Esto dificulta la implementación efectiva de este modelo de convivencia.
- **Plataforma moderna e intuitiva:** Se busca ofrecer una aplicación robusta, con una interfaz moderna y fácil de usar, que permita a los usuarios interactuar de forma sencilla y eficiente.
- **Solución a un problema actual:** El acceso a la vivienda es un desafío creciente. Según el informe del Parlamento Europeo ([Europarl, 2024](#)), el coste de la vivienda ha aumentado significativamente en la gran mayoría de países de la Zona Euro [3], lo que ha llevado a muchas personas a optar por viviendas compartidas como alternativa asequible. Sin embargo, la falta de herramientas adecuadas para gestionar esta convivencia puede generar conflictos y reducir la calidad de vida de los residentes.

Con base en estos puntos, este trabajo propone el desarrollo de una plataforma tecnológica que afronte estas necesidades. De esta manera, no solo se atiende una demanda real del mercado, sino que también se aplican conocimientos avanzados de ingeniería de software, siguiendo estándares de desarrollo profesional.

1.3. Objetivos

El objetivo principal de este proyecto es la creación de un sistema multiplataforma que permita la gestión de pisos compartidos, ofreciendo una respuesta moderna que aborde los principios de búsqueda por afinidad y gestión eficiente de tareas. Este objetivo se ha articulado en torno a la consecución de los siguientes objetivos:

- **OE1:** Evaluar la arquitectura y tecnologías más convenientes para el problema.
- **OE2:** Estudiar la metodología a seguir y realizar una planificación en torno a ella.

- **OE3:** Utilizar *GitHub* [14] como control de versiones e integrarlo en todo el ciclo de vida de desarrollo.
- **OE4:** Desarrollar una interfaz moderna e intuitiva para el usuario con tiempos de carga reducidos.
- **OE5:** Desarrollar la gestión de tareas que permita al usuario ver qué tiene que hacer en cada momento y qué le toca realizar a sus compañeros.
- **OE6:** Permitir la visualización del estado de desarrollo de las tareas.
- **OE7:** Desarrollar un algoritmo de recomendación que permita poner en contacto usuarios afines.
- **OE8:** Aprender sobre los frameworks de desarrollo *Spring Boot* [32] y *React* [22] y ponerlos en práctica en el desarrollo del proyecto.

1.4. Estructura de la memoria

Durante el primer capítulo se han presentado tanto el contexto como la motivación para este trabajo, así como los objetivos principales que permitirán abordar el problema. El resto del contenido de la memoria se ha estructurado de la siguiente forma:

- En el capítulo 2 se ha realizado una descripción del estado del arte en relación a las principales tecnologías y técnicas presentes en el mercado sobre gestión de viviendas compartidas, algoritmos de recomendación y desarrollo de aplicaciones multiplataforma.
- En el capítulo 3 se va a proceder a realizar ...
- El último capítulo aborda las conclusiones y los trabajos futuros que pueden surgir a partir de este TFG.
- Para finalizar, se muestran las referencias bibliográficas usadas durante la realización de este trabajo.

Capítulo **2**

Estado del arte

2.1. Análisis de las posibles soluciones al problema propuesto

La gestión completa de pisos compartidos siguiendo los principios de convivencia propuestos por el *co-housing* plantea diversos desafíos, donde la tecnología puede ofrecer una solución efectiva que facilite su implementación.

En este contexto, es fundamental analizar y entender las soluciones tecnológicas existentes en el mercado, analizando tanto su enfoque como las herramientas utilizadas.

Para la realización de este estudio, se han analizado distintos proyectos y aplicaciones con el objetivo de identificar soluciones propuestas con anterioridad a este problema, se ha hecho especial enfoque en la usabilidad y en las herramientas utilizadas, tratando de extraer las mejores prácticas que sirvan de referencia para la creación de una solución efectiva.

2.2. Revisión del estado del arte

Para la realización de la revisión se va a seguir un esquema de temáticas, el proyecto se sustenta en dos grandes bloques: la gestión de tareas en viviendas compartidas y la búsqueda de pisos en función de afinidad entre personas. Dentro de estas dos temáticas será donde se analizarán las soluciones existentes, buscando su aportación al enfoque de co-housing en pisos compartidos.

2.2.1. Búsqueda de pisos

2.2.1.1. Idealista

Idealista [17] es un portal inmobiliario que permite la búsqueda de pisos y habitaciones en función de criterios como la ubicación, rango de precio, acep-

tación de fumadores y predisposición a mascotas. Es un portal ampliamente usado, por lo que contiene un gran volumen de ofertas en distintas ciudades. Sin embargo, no incorpora ninguna funcionalidad que permita evaluar la afinidad entre posibles inquilinos en función de sus intereses o estilo de vida.

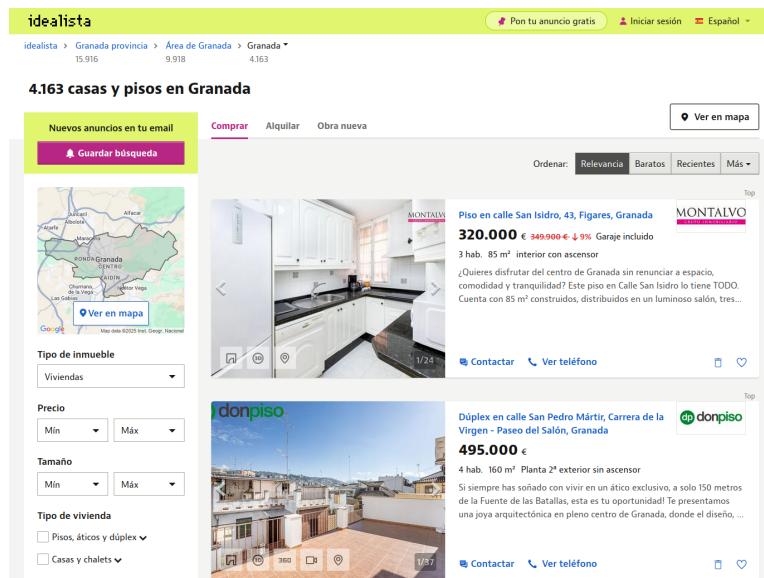


Figura 2.1: Interfaz de búsqueda de Idealista.

Interfaz y usabilidad

Como se puede ver en la imagen 2.1, la interfaz es simple e intuitiva lo que permite que se entienda toda su funcionalidad a la perfección. Es de destacar que la aplicación te permite, con pocas configuraciones, empezar la búsqueda de piso por lo que se hace muy cómoda de usar.

Características destacadas

- **Gran volumen de ofertas:** Al ser una plataforma muy popular cuenta con un gran número de ofertas en distintas ciudades.
- **Filtros de búsqueda:** Permite aplicar una serie de criterios en la búsqueda como rango de precios, aceptación de fumadores, predisposición a mascotas, lo cual ayuda al usuario a encontrar un piso más acorde a sus necesidades
- **Comunicación cliente-ofertante sencilla:** Facilita con su chat integrado la comunicación entre cliente y ofertante ya sea para consultar alguna duda o para realizar una reserva.

Limitaciones

- No tiene en cuenta la afinidad entre los inquilinos, lo que puede provocar que personas con diferentes intereses acaben conviviendo juntos.
- Se centra únicamente en la oferta de pisos, sin ofrecer ninguna herramienta para gestionar la vivienda compartida.
- La aplicación está orientada simplemente a encontrar potenciales inquilinos, ignorando la integración del nuevo miembro en la vivienda.

Conclusión

Idealista es una buena opción como portal de viviendas, ofrece posibilidades en prácticamente todas las ciudades y su facilidad de uso es correcta. Sin embargo, para la idea que buscamos es incompleta al no ofrecer nada relacionado con la búsqueda por afinidad.

2.2.1.2. Flatmate Finders

Flatmate Finders [10] es una plataforma que ofrece una solución bastante precisa al problema de búsqueda de pisos por afinidad. Permitiendo al usuario dos opciones principales:

- 1) **Buscar un piso:** el usuario puede crearse un perfil con información sobre su personalidad, gustos y preferencias. A partir de estos datos, la aplicación genera recomendaciones de ofertas de pisos mostrando un porcentaje de compatibilidad.
- 2) **Ofrecer un piso:** los propietarios pueden registrar su vivienda, estableciendo detalles como ubicación, precio, características y criterios sobre el perfil de los inquilinos afines.

Capítulo 2. Estado del arte

Pedro
Online: Today

Mobile
None provided



Date available	:	9 Apr 2025	
Length of stay	:	4 to 5 months	
Number of adults	:	One person	
Age group	:	18 to 24	
Gender	:	Man	
Sexuality	:	Prefer not to say	
Employment	:	Fully employed	
Smoking at home	:	Non-smoker	
Pets	:	No cat or dog	
Main interests	:	Art and culture • Bars, pubs or clubs	

About this person 

21 years old, computer science, 8 hours per day working, a relaxed and quiet person.

Important personal qualities 

Organization skills, very polite and respectful with people.

Figura 2.2: Configuración del perfil de usuario .

Home Description

Description of the home
Add information

Home features
Air-conditioning • BBQ facilities • Neighbourhood views

Bills and expenses
1000

Occupants Description

This home doesn't have any occupants at the moment.

Flatmate Preferences

Age group	:	18+
Gender	:	Any gender
Sexuality	:	Any sexuality
Smoking at home	:	Smokers okay
Pets	:	Pets okay

Figura 2.3: Configuración del perfil de la vivienda .

Interfaz y usabilidad

La aplicación presenta una interfaz intuitiva con un buen tiempo de respuesta, la aplicación destaca por ser fácil de usar y ofrecer en todo momento información al usuario sobre las recomendaciones. Es de destacar que tanto la parte de ofertante como la de buscador tienen coherencia en cuanto a estilo y son muy personalizables.

Características destacadas

- **Sistema de recomendación:** Utiliza un porcentaje de compatibilidad entre posible inquilino y vivienda, lo que facilita la búsqueda.
- **Explicación de compatibilidad:** Muestra un porcentaje indicando el grado de compatibilidad, además refleja las razones de incompatibilidad para dar "feedback" al usuario.
- **Configuración de perfil:** La creación de perfiles tanto para usuario como para propietario permite representar bien los intereses que se buscan en la comunidad o que tienen los usuario, siendo además sencillo y rápido de configurar.

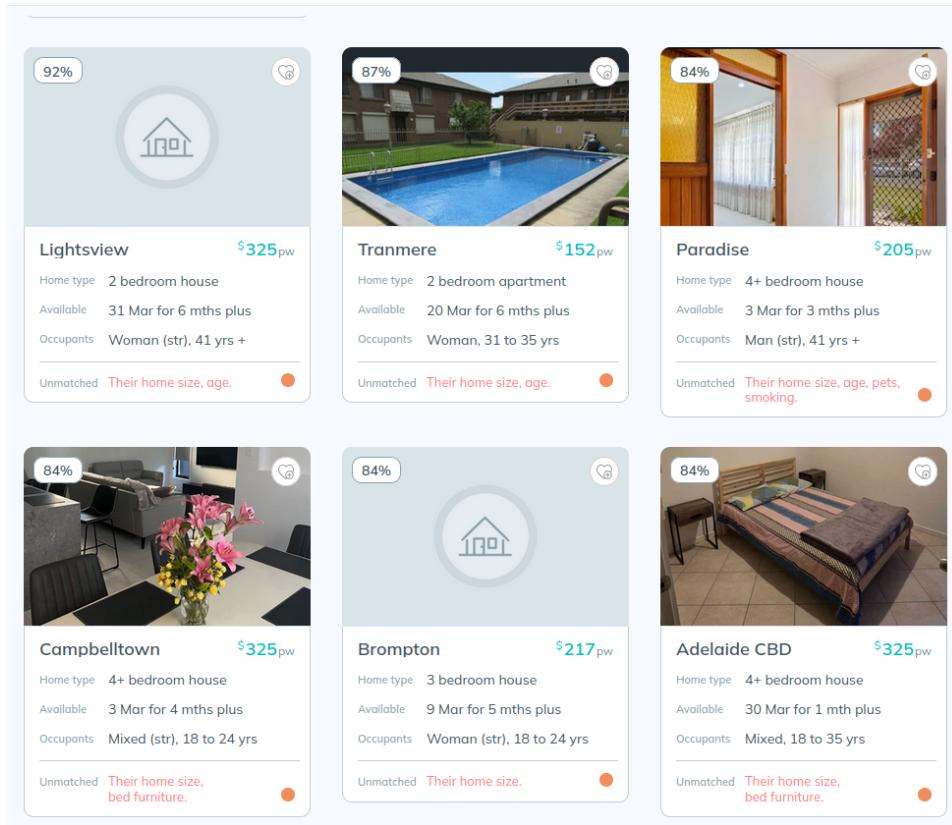


Figura 2.4: Sección de búsqueda de vivienda por afinidad.

Limitaciones

- La aplicación únicamente está disponible en Australia, lo que limita enormemente su uso.
- No ofrece ningún tipo de herramienta para administrar las tareas de la vivienda una vez creado el piso compartido.
- La compatibilidad se basa en aspectos básicos como la edad, el género y si se es fumador, sin un análisis más profundo de la afinidad entre compañeros.

Conclusión

Es una aplicación que ofrece una buena solución para buscar piso en función de afinidad, además el feedback que da al usuario con los porcentajes e indicaciones sobre porque no coinciden son puntos muy favorables. Sin embargo, su única disponibilidad en Australia y la poca complejidad del algoritmo de recomendación provoca que no ofrezca una solución completa.

2.2.2. Gestión de tareas en pisos compartidos

2.2.2.1. Flatify

Flatify [9] ofrece diversas funcionalidades enfocadas en la gestión de un piso compartido, entre las que destacan:

- **Calendario compartido:** Permite añadir eventos para todos los integrantes.
- **Gestión de gastos:** Facilita el registro de pagos y el reparto de costes.
- **Lista de compras:** Los usuarios pueden añadir y marcar productos comprados.
- **Plan de limpieza:** Asigna tareas equitativamente entre los miembros del hogar.
- **Chat grupal:** Comunicación integrada para coordinarse fácilmente.

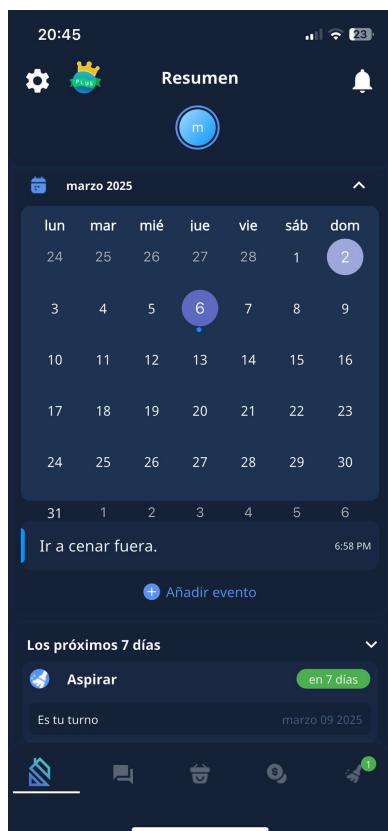


Figura 2.5: Calendario de la aplicación.

Interfaz y usabilidad

La aplicación cuenta con una interfaz limpia e intuitiva, facilitando la navegación. Además, incluye una guía de primeros pasos útil para familiarizarse rápidamente con sus funciones.

Características destacadas

- **Distribución de tareas:** Permite establecer una secuencia de usuarios para rotar las responsabilidades de forma automática, asegurando una distribución justa.
- **Lista de compras:** Funciona correctamente, aunque la funcionalidad de subir facturas para gestionar los pagos está limitada a la versión premium.
- **Calendario:** Útil para eventos, pero no integra las tareas del hogar, lo que limita su funcionalidad.



Figura 2.6: Sección para crear y editar una tarea.

Limitaciones

- Muchas características clave requieren pago o implican ver anuncios para su uso.
- La gestión de pagos y la lista de la compra no están bien sincronizadas, lo que reduce su utilidad.
- El calendario no permite gestionar tareas, desaprovechando su potencial.

Conclusión

A pesar de las limitaciones de su versión gratuita, esta aplicación es la mejor solución encontrada para la gestión de pisos compartidos, ya que ofrece diversas herramientas útiles para la organización del hogar. Sin embargo, presenta algunos inconvenientes como una funcionalidad de calendario que resulta incompleta y la ausencia de herramientas para la creación de hogares basados en afinidades. Estas carencias hacen que la aplicación no sea una opción viable para nuestro problema.

2.3. Tecnologías en el mercado

Para el desarrollo del sistema será necesario apoyarse en diferentes tecnologías que permitan constuir un sistema que responda a las necesidades planteadas. Con el actual auge del desarrollo software, numerosas tecnologías están apareciendo en el mercado para facilitar y mejorar la calidad del desarrollo. A continuación, se realiza un análisis de las tecnologías más relevantes y las ventajas que pueden ofrecer.

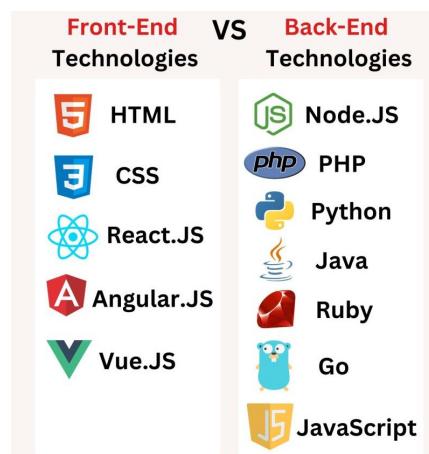


Figura 2.7: Lenguajes de programación y tecnologías presentes en el mercado.

2.3.1. Back-End

Python-Django

Con el aumento de popularidad de Python [13], varios frameworks han aparecido para llevar su funcionamiento al Back-End, como se muestra en la imagen 2.7. Entre ellos, destacan FastAPI, Django y Flask. Estos frameworks aprovechan las ventajas de Python, un lenguaje orientado a la legibilidad y popularizado por el auge de la inteligencia artificial. Django es el más destacable por su robustez y numerosas utilidades. Aunque FastAPI y Flask son excelentes para desarrollar APIs por ser ligeros y especializados, siendo más adecuados para aplicaciones menores, mientras que Django cubre aplicaciones complejas que requieren escalabilidad o seguridad a partir del uso del patrón MVC [4].

NodeJS- Express

NodeJS [24] ha revolucionado el desarrollo Back-End al permitir la ejecución de código JavaScript en el lado del servidor, facilitando así la unificación de tecnologías utilizadas tanto en el Front-End como en el Back-End, como muestra la imagen 2.7, con tecnologías como React en el Front-End y NodeJS en el Back-End. Entre sus características principales se encuentra su gestión de la asincronía mediante un modelo no bloqueante que utiliza un “event loop”, lo cual permite procesar solicitudes sin bloquear el hilo principal, tal como se muestra en la Figura 2.8. Además, ofrece facilidad para trabajar con objetos JavaScript similares a JSON y ha dado lugar a la aparición de frameworks que lo complementan; en este caso, nos centraremos en Express, que facilita la gestión de rutas y la creación de aplicaciones web escalables.

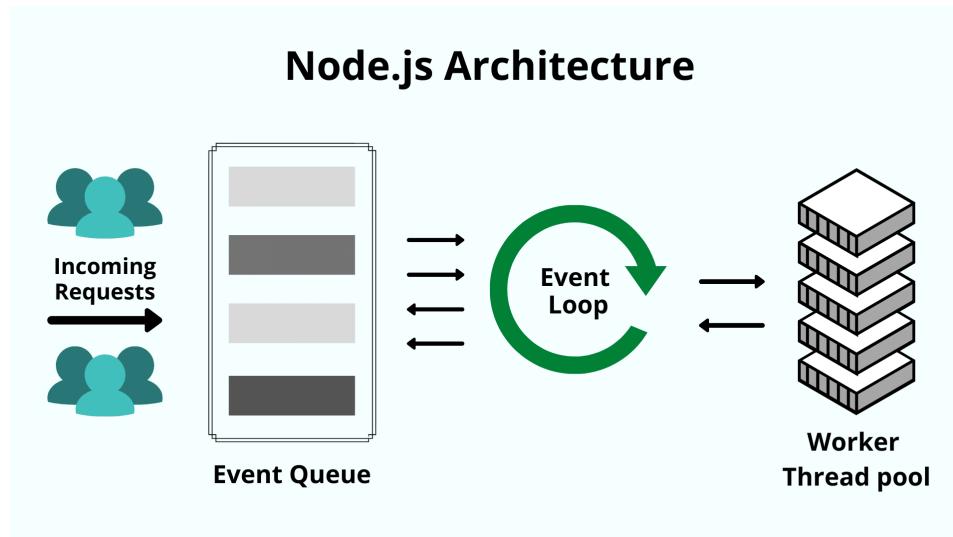


Figura 2.8: Arquitectura NodeJS.

Java

Java, conocido por su fuerte tipado y su compilación, ofrece una solución robusta para el desarrollo Back-End con Spring Boot. Este framework simplifica la gestión de dependencias, proporcionando una estructura más fácil de manejar que la versión anterior de Spring. Es ampliamente utilizado en entornos empresariales debido a su capacidad para soportar arquitecturas complejas, como microservicios o arquitectura hexagonal. Además, Spring Boot promueve prácticas avanzadas de desarrollo como la inyección de dependencias y sus starters, los cuales se pueden apreciar en la imagen 2.9, incluyen desde seguridad hasta manejo reactivo de peticiones. Es una opción robusta para aplicaciones a gran escala.



Figura 2.9: Starters de Spring Boot.

2.3.2. Front-End

En el desarrollo Front-End, JavaScript se ha consolidado como el estándar principal con la aparición de numerosos frameworks. A continuación se van a analizar las opciones más interesantes para la parte visual de la aplicación.

React

React es en la actualidad la biblioteca de desarrollo Front-End más demandada [5]. Su enfoque basado en la creación de componentes reutilizables, el uso de JSX que permite escribir código similar a HTML dentro de archivos JavaScript, como se puede observar en la imagen 2.10, junto con la gestión de estados y hooks, permite estructurar aplicaciones de manera escalable y eficiente de forma sencilla. Además su amplia adopción y su extensa comunidad y documentación lo convierten en una gran herramienta a la hora de crear interfaces complejas y dinámicas.



Figura 2.10: Explicación del concepto JSX.

Angular

Angular es un framework más completo y estructurado en comparación con React, que se centra más en ofrecer al usuario herramientas listas para usar. Sigue también el concepto de componentes pero con un enfoque más estricto en cuanto a arquitectura y estructura del proyecto, lo cual ayuda en la creación de aplicaciones robustas y mantenibles.

TypeScript

El notable crecimiento en la popularidad de TypeScript se debe a que proporciona un sistema de tipado estático sobre JavaScript, lo cual ayuda a reducir errores en tiempo de compilación y mejora la detección de errores y legibilidad del código como se muestra en la imagen 2.11. TypeScript se está convirtiendo en un estándar dentro del desarrollo Front-End profesional debido a la gran ayuda que ofrece para detectar de forma temprana errores y su integración con cualquier tipo de framework moderno como React y Angular.

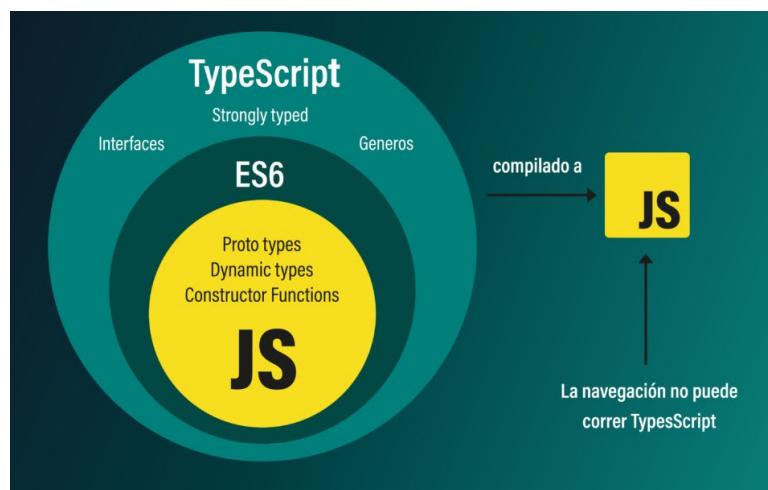


Figura 2.11: Explicación del concepto de TypeScript.

2.3.3. Base de Datos

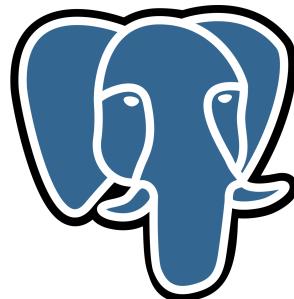


Figura 2.12: Logo PostgreSQL.

PostgreSQL

PostgreSQL 2.12 es una base de datos relacional *open source* que se destaca por su alto rendimiento, escalabilidad y fiabilidad. Garantiza transacciones *ACID* (Atomicidad, Consistencia, Aislamiento y Durabilidad) [16], lo que asegura que las operaciones en la base de datos se realicen de forma segura y consistente. Además, PostgreSQL ofrece una robusta seguridad y es respaldada por una comunidad activa que mantiene una documentación extensa y detallada.



Figura 2.13: Logo MongoDB.

MongoDB

MongoDB 2.13 es una base de datos no relacional que almacena los datos en formato BSON (Binary JSON), lo que le permite manejar grandes volúmenes de datos de manera eficiente. Se caracteriza por su alto rendimiento, flexibilidad y facilidad para integrarse con aplicaciones modernas. A diferencia de las bases de datos relacionales, MongoDB no requiere esquemas estrictos, lo que lo hace ideal para manejar datos no estructurados o semi-estructurados, facilitando la evolución y adaptación de los modelos de datos.

2.4. Conclusión tras estudiar el mercado

Tras analizar diversas alternativas disponibles en el mercado, hemos identificado aplicaciones que abordan parcialmente la problemática planteada. *Flatmate*

Finders facilita la búsqueda de compañeros de piso en función de la afinidad, mientras que *Flatify* destaca en la gestión de la convivencia, ofreciendo herramientas para la organización de tareas, calendario compartido y lista de la compra.

Sin embargo, ninguna de estas soluciones integra ambas funcionalidades en una única plataforma. La falta de una aplicación que combine tanto la búsqueda de vivienda como su gestión diaria con todos los requisitos necesarios, evidencia la necesidad de desarrollar este proyecto, ofreciendo una solución integral que cubra todas las necesidades de la vida en pisos compartidos basado en el *co-housing*. En cuanto a las tecnologías todas las comentadas cumplen con los requisitos mínimos de funcionalidad para poder llevar a cabo el proyecto, por tanto la tecnología escogida dependerá de la arquitectura, preferencia del equipo de desarrollo y complejidad de la aplicación.

Capítulo **3**

Especificación de Requisitos

3.1. Propósito

En este capítulo se definirán las distintas características, funcionalidades y restricciones que debe cumplir la aplicación que se va a desarrollar. Se redactará un conjunto de requisitos esenciales que guiarán el proceso de creación del sistema. Además, se describirán los requisitos de las interfaces visuales, estableciendo un patrón a seguir durante el desarrollo de la aplicación.

3.2. Alcance

El sistema tiene como objetivo proporcionar a los usuarios una solución eficaz para la búsqueda y gestión de viviendas compartidas, alineada con los principios del *co-housing*. Este proyecto abordará las siguientes tareas principales:

- Registro y gestión de usuarios.
- Registro y gestión de comunidades.
- Recomendación de viviendas por afinidad.
- Gestión de tareas y actividades.

Los datos serán almacenados en bases de datos, y se hará especial énfasis en la escalabilidad y robustez de la aplicación, con el objetivo de garantizar un rendimiento escalable y efectivo.

3.3. Definiciones, siglas y abreviaturas

- **Usuario Buscador:** Persona que se da de alta en la plataforma con el rol de *Buscador*. Su objetivo es buscar comunidades y unirse a ellas. Puede enviar solicitudes e interactuar con el gestor de tareas.

- **Usuario Ofertante:** Persona que registra una comunidad en la plataforma, introduciendo datos específicos como fotos, metros cuadrados, número de habitaciones, entre otros. Se le ofrecerá funcionalidad para administrar la comunidad, definiendo sus criterios de afinidad y gestionando las solicitudes de unión, aceptándolas o rechazándolas.
- **Comunidad:** Concepto que hace referencia a la vivienda compartida entre varios integrantes. Esta podrá ser gestionada a través de actividades y tareas que se propondrán y deberán ser realizadas por los integrantes de la comunidad.
- **ShareSpace:** Nombre por el que será identificado el proyecto, por tanto a partir de ahora su uso referencia a la aplicación.

3.4. Perspectiva del producto

Este sistema tiene como objetivo ofrecer una solución eficiente que adapte las distintas necesidades de los usuarios. A medida que la aplicación vaya creciendo se podría integrar nueva funcionalidad que aporte valor añadido a la experiencia del usuario. De forma inicial la aplicación se construirá en torno a distintos servicios que encapsularán parte de la lógica de negocio. Como se puede ver en la imagen 3.1 estos componentes se comunicarán con la interfaz gráfica, facilitando la experiencia de usuario de manera sencilla y accesible.

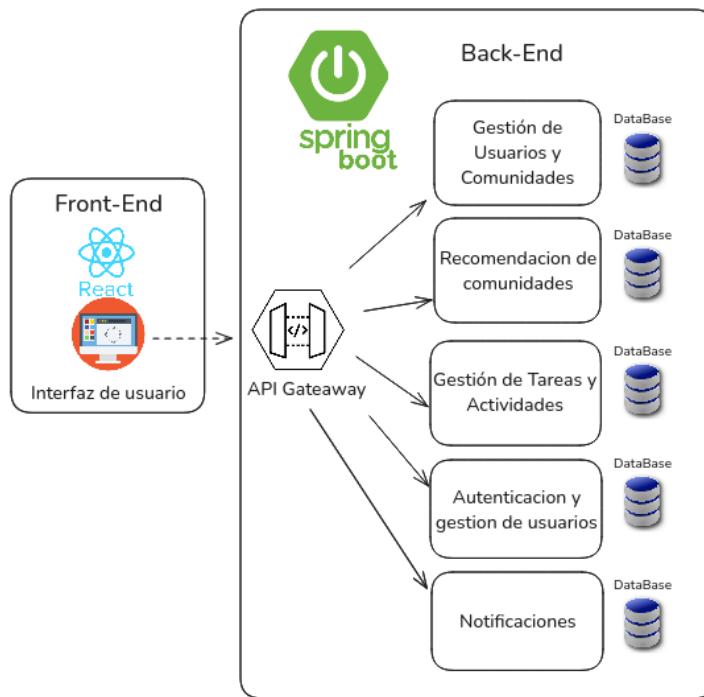


Figura 3.1: Diagrama interacción entre componentes.

3.5. Restricciones

Las restricciones principales de la aplicación para que la experiencia sea la adecuada tanto en móvil como en ordenador serán las siguientes:

- **Hardware:**

- Se recomienda tener al menos 8 GB de RAM para un rendimiento óptimo cuando se usen múltiples pestañas o aplicaciones al mismo tiempo.

- **Navegadores Web:**

- Google Chrome versión 90 o superior.
- Mozilla Firefox versión 80 o superior.
- Safari 14.0 o superior.
- Microsoft Edge versión 90 o superior.

