



Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

**Desarrollo basado en microservicios de una
aplicación web para la gestión de espacios
compartidos de comunidades vecinales.**

Autor:
Sergio Motrel Bajo



Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería de Software

**Desarrollo basado en microservicios de una
aplicación web para la gestión de espacios
compartidos de comunidades vecinales.**

Autor:
Sergio Motrel Bajo

Tutora:
Irene Lavín Perrino

No hay caminos cortos hacia lugares que valen la pena.

Agradecimientos

Le dedico este TFG a mis padres, cuyo apoyo incondicional y confianza en mí han sido fundamentales durante todos estos años. A mi hermano, cuyo ejemplo me han inspirado y motivado a seguir adelante.

A mis amigos Sergio, Gonzalo y Miguel Ángel por su amistad y por estar siempre a mi lado incluso en los momentos más difíciles.

A mis compañeros Carlos y Pablo, con quienes he compartido innumerables buenos momentos durante la carrera y la administración de otros proyectos, haciendo de esta experiencia algo inolvidable.

Y a mi tutora Irene, quien ha tenido la paciencia y disposición para guiarme a lo largo de este proceso.

Gracias a todos.

Resumen

El resultado de este Trabajo de Fin de Grado es el desarrollo de una aplicación web de gestión de espacios y recursos compartidos de comunidades de vecinos, así como la gestión de los usuarios. El desarrollo se ha realizado basado en microservicios y las tecnologías utilizadas han sido Angular para el Front-end y Spring, NestJs y Slim Framework para el Back-end.

La aplicación web es totalmente responsive y esta pensada para su acceso en cualquier dispositivo. Se han aplicado buenas prácticas y diseño de interfaces facilitando su uso para usuarios no familiarizados con las nuevas tecnologías.

Abstract

The result of this Final Degree Project is the development of a web application for the management of shared spaces and resources of neighborhood communities, as well as the management of users. The development has been based on microservices and the technology stack used is Angular for the Front-end and Spring, NestJs and Slim Framework for the Back-end microservices.

The web application is fully responsive and is designed to be access on any device. The use of good practices and interface design has made the application accessible to users who may not be familiar with new technologies.

Índice general

Agradecimientos	III
Resumen	V
Abstract	VII
Lista de figuras	XVII
Lista de tablas	XX
1. Introducción	1
1.1. Contexto	1
1.2. Motivación	2
1.3. Estado del arte	2
1.4. Objetivo	3
1.4.1. Objetivos de desarrollo	3
1.5. Objetivos personales	4
1.5.1. Estructura de la memoria	4
2. Metodología, Planificación y Análisis de riesgos	6
2.1. Metodología	6
2.1.1. Agile	6

2.1.2. Extreme programming	7
2.1.3. Roles en el proyecto	7
2.2. Planificación	9
2.3. Análisis de riesgos	9
2.3.1. Análisis de riesgos de un TFG	10
2.3.2. Análisis de riesgos de la aplicación	13
2.3.3. Análisis de riesgos para la protección de datos.	14
2.4. Análisis de costes	17
2.5. Iteraciones del desarrollo	18
3. Tecnologías	19
3.1. Tecnologías	19
3.1.1. Angular	19
3.1.2. Gestores de paquetes y dependencias	20
3.1.3. Java	21
3.1.4. Spring Boot	21
3.1.5. Maven	22
3.1.6. Node.JS	22
3.1.7. Npm	23
3.1.8. NestJs	23
3.1.9. PHP	24
3.1.10. Composer	25
3.1.11. Slim Framework	26
3.1.12. Tokens JWT	26
3.1.13. Docker	28
3.1.14. Git y Github	28
3.1.15. MySQL	29

3.1.16. PostgreSQL	30
3.1.17. MongoDB y NoSQL	31
3.1.18. Tailwind CSS	31
3.2. Herramientas	32
3.2.1. Visual Studio Code	32
3.2.2. Trello	33
3.2.3. Draw.io	34
3.2.4. Microsoft Teams	35
3.2.5. Overleaf y TexStudio	35
3.2.6. GitHub Copilot	35
3.3. Herramientas menores	36
3.3.1. Thunder Client	36
3.3.2. JWT.io	37
4. Análisis	38
4.1. Requisitos	38
4.2. Requisitos funcionales	38
4.3. Requisitos no funcionales	40
4.4. Casos de uso	40
4.5. Descripciones de casos de uso	42
4.5.1. Identificarse	42
4.5.2. Editar perfil	42
4.5.3. Crear Usuario	43
4.5.4. Gestión Reservas	43
4.5.5. Gestión Incidencias	44
4.5.6. Ver Comunidades	44
4.5.7. Ver Espacios	45

4.5.8. Crear Comunidad	45
4.5.9. Borrar Comunidad	46
4.5.10. Editar Comunidad	46
4.5.11. Crear Espacio	47
4.5.12. Borrar Espacio	47
4.5.13. Editar Espacio	48
4.6. Diagrama inicial de clases	49
5. Diseño	51
5.1. Microservicios o Monolito	51
5.1.1. Monolito	51
5.1.2. Microservicios	53
5.1.3. El por qué de los microservicios	56
5.2. Arquitectura y estructura del proyecto	57
5.2.1. Modelo Vista Controlador	57
5.2.2. DDD	59
5.2.3. Arquitecturas por capas	61
5.2.3.1. Arquitectura limpia	61
5.2.4. Arquitectura hexagonal	62
5.2.5. Comparación y Elección de la Arquitectura	63
5.3. Patrones de diseño utilizados	63
5.3.1. DTO	63
5.3.2. Data Mapper	64
5.3.3. Repositorio	65
5.3.4. Singleton	66
5.4. Separación en microservicios	67
5.5. Diagrama de despliegue	70
5.6. Funcionamiento de las APIs	71

6. Implementación	73
6.1. Trabajo previo	73
6.2. Implementaciones en común	73
6.2.1. Orquestación de cookies	74
6.2.2. Validación de peticiones	76
6.2.3. Control de errores y respuesta de las APIs	76
6.2.4. Estructura de carpetas	77
6.2.5. Middleware JWT	78
6.3. API de usuarios	79
6.3.1. Estructura básica	79
6.3.2. Endpoints	81
6.4. API de comunidades y espacios	82
6.4.1. Estructura básica	82
6.4.2. Servicio de almacenamiento	83
6.4.3. Spring Security	84
6.4.4. Endpoints	86
6.5. API de reservas	86
6.5.1. Estructura básica	86
6.5.2. Comunicación con API de comunidades	88
6.6. Frontend con Angular	89
6.6.1. Estructura básica	89
6.6.2. Servicio de autenticación	91
6.6.3. Formularios	92
6.6.4. Maquetado	92
6.7. Experiencia de usuario	93
6.7.1. Enlaces y botones	93

6.7.2. Accesibilidad	94
6.7.3. Core Web Vitals	94
6.7.4. Herramientas	94
6.7.5. Imágenes	97
6.7.6. Tiempo de bloqueo	98
7. Despliegue	100
7.1. Configuración de dominio	100
7.2. Configuración de servidores	101
8. Resumen de la realización del proyecto	104
8.1. Problemas	104
8.2. Riesgos materializados	106
9. Conclusiones y trabajo futuro	107
9.1. Conclusiones	107
9.2. Trabajo futuro	108
9.2.1. SEO	108
9.2.2. Patrón Criteria	109
9.2.3. Implementación de incidencias	110
9.2.4. Añadido de API Gateway	110
9.2.5. Renderizado en el lado del servidor(SSR)	110
9.2.6. Integración y despliegue continuos	111
A. Manual de arranque en local	112
A.1. Requisitos	112
A.2. Clonado del repositorio	112
A.3. Despliegue de contenedores	113
A.4. Ejecución de Angular	113

B. Manual de despliegue en producción	114
B.1. Requisitos	114
B.2. Configuración previa de los servidores	114
B.3. Código de configuración de Nginx	117
B.4. Despliegue de contenedores	119
C. Manual de usuario	120
C.1. Registro e Identificación	120
C.1.1. Registro	120
C.1.2. Identificación	122
C.2. Estructura de la web	122
C.3. Vecino	123
C.3.1. Unirse a comunidades	123
C.3.2. Realizar una reserva	124
C.3.3. Listado de reservas	126
C.4. Administrador	126
C.5. Superusuario	130
D. Manual de endpoints de las APIs	132
D.1. API de usuarios	132
D.1.0.1. Registro	132
D.1.0.2. Login	132
D.1.0.3. Comprobación de token JWT	133
D.1.0.4. Petición de todos los usuarios	133
D.1.0.5. Información de un usuario con id	134
D.1.0.6. Información de un usuario con token	134
D.1.0.7. Modificación de usuario	134

D.1.0.8. Borrado de usuario	135
D.2. API de comunidades	135
D.2.0.1. Añadir comunidad	135
D.2.0.2. Petición de comunidad con su id	135
D.2.0.3. Petición de todas las comunidades	136
D.2.0.4. Petición de todas las comunidades de un usuario	136
D.2.0.5. Actualizar una comunidad	136
D.2.0.6. Unirse a una comunidad con código de invitación	136
D.2.0.7. Borrar comunidad	137
D.2.0.8. Añadir espacio	137
D.2.0.9. Petición de espacio con su id	137
D.2.0.10. Petición de todos los espacios	137
D.2.0.11. Petición de todos los espacios de una comunidad	138
D.2.0.12. Actualizar un espacio	138
D.2.0.13. Borrar espacio	138
D.3. API de reservas	138

Índice de figuras

3.1. Uso de php a lo largo del tiempo	25
3.2. Payload de un JWT	27
3.3. Encuesta de uso de bases de datos StackOverflow 2022	29
3.4. Esquema de una vista materializada	30
3.5. Comparación entre Mysql y MongoDB	31
3.6. Interés por años (Frameworks css)	32
3.7. Tablero kanban de Trello	34
3.8. Sugerencia de código de GitHub copilot	36
4.1. Diagrama de Casos de uso	41
4.2. Diagrama de Clases iniciales	49
5.1. Gastos en modernización desde 2010	53
5.2. Ejemplo de microservicios de una tienda.	54
5.3. Arquitectura y tecnologías de Netflix.	56
5.4. Esquema básico de MVC	58
5.5. Ejemplo de agregado de SCRUM	60
5.6. Aplicación con dos <i>bounded context</i>	61
5.7. Esquema de la arquitectura limpia	62
5.8. Esquema de la arquitectura hexagonal	63

5.9. Esquema de los patrones DTO y Mapper	65
5.10. Funcionamiento general de Repositorio y Criteria.	66
5.11. Diagrama de clases por microservicios	68
5.13. Esquema de flujo de ejecución API	71
5.12. Diagrama despliegue y comunicación entre servicios.	72
6.1. Esquema de un ataque XSS	75
6.2. Esquema de manejo de excepciones en arquitectura por capas.	77
6.3. Estructura de carpetas de las APIs	78
6.4. User Entity	80
6.5. Inicialización de Mongo con variables de entorno	81
6.6. Añadido de cookie a la respuesta con variables de entorno.	82
6.7. Interfaz StorageService.	84
6.8. Cadena de filtros de SpringWebSecurity.	85
6.9. Implementación de repositorio con JpaRepository.	86
6.10. Parte del contenedor de dependencias de Slim.	87
6.11. Parte del contenedor de dependencias de Slim.	88
6.12. Servicio de comunicación con la API de comunidades.	89
6.13. Plantilla básica de la web.	90
6.14. Estructura de carpetas básico en el frontend.	90
6.15. Directiva de control por roles.	91
6.16. Ruta protegida por un guard de rol administrativo.	91
6.17. Ejemplo de validación de formularios en Angular.	92
6.18. Puntuación lighthouse de una página de la web.	96
6.19. Puntuación baja debido al 3g.	97
7.1. Dominios registrados en Cloudflare.	100

7.2. Redirecciones DNS en Cloudflare	101
B.1. Archivo de configuracion Nginx, Servidor 1	117
B.2. Archivo de configuracion Nginx, Servidor 2	118
C.1. Formulario de identificación.	121
C.2. Formulario de registro.	121
C.3. Estructura general de la web.	122
C.4. Campo de unión a una comunidad con código de invitación	123
C.5. Comunidad a la que se ha unido con el código de invitación.	124
C.6. Lista de espacios disponibles para reservar.	125
C.7. Página para reservar un espacio.	126
C.8. Listado de reservas.	127
C.9. Menú lateral de un usuario Administrador.	127
C.10.Menú de selección de administrador.	128
C.11.Listado de administración de comunidades.	128
C.12.Formulario de edición de comunidades.	129
C.13.Formulario de adición de comunidades.	130
D.1. Añadido de cookie a la respuesta con variables de entorno.	133

Índice de tablas

2.1. Tabla de planificación por semanas.	9
2.2. Tabla de probabilidad e impacto	10
2.3. Riesgo 3 - Ser muy optimista	11
2.4. Riesgo 2 - Mala estimación del tiempo.	11
2.5. Riesgo 3 - No poder presentar el TFG.	11
2.6. Riesgo 4 - Abandono del proyecto.	12
2.7. Riesgo 5 - Problemas de escalado	13
2.8. Riesgo 6 - Dificultades de uso de la aplicación	13
2.9. Riesgo 7 - Mala seguridad	14
2.10. Riesgo 8 - Robo de datos interno	14
2.11. Riesgo 9 - Ataque a la base de datos	15
2.12. Riesgo 10 - Phishing	15
2.13. Riesgo 11 - Mal uso de datos	16
2.14. Previsión de costes para el desarrollo del proyecto	17
4.1. Tabla de requisitos funcionales	40
4.2. Tabla de requisitos no funcionales	40
4.3. Descripción de caso de uso - Identificarse	42
4.4. Descripción de caso de uso - Editar perfil	42
4.5. Descripción de caso de uso - Crear Usuario	43

4.6. Descripción de caso de uso - Gestión Reservas	43
4.7. Descripción de caso de uso - Gestión Incidencias	44
4.8. Descripción de caso de uso - Ver Comunidades	44
4.9. Descripción de caso de uso - Ver Espacios	45
4.10. Descripción de caso de uso - Crear Comunidad	45
4.11. Descripción de caso de uso - Borrar Comunidad	46
4.12. Descripción de caso de uso - Editar Comunidad	46
4.13. Descripción de caso de uso - Crear Espacio	47
4.14. Descripción de caso de uso - Borrar Espacio	47
4.15. Descripción de caso de uso - Editar Espacio	48

Capítulo 1

Introducción

1.1. Contexto

Hoy en día, muchas gestiones toman mas tiempo del necesario debido a trámites innecesarios y tiempos de espera elevados. La aplicación de este TFG puede proporcionar una solución eficiente para la reserva de pistas deportivas y otros espacios.

Puede reemplazar sistemas arcaicos que hacen perder el tiempo tanto a los administradores como a los propios vecinos. Con esta aplicación, los usuarios podrán ver en tiempo real la disponibilidad de los espacios compartidos y reservarlas desde la comodidad de un click. Se acabaría la desorganización, los conflictos por malentendidos, y requeriría mucho menos esfuerzo de administrar.

También, esta aplicación facilitaría la comunicación entre los vecinos y la gestión, haciendo que los usuarios puedan comunicar directamente cualquier tipo de problema o desperfecto. Esto mejora tanto el tiempo de reacción como la calidad del servicio.

La finalidad de todo esto es ofrecer un servicio que en general traería comodidad para ambas partes, teniendo en mente una aplicación muy sencilla de utilizar y que prácticamente se autogestione. Esto haría también que al organizar los horarios mejor, los espacios compartidos se utilizarían de forma más eficiente.

Además, este proyecto se podría seguir escalando, ya que la idea es basarlo en microservicios, haciendo que ampliarlo para cubrir otros aspectos de las comunidades vecinales fuese tarea sencilla en un futuro.

El proyecto es parte del Trabajo de Fin del Grado [1] de Ingeniería Informática por la Universidad de Valladolid. Se desarrollará una aplicación web para la mención de Ingeniería de Software.

1.2. Motivación

Desde hace tiempo tengo bastante claro que me quiero especializar en desarrollo web, especialmente en tecnologías como Angular o React. Con este proyecto, tengo la oportunidad perfecta para probarme a mi mismo y superar las dificultades que me encuentre en el camino.

La motivación principal es comprobar que todo lo aprendido ha dado sus frutos y profundizar un poco mas en un desarrollo en solitario, forzándome así a hacer todas las partes del desarrollo, tanto front-end como back-end.