- **Sistemas Operativos Compatibles:**

- **Windows:** Windows 10 o superior.
- **macOS:** macOS 10.15 o superior.

- **Linux:** Cualquier distribución moderna de Linux, como Ubuntu 20.04 o superior.

3.6. Funciones del producto

El sistema contará con una serie de funcionalidades clave para garantizar su cumplimiento del objetivo establecido. A continuación, se detallan las principales funciones que ofrecerá la plataforma:

- **Registro de usuarios y comunidades:** El sistema permitirá registrar tanto a usuarios como a comunidades, gestionando sus respectivos datos. Los usuarios podrán crear su perfil y las comunidades podrán ser configuradas con detalles como ubicación, número de habitaciones, y características específicas.
- **Búsqueda de comunidad:** La plataforma ofrecerá recomendaciones de comunidades basadas en afinidad y preferencias del usuario. Además, permitirá enviar solicitudes de unión a los propietarios de las comunidades, estableciendo una comunicación directa entre ambos para facilitar el proceso de integración.
- **Gestión de tareas:** Los usuarios podrán registrar nuevas tareas, asignarlas a la comunidad y consultar el estado de las que tienen que realizar. Las tareas estarán integradas en un calendario compartido, asegurando que todos los miembros estén al tanto de las actividades pendientes y los plazos de realización.
- **Gestión de eventos y actividades:** Se podrán crear y gestionar eventos dentro de la comunidad, desde reuniones hasta actividades sociales, fomentando la convivencia.

3.7. Redacción de requisitos

Los requisitos van a describir de forma clara y directa las características que el sistema debe proporcionar. Vamos a encontrar tres secciones generales:

- **Requisitos funcionales:** Definen las funciones que la aplicación debe ser capaz de realizar.
- **Requisitos no funcionales:** Evaluarán cómo debe comportarse el sistema, dentro de ellos se definirán los requisitos de seguridad.
- **Requisitos de interfaces:** Especificarán los criterios a seguir en el apartado visual, el análisis de las interfaces de usuarios se va a realizar a partir de dos bocetos.

Los requisitos funcionales se agruparán en función de grandes bloques de funcionalidad.

3.8. Requisitos funcionales

3.8.1. Bloque Gestión de Usuarios y Autenticación

- **RF-01: Registro de Usuarios** Los usuarios podrán registrarse como buscadores u ofertantes en la aplicación mediante un formulario que recopile información básica. Los campos mínimos requeridos son:
 - Nombre
 - Correo Electrónico
 - Contraseña
 - Localización
 - Afinidad
- **RF-02: Inicio de Sesión** Los usuarios podrán iniciar sesión introduciendo su correo electrónico y contraseña previamente registrados. En caso de error en los datos, el sistema notificará al usuario con un mensaje adecuado.
- **RF-03: Gestión de Perfiles** Los usuarios deberán poder visualizar y modificar sus datos personales, como la localización y preferencias de afinidad.
- **RF-04: Eliminación de Usuario** Los usuarios podrán borrar su cuenta del sistema, eliminando toda información asociada con ellos.

3.8.2. Bloque Gestión de Comunidades

- **RF-05: Creación de Comunidades** Los ofertantes podrán crear nuevas comunidad mediante un formulario que incluirá:
 - Imágenes representativas
 - Descripción detallada de la comunidad
 - Criterios de afinidad para los miembros
 - Número de integrantes deseado
- **RF-06: Unión a Comunidades** Los propietarios deben recibir una notificación cuando un usuario solicite unirse a una comunidad pudiendo ver su información y aceptarlo o rechazarlo.
- **RF-07: Petición de Unión** Los usuarios podrán solicitar unirse a una comunidad concreta mediante un proceso de petición sencilla.

- **RF-08: Notificación de respuesta a Unión** Los usuarios serán notificados con la decisión del propietario de la comunidad a la que han pedido unirse.
- **RF-09: Eliminación de Comunidad** Los propietarios deben poder eliminar la comunidad que han registrado previamente, eliminando toda la información contenida en el sistema.
- **RF-10: Gestión de Comunidad** Los usuarios ofertantes podrán modificar los datos del perfil de la comunidad.

3.8.3. Bloque Gestión de Tareas

- **RF-11: Creación de Tareas/Eventos** Los usuarios podrán crear tareas/eventos comunitarios mediante un formulario sencillo donde introducirán mínimo: el nombre, una descripción breve y la frecuencia con la que se deberá realizar.
- **RF-12: Gestión del Calendario** El sistema debe mostrar un calendario personalizado a cada usuario, que incluirá:
 - Tareas asignadas
 - Eventos destacados
- **RF-13: Visibilidad de las Tareas** Los usuarios podrán ver la descripción detallada de cada tarea que se les haya asignado, incluyendo fechas de realización y compañeros con los que tienen que colaborar.
- **RF-14: Marcar Tareas** Los usuarios deberán poder marcar las tareas como completadas y consultar el estado de las tareas pendientes con su respectiva fecha límite.
- **RF-15: Reparto de Tareas** El sistema distribuirá las tareas semanalmente de manera justa, asignando responsabilidades en un orden secuencial.
- **RF-16: Flexibilidad en la Asignación de Tareas** Los usuarios podrán distribuir de forma libre sus tareas asignadas a lo largo de la semana.

3.8.4. Bloque Recomendación de Comunidades

- **RF-17: Recomendación de Comunidades** El sistema recomendará comunidades que coincidan con los criterios tanto de afinidad como de vivienda introducidos por los usuarios.
- **RF-18: Guardar Comunidades de Interés** El usuario podrá guardar sus comunidades de interés utilizando un botón de “guardar” o “me gusta”.

- **RF-19: Grado de afinidad** El sistema mostrará a los usuarios el grado de afinidad que existe entre una comunidad y ellos con un porcentaje sobre cien.

3.9. Requisitos No Funcionales

- **RNF-01: Notificaciones** Las notificaciones deben ser rápidas y entregarse a los usuarios en un período de tiempo corto (entorno a uno o dos segundos).
- **RNF-02: Mantenibilidad y Escalabilidad** El sistema debe estar diseñado para ser fácil de mantener, pudiendo escalar a medida que crecen los usuarios y las funcionalidades.
- **RNF-03: Alta Disponibilidad** El sistema debe permanecer operativo en todo momento, asegurando que si aparece un fallo en alguna parte del sistema el resto de servicios seguirán operativos.
- **RNF-04: Soporte de Carga** El sistema debe soportar un volumen de usuarios considerable de forma simultánea, asegurando que el servicio esté disponible ante un flujo alto de usuarios.
- **RNF-05: Mantenimiento** El sistema debe permitir al administrador realizar tareas de mantenimiento sin afectar el servicio a los usuarios.
- **RNF-06: Gestión de Usuarios** El sistema debe poder permitir al administrador gestionar los usuarios pudiendo activar/desactivar cuentas y modificar roles.
- **RNF-07: Monitoreo** El sistema debe alertar al usuario en caso de caídas de servicio que afecten a los usuarios.

3.9.1. Requisitos de Seguridad

- **RNF-08: Autorización y autenticación** La aplicación deberá contar con un sistema de autenticación y autorización seguro que garantice el control de los accesos.
- **RNF-09: Seguridad en Base de Datos** El sistema debe implementar mecanismos de seguridad que protegan a las bases de datos de accesos no autorizados.
- **RNF-10: Protección contra Inyecciones SQL** El sistema debe prevenir ataques de inyección SQL mediante el uso de validaciones adecuadas en los datos de entrada.

- **RNF-11: Protección de datos** El sistema debe asegurar el cumplimiento de las normativas vigentes de [protección de datos de la Unión Europea](#), para garantizar un uso adecuado de los datos personales.

3.9.2. Requisitos de Interfaces Externas

Dentro de las interfaces externas distinguimos dos grandes bloques, las APIs y las Interfaces de usuario:

3.9.2.1. APIs

Cada servicio del sistema expondrá sus operaciones a través de una API, lo que permitirá una comunicación eficiente entre los módulos.

3.9.2.2. Interfaces de Usuario

La interfaz de usuario permitirá un uso de la funcionalidad ofrecida por el sistema claro y sencillo.

Cada interfaz va a cumplir una serie de criterios de usabilidad y una estética uniforme. A modo de ejemplo se va a presentar dos bocetos esquematizados para presentar el flujo de diseño a seguir.

Página de Bienvenida

La "Landing Page" tiene como objetivo ser la puerta de entrada a la aplicación para el usuario, por lo tanto tiene que ser directa y llamativa para captar su atención.

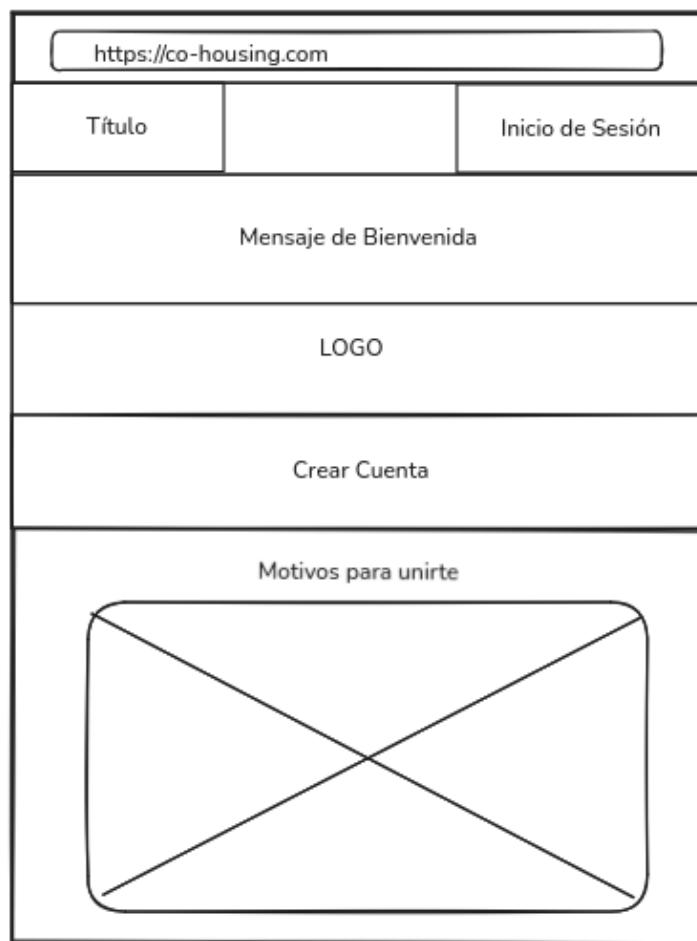


Figura 3.2: Boceto "Landing Page".

Descripción:

La imagen 3.2 muestra la página a la que accede el usuario mediante el buscador al introducir el nombre de la aplicación, esta tiene como objetivo mostrar una pequeña presentación de la app y permitir la realización de una acción muy concreta, en este caso registrarse o iniciar sesión.

Página Buscador

La página del buscador tiene como objetivo poner en contacto a los usuarios con comunidades afines por lo que se debe mostrar la información de manera clara y bien estructurada.

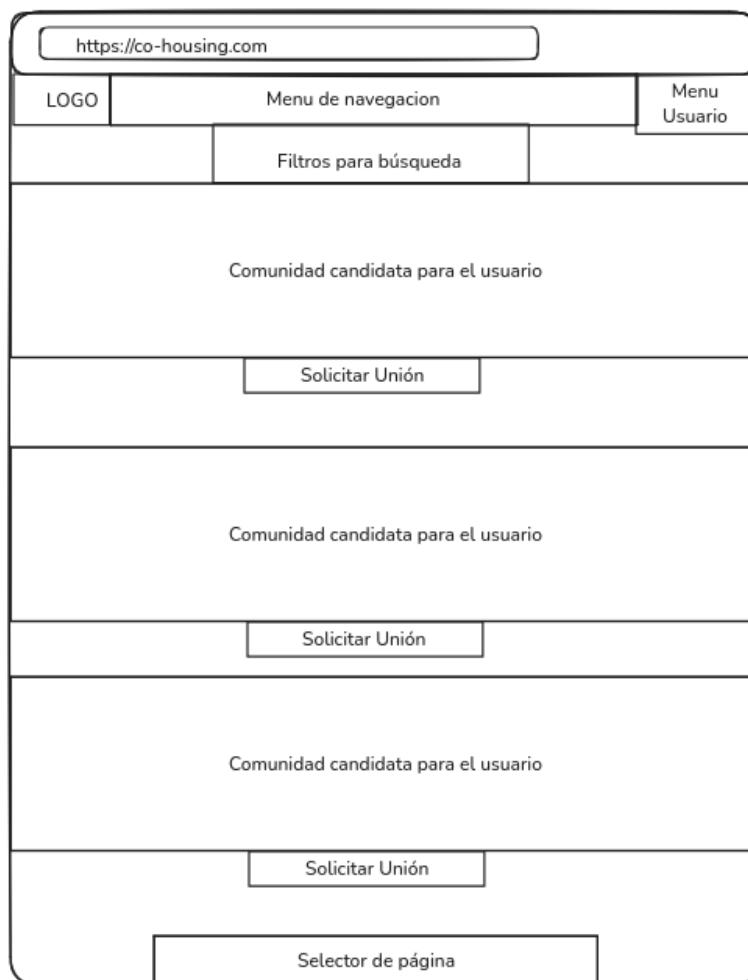


Figura 3.3: Boceto Página Buscador.

Descripción

La imagen 3.3 muestra la página que visualizará el usuario cuando accede a la sección "Buscador" de la aplicación, en ella aparecen las distintas comunidades sugeridas y un botón para solicitar la unión, además cuenta con un menú de navegación y un menú para el usuario.

Capítulo 4

Planificación del Proyecto

4.1. Propósito

En este capítulo se llevará a cabo la planificación de las diferentes etapas que compondrán el desarrollo del proyecto. Para la realización de esta planificación de la manera más profesional y cercana a la realidad del desarrollo software moderno, se van a aplicar algunos conceptos propios de **Scrum** [21] como metodología ágil 4.1 .



Figura 4.1: Metodología ágil Scrum.

Tras estudiar los distintos principios en los que se basa Scrum se han identificado varios elementos clave entre los que destacan: la necesidad de participación activa por parte del cliente y la existencia tanto de un equipo de desarrollo como de profesionales con roles específicos tales como *Scrum Master* y *Product Owner*. Sin embargo, en el contexto de la realización de un Trabajo de Fin de

Grado, estas condiciones resultan inviables debido al carácter individual del mismo. No se cuenta ni con un equipo de desarrollo, ni existe ningún integrante con capacidad para realizar dichos roles. Además la participación del cliente es totalmente inexistente.

A pesar de estas limitaciones, y con el objetivo de que el proyecto tenga un desarrollo lo más profesional y realista posible, se adaptarán varios elementos presentes en la metodología Scrum en la planificación. Estos elementos a considerar van a ser:

- **Iteraciones:** El proyecto se desarrollará a través de ciclos iterativos en los que se encapsulan bloques importantes de funcionalidad, se asemejará en concepto a los Sprint de Scrum pero sin la necesidad de terminar siempre con un objeto mínimamente viable evaluado por el cliente.
- **Historias de Usuario:** Se usarán las historias de usuario para describir funcionalidades específicas del sistema, desde la perspectiva del usuario final.
- **Incremento de Funcionalidad:** En cada iteración se conseguirá un incremento funcional del proyecto, que podrá ser revisado y mejorado en las siguientes iteraciones.
- **Adaptabilidad:** La planificación de cada iteración podrá ser modificada en función del progreso realizado y las necesidades detectadas.
- **Revisión:** Al final de cada iteración se realizará un análisis para identificar qué elementos se pueden mejorar y cuáles se están haciendo bien.

Para mayor referencia, se puede consultar la guía oficial de Scrum [15]. Una vez establecidas las bases que van a constituir la planificación del proyecto, se va a proceder a realizar un análisis de la velocidad de desarrollo.

4.2. Cálculo de la Velocidad de Desarrollo

Para realizar la estimación de la velocidad de desarrollo de las distintas funcionalidades del sistema, vamos primero a analizar las condiciones iniciales:

- Contamos con un **único desarrollador**.
- Se dedicarán **2.5 horas efectivas** al día al proyecto.
- Cada iteración tendrá una duración de **3 semanas**, siendo un total de **4 iteraciones**.
- Se trabajará un total de 5 días efectivos a la semana al proyecto.

A continuación, estimaremos el tiempo medio de desarrollo de un **Punto de Historia de Usuario**, que será la unidad de medida del esfuerzo requerido para cada tarea.

Inicialmente, asignaremos un total de 3 horas de trabajo a cada Punto de Historia de Usuario. Con esto establecido, el cálculo de las horas efectivas por iteración es el siguiente:

$$\text{Horas efectivas por iteración} = 3 \text{ semanas} \times 5 \text{ días por semana} \times 3 \text{ horas diarias}$$

$$= 3 \times 5 \times 3 = 45 \text{ horas por iteración}$$

Si consideramos que cada Punto de Historia de Usuario requiere **3 horas de trabajo**, entonces la cantidad de Puntos de Historia de Usuario que pueden desarrollarse en una iteración será:

$$\frac{45 \text{ horas}}{2,5 \text{ horas por punto}} = 18 \text{ puntos de historia por iteración}$$

Por tanto, de manera inicial tomaremos como referencia que en cada iteración se podrán desarrollar **18 Puntos de Historia de Usuario**.

Este cálculo podrá verse modificado en función del resultado real de cada iteración, por lo que será revisado y ajustado si es necesario.

Finalmente se muestra una tabla resumen con los valores más relevantes de esta sección.

Parámetro	Valor
Desarrolladores	1
Horas efectivas diarias	2.5 horas
Días efectivos por semana	5 días
Duración de cada iteración	3 semanas (21 días)
Horas efectivas por iteración	45 horas
Tiempo estimado por Punto de Historia	2,5 horas
Velocidad total por iteración	18 PHUs

Tabla 4.1: Resumen de parámetros para la estimación de velocidad de desarrollo

4.3. Listado de historias de usuario

A continuación, se muestra una lista con las historias de usuario que describirán la funcionalidad del proyecto, a cada una se le ha asignado una serie de puntos de historia de usuario (PHU) en función de su aparente complejidad.

Código	Título	PHU
HU01	Como usuario buscador, quiero poder darme de alta en la aplicación para encontrar una comunidad a la que unirme.	3
HU02	Como usuario buscador, quiero poder darme de baja de la aplicación, eliminando todos mis datos.	2
HU03	Como usuario buscador, quiero poder configurar mis gustos y preferencias para recibir recomendaciones de comunidades afines.	2
HU04	Como usuario buscador, quiero que el sistema me recomiende comunidades en función del grado de afinidad que compartimos.	3
HU05	Como usuario buscador, quiero poder dar me gusta a los pisos que me interesan para guardarlos en una colección.	1
HU06	Como usuario buscador, quiero poder consultar todas las comunidades a las que les he dado me gusta.	2
HU07	Como usuario buscador, quiero visualizar en detalle la información de una comunidad (integrantes, preferencias, imágenes, localización y precio).	2
HU08	Como usuario buscador, quiero poder solicitar unirme a una comunidad que me interese, para pasar a formar parte de ella.	3
HU09	Como usuario buscador, quiero recibir notificaciones cuando acepten o rechacen mi solicitud de unión a una comunidad.	1
HU10	Como usuario buscador, quiero establecer filtros en la búsqueda de vivienda (localización, número de habitaciones, precio y espacio).	2
HU11	Como usuario buscador, quiero que el sistema me muestre las comunidades ordenadas en función del grado de afinidad.	2
HU12	Como usuario ofertante, quiero poder darme de alta en el sistema para buscar integrantes con los que compartir mi comunidad.	3
HU13	Como usuario ofertante, quiero poder darme de baja en el sistema, eliminando todos mis datos contenidos.	1
HU14	Como usuario ofertante, quiero poder dar de alta una comunidad, introduciendo nombre, integrantes, localización, precio, criterios de afinidad y fotos.	3
HU15	Como usuario ofertante, quiero consultar el perfil de cada usuario que envía una solicitud de unión, para conocer sus preferencias y el grado de afinidad.	2
HU16	Como usuario ofertante, quiero poder aceptar o rechazar usuarios que solicitan unirse a mi comunidad.	2
HU17	Como usuario ofertante, quiero recibir una notificación cuando un usuario solicite unirse a mi comunidad.	1

Capítulo 4. Planificación del Proyecto

HU18	Como usuario ofertante, quiero poder modificar la información de mi comunidad para mantenerla actualizada.	2
HU19	Como usuario ofertante, quiero poder eliminar una comunidad, borrando toda la información contenida en el sistema.	1
HU20	Como usuario ofertante, quiero poder eliminar un usuario de mi comunidad en caso de incumplimiento de normas.	1
HU21	Como usuario buscador u ofertante, quiero poder consultar el calendario de tareas y actividades del piso.	3
HU22	Como usuario buscador u ofertante, quiero visualizar en el calendario las tareas o actividades previstas en un día concreto, con detalles como hora, descripción y participantes.	3
HU23	Como usuario buscador u ofertante, quiero que tanto las actividades como las tareas se añadan al calendario automáticamente para que mis compañeros las vean.	2
HU24	Como usuario buscador u ofertante, quiero poder añadir eventos al calendario para que mis compañeros puedan consultarlos fácilmente.	2
HU25	Como usuario buscador u ofertante, quiero poder registrar tareas a realizar mediante un formulario donde especifique dificultad, tiempo para realizarla y descripción.	3
HU26	Como usuario buscador u ofertante, quiero poder registrar actividades a realizar mediante un formulario indicando lugar, fecha, hora y descripción.	3
HU27	Como usuario buscador u ofertante, quiero recibir una notificación cuando se asignen las tareas semanales.	2
HU28	Como usuario buscador u ofertante, quiero distribuirme las tareas libremente a lo largo de la semana según mis necesidades.	3
HU29	Como usuario buscador u ofertante, quiero recibir una notificación con una hora de antelación antes de que termine el plazo de una tarea.	2
HU30	Como usuario buscador u ofertante, quiero que al final de la semana se genere un resumen con las tareas completadas por cada integrante.	2
HU31	Como usuario buscador u ofertante, quiero que si no distribuyo manualmente mis tareas en un tiempo límite, el sistema las asigne automáticamente.	2
HU32	Como usuario buscador u ofertante, quiero poder reorganizar las tareas asignadas para poder cumplir con contratiempos personales.	2
HU33	Como usuario buscador u ofertante, quiero que la asignación de tareas semanales sea cíclica, para que todos los integrantes hagan todas las tareas equitativamente.	2

HU34	Como usuario buscador u ofertante, quiero que los usuarios que no realicen sus tareas sean penalizados, reasignándoselas con un plazo de tiempo limitado.	2
------	---	---

4.4. Descripción de las iteraciones

El desarrollo del proyecto se estructurará en cuatro iteraciones, permitiendo descomponer el trabajo en grandes grupos de tareas bien definidos. En cada iteración se llevará a cabo la realización de un bloque completo de funcionalidad de manera que se vaya iterando hacia un producto más completo.

4.4.1. Duración y Esfuerzo

Para establecer las iteraciones del proyecto, se ha planteado un desarrollo basado en cuatro iteraciones de una duración de tres semanas cada una. De este modo, el esfuerzo total requerido por el proyecto se representa con **72 Puntos de Historia de Usuario (PHUs)**, asignando una velocidad de desarrollo previamente calculada de **18 PHUs** por iteración.

4.4.2. Distribución de Funcionalidades

Dado que en cada iteración se completarán 18 PHUs, en la planificación se ha buscado priorizar aquellas historias de usuario de mayor complejidad y que aportan más valor al sistema en las primeras iteraciones. Este enfoque garantiza que, en caso de retrasos, haya tiempo suficiente para asegurar una funcionalidad básica.

- **Iteración 0 (Documentación e investigación):** Esta iteración, visible en el diagrama de Gantt 4.2, abarca todo el proceso previo de documentación realizado hasta el momento, así como la investigación y el aprendizaje de las tecnologías necesarias para el desarrollo, este proceso de documentación pese a que se seguirá realizando en las siguientes iteraciones en esta ha tenido un papel clave por lo que no seguirá la misma estructura ni temporal ni de historias de usuario que las demás.
- **Iteración 1 (Registro de usuarios y configuración de comunidades, Diagrama de Gantt 4.3):**
 - Objetivo: Implementar la lógica básica del sistema, permitiendo el registro de usuarios buscadores como ofertantes, la configuración de preferencias en su perfil y la creación de comunidades. Se continua con la documentación del proyecto.
 - Historias de usuario: HU01, HU02, HU03, HU12, HU13, HU14, HU18, HU19, HU20.

- Puntos de Historia asignados: 18 PHUs.
- **Iteración 2** (*Búsqueda de piso y algoritmo de recomendación, Diagrama de Gantt 4.4*):
 - Objetivo: Implementar la búsqueda de pisos mediante filtros, las solicitudes de unión a comunidades y el algoritmo de recomendación basado en afinidad. Se continua con la documentación del proyecto.
 - Historias de usuario: HU04, HU07, HU08, HU09, HU10, HU11, HU15, HU16, HU17.
 - Puntos de Historia asignados: 18 PHUs.
- **Iteración 3** (*Gestión de tareas y calendario, Diagrama de Gantt 4.5*):
 - Objetivo: Desarrollar la lógica relacionada con la gestión de tareas y actividades, desde su creación, modificación y asignación, incluyendo el calendario compartido. Se continua con la documentación del proyecto.
 - Historias de usuario: HU21, HU22, HU23, HU24, HU25, HU26, HU33.
 - Puntos de Historia asignados: 18 PHUs.
- **Iteración 4** (*Funcionalidades adicionales, Diagrama de Gantt 4.6*):
 - Objetivo: Incorporar funcionalidades complementarias para mejorar la experiencia del usuario. Se continua con la documentación del proyecto.
 - Historias de usuario: HU05, HU06, HU27, HU28, HU29, HU30, HU31, HU32, HU34.
 - Puntos de Historia asignados: 18 PHUs.

4.5. Diagrama de Gantt

Para poder visualizar de forma esquematizada el contenido de las distintas iteraciones que va a tener el proyecto se ha realizado un diagrama de Gantt.

En el diagrama podremos analizar diversas características clave, como la duración de cada iteración, el tiempo requerido para completar cada historia de usuario y el avance progresivo del proyecto.

Capítulo 4. Planificación del Proyecto

Diagrama de Gantt TFG

GRÁFICO GANTT SIMPLE
<https://www.vertex42.com>

Proyecto TFG
Alonso Doña Martínez



Figura 4.2: Diagrama de Gantt iteración 0.

Diagrama de Gantt TFG

GRÁFICO GANTT SIMPLE de Vertex42
<https://www.vertex42.com/ExcelTemplates/gantt-charts.html>

Proyecto TFG
Alonso Doña Martínez

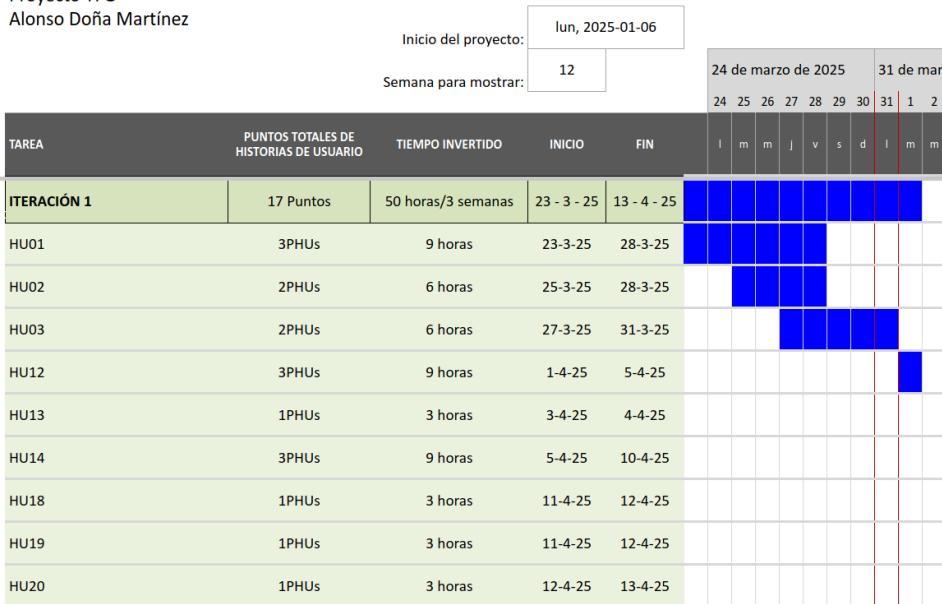


Figura 4.3: Diagrama de Gantt iteración 1.

Capítulo 4. Planificación del Proyecto

Diagrama de Gantt TFG

Proyecto TFG
Alonso Doña Martínez

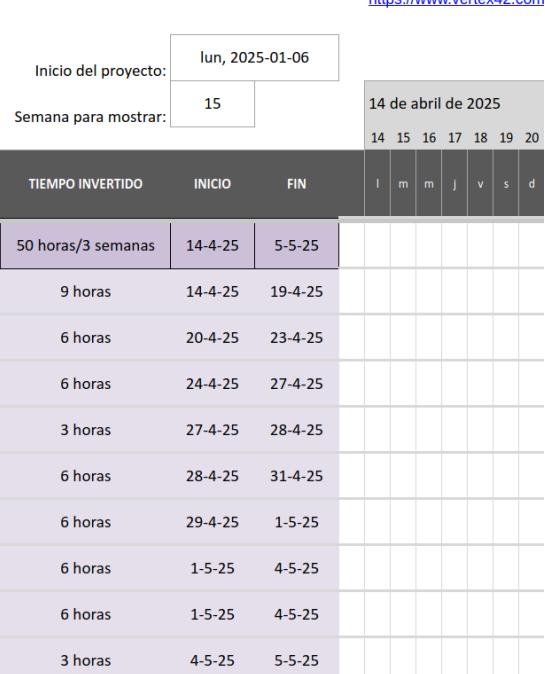


Figura 4.4: Diagrama de Gantt iteración 2.

Diagrama de Gantt TFG

Proyecto TFG
Alonso Doña Martínez

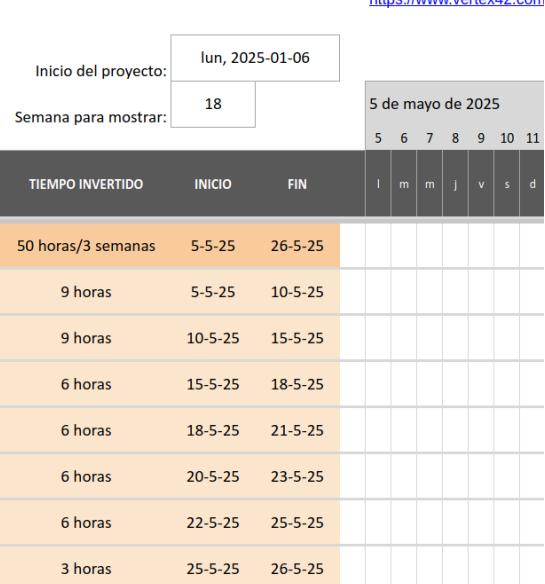


Figura 4.5: Diagrama de Gantt iteración 3.