Como creo que tanto Angular como React son grandes tecnologías con gran cuota de mercado [2], cualquiera de las dos hubiese sido una elección perfecta. Finalmente, me decidí por lo ya visto en la asignatura Desarrollo basado en Componentes y Servicios para poder seguir profundizando y afianzando conocimientos.

Además, me he dado cuenta de que hay un nicho de mercado que claramente no está cubierto, ya que en España no hay nada parecido que esté asentado en la sociedad. El hecho de lo complicado que supone muchas veces organizar una comunidad de vecinos grande hace que una herramienta de este estilo sea algo que tenga cabida.

Por último, quería que el desarrollo se centrara en los microservicios porque creo que es una arquitectura muy fundamental en la actualidad ya que en muchos proyectos grandes y aplicaciones a nivel empresarial es muy utilizado y es para mi una de las mejores maneras para prepararse para una futuro escalado de los proyectos.

1.3. Estado del arte

Actualmente existen varias aplicaciones y webs similares a lo que propongo en este proyecto, aunque presentan diversas limitaciones y desafíos que justifican la necesidad de una nueva solución.

PropApp[3] es una de las aplicaciones similares, pero es bastante compleja y cuenta con una interfaz recargada, lo que puede hacer que no se fidelice correctamente a usuarios de edad avanzada. Centrarse solo en usuarios jóvenes puede ser un error, ya que al fin y al cabo el ciudadano promedio propietario de una vivienda es de mediana o avanzada edad, estando la media en 41 años de edad [4].

Otra aplicación similar es **TusVecinos**, que está orientada a ser más informativa. Aunque tiene algunas funciones interesantes y similares, el nicho está poco aprovechado, ya que es raro encontrar una comunidad vecinal que utilice una aplicación de este estilo. TusVecinos ofrece funcionalidades como la reserva de espacios comunes, gestión documental, y notificaciones en tiempo real, lo que la hace bastante completa para la gestión comunitaria. Sin embargo, su enfoque informativo y la falta de adopción generalizada limitan su impacto en el mercado.

Posiblemente la aplicación más trabajada y con más usuarios con un millón de descargas es **Neighbors**[5] de la empresa Ring, tiene muchas funcionalidades y tiene una buena interfaz,

moderna e intuitiva. Sin embargo, esta aplicación no tiene formato web, lo que puede limitar su expansión y, además, no está disponible en algunos países como España.

Por último existe también **FincApp**[6], que aunque parece la más completa y con más funciones, es de pago y tiene malas reseñas. La necesidad de pagar por el servicio y las críticas negativas afectan su popularidad.

En general, pienso que hay un hueco en el mercado que se puede satisfacer, y aunque hay algunas alternativas, creo que no están bien planteadas ni anunciadas, ya sea por falta de SEO o por malas valoraciones.

1.4. Objetivo

Debido a que hay objetivos que son puramente enfocados al ámbito académico y otros que son más del personal, se han dividido en dos secciones.

1.4.1. Objetivos de desarrollo

Este proyecto se centra en el desarrollo de una aplicación web para la gestión de espacios compartidos de comunidades vecinales. Esto significa poder hacer reservas de pistas de deportes, salas de reuniones, salas de fiestas, etc...

Claramente, una web de este tipo con los tiempos que corren debe ser responsiva, para que incluso se pudiese utilizar como aplicación con un **WebView** [7] o una **PWA** [8] (Progressive Web App). Para ello, se busca enfocarse en que todas las pantallas de la misma tengan tanto un formato web como móvil.

Se busca implementar una interfaz sencilla e intuitiva para que incluso los vecinos en edades avanzadas sean capaces de hacer reservas a través de la misma. No se ha utilizado nada muy dinámico ni complejo principalmente para mantener la facilidad de uso y la buena experiencia de usuario.

Además, otro de los objetivos es desarrollar todo esto en una arquitectura basada en microservicios, ya que hace un posible escalado de la aplicación más sencillo y habilita la posibilidad a poder reutilizar ciertas partes.

En general, el objetivo ha sido el de construir un buen producto mínimo, sencillo pero eficaz, mantenible, ajustado al tiempo del TFG y proponer mejoras y ampliaciones como trabajo futuro. Y para ello lo primero es construir un producto útil, y aunque la meta a largo plazo pudiera ser cubrir la necesidad de una aplicación de gestión total, lo más inteligente es empezar por una aplicación mínima útil y desde ahí ir construyendo. Por ello, empezar con una aplicación sencilla para la gestión de reservas es una buena manera de hacerlo.

1.5. Objetivos personales

Por otro lado, el proyecto busca la formación en algunos aspectos esenciales y muy interesantes del desarrollo web, más enfocado a adquirir una experiencia más global y enfocarse más en el aspecto backend, la arquitectura y la infraestructura de servidores en general.

Unido a esto, ampliar los conocimientos en Angular, maquetación y diseño frontend, ya que aunque las bases en el grado han sido buenas, aún queda mucho camino por recorrer.

También, se ha buscado utilizar diversas tecnologías para conocer los detalles de las mismas y tener un entendimiento más global de algunas herramientas que se utilizan en el sector, es por esto que se ha decidido utilizar tantos frameworks y lenguajes distintos.

Por último, la búsqueda de experiencia en como configurar, preparar y desplegar en múltiples servidores y coordinar la comunicación entre ellos desde cero, ya que es un conocimiento que no ocupa lugar y que puede venir bien en múltiples escenarios.

1.5.1. Estructura de la memoria

Capítulo 2 Metodología, Planificación y Análisis de riesgos: Explicación de la metodología de trabajo llevada a cabo por el alumno en este TFG, la planificación que se ha seguido y análisis de los riesgos en los que se podría incurrir para así tener un plan para la prevención y actuación.

Capítulo 3 Tecnologías: Especificación y explicación de las tecnologías que se han empleado en este proyecto y por qué se han escogido.

Capítulo 4 Análisis: Estructura general preliminar de la aplicación, las clases que lo forman, el comportamiento de la aplicación en base a sus requisitos y una descripción detallada de los casos de uso.

Capítulo 5 Diseño: Descripción más detallada de la estructura de la aplicación, separación en microservicios y de la arquitectura del mismo.

Capítulo 6 Diseño: Descripción más detallada de la estructura de la aplicación, separación en microservicios y de la arquitectura del mismo.

Capítulo 6 Implementación: Detalles sobre la implementación de ciertas partes de las APIs, la web y la interacción entre ambas partes.

Capítulo 7 Despliegue: Se detalla el proceso para llevar la aplicación a un entorno de producción en servidores reales.

Capítulo 8 Resumen de la realización del proyecto: Descripción de algunos problemas surgidos durante el proyecto y riesgos que se han materializado.

Capítulo 9 Conclusiones: Conclusiones extraídas de la realización de este proyecto y de la forma de trabajo.

Apéndice A - Manual de arranque en local: Una guía de como arrancar el proyecto en local para el desarrollo.

Apéndice B - Manual de despliegue en producción: Una guía con los detalles para hacer un despliegue del proyecto en producción.

Apéndice C - Manual de usuario: Una guía de como utilizar la aplicación para los distintos tipos de usuario.

Apéndice D - Manual de endpoints de las APIs: Una guía de las opciones que ofrece cada una de las APIs.

Capítulo 2

Metodología, Planificación y Análisis de riesgos

2.1. Metodología

2.1.1. Agile

La metodología **Agile** de gestión de proyectos prima la adaptabilidad sobre la planificación. Se basa en hacer iteraciones cortas de desarrollo para así poder entregar regularmente pequeñas piezas del desarrollo al usuario.

Agile busca equipos pequeños y multidisciplinares, haciendo así que la auto-organización y la comunicación sean fluidas.

La metodología se apoya en el manifiesto Agile [9], un documento creado en colaboración de 17 gurús programadores. El manifiesto define cuatro valores y doce principios sobre los cuales se construye la idea de Agile. Como micro-resumen, estos son algunos de sus puntos:

1. Individuos e interacciones sobre procesos y herramientas.
2. Software funcionando sobre documentación exhaustiva.
3. Colaboración con el cliente sobre negociación de contratos.
4. Respuesta ante el cambio sobre seguir un plan.

Aunque en el caso de este TFG no se puedan aplicar todos los beneficios de Agile, como tener un equipo pequeño y la interacción con el cliente, se ha aplicado la mentalidad y la forma de trabajar de la misma. El enfoque principal ha sido la adaptación al cambio y las cortas iteraciones de desarrollo.

2.1.2. Extreme programming

De entre los marcos de trabajo de la metodología Ágil el más popular sin duda alguna es Scrum. Además de ser una de las formas de trabajo más asentadas en la industria, de lo visto en la asignatura Planificación y Gestión de Proyectos. Pero por el trabajo y el poco tiempo libre del que se ha dispuesto, se adecuaba más otra metodología Ágil bastante menos popular y conocida que es **Extreme Programming** (o **XP**).

XP tiene como objetivo mejorar la calidad del producto y ser muy flexible a cambios y a los problemas que se presentan, así como de los cambios requeridos por el cliente. Se basa en la simplicidad y retroalimentación constante para mejorar la velocidad y flexibilidad a través de el desarrollo iterativo. Algunos puntos de extreme programming:

- **Desarrollo Iterativo:** El trabajo se divide en pequeñas tareas para introducción de *features* y implementaciones más rápidas.
- **Peer programming:** La programación en pareja es la revisión, trabajo y implementación en parejas de programadores para un mayor aprendizaje y mentorización.
- **Refactorización:** Se busca una mejora constante del código y del producto a través de re-escrituras del mismo sin afectar al funcionamiento, con esto se consigue un mayor estándar de calidad.
- **Integración Continua:** Compilado y prueba continua del código para la búsqueda temprana de fallos y bugs para que no lleguen a producción.
- **Simplicidad:** Prima lo simple del código para un mayor entendimiento y evitar problemas.

Obviamente algunos puntos como el peer programming no se han podido llevar a cabo pero el resto se ha intentado dentro de lo posible y de el tiempo de adaptación y al horario.

Si hay que destacar algo, es que no ha supuesto ningún problema rehacer partes del código que debido a la experiencia ganada se ha visto que estaban mal implementadas, como por ejemplo rehacer casi dos APIs al completo por estar siguiendo una mala estructura y arquitectura.

2.1.3. Roles en el proyecto

Durante el desarrollo del proyecto, se han asumido tres roles fundamentales que se alinean con un entorno laboral real: Dueño del Producto, Desarrollador Full-Stack y Administrador de Sistemas. A continuación, se detallan las responsabilidades y justificaciones de cada uno de estos roles.

Como **Dueño del Producto** (Product Owner), se han definido y gestionado todos los requisitos y funcionalidades del proyecto. Este rol es crucial en cualquier entorno de desarrollo

ágil, ya que implica tomar decisiones estratégicas sobre el producto, definir las funcionalidades que debe cumplir y garantizar que el equipo de desarrollo se enfoque en entregar el máximo valor.

En este proyecto, se han tomado todas las decisiones clave sobre el diseño, la funcionalidad y la dirección general del producto, asegurando que cumpla con las expectativas y las futuras necesidades del usuario. La responsabilidad del product owner también incluye realizar reuniones con el equipo de desarrollo, pero como en este caso el equipo de desarrollo era el mismo product owner, nos se ha podido realizar.

El rol de **Desarrollador Full-Stack** ha permitido trabajar tanto en el frontend como en el backend. En el frontend, se han utilizado tecnologías como HTML, CSS y TypeScript para crear interfaces de usuario intuitivas y responsivas.

En el backend, se ha trabajado con lenguajes de programación y frameworks para construir APIs robustas y escalables y seguir la lógica del negocio que había pensado como product owner.

Este rol requiere una comprensión profunda de todo el stack tecnológico utilizado en el proyecto, así como la capacidad de solucionar problemas y optimizar el rendimiento en todas las capas de la aplicación. Además, en un proyecto con varias tecnologías como es este, es todavía más importante. La capacidad de trabajar en todo el stack ha sido esencial para garantizar que todo funcione.

Finalmente, se ha aplicado parcialmente el rol de **Administrador de Sistemas** para el despliegue y mantenimiento de la infraestructura del proyecto. Esto ha incluido la contratación y configuración de servidores, la gestión de bases de datos y la implementación de medidas de seguridad para proteger los datos y la aplicación.

Se han tenido que configurar entornos de desarrollo, pruebas y producción, y asegurarse de que los servidores estén correctamente configurados para manejar el tráfico y las solicitudes de los usuarios, así como de configurar DNSs y dominios. Sin tener los conocimientos básicos de este rol, habría ido imposible llevar a producción el proyecto y se habría quedado como algo local.

2.2. Planificación

El tiempo objetivo a destinar al Trabajo de Fin de Grado es de 300 horas que corresponden a los 12 ECTs. La intención inicial contemplaba aproximadamente unas 45 horas semanales. El inicio de este TFG se hace a principios de Junio, pero debido a exámenes se empleará menos horas al principio y más durante el verano.

Semana	Horas a emplear	Trabajo a realizar
5/5/2023	10 horas	Introducción memoria
12/5/2023	10 horas	Requisitos y Análisis horas
19/5/2023	10 horas	Diseño
19/5/2023	10 horas	Elección de Arquitectura
26/5/2023	(Exámenes)	(Exámenes)
3/6/2023	25 horas	Primera API
10/6/2023	25 horas	Primera API
17/6/2023	25 horas	Segunda API
24/6/2023	25 horas	Segunda API
31/6/2023	25 horas	Tercera API
7/7/2023	25 horas	Tercera API
14/7/2023	30 horas	Frontend
21/7/2023	30 horas	Frontend y Despliegue
28/7/2023	30 horas	Memoria
4/8/2023	20 horas	Memoria

Tabla 2.1: Tabla de planificación por semanas.

Observando la tabla 2.1, el objetivo es de acabar la primera semana de Septiembre y entregar ese mes, ya que se realizarán las prácticas en empresa a la vez que el TFG en verano y tengo la posibilidad de entrega después. Aún quedarían otras dos semanas de margen por si algo saliera mal.

Respecto a como se ha organizado el trabajo, creo que lo mejor es primero realizar el análisis y diseño de la aplicación previo a empezar la implementación del código, para así tener más claro como se organizará la arquitectura y los servicios.

En el siguiente apartado se hablará más en detalle al respecto, pero se podría llegar al caso de no tener suficiente tiempo tiempo debido a las prácticas en empresa, ya que puede que consuman más tiempo del esperado. Por ello, no se asignan más de tres o cuatro horas al día en las semanas con más carga de trabajo.

2.3. Análisis de riesgos

Se ha decidido hacer dos pequeños análisis de riesgos. Uno de ellos está centrado en los posibles problemas a la hora de realizar un TFG cualquiera, no necesariamente este, mientras

que el otro se centra en los problemas que podrían surgir en el desarrollo y futura distribución de la aplicación.

PROB/IMP	BAJA	MEDIA	ALTA
BAJO	BAJO	BAJO	MEDIO
MEDIO	BAJO	MEDIO	ALTO
ALTO	MEDIO	ALTO	ALTO

Tabla 2.2: Tabla de probabilidad e impacto

Se puede observar una relación de probabilidad e impacto relacionada con los riesgos mencionados en la tabla 2.2. Es una referencia esencial en la gestión de riesgos, siendo muy utilizada para evaluar y priorizar los riesgos en función de su probabilidad de ocurrencia y su impacto potencial. Esta matriz ayuda calcular el riesgo resultante de la combinación de la probabilidad del riesgo y el impacto del mismo y se utilizara como referencia a la hora de calcular las próximas tablas.

En la tabla, los riesgos están categorizados en tres niveles de probabilidad (Bajo, Medio, Alto) y tres niveles de impacto (Baja, Media, Alta), se ha coloreado cada celda de la tabla está para indicar la severidad del riesgo para que sea más visual. El nivel de riesgo alto se marca con rojo, el medio con amarillo y el bajo con verde.

Tanto esta tabla como la descripción de riesgos se realiza para hacer un mejor análisis de la toma de decisiones y asignar mejor los recursos, así como para estar preparados a futuros contratiempos y tener un plan de respuesta.

2.3.1. Análisis de riesgos de un TFG

Uno de los principales problemas en los desarrollos de TFG es no ser realista en cuanto al trabajo que se puede realizar en 300 horas. Esto lleva a una mala documentación en muchos casos o lo que es peor al proyecto sin terminar o con funcionalidad importante sin hacer.

Sin embargo, dado que la tutora avisó de este peligro, se recortaron funcionalidades de la idea original ya que el proyecto era muy grande y actualmente la probabilidad de que suceda es baja.

El tiempo también puede ser un factor clave, ya que aunque la estimación de las dimensiones del trabajo sea correcta, se puede caer en un mala organización del tiempo y acabarse alargando algunas partes del mismo, haciendo que no se llegue al objetivo en el tiempo previsto.

Por último, estos dos riesgos son los que constituyen el mayor impacto tanto al TFG como al curso académico, ya que son las principales razones por las que un TFG podría no ser terminado y presentado. Aunque la probabilidad sea pequeña, el impacto sería tal en ambos casos que muy probablemente quedase pendiente para futuros cursos.

Riesgo	RSK-1 Ser muy optimista.
Descripción	Querer abarcar mucho en el proyecto y en el desarrollo, haciendo imposible completarlo en las 300 horas pautadas. Esto puede desembocar en el proyecto a medias o con las partes importantes sin terminar.
Probabilidad	Baja
Gravedad	Alta
Nivel de riesgo	MEDIO
Plan de prevención	Acotar el proyecto a las partes más importantes e interesantes, ideando un desarrollo realista.
Plan de respuesta	Recortar funcionalidad añadida o no esencial.

Tabla 2.3: Riesgo 3 - Ser muy optimista

Riesgo	RSK-2 Mala estimación o gestión del tiempo.
Descripción	Los tiempos estimados del trabajo no son los correctos o la organización ha sido mala y parte del trabajo queda sin hacer
Probabilidad	Baja
Gravedad	Alta
Nivel de riesgo	MEDIO
Plan de prevención	Con una buena planificación, ir comprobando que cada parte del proyecto se esta haciendo en el tiempo estimado, para así asegurarse que la organización es buena.
Plan de respuesta	Recortar tiempo de las funcionalidades secundarias.

Tabla 2.4: Riesgo 2 - Mala estimación del tiempo.

Riesgo	RSK-3 No poder presentar el TFG.
Descripción	A causa de preparar el TFG, otras asignaturas podrían quedar desatendidas y por consiguiente no superadas, por lo que este tendría que ser aplazado a otro curso.
Probabilidad	Media
Gravedad	Alta
Nivel de riesgo	ALTO
Plan de prevención	Dar prioridad a las tareas mas urgentes, siendo las de máxima prioridad las tareas mas cercanas en el calendario.
Plan de respuesta	Hacer horas extra.

Tabla 2.5: Riesgo 3 - No poder presentar el TFG.

2.3. ANÁLISIS DE RIESGOS

Riesgo	RSK-4 Abandono del proyecto.
Descripción	A causa de la desmotivación u otros factores el proyecto puede ser abandonado.
Probabilidad	Media
Gravedad	Alta
Nivel de riesgo	ALTO
Plan de prevención	Estar centrado en el objetivo y sacar motivación dado que es el último proyecto del grado.
Plan de respuesta	Buscar consejo y apoyo en compañeros y amigos.

Tabla 2.6: Riesgo 4 - Abandono del proyecto.

Con esto, teniendo en cuenta los riesgos más comunes en un proyecto de este tipo y teniendo los planes en mente, es más fácil no caer en estos errores y avanzar hacia completar el TFG y con ello la titulación. También, hay que dar las gracias a la tutora ya que se encargó de poner en conocimiento desde el principio de algunos de estos riesgos.

2.3.2. Análisis de riesgos de la aplicación

Cuando hablamos de un desarrollo de una aplicación pensada idealmente para que la utilice un gran número de personas, como es el caso, hay que tener en mente que uno de los retos es como afrontar un posible escalado de la aplicación, ya que aunque los casos de éxito masivo de una aplicación sean pocos, estar preparado para ello puede evitar muchos problemas a futuro.

Como en este caso se ha optado por un desarrollo basado en microservicios, los problemas de escalado se ven reducidos en gran medida, ya que es mucho mas fácil escalar cuatro servidores por separado que hacerlo con uno, pero de forma masiva, ya que en muchas ocasiones el precio y los costes se disparan.

Riesgo	RSK-5 Problemas de escalado.
Descripción	Se pueden producir problemas de escalado debido a un mal diseño de la aplicación o una falta de potencia de los servidores.
Probabilidad	Baja
Gravedad	Alta
Nivel de riesgo	MEDIO
Plan de prevención	Hacer un diseño basado en microservicios para que escalar ciertas partes de la aplicación se mucho más fácil y económico. También, revisar periodicamente las métricas de los servidores y comprobar los datos de uso, para ampliar si fuese necesario.
Plan de respuesta	Ampliar urgentemente la potencia y tamaño de los servidores.

Tabla 2.7: Riesgo 5 - Problemas de escalado

Como ya se ha mencionado con anterioridad, uno de los retos más importantes de esta aplicación será el desarrollo de una interfaz tanto intuitiva como fácil de aprender. Ya que entre el público objetivo de la aplicación se encuentran usuarios que no son asiduos a las nuevas tecnologías, para este proyecto sería un riesgo muy grave que no se cumpla con este punto.

Riesgo	RSK-6 Dificultades para utilizar o aprender a usar la aplicación.
Descripción	Debido a un mal diseño de la interfaz, la aplicación podría tener una mala usabilidad o mala facilidad de aprendizaje, sobre todo para los usuarios no asiduos a las tecnologías.
Probabilidad	Baja
Gravedad	Alta
Nivel de riesgo	MEDIO
Plan de prevención	Hacer mucho incapié en la interfaz a la hora del desarrollo.
Plan de respuesta	Rehacer las partes de la interfaz que sean más confusas e incluso añadir pequeños tutoriales la primera vez que se visiten dichas partes..

Tabla 2.8: Riesgo 6 - Dificultades de uso de la aplicación

Otro de los retos era también, ya que es un problema muy común en estos días, el de brindar una protección decente ante los posibles ataques de ciberseguridad que se pudieran dar. Aunque en este caso no sea un servicio que destaque por tener muchos datos comprometidos de usuarios, ha sido aún así una prioridad aplicar buenas prácticas a la hora de codificar para mantener el código más seguro.

Riesgo	RSK-7 Mala seguridad.
Descripción	Los servicios y servidores podrían tener vulnerabilidades.
Probabilidad	Media
Gravedad	Media
Nivel de riesgo	MEDIO
Plan de prevención	Aplicar buenas prácticas tanto en el despliegue de los servidores como a la hora de la codificación.
Plan de respuesta	Investigar y arreglar los ataques recibidos.

Tabla 2.9: Riesgo 7 - Mala seguridad

2.3.3. Análisis de riesgos para la protección de datos.

Una de las partes esenciales para cumplir tanto la Ley Orgánica 7/2021, 26 de mayo [10] como el Reglamento (UE) 2016/679 del Parlamento Europeo, 27 de abril de 2016 [11], es elaborar un análisis de riesgos sobre el tratamiento de los datos manejados por la web. Gracias a esto se tiene un plan de respuesta antes las posibles consecuencias de un ataque a los servidores y bases de datos.

Riesgo	RSK-8 Robo de datos interno
Descripción	Un trabajador interno con el rol de Superusuario roba los datos de los usuarios.
Probabilidad	Baja
Gravedad	Media
Nivel de riesgo	BAJO
Plan de prevención	Proteger los registros de acceso a los datos y cifrar los mismos.
Plan de respuesta	Contactar con un especialista en informática forense y solucionar las vulnerabilidades.

Tabla 2.10: Riesgo 8 - Robo de datos interno

Ya que los datos recopilados en la aplicación no son muy delicados se ha considerado como gravedades medias. Sin embargo, obviamente es una obligación reaccionar a los posibles problemas y preparar el sistema para que los riesgos no se lleguen a materializar.

Riesgo	RSK-9 Ataque a la base de datos
Descripción	Las bases de datos podrían ser atacadas por ciberdelincuentes, ya que se alojan en servidores online, exponiendo los datos de los usuarios.
Probabilidad	Media
Gravedad	Media
Nivel de riesgo	MEDIO
Plan de prevención	Establecer autenticaciones seguras en todas las bases de datos. Obligar a la renovación de estas autenticaciones. Mantener las bases de datos cifradas.
Plan de respuesta	Contactar con un especialista en informática forense y solucionar las vulnerabilidades.

Tabla 2.11: Riesgo 9 - Ataque a la base de datos

Riesgo	RSK-10 Phishing
Descripción	Se podrían realizar ataques de phishing a los trabajadores de la aplicación para robar su autenticación o los propios datos de los usuarios.
Probabilidad	Media
Gravedad	Media
Nivel de riesgo	MEDIO
Plan de prevención	Formar a los trabajadores para evitar caer en estos ataques. Aplicar firewalls a los equipos de los integrantes del equipo configurados para evitar este tipo de ataque.
Plan de respuesta	Bloquear rápidamente el acceso y los datos sustraídos.

Tabla 2.12: Riesgo 10 - Phishing

Riesgo	RSK-11 Mal uso de los datos
Descripción	Los datos podrían filtrarse inintencionadamente debido a negligencias.
Probabilidad	Media
Gravedad	Media
Nivel de riesgo	MEDIO
Plan de prevención	<p>Establecer unos procedimientos estándar que respeten la privacidad de los datos. Esto incluye el cifrado de contraseñas, la limitación de permisos en base de datos y a los usuarios, y la privatización de todas las APIs de consumo de información.</p> <p>Además, se dará la opción en todo momento a que los usuarios pidan el total borrado de su información de los servidores.</p> <p>Se redactarán unos Términos y Condiciones, Aviso legal y Política de privacidad junto a un asesor legal para explicar cómo y por qué se almacenan los datos. Solo se almacenarán los datos estrictamente necesarios.</p> <p>Se contratarán auditorías regularmente para comprobar el cumplimiento del uso de datos. Formar a los trabajadores con estos procedimientos y asegurar que se respeten.</p>
Plan de respuesta	Lo primero, es obligatorio informar a las autoridades sobre el robo de información. Luego se investigará por qué ha pasado la incidencia. Reformular los procedimientos si fuese necesario. Reformar a los trabajadores si fuese necesario.

Tabla 2.13: Riesgo 11 - Mal uso de datos

2.4. Análisis de costes

Para hacerse una idea aproximada del coste que supondría el proyecto, se ha realizado un pequeño análisis de costes. Para este análisis, se ha supuesto como escenario el de un empleado a media jornada y teletrabajando. Para empezar, se ha buscado cuanto sería el sueldo medio anual de un desarrollador Full Stack Angular, que son unos 22000€ brutos al año [12], que haciendo los cálculos sale a unos 11,46€ la hora.

Añadido a lo anterior, hay que tener en cuenta un extra en concepto de seguridad social del trabajador, que es aproximadamente un 30% [13]. Con este dato y el coste anterior, tenemos que son unos 3,438€ la hora [13].

También se han tenido en cuenta el coste de electricidad con el precio medio de 2024 [14] y la tarifa de internet. Para la electricidad se ha consultado que el consumo medio de un ordenador con dos monitores es de 250W, que para 21 días al mes y 4 horas al día es de 21kWh, junto al dato de 0,0806€/kWh[15] en 2024 da 1,69€ al mes. Para internet se ha hecho un cálculo similar haciendo la parte equivalente de 35€ al mes que es el precio medio aproximado de una conexión de fibra, al final, el coste es de 4,082€ al mes.

Para el ordenador, se ha tenido en cuenta un Mac Mini que empieza en 718,85€[16] por ser económico y una buena opción para desarrolladores, que proporcionalmente suponiendo una vida útil de 5 años da unos 11,96€ al mes. Por último se ha tenido en cuenta dos servidores virtuales de 5€/mes para el despliegue del proyecto.

En la tabla 2.14 se ve un resumen con todos los costes y el cálculo total bruto, que serían 4635,89€ de coste estimado para el proyecto en su totalidad en condiciones normales, media jornada y 300 horas exactas de desarrollo.

Concepto	Precio por unidad (EUR)	Unidades	Coste total (EUR)
Coste del desarrollador	11,46€/hora	300 horas	3438€
Gastos de Seguridad Social	3,438€/hora	300 horas	1031,40€
Ordenador	11,96€/mes	6 meses	71,76€
Electricidad	1,69€/mes	6 meses	10,14€
Conexión fibra	4,082€/mes	6 meses	24,494€
Servidores	5€/mes/servidor	2 servidores ×6 meses	60€
Total			4635,79€

Tabla 2.14: Previsión de costes para el desarrollo del proyecto

Obviamente, algunos costes como el salario, la seguridad social y el ordenador es algo simulado ya que en este proyecto no se ha tenido que hacer ese gasto, pero es más o menos lo que costaría en un entorno de desarrollo real.

2.5. Iteraciones del desarrollo

En cuestiones del desarrollo, se ha intentado seguir un desarrollo iterativo, acabando aproximadamente cada semana con una pequeña versión del software y yendo mejorándola a través de todo el proyecto.

Siguiendo la tabla de la sección 2.2, al final de cada una de las fases del desarrollo (Primera API, Segunda API, etc...), se ha revisado en perspectiva global el estado del proyecto para ver si alguna de las áreas necesitaba mejora para visitar el código y desarrollar una iteración más lograda. A continuación se comentarán algunas de ellas.

Cuando se acabó la primera API, la funcionalidad esperada era la correcta, pero en el desarrollo de la segunda, algunas cosas de la comunicación de autenticación entre ambas no era correcta, así que se hicieron mejoras para gestionar el estado de autenticación y que las otras APIs pudiesen consultar algunos datos.

Al terminar esta segunda API, el funcionamiento era el esperado, pero tanto al desarrollar la tercera como al desarrollar el frontend, se vió que había algunas prácticas mal aplicadas en Spring que se discutirán en el capítulo 8. Esto llevó a tener que replantear bastantes implementaciones y hacer otra iteración sobre la misma y hacer una versión mejorada.

Por último, mientras se desarrollaba el frontend y cuando se procedió con el despliegue, así como haciendo pruebas de calidad al acabar todo, hubo que replantear pequeños errores y fallos de el proyecto en general. Aquí se aplicaron varias tandas de arreglos e integraciones para ofrecer una mejor experiencia.

Gracias al Extreme Programming mencionado anteriormente y teniendo la agilidad que brinda de centrarse en los problemas según van surgiendo, en general durante todo el desarrollo se han aplicado cambios, pruebas y iteraciones de mejora para ir mejorando poco a poco el resultado.

Capítulo 3

Tecnologías

3.1. Tecnologías

3.1.1. Angular

Angular es un framework de desarrollo basado en TypeScript. Para conocer Angular, primero hay que conocer que es TypeScript.

TypeScript [17] es un lenguaje de programación fuertemente tipado pensado como extensión para JavaScript, por encima del cual está construido. Es conocido que la falta de tipos de JavaScript trae a veces algunos problemas (como que se puedan sumar cadenas y números), y TypeScript es la solución a esto. Además, es totalmente compatible con JavaScript, ya que es traducido a este para que se pueda ejecutar en cualquier sitio. Este proceso se llama transpilación de código, y se hace en el momento del **build** de la aplicación, por lo que a la hora de la verdad, el navegador solo verá código JavaScript.

Con esto, se sabe que el lenguaje base de Angular es TypeScript, en el cual se programan la parte de funcionalidad. Todavía queda decir que Angular es una SPA [18] (Single Page Application), que son aplicaciones que cargan un único documento (por lo general el index.html) y que actualizan el contenido de la web basado en las acciones del usuario. Por ejemplo, una navegación de una aplicación clásica haría hacer una recarga de la web, mientras que una SPA simplemente destruye el contenido que ya no hace falta y renderiza el nuevo en tiempo real. Esto es muy poderoso y da una sensación de velocidad que una aplicación clásica es muy difícil que consiga. Los tres framework más populares en la actualidad son SPA (Angular, React y Vue).

Por último, y aunque se podría extender más, queda decir que Angular es una aplicación basada en componentes. La arquitectura en componentes radica en la intención de reutilizar partes de una aplicación todo lo que se pueda. Un componente sería eso, pequeñas partes que se pretenden reutilizar, puede ser desde un botón hasta el propio template de la web, los

cuales con unas simples instrucciones en forma de inputs pueden adaptarse al sitio donde son usados. Los componentes son partes esenciales de Angular y son bloques que contienen lo siguiente:

- Una parte **HTML** para la visualización de la web.
- Una parte **CSS** para dar estilo.
- Una parte **Typescript** con el código que dinamiza lo anterior.

Las ventajas de Angular no son solo los componentes, también hay servicios, enrutadores, formularios reactivos, validación en inputs, interceptores de peticiones, Observables (la forma en la que Angular gestiona las peticiones), guardas para prohibir rutas, etc...