Diagrama de Gantt TFG

GRÁFICO GANTT SIMPLE
<https://www.vertex42.com>

Proyecto TFG
 Alonso Doña Martínez

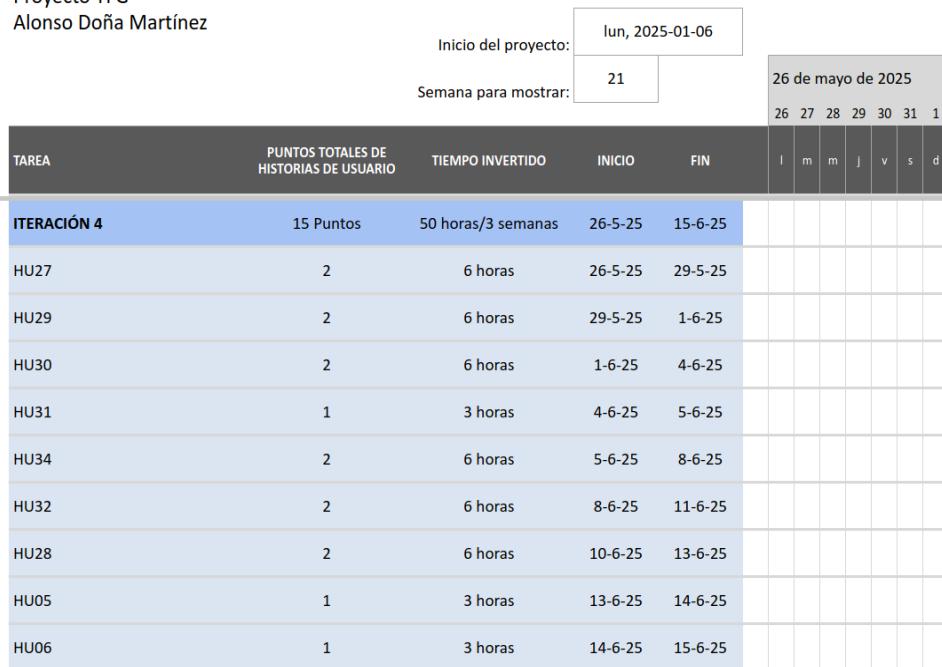


Figura 4.6: Diagrama de Gantt iteración 4.

4.6. Estimación de costes

Para el cálculo de la estimación de costes del proyecto vamos a desglosar el presupuesto en varias secciones.

4.6.1. Costes de Herramientas Utilizadas

El desarrollo del proyecto se ha realizado utilizando un ordenador portátil HP Pavilion adquirido en el año 2021, por un coste de **782 euros**. Se ha decidido amortizar este equipo con una tasa del 20% por año, lo que resulta en un coste total de:

$$\text{Coste anual} = 782 \times 0,20 = 156,4 \text{ euros}$$

$$\text{Resto total} = 156,4 \times 4 = 625,6 \text{ euros}$$

Por tanto, si restamos este coste al valor total del equipo, obtenemos:

$$\text{Coste final: } 782 - 625,6 = 156,4 \text{ euros}$$

4.6.2. Coste de Desarrollo

Para calcular el coste de desarrollo, utilizamos la velocidad estimada en el capítulo 4.2. En él se determinó que cada iteración contará con un total de 45 horas de trabajo, contando con un total de 4 iteraciones, además de la fase de investigación y aprendizaje previa a la que se le ha asignado un total de 100 horas.

El total de horas dedicadas será:

$$4 \times 45 + 100 = 280 \text{ horas}$$

Si consideramos que el sueldo medio anual de un ingeniero de software en España en 2025, es de 32,850 euros, según [Glassdoor](#), y suponiendo una jornada laboral de 1,800 horas anuales y unos 2,190 euros netos al mes, el coste por hora se obtiene de la siguiente forma:

$$\text{Coste por hora} = \frac{2190 \text{ euros}}{40 \text{ horas/semana} \times 4 \text{ semanas/mes}} = \frac{2190}{160} = 13,68 \text{ euros/hora}$$

A continuación, si se han invertido aproximadamente 280 horas en el desarrollo, el coste total de desarrollo sería:

$$\text{Coste de desarrollo} = 280 \text{ horas} \times 13,68 \text{ euros/hora} = 3830,4 \text{ euros}$$

Así, el coste total de desarrollo, basado en las horas dedicadas y el coste por hora de un ingeniero de software, sería de 3830,4 euros.

4.6.3. Costes de Infraestructura

Se ha optado por el uso de tecnologías de código libre y herramientas gratuitas, lo que permite que todos los costes asociados a la infraestructura de desarrollo sean nulos.

4.6.4. Coste de Mantenimiento

El mantenimiento de la aplicación incluirá la corrección de errores y la implementación de nuevas funcionalidades. Se estima que representará el 25% del total de horas dedicadas al desarrollo del proyecto.

Por tanto, el número de horas de mantenimiento se calcula como:

$$Horas_{mantenimiento} = 0,25 \times 280 \text{ (horas totales dedicadas al desarrollo)} = 70 \text{ horas}$$

El coste total de mantenimiento será:

$$Coste_{mantenimiento} = 70 \times 13,68 = 957,6 \text{ euros}$$

4.6.5. Resumen de Costes

En la Tabla 4.3 se presentan los costes asociados al desarrollo del proyecto. Dichos costes suman un total final de 5132,81€

Concepto	Coste (euros)
Amortización del portátil	156,4€
Coste de desarrollo	3830,4€
Coste de infraestructura	0€
Coste de mantenimiento	957,6€
Total	4944,4

Tabla 4.3: Resumen de costes del proyecto

Capítulo 5

Análisis

5.1. Objetivo

En este capítulo se realizará un análisis de la funcionalidad a implementar en el sistema. Para ello, se tomarán las historias de usuario definidas con anterioridad, desarrollándolas en mayor detalle a partir de sus respectivas tarjetas. Esto nos permitirá obtener una comprensión profunda de la funcionalidad a implementar, así como de los requisitos que deben cumplir para considerarse completadas.

Finalmente, se realizará un diagrama de análisis que permitirá presentar de forma inicial la estructura de clases, atributos y relaciones entre entidades que compondrán el proyecto.

5.2. Product backlog

En primer lugar se presenta el *product backlog* [1], una lista priorizada de historias de usuario. Este establecerá en primer lugar aquellas historias que aporten mayor valor a la lógica de negocio del sistema, identificando aquellas que se deben entregar en primer lugar. Esto será fundamental para asegurar que el desarrollo se centre en los objetivos más relevantes desde el inicio del proyecto.

Código	Título	Prioridad
HU01	Como usuario buscador, quiero poder darme de alta en la aplicación para encontrar una comunidad a la que unirme.	Alta
HU02	Como usuario buscador, quiero poder darme de baja de la aplicación, eliminando todos mis datos.	Alta

Código	Título	Prioridad
HU03	Como usuario buscador, quiero poder configurar mis gustos y preferencias para recibir recomendaciones de comunidades afines.	Alta
HU04	Como usuario buscador, quiero que el sistema me recomiende comunidades en función del grado de afinidad que compartimos.	Alta
HU08	Como usuario buscador, quiero poder solicitar unirme a una comunidad que me interese, para pasar a formar parte de ella.	Alta
HU12	Como usuario ofertante, quiero poder darme de alta en el sistema para buscar integrantes con los que compartir mi comunidad.	Alta
HU13	Como usuario ofertante, quiero poder darme de baja en el sistema, eliminando todos mis datos contenidos.	Alta
HU14	Como usuario ofertante, quiero poder dar de alta una comunidad, introduciendo nombre, integrantes, localización, precio, criterios de afinidad y fotos.	Alta
HU19	Como usuario ofertante, quiero poder eliminar una comunidad, borrando toda la información contenida en el sistema.	Alta
HU07	Como usuario buscador, quiero visualizar en detalle la información de una comunidad (integrantes, preferencias, imágenes, localización y precio).	Media
HU10	Como usuario buscador, quiero establecer filtros en la búsqueda de vivienda (localización, número de habitaciones, precio y espacio).	Media
HU11	Como usuario buscador, quiero que el sistema me muestre las comunidades ordenadas en función del grado de afinidad.	Media
HU15	Como usuario ofertante, quiero consultar el perfil de cada usuario que envía una solicitud de unión, para conocer sus preferencias y el grado de afinidad.	Media
HU16	Como usuario ofertante, quiero poder aceptar o rechazar usuarios que solicitan unirse a mi comunidad.	Media

Código	Título	Prioridad
HU21	Como usuario buscador u ofertante, quiero poder consultar el calendario de tareas y actividades del piso.	Media
HU22	Como usuario buscador u ofertante, quiero visualizar en el calendario las tareas o actividades previstas en un día concreto, con detalles como hora, descripción y participantes.	Media
HU23	Como usuario buscador u ofertante, quiero que tanto las actividades como las tareas se añadan al calendario automáticamente para que mis compañeros las vean.	Media
HU24	Como usuario buscador u ofertante, quiero poder añadir eventos al calendario para que mis compañeros puedan consultarlos fácilmente.	Media
HU25	Como usuario buscador u ofertante, quiero poder registrar tareas a realizar mediante un formulario donde especifique dificultad, tiempo para realizarla y descripción.	Media
HU26	Como usuario buscador u ofertante, quiero poder registrar actividades a realizar mediante un formulario indicando lugar, fecha, hora y descripción.	Media
HU05	Como usuario buscador, quiero poder dar me gusta a los pisos que me interesan para guardarlos en una colección.	Media
HU06	Como usuario buscador, quiero poder consultar todas las comunidades a las que les he dado me gusta.	Baja
HU09	Como usuario buscador, quiero recibir notificaciones cuando acepten o rechacen mi solicitud de unión a una comunidad.	Baja
HU17	Como usuario ofertante, quiero recibir una notificación cuando un usuario solicite unirse a mi comunidad.	Baja
HU18	Como usuario ofertante, quiero poder modificar la información de mi comunidad para mantenerla actualizada.	Baja
HU20	Como usuario ofertante, quiero poder eliminar un usuario de mi comunidad en caso de incumplimiento de normas.	Baja

Código	Título	Prioridad
HU27	Como usuario buscador u ofertante, quiero recibir una notificación cuando se asignen las tareas semanales.	Baja
HU28	Como usuario buscador u ofertante, quiero distribuirme las tareas libremente a lo largo de la semana según mis necesidades.	Baja
HU29	Como usuario buscador u ofertante, quiero recibir una notificación con una hora de antelación antes de que termine el plazo de una tarea.	Baja
HU30	Como usuario buscador u ofertante, quiero que al final de la semana se genere un resumen con las tareas completadas por cada integrante.	Baja
HU31	Como usuario buscador u ofertante, quiero que si no distribuyo manualmente mis tareas en un tiempo límite, el sistema las asigne automáticamente.	Baja
HU32	Como usuario buscador u ofertante, quiero poder reorganizar las tareas asignadas para poder cumplir con contratiempos personales.	Baja
HU33	Como usuario buscador u ofertante, quiero que la asignación de tareas semanales sea cíclica, para que todos los integrantes hagan todas las tareas equitativamente.	Baja
HU34	Como usuario buscador u ofertante, quiero que los usuarios que no realizan sus tareas sean penalizados, reasignándoselas con un plazo de tiempo limitado.	Baja

5.2.1. Tarjetas de historia de usuario

A continuación, se presenta la lista de tarjetas de historias de usuario, las cuales serán fundamentales durante el desarrollo del sistema. Estas tarjetas permiten identificar las distintas tareas y criterios que deben implementarse para cada funcionalidad.

Historia de Usuario
Código: HU-01
Título: Como usuario buscador, quiero poder darme de alta en la aplicación para encontrar una comunidad a la que unirme.
Prioridad: Alta
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-01, RF-02
Descripción detallada: Como usuario nuevo del tipo buscador, quiero poder registrarme en la plataforma proporcionando un correo, un nombre de usuario, contraseña y rol válidos, para poder acceder a las funcionalidades del sistema.
Tareas: <ul style="list-style-type: none">■ Crear formulario de registro en frontend.■ Comprobar que el correo no está ya registrado.■ Implementación de la lógica del backend para registrar un nuevo usuario.■ Validación de longitud adecuada en el campo contraseña.
Pruebas de widget: <ul style="list-style-type: none">■ Renderizado del formulario con los campos username, email y contraseña.■ El formulario envía de forma correcta los datos al backend
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El método guarda correctamente el usuario si es válido.■ Se lanzan excepciones controladas si el registro del usuario falla
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario rellena el formulario y se envía de forma correcta.■ Tras el registro se muestra al usuario la pantalla del Home.

Historia de Usuario
Código: HU-02
Título: Como usuario buscador, quiero poder darme de baja de la aplicación eliminando todos mis datos.
Prioridad: Alta
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-04
Descripción detallada: El sistema debe permitir a los usuarios eliminar toda la información correspondiente a su cuenta de la base de datos. Esto implica que el usuario no podrá volver a acceder a su cuenta en la aplicación.
Tareas: <ul style="list-style-type: none">■ Creación de un endpoint en el backend para la eliminación de cuentas.■ Desarrollo de un botón en el frontend que permita solicitar la baja al usuario.■ Implementación de la lógica del backend para eliminar a un nuevo usuario.■ Actualización de la base de datos tras la eliminación.
Pruebas de widget: <ul style="list-style-type: none">■ El botón 'eliminar cuenta' es visible y funcional.■ El botón envía correctamente la llamada al backend
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El método elimina correctamente al usuario de la base de datos.■ Se lanzan excepciones si el usuario a eliminar no existiese.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario pulsa el botón y se envía la petición de forma correcta.■ Tras eliminar la cuenta el usuario no puede volver a acceder con las mismas credenciales.

Historia de Usuario
Código: HU-03
Título: Como usuario buscador, quiero poder configurar mis gustos y preferencias para recibir recomendaciones de comunidades afines.
Prioridad: Alta
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-03
Descripción detallada: El sistema debe permitir a los usuarios introducir sus gustos e intereses a través de una serie de preguntas. Estas preferencias deben poder ser editadas en cualquier momento.
Tareas: <ul style="list-style-type: none">■ Diseño de una interfaz donde el usuario introduzca sus preferencias y estilo de vida.■ Desarrollo de un endpoint en el backend para registrar y actualizar las preferencias del usuario.
Pruebas de widget: <ul style="list-style-type: none">■ El formulario permite al usuario añadir sus preferencias y editarlas.■ Tras introducir las preferencias de forma correcta se da una confirmación visual al usuario.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El método enlaza correctamente las preferencias a la entidad usuario.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario rellena los campos del formulario y estos se asocian a su perfil.

Historia de Usuario
Código: HU-04
Título: Como usuario buscador, quiero que el sistema me recomiende comunidades en función del grado de afinidad que compartimos.
Prioridad: Alta
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-17
Descripción detallada: El sistema debe ser capaz de analizar los intereses de cada usuario y compararlos con los perfiles de cada comunidad. De esta forma se debe ofrecer una lista de recomendaciones, mostrando el grado de afinidad que comparten usuario y comunidad.
Tareas: <ul style="list-style-type: none">■ Implementación de un algoritmo para el cálculo de afinidad en base a las preferencias del usuario y las características de la comunidad.■ Desarrollo de la interfaz visual para mostrar al usuario las comunidades con el grado de afinidad resaltado.■ Creación del endpoint que devuelve la lista de comunidades en función del cálculo realizado.
Pruebas de widget: <ul style="list-style-type: none">■ Si las preferencias del usuario se actualizan, la interfaz se recarga mostrando las nuevas recomendaciones.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El algoritmo calcula de forma correcta las recomendaciones en función de los parámetros introducidos.■ El endpoint devuelve la lista calculada de comunidades.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario tras configurar sus preferencias puede observar la lista de comunidades recomendadas.■ Tras actualizar las preferencias el usuario observa la nueva lista de recomendaciones.

Historia de Usuario
Código: HU-05
Título: Como usuario buscador, quiero poder dar me gusta a las comunidades que me interesan para guardarlas en una colección.
Prioridad: Baja
Estimación (PHU): 1
Requisitos funcionales relacionados: RF-18
Descripción detallada: El sistema debe permitir al usuario marcar aquellas comunidades que más le interesen, de forma que puede acceder a ellas en una sección aparte.
Tareas: <ul style="list-style-type: none">■ Creación de un botón de "me gusta" para cada comunidad mostrada en la interfaz de búsqueda.■ Creación de un endpoint que permita añadir y eliminar la comunidad marcada a la colección personal del usuario.
Pruebas de widget: <ul style="list-style-type: none">■ El botón de me gusta responde a la interacción del usuario pudiéndose marcar y desmarcar.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Tras marcar la comunidad, esta se añade a la colección del usuario.■ Tras desmarcar la comunidad, esta se elimina de la colección del usuario.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario puede marcar y desmarcar una comunidad desde la interfaz de búsqueda de comunidades.■ Tras pulsar el botón la comunidad se añade o se elimina de la colección del usuario.

Historia de Usuario
Código: HU-06
Título: Como usuario buscador, quiero poder consultar todas las comunidades a las que les he dado "me gusta".
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-18
Descripción detallada: El usuario tiene acceso a una sección dentro de la aplicación donde puede consultar todas las comunidades a las que le ha dado "me gusta".
Tareas: <ul style="list-style-type: none">■ Creación de la interfaz que muestra la colección de comunidades guardadas al usuario■ Desarrollo del endpoint en el backend para recuperar las comunidades marcadas por el usuario.
Pruebas de widget: <ul style="list-style-type: none">■ La sección muestra las comunidades marcadas al usuario.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Se recuperan desde la base de datos las comunidades correctas.■ Tras desmarcar la comunidad esta se elimina de la colección del usuario.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario accede a la sección y consulta todas las comunidades que ha marcado con 'me gusta'.

Historia de Usuario
Código: HU-07
Título: Como usuario buscador, quiero visualizar en detalle la información de una comunidad.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-05
Descripción detallada: Cuando el usuario seleccione una comunidad, el sistema le debe redirigir a una sección donde se muestre su perfil con la siguiente información: nombre, descripción, integrantes actuales, preferencias, galería de imágenes, localización y precio requerido.
Tareas: <ul style="list-style-type: none">■ Creación de la interfaz en el frontend que muestre la información de la comunidad■ Desarrollo del endpoint en el backend para obtener toda la información de una comunidad específica.
Pruebas de widget: <ul style="list-style-type: none">■ La sección muestra los datos de la comunidad consultada.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Se recuperan desde la base de datos la comunidad correcta.■ El endpoint devuelve la información correcta de la comunidad siguiendo siempre el mismo formato de respuesta.
Descripción detallada: El usuario accede a la sección y consulta la información de la comunidad seleccionada.

Historia de Usuario
Código: HU-08
Título: Como usuario buscador, quiero poder solicitar unirme a una comunidad que me interese, para pasar a formar parte de ella.
Prioridad: Alta
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-07
Descripción detallada: El sistema debe permitir al usuario enviar una solicitud para unirse a una comunidad concreta desde la interfaz de búsqueda de comunidades.
Tareas: <ul style="list-style-type: none">■ Creación del botón 'Solicitar Unirse' en la interfaz de búsqueda de comunidad.■ Desarrollo del endpoint en el backend para realizar el registro de la solicitud.■ Desarrollo de la lógica del backend para enviar la solicitud al usuario ofertante de dicha comunidad.
Pruebas de widget: <ul style="list-style-type: none">■ Se muestra el botón en la sección de búsqueda de comunidades.■ Al pulsar el botón de 'Solicitar Unirse' se envía la petición.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint envía la solicitud al ofertante de la comunidad seleccionada.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario pulsa el botón 'Solicitar Unirse' en la sección de búsqueda de comunidad, el usuario ofertante propietario de la comunidad recibe la petición.

Historia de Usuario
Código: HU-09
Título: Como usuario buscador, quiero recibir notificaciones cuando acepten o rechacen mi solicitud de unión a una comunidad.
Prioridad: Baja
Estimación (PHU): 1
Requisitos funcionales relacionados: RF-08
Descripción detallada: El sistema debe notificar al usuario buscador cuando el estado de una solicitud de unión a una comunidad cambie, esto debe generar una notificación que se puede consultar en una sección dentro de la aplicación.
Tareas: <ul style="list-style-type: none">■ Creación de una sección en el frontend donde se muestra una lista con todas las notificaciones recibidas por el usuario.■ Desarrollo de la lógica del backend para generar notificaciones cuando se detecte un cambio en el estado de una solicitud.■ Almacenar las notificaciones en la base de datos.
Pruebas de widget: <ul style="list-style-type: none">■ Las notificaciones enviadas al usuario aparecen en la interfaz.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Se genera la notificación al detectar un cambio en el estado de la solicitud.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario ofertante acepta/rechaza la petición del usuario el cuál puede entrar en la sección Notificaciones para ver la respuesta.

Historia de Usuario
Código: HU-10
Título: Como usuario buscador, quiero establecer filtros en la búsqueda de vivienda.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-17
Descripción detallada: El usuario debe poder filtrar los resultados de búsqueda de comunidades en función de los siguientes filtros: localización, número de integrantes, precio y espacio. Estos filtros se deben poder combinar permitiendo ofrecer al usuario una búsqueda más precisa.
Tareas: <ul style="list-style-type: none">■ Creación de los filtros en el frontend en la sección de búsqueda de comunidades.■ Desarrollo de la lógica del backend para filtrar los resultados obtenidos durante la búsqueda por afinidad.■ Creación de un endpoint en el backend que permita obtener las comunidades que cumplen con los filtros especificados.
Pruebas de widget: <ul style="list-style-type: none">■ Las recomendaciones se actualizan de forma automática al aplicar los filtros.■ Los filtros se pueden marcar y desmarcar pudiéndose combinar a gusto del usuario.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint devuelve solo las comunidades que cumplen los criterios.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario marca los filtros de búsqueda que le convienen y las recomendaciones de las comunidades se actualizan cumpliendo los criterios.

Historia de Usuario
Código: HU-11
Título: Como usuario buscador, quiero que el sistema me muestre las comunidades ordenadas en función del grado de afinidad.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-17,RF-19
Descripción detallada: El sistema debe ofrecer al usuario una lista de comunidades ordenadas en función de su grado de afinidad.
Tareas: <ul style="list-style-type: none">■ Creación de un endpoint en el backend que devuelva las comunidades ordenadas en función de su afinidad.
Pruebas de widget: <ul style="list-style-type: none">■ La vista de búsqueda de comunidades muestra las recomendaciones ordenadas en función de la afinidad.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint devuelve las comunidades ordenadas en función de grado de afinidad.

Historia de Usuario
<p>Código: HU-12</p> <p>Título: Como usuario ofertante, quiero poder darme de alta en el sistema para buscar integrantes con los que compartir mi comunidad.</p> <p>Prioridad: Alta</p> <p>Estimación (PHU): 3</p> <p>Requisitos funcionales relacionados: RF-01, RF-02</p> <p>Descripción detallada: El sistema debe permitir que nuevos usuarios se registren con el rol de ofertante mediante un formulario en el que se rellenen datos personales como : Nombre de usuario, correo electrónico, contraseña y rol.</p> <p>Tareas:</p> <ul style="list-style-type: none">■ Crear formulario de registro en el frontend.■ Validación de que el correo introducido no este ya registrado.■ Implementación de la lógica del backend para registrar un nuevo usuario de rol Ofertante.■ Validación de longitud adecuada en el campo contraseña. <p>Pruebas de widget:</p> <ul style="list-style-type: none">■ Renderizado del formulario con los campos username, email y contraseña.■ El formulario envía los datos de forma correcta al backend. <p>Pruebas unitarias del backend:</p> <ul style="list-style-type: none">■ El endpoint registra correctamente el usuario en la base de datos si este es válido.■ Se lanzan excepciones controladas si el registro del usuario falla. <p>Pruebas de sistema:</p> <ul style="list-style-type: none">■ El usuario rellena el formulario y los datos se envían de forma correcta.■ Tras el registro se muestra al usuario la pantalla del home. <p>Observaciones: Esta HU está estrechamente relacionada con la HU-01 (registro de usuarios buscadores) y puede compartir lógica y vistas comunes, la diferencia reside únicamente en la selección de rol.</p>

Historia de Usuario
<p>Código: HU-13</p> <p>Título: Como usuario ofertante, quiero poder darme de baja en el sistema, eliminando todos mis datos contenidos.</p> <p>Prioridad: Alta</p> <p>Estimación (PHU): 1</p> <p>Requisitos funcionales relacionados: RF-04</p> <p>Descripción detallada: El sistema debe permitir a los usuarios ofertantes eliminar toda su información contenida en la base de datos de forma sencilla. Esto implica que el usuario no podrá volver a acceder a su cuenta en la aplicación.</p> <p>Tareas:</p> <ul style="list-style-type: none">■ Creación de un endpoint en el backend para eliminación de cuentas.■ Creación de un botón en el frontend que permita solicitar la baja al usuario.■ Implementación de la lógica del backend para eliminar a un nuevo usuario.■ Actualización de la base de datos tras la eliminación.■ La eliminación tiene que ser en cascada, por lo que si el usuario ofertante tiene asociada una comunidad, esta también se eliminará por completo. <p>Pruebas de widget:</p> <ul style="list-style-type: none">■ El botón de 'eliminar cuenta' es visible y funcional■ El botón realiza correctamente la llamada al backend. <p>Pruebas unitarias del backend:</p> <ul style="list-style-type: none">■ El método elimina correctamente al usuario de la base de datos.■ Se elimina la comunidad asociada al usuario ofertante.■ Se lanzan excepciones si el usuario no existiese. <p>Pruebas de sistema:</p> <ul style="list-style-type: none">■ El usuario pulsa el botón y se envía la petición de forma correcta.■ Tras eliminar la cuenta el usuario no puede acceder con las mismas credenciales <p>Observaciones (optativas): La HU13 guarda similitudes con la HU-02 (baja de usuario buscador), por lo que ambos procesos pueden compartir lógica común, la diferencia reside en la eliminación en cascada que se produce con la entidad Comunidad asociada.</p>

Historia de Usuario
Código: HU-14
Título: Como usuario ofertante, quiero poder dar de alta una comunidad, introduciendo nombre, integrantes actuales, localización, precio, criterios de afinidad y fotos.
Prioridad: Alta
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-05
Descripción detallada: El usuario ofertante debe poder registrar una comunidad que pasará a estar disponible a los usuarios buscadores. Para el registro se ofrecerá un formulario en el que introducir la siguiente información: nombre, perfil de los integrantes actuales, localización, precio, criterios de afinidad (valores, intereses, estilo de vida) e imágenes de la comunidad.
Tareas: <ul style="list-style-type: none">■ Creación de un formulario de registro en el frontend.■ Creación de un endpoint en el backend que permita crear las nuevas comunidades a los usuarios Ofertantes.
Pruebas de widget: <ul style="list-style-type: none">■ El formulario se muestra al usuario, permitiéndole llenar los campos correspondientes.■ Tras finalizar el formulario, la información se envía correctamente al backend.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint en el backend registra de forma correcta las comunidades.■ Comprobación de que las comunidades se almacenan en la base de datos.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario ofertante accede a la sección registrar comunidad, rellena los datos del formulario y la comunidad pasa a estar visible para usuarios buscadores y vinculada con su perfil.

Historia de Usuario
Código: HU-15
Título: Como usuario ofertante, quiero consultar el perfil de cada usuario que envía una solicitud de unión, para conocer sus preferencias y el grado de afinidad.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-06
Descripción detallada: El usuario ofertante podrá consultar el perfil de los usuarios buscadores que le envían solicitud de unión, de esta forma podrá ver el grado de afinidad y más información para decidir si aceptar la solicitud o rechazarla.
Tareas: <ul style="list-style-type: none">■ Implementación en el frontend de una vista que permita acceder al perfil de un usuario buscador mediante su identificador.
Pruebas de widget: <ul style="list-style-type: none">■ Probar el acceso a los perfiles desde la lista de solicitudes.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El backend redirige al perfil correspondiente del usuario buscador.
Observaciones: Esta HU15 está muy ligada a la HU16 debido a que se va a construir sobre la lista de solicitudes de unión a una comunidad.

Historia de Usuario
Código: HU-16
Título: Como usuario ofertante, quiero poder aceptar o rechazar usuarios que soliciten unirse a mi comunidad.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-06
Descripción detallada: El sistema debe permitir al usuario ofertante aceptar o rechazar las solicitudes que recibe para participar en la comunidad. Tras realizar la decisión se modifica el estado de la solicitud enviando la respuesta al usuario buscador.
Tareas: <ul style="list-style-type: none">■ Creación de una vista en el frontend que muestra la lista de solicitudes que se han realizado al usuario.■ Creación de dos botones para cada solicitud que permitan aceptar o rechazar la solicitud■ Creación de un endpoint en el backend que permita enviar la respuesta al usuario buscador.
Pruebas de widget: <ul style="list-style-type: none">■ Se muestra la lista de peticiones enviadas al usuario.■ Los botones de aceptar y rechazar envían correctamente la información al backend.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El backend envía una notificación al usuario buscador con la respuesta del ofertante■ La solicitud se actualiza con un nuevo estado en función de la respuesta del usuario.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario decide si aceptar o rechazar una petición de unión pulsando un botón u otro.■ El usuario buscador recibe una notificación con la respuesta.

Historia de Usuario
Código: HU-17
Título: Como usuario ofertante, quiero recibir una notificación cuando un usuario solicite unirse a una comunidad.
Prioridad: Baja
Estimación (PHU): 1
Requisitos funcionales relacionados: RF-06
Descripción detallada: El usuario ofertante debe recibir una notificación en tiempo real cuando un usuario buscador solicite unirse a su comunidad, estas notificaciones serán visibles desde un centro de notificaciones.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la lógica del backend para generar una notificación al usuario Ofertante cuando un usuario Buscador solicite unirse.
Pruebas de widget: <ul style="list-style-type: none">■ Se muestra al usuario una sección con todas las notificaciones recibidas por el usuario.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Verificar que se crea una notificación en tiempo real cuando el usuario recibe la solicitud

Historia de Usuario
Código: HU-18
Título: Como usuario ofertante, quiero poder modificar la información de mi comunidad para mantenerla actualizada.
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-10
Descripción detallada: El sistema debe permitir al usuario ofertante modificar la información de su comunidad registrada previamente, permitiéndole cambiar los diferentes campos del perfil.
Tareas: <ul style="list-style-type: none">■ Creación de un endpoint en el backend que permita modificar la información de una comunidad.■ Creación de un formulario en el frontend prerellenado con la información actual para poder modificarla.
Pruebas de widget: <ul style="list-style-type: none">■ El formulario se muestra con los datos anteriores prerellenados.■ El formulario envía la información actualizada al backend.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint actualiza la información contenida en la base de datos.
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario ofertante accede a la sección de la comunidad, cambia los datos y estos quedan registrados en el sistema.

Historia de Usuario
Código: HU-19
Título: Como usuario ofertante, quiero poder eliminar una comunidad, borrando toda la información contenida en el sistema.
Prioridad: Alta
Estimación (PHU): 1
Requisitos funcionales relacionados: RF-09
Descripción detallada: El usuario ofertante debe poder eliminar una comunidad previamente registrada, eliminando de forma completa toda la información asociada en el sistema. En caso de que hubiera usuarios pertenecientes a dicha comunidad, se desvincularán automáticamente.
Tareas: <ul style="list-style-type: none">■ Creación de un endpoint en el backend para eliminar una comunidad.■ Desarrollo en el frontend de un botón para eliminar una comunidad.■ Eliminación en cadena de datos, eliminado la comunidad del perfil de los miembros participantes.
Pruebas de widget: <ul style="list-style-type: none">■ El botón de "eliminar" desvincula al usuario de la comunidad.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint elimina la comunidad de la base de datos.■ La comunidad se elimina del perfil de los usuarios
Pruebas de sistema: <ul style="list-style-type: none">■ El usuario ofertante accede a la sección comunidad y pulsa el botón de 'eliminar', desde ese momento los usuarios buscadores unidos a esa comunidad paran de formar parte de la misma.

Historia de Usuario
Código: HU-20
Título: Como usuario ofertante, quiero poder eliminar un usuario de mi comunidad en caso de incumplimiento de normas.
Prioridad: Baja
Estimación (PHU): 1
Requisitos funcionales relacionados: RF-10
Descripción detallada: El usuario ofertante debe poder eliminar un usuario perteneciente a la comunidad en el caso de que se salte algún tipo de norma fijado por los integrantes.
Tareas: <ul style="list-style-type: none">■ Desarrollo de un botón en la interfaz de la comunidad para eliminar un usuario.■ Creación de un endpoint en el backend que permita eliminar a un usuario de una comunidad.
Pruebas de widget: <ul style="list-style-type: none">■ La llamada al backend se realiza al pulsar el botón.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint elimina al usuario marcado de la comunidad.

Historia de Usuario
Código: HU-21
Título: Como usuario buscador u ofertante, quiero poder consultar el calendario de tareas y actividades del piso.
Prioridad: Media
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-12
Descripción detallada: Los usuarios pertenecientes a una comunidad deben poder visualizar el calendario de la comunidad, donde se muestran los eventos globales y las tareas de cada usuario.
Tareas: <ul style="list-style-type: none">■ Creación de un endpoint en el backend que devuelve los eventos relacionados para una comunidad.■ Creación de un endpoint en el backend que devuelve las tareas a realizar de un usuario de la comunidad.■ Desarrollo de una vista en el frontend que muestre un componente de calendario con navegación entre fechas y tareas-eventos.
Pruebas de widget: <ul style="list-style-type: none">■ El calendario muestra solo las actividades(eventos y tareas) en las que está involucrado el usuario.■ Se muestran tanto tareas como eventos en su fecha específica.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint devuelve las actividades asignadas al usuario.■ El endpoint devuelve los eventos de la comunidad.■ El calendario se mantiene actualizado con las tareas y eventos del usuario.

Historia de Usuario
Código: HU-22
Título: Como usuario buscador u ofertante, quiero visualizar en el calendario las tareas y eventos previstos en un día concreto, con detalles como hora, descripción y participantes.
Prioridad: Media
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-12, RF-13
Descripción detallada: Los usuarios deben poder consultar las tareas y eventos asignados a un día concreto del calendario, al pulsar en dicho día, se debe mostrar una interfaz donde se visualice el nombre de la actividad, descripción y los participantes
Tareas: <ul style="list-style-type: none">■ Creación de una vista lateral en el frontend al seleccionar un día, mostrando los detalles de la tarea o evento.■ Creación de un endpoint en el backend que devuelve las actividades y tareas a realizar en un día concreto para un usuario específico.
Pruebas de widget: <ul style="list-style-type: none">■ Al pulsar en un día concreto, la vista lateral se despliega.■ Se muestra la información correspondiente a dicho día para el usuario específico.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint devuelve las tareas y eventos asignadas al usuario en un día concreto recuperándolas de la base de datos.

Historia de Usuario
Código: HU-23
Título: Como usuario buscador u ofertante, quiero que los nuevos eventos y tareas registrados se añadan al calendario automáticamente para que el resto de integrantes las vean.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-11, RF-12
Descripción detallada: Las tareas y eventos creados por los usuarios se deben de añadir al calendario automáticamente. De esta forma, todos los involucrados podrán consultarlos de forma sencilla y rápida.
Tareas: <ul style="list-style-type: none">■ Creación de efecto visual para refresco del calendario.■ Creación de la lógica en el backend para que, al registrar una tarea/evento, esta se incluya en los calendarios de los usuarios involucrados.
Pruebas de widget: <ul style="list-style-type: none">■ Al crear una nueva tarea/evento, esta se visualiza en el calendario del usuario.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Al registrar una nueva tarea o evento, esta se asocia a los usuarios involucrados.■ El endpoint que devuelve las tareas/eventos devuelve las nuevas registradas.

Historia de Usuario
Código: HU-24
Título: Como usuario buscador u ofertante, quiero poder añadir eventos al calendario para que mis compañeros puedan consultarlos fácilmente.
Prioridad: Media
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-11, RF-12
Descripción detallada: El sistema debe permitir a todos los integrantes de una comunidad añadir eventos al calendario de manera que los miembros puedan verlas de forma más visual.
Tareas: <ul style="list-style-type: none">■ Creación del botón en el frontend para añadir eventos.■ Creación del endpoint que devuelva todos los eventos creados.
Pruebas de widget: <ul style="list-style-type: none">■ Los usuarios pueden visualizar el botón para añadir evento.■ El botón despliega una lista con todos los eventos registrados
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El endpoint devuelve todos los eventos registrados en la comunidad.

Historia de Usuario
Código: HU-25
Título: Como usuario buscador u ofertante, quiero poder registrar tareas a realizar mediante un formulario donde especifique lugar, fecha, hora y descripción
Prioridad: Media
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-11
Descripción detallada: El sistema debe permitir a todos los integrantes de una comunidad crear una nueva tarea a realizar. Para ello, se llenará un formulario indicando lugar, fecha, hora y descripción.
Tareas: <ul style="list-style-type: none">■ Creación del formulario en el frontend para que el usuario pueda introducir los campos mencionados.■ Desarrollo del endpoint en el backend para recoger los datos del formulario y registrar la nueva tarea en la base de datos
Pruebas de widget: <ul style="list-style-type: none">■ El formulario realiza la llamada al endpoint correctamente enviando los datos correctos.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Al registrar una nueva tarea esta se asocia a la comunidad y se almacena en la base de datos.■ El endpoint recoge los datos correctamente y crea la entidad con la información recopilada.

Historia de Usuario
Código: HU-26
Título: Como usuario buscador u ofertante, quiero poder registrar eventos a realizar mediante un formulario indicando lugar, fecha, hora y descripción.
Prioridad: Media
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-11
Descripción detallada: El sistema debe permitir a todos los integrantes de una comunidad crear un nuevo evento a realizar, para ello se llenará un formulario indicando lugar, fecha, hora y descripción. Los eventos están pensados como actividades grupales voluntarias, por ejemplo ir al cine.
Tareas: <ul style="list-style-type: none">■ Creación del formulario en el frontend para que el usuario pueda introducir los campos mencionados.■ Desarrollo del endpoint en el backend para recoger los datos del formulario y registrar el nuevo evento en la base de datos
Pruebas de widget: <ul style="list-style-type: none">■ El formulario realiza la llamada al endpoint correctamente enviando los datos correctos.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Al registrar un nuevo evento, este se asocia a la comunidad y se almacena en la base de datos.■ El endpoint recoge los datos correctamente y crea la entidad con la información recopilada.
Notas adicionales: La HU26 conlleva las mismas tareas que la HU25 debido a que las tareas y eventos están pensadas de forma muy similar. Las dos son actividades a realizar por los usuarios de la comunidad, siendo la HU25 tareas obligatorias y la HU26 más general y voluntaria.

Historia de Usuario
Código: HU-27
Título: Como usuario buscador u ofertante, quiero recibir una notificación cuando se asignen las tareas semanales.
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-13
Descripción detallada: El sistema debe informar a los miembros de la comunidad cuando las tareas semanales se han asignado. La idea es realizar un reparto semanal de las mismas. El modo de reparto será cíclico.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la lógica en el backend para enviar las notificaciones a los usuarios.
Pruebas de widget: <ul style="list-style-type: none">■ Los usuarios pueden visualizar las notificaciones indicándoles las tareas asignadas.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ Las notificaciones con las tareas a realizar se envian a los usuarios.

Historia de Usuario
Código: HU-28
Título: Como usuario buscador u ofertante, quiero distribuirme las tareas libremente a lo largo de la semana según mis necesidades.
Prioridad: Baja
Estimación (PHU): 3
Requisitos funcionales relacionados: RF-16
Descripción detallada: Los usuarios deben poder distribuirse las tareas asignadas de forma libre a lo largo de la semana. Cuando se realiza el reparto de tareas el usuario elige que días va a realizar dichas tareas.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la vista para poder colocar las tareas en los días escogidos por el usuario.■ Desarrollo de la lógica en el backend para establecer como fecha tope de realización de la tarea la establecida por el usuario.
Pruebas de widget: <ul style="list-style-type: none">■ Los usuarios pueden escoger mediante un selector el día de realización de la tarea.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El sistema actualiza el día de finalización de la tarea al escogido por el usuario.

Historia de Usuario
Código: HU-29
Título: Como usuario buscador u ofertante, quiero recibir una notificación antes de que termine el plazo de una tarea.
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-13
Descripción detallada: El sistema, a modo de aviso, debe alertar a los usuarios de la fecha tope de realización de una tarea. Para ello se enviará una notificación al usuario.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la lógica en el backend para enviar una notificación si la tarea sigue en estado sin realizar y queda una hora para la finalización del tiempo.
Pruebas de widget: <ul style="list-style-type: none">■ Los usuarios reciben la nueva notificación en tiempo real una hora antes del fin de la tarea.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El sistema detecta que el tiempo de tarea llega a su fin y envía la notificación al respectivo usuario.

Historia de Usuario
Código: HU-30
Título: Como usuario buscador u ofertante, quiero que al final de la semana se genere un resumen con las tareas completadas por cada integrantes.
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-13, RF-14
Descripción detallada: El sistema, a final de cada semana, antes de realizar el siguiente reparto de tareas, debe de generar un resumen de las tareas completadas por cada usuario a modo de guía.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la vista para poder mostrar de forma visual las tareas realizadas por cada integrante.■ Desarrollo de la lógica en el backend para generar un informe con las tareas realizadas por cada usuario
Pruebas de widget: <ul style="list-style-type: none">■ Los usuarios pueden acceder a una sección de resumen donde pueden ver las tareas que han realizado durante la semana.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El sistema recopila todas las tareas asignadas a un usuario y las muestra según su estado de realización.

Historia de Usuario
Código: HU-31
Título: Como usuario buscador u ofertante, quiero que si no distribuyo manualmente mis tareas en un tiempo límite, el sistema las asigne automáticamente
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-16
Descripción detallada: En caso de que el usuario no haya realizado el reparto de tareas asignadas en un tiempo límite, el sistema le asignará automáticamente las tareas en días aleatorios de la semana.
Tareas: <ul style="list-style-type: none">■ Implementación de la lógica en el backend para distribuir automáticamente las tareas cuando se detecta que se ha superado el tiempo máximo asignado.
Pruebas de widget: <ul style="list-style-type: none">■ Las tareas se añaden al calendario de forma automática
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El sistema detecta automáticamente cuándo se ha superado el tiempo máximo establecido y procede a redistribuir las tareas de manera aleatoria a lo largo de los días de la semana.

Historia de Usuario
<p>Código: HU-32</p> <p>Título: Como usuario buscador u ofertante, quiero poder reorganizar las tareas asignadas para poder cumplir con contratiempos personales.</p> <p>Prioridad: Baja</p> <p>Estimación (PHU): 2</p> <p>Requisitos funcionales relacionados: RF-16</p> <p>Descripción detallada: El sistema debe permitir al usuario modificar la fecha de realización de las tareas de forma que se pueda adaptar a contratiempos personales.</p> <p>Tareas:</p> <ul style="list-style-type: none">■ Desarrollo de la opción en el frontend para poder modificar la fecha de realización de la tarea.■ Desarrollo de la lógica en el backend para modificar el campo fechaTope en la entidad Tarea. <p>Pruebas de widget:</p> <ul style="list-style-type: none">■ El usuario puede modificar la fecha a partir de una opción de alteración. <p>Pruebas unitarias del backend:</p> <ul style="list-style-type: none">■ La entidad de la tarea seleccionada se modifica correctamente.■ Se registra la entidad con la nueva fechaTope

Historia de Usuario
<p>Código: HU-33</p> <p>Título: Como usuario buscador u ofertante, quiero que la asignación de tareas semanales sea secuencial, para que todos los integrantes hagan todas las tareas equitativamente.</p> <p>Prioridad: Baja</p> <p>Estimación (PHU): 2</p> <p>Requisitos funcionales relacionados: RF-15</p> <p>Descripción detallada: El sistema realiza la asignación de tareas de forma secuencial, es decir, si en la tarea limpieza del salón, el orden de realización es: Ana, Adrián y Rocío; la siguiente semana le tocará a Adrián y la siguiente, a Rocío.</p> <p>Tareas:</p> <ul style="list-style-type: none">■ Desarrollo de la lógica en el backend para que al realizar el reparto de las tareas a los usuarios se tenga en cuenta el orden secuencial establecido. <p>Pruebas unitarias del backend:</p> <ul style="list-style-type: none">■ Al realizar la asignación de la tarea a un usuario se sigue el orden secuencial.

Historia de Usuario
Código: HU-34
Título: Como usuario buscador u ofertante, quiero que los usuarios que no realicen sus tareas sean penalizados, reasignándoselas con un plazo de tiempo limitado.
Prioridad: Baja
Estimación (PHU): 2
Requisitos funcionales relacionados: RF-15
Descripción detallada: El sistema debe penalizar a los usuarios que no realizan sus tareas. Para ello, en el caso de que no se realicen, estas volverán a ser reasignadas a los mismos usuarios con un plazo de tiempo de dos días.
Tareas: <ul style="list-style-type: none">■ Desarrollo de la lógica en el backend para volver a asignar una tarea que el usuario no ha marcado como completada en el tiempo límite.
Pruebas unitarias del backend: <ul style="list-style-type: none">■ El sistema vuelve a asignar al usuario las tareas que no ha realizado■ Las tareas no realizadas se marcan con fecha límite de dos días.

5.3. Diagrama de clases de análisis

Tras analizar las diferentes funcionalidades a implementar en el sistema y los criterios que se deben pasar para considerarse completadas, se va a proceder a realizar una descripción del sistema a partir de un diagrama de clases de análisis UML [19].

Este diagrama representará de forma visual las diferentes entidades implicadas en el funcionamiento del sistema. Para ello se identificarán sus atributos, las distintas estructuras de datos que se verán involucradas en el proceso y las relaciones que existirán entre entidades.

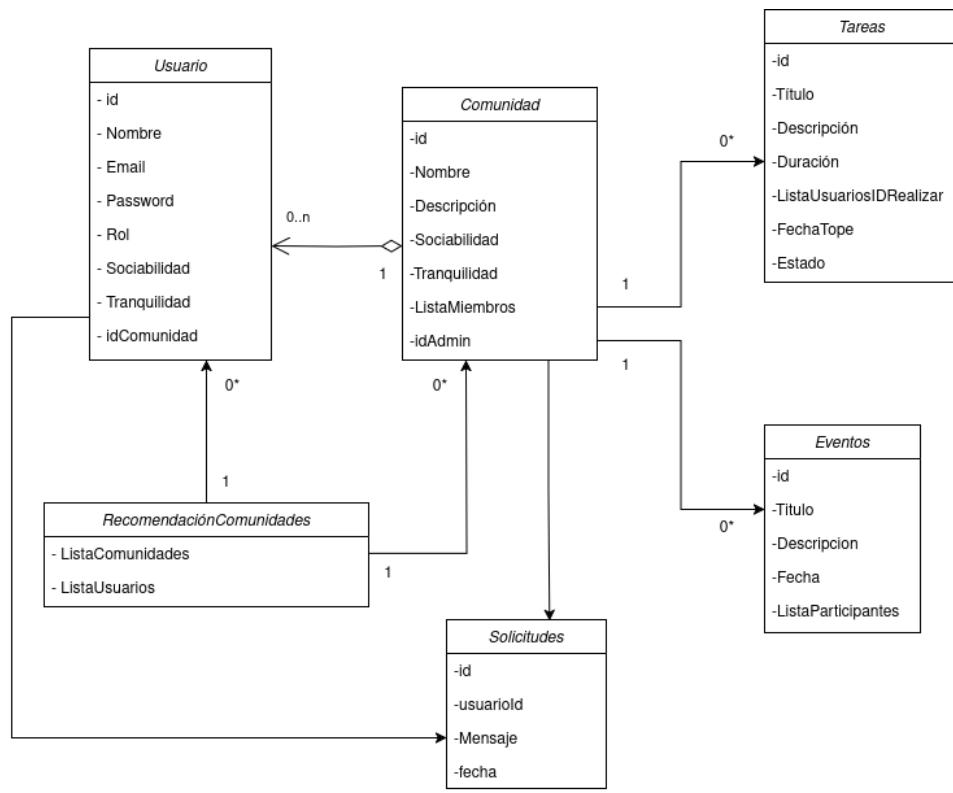


Figura 5.1: Diagrama UML de clases de análisis.

Capítulo **6**

Diseño

6.1. Arquitectura del sistema

Para abordar la estructuración y diseño del sistema, se ha optado por una arquitectura que asegure la escalabilidad, la resiliencia y la mantenibilidad. Con este objetivo, se ha decidido implementar una arquitectura basada en microservicios con el frontend desacoplado. Esta decisión permitirá dividir la lógica de negocio en servicios individuales e independientes, facilitando buenas prácticas de desarrollo y la evolución del sistema.

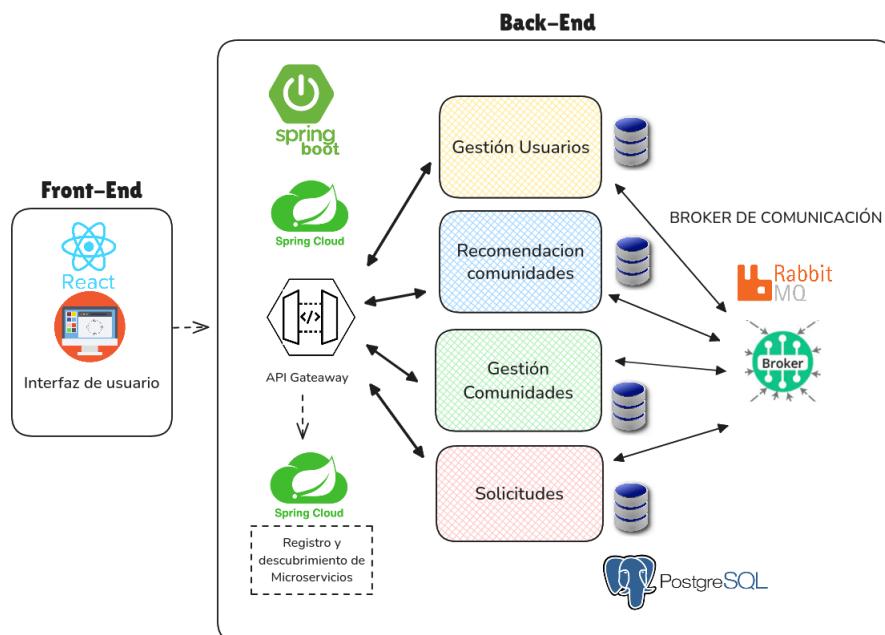


Figura 6.1: Diagrama de la arquitectura del sistema.

6.2. Arquitectura del Backend

El back-end del sistema estará basado en una **arquitectura de microservicios** [31]. Esta decisión responde a las ventajas que ofrece este estilo arquitectónico en términos de modularidad, escalabilidad y mantenimiento. Sin embargo, es importante tener en cuenta que este tipo de estilo arquitectónico, también implica una mayor complejidad en términos de creación y comunicación entre los servicios. A continuación, se expondrán las características fundamentales que seguirá nuestra estructura definida en la figura 6.1.

6.2.1. Estructura de los Microservicios

- **Microservicios Independientes:** Cada microservicio es responsable de un conjunto de funcionalidades relacionadas, que constituyen una parte lógica del sistema.
- **Base de Datos Independiente:** Cada microservicio tiene su propia base de datos, lo que permite un aislamiento completo de sus datos y mejora la resiliencia del sistema.
- **Comunicación Asíncrona:** La comunicación entre microservicios se realiza a través de un broker de mensajería **RabbitMQ** [27], basado en eventos. Esto permite mantener la asincronía en las operaciones, lo cual mejora la eficiencia y la escalabilidad.
- **API Gateway:** Para facilitar la comunicación entre el frontend y los microservicios, se implementará una **API Gateway**. Este componente será responsable de verificar los tokens de autenticación y enrutar las solicitudes al microservicio adecuado. Además, la API Gateway proporcionará un punto único de entrada para el frontend, simplificando el acceso a los servicios backend.

6.2.2. Estructura Interna de los Microservicios

Dentro de cada microservicio se ha seguido una arquitectura basada en capas, en general es muy cercana al Modelo-Vista-Controlador a excepción de que en este caso, la vista se estructura de forma completamente aislada. Como se observa en la imagen 6.2, en la estructura del sistema encontramos las siguientes capas:

- **Capa de Datos:** En ella se definen las entidades que serán mapeadas a la base de datos. Estas entidades representan los objetos de dominio del microservicio.
- **Capa de Presentación:** Se encarga de definir las operaciones de la API del microservicio. Define las rutas, valida las solicitudes y delega la lógica del sistema a la capa de negocio.
- **Capa de Negocio:** Contiene la lógica de negocio. Aquí se implementan las operaciones del sistema interactuando con la capa de datos para realizar

operaciones en la base de datos.

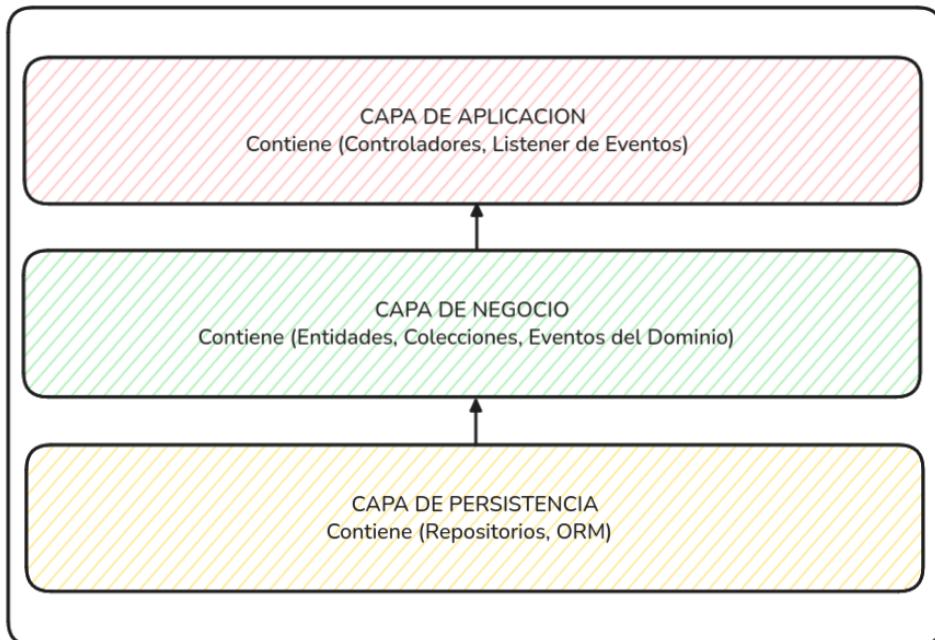


Figura 6.2: Diagrama de la arquitectura del backend

6.3. Arquitectura del Frontend

Para el frontend se ha seguido una arquitectura conocida como **Scream Architecture**, la cual nace del concepto de Clean Code [20] que busca mejorar la organización de los proyectos software asegurando que tanto el diseño como el código comuniquen claramente su intención.

6.3.1. Scream Architecture

La **Scream Architecture** se basa en organizar el código de manera que cada dominio o característica de la aplicación tenga su propia estructura de carpetas. Como vemos en la imagen 6.3, esto permite sustituir la clásica agrupación por componentes, por una arquitectura que facilita la mantenibilidad, escalabilidad del código y facilidad de entendimiento.

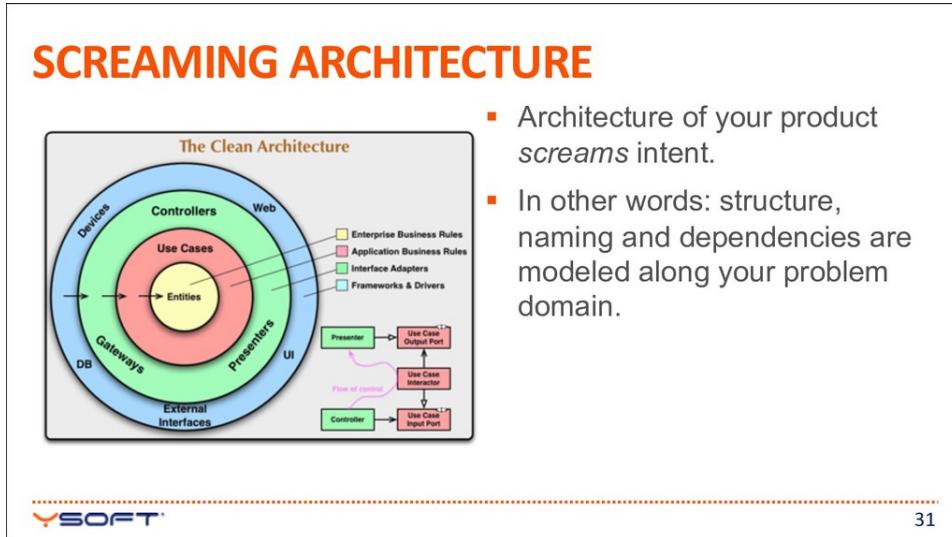


Figura 6.3: Explicación arquitectura screaming. Fuente: [25]

6.3.2. Estructura de la Carpeta Feature

Siguiendo la filosofía de esta arquitectura, nuestro frontend tiene un directorio feature que contiene diferentes subcarpetas para cada dominio del sistema. Por ejemplo, si se tiene un dominio *usuario*, dentro del directorio *usuario* se organizarán todos los archivos relacionados con ese dominio, tales como: Login del usuario, Registro del usuario y sus respectivos formularios. Para cada feature encontraremos las siguientes carpetas:

- **API Calls:** Se configuran las funciones que realizan las llamadas a la API del back-end relacionadas con la entidad *usuario*.
- **Types/Interfaces:** Se definen los tipos de datos utilizados en la entidad, como las interfaces de los objetos *usuario*.
- **Pages:** Se configuran las páginas o vistas relacionadas con esa entidad, organizadas dentro de la misma carpeta para mantener el código ordenado.
- **Components:** Si la entidad necesita componentes específicos, estos también se colocan dentro de su carpeta correspondiente.

Esta estructura permite que cada entidad se gestione de forma aislada y autónoma dentro del proyecto, lo que facilita la localización de funcionalidades y el mantenimiento de cada módulo.

6.4. Descripción de la API de cada microservicio

Como se mencionó anteriormente, el back-end estará compuesto por una serie de microservicios, cada uno encargado de encapsular una parte específica de la lógica de negocio del sistema. Cada microservicio expondrá una API que

ofrecerá las distintas operaciones del sistema. A continuación, se va a mostrar los distintos endpoints que compondrán la API de cada microservicio.

6.4.1. Microservicio Gestión de Usuarios

/usuarios/ – POST

Descripción: Registra un nuevo usuario en el sistema.

Body (DTO): UsuarioDTO : Nombre de usuario, contraseña, correo, LifestyleDTO

Salida: token JWT

Respuesta:

- 201 Created: Usuario creado exitosamente.
- 400 Bad Request: Datos inválidos o usuario ya existe.

/usuario/login – POST

Descripción: Autenticación de un usuario.

Body:

- Nombre de usuario.
- Contraseña.

Salida: token JWT

Respuesta:

- 200 OK: Autenticación correcta.
- 400 Bad Request: Credenciales inválidas.

/usuarios/ – GET

Descripción: Devuelve todos los Usuarios contenidos en el sistema.

Salida: Lista de usuarios

Respuesta:

- 200 OK: Usuarios encontrados.
- 404 Not Found: No hay usuarios.

/usuario/delete/{id-usuario} – POST

Descripción: Elimina un usuario específico.

Parámetro: id-usuario (URL path)

Respuesta:

- 204 No Content: Usuario eliminado.
- 500 Internal Server Error: Fallo al eliminar.

/usuario/{id-usuario} – PATCH

Descripción: Actualiza los datos de un usuario concreto.

Body (DTO): Campos a modificar del UsusarioDTO

Parámetro: id-usuario (URL path)

Respuesta:

- 200 OK: Usuario modificado.
- 404 Not Found / 500 Internal Server Error: Error.

/usuario/{id-usuario} – GET

Descripción: Obtiene los datos de un usuario específico.

Salida (DTO): UsuarioDTO

Parámetro: id-usuario (URL path)

Respuesta:

- 200 OK: Usuario encontrado.
- 404 Not Found: No existe el usuario.

6.4.2. Microservicio Gestión de Comunidades

GET /comunidades

Descripción: Devuelve todas las comunidades disponibles.

Salida: Lista de comunidades

Respuesta:

- 202 Accepted – Comunidades encontradas.
- 500 Internal Server Error – Error interno al obtener los datos.

GET /comunidades/{id-comunidad}

Descripción: Devuelve los datos de una comunidad específica.

Parámetro: id-comunidad (URL path)

Salida: ComunidadDTO

Respuesta:

- 202 Accepted – Comunidad encontrada.
- 404 Not Found – Comunidad no encontrada.

POST /comunidades/

Descripción: Registra una nueva comunidad.

Body – ComunidadDTO:

- nombre: String
- descripcion: String
- localizacion: String
- preferencias: String
- idAdmin: Long

Respuesta:

- 202 Accepted – Comunidad creada (devuelve ID).
- 500 Internal Server Error – Error al crear comunidad.

PATCH /comunidades/{id-comunidad}

Descripción: Modifica los datos de una comunidad existente.

Parámetro: id-comunidad (URL path)

Body – ComunidadDTO:

- nombre, descripción, localización, preferencias.

Respuesta:

- 202 Accepted – Comunidad modificada.
- 500 Internal Server Error – Error al modificar.

DELETE /comunidades/delete/{id-comunidad}

Descripción: Elimina una comunidad existente.

Parámetro: id-comunidad (URL path)

Respuesta:

- 202 Accepted – Comunidad eliminada correctamente.
- 400 Bad Request – Error al borrar la comunidad.

POST /comunidades/{id-comunidad}/usuarios/{id-usuario}/

Descripción: Añade un usuario a la comunidad.

Parámetros:

- Id de la comunidad: Id de la comunidad
- Id del usuario: Id del usuario a añadir.

Respuesta:

- 200 Accepted – Usuario añadido correctamente.
- 400 Bad Request – Error al añadir usuario a la comunidad.

DELETE /comunidades/{id-comunidad}/usuarios/{id-usuario}

Descripción: Elimina un usuario de la comunidad.

Parámetro:

- id-comunidad: Id de la comunidad a la que pertenece el usuario a eliminar
- id-usuario: Id del usuario a eliminar.

Respuesta:

- **200 Accepted** – Usuario eliminado de la comunidad correctamente.
- **400 Bad Request** – Error al eliminar la comunidad.

GET /comunidades/{id-comunidad}/tareas

Descripción: Devuelve las tareas asociadas a una comunidad específica.