Entre las razones del por qué se ha elegido esta tecnología, están la gran documentación al respecto que se puede encontrar, que además ha sido refactorizada este año y ahora es muy sencillo navegar por ella [19], la familiaridad que tengo con el framework y la buena cuota de mercado actual.

Como punto polémico, hay que decir que Angular es un "framework opinado", es decir, mientras que React es una librería que te deja total libertad de implementación, Angular tiene su propia manera de hacer las cosas (conocida como "The Angular way"), la cual está documentada. Para muchos esto es malo, pero hay que tener en cuenta que Angular te obliga a hacer las cosas de cierta manera para que mantengas unas buenas prácticas.

Para concluir, Angular es increíble, pero hay que decir que no ha sido fácil, ya que la curva de aprendizaje es conocida por ser muy extrema y hay que dar la razón. Quizá algo más sencillo como Vue podría haber sido acertado para este proyecto, pero ha sido una buena elección.

3.1.2. Gestores de paquetes y dependencias

Los gestores de paquetes y dependencias [20] están a la orden del día y prácticamente todos los lenguajes tienen el suyo, es una herramienta imprescindible y que agiliza mucho el desarrollo a la par que evita problemas de dependencias, como ventajas de la utilización de los mismos tenemos lo siguiente:

- Simplifica el desarrollo: Facilita mucho la instalación y la gestión de las dependencias en un proyecto.
- Reutilización y optimización de dependencias: Evita que distintos desarrolladores instalen dependencias o librerías en el proyecto y que pasen inadvertidas, haciendo que no se instalen varias librerías para lo mismo
- Mejora la seguridad: Mantiene al día las dependencias, pudiéndose actualizar de forma muy sencilla.

- Homogeneidad: Casi todos los gestores de paquetes operan de forma muy similar, así que cambiar de unos a otros es muy sencillo.

Debido a que en este proyecto se han utilizado tres lenguajes distintos, también hay tres gestores de paquetes distintos que se irán introduciendo en este capítulo.

3.1.3. Java

Java[21], propiedad de Oracle[22], es conocido por su robustez y fiabilidad, siendo una de las opciones preferidas en la industria del desarrollo sobre todo en España. A menudo se asocia con código Legacy y proyectos antiguos, haciendo que algunos desarrolladores odien el lenguaje, pero nada más lejos de la realidad. Su arquitectura multiplataforma junto a la máquina virtual (**JVM**), permite que las aplicaciones desarrolladas en Java se ejecuten en cualquier dispositivo que tenga instalada JVM, garantizando que se pueda ejecutar.

Java es un lenguaje sencillo de aprender, pero al mismo tiempo muy potente y profundo. Su sintaxis clara y su fuerte sistema de tipado hacen que el desarrollo sea grato y su extensa biblioteca estándar da herramientas para prácticamente todo lo que necesites. Además, Java ha demostrado ser altamente eficiente en el manejo de datos y en la implementación de sistemas complejos, lo que lo convierte en una opción ideal para el desarrollo de backend.

A pesar de la aparición de nuevos lenguajes de programación, Java no tiene nada que envidiarles, ya que se lleva diciendo años que Java está muerto y siempre se mantiene gracias a constantes actualizaciones y mejoras que lo mantienen a la vanguardia de la tecnología.

Aunque con el movimiento de Android a Kotlin Java haya perdido algunos adeptos, sigue siendo un lenguaje más que capaz y que se presta mucho para hacer backend, por lo que se ha utilizado un framework para una parte del mismo. Además de porque es el lenguaje con el que se suele empezar, Java asegura una base sólida para la aplicación y una forma escalable y mantenible de ampliarla.

3.1.4. Spring Boot

Spring Boot es un framework basado en Java muy popular en el sector empresarial y de código abierto, acelerando y simplificando el desarrollo de aplicaciones web y microservicios y ayudando a seguir una arquitectura basada en capas gracias a la inyección de dependencias.

Aunque las aplicaciones con Spring están pensadas para brindar la posibilidad de llegar a ser grandes, también se puede utilizar para la creación de pequeñas APIs que, para este proyecto ha sido de gran ayuda, ya que para esta aplicación en microservicios se ha utilizado Spring Boot en concreto en una de las APIs que conforman la aplicación. El fuerte de Spring en este aspecto es la capacidad de acceso y recuperación de información de bases de datos a través de patrones Repository y Entity junto a los controladores para REST, ha sido realmente sencillo crear una API programando las respuestas de las peticiones que quería que la API aceptase.

Además de lo mencionado, Spring ofrece una poderosa herramienta que es la cadena de seguridad y todo lo relacionado a **Spring Security**, que si bien es complejo al principio, ayuda a blindar las rutas de la API y a dejar acceso solo a las partes de la aplicación que interese. También ayuda a la autenticación y autorización, pudiendo crear filtros realmente complejos por los que se pueden hacer pasar a las peticiones recibidas por la API, o simplemente creando distintos filtros para distintas peticiones, lo cual puede resultar realmente útil.

3.1.5. Maven

Maven es uno de los gestores de paquetes más famosos de Java junto con Gradle, sirve tanto para automatización de builds como para la gestión de todas las dependencias a través del fichero pom.xml, que es el archivo principal de cualquier proyecto hecho con este gestor. En este se incluye todo lo que queremos importar al proyecto declarándolas como nodos XML. En este archivo es en el que se han declarado todas aquellas librerías sobre todo de Spring que fueron utilizadas para el proyecto.

Para la instalación y creación de distribuibles, se puede ejecutar el comando **mvn install**, el cual genera un archivo JAR, reduciendo errores en el desarrollo y asegurando la consistencia, lo que se puede juntar con CI/CD(Integración continua, entrega continua) para un desarrollo muy iterativo.

Se decidió utilizar Maven ya que es el gestor con el que había experiencia en diversos proyectos durante el grado, aunque que Gradle es una alternativa totalmente válida pero menos utilizada. Es una tecnología que ha venido bien pero tampoco se ha profundizado mucho en ella, pues solo se ha utilizado como gestor de dependencias para agilizar el desarrollo.

3.1.6. Node.JS

En desarrollo web y en la ejecución en servidores, Node.js [23] se ha consolidado como una herramienta fundamental para la creación de aplicaciones y APIs. Node es un entorno de ejecución de JavaScript de código abierto y multiplataforma, lo que permite ejecutar código JavaScript en el lado del servidor sin que esté vinculado a un navegador, a diferencia del JavaScript tradicional. Esto abre muchas posibilidades para el desarrollo web y para el backend, ya que permite crear aplicaciones web completas con un solo lenguaje de programación, simplificando notablemente el proceso de desarrollo y mantenimiento.

Un aspecto fundamental de Node es su arquitectura asíncrona y basada en eventos, lo que permite a Node manejar múltiples solicitudes simultáneamente de manera eficiente, sin necesidad de esperar a que cada una finalice antes de procesar la siguiente. Esta característica lo convierte en la herramienta ideal para desarrollar aplicaciones web en tiempo real, como chats, juegos online y plataformas de streaming.

El ecosistema de Node.js es otro punto a destacar, ya que tiene muchas librerías, frameworks (como Express), módulos y muchas herramientas a su alcance, aportando lo necesario

para construir aplicaciones web robustas y escalables con relativa facilidad. Además, la comunidad de desarrolladores de Node y de JavaScript es excepcionalmente activa y colaborativa, lo que garantiza un flujo constante de apoyo y conocimiento.

En el contexto de este TFG, Node se ha utilizado indirectamente a través de NestJs. La posibilidad de crear aplicaciones web dinámicas y en tiempo real (lo cual es muy interesante para aplicaciones que tienen Server Side Rendering), junto con la simplicidad y flexibilidad que ofrece Node.js, permite enfocarse en la innovación y el desarrollo de soluciones de alto impacto.

Node es hoy en día un entorno de ejecución que, aunque tiene sus detractores y alternativas (Bun y Deno), forma parte del día a día de muchas empresas.

3.1.7. Npm

Npm (Node Package Manager), es el gestor de paquetes por defecto para Node.js y en muchos de los frameworks JavaScript (React, Angular, Vue). Es una herramienta fundamental y muy beneficiosa para cualquier proyecto que desarrolle con Node o JavaScript, ya que permite instalar, actualizar, eliminar dependencias y llevar a cabo un seguimiento y un repositado de las versiones de cada una de forma sencilla y eficiente. Como beneficios de utilizar npm tenemos:

Además de la gestión de paquetes y de los beneficios que ya se comentaron sobre los gestores de paquetes, Npm permite la ejecución de scripts personalizados y de comandos específicos, como en el caso de angular, hacer build y servir la aplicación.

3.1.8. NestJs

NestJS es un framework de Node.js y basado en TypeScript para la construcción de aplicaciones en el lado del servidor. Es una tecnología muy eficiente ya que a parte de ser sencilla, es muy rápido programar en ella y recorta mucho el código necesario para ciertas funciones.

Gracias a poder utilizar muchas de las funcionalidades de Node, NestJS es una excelente elección para la parte de APIs, y no por nada está ganando mucha popularidad día de hoy. Si juntamos esto con el ahorro de tiempo ya mencionado, es una elección perfecta para el proyecto. Tuve en consideración utilizar Express o incluso Django, pero NestJS es muy parecido a Angular por lo que hace un tándem muy interesantes, además de que ya había trabajado con ello anteriormente.

Aporta una arquitectura modular igual que Angular y un funcionamiento muy similar, teniendo la inyección de dependencias, las guardas, interceptores, servicios y otras funciones idénticas a las de Angular. Además, como es TypeScript, me ahorro mucha problemática asociada a JavaScript por el tema del tipado.

En resumen, NestJS tiene un gran rendimiento, es fácil de utilizar y ahorra tiempo, aparte de ser una herramienta muy potente que todo programador familiarizado con Angular y que quiera saber también back-end debería conocer.

Nest es la tecnología elegida para el desarrollo de otra de las APIs que forma parte del back-end del proyecto.

3.1.9. PHP

PHP es un lenguaje que se ejecuta en el lado del servidor creado en 1994. Específicamente se diseñó para el desarrollo web y ha evolucionado considerablemente, siendo actualmente compatible con la mayoría de servidores web como Apache, Nginx e IIS. Su popularidad se debe en gran parte a su facilidad de uso y a su integración nativa con HTML, lo que permite a los desarrolladores insertar scripts PHP directamente en páginas web sin necesidad de archivos separados.

Una de las características más destacadas de PHP es su capacidad para interactuar con bases de datos. Es compatible con una amplia variedad de sistemas de gestión de bases de datos, incluyendo MySQL, PostgreSQL, SQLite y Oracle, lo cual permite crear aplicaciones web dinámicas que pueden almacenar y recuperar datos de manera eficiente. Además, PHP ofrece soporte para PDO (PHP Data Objects), una extensión que proporciona una interfaz de acceso a bases de datos consistente, lo que facilita la escritura de código seguro y portable.

Sin embargo, hay que decir que PHP no son todo cosas buenas, es un lenguaje arcaico, tiene carencias en muchas cosas. Hay funcionalidades básicas como algunas formas de tipado o los genéricos que no son nativos, hay que anotarlos en los comentarios del IDE para que este lo entienda, y como esta muchas cosas más. PHP fue el rey y lo sigue siendo, pero caso toda su cuota de mercado se debe a proyectos muy legacy.

PHP también es conocido por ser muy inseguro, ya que es extremadamente fácil cometer errores que provoquen RCE (Ejecución remota de comandos), ataques XSS a través de JavaScript, inyecciones SQL y un sin fin más de problemas. El lenguaje como tal en las últimas versiones no tiene por que ser inseguro, pero la poca pericia de algunos programadores provoca que lo sea.

La comunidad de php también esta en decadencia, cada vez hay menos librerías e incluso en cosas tan populares como Google Firebase tiene que apoyarse en librerías no oficiales.

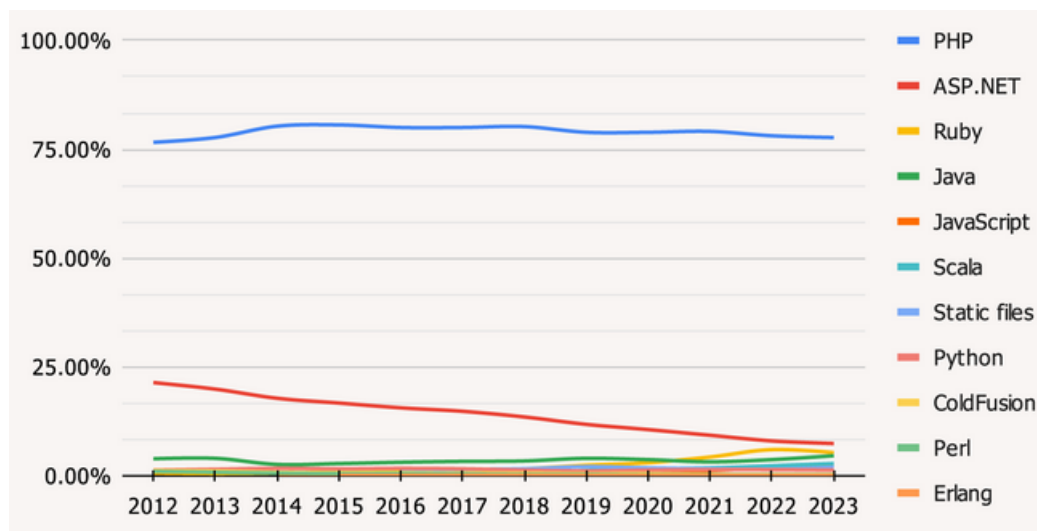


Figura 3.1: Uso de php a lo largo del tiempo

Básicamente, PHP a día de hoy vive de sus frameworks(Laravel, Symfony y CodeIgniter), ofreciendo soluciones avanzadas para la creación de aplicaciones web robustas y escalables. Estos frameworks proporcionan estructuras y componentes reutilizables, lo que acelera el desarrollo y fomenta las mejores prácticas en la programación.

En términos de rendimiento, las versiones recientes de PHP (a partir de PHP 7) han introducido mejoras significativas, sobre todo OP caché, que es un sistema de cacheado muy potente.

Aunque es un lenguaje antiguo, poco eficiente y a veces con carencias, se ha elegido un framework de PHP para el backend ya que mi puesto actual se basa en cierta medida en PHP y es un lenguaje fácil y rápido de programar.

3.1.10. Composer

Composer es una herramienta esencial en el ecosistema de desarrollo de PHP, actuando como un gestor de dependencias. A día de hoy, casi todos los proyectos modernos de PHP implementan Composer como gestor de paquetes, asegurándose de que se instalen las versiones correctas y que no haya conflictos entre ellas.

Estas dependencias en un archivo `composer.json`, el cual automatiza el proceso de descarga e instalación de los paquetes necesarios, facilitando enormemente la configuración del entorno de desarrollo para nuevos programadores que se unan al proyecto. Esto, además de ahorrar tiempo, evita muchos problemas de incompatibilidades entre distintas librerías

Además de la gestión de dependencias, Composer también introduce el concepto de *autoload*, lo que simplifica el uso de clases y archivos dentro del proyecto. Con el autoloading,

no es necesario incluir los archivos PHP, sino que automáticamente hace una búsqueda de los archivos cuando se utilizan.

Igual que los otros gestores de paquetes, ha agilizado el desarrollo y se ha tenido que gastar menos tiempo en gestión de dependencias.

3.1.11. Slim Framework

Slim es un microframework de PHP diseñado para desarrollar aplicaciones web y APIs de manera rápida y eficiente. Se beneficia de la robustez del lenguaje, proporcionando una estructura ligera y flexible que facilita el manejo de rutas, peticiones y respuestas HTTP. Slim tiene su nicho gracias a su simplicidad y la facilidad con la que se integra con otras herramientas y bibliotecas de PHP.

Básicamente, el framework tiene todo lo que necesitaba en un framework backend menos un ORM, el cual se puede instalar a parte, por ejemplo **Doctrine**(nativo de Symfony) o **Eloquent**(nativo de Laravel), pero a mitad de la instalación de Doctrine decidí dar un paso a tras debido a que me iba a quitar más tiempo que los beneficios que me iba a aportar para una API tan pequeña

El diseño minimalista permite ejecutar una arquitectura a capas de forma muy sencilla. Además, Slim ofrece soporte nativo para middleware, permitiendo la adición de funcionalidades como autenticación con JWT, validación y manejo de errores de una manera modular y reutilizable.