Parámetro: Id de la comunidad (URL path)

Salida (Lista): Tareas

Respuesta:

- **202 Accepted** – Tareas encontradas.
- **404 Not Found** – Comunidad no encontrada.

GET /comunidades/{id-comunidad}/eventos/

Descripción: Devuelve los eventos asociados a una comunidad específica.

Parámetro: id-comunidad (URL path)

Salida (Lista): Eventos

Respuesta:

- **202 Accepted** – Eventos encontrados.
- **404 Not Found** – Comunidad no encontrada.

GET /comunidades/{id-comunidad}/tareas/{id-usuario}/

Descripción: Devuelve las tareas asociadas a una comunidad y usuario específico.

Parámetro:

- id-comunidad: Comunidad a obtener los datos (String)
- id-usuario: Usuario a obtener los datos (String)

Salida (Lista): Tareas

Respuesta:

- **202 Accepted** – Tareas encontradas.
- **404 Not Found** – Comunidad/Usuario no encontrada.

6.4.3. Recomendación de Comunidades

GET /usuarios/{id-usuario}/recomendaciones

Descripción: Busca las comunidades afines al usuario y las devuelve de forma ordenada.

Parámetro: id-usuario (URL path)

Lógica: Se calcula las comunidades afines con una función interna, calcularRecomendacion.

Salida (Lista): Comunidades

Respuesta:

- 202 Accepted – Devuelve una Lista de Comunidades afines
- 400 Bad Request – Error al calcular las comunidades.

GET /usuarios/{id-usuario}/recomendaciones/filtrado

Descripción: Busca las comunidades afines al usuario y les aplica el filtro seleccionado.

Parámetro: id-usuario (URL path)

Entrada:

- Filtros a aplicar (Distancia, precio, número de integrantes).

Lógica: Se calcula las comunidades afines con una función interna, calcularRecomendacion. Entre esas comunidades se descartan las que no pasan el filtro.

Salida (Lista): Comunidades

Respuesta:

- 202 Accepted – Devuelve una Lista de Comunidades afines filtradas.
- 400 Bad Request – Error al calcular las comunidades.

6.4.4. Solicitudes

GET /usuarios/{id-usuario}/solicitudes

Descripción: Obtiene las solicitudes de un usuario concreto.

Parámetro: id-usuario (URL path)

Salida (Lista): Solicitudes

Respuesta:

- 202 Accepted – Devuelve todas las solicitudes del usuario
- 400 Bad Request – Error al encontrar las solicitudes del usuario.

POST /usuarios/{id-usuario}/solicitudes

Descripción: Crea una nueva solicitud.

Body:

- Id del usuario destinatario
- Mensaje a enviar con la solicitud

Respuesta:

- 200 Accepted – Solicitud creada correctamente.
- 400 Bad Request – Error al crear la solicitud.

6.5. Diagrama Entidad-Relación

El diagrama de Entidad-Relación permitirá representar de forma visual la relación entre las entidades del sistema contenidas en la base de datos. Esto permitirá visualizar los atributos que componen cada entidad y las relaciones entre ellas.

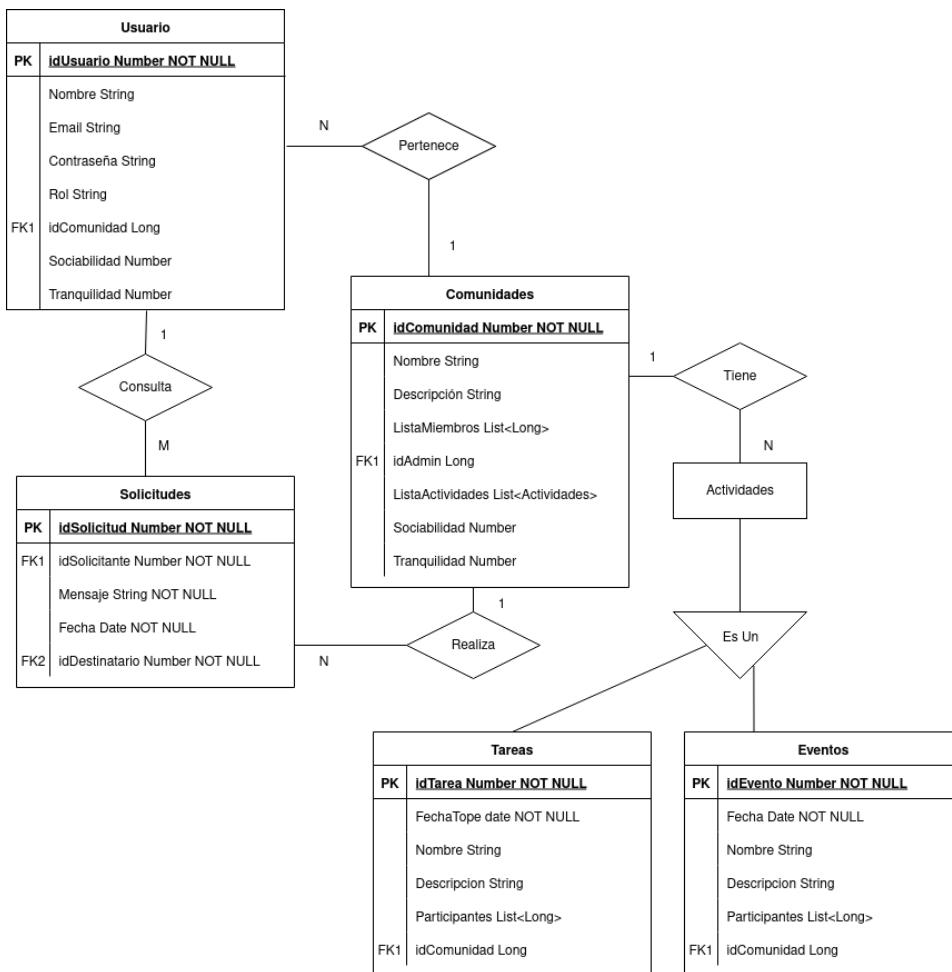


Figura 6.4: Diagrama de Entidad-Relación del sistema.

6.6. Diagrama de Clases

El diagrama de clases muestra los distintos atributos de las entidades del sistema junto a sus operaciones y relaciones. Esto permitirá crear un mapa conceptual del sistema entendiendo que debe realizar cada entidad y como se relacionan entre ellas.

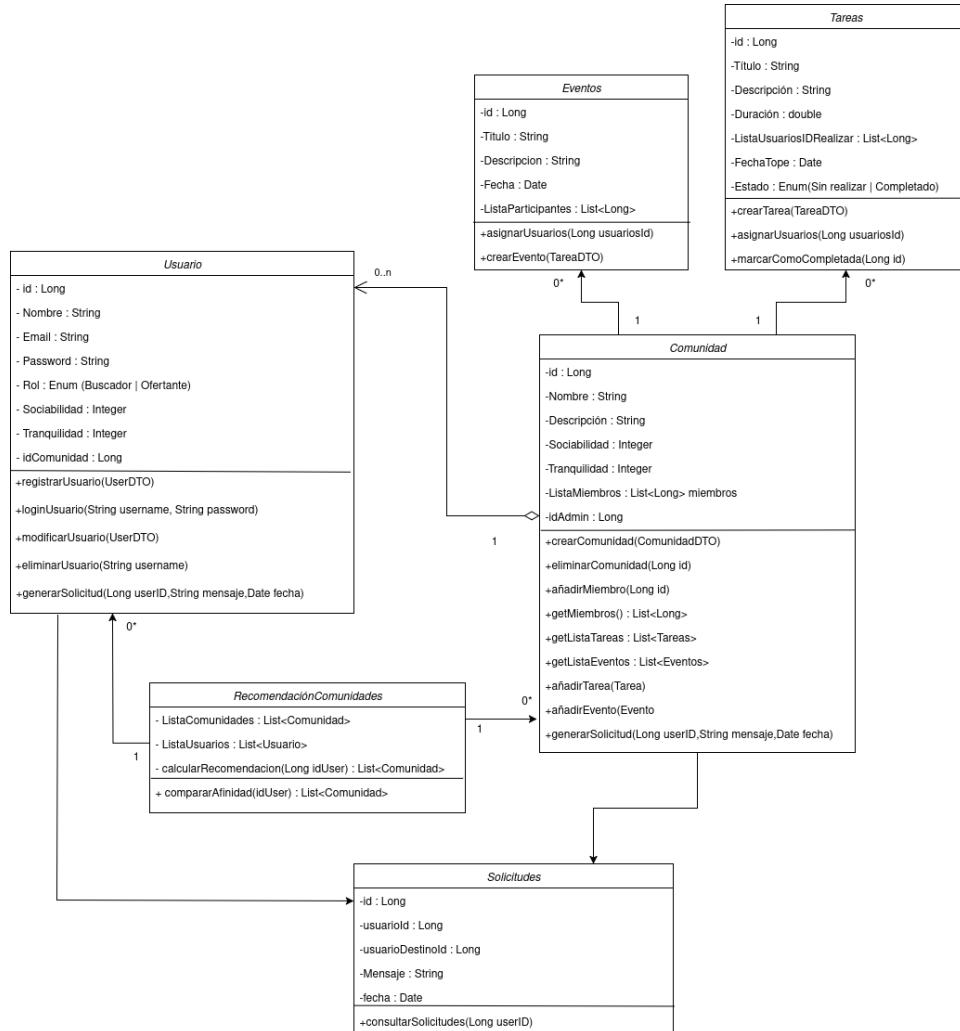


Figura 6.5: Diagrama UML de entidades del sistema.

6.7. Diagramas de Secuencia

Con el objetivo de ilustrar la implementación de los métodos más relevantes del sistema, se llevará a cabo los diagramas de secuencia de las siguientes operaciones: Registro de usuario, Inicio de sesión, Unión a comunidad, Búsqueda de

Comunidad, Consultar Tareas, Reparto de tareas y Crear Tarea.

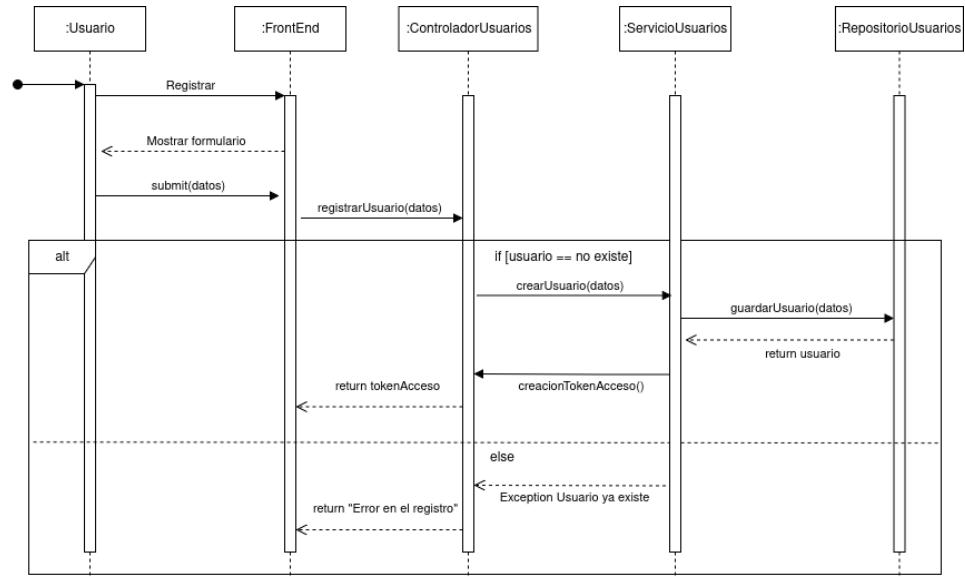


Figura 6.6: Diagrama de secuencia del registro.

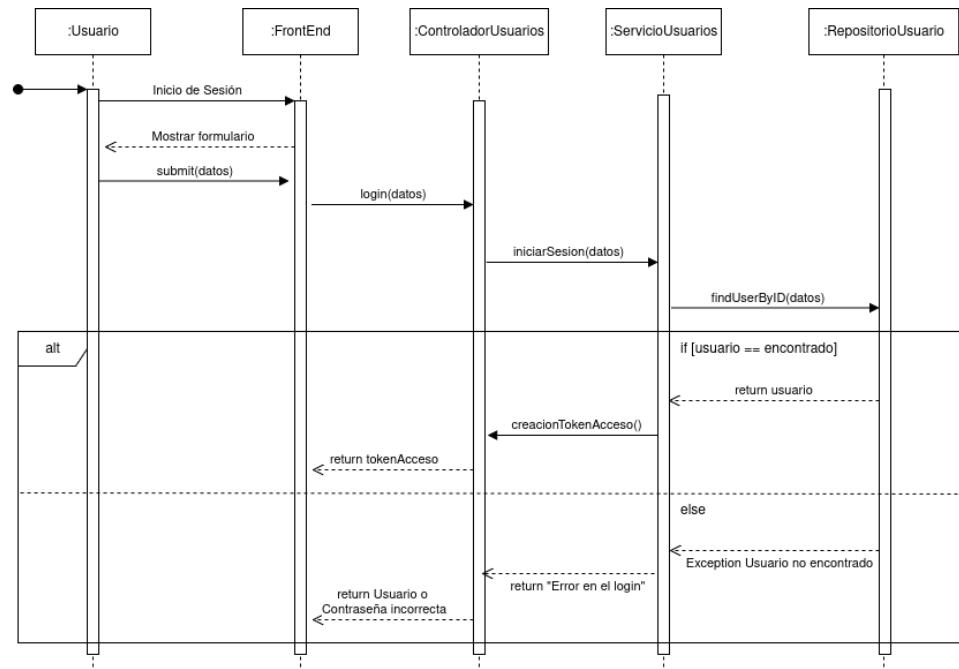


Figura 6.7: Diagrama de secuencia del login.

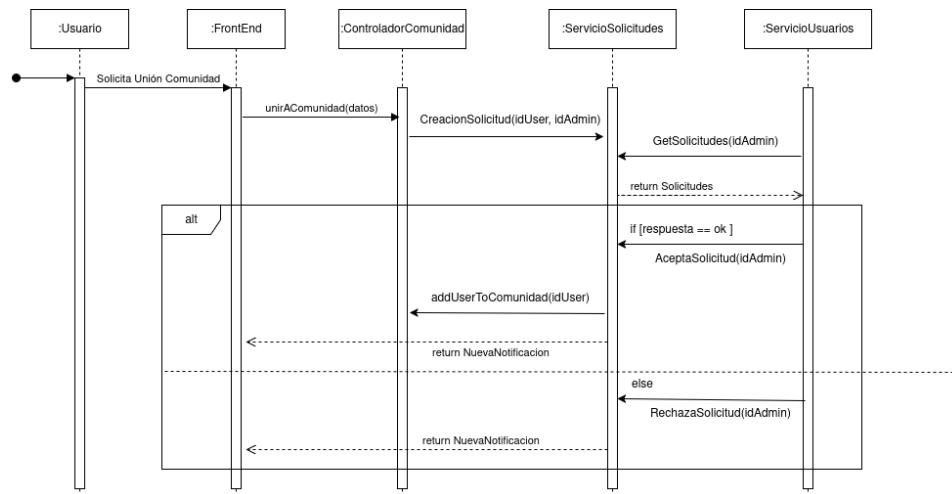


Figura 6.8: Diagrama de secuencia unión a una comunidad.

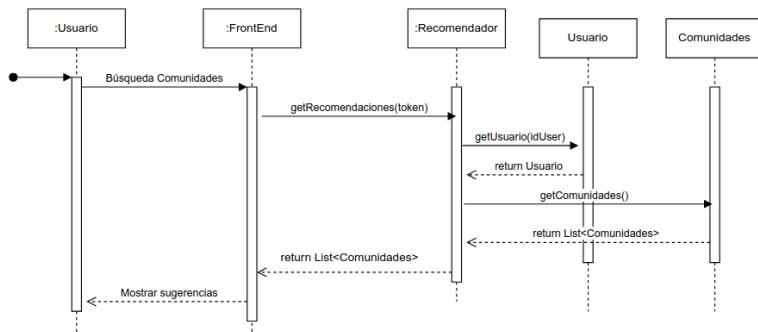


Figura 6.9: Diagrama de secuencia búsqueda de comunidades.

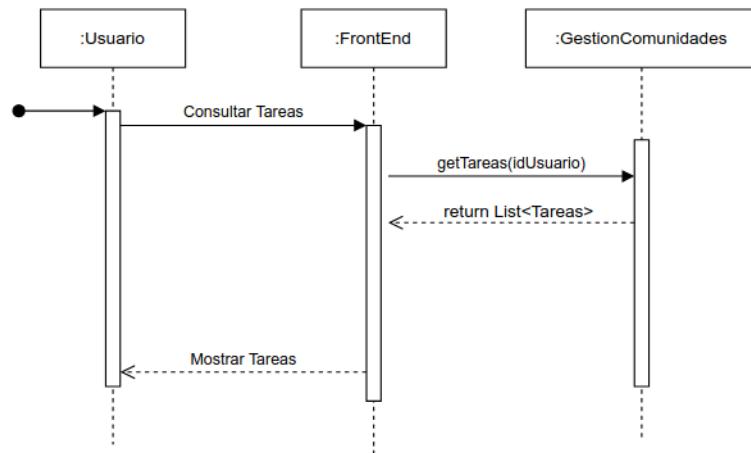


Figura 6.10: Diagrama de secuencia consultar tareas.

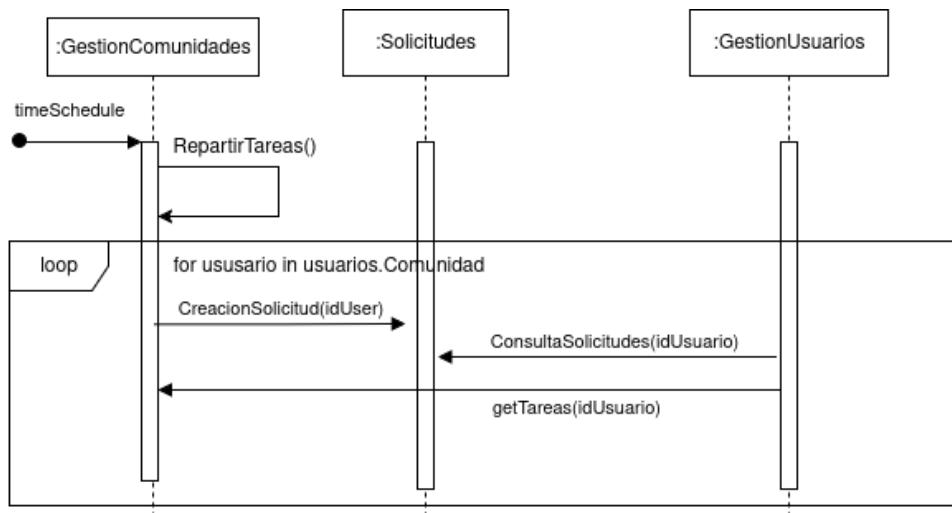


Figura 6.11: Diagrama de secuencia repartir tareas.

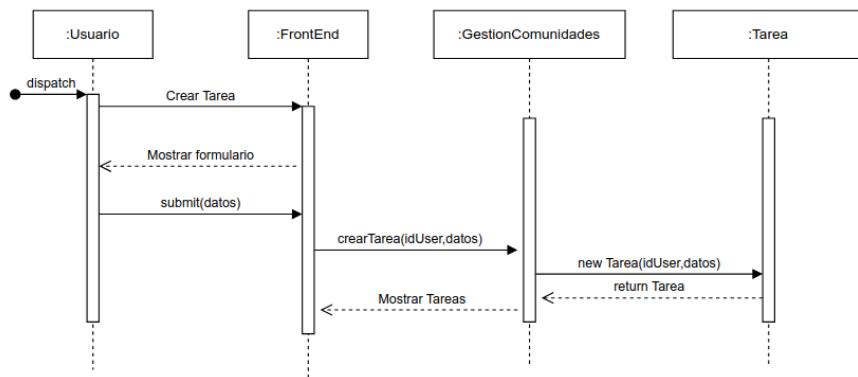


Figura 6.12: Diagrama de secuencia crear tarea.

6.8. Diseño de la interfaz de usuario

A continuación, se presentarán los distintos bocetos que compondrán la parte visual del sistema realizando una descripción y análisis de sus características.

6.8.1. Landing

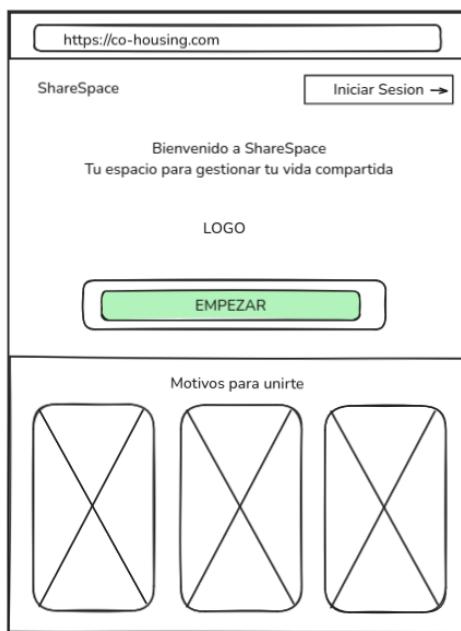


Figura 6.13: Boceto de la Landing Page.

6.8.1.1. Descripción

La interfaz del Landing muestra una pantalla de presentación al usuario de la aplicación, en ella de forma visual se permitirá iniciar sesión o registrarse mostrando además, distintos motivos por los que unirse a la aplicación. Los botones 'Empezar' e 'Iniciar Sesión' actúan como un acceso directo a la sección del Login y Registro.

6.8.1.2. Elementos de la interfaz

- Campo de texto con el nombre de la app y mensaje de bienvenida.
- Logo.
- Botón para Iniciar Sesión.
- Botón para Registrarse.
- Lista de tarjetas con motivos para unirse.

6.8.1.3. Funcionalidad

El usuario puede entender de forma sencilla las distintas características que ofrece la aplicación. Además, le permitirá acceder rápidamente tanto al inicio de sesión como al registro. En general, su funcionalidad es dar una imagen de presentación al usuario de la aplicación.

6.8.1.4. Criterios de usabilidad

- La información debe presentarse de forma clara, con una jerarquía visual intuitiva.
- Los botones deben ser accesibles, estar correctamente etiquetados y ser fácilmente accionables tanto con ratón como con teclado.
- La interfaz debe ser completamente responsive, adaptándose a distintos tamaños de pantalla.

6.8.2. Login

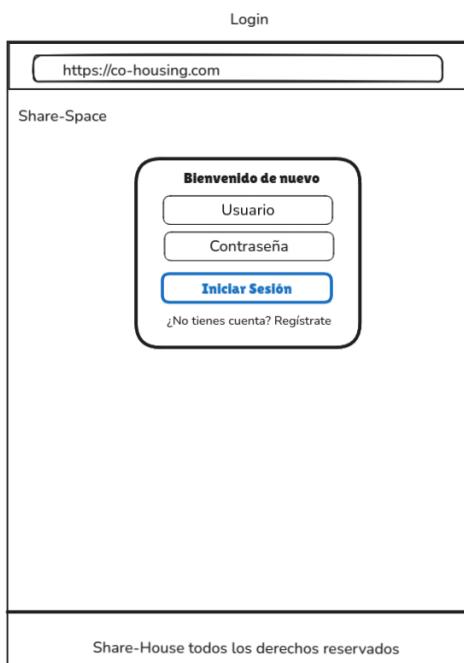


Figura 6.14: Boceto del login.

6.8.2.1. Descripción

Esta interfaz corresponde a la pantalla de inicio de sesión de la aplicación. Su propósito es permitir que el usuario se autentique.

Elementos de la interfaz

- Campo de texto para el nombre de usuario.
- Campo de texto para la contraseña (con ocultación de caracteres).
- Botón de acción para iniciar sesión.

6.8.2.2. Funcionalidad

La interfaz permite al usuario introducir su nombre de usuario y contraseña. Al pulsar el botón de inicio de sesión, el sistema valida las credenciales. Si la autenticación es exitosa, el usuario será redirigido a la pantalla principal (Home) de la aplicación. En caso contrario, se mostrará un mensaje de error informando de la autenticación fallida.

6.8.2.3. Criterios de usabilidad

- Los campos de entrada deben ser claramente visibles y estar correctamente etiquetados.
- El formulario debe proporcionar retroalimentación inmediata en caso de errores, como credenciales inválidas.
- El diseño debe ser accesible e intuitivo, permitiendo una navegación fluida desde el teclado y adaptándose a distintos tamaños de pantalla.

6.8.3. Registro

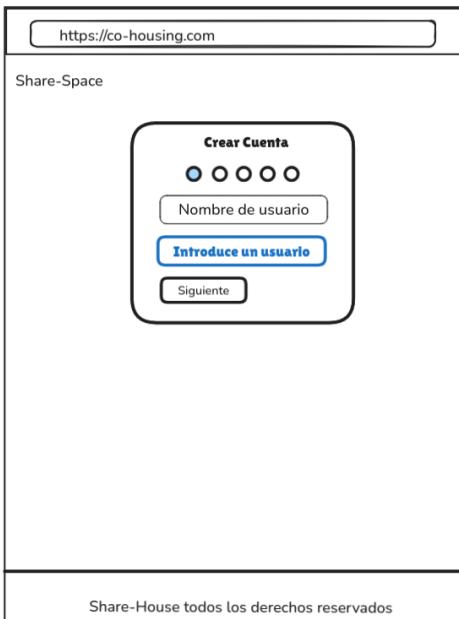


Figura 6.15: Boceto del registro.

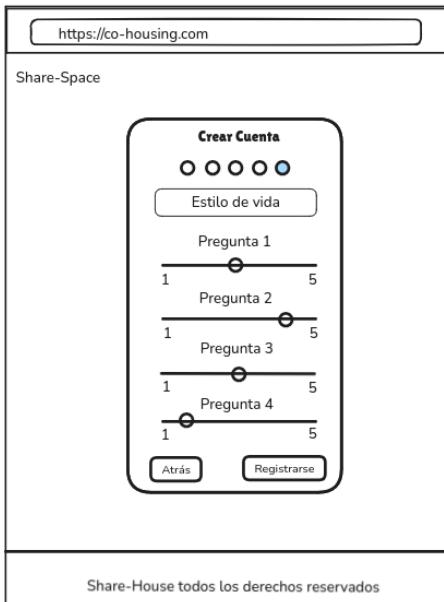


Figura 6.16: Boceto del registro selección de preferencias.

6.8.3.1. Descripción

Esta interfaz corresponde a la pantalla de registro de la aplicación. Su propósito es permitir que el usuario se registre introduciendo sus datos personales y preferencias.

6.8.3.2. Elementos de la interfaz

- Una tarjeta que va variando para cada paso del registro.
- Campo de texto para introducir credenciales.
- Campo seleccionable para introducir el valor de los gustos.
- Botón de acción para ir al siguiente paso.

6.8.3.3. Funcionalidad

La interfaz permite al usuario introducir sus datos personales y sus gustos. Al pulsar el botón de siguiente, el sistema muestra el siguiente campo, así hasta llegar al final. El último campo es el de los gustos personalizados y será establecido con un selector. Si el registro es exitoso, el usuario será redirigido a la pantalla principal (Home) de la aplicación. En caso contrario, se mostrará un mensaje de error informando del error del registro.

6.8.3.4. Criterios de usabilidad

- Los campos de entrada deben ser claramente visibles y estar correctamente etiquetados.
- El formulario debe proporcionar retroalimentación inmediata en caso de errores, como credenciales inválidas.
- El diseño debe ser accesible e intuitivo, permitiendo una navegación fluida desde el teclado y adaptándose a distintos tamaños de pantalla.

6.8.4. Home

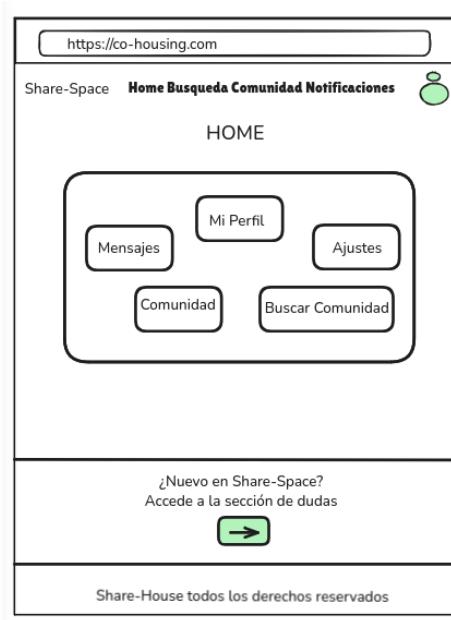


Figura 6.17: Boceto del home.

6.8.4.1. Descripción

La interfaz de inicio (pantalla Home) es la primera vista que el usuario ve tras autenticarse correctamente. Su objetivo es mostrar de forma visual y organizada las distintas funcionalidades que ofrece la aplicación, permitiendo un acceso rápido e intuitivo a cada una de ellas.

6.8.4.2. Elementos de la interfaz

- Una tarjeta contenedora que agrupa visualmente todas las funcionalidades disponibles.
- Una tarjeta individual por cada funcionalidad, que incluye su nombre y diseño distintivo.

- Un botón específico para acceder a la sección de dudas o ayuda.

6.8.4.3. Funcionalidad

El usuario puede interactuar con las distintas tarjetas funcionales. Al hacer clic sobre una tarjeta, se le redirige a la sección correspondiente de la aplicación. Asimismo, al pulsar el botón de dudas, el sistema lleva al usuario a la sección de ayuda, donde puede resolver preguntas frecuentes o contactar con soporte.

6.8.4.4. Criterios de usabilidad

- Las funcionalidades deben estar claramente identificadas mediante texto y/o iconografía.
- Las tarjetas deben ser fácilmente seleccionables, tanto por ratón como por teclado.
- El diseño debe ser responsive y adaptarse a diferentes dispositivos y tamaños de pantalla.
- La interfaz debe proporcionar una experiencia de navegación fluida y accesible para todo tipo de usuarios.

6.8.5. Home de la Comunidad

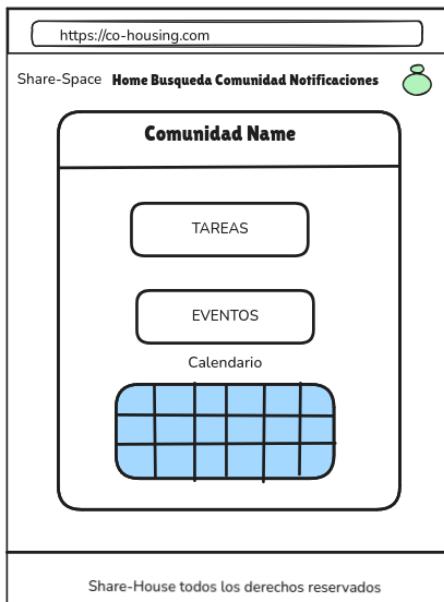


Figura 6.18: Boceto del home de la comunidad.

6.8.5.1. Descripción

La interfaz del Home de la Comunidad permite al usuario visualizar y acceder de forma organizada a las tareas y eventos asignados dentro de la comunidad. Asimismo, integra un calendario interactivo que muestra de manera clara y estructurada todas las actividades asociadas al usuario.