Slim también destaca por su contenedor de dependencias, el cual es muy potente para hacer las inyecciones necesarias en cada una de las clases de la API de forma individual. Utilizando Composer, el gestor de paquetes estándar de PHP, ha sido muy fácil integrar librerías externas.

Sin embargo, aunque Slim facilita enormemente el desarrollo de aplicaciones web, no está exento de las limitaciones inherentes a PHP. Como se mencionó anteriormente, la falta de tipado fuerte y genéricos nativos y el mal rendimiento (sobre todo en local debido a incompatibilidades con Windows) han estado presentes. La seguridad, al ser una API que además se ha protegido con JWT y no tener frontend no me ha preocupado tanto, además de que la intención fue de aplicar buenas prácticas.

Slim tiene todo lo malo de PHP, pero es ligero, rápido, fácil de montar y viene con todo lo básico, así que ha sido una buena elección.

3.1.12. Tokens JWT

JWT es el acrónimo de JSON Web Token y es un estándar(RFC 7519) de seguridad para la transmisión de información como un objeto JSON como su propio nombre indica. Los tokens JWT están firmados digitalmente, ya sea por clave secreta o con un par de claves

pública y privada, siendo esta última la más conveniente. Estos tokens son habitualmente utilizados a la hora de hacer identificaciones frente a servicios web, como por ejemplo APIs del backend que necesiten protección y se busque que no sean públicas.

Los JWT tienen la siguiente estructura:

- **Encabezado:** Tiene la información del token referente al algoritmo utilizado y que tipo de token es.
- **Payload:** Contiene los datos importantes, entre los que suelen estar el **iat** y el **sub**, siendo el momento en el que se firmó el token y el tema (subject) del mismo. Además, en este apartado se puede añadir todos los datos que sean valiosos para el desarrollo. Por ejemplo en esta parte se ha incluido el rol y el id de usuario como se puede ver en la figura 3.2.
- **Firma:** La firma digital que blinda el JWT y asegura que el token sea original. Se genera con las claves previamente mencionadas.

PAYLOAD: DATA

```
{
  "id": "66641*****9f4c",
  "role": "superuser",
  "iat": 1718652800,
  "exp": 1718656400
}
```

Figura 3.2: Payload de un JWT

Lo bueno de este tipo de autenticación es que puedes mandar datos contenidos en el propio token y simplemente verificar que el token sea válido (a través de la clave pública), ya que aunque el contenido del payload se pueda ver sin claves, lo cual hace que no debamos poner contenido sensible, al verificar que con la clave el token es bueno, tenemos una forma de asegurarnos de que los datos no han sido modificados.

Las tácticas habituales con los tokens son dos: utilizarlos como header `.^authorization.` como una cookie. Ambos escenarios tienen sus pros y contras, los cuales veremos en la parte de implementación. Como recurso para informarme de todo esto y para probar los tokens se ha acudido a [JWT.io \[24\]](#), un recurso fantástico y muy conocido.

En el proyecto se han utilizado tokens JWT para los requisitos de información entre los microservicios y desde el frontend. La API de login genera un token JWT que autoriza al usuario y representa la sesión almacenada. También, se usa esta autenticación para admitir el uso de ciertas peticiones enviadas, más concretamente todas de las tres APIs que conllevan un riesgo si se dejan públicas. Se han implementado como cookie en vez de como header ya

que al estar todos los microservicios bajo el mismo dominio se puede añadir una capa mas de seguridad gracias a las propiedades para blindar las cookies.

3.1.13. Docker

Docker es una plataforma para virtualizar paquetes de software en contenedores permitiendo empaquetar distintas partes de la aplicación para la ejecución sin necesidad de prerequisites ni dependencias, ya que se incluyen en el mismo contenedor. Con esto, es muy fácil ejecutar la aplicación en cualquier máquina sin necesidad de instalar nada previo y convierte la migración entre servidores en algo trivial. Hace que las mismas sean portables y, como es tan fácil de desplegar, da la posibilidad de un escalado horizontal(en vez de adquirir servidores mas potentes, se adquiere más número) fácil [25].

Estos paquetes mencionados se categorizan como contenedores, que son componentes ejecutables por si mismos. Si fuese necesario también se puede coordinar los contenedores entre ellos, cosa que en este proyecto se ha tenido que aplicar.

En microservicios, la forma mas común de aplicar Docker es separar cada servicio en un contenedor distinto. Esto hace extremadamente fácil el escalado de la aplicación, ya que por ejemplo una API se podría mover a otro servidor solo con mover el contenedor en cuestión. Además, se puede incluso ser más modular y separar también la base de datos en otro contenedor y hacer una comunicación entre contenedores.

En este caso, se ha hecho contenedores de cada una de las bases de datos, cada una de las APIs y del frontend Angular.

3.1.14. Git y Github

En el sector estas tecnologías son más que conocidas y son los reyes del control de versiones. Git [26] nos permite gestionar el trabajo en equipo, el versionado y el historial de las versiones de nuestra aplicación para tener un contexto de la misma y trabajar de una forma más cómoda. El funcionamiento es más que conocido, creas un **repositorio**, el cual no es más que el archivo donde se van a guardar todas las versiones de tu proyecto, y luego al hacer los cambios les guardas en el repositorio a través de **commits**, que son pequeños hitos que marcamos en el desarrollo. Luego, se hacen públicos los cambios y cualquier contribuidor les verá. Junto a esto están las ramas que representan bifurcaciones en el desarrollo para trabajo diversificado en distintas áreas de la aplicación que luego se unirán para hacer una **release** (liberación de una versión) [27].

La forma establecida para trabajar en equipo son las ramas, que permiten a los desarrolladores trabajar en diferentes características o bugfixes de manera aislada y en paralelo, sin afectar el código de las otras ramas o de la versión en producción. Git también ofrece soluciones para la resolución de conflictos generados por la unión de ramas y la integración continua (por ejemplo GitHub con **Github Actions**).

Por otro lado, GitHub es una plataforma basada en Git para entornos de trabajo colaborativos, ya sea de forma pública o de forma privada y empresarial. GitHub integra todas las ventajas que tiene Git en una interfaz fácil de usar y que facilita el trabajo con Git gracias al entorno gráfico. En otros entornos con más personas habilita también las "pull request" para que distintos compañeros se revisen el código antes de llevarlo a producción, y además, un buen control del repositorio y de los permisos.

Hay otras alternativas populares como BitBucket o GitLab (utilizado por la UVa), pero aunque esté más acostumbrado a las dos anteriores por mi trabajo y la facultad, la preferencia ha sido utilizar GitHub ya que es mucho más común en el sector.

Git y GitHub son temas extensos y con diversos flujos de trabajo populares[28], pero en este trabajo al ser solo una persona se va a utilizar simplemente para llevar el control de versiones y gestionar los archivos del proyecto.

3.1.15. MySQL

Es el motor de bases de datos por excelencia, cuando se piensa en SQL se suele pensar en este motor, y como es uno de los más habituales se ha utilizado para una de las APIs. Las claves de este sistema de gestión cabe recalcar la fiabilidad, rendimiento y facilidad de uso, pues es de los sistemas que históricamente más se ha utilizado en empresas. Como se puede ver en la figura 3.3, en la encuesta de uso de StackOverflow 2022 salió en primera posición, y aunque otros sistemas como PostgreSQL vienen pisándole los talones debido a las innovadoras características, se sigue manteniendo en lo más alto.

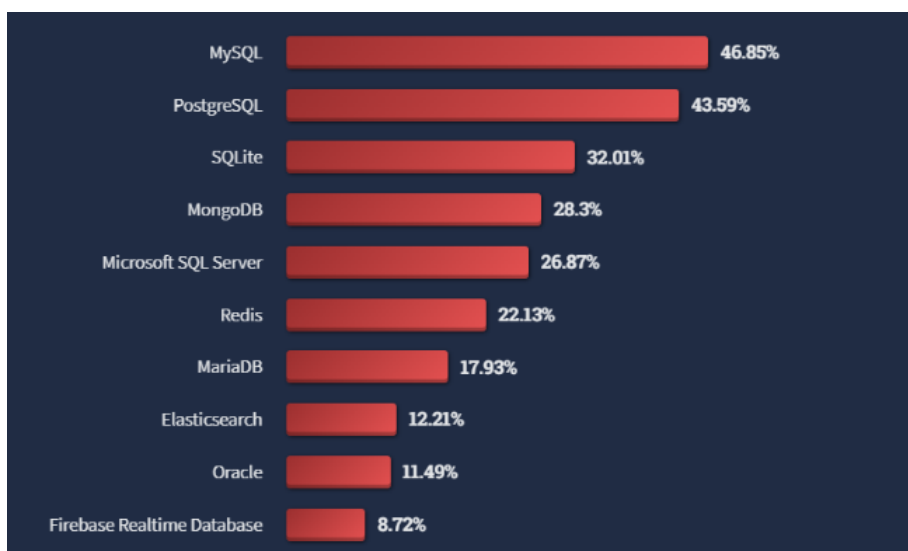


Figura 3.3: Encuesta de uso de bases de datos StackOverflow 2022

Otras de las ventajas es su capacidad de manejar grandes volúmenes de datos, la consistencia, el enfoque bloqueante de las transacciones y la posibilidad de hacer rollback de las

mismas y la facilidad de crear y cargar backups. Además, tiene una comunidad muy activa, hay muchos recursos online y es muy accesible. Las razones de elegir MySQL es que estoy muy habituado a utilizar esta tecnología y creo que es muy potente y cómoda.

3.1.16. PostgreSQL

También llamada **Postgres**, es un sistema de gestión SQL de código abierto y con características muy interesantes que no tienen muchos de sus competidores. Se está convirtiendo rápidamente en el nuevo rey de SQL y es gracias a lo robusto y escalable que es. Las principales ventajas de utilizar Postgres son que ofrece un amplio conjunto de características avanzadas, incluyendo tipos de datos personalizados, herencia de tablas y soporte para JSON, lo que permite a los desarrolladores trabajar con datos relacionales y no relacionales si fuese necesario de manera eficiente.

Una de las cosas que más me han gustado a la hora de utilizar Postgres en otros proyectos son las **Vistas materializadas** [29], un concepto similar a las vistas clásicas de SQL pero que ofrecen un extra de rendimiento, ya que en vez de ser una mera forma de mostrar los datos como las vistas, reserva tablas especiales con los datos listos, haciendo que en sistemas que es más importante leer que escribir sean muy interesantes.

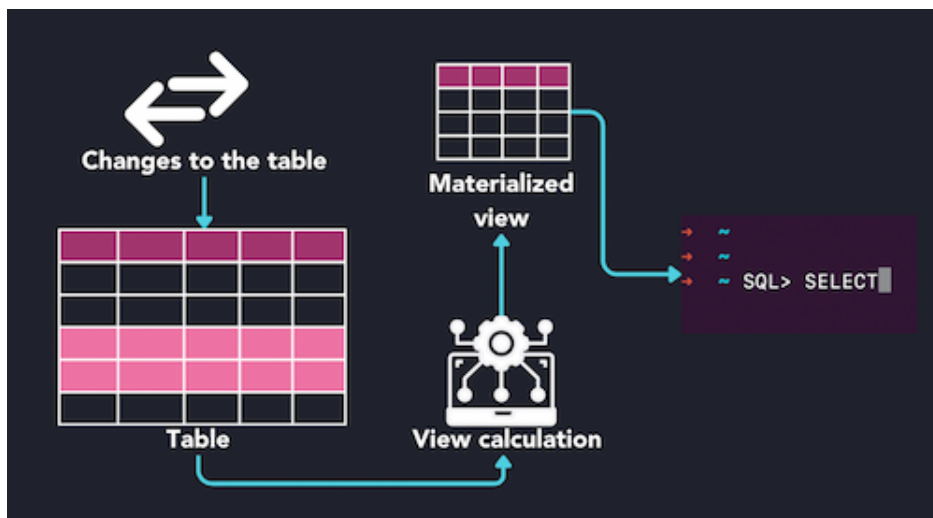


Figura 3.4: Esquema de una vista materializada

Los contras que pudiera tener (más por desconocimiento que por ser contras per se) son mucho menores que las ventajas, y aunque es cierto que todas estas técnicas avanzadas no se han aplicado, ha resultado muy cómodo trabajar con Postgres en otra de las APIs y las pequeñas diferencias que tiene con MySQL no han sido ningún problema.

3.1.17. MongoDB y NoSQL

Mongo es la alternativa gratuita más popular que implementa NoSQL ofreciendo una alternativa moderna a las bases de datos SQL relacionales que ya conocemos (MySQL, Postgres, MariaDB, etc...). NoSql trabaja con documentos JSON que no se almacenan en tablas relacionadas, es decir, que no se guardan relaciones entre las columnas de las mismas, haciendo que la forma en la que se guardan los datos sea más libre. Es muy utilizado por todas aquellas aplicaciones y proyectos que van a tener ingentes cantidades de datos y no necesitan de relaciones. Los campos donde más se utiliza NoSQL es el big data, machine learning, data science, redes sociales, etc...

Una de las principales ventajas frente a SQL es el rendimiento y escalabilidad, ya que NoSQL es mucho más rápido en términos de lectura como podemos ver en la figura 3.5, lo cual se va notando más a medida que hay más datos almacenados que es cuando brilla NoSQL [30].

Obviamente para esta aplicación no hacía falta ni mucho menos fijarse en un máximo rendimiento ya que los potenciales datos que vamos a tener no llegan hasta un posible cuello de botella, además de que las tablas SQL equivalentes no serían tan complejas y las consultas tampoco, pero se buscaba probar distintas bases de datos y trabajar con ellas para ver sus diferencias.

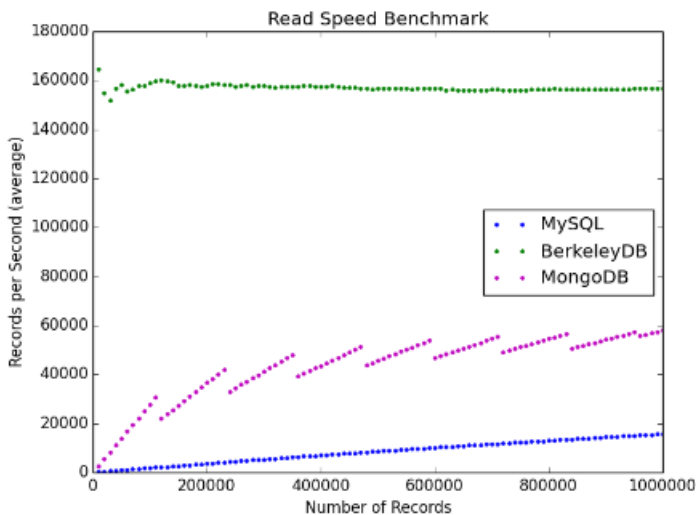


Figura 3.5: Comparación entre Mysql y MongoDB

3.1.18. Tailwind CSS

Tailwind es un framework de CSS que se ha ido convirtiendo en el principal y poco a poco le esta ganando terreno al ya conocido Bootstrap. Sin embargo, Tailwind es distinto

a Bootstrap en el sentido de que este último es una librería de componentes mientras que Tailwind es un framework que aporta clases de utilidad y sistema de grid para prácticamente cualquier cosa que se quiera hacer en CSS. Aunque Bootstrap también aporta el sistema de grid y algunas clases de utilidad, Tailwind es mucho más completo en este aspecto.

La comodidad de este framework es tal que han sido muy pocas las líneas de CSS clásico las que han hecho falta para el proyecto entero. Además, Tailwind es muy configurable, pudiendo importar solo los módulos que se necesiten o generando tus propias paletas de colores. Juntando el framework con TailwindUI, una librería de componentes muy completa de los mismos desarrolladores, la parte de css y html ha sido muy rápida de hacer, teniendo que hacer modificaciones para adaptar los componentes a angular y a la reactividad necesaria en los formularios.

Este framework es muy querido por la comunidad y ya en el StateOfCss de 2023 se ha visto que el sector tiende a abandonar Bootstrap y otros competidores y a trabajar con Tailwind, en la figura 3.6 se puede ver como el interés que genera uno y otro framework es impactante.

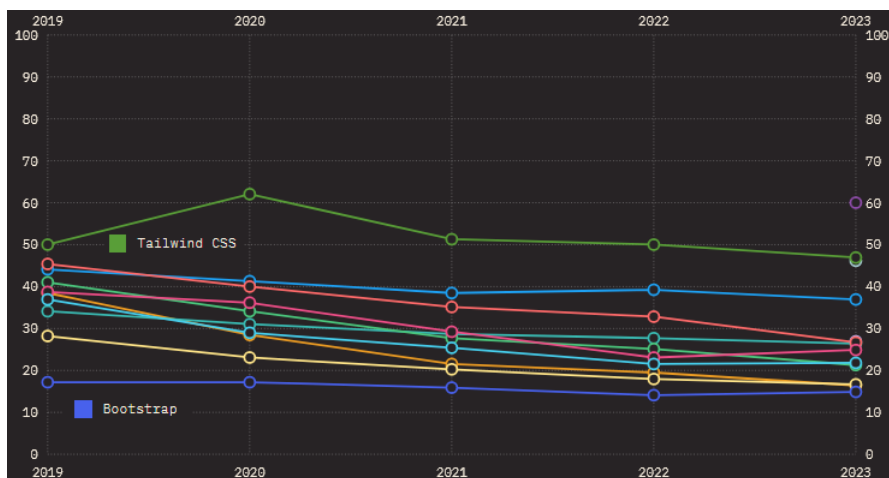


Figura 3.6: Interés por años (Frameworks css)

3.2. Herramientas

3.2.1. Visual Studio Code

Para el desarrollo de este Trabajo de Fin de Grado, se ha utilizado Visual Studio Code (VS Code) como el entorno de desarrollo integrado (IDE) de confianza. VS Code, desarrollado por Microsoft y basado en el editor Atom, se ha ganado la preferencia de muchos desarrolladores gracias a su versatilidad, robustez y gran personalización. Heredando la flexibilidad y los estilos de Atom pero con un toque mucho más moderno, VS Code ha evolucionado para ofrecer un rendimiento superior y una experiencia de usuario mejorada.

VS Code ha facilitado enormemente la tarea durante el desarrollo de este proyecto, han sido muchos los IDEs que se han probado durante el grado, pero unos están totalmente obsoletos (Netbeans y Eclipse), y otros eran muy poco personalizables y requerían licencia como la familia IDEA (PHPstorm, IntelliJ, PyCharm...), pero ninguno me ha llegado a parecer tan bueno como Code. Sus extensiones y plugins, como el soporte para múltiples lenguajes de programación y herramientas de control de versiones como Git, han sido cruciales para mantener la organización y la eficiencia en el trabajo. Hay que tener en cuenta que se ha trabajado con cuatro lenguajes y frameworks distintos, docker y otras tecnologías y VS Code no ha dado problemas con ninguno.

La capacidad de personalizar el entorno de desarrollo, desde los temas de color hasta los atajos de teclado, ha permitido optimizar el flujo de trabajo y centrarme en la escritura y depuración del código sin distracciones.

Además, las características integradas de VS Code, como la terminal, el depurador y el IntelliSense, han mejorado significativamente la productividad. La terminal integrada me ha permitido ejecutar comandos directamente desde el editor, mientras que el depurador visual ha simplificado la identificación y resolución de errores.

3.2.2. Trello

Trello es una herramienta desarrollada por Atlassian. Es una de las herramientas más utilizadas en el ámbito de las empresas de desarrollo software para la planificación y la implementación de Scrum.

La decisión de utilizar esta herramienta ha venido para aprovechar la familiaridad ya que se han impartido clases de la misma en la asignatura Planificación y Gestión de Proyectos. Por ello, me ha facilitado las cosas y se ha ahorrado tiempo de aprendizaje, no sin además aprender más a fondo sobre ella.

Trello contiene diferentes herramientas, de las cuales las que más se han utilizado han sido:

- **Tablero Kanban:** El tablero kanban es una herramienta para mejorar el flujo de trabajo, tanto individual como en equipo. Consta de columnas que representan fases del desarrollo de ciertas tareas. Con ella podemos ver de forma muy visual y sencilla en qué estado se encuentran las tareas que se están realizando.
- **Listas y Tarjetas:** Trello permite crear listas y tarjetas para organizar el backlog del producto, asignar tareas a los miembros del equipo y definir puntos de historia para cada tarea en conjunto.
- **Control del progreso:** Paquete de herramientas para comprobar el progreso del sprint y gestionar el tiempo restante, incluyendo etiquetas y fechas límite para mantener el seguimiento de las tareas.

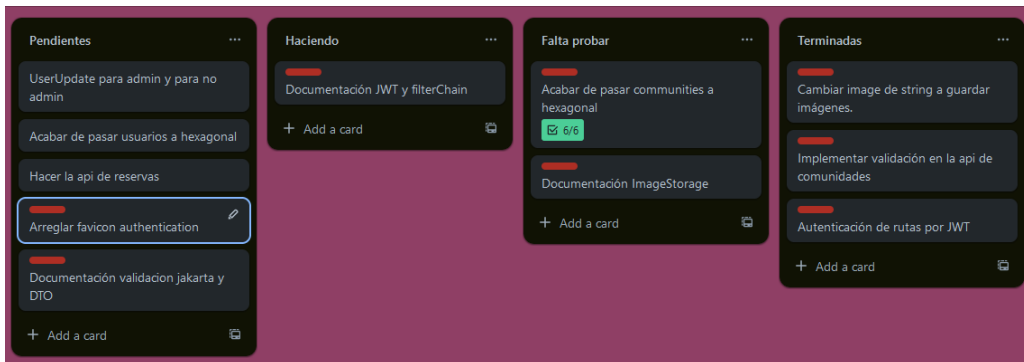


Figura 3.7: Tablero kanban de Trello

Había otras opciones disponibles como MS Project, Jira (también de Atlassian) o tableros kanban independientes, pero como herramienta unificada y sencilla, Trello fue la mejor opción.

3.2.3. Draw.io

Para el modelado del análisis y el diseño de este proyecto se ha optado por usar draw.io [31], una herramienta ya conocida en el sector creada por Jgraph. Es una herramienta para la creación de diagramas UML (Lenguaje de Modelado Unificado) pero también para otro tipo de diagramas. Se pueden crear de diversos tipos y en el proyecto se ha utilizado para los mencionados a continuación.

- **Diagrama de Clases:** Representación visual de las clases e interfaces que conforman la aplicación y la relación entre ellas. Junto a esto, se muestran también los atributos y operaciones de dichas clases, así como herencia, composición y la relación numérica entre dos clases. Se han realizado dos, uno en el Análisis y otro en el Diseño.
- **Diagrama de Despliegue:** Muestra la arquitectura del sistema. Consta de varios nodos y de una forma visual plasma cuáles son las relaciones entre los nodos y cómo se conectan entre sí. Los nodos suelen ser sistemas hardware.
- **Diagrama de Secuencia:** Representación gráfica de cómo las distintas clases van a intercambiar mensajes entre sí. Tiene líneas de vida que representan el paso del tiempo y también están representados los actores que interactúan con el sistema.

La elección ha sido por esta en vez de las dos alternativas que se han visto durante el grado (Visual Paradigm y Astah Professional), porque en la investigación previa parecía más rápida, práctica, moderna e intuitiva, de esta forma, la idea era agilizar el análisis y el diseño.

3.2.4. Microsoft Teams

Microsoft Teams es la aplicación colaborativa de mensajería para entornos de trabajo. Esta pensada para la comunicación entre miembros y equipos de trabajo. Es una de las herramientas mas utilizadas de hoy en día en empresas. Las funciones mas importantes son:

- **Mensajería instantanea:** Permite mandar mensajes de forma instantanea tanto a usuarios online como offline, permitiendo así la comunicación a distancia.
- **Videollamadas y llamadas de voz:** Con esta utilidad es muy fácil hacer reuniones con equipos de trabajo. Es muy fácil y sencillo de utilizar y permite tanto las llamadas por voz como la conexión con cámara.
- **Pertenencia a grupos:** Un usuario puede pertenecer a varios grupos de trabajo, equipos o empresas, cada uno con su chat independiente. Esto es especialmente útil para Project Managers que estén a cargo de varios equipos.

Esta ha sido la elección de herramienta de comunicación entre el alumno y la tutora del proyecto, facilitando la comunicación gracias a la mensajería instantánea.

3.2.5. Overleaf y TeXStudio

La escritura de esta memoria comenzó en Overleaf, una plataforma de edición colaborativa en la nube para documentos en LaTeX. Overleaf es ideal para trabajos colaborativos debido a su interfaz sencilla y sus numerosas ayudas de autocompletado, lo que facilita enormemente el trabajo con LaTeX.

LaTeX es un sistema de software para la preparación de documentos que, a diferencia de otros sistemas como Microsoft Word, permite al usuario trabajar en texto plano en lugar de formato visual. Aunque puede ser tedioso al principio, opté por esta opción porque los documentos escritos en LaTeX suelen tener un formato más formal y estructurado. Las virtudes de LaTeX incluyen la generación automática de índices, la separación por secciones y la facilidad para enlazar la bibliografía, lo que resulta en documentos de alta calidad.

Sin embargo, debido a la pesadez y lentitud de Overleaf, especialmente cuando el proyecto creció en tamaño y complejidad, se decidió migrar a TeXstudio. TeXstudio es un editor local que, aunque no ofrece las ventajas online, es un entorno más ágil y rápido para la edición de documentos extensos. La capacidad de trabajo compilación rápidos hicieron de TeXstudio la opción preferida para finalizar este trabajo. A pesar de ser una aplicación local, su eficiencia superó las limitaciones encontradas en Overleaf, permitiendo una mucho mejor experiencia.

3.2.6. GitHub Copilot

GitHub Copilot es el asistente de código de GitHub para tu entorno de desarrollo, en el caso de VS Code tiene integración nativa. Este asistente impulsado por inteligencia artificial

sugiere fragmentos de código y soluciones en tiempo real con el contexto del código cercano, llegando incluso a ver el contexto de gran parte del proyecto. Trabajar con esta herramienta, por lo general, puede acelerar significativamente el proceso de desarrollo.

Sin embargo, es crucial destacar que el ingeniero siempre debe revisar y validar el código sugerido por Copilot, ya que muchas soluciones no son adecuadas para ciertos casos concretos o son directamente malos. La calidad de las sugerencias de Copilot dependen en gran medida de la calidad del código del programador y del contexto del proyecto, si el código es deficiente, Copilot puede aprender "mal" y sugiere soluciones inadecuadas.

Por lo tanto, aunque GitHub Copilot puede aumentar la eficiencia, la responsabilidad final de asegurar que el código sea correcto, seguro y eficiente recae en el programador, así como de entrenarlo con buenas prácticas.

Un punto polémico de esta herramienta es que se debe desactivar la opción de que GitHub utilice tus datos para entrenar a la IA, ya que si no los datos privados de tu proyecto se verían expuestos.

```
1 function calculateDaysBetweenDates(begin, end) {  
    var beginDate = new Date(begin);  
    var endDate = new Date(end);  
    var days = Math.round((endDate - beginDate) / (1000 * 60 * 60 * 24));  
    return days;  
}  
2
```

Figura 3.8: Sugerencia de código de GitHub copilot

3.3. Herramientas menores

3.3.1. Thunder Client

Thunder Client es una extensión de Visual Studio Code para hacer pruebas y peticiones a API Rest, es muy ligera y rápida y esta muy bien integrada con VS Code al ser una extensión, lo que la hace especialmente conveniente para el flujo de trabajo.

Aunque Postman es una herramienta más completa y con una mayor cantidad de funcionalidades avanzadas, Thunder Client era una mejor elección en este caso. La comodidad de no tener que cambiar entre aplicaciones y poder realizar pruebas de API directamente desde el mismo entorno donde estoy desarrollando el código ha agilizado significativamente el proceso de desarrollo.

Ha facilitado la gestión y prueba de APIs durante todo el proyecto, tarea importante teniendo en cuenta que son tres APIs y hay interacción entre algunas.

3.3.2. JWT.io

Ya se ha mencionado en la sección anterior, la página web jwt.io ha sido una herramienta esencial para el manejo y verificación de JSON Web Tokens (**JWT**). Proporciona una interfaz intuitiva donde se pueden codificar, decodificar y verificar JWT de manera sencilla y rápida.

Es particularmente útil para depurar y validar tokens durante el desarrollo, permitiéndome pegar un token y ver instantáneamente su contenido decodificado, así como cualquier firma y payload asociados. Además, la web ofrece documentación y ejemplos claros que han facilitado la comprensión y uso de JWT en el proyecto.

Capítulo 4

Análisis

El análisis de software es una fase importante en el proceso del desarrollo de sistemas. Permite identificar y definir las necesidades del usuario y los requisitos del sistema de manera detallada. Este proceso no solo abarca la creación de casos de uso y la especificación de requisitos, sino que también incluye la evaluación y modelado del comportamiento y la estructura del sistema.

Una de las principales funciones del análisis de software es minimizar el riesgo de errores y malentendidos durante las etapas posteriores del desarrollo. Al identificar claramente los objetivos, limitaciones y requisitos del sistema desde el principio, se pueden evitar costosos refactorizados y retrasos.

4.1. Requisitos

A continuación se detallaran los requisitos de la aplicación, que es una recopilación de necesidades para ponerle solución a las necesidades que plantea el sistema. Esto se divide en **requisitos funcionales**, es decir, necesidades de como funciona el programa, y **requisitos no funcionales**, que son necesidades de como debe funcionar el sistema en general sin centrarse en necesidades de funcionamiento específicas.

4.2. Requisitos funcionales

RF-01	El sistema debe permitir al Superusuario crear usuarios.
RF-02	El sistema debe permitir al Superusuario borrar cualquier usuario.
RF-03	El sistema debe permitir al Superusuario ver una lista de usuarios total.
RF-04	El sistema debe permitir al Superusuario editar cualquier usuario.

RF-05	El sistema debe permitir al Superusuario crear comunidades.
RF-06	El sistema debe permitir al Superusuario borrar cualquier comunidad.
RF-07	El sistema debe permitir al Superusuario ver una lista de comunidades totales.
RF-08	El sistema debe permitir al Superusuario editar cualquier comunidad.
RF-09	El sistema debe permitir al Superusuario crear reservas.
RF-10	El sistema debe permitir al Superusuario borrar reservas.
RF-11	El sistema debe permitir al Superusuario ver una lista de reservas totales.
RF-12	El sistema debe permitir al Superusuario editar reservas.
RF-13	El sistema debe permitir al Superusuario crear espacios compartidos.
RF-14	El sistema debe permitir al Superusuario borrar espacios compartidos.
RF-15	El sistema debe permitir al Superusuario ver espacios compartidos.
RF-16	El sistema debe permitir al Superusuario editar espacios compartidos.
RF-17	El sistema debe permitir al Superusuario crear incidencias.
RF-18	El sistema debe permitir al Superusuario borrar incidencias.
RF-19	El sistema debe permitir al Superusuario ver incidencias.
RF-20	El sistema debe permitir al Superusuario editar incidencias.
RF-21	El sistema debe permitir al Administrador ver una lista de los usuarios de sus comunidades.
RF-22	El sistema debe permitir al Administrador crear espacios compartidos en sus comunidades.
RF-23	El sistema debe permitir al Administrador borrar espacios compartidos en sus comunidades.
RF-24	El sistema debe permitir al Administrador ver espacios compartidos en sus comunidades.
RF-25	El sistema debe permitir al Administrador editar espacios compartidos en sus comunidades.
RF-26	El sistema debe permitir al Administrador crear reservas en sus comunidades.
RF-27	El sistema debe permitir al Administrador borrar reservas en sus comunidades.
RF-28	El sistema debe permitir al Administrador ver reservas en sus comunidades.
RF-29	El sistema debe permitir al Administrador editar reservas en sus comunidades.
RF-30	El sistema debe permitir al Administrador crear incidencias en sus comunidades.
RF-31	El sistema debe permitir al Administrador borrar incidencias en sus comunidades.
RF-32	El sistema debe permitir al Administrador ver incidencias en sus comunidades.
RF-33	El sistema debe permitir al Administrador editar incidencias en sus comunidades.
RF-34	El sistema debe permitir al Usuario ver una lista de los espacios compartidos de las comunidades a las que pertenecen.

RF-35	El sistema debe permitir al Usuario crear reservas para los espacios compartidos de las comunidades a las que pertenecen.
RF-36	El sistema debe permitir al Usuario borrar sus reservas.
RF-37	El sistema debe permitir al Usuario modificar sus reservas.
RF-38	El sistema debe permitir al Usuario crear incidencias de los espacios compartidos a los que pueda acceder.
RF-39	El sistema debe permitir el registro de usuarios.
RF-40	El sistema debe permitir al Superusuario ver todas las incidencias.
RF-41	El sistema debe permitir al Administrador ver todas las incidencias de las comunidades que maneje.
RF-42	El sistema debe permitir al Usuario ver las incidencias creadas por él mismo.