6.8.5.2. Elementos de la interfaz

- Tarjeta de acceso a las tareas asignadas al usuario dentro de la comunidad.
- Tarjeta de acceso a los eventos en los que el usuario participa.
- Calendario interactivo que muestra visualmente las tareas y eventos programados.

6.8.5.3. Funcionalidad

El usuario puede interactuar con las distintas tarjetas funcionales para acceder a secciones específicas:

- Al seleccionar la tarjeta de tareas, se accede al listado detallado de actividades que debe realizar.
- Al seleccionar la tarjeta de eventos, se muestran los eventos comunitarios en los que está inscrito.
- El calendario ofrece una vista global donde se reflejan las tareas y eventos del usuario, facilitando su planificación y seguimiento.

6.8.5.4. Criterios de usabilidad

- Cada elemento funcional debe estar claramente identificado mediante texto e iconografía representativa.
- La interfaz debe ser completamente navegable mediante teclado y compatible con tecnologías de asistencia.
- El diseño debe adaptarse correctamente a distintos dispositivos (responsive design).
- Las interacciones deben ser intuitivas, con retroalimentación visual ante acciones del usuario (hover, selección, etc.).

6.8.6. Tareas

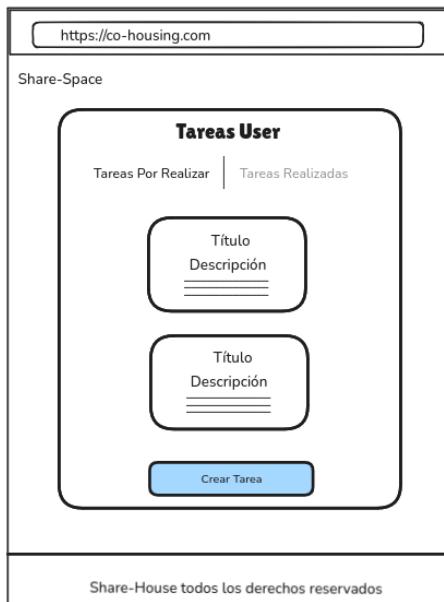


Figura 6.19: Boceto de la lista de tareas del usuario.

6.8.6.1. Descripción

La interfaz de la lista de tareas permite al usuario ver todas las tareas que tiene asignado en esta semana, aquí se mostrará una tarjeta para cada tarea mostrando el título y su descripción. Las tareas mostradas se dividirán en dos listas diferentes: las que están por realizar y las realizadas. Finalmente contiene un botón que redirecciona a un formulario para crear Tarea.

6.8.6.2. Elementos de la interfaz

- Tarjeta con la información principal de cada Tarea, si se pulsa la tarjeta redirecciona al perfil de la tarea.
- Botón para crear Tarea.
- Campo de texto con la descripción detallada de la tarea.
- Filtro de tareas realizadas y por realizar.

6.8.6.3. Funcionalidad

El usuario puede:

- Consultar todas las tareas asignadas en esta semana
- Crear una nueva tarea
- Pulsar la tarjeta para acceder al perfil de una tarea

6.8.6.4. Criterios de usabilidad

- Los botones y elementos interactivos deben ser fácilmente identificables y tener etiquetas comprensibles.
- El diseño debe ser adaptable a distintos tamaños de pantalla, priorizando la usabilidad en dispositivos móviles.

6.8.7. Tarea Específica

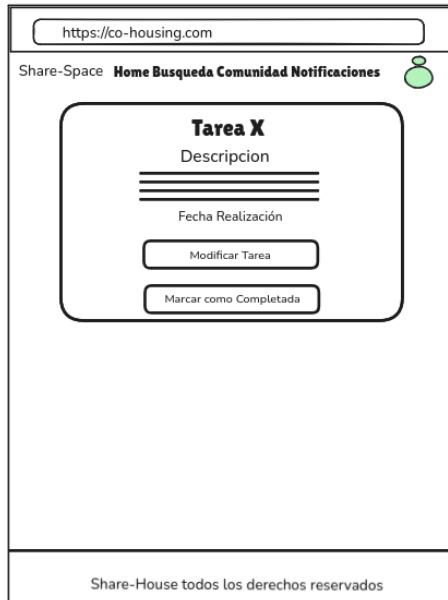


Figura 6.20: Boceto de la tarea.

6.8.7.1. Descripción

La interfaz de visualización de Tarea específica permite al usuario consultar la información detallada de una tarea concreta. Funciona como un perfil individual de la tarea, mostrando su título, descripción, fecha límite de realización y un botón para modificar la tarea, que redirige a un formulario de edición.

6.8.7.2. Elementos de la interfaz

- Tarjeta principal que agrupa toda la información de la tarea.
- Campo de texto con el título de la tarea.
- Campo de texto con la descripción detallada de la tarea.
- Fecha límite (fecha tope) de realización claramente visible.
- Botón de modificación que redirecciona al formulario de edición.
- Botón para marcar la tarea como completada.

6.8.7.3. Funcionalidad

El usuario puede:

- Consultar todos los detalles relevantes de una tarea específica.
- Ver la fecha límite para su realización.
- Pulsar el botón de edición para modificar la información de la tarea, siendo redirigido al formulario correspondiente.
- Pulsar el botón de completada, para marcar una tarea como finalizada.

6.8.7.4. Criterios de usabilidad

- Los botones deben ser visibles y accesibles.
- El diseño debe garantizar la legibilidad de los campos.

6.8.8. Buscador de comunidades

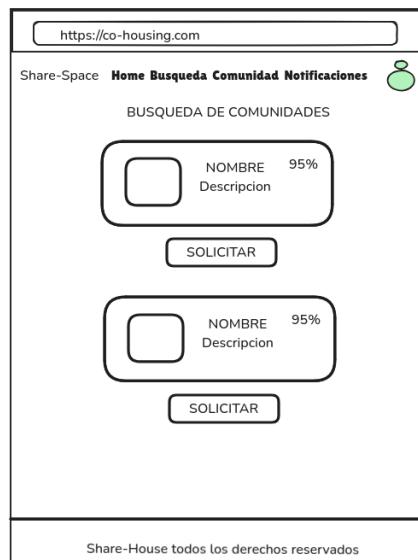


Figura 6.21: Boceto de la sección de búsqueda.

6.8.8.1. Descripción

La interfaz de búsqueda de comunidades afines muestra al usuario un listado de comunidades recomendadas en función de su afinidad. Cada comunidad se presenta mediante una tarjeta que incluye su imagen principal, nombre, descripción y el porcentaje de afinidad calculado. Además, se incluye un botón que permite al usuario solicitar unirse a dicha comunidad.

6.8.8.2. Elementos de la interfaz

- Tarjeta principal que agrupa la información visual de cada comunidad sugerida.
- Imagen representativa de la comunidad.
- Campo de texto con el nombre de la comunidad.
- Descripción resumida de la comunidad.
- Indicador del porcentaje de afinidad entre el usuario y la comunidad.
- Botón de acción para solicitar la unión a la comunidad.

6.8.8.3. Funcionalidad

El usuario puede explorar visualmente las distintas comunidades sugeridas, revisando su información básica y valorando su grado de afinidad. Si desea formar parte de una comunidad concreta, puede pulsar el botón de "Solicitar unión", lo que enviará una solicitud al sistema o a los administradores de dicha comunidad.

6.8.8.4. Criterios de usabilidad

- Los botones de acción deben ser fácilmente reconocibles y estar bien etiquetados.
- La interfaz debe ser adaptable a distintos dispositivos (responsive), asegurando la legibilidad y usabilidad.

6.8.9. Perfil del usuario

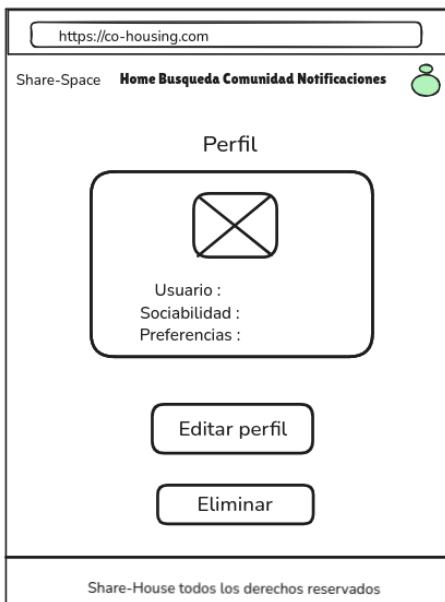


Figura 6.22: Boceto del perfil del usuario.

6.8.9.1. Descripción

La interfaz del perfil de usuario permite visualizar y gestionar los datos personales del usuario. En esta vista se muestran su avatar, nombre de usuario y los criterios de afinidad o gustos que ha definido. Además, se incluyen dos botones de acción: uno para acceder al formulario de edición del perfil y otro para eliminar la cuenta de usuario.

6.8.9.2. Elementos de la interfaz

- Tarjeta principal que agrupa la información visual del usuario.
- Imagen (avatar) del usuario.
- Campo de texto con el nombre de usuario.
- Visualización de los criterios de afinidad o gustos del usuario.
- Botón para acceder al formulario de edición del perfil.
- Botón para eliminar el perfil del usuario.

6.8.9.3. Funcionalidad

El usuario puede consultar de forma clara la información de su perfil, incluyendo su imagen, nombre y preferencias personales. Asimismo, se proporcionan

opciones para modificar estos datos o eliminar su cuenta mediante los botones de edición y eliminación, respectivamente.

6.8.9.4. Criterios de usabilidad

- Los botones deben ser accesibles, estar bien etiquetados y ser fácilmente accionables.
- La interfaz debe ser completamente responsive, adaptándose a distintos tamaños de pantalla.

6.8.10. Solicitudes

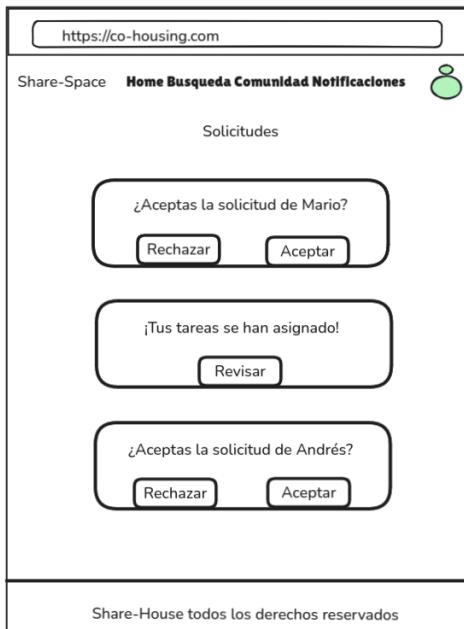


Figura 6.23: Boceto de la sección de Solicitudes.

6.8.10.1. Descripción

La interfaz de solicitudes muestra todas las solicitudes pendientes del usuario, incluyendo el mensaje de la solicitud y el nombre del remitente. Para cada solicitud, se presentan tres opciones mediante botones: aceptar, rechazar o revisar. El botón 'revisar' actúa como un acceso directo a secciones relacionadas, como tareas asignadas.

6.8.10.2. Elementos de la interfaz

- Tarjetas individuales para cada una de las solicitudes.

- Campo de texto con la descripción de la solicitud.
- Botón para aceptar la solicitud.
- Botón para rechazar la solicitud.
- Botón para revisar tareas o ver detalles de la comunidad, que redirecciona a otra sección.

6.8.10.3. Funcionalidad

El usuario puede visualizar todas las solicitudes pendientes y leer sus descripciones. Puede interactuar con ellas mediante los botones de aceptar, rechazar o revisar. Al realizar cualquiera de estas acciones, la solicitud correspondiente desaparece de la lista.

6.8.10.4. Criterios de usabilidad

- La información debe presentarse de forma clara, con una jerarquía visual intuitiva.
- Los botones deben ser accesibles, estar correctamente etiquetados y ser fácilmente accionables tanto con ratón como con teclado.
- La interfaz debe ser completamente responsive, adaptándose a distintos tamaños de pantalla.

6.9. Diagrama de navegabilidad

Para poder entender como va a ser el flujo de navegabilidad por las distintas pantallas de la aplicación se ha realizado el siguiente diagrama.

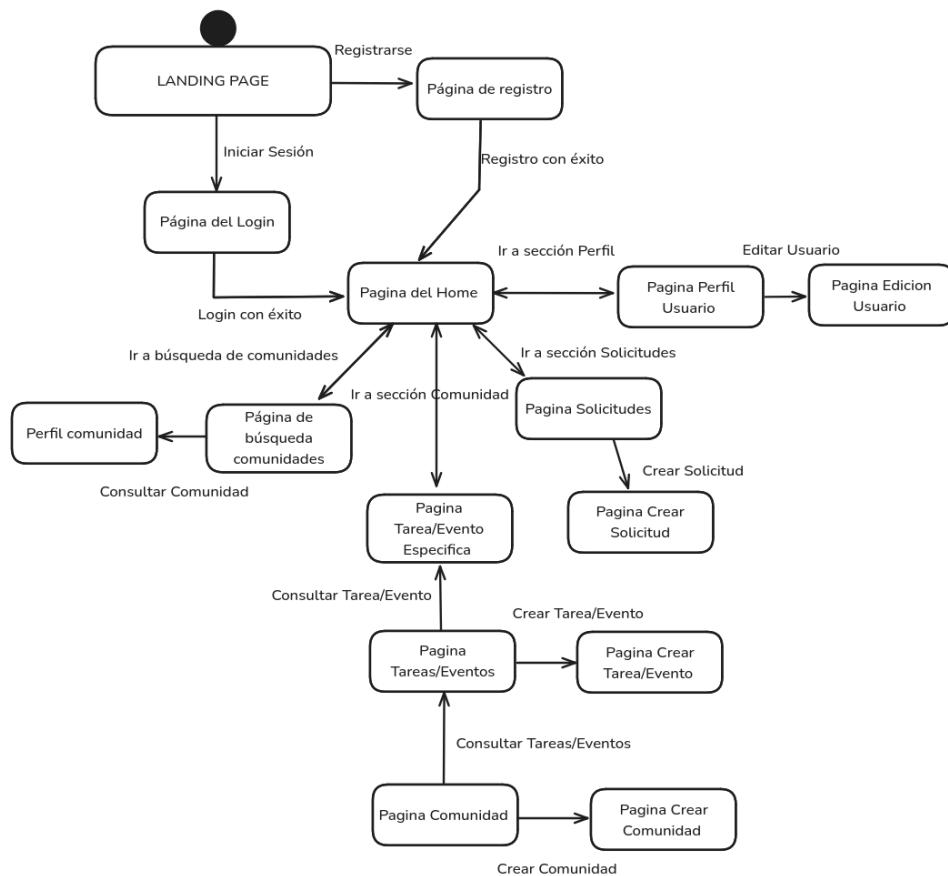


Figura 6.24: Diagrama de navegabilidad del sistema.

Capítulo **7**

Implementación

A continuación se procederá a explicar la implementación de las funcionalidades más representativas dentro del sistema, explicando su lógica, flujo y estructura. Además, se hará una presentación de las herramientas que han complementado el desarrollo de la aplicación de forma profesional y eficiente.

7.1. Herramientas

Durante todo el ciclo de vida del proyecto se han utilizado diversas herramientas que han complementado las distintas fases del desarrollo, desde la construcción de código hasta la integración continua (CI/CD), control de versiones y contenedorización.

7.1.1. VSCode IDE

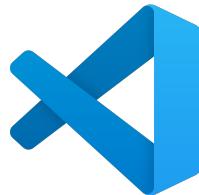


Figura 7.1: Logo IDE VSCode

Como entorno de desarrollo integrado (IDE) se ha utilizado Visual Studio Code (VSCode), un editor de código fuente desarrollado por Microsoft [23]. VSCode ha sido escogido por el gran número de extensiones gratuitas que ofrece, permitiendo trabajar con múltiples tecnologías y lenguajes de programación simultáneamente. Su integración con sistemas de control de versiones(GitHub)

y herramientas de desarrollo lo convierte en una gran opción para proyectos software profesionales.

7.1.2. GitHub



Figura 7.2: Logo GitHub

GitHub ha desempeñado un papel central durante el desarrollo del proyecto, no solo como plataforma de control de versiones, sino también como una herramienta clave para la automatización del desarrollo mediante GitHub Actions [12] y su sistema de integración y entrega continua (CI/CD). Gracias a estas funcionalidades, ha sido posible estructurar y automatizar el ciclo de vida del desarrollo de forma organizada y eficiente.

El repositorio del proyecto está disponible en: [Repositorio en GitHub](#)

Flujo de desarrollo

El repositorio asociado al proyecto cuenta con tres ramas principales permitiendo una gestión organizada del desarrollo:

- **main**: Rama asociada al entorno de producción, en ella se integran las iteraciones finalizadas que representan versiones estables y listas para ser desplegadas.
- **develop**: Rama de desarrollo, en ella se integran las tareas y las historias de usuario completadas, sirviendo como base para futuras versiones.
- **feature/<nombre-funcionalidad>**: Ramas individuales para el desarrollo de cada historia de usuario o funcionalidad específica.

Cada rama del proyecto cuenta con su propio flujo de integración y entrega continuas (CI/CD) [30], implementado mediante GitHub Actions. Ante cualquier operación de push, se desencadenan automáticamente distintas tareas y validaciones, como se ilustra en la figura 7.3, que incluyen la compilación del código, la ejecución de pruebas y el despliegue. Este proceso automatizado asegura la calidad y la estabilidad del software a lo largo del desarrollo.

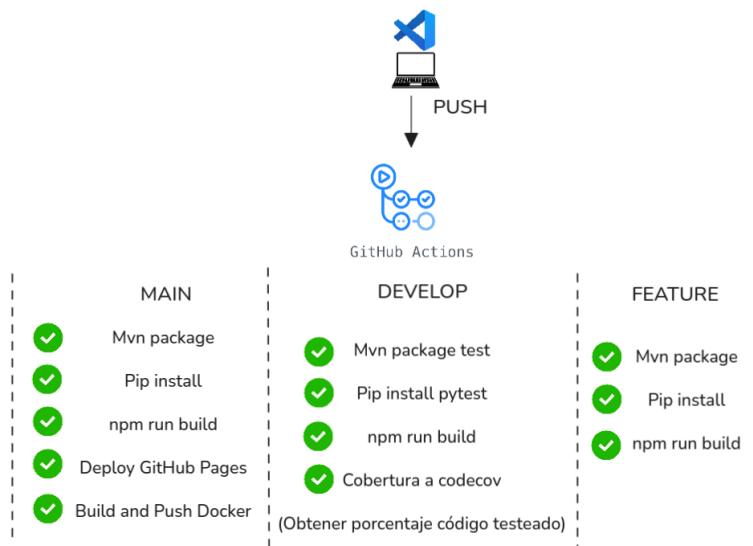


Figura 7.3: Flujo de desarrollo github actions

Workflow para ramas feature/*

Este workflow se activa con un push a cualquier rama que empiece por `feature/` y con Pull Requests dirigidos a `develop`. Las acciones que realizan son:

- Compilación de microservicios en Spring Boot
- Instalación de dependencias Python.
- Instalación de dependencias Node.js.
- Construcción del frontend para validar que no haya errores.

Workflow para la rama develop

Este workflow se activa con cada push a la rama `develop`. Su objetivo principal es ejecutar pruebas automatizadas y medir la cobertura del código. Las acciones que realiza son:

- Configuración del entorno con Ubuntu y Java 17.
- Restauración de la caché de dependencias Maven.
- Construcción del frontend para validar que no haya errores.
- Ejecución de tests en todos los microservicios Java con generación de reportes Jacoco.
- Ejecución de tests en Python para microservicios específicos.
- Envío de los reportes de cobertura a Codecov [2].

Workflow para la rama main

Este workflow se activa al hacer push a la rama `main` y se centra en realizar el despliegue de la aplicación. Las acciones que realiza son:

- Compilación completa de microservicios Java y construcción de imágenes Docker.
- Construcción y despliegue del frontend en GitHub Pages.
- Construcción del microservicio en Python.
- Login en Docker Hub para subir imágenes Docker de los microservicios Java y Python.

Además, se ha creado un [proyecto en Github](#), donde se han definido las historias de usuario y se han organizado en las distintas iteraciones del trabajo. De esta forma, GitHub se ha convertido en la herramienta central para la planificación y el desarrollo.

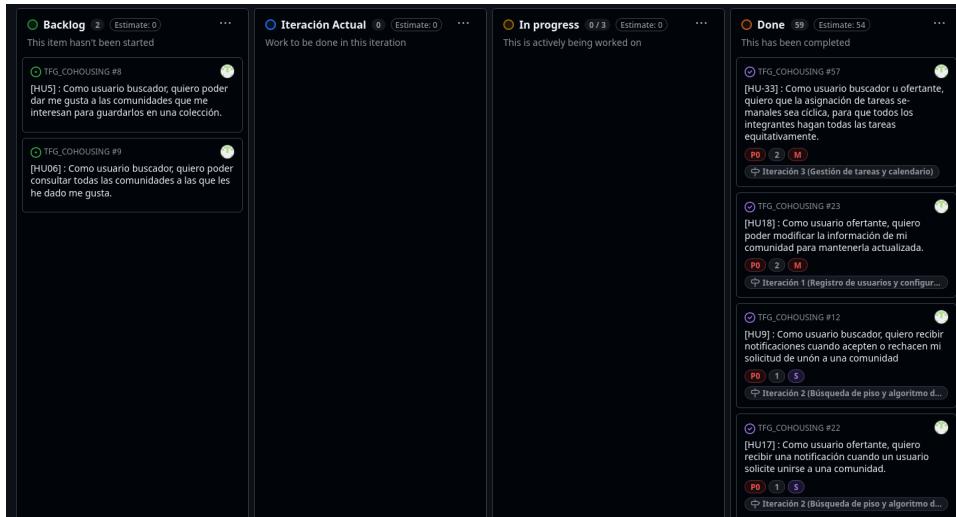


Figura 7.4: Imagen tablero GitHub Project

7.1.3. Contenerización con Docker

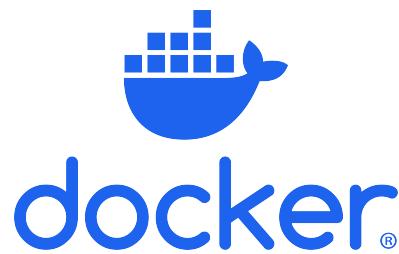


Figura 7.5: Logo docker

Cada microservicio y front-end disponen de su propio Dockerfile [7], un archivo mediante el cuál se configuran los recursos y dependencias necesarias para su ejecución en un entorno controlado. Esto permite crear un sistema modular donde cada microservicio se ejecuta de forma aislada en su contenedor específico.

A continuación se procede a mostrar el ejemplo de configuración de un Dockerfile para los microservicios en Java.

```
FROM openjdk:17-jdk-slim
COPY target/demo.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java","-jar","/app.jar"]
```

Como se puede observar, lo único que se realiza es copiar el .jar del microservicio ya compilado en el contenedor y ejecutarlo. Esta configuración permite desplegar servicios independientes en cualquier entorno compatible con Docker de forma sencilla y efectiva.

7.1.4. Orquestación con Docker Compose

Se emplea un fichero docker-compose.yml [6] para levantar todo el ecosistema de servicios que componen la aplicación. En general la estructura del archivo es la siguiente

- Construcción de microservicios desde sus contextos a partir del DockerFile de cada servicio.
- Definición de bases de datos PostgreSQL y servicio RabbitMQ.
- Red interna cohousing para comunicación entre contenedores.
- Uso de depends_on para gestionar el orden de arranque.

Finalmente para poder ejecutar la aplicación se levanta el entorno completo, esto permite ejecutar en local todos los servicios simultáneamente, para ello se ejecuta el siguiente comando.

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

Esta configuración permite una gestión modular, escalable y replicable de la aplicación, facilitando tanto el desarrollo local como el despliegue en entornos de producción.

7.2. Funcionalidades clave del sistema

A continuación se procederá a explicar las diferentes funcionalidades del sistema, mostrando su lógica y flujo de desarrollo.

7.2.1. Inicio de sesión y Registro

Con el objetivo de garantizar la seguridad dentro de la aplicación y asegurar la autenticación y autorización se ha aplicado un sistema de inicio de sesión basado en JWT (JSON Web Token) [18]. El flujo de esta funcionalidad es el siguiente:

- El usuario envía sus credenciales al *API Gateway*.
- El gateway se comunica con el microservicio *Gestión de Usuarios* donde se autentica al usuario y se genera el JWT si las credenciales son correctas.
- Este token es enviado al cliente y debe ser adjuntado en el header de las peticiones posteriores para validar la sesión.
- En el microservicio *API Gateway*, cada vez que un cliente accede a una ruta protegida, se aplica un filtro personalizado de Spring Security que intercepta la petición y valida la autenticidad del token JWT.

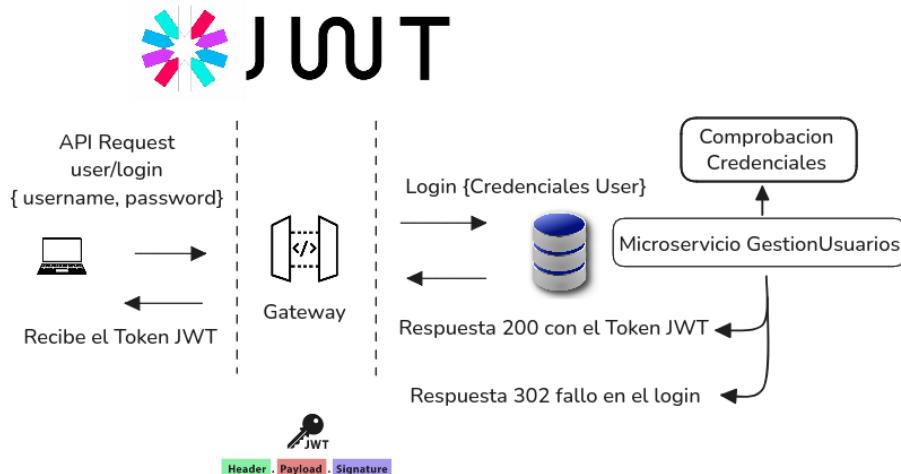


Figura 7.6: Flujo de autenticación a través del API Gateway.

Para implementar el esquema de autenticación basado en JWT representado en la figura 7.6, se han desarrollado los siguientes componentes en la aplicación Spring Boot:

1. Filtro de seguridad en la API Gateway:

Se definió un filtro de seguridad que se inserta en la cadena de filtros de Spring Security [33]. Este filtro es responsable de establecer los requisitos de seguridad correspondientes para cada ruta definida y de comprobar la validez del token JWT en cada solicitud a una ruta protegida.

2. Generación del token JWT:

Cuando los datos proporcionados por el usuario durante el proceso de inicio

de sesión sean correctos, se procede a generar un token JWT que contiene la información de autenticación del usuario, en nuestro caso el nombre de usuario y el rol.

3. Controlador de autenticación:

Se definió un controlador REST que gestiona las peticiones de autenticación. Este controlador valida las credenciales del usuario y, en caso de ser válidas, devuelve el token JWT en la respuesta.

Listing 7.1: Pseudocódigo del controlador login

```

1  function login(loginRequestDTO, response):
2
3      try:
4          # Intentar autenticar al usuario
5          token = authService.login(loginRequestDTO)
6
7          # Guardar el token como cookie de autenticacion
8          addAuthCookie(response, token)
9
10         # Obtener el rol del usuario desde el token
11         role = tokenService.getRoleFromToken(token)
12
13         # Devolver respuesta de exito con datos del
14         # usuario
15         return ResponseEntity.ok({
16             "message": "Login correcto",
17             "username": loginRequestDTO.username,
18             "role": role
19         })
20
21     except AuthenticationException:
22         # Si las credenciales no son validas, devolver
23         # error
24         return ResponseEntity.status(BAD_REQUEST).body({
25             "error": "Credenciales invalidas"
26         })

```

En el frontend simplemente se rellena el formulario de inicio de sesión, mostrado en la figura 7.7, enviando los datos necesarios para la autenticación.

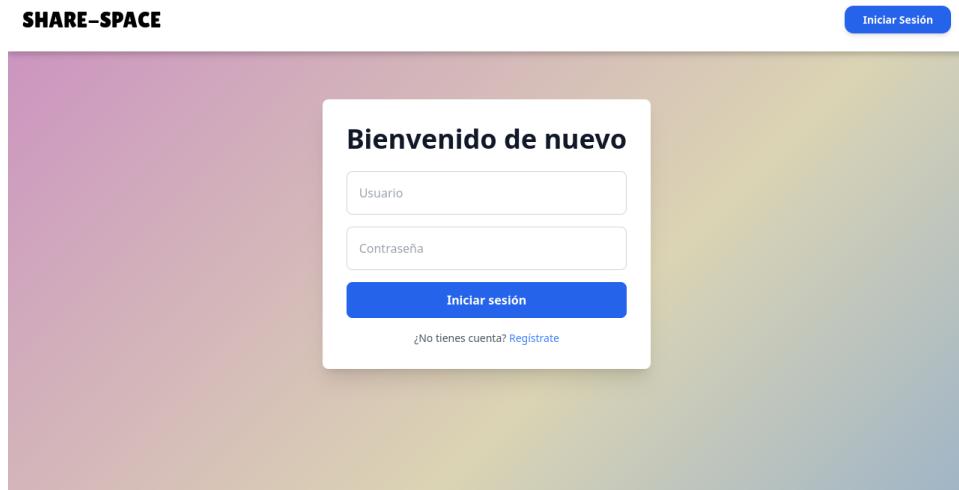


Figura 7.7: Pantalla del login en el front-end

7.2.2. Comunicación Asíncrona entre Microservicios

Para garantizar la resiliencia e independencia entre los distintos microservicios que componen el sistema, es fundamental que la comunicación entre ellos sea asíncrona ya que de otra forma podrían producirse cuellos de botella o incluso una caída del sistema entero ante algún fallo en uno de los servicios. Con el objetivo de implementar este tipo de comunicación, se ha integrado el broker de mensajería asíncrona **RabbitMQ**, que actúa como intermediario en el intercambio de mensajes entre microservicios.

Flujo de comunicación

El flujo seguido para establecer esta arquitectura de mensajería es el siguiente:

- Los microservicios publican eventos en colas específicas gestionadas por RabbitMQ.
- Otros microservicios se suscriben a dichas colas, reaccionando ante los mensajes recibidos.
- Esta arquitectura permite reducir el acoplamiento entre los microservicios, consiguiendo una mayor escalabilidad, tolerancia a fallos e independencia entre ellos.

Funcionamiento del sistema

La lógica general de funcionamiento se basa en los siguientes pasos:

1. El microservicio define un evento que producirá una comunicación con otro microservicio, por ejemplo solicitar la unión a una comunidad.

2. El microservicio productor registra una nueva cola específica en RabbitMQ que estará asociada al evento definido anteriormente.
3. Al producirse el evento, se crea un mensaje con los datos relevantes y se publica en dicha cola.
4. Los microservicios consumidores definen *listeners* (funciones observadoras) que se mantienen a la escucha.
5. Cuando el mensaje llega a la cola correspondiente, el *listener* lo recibe, extrae los datos y ejecuta una lógica específica en respuesta al evento.

Este patrón de comunicación mostrado en la figura 7.8 favorece un diseño desacoplado, reactivo y preparado para entornos distribuidos y de alta disponibilidad.

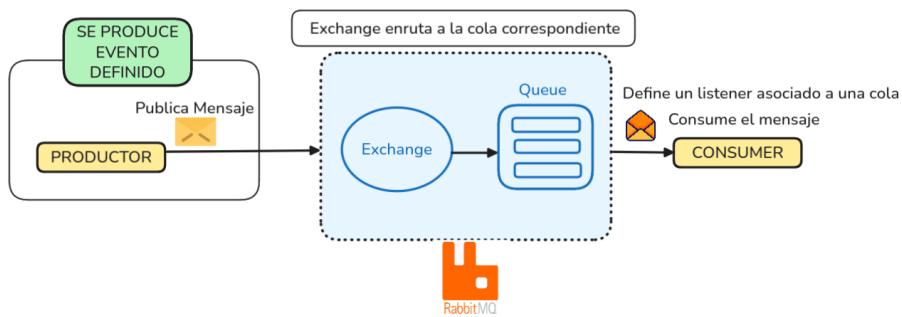


Figura 7.8: Flujo de eventos y comunicación entre microservicios usando RabbitMQ

7.2.3. Recomendación de Comunidades

Toda la lógica e implementación relacionada con la recomendación de comunidades se ha encapsulado en un microservicio independiente denominado **Recomendaciones**. Esta decisión se basa en que el módulo presenta suficientes reglas de negocio como para ser considerado una entidad lógica separada dentro del sistema.

Para la implementación del microservicio se ha utilizado el lenguaje de programación **Python**, apoyado en las siguientes bibliotecas:

- Pandas [26], para el tratamiento y análisis de datos.
- FastAPI [29], para la creación y estructuración de la API REST.

Algoritmo de recomendación

Con el objetivo de conseguir una recomendación personalizada para cada usuario se ha seguido un enfoque basado en técnicas de *clustering*, mediante el algoritmo **K-Means** [35]. Esto ha permitido agrupar las comunidades en función de

la similitud de sus parámetros de afinidad. Una vez organizados estos grupos, se ha identificado el más cercano al usuario calculando la similitud entre los valores de afinidad del usuario y los de las comunidades de cada grupo. La lógica general del sistema se puede resumir del siguiente modo:

- Las comunidades se agrupan en *clusters* en función de sus características individuales (afinidad, sociabilidad, limpieza, etc.).
- Se entrena un modelo KMeans únicamente con las comunidades.
- Se predice a qué *cluster* pertenecería un usuario en base a sus características.
- Se seleccionan las comunidades que pertenecen al mismo *cluster* que el usuario.
- Se ordenan esas comunidades según su cercanía al vector escalado del usuario, de forma que sea lo más personalizado posible.

Procedimiento de recomendación

Dentro del propio microservicio se ha seguido el siguiente flujo para generar las recomendaciones personalizadas para cada usuario:

1. Se recuperan los datos del usuario objetivo y de las comunidades desde sus respectivas bases de datos.
2. Se normalizan los parámetros de afinidad tanto del usuario como de las comunidades, esto es necesario para que la distancia euclídea tenga sentido.
3. Se entrena un modelo de K-Means con los datos escalados de las comunidades.
4. Se predice a qué *cluster* pertenecería el usuario, para ello se calcula la diferencia (distancia euclídea) entre el vector del usuario y el centroide de cada grupo, que funciona como representación del perfil idea de ese grupo de comunidades.
5. Se calcula la distancia euclídea entre cada comunidad del mismo *cluster* escogido y el vector escalado del usuario.
6. Se transforma la distancia en un porcentaje de afinidad (0 a 100).
7. Las comunidades del clúster se ordenan por mayor afinidad y se seleccionan las *n* más afines para formar la recomendación.

Listing 7.2: Pseudocódigo de la función recommend_communities_by_user

```

1 def recommend_communities_by_user(user, communities,
2                                     n_recommendations=10):
3
4     # Si no hay comunidades, devolvemos la lista vacia
5     if communities is None or len(communities) == 0:
6         return []
7
8     # Preprocesar y escalar datos, es necesario tener los
9     # datos escalados y en un formato correcto para
10    # aplicar K-Means

```

```

8     user_scaled, communities_scaled = preprocess_data(
9         user, communities)
10
11     #Si no hay usuarios, devolvemos error
12     if user_scaled is None or communities_scaled is None:
13         raise ValueError("Error al escalar los datos")
14
15     # Definimos el numero de clusters que tendremos
16     n_clusters = min(3, len(communities_scaled))
17     if n_clusters == 0:
18         return []
19
20     # Entrenar modelo KMeans
21     kmeans = KMeans(n_clusters=n_clusters, random_state
22                      =42)
23     kmeans.fit(communities_scaled)
24
25     # Predecir cluster del usuario y de las comunidades
26
27     # K-Means nos dice a que cluster pertenece el usuario
28     user_cluster = kmeans.predict(user_scaled)[0]
29
30     # K-Means predice en que cluster cae cada comunidad
31     # Devuelve un vector indicando en que cluster se
32     # encuentra cada comunidad
33     communities_clusters = kmeans.predict(
34         communities_scaled)
35
36     # Calculamos el centroide del cluster del usuario
37     # El centroide es un vector con las coordenadas
38     # medias de ese grupo
39     user_center = kmeans.cluster_centers_[user_cluster]
40
41     # Calcular distancias entre las comunidades del
42     # cluster escogido y el centroide del usuario
43     distances = []
44     # Recorremos todas las comunidades
45     for i, community_scaled in enumerate(
46         communities_scaled):
47         # Solo nos fijamos en aquellas que estan en el
48         # mismo cluster que el usuario
49         if communities_clusters[i] == user_cluster:
50             # Se calcula la distancia euclidea (np.linalg
51             # .norm(...)) entre la comunidad[i] y el
52             # usuario
53             dist = distancia_euclidiana(community_scaled,
54                                         user_scaled[0])
55             #Esto devuelve una lista de pares indicando
56             # primero el id de la comunidad y segundo

```

```

        un valor
45     # que indica que tan lejos estan de el
46     distances.append((i, dist))

47
48     if not distances:
49         return []
50
51     # Normalizar distancias para calcular afinidad
52     max_distance = max(dist for _, dist in distances)
53     if max_distance == 0:
54         max_distance = 1

55
56     MIN_AFFINITY = 30
57     affinities = []
58     for i, dist in distances:
59         if dist es finita:
60             affinity = 100 * (1 - dist / max_distance)
61         else:
62             affinity = 0
63         # Limitar entre MIN_AFFINITY y 100
64         affinity = max(min(affinity, 100), MIN_AFFINITY)
65         affinities.append((i, round(affinity)))

66
67     # Ordenar comunidades por afinidad descendente
68     affinities.sort(descending by affinity)

69
70     # Construir lista final de recomendaciones
71     recommended = []
72     for i, affinity in affinities[:n_recommendations]:
73         recommended.append(
74             community_to_dict(communities[i]) + {"affinity": affinity}
75         )

76
77     return recommended

```

Este enfoque permite recomendar comunidades al usuario de forma personalizada y priorizando la afinidad entre ambos.

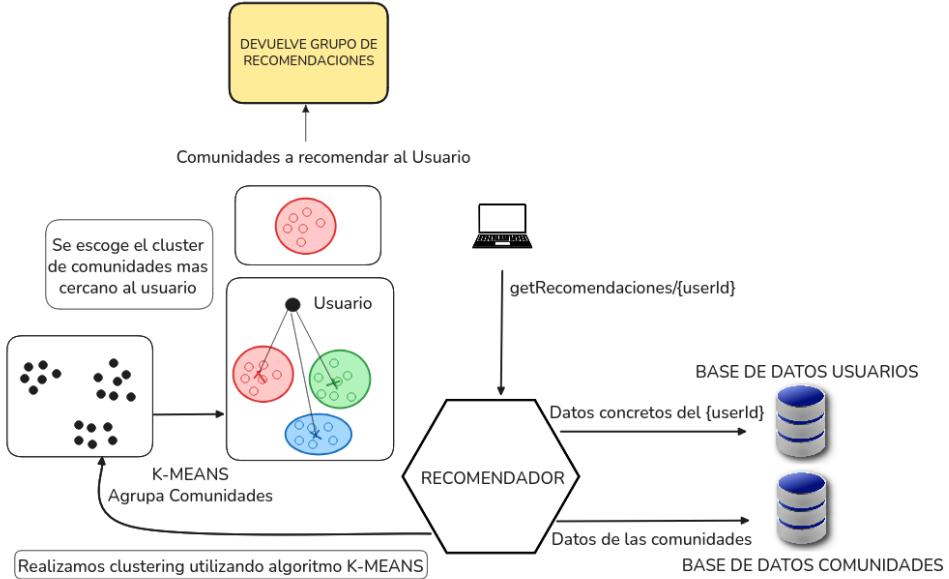


Figura 7.9: Explicación del sistema de recomendación para usuarios

Este sistema de recomendación mostrado en la figura 7.9 es especialmente útil en escenarios de *cold start*, ya que no requiere un historial de interacciones del usuario, sino que se basa en similitud de perfiles. Cabe destacar que la aplicación permite refinar aún más las recomendaciones de comunidades mediante filtros adicionales, como la distancia al usuario y el rango de precio. Esto proporciona una lista de comunidades más personalizada y ajustada a las preferencias del usuario.

Finalmente, tal como se observa en la figura 7.10, el usuario dispone de opciones para aplicar filtros adicionales y se le muestran las comunidades recomendadas junto a sus datos más representativos.

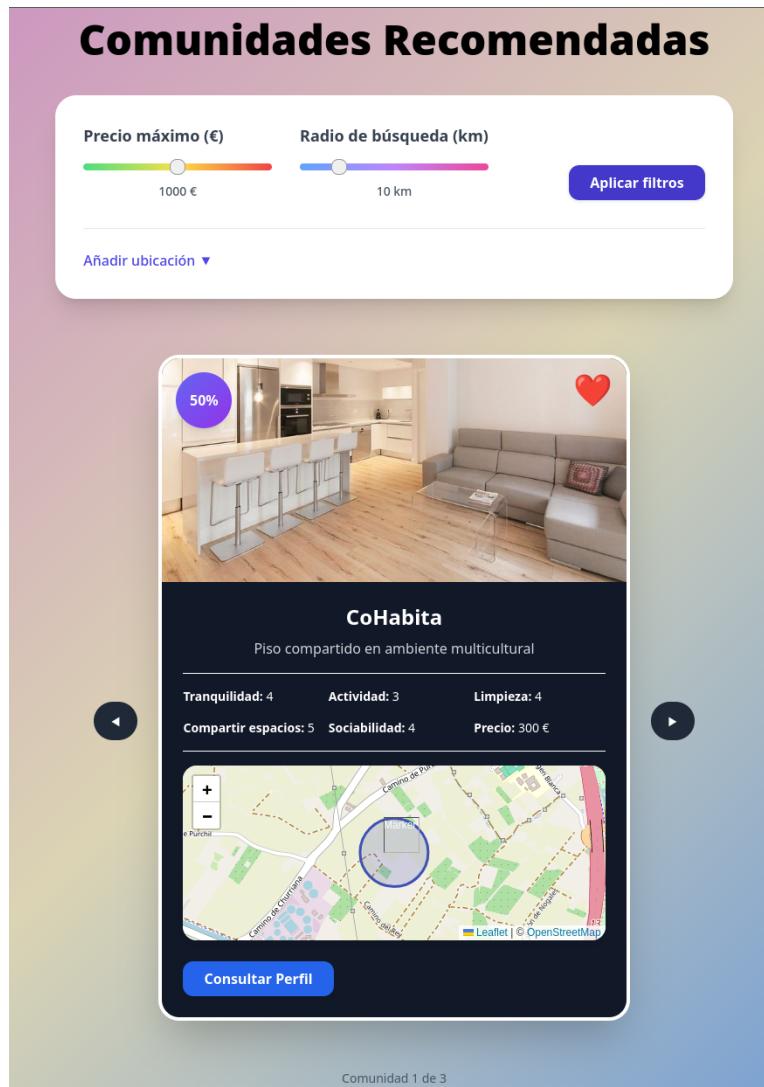


Figura 7.10: Pantalla de recomendaciones en el front-end

7.2.4. Proceso de Unión a una Comunidad

El proceso de unión de un usuario a una comunidad involucra la interacción de varios de los microservicios que componen el sistema, entre ellos los microservicios de *Comunidad*, *Solicitudes* y *GestionUsuarios*. Para la comunicación entre dichos microservicios se usará como broker RabbitMQ [27]. El flujo para realizar esta operación dentro del sistema es el siguiente:

1. El usuario solicita unirse a una comunidad a través de la llamada al endpoint `unirse-comunidad` del microservicio *GestionComunidades*.
2. Este microservicio genera un evento y lo publica en RabbitMQ.

3. El microservicio de *Solicitudes* escucha el evento, registra la solicitud y la notifica al administrador de la comunidad.
4. El administrador puede aceptar o rechazar la solicitud mediante el microservicio de *Solicitudes*. Al aceptar o rechazar el microservicio genera un evento y lo publica en RabbitMQ
5. El microservicio de *GestionComunidades* escucha el evento y si el administrador ha aceptado, une al usuario a la comunidad. Finalmente este microservicio emite un evento y lo publica en RabbitMQ.
6. El microservicio de *Solicitudes* escucha el evento, registra la respuesta y la notifica al usuario buscador.

Este flujo de comunicación entre los diferentes microservicios queda resumido en la figura 7.11, mostrando como los diferentes microservicios van generando eventos en función del estado de la comunicación actual y como todo el proceso es orquestado por RabbitMQ.

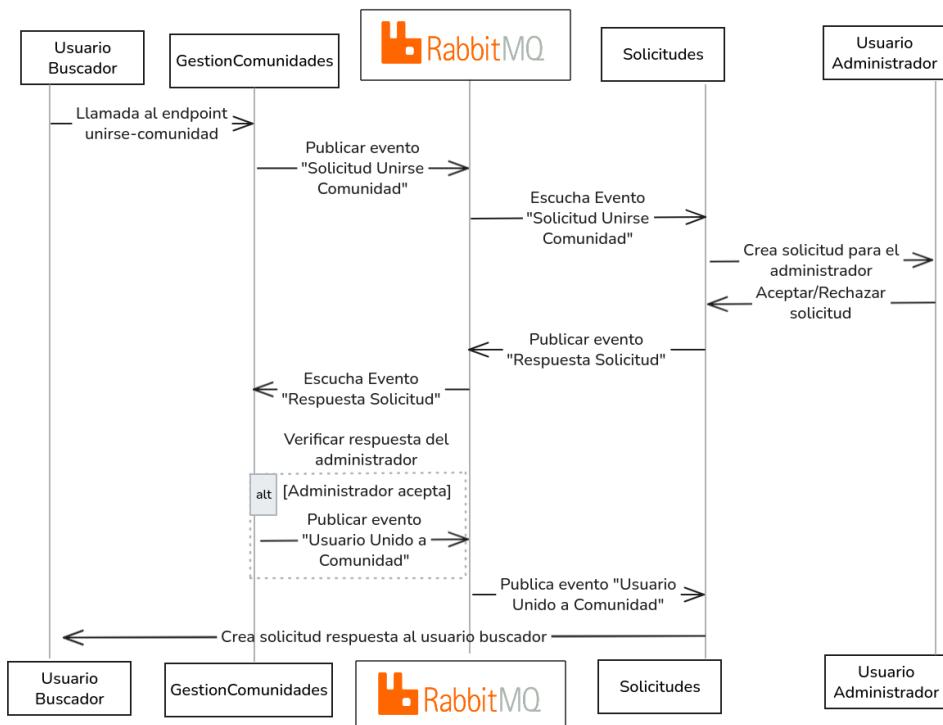


Figura 7.11: Esquema del flujo para la unión de un usuario a una comunidad

7.2.5. Gestión de comunidades

El microservicio de **Gestión de Comunidades** encapsula el mayor volumen de lógica de negocio dentro del sistema, siendo responsable de coordinar todos los aspectos relacionados con las comunidades: sus eventos, la asignación y la gestión de tareas.

Entre las funcionalidades más relevantes implementadas en este servicio, destacan:

- **Reparto automático de tareas:** El sistema realiza una distribución automática de las tareas entre los usuarios que forman parte de una comunidad, siguiendo un orden cíclico. Este reparto se ejecuta de forma semanal, asegurando así un equilibrio en la carga de trabajo entre los miembros.
- **Administración de tareas por parte del usuario:** Los usuarios tienen la posibilidad de consultar y reorganizar sus tareas mediante una interfaz de calendario interactivo. Esta herramienta permite una gestión más intuitiva y visual de sus responsabilidades.

Reparto Automático de Tareas

El reparto automático de tareas tiene como objetivo asegurar una distribución equitativa del trabajo entre los miembros de cada comunidad. Para ello, se sigue un algoritmo cíclico que asigna las tareas disponibles a los usuarios de manera rotativa.

Con este objetivo en mente se ha diseñado el siguiente algoritmo teniendo en cuenta:

- El conjunto de usuarios activos en la comunidad.
- Las tareas asociadas a la comunidad.
- El historial de asignaciones anteriores, para mantener el reparto equitativo.

De esta manera, el sistema garantiza una planificación justa y eficiente, mejorando la organización interna de cada comunidad.

Lógica del reparto cíclico

El reparto de tareas se ha diseñado siguiendo una lógica sencilla pero equitativa. A continuación, se describe el comportamiento implementado:

- Las tareas se distribuyen de forma **cíclica** entre los integrantes de la comunidad. De este modo, con el tiempo, todos los miembros participarán en todas las tareas.
- Al crear una nueva tarea, se define el **número de participantes requeridos**.
- En el primer reparto, se asignan por defecto los primeros usuarios disponibles. De forma que, si hay cuatro tareas y tres miembros, se asignará la primera tarea al primer miembro; la segunda, al segundo y así sucesivamente en orden cíclico.

- En repartos sucesivos, se utiliza un índice rotativo almacenado en la comunidad para determinar desde qué usuario comenzar la asignación. Este índice se va actualizando con cada tarea para que todos los usuarios participen proporcionalmente en alguna tarea en cada iteración. Si un nuevo usuario se une a la comunidad, se incorpora automáticamente a la lista de usuarios rotativos.

Para lograr la ejecución de forma semanal del reparto se ha utilizado la anotación `@Scheduled(cron =)`. Con esta anotación podemos ejecutar este método definido en el backend de forma periódica; en nuestro caso cada semana.

Listing 7.3: Pseudocódigo de la función repartirTareasSemanalmente

```

1  @Scheduled(cron = "0 0 23 * * SUN")
2  @Transactional
3  def void repartirTareasSemanalmente():
4      # Obtener todas las comunidades
5      comunidades = obtenerTodasLasComunidades()
6      hoy = fechaActual()
7      # Iterar sobre cada comunidad
8      for comunidad in comunidades:
9
10         # Repartir tareas de forma ciclica
11         repartirTareasCiclico(comunidad.id)
12
13         # Generar resumen semanal
14         generarResumenSemanal(comunidad, hoy)
15
16         # Obtener lista de identificadores de usuarios de
17         # la comunidad
18         idUsuarios = lista(comunidad.integrantes)
19
20         # Crear notificación de reparto de tareas
21         payload = crearNotificacionReparto(
22             comunidad.id,
23             comunidad.nombre,
24             "Se ha realizado el reparto de tareas en tu
25             comunidad",
26             idUsuarios
27         )
28
29         # Enviar notificación a la cola de mensajería (
30         # RabbitMQ)
31         enviarMensajeCola(
32             EXCHANGE_REPARTO_TAREAS,
33             ROUTING_KEY_REPARTO_TAREAS,
34             payload
35         )

```

Notificación y visualización

Para asegurar que los usuarios estén al tanto de sus tareas, el sistema emite notificaciones cuando se realiza el reparto semanal:

- De manera semanal se emite una **notificación automática a todos los integrantes de la comunidad** de forma que se les avise del nuevo reparto. Esto se implementa con la definición de un evento dentro de la función de reparto semanal que es gestionada por RabbitMQ.

A continuación pasamos a comentar la Administración de tareas por parte del usuario.

7.2.6. Administración de Tareas por Parte de los Usuarios

Con el objetivo de facilitar la administración de tareas dentro de la comunidad, se ha desarrollado una interfaz interactiva que permite a los usuarios visualizar, asignar y reprogramar sus tareas mediante un calendario dinámico.

Buscando la mejor experiencia de usuario posible, se ha implementado una funcionalidad de *drag and drop* (arrastrar y soltar), que permite al usuario mover las tareas a diferentes franjas horarias en el calendario, asignándoles así una fecha y hora de inicio de forma intuitiva.

El flujo de funcionamiento del componente es el siguiente:

1. **Carga de las tareas del usuario:** En primer lugar, se obtienen todas las tareas correspondientes al usuario mediante el endpoint `/tareas/idUsuario`.
 - Las tareas que no tienen fecha asignada se muestran en una columna lateral para ser planificadas por el usuario.
 - Las tareas ya planificadas se adaptan al formato requerido por la biblioteca FullCalendar [11](`id, title, start`) y se representan directamente en el calendario.
2. **Funcionalidad Drag and Drop:** Las tareas sin fecha se convierten en elementos arrastrables mediante la clase `Draggable` de FullCalendar. Esto permite al usuario arrastrar una tarea desde la columna lateral y soltarla en una franja del calendario.
3. **Asignación de fecha al soltar la tarea:** Cuando el usuario suelta una tarea sobre el calendario, se ejecuta el método `handleReceive`. Este método extrae el `id` de la tarea y la fecha seleccionada, y envía esta información al backend mediante el método `updateTarea` para registrar la nueva fecha.
4. **Reprogramación de tareas:** Si el usuario mueve una tarea ya planificada a una nueva franja horaria dentro del calendario, se ejecuta el método `handleEventDrop`, el cual recalculará la nueva fecha y la actualizará en el backend.

Listing 7.4: Pseudocódigo de tareas drag and drop y asignacion fecha al soltar tarea

```

41     setTareasSinFecha( tareasSinFecha.filter(t -> t.
42         id != id) )
43
44     # Agregar la tarea a la lista con fecha
45     setTareasConFecha( tareasConFecha + { id, title:
46         info.event.title, start: fecha } )
47
48     catch error:
49         # En caso de error revertir el drag and drop
          log(error)
          info.revert()

```

Gracias a esta implementación, se ha creado una vista de administración de tareas (figura 7.12) que permite al usuario gestionarlas de forma dinámica. Desde esta interfaz se puede tanto planificar las tareas por primera vez como reorganizarlas posteriormente.

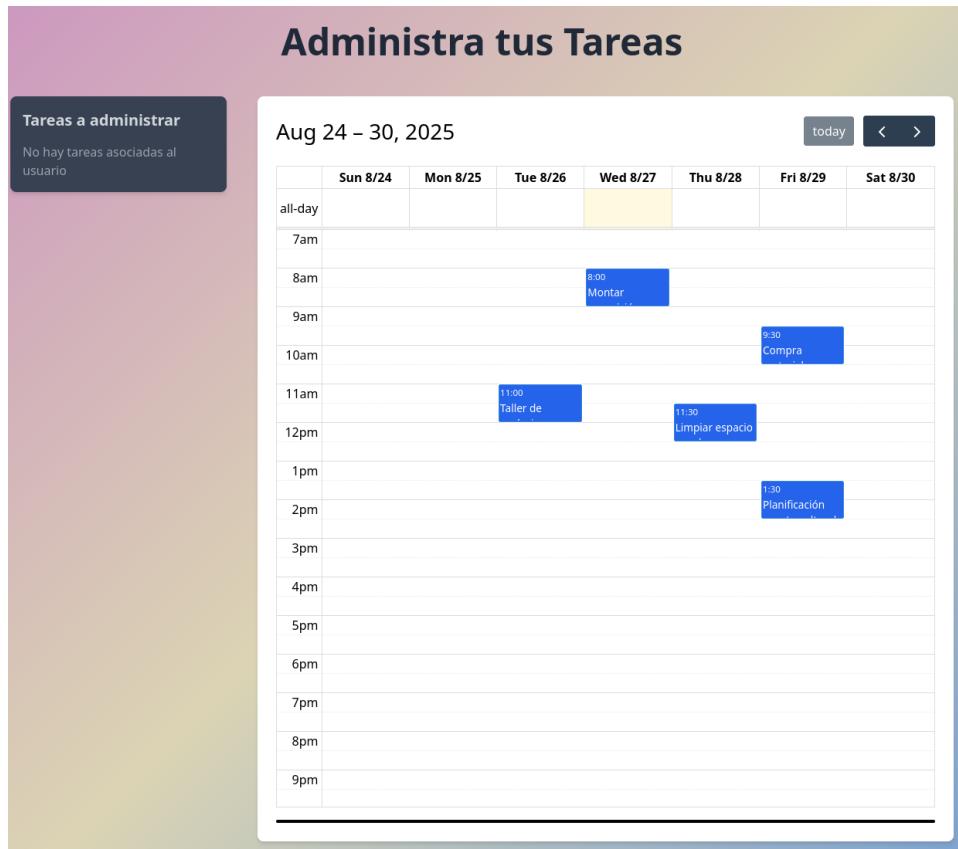


Figura 7.12: Vista del usuario para la administración de tareas dinámicamente

Capítulo 8

Pruebas

En este capítulo se describen las pruebas realizadas sobre los distintos microservicios que componen la aplicación, así como sobre el frontend. Estas pruebas serán esenciales para asegurar la calidad del software, detectar errores antes de llegar a estados más avanzados y garantizar un comportamiento estable.

Además, se han integrado en el flujo de integración continua (CI) a través de las GitHub Actions, de manera que se ejecutan automáticamente en cada *subida* a las ramas del repositorio. De este modo, las pruebas forman parte activa del ciclo de vida del desarrollo.

Para cada componente se han definido distintos tipos de pruebas con el objetivo de cubrir tanto la lógica individual como el comportamiento global del sistema:

1. **Pruebas Unitarias:** Verifican el comportamiento de clases y funciones de forma aislada, utilizando mocks para simular dependencias externas. Se emplean principalmente JUnit y Mockito para los servicios en Spring Boot, Pytest para los microservicios en Flask y Jest [8] con React Testing Library para el frontend.
2. **Pruebas de Sistema (End to End):** Validan flujos completos que implican a varios componentes del sistema, especialmente los endpoints más críticos.
3. **Pruebas de Widgets (Frontend):** Evalúan la interacción de los elementos de la interfaz de usuario, garantizando su correcto comportamiento frente a las acciones del usuario.

8.1. Microservicio de Gestión de Usuarios

8.1.1. Pruebas Unitarias

Se han probado los siguientes métodos de la capa de servicio:

- `registrarUsuario(UserDTO dto)`: Verifica el registro de un nuevo usuario comprobando si el usuario o alguno de sus datos no han sido previamente registrados. En caso contrario, lanza una excepción controlada.
- `getUsuarioPorId(Long id)`: Comprueba la correcta recuperación de un usuario existente por ID. También se verifica el caso de error mediante una excepción.
- `getComunidadesGuardadas(Long userId)`: Comprueba que se recuperan correctamente las comunidades guardadas por un usuario, incluyendo el caso en que la lista esté vacía.

8.1.2. Pruebas End to End

- `POST /user/login`: Se comprueba que con credenciales válidas se obtiene un JWT en la respuesta. También se prueba el caso de credenciales inválidas.
- `GET /user/{id}`: Verificación de que se devuelve la información del usuario correspondiente, y se maneja correctamente el caso de ID inexistente.
- `PUT /user/{id}`: Se prueba la actualización de datos del usuario y que los cambios se persisten correctamente en la base de datos.

Los test han sido pasados todos correctamente, como se puede ver en la imagen 8.1

```
[INFO] Results:
[INFO]
[INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco:0.8.12:report (report) @ demo ---
[INFO] Loading execution data file /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios/Gestion
[INFO] Analyzed bundle 'demo' with 17 classes
[INFO]
[INFO] --- jar:3.4.2:jar (default-jar) @ demo ---
[INFO] Building jar: /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios/GestionUsuarios/demo/
[INFO]
[INFO] --- spring-boot:3.4.4:repackage (repackage) @ demo ---
[INFO] Replacing main artifact /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios/GestionUsua
archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microser
r.original
[INFO]
[INFO] --- jacoco:0.8.12:prepare-agent (default) @ demo ---
[INFO] argLine set to -javaagent:/home/alonso/.m2/repository/org/jacoco/org.jacoco.agent/0.8.12/org
i/Cuarto/TFG/TFG_COHOUSING/microservicios/GestionUsuarios/demo/target/jacoco.exec
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ demo ---
[INFO] Copying 3 resources from src/main/resources to target/classes
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.13.0:compile (default-compile) @ demo ---
[INFO] Nothing to compile - all classes are up to date.
[WARNING] Overwriting artifact's file from /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios
/home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios/GestionUsuarios/demo/target/classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ demo ---
[INFO] skip non existing resourceDirectory /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios
[INFO]
[INFO] --- compiler:3.13.0:testCompile (default-testCompile) @ demo ---
[INFO] Nothing to compile - all classes are up to date.
[INFO]
[INFO] --- surefire:3.5.2:test (default-test) @ demo ---
[INFO] Skipping execution of surefire because it has already been run for this configuration
[INFO]
[INFO] --- jacoco:0.8.12:report (report) @ demo ---
[INFO] Loading execution data file /home/alonso/uni/Cuarto/TFG/TFG_COHOUSING/microservicios/Gestion
[INFO] Analyzed bundle 'demo' with 17 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 14.378 s
[INFO] Finished at: 2025-08-26T13:54:41+02:00
[INFO] -----
```

Figura 8.1: Muestra del resultado de los test del microservicio Gestión de Usuarios

8.2. Microservicio de Gestión de Comunidades

Este microservicio contiene la lógica central del sistema, especialmente en cuanto a tareas colaborativas y reparto de responsabilidades.

8.2.1. Pruebas Unitarias

- `encontrarPorNombre(String name)` y `encontrarPorId(Long id)`: Verificación de la recuperación correcta de comunidades por nombre o ID.
- `crearComunidad(CommunityDTO dto)`: Se prueba la creación de comunidades, validando que no exista ya una con el mismo nombre.
- `asignarTareasSemanales()`: Se comprueba que el algoritmo de reparto automático semanal de tareas se ejecuta de forma correcta y equilibrada.
- `addTareaAComunidad(Long communityId, TaskDTO dto)` y `addEventoAComunidad(Long communityId, EventDTO dto)`: Validación de la correcta asociación de tareas y eventos a una comunidad.

Listing 8.1: Pseudocódigo del test crearComunidad

```

1  test "crear comunidad" {
2      // Creamos una comunidad falsa
3      lifestyleDTO = new LifestyleDTO(1,1,1,1,1)
4      dto = new CommunityDTO(
5          id = 1L,
6          name = "EcoWarning",
7          description = "Comunidad preocupada con el medio
8              ambiente",
9          ownerId = 7L,
10         lifestyle = lifestyleDTO,
11         members = [1L, 4L],
12         imageUrl = "https://foto",
13         latitude = 40.423,
14         longitude = 3.45,
15         address = "Calle Falsa",
16         price = 300.45,
17         capacity = 2
18     )
19
20     // Mock: metodo para guardar la nueva comunidad
21     mock communityRepository.save(any Community)
22         -> devuelve la misma Comunidad pasada
23
24     // Probamos el metodo registrarComunidad
25     comunidad = communityService.registrarComunidad(dto)
26
27     // Comprobamos que la comunidad guardada tiene el
28     // nombre de la comunidad nueva registrada
29     assert comunidad.getName() == "EcoWarning"
30 }
```

8.2.2. Pruebas End to End

- POST /community/{id}/unir: Verifica que un usuario puede unirse correctamente a una comunidad.
- GET /community/{id}/tasks: Comprueba la obtención de tareas para su posterior planificación.
- DELETE /community/{id}: Se prueba la eliminación completa de una comunidad y la cascada de datos relacionada.