Tabla 4.1: Tabla de requisitos funcionales

4.3. Requisitos no funcionales

RN-01	El sistema debe cumplir unos mínimos de seguridad.
RN-02	El sistema debe ser fácil de utilizar.
RN-03	El sistema debe tener una interfaz de fácil aprendizaje, habilitando su uso para personas sin asiduidad a las nuevas tecnologías.
RN-03	El sistema debe ser fácilmente escalable.
RN-04	El sistema debe estar operativo la gran mayoría del tiempo.
RN-05	El sistema debe cumplir la Ley Orgánica 7/2021, 26 de mayo.
RN-06	El sistema debe cumplir el Reglamento (UE) 2016/679 del Parlamento Europeo, 27 de abril de 2016.
RN-07	El sistema debe tener un diseño responsivo.
RN-08	El sistema debe poder ser ejecutado y tener soporte en las versiones más modernas de los navegadores más populares.
RN-09	El sistema debe tener buen rendimiento.
RN-10	El sistema se debe poder migrar con facilidad.
RN-11	El sistema debe tener un historial de versiones.

Tabla 4.2: Tabla de requisitos no funcionales

4.4. Casos de uso

El diagrama de casos de uso presentado en la figura 4.1 es una representación gráfica de las interacciones entre los diferentes actores (Vecino, Administrador y Superusuario) y el

sistema. En el diagrama, cada actor está asociado a diferentes casos de uso que describen las acciones que pueden realizar dentro del sistema. Los Vecinos, que representan a los usuarios finales del sistema, tienen capacidades como gestionar sus reservas e incidencias así como de ver las comunidades y espacios a las que pertenecen. Estos casos de uso aseguran que los Vecinos pueden gestionar su interacción con los recursos compartidos y reportar cualquier problema de manera autónoma.

El diagrama de casos de uso actual es bastante completo pero está sintetizado, sobre todo en la parte de *Gestión de reservas* y *Gestión de incidencias*, ya que de otra forma se haría demasiado grande. Cubre la mayoría de las interacciones esperadas, sin embargo, si tuviese que ser más riguroso se haría un diagrama para cada actor.

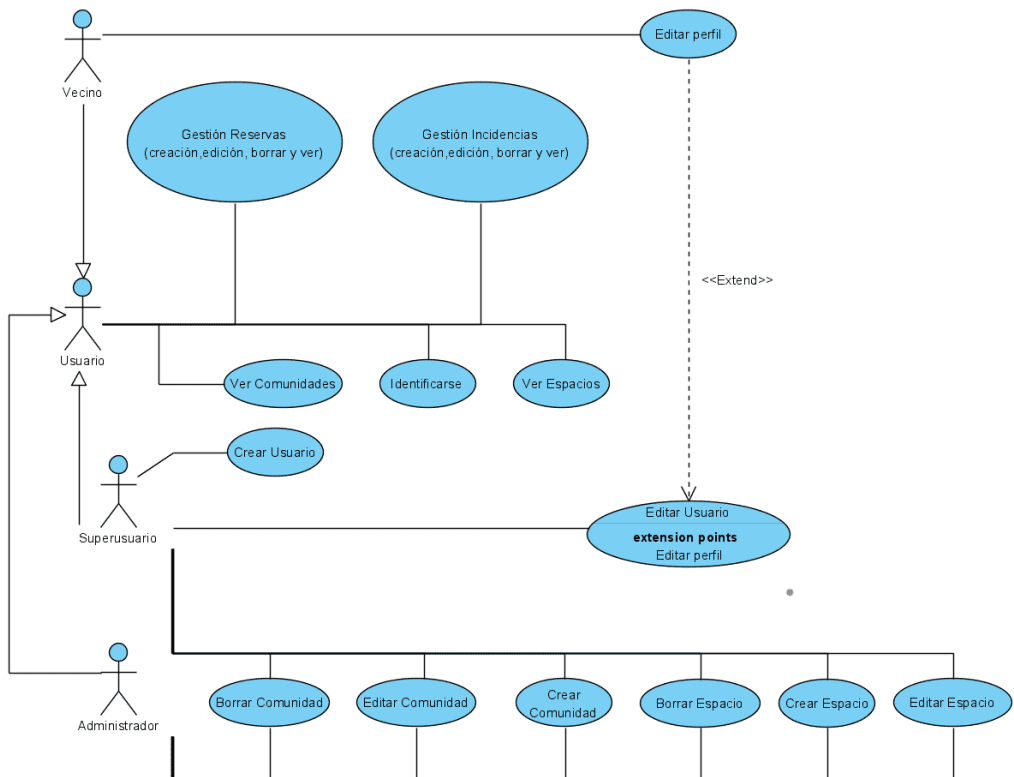


Figura 4.1: Diagrama de Casos de uso

Obviamente, es un diagrama resumido y en la implementación real cada tipo de usuario solo puede acceder, editar y borrar a lo que debería tener permiso, por ejemplo, un vecino puede *Ver comunidades*, pero solo podrá ver las suyas, así como puede *Borrar reservas*, pero solo puede hacerlo de sus reservas. También, se representa una herencia a un actor Usuario que no existe como tal, sino que es una forma de plasmar que todos los usuarios de la aplicación realizan esos casos de uso.

4.5. Descripciones de casos de uso

Para acompañar el diagrama anterior, se han realizado descripciones para cada uno de los casos de uso indicando los flujos de información, los requisitos y quién son los actores que se pueden ver envueltos en los casos de uso.

4.5.1. Identificarse

UC-1	Identificarse
Precondiciones	El usuario debe estar registrado en el sistema
Actores principales	Usuario
Flujo principal	<ol style="list-style-type: none">1. El usuario solicita identificarse2. El sistema solicita las credenciales del usuario3. El usuario proporciona las credenciales4. El sistema verifica las credenciales en la base de datos5. El sistema autentica al usuario6. El sistema notifica al usuario el resultado de la autenticación y redirige al usuario a la página principal
Flujos alternativos	<ol style="list-style-type: none">4a. Si las credenciales son incorrectas, el sistema notifica con un mensaje de error

Tabla 4.3: Descripción de caso de uso - Identificarse

4.5.2. Editar perfil

UC-2	Editar perfil
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Vecino o Superusuario
Flujo principal	<ol style="list-style-type: none">1. El usuario solicita editar el perfil de usuario2. El sistema muestra el formulario de edición del perfil3. El usuario realiza los cambios necesarios4. El sistema valida los cambios5. El sistema actualiza la información del perfil en la base de datos6. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none">1a. Si el usuario no tiene permisos sobre el perfil, el sistema muestra un mensaje de error4a. Si los cambios no son válidos, el sistema muestra un mensaje de error

Tabla 4.4: Descripción de caso de uso - Editar perfil

4.5.3. Crear Usuario

UC-3	Crear Usuario
Precondiciones	El superusuario debe estar autenticado en el sistema
Actores principales	Superusuario
Flujo principal	<ol style="list-style-type: none">1. El superusuario solicita crear un usuario2. El sistema solicita la información del nuevo usuario3. El superusuario proporciona la información del usuario4. El sistema valida la información proporcionada5. El sistema crea la cuenta de usuario6. El sistema notifica al superusuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none">4a. Si la información proporcionada no es válida, el sistema muestra un mensaje de error

Tabla 4.5: Descripción de caso de uso - Crear Usuario

4.5.4. Gestión Reservas

UC-4	Gestión Reservas
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Vecino, Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none">1. El vecino solicita gestionar una reserva2. El sistema muestra las opciones de gestión3. El vecino selecciona la acción deseada y aporta los datos necesarios4. El sistema procesa la solicitud y si es válida actualiza la base de datos5. El sistema notifica al vecino el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none">4a. Si la solicitud no es válida, el sistema muestra un mensaje de error4b. Si la actualización de la base de datos es inválida, el sistema muestra un mensaje de error

Tabla 4.6: Descripción de caso de uso - Gestión Reservas

4.5.5. Gestión Incidencias

UC-5	Gestión Incidencias
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Vecino, Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none">1. El vecino solicita gestionar una incidencia2. El sistema muestra las opciones de gestión3. El vecino selecciona la acción deseada y aporta los datos necesarios4. El sistema procesa la solicitud y si es válida actualiza la base de datos5. El sistema notifica al vecino el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none">4a. Si la solicitud no es válida, el sistema muestra un mensaje de error4b. Si la actualización de la base de datos es inválida, el sistema muestra un mensaje de error

Tabla 4.7: Descripción de caso de uso - Gestión Incidencias

4.5.6. Ver Comunidades

UC-6	Ver Comunidades
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Usuario
Flujo principal	<ol style="list-style-type: none">1. El usuario solicita ver las comunidades sobre las que tiene permisos2. El sistema recupera la lista de comunidades de la base de datos3. El sistema muestra las comunidades al usuario4. El usuario visualiza la información de las comunidades
Flujos alternativos	<ol style="list-style-type: none">1a. Si no hay comunidades disponibles, el sistema muestra un mensaje de error

Tabla 4.8: Descripción de caso de uso - Ver Comunidades

4.5.7. Ver Espacios

UC-7	Ver Espacios
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Usuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita ver los espacios sobre los que tiene permisos 2. El sistema recupera la lista de espacios de la base de datos 3. El sistema muestra los espacios al usuario 4. El usuario visualiza la información de los espacios
Flujos alternativos	1a. Si no hay espacios disponibles, el sistema muestra un mensaje de error

Tabla 4.9: Descripción de caso de uso - Ver Espacios

4.5.8. Crear Comunidad

UC-8	Crear Comunidad
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita crear un Espacio 2. El sistema solicita la información de la nueva Comunidad 3. El usuario proporciona los datos de la nueva Comunidad 4. El sistema valida la información proporcionada 5. El sistema crea la Comunidad en la base de datos 6. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 4a. Si la información proporcionada no es válida, el sistema muestra un mensaje de error 5a. Si no se puede guardar la información en la base de datos, el sistema muestra un mensaje de error

Tabla 4.10: Descripción de caso de uso - Crear Comunidad

4.5.9. Borrar Comunidad

UC-9	Borrar Comunidad
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita borrar una Comunidad 2. El sistema solicita confirmación para borrar la Comunidad 3. El usuario confirma la eliminación 4. El sistema elimina la Comunidad de la base de datos 5. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 1a. Si no tiene permiso sobre esa comunidad, el sistema muestra un mensaje de error 4a. Si no se puede borrar la Comunidad de la base de datos, el sistema muestra un mensaje de error

Tabla 4.11: Descripción de caso de uso - Borrar Comunidad

4.5.10. Editar Comunidad

UC-10	Editar Comunidad
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita editar una Comunidad 2. El sistema muestra el formulario de edición de Comunidad 3. El usuario realiza los cambios necesarios 4. El sistema valida los cambios 5. El sistema actualiza la información de la Comunidad en la base de datos 6. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 1a. Si el usuario no tiene permisos sobre la Comunidad, el sistema muestra un mensaje de error 4a. Si los cambios no son válidos, el sistema muestra un mensaje de error 5a. Si no se puede editar la Comunidad de la base de datos, el sistema muestra un mensaje de error

Tabla 4.12: Descripción de caso de uso - Editar Comunidad

4.5.11. Crear Espacio

UC-11	Crear Espacio
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El superusuario solicita crear un Espacio 2. El sistema solicita la información de el nuevo Espacio 3. El superusuario proporciona los datos de el nuevo Espacio 4. El sistema valida la información proporcionada 5. El sistema crea el Espacio en la base de datos 6. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 4a. Si la información proporcionada no es válida, el sistema muestra un mensaje de error 5a. Si no se puede guardar la información en la base de datos, el sistema muestra un mensaje de error

Tabla 4.13: Descripción de caso de uso - Crear Espacio

4.5.12. Borrar Espacio

UC-12	Borrar Espacio
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita borrar un Espacio 2. El sistema solicita confirmación para borrar el Espacio 3. El usuario confirma la eliminación 4. El sistema elimina el Espacio de la base de datos 5. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 1a. Si no tiene permiso sobre esa comunidad, el sistema muestra un mensaje de error 4a. Si no se puede borrar el Espacio de la base de datos, el sistema muestra un mensaje de error

Tabla 4.14: Descripción de caso de uso - Borrar Espacio

4.5.13. Editar Espacio

UC-13	Editar Espacio
Precondiciones	El usuario debe estar registrado y autenticado en el sistema
Actores principales	Administrador o Superusuario
Flujo principal	<ol style="list-style-type: none"> 1. El usuario solicita editar un Espacio 2. El sistema muestra el formulario de edición de Espacio 3. El usuario realiza los cambios necesarios 4. El sistema valida los cambios 5. El sistema actualiza la información de el Espacio en la base de datos 6. El sistema notifica al usuario el resultado de la operación
Flujos alternativos	<ol style="list-style-type: none"> 1a. Si el usuario no tiene permisos sobre el Espacio, el sistema muestra un mensaje de error 4a. Si los cambios no son válidos, el sistema muestra un mensaje de error 5a. Si no se puede editar el Espacio de la base de datos, el sistema muestra un mensaje de error

Tabla 4.15: Descripción de caso de uso - Editar Espacio

4.6. Diagrama inicial de clases

Para comprender un poco el dominio de la aplicación y empezar a organizar como iba a estar distribuida en clases de dominio, se decidió hacer un *Diagrama de clases iniciales*. El diagrama presentado en la figura 4.2 es una representación detallada de la estructura del sistema, mostrando las principales entidades, tipos de datos y enumerables asociados a tipos de entidades.

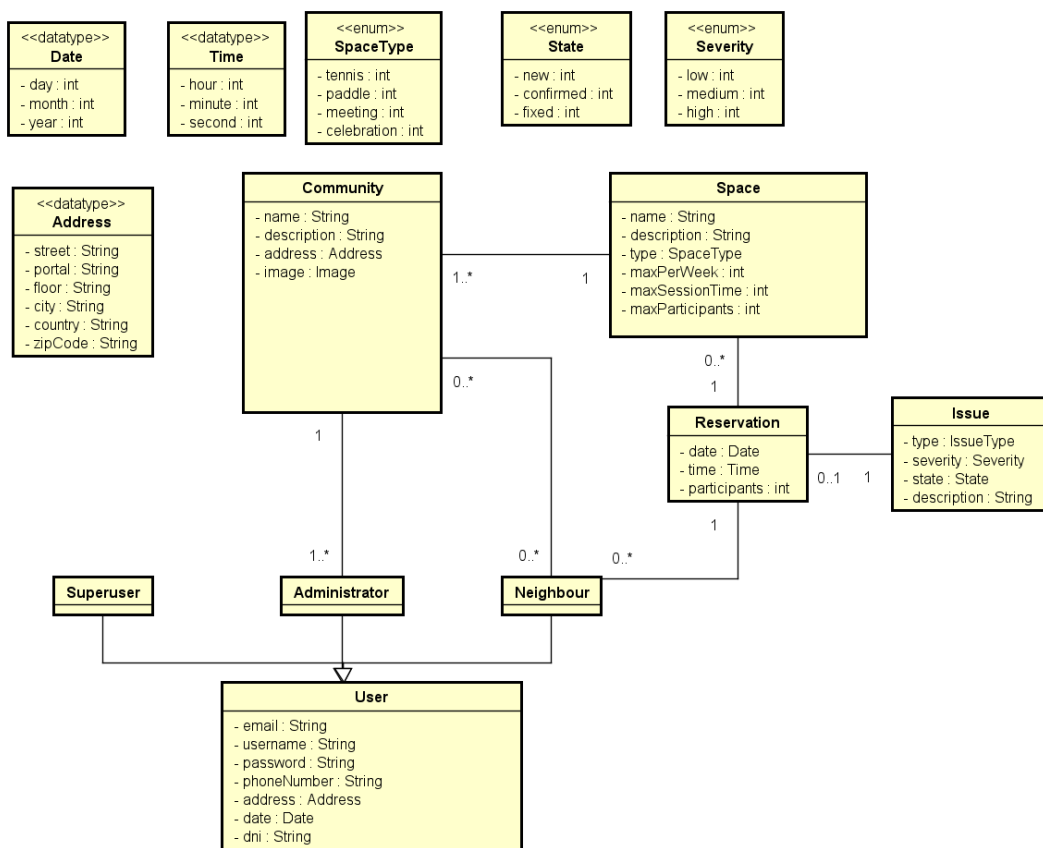


Figura 4.2: Diagrama de Clases iniciales

El diseño del diagrama de clases refleja una arquitectura bien organizada y modular del sistema, o cómo sería en un entorno monolítico. La jerarquía de usuarios permite una clara separación de roles y responsabilidades, facilitando la gestión de permisos y funcionalidades específicas para cada tipo de usuario. Las entidades más importantes son *Community* y *Space*, que además son muy ampliables, teniendo muchas más posibilidades de atributos que las que tienen ahora mismo.

La relación entre *Reservation* y *Space* muestra claramente que las reservas están directamente ligadas a los espacios, y la relación entre *Issue* y *Reservation* indica que las incidencias

están relacionadas con eventos específicos (reservas), lo cual es fundamental para el seguimiento y resolución de problemas. Se decidió implementar así la lógica de negocio para que una incidencia venga asociada a una reserva precisamente por el seguimiento. Aunque a la larga y ampliando la aplicación es posible que fuese mejor tener incidencias generales, para este proyecto el diseño se consideró que era mejor así.

Capítulo 5

Diseño

5.1. Microservicios o Monolito

Para empezar a entender la diferencia entre estas dos arquitecturas, hay que dar una pequeña definición y explicación de ambas y cuales son sus ventajas, ya que antes de elegir un diseño para una aplicación se debe estudiar si este se adecua bien a los requisitos de la misma.

5.1.1. Monolito

La **arquitectura monolítica** es un diseño de software que se implementa como una sola unidad unificada y que no depende de ningún otro servicio y se llama en referencia a un monolito (roca de gran tamaño) precisamente por esa razón, para visualizar que es un único elemento grande. Hasta hace no muchos años esta era la arquitectura por defecto de prácticamente todos los proyectos del sector y por esta razón a día de hoy tiene connotaciones de software legacy.

La arquitectura tiene todos los elementos contenidos en el mismo servicio y por esta razón es autocontenido, es decir, puede funcionar sin necesidad de ninguna dependencia ni de ningún servicio externo, esto significa que la lógica de negocio, de acceso a datos y las vistas están juntas, además de la base de datos. Esto obviamente significa que si hay que desplegar cambios hay que recompilar toda la aplicación (si fuese necesario compilar) o resubirla entera y que los pipelines de CI/CD serán más lentos y costosos. Los puntos fuertes y débiles de un monolito son los siguientes:

Ventajas

- **Desarrollo sencillo:** Trabajar en una aplicación monolítica es más sencillo por lo

general, ya que aunque es cierto que no hay una separación del frontend y el backend, es más fácil tener un contexto de todo el conjunto y entender mejor el proyecto.

- **Rendimiento:** Debido a estar todo junto no hay tiempos de respuesta mayores debidos a las peticiones y orquestación a otros servicios.
- **Testeo:** Es más fácil encontrar la fuente de errores y de tener un conjunto de tests bien probados, ya que realizar testing E2E es más fácil al estar junto.

Desventajas

- **Tendencia al caos:** Tener el proyecto entero bien estructurado y seguir unas buenas prácticas es más difícil al ser mucho más grande, requiere mayor esfuerzo y más organización
- **Tendencia al legacy:** Los monolitos tienden a quedarse atrás en las versiones de sus stack tecnológicos, ya que cuando un framework o lenguaje pasa a una versión superior suele haber marcados unos *breaking changes*, es decir, cambios que rompen la aplicación. Si lo pensamos, cambiar en todos los lugares donde se utilizase eso en una aplicación de un tamaño grande es mucho más complicado que si fuesen pequeños servicios.
- **Mal onboarding:** Si un programador nuevo entra al proyecto, tiene una barrera de entrada mucho mayor, ya que coger el contexto del proyecto es más complicado y acaba traducándose a más meses introductorios.
- **Mal escalado:** Los monolitos carecen de escalado horizontal y solo pueden crecer verticalmente (obviando técnicas como balanceadores de carga y similares), lo que hace que el coste de infraestructura y de servidores sea mucho mayor.
- **Acoplamiento:** Es muy habitual que el código de una aplicación de este estilo esté fuertemente acoplado y sea muy difícil reutilizarlo.

Los monolitos pueden resultar prácticos al principio de un proyecto para aliviar la complejidad inicial y empezar directamente a desarrollar y avanzando más rápido debido a tener menos estructura, sin embargo, los monolitos a la larga suelen acabar siendo muy complejos, difíciles de tratar y de programar, excesivamente complicados y dan problemas de rendimiento y de seguridad debido a las versiones como ya se ha comentado. Se estima que un 75 % [32] de los bancos tienen sistemas legacy y que un 60 % [33] de las empresas tienen problemas críticos intentando integrar sistemas legacy con nuevas tecnologías haciendo que se pierda rapidez, por lo que, que un proyecto se convierta en legacy se podría decir que es muy contraproducente y, por desgracia, muy común.

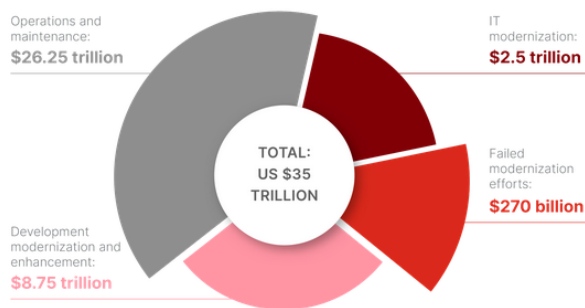


Figura 5.1: Gastos en modernización desde 2010

Por todo esto, creo que dedicar todos los esfuerzos posibles a que una aplicación nueva no quede desfasada rápidamente debería ser una prioridad para todo desarrollador y diseñador de software. Como se puede ver en la figura 5.1 el coste desde el 2010 en mantenimiento e intentos de modernización de proyectos en el sector es 35 billones (trillones americanos) de dolares, lo cual es una autentica barbaridad. Esto no significa que por fuerza los proyectos monolíticos siempre tengan que acabar así, ya que hay proyectos monolíticos que han sido muy exitosos como eBay, Uber, LinkedIn y Twitter, pero precisamente todos han acabado migrando a microservicios.

5.1.2. Microservicios

La **arquitectura basada en microservicios** en diseño de software se refiere a que una aplicación tiene sus partes divididas en servicios independientes que cada uno realiza ejecuta independiente al resto, aunque luego se comuniquen entre ellos a través de APIs e interfaces definidas para ello. Permite que cada servicio sea desarrollado, desplegado y escalado de forma independiente, ofreciendo una mayor flexibilidad, y agilidad en comparación con las arquitecturas monolíticas tradicionales, además de menos acoplamiento en general y más cohesión.

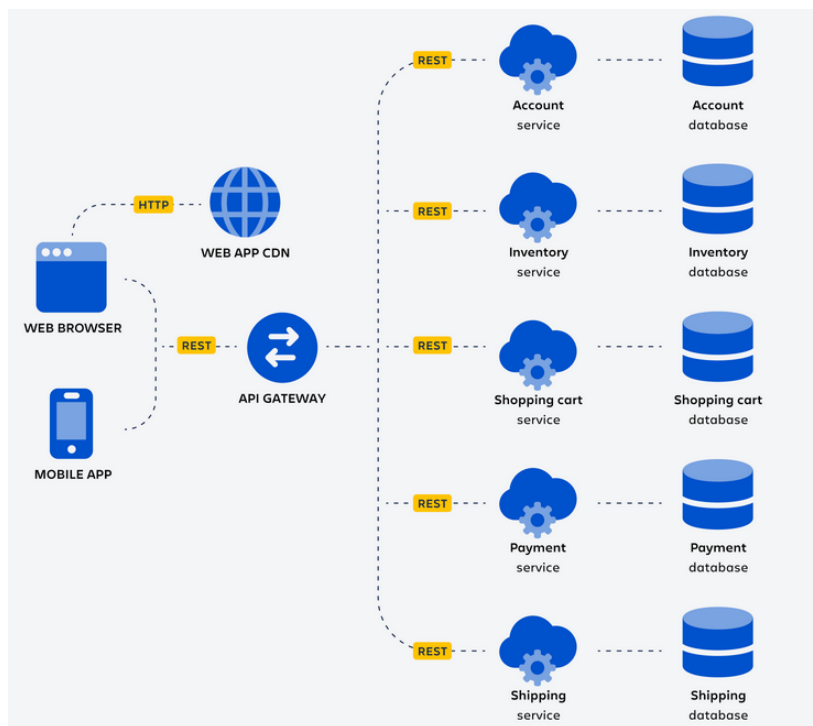


Figura 5.2: Ejemplo de microservicios de una tienda.

La principal ventaja de trabajar de esta forma es el escalado, ya que cada microservicio tiene unas necesidades distintas y algunos en concreto pueden tener más demanda por lo que pueden ser escalados sin tener que escalar el resto. Por ejemplo, si la aplicación consta de tres APIs y una de ellas recibe muchas peticiones, se podría ampliar el servidor que aloja esa API o incluso alojarlo en un nuevo servidor separado del resto y que todo funcionase igual que antes. Permite además a los equipos de desarrollo trabajar en diferentes servicios, lo que acelera el ciclo iterativo de desarrollo. La independencia de cada microservicio también facilita la adopción de diferentes tecnologías y lenguajes de programación según las necesidades específicas de cada uno, pudiendo tener cada uno en un framework distinto y que funcione perfectamente.

Ventajas

- **Escalado:** Los microservicios pueden ser escalados de manera independiente, permitiendo una optimización de recursos más eficiente y habilitando el escalado horizontal. Es mucho más fácil escalar una aplicación poniendo un microservicio en cada servidor que contratar un servidor muy potente para todo.
- **Despliegue Rápido:** Se puede desplegar y compilar servicios de forma independiente, lo que reduce el tiempo de inactividad y acelera la entrega de nuevas funcionalidades, además de tener un CI/CD más barato.

- **Flexibilidad Tecnológica:** Cada microservicio puede ser desarrollado con diferentes tecnologías y lenguajes de programación, optimizando cada componente según sus necesidades específicas.
- **Robustez:** Que un microservicio no crítico falle no hace que toda la aplicación deje de funcionar, al contrario del monolito.
- **Mantenibilidad:** Los servicios más pequeños y enfocados son más fáciles de mantener y actualizar, reduciendo la complejidad. También es fácil tener test unitarios y de integración preparados y con buena cobertura, ya que es menos código que cubrir.
- **Tecnologías y soluciones:** Habilita la convivencia de varias tecnologías si fuese necesario y abre la posibilidad a que microservicios distintos tengan soluciones más personalizadas. Por ejemplo, puedo tener un microservicio que necesite una base de datos relacional y otro que le venga mejor NoSQL y no habría ningún problema.

Desventajas

- **Complejidad:** La comunicación entre microservicios puede ser compleja y requiere una gestión eficiente de las APIs y los protocolos de comunicación.
- **Gestión de datos y fuente de verdad:** Mantener la consistencia de datos entre múltiples servicios puede ser complicado, por lo que hay que hacer un buen planteamiento de la base de datos y de los modelos previamente.
- **Curva de aprendizaje:** La adopción de una arquitectura de microservicios tiene una curva de aprendizaje grande.

Los microservicios son muy buenos en entornos donde la mantenibilidad, la escalabilidad y la entrega continua son esenciales. Permiten a las empresas responder a los cambios del mercado y a las demandas de los usuarios, implementando nuevas funcionalidades con mayor rapidez y eficiencia. También, favorece al trabajo en grupo y en equipos, haciendo que los desarrolladores se conviertan en expertos de un trozo de la aplicación, que es mejor que tener un vago contexto de la aplicación en su conjunto.

Por estas razones, la gran mayoría de proyectos modernos han migrado o están considerando migrar a arquitecturas de microservicios para aprovechar sus beneficios y mantener la competitividad en un entorno tecnológico en constante evolución. Ejemplos exitosos de esta transición incluyen empresas como Netflix [34], Amazon y Spotify, que han logrado mejorar su eficiencia operativa a través de la adopción de microservicios. Es muy interesante el caso de Netflix, pues tiene muchos blogs y recursos explicando como fue su migración, los problemas y retos enfrentado y como mejoró muy notablemente su servicio, compartiendo incluso su arquitectura actual que se puede ver en la figura 5.3.

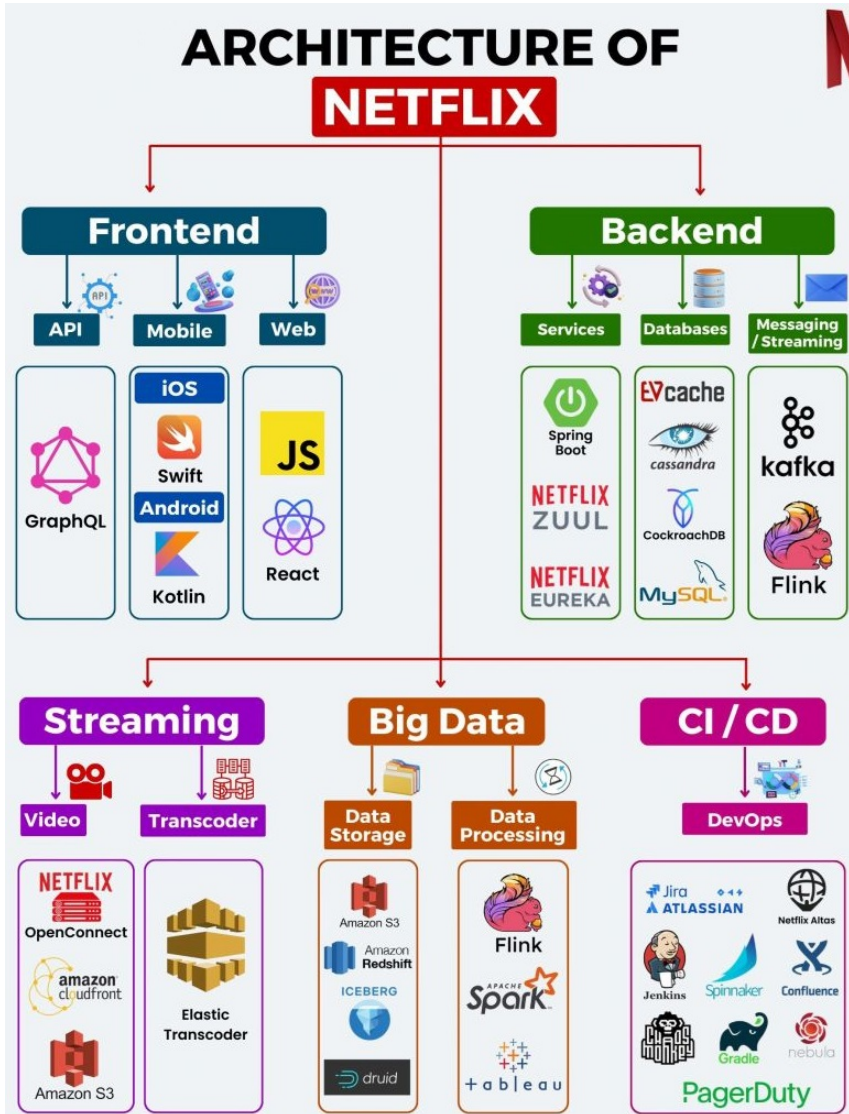


Figura 5.3: Arquitectura y tecnologías de Netflix.

5.1.3. El por qué de los microservicios

Cuando se empezó el proyecto y cada día más, había una tendencia hacia la arquitectura de microservicios por su capacidad para manejar la complejidad y escalar aplicaciones de manera más eficiente. A diferencia de la arquitectura monolítica, los microservicios permiten desarrollar, desplegar y escalar servicios de manera mucho más organizada y fácil de programar, además de que tienen mucha más proyección a la larga y la posibilidad de rehacer microservicios enteros sin tener que rehacer todo el proyecto. Todo esto mejora la agilidad y

la capacidad, además de que cada equipo puede trabajar de forma autónoma en diferentes microservicios, haciendo desarrollos más enfocados y una mejor jerarquía en el proyecto.

La robustez es otro beneficio crucial, ya que un fallo en un microservicio no compromete la disponibilidad de toda la aplicación. Esto mejora fiabilidad del sistema y en consecuencia la percepción del producto y de la marca, además de permitir una recuperación rápida y eficiente de errores. También, los microservicios facilitan el mantenimiento y la actualización de las tecnologías sin tener que rehacer o cambiar gran cantidad de implementaciones de código

En términos de despliegue, los microservicios permiten una integración y entrega continuas (CI/CD) mucho más eficiente, y sabiendo que es algo muy costoso para una empresa y un producto, es muy importante. Esto acelera los ciclos