8.3. Microservicio Recomendador

Este microservicio ha sido implementado con Flask y probado mediante pytest.

8.3.1. Pruebas Unitarias

- `obtener_todas_comunidades()`: Verifica que se recuperan correctamente todas las comunidades desde la base de datos.
- `get_usuario_perfil(user_id)`: Prueba la obtención del perfil completo del usuario.
- `recomendar_comunidades(UserDTO dto)`: Valida que el algoritmo de recomendación devuelve resultados coherentes según los intereses y preferencias del usuario.

8.3.2. Pruebas End to End

- GET `/recomendaciones`: Comprueba que se devuelven comunidades recomendadas al usuario autenticado.
- GET `/recomendaciones-filtradas?filtro=...`: Se valida que los filtros personalizados funcionan correctamente y modifican los resultados retornados.

Listing 8.2: Pseudocódigo del test recomendacionesFiltradas

```

1  test "recomendaciones filtradas" {
2      // Se carga el usuario falso creado para las pruebas
3      usuario = test_user
4      user_id = usuario.id
5      // Se define parametros de precio y distancia para el
6      // filtro
7      parametros = {
8          "precio": 400,
9          "distancia": 5    # 5 km de radio
10     }
11     // Mock: get_usuario_por_id devuelve el usuario de
12     // prueba
13     // Mock: get_comunidades_incompletas devuelve
14     // comunidades de prueba
15     mock get_usuario_por_id -> usuario
16     mock get_comunidades_incompletas -> test_communities
17
18     // Se realiza la llamada al endpoint
19     respuesta = client.get("usuarios/recomendaciones -
20         filtradas/{user_id}", params=parametros)
21
22     // Se realizan las comprobaciones de la respuesta del
23     // endpoint
24     assert respuesta.status_code == 200
25     datos = respuesta.json()
26     assert len(datos) == 1                      // Solo una
27     comunidad
28     assert datos[0]["name"] == "Comunidad Cerca"
29 }
```

8.4. Microservicio API Gateway

8.4.1. Pruebas Unitarias

- `extraerToken(HttpHeaders headers)`: Valida que el token JWT es correctamente extraído del encabezado Authorization.
- `validarToken(String token)`: Comprueba que el token es válido y no ha expirado.

Listing 8.3: Pseudocódigo del test `validarToken` en `GatewayServiceTest`

```

1  test "validarToken" {
2      // En primer lugar creamos un mock de los servicios que
3      // intervienen en la prueba
4      mock JwtUtil, GatewayFilterChain, ServerWebExchange,
5          ServerHttpRequest, ServerHttpResponse, HttpCookie
6
7      cookie.value = "tokenValido"
8      request.cookies["auth-token"] = cookie
9      exchange.request = request
10     exchange.response = response
11     jwtUtil.validate("tokenValido") -> true
12     chain.filter(exchange) -> empty
13
14     // Realizamos la llamada al metodo a probar
15     // JwtAuthenticationFilter
16     result = JwtAuthenticationFilter(jwtUtil).filter(
17         exchange, chain)
18
19     // Comprobaciones finales
20     expect(result).completes()
21     verify(chain.filter(exchange)).calledOnce()
22     verify(response.setComplete()).neverCalled()
23 }
```

8.4.2. Pruebas End to End

- Se verifica que las peticiones entrantes se enrutan correctamente hacia los microservicios destino según el path y el método HTTP.

8.5. Microservicio de Solicitudes

8.5.1. Pruebas Unitarias

- `getSolicitudesUsuario(Long userId)`: Valida que se obtienen todas las solicitudes activas del usuario.
- `deleteSolicitudes(Long userId)`: Prueba que se eliminan correctamente todas las solicitudes del usuario.

- `aceptarSolicitudUnion(Long solicitudId)`: Verifica el cambio de estado de la solicitud a Aceptada.

Listing 8.4: Pseudocódigo del test `aceptarSolicitudUnion`

```

1  test "Aceptar Solicitud Union" {
2
3      // Preparar datos de prueba
4      solicitudId = 1
5      solicitud = crearSolicitud(
6          id = solicitudId,
7          userOrigenId = 10,
8          communityId = 20
9      )
10
11     // Mock: al buscar la solicitud, devolver el objeto
12     // creado
12     mock(solicitudesRepository.findById(solicitudId)).
13         returns(Optional.of(solicitud))
14
15     // Ejecutar la función a probar
15     response = solicitudesService.aceptarSolicitudUnion(
16         solicitudId)
17
18     // Verificar la respuesta HTTP
18     assert response.statusCode == OK
19     assert "Solicitud aceptada" en response.body
20
21     // Verificar que se envió un mensaje a RabbitMQ
22     verify(rabbitTemplate.convertAndSend(
23         EXCHANGE_NAME,
24         "comunidad.response",
25         cualquierObjeto(UnionResponseDTO)
26     ), veces=1)
27
28     // Verificar que la solicitud fue eliminada del
28     // repositorio
29     verify(solicitudesRepository.deleteById(solicitudId),
30         veces=1)
30 }
```

8.5.2. Pruebas End to End

- `GET /solicitudes/{id}`: Se obtiene la solicitud específica en función de Id.
- `GET /solicitudes/usuario/{id}`: Devuelve las solicitudes asociadas al usuario.

8.6. Frontend

Para las pruebas del frontend, desarrollado en React con TypeScript, se han utilizado las bibliotecas React Testing Library y Jest.

8.6.1. Pruebas de Componentes

- Componente de comunidades recomendadas: Verifica que se renderizan correctamente los elementos y que los botones de acción funcionan como se espera.
- Componente de detalles de comunidad: Comprueba que los datos se cargan correctamente y que los botones permiten unirse o abandonar una comunidad.
- Panel de inicio: Se prueba la carga condicional del contenido según el rol del usuario (administrador, miembro) y su pertenencia a una comunidad.
- Solicitudes: Se valida que se visualizan y actualizan correctamente las solicitudes activas del usuario.

Listing 8.5: Pseudocódigo del test de visualización y actualización de solicitudes

```

1  test "Mostrar Solicitudes"{
2
3      // Mockear la API para devolver solicitudes de prueba
4      mock(getSolicitudesUsuario(userId=123)).returns([
5          { id: 1, tipo: "compleja", descripcion: "Solicitud de prueba 1" },
6          { id: 2, tipo: "basica", descripcion: "Solicitud de prueba 2" }
7      ])
8      mock(aceptarSolicitud, rechazarSolicitud,
9          eliminarSolicitud).returns({})
10
11     // Renderizar la pagina con MemoryRouter
12     render(SolicitudesPage, route="/solicitudes/123")
13
14     // Comprobar que las solicitudes aparecen
15     assert screen.findByText("Solicitud #1") exists
16     assert screen.findByText("Solicitud de prueba 1") exists
17     assert screen.findByText("Solicitud #2") exists
18     assert screen.findByText("Solicitud de prueba 2") exists
19 }
20 test "Actualizar Solicitudes"{
21
22     // Renderizar la pagina con MemoryRouter
23     render(SolicitudesPage, route="/solicitudes/123")
24
25     // Seleccionar y clickar el boton de actualizar

```

```

25     updateButton = screen.getByRole("button", name="Actualizar solicitudes")
26     fireEvent.click(updateButton)
27
28     // Verificar que la primera solicitud sigue visible
29     assert screen.findByText("Solicitud #1") exists
30 }
```

8.7. Ejecución y Cobertura de Pruebas

Gracias a la integración de las pruebas en el flujo de CI mediante GitHub Actions, estas se ejecutan automáticamente tras cada cambio. Como resultado:

- Se asegura que cualquier modificación del código no rompa funcionalidades existentes.
- Se generan informes automáticos de cobertura de código mediante Codecov.
- Se garantiza que el desarrollo sigue un enfoque orientado a la calidad desde sus primeras etapas.

A continuación, se muestra una captura del informe de cobertura generado tras un merge exitoso a la rama **develop**, imagen 8.2, se puede observar que el porcentaje de código cubierto tras los test realizados es del 42 por ciento.

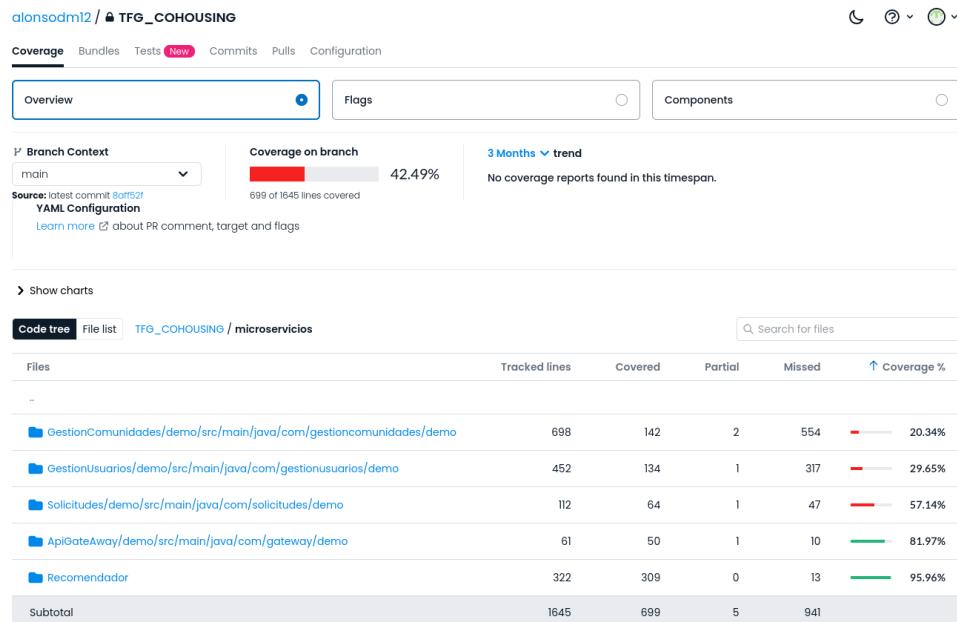


Figura 8.2: Muestra del informe de cobertura por Codecov

Capítulo **9**

Despliegue

En este capítulo se va a proceder a explicar los pasos seguidos para realizar el despliegue en la nube de la aplicación. Para ello se han utilizado dos plataformas diferentes: **Github Pages**, que ha permitido desplegar el frontend y **Railway** [28] donde se han desplegado los 5 microservicios, las tres bases de datos PostgreSQL y un servicio de RabbitMQ. A continuación, se va a mostrar cómo se han realizado ambos despliegues.

9.1. Despliegue del frontend



Figura 9.1: Logo Github Pages

Para el despliegue del frontend se ha utilizado la plataforma **GitHub Pages**, una herramienta gratuita que ofrece GitHub para desplegar sitios estáticos. En nuestro caso, se trata de un proyecto React compilado.

El procedimiento seguido consiste en:

1. Instalar en el proyecto React la dependencia `gh-pages` mediante el comando:

```
npm install --save-dev gh-pages
```

2. Configurar la variable `homepage` en el archivo `package.json` para indicar la URL de despliegue.
3. Ejecutar el comando:

```
npm run deploy
```

Para integrar este proceso dentro del flujo de desarrollo de la aplicación, el frontend se ha desplegado automáticamente cada vez que se realizaba un **merge** a la rama `main` (equivalente a producción), gracias a una acción específica configurada en **GitHub Actions**.

Finalmente, para que el frontend pueda realizar las llamadas correspondientes tanto en local como en producción, se han configurado una serie de archivos de entorno (`.env`) que determinan la ruta de consulta correcta en cada caso.

Enlace al frontend desplegado: [Share-Space](#)

9.2. Despliegue del backend



Figura 9.2: Logo Railway

Para el despliegue del backend se ha utilizado Railway, un PaaS (Platform as a Service) en la nube que facilita el despliegue de servicios directamente desde repositorios de GitHub, Dockerfiles o plantillas de servicios.

Dentro de Railway se han definido dos proyectos independientes, imagen 9.3 e imagen 9.4, en los que se han desplegado los distintos servicios que componen el sistema.

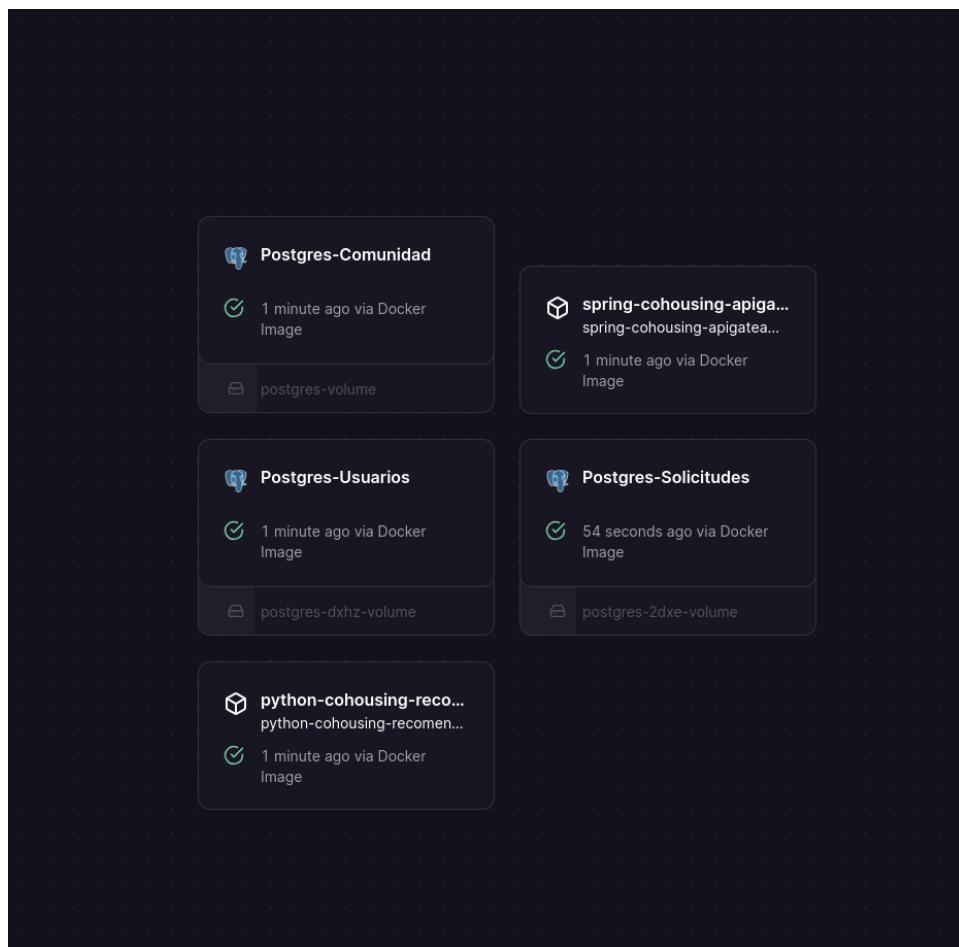


Figura 9.3: Muestra del tablero de despliegue del primer proyecto en Railway

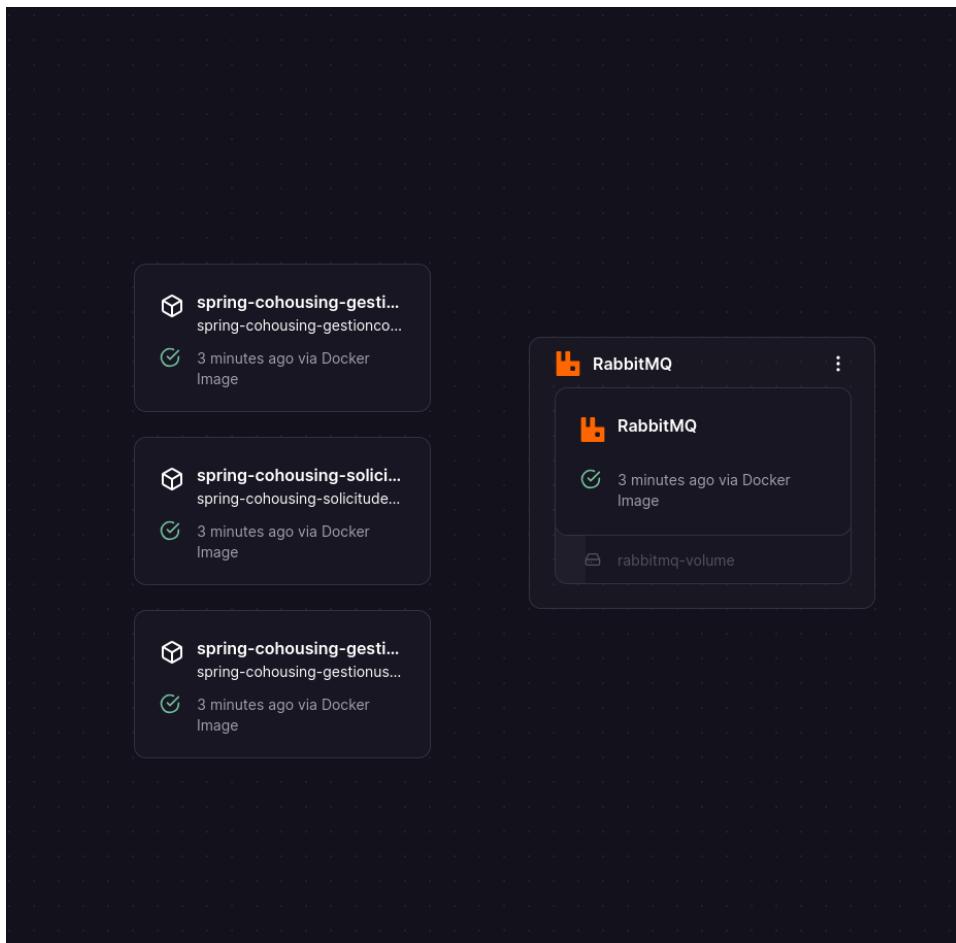


Figura 9.4: Muestra del tablero de despliegue del segundo proyecto en Railway

Cada microservicio ha sido desplegado en Railway a partir de su respectivo Dockerfile subido a DockerHub, estos han ido actualizándose cada vez que se realizaba un **merge** a la rama `main` (producción). Además, se ha configurado una serie de variables para cada servicio, de forma que puedan conectarse con las bases de datos y con el servicio de RabbitMQ. Estas variables son inyectadas automáticamente en el código a partir de un archivo `application-prod.properties` y un perfil activo `prod`.

En cuanto a la seguridad, tanto Railway como GitHub Pages proporcionan URLs para cada servicio con **HTTPS**, lo que garantiza que el tráfico entre el cliente y los servicios esté cifrado, evitando el envío en texto plano.

Los servicios desplegados en Railway han sido los siguientes:

- **Microservicio de Gestión de Usuarios:** <https://spring-cohousing-GestionUsuarios-production.up.railway.app>

- **Microservicio de Gestión de Comunidades:** <https://spring-cohousing-GestionComunidades-production.up.railway.app>
- **Microservicio Recomendador:** <https://python-cohousing-recomendador-production.up.railway.app>
- **Microservicio de Solicitudes:** <https://spring-cohousing-Solicitudes-production.up.railway.app>
- **Bases de datos:** Tres bases de datos PostgreSQL denominadas Postgres-Comunidad, Postgres-Usuarios y Postgres-Solicitudes
- **Microservicio ApiGateway:** <https://spring-cohousing-apigateaway-production.up.railway.app>
- **Servicio RabbitMQ:** Desplegado a partir de una plantilla obtenida en DockerHub.

Capítulo 10

Conclusiones

En este último capítulo se presenta un análisis de las conclusiones del proyecto realizado. Para ello, se realizará un balance general de los objetivos planteados y de su completitud durante el transcurso del proyecto. Además, se incluirá una reflexión sobre el impacto que este trabajo ha tenido en mi formación y sobre cómo las habilidades obtenidas se podrán aplicar en el desarrollo de una carrera como Ingeniero de Software. Finalmente, se comentarán posibles trabajos a futuro orientados a mejorar la aplicación desarrollada.

10.1. Balance general

Tras los meses de trabajo en el desarrollo del proyecto, **ShareSpace**, se ha logrado estructurar un sistema completo de **microservicios**, garantizando tanto la resiliencia frente a errores como la escalabilidad. Además, se ha diseñado un **frontend** que permite al usuario interactuar con la aplicación de manera intuitiva. Todo el desarrollo ha sido complementado con un **flujo completo de CI/CD** mediante **GitHub Actions**. Finalmente, se ha realizado un despliegue real utilizando las plataformas **GitHub Pages** y **Railway** [28]. Todo ello ha permitido crear una solución al problema inicial planteado, asegurando la calidad y aplicando técnicas de desarrollo de software actuales.

10.1.1. Objetivos conseguidos

En primer lugar, se logró establecer una metodología de trabajo basada en Scrum, adaptada al carácter individual de un TFG. Se identificaron las tareas a implementar en la aplicación, se crearon historias de usuario a partir de ellas, se agruparon en iteraciones y se organizaron en un diagrama de Gantt. Todo ello quedó reflejado en **GitHub**, garantizando que el desarrollo del software estuviese estrechamente ligado a la planificación del proyecto.

Se ha conseguido integrar **GitHub** de forma central durante todo el desarrollo. No solo se reflejaron las historias de usuario junto a las iteraciones a las que pertenecen en GitHub Project, sino que las tres ramas principales del repositorio han guiado el flujo de desarrollo de la funcionalidad. Además, las **GitHub Actions**, como herramienta de **CI/CD**, han contribuido a mejorar la calidad y fiabilidad del código, siendo parte esencial del flujo completo del desarrollo.

Tecnológicamente, se ha logrado profundizar en herramientas actuales presentes en entornos profesionales, como **Docker** y **Docker Compose** para la contenedorización, **Spring Boot** para el backend y **React** con **TypeScript** para el frontend. Asimismo, se ha comprendido el diseño completo de una arquitectura compleja basada en microservicios, identificando responsabilidades de cada servicio, implementando **comunicación asíncrona** entre microservicios mediante **RabbitMQ**, y desarrollando componentes esenciales como el **API Gateway**.

A nivel de desarrollo, se ha conseguido asegurar la calidad del código, profundizando en el uso de patrones de diseño y arquitecturas limpias.

En cuanto a la funcionalidad, se ha logrado ofrecer al usuario una plataforma que combine tanto la **búsqueda de comunidades** como su propia gestión. El sistema es capaz de recomendar comunidades al usuario en función del grado de afinidad, utilizando un **algoritmo de recomendación** basado en clustering mediante k-means.

Las solicitudes son enviadas al administrador, que será el encargado de aceptar o rechazar la petición de unión.

Una vez unido a una comunidad, el usuario dispone de una **gestión completa de tareas y eventos**, pudiendo crear, modificar y organizar tanto tareas como eventos. Esta gestión se facilita mediante un **calendario interactivo**, y al final de cada semana se genera un resumen semanal, el cual se envía por correo a cada usuario.

De esta forma, se han conseguido cubrir los dos grandes bloques de necesidad detectados al inicio: la búsqueda de comunidades y su gestión completa.

Finalmente, se ha podido realizar un despliegue completo del sistema utilizando **Railway**, una herramienta que ha permitido desplegar el Dockerfile de cada microservicio subido a **Docker Hub**, así como el servicio de **RabbitMQ** y las bases de datos. Con esto se ha concluido el desarrollo completo del proyecto, desde la creación de las bases hasta su despliegue en la nube.

10.2. Valoración final

Tras la realización de este proyecto, se ha proporcionado una solución al problema identificado, creando una propuesta final que ofrece una experiencia completa al usuario.

Desde el punto de vista académico, este trabajo ha supuesto una oportunidad para aplicar de forma práctica los conocimientos adquiridos a lo largo del grado, especialmente en asignaturas vinculadas a la mención de Ingeniería del

Software. Asignaturas como *Metodología de Desarrollo Ágil* (MDA) o *Desarrollo y Gestión de Proyectos* (DGP) han resultado fundamentales para planificar y organizar el proyecto mediante la metodología ágil SCRUM, permitiendo estructurar adecuadamente las tareas, la planificación temporal y la entrega iterativa de versiones.

Desde una perspectiva personal, me siento orgulloso del resultado obtenido. Este proyecto nació de una necesidad real que me afectaba personalmente y ha sido desarrollado desde cero con una visión profesional. No se ha tratado únicamente de programar, sino de diseñar y tomar decisiones técnicas justificadas, poniendo al usuario en el centro del proceso. Cada aspecto del desarrollo ha seguido principios sólidos de ingeniería del software, incorporando herramientas y prácticas comunes en el mundo profesional, como el uso de contenedores, GitHub, CI/CD y despliegue automatizado.

Considero que todo lo aprendido y aplicado en este proyecto se puede trasladar al ámbito profesional, donde es imprescindible entender cómo estructurar la lógica de negocio de una propuesta y cómo aplicar una arquitectura en consecuencia. Al analizar ofertas de trabajo en el sector, se puede observar que muchas de las competencias demandadas por las empresas han sido puestas en práctica a lo largo de este trabajo.

10.3. Trabajos Futuros

Con el objetivo de ampliar y mejorar el proyecto desarrollado, se han identificado y analizado una serie de mejoras que podrían integrarse en la aplicación para optimizar la experiencia del usuario. A continuación, se exponen las más representativas:

- **Integrar un sistema de chat:** Permitiría a los usuarios comunicarse directamente con los administradores antes de solicitar la unión a una comunidad, resolviendo dudas previas y acercando a los usuarios buscadores a los ofertantes.
- **Desarrollar una versión móvil:** Ofrecer una versión adaptada para dispositivos móviles. Usando tecnologías como React Native, se podría adaptar gran parte del código actual del frontend a un formato móvil.
- **Automatizar el despliegue móvil:** Implementar una GitHub Action que automatice el despliegue de la versión móvil. Una vez desarrollado, se podría incluir en el CI/CD para que la aplicación evolucione a medida que se desarrolla funcionalidad.
- **Solicitudes en tiempo real:** Crear un sistema de notificaciones basado en WebSocket para permitir la actualización en tiempo real. Este sistema permitirá que el usuario reciba al instante las actualizaciones sobre sus solicitudes pendientes, mostrando además un contador que refleje en todo momento el número de solicitudes.
- **Pruebas de rendimiento:** Realizar pruebas de rendimiento para evaluar

cómo responde cada microservicio bajo distintas cargas en escenarios reales de uso, comprobando la eficiencia y respuesta del sistema.

- **Refuerzo de la seguridad:** Mejorar la seguridad en el almacenamiento de datos para proteger la información de los usuarios.
- **Publicar en plataformas oficiales:** Desplegar la aplicación móvil en plataformas oficiales, como App Store, permitiendo su acceso a un mayor volumen de usuarios.

Las mejoras propuestas se centran principalmente en ofrecer una solución móvil que permita a los usuarios disfrutar de una experiencia de calidad. Para ello, será necesario adaptar el código a React Native o alguna plataforma similar. Además, la implementación de WebSocket para las solicitudes supondría una mejora significativa en la interacción y usabilidad para el usuario. Personalmente, me gustaría continuar con las mejoras de la aplicación, ya que su implementación podría suponer un gran aprendizaje.

Bibliografía

- [1] Atlassian. ¿qué es un backlog de producto en scrum?, s.f.
- [2] Codecov. Codecov Docs. <https://docs.codecov.com/docs>, 2023. Último acceso: julio 2025.
- [3] DatosMacro. Zona euro. 2025.
- [4] DevJobsScanner. The most demanded frontend frameworks. *DevJobsScanner Blog*, 2024.
- [5] DevJobsScanner. The most demanded frontend frameworks. *DevJobsScanner*, 2024.
- [6] Docker Inc. Docker compose documentation. <https://docs.docker.com/compose/>, 2024. Último acceso: julio de 2025.
- [7] Docker Inc. Docker documentation. <https://docs.docker.com/>, 2024. Último acceso: julio de 2025.
- [8] Inc. Facebook. Jest: Delightful javascript testing framework, 2025.
- [9] Flatify. Aplicación para gestión de viviendas compartidas. <https://www.flatify.com>.
- [10] FlatmateFinders. Aplicación para búsqueda de pisos por afinidad. <http://www.flatmatefinders.com.au>.
- [11] FullCalendar. Fullcalendar documentation. <https://fullcalendar.io/docs>, 2025. Accedido el 13 de julio de 2025.
- [12] GitHub. GitHub Docs: GitHub Actions. <https://docs.github.com/en/actions>, 2023. Último acceso: julio 2025.
- [13] GitHub. Octoverse 2024: Ai leads python to top language as the number of global developers surges. *GitHub Blog*, 2024.

- [14] Inc. GitHub. Github: A platform for version control and collaboration, 2025.
- [15] Scrum Guide. The scrum guide, 2020.
- [16] IBM. Propiedades acid en el procesamiento de transacciones. 2025.
- [17] Idealista. Portal inmobiliario para la búsqueda de pisos y habitaciones. <https://www.idealista.com>, 2022.
- [18] JWT.io. Introduction to json web tokens. <https://jwt.io/introduction>, 2024. Último acceso: julio de 2025.
- [19] Lucidchart. Tutorial de diagrama de clases uml, 2024.
- [20] Robert C. Martin. The clean architecture, 2012. Último acceso: 7 de mayo de 2025.
- [21] Julia Martins. Scrum: conceptos clave y cómo se aplica en la gestión de proyectos. *asana*, 2025.
- [22] Meta. React: A javascript library for building user interfaces, 2025.
- [23] Microsoft. Visual Studio Code. <https://code.visualstudio.com/>, 2023. Último acceso: julio 2025.
- [24] Node.js. Node.js v16.x.x documentation, s. f.
- [25] Jan Ondrášek. Software architecture and how it is related to software quality. <https://speakerdeck.com/ondrasek/pv260-software-architecture-and-how-it-is-related-to-software-quality>, 2019. Accedido: 27 de agosto de 2025.
- [26] The pandas development team. pandas-dev/pandas: Pandas. <https://pandas.pydata.org/>, 2023. Accedido el 13 de julio de 2025.
- [27] RabbitMQ. Rabbitmq, 2025.
- [28] Railway. Railway: Deploy apps with simplicity, 2025.
- [29] Sebastián Ramírez. Fastapi. <https://fastapi.tiangolo.com/>, 2023. Accedido el 13 de julio de 2025.
- [30] Red Hat. ¿qué es ci/cd? <https://www.redhat.com/es/topics/devops/what-is-ci-cd>, 2023. Último acceso: julio 2025.
- [31] Chris Richardson. Microservices pattern: Microservice architecture.
- [32] Pivotal Software. Spring boot: A framework for building java applications, 2025.

Bibliografía

- [33] Spring Team. Spring security with jwt. <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html>, 2024. Último acceso: julio de 2025.
- [34] Maria José Vañó Vañó. Cohousing cooperativo: Un análisis de retos y oportunidades. 2022.
- [35] Eda Kavlakoglu y Vanna Winland. ¿qué es la agrupación en clústeres k-means? *IBM*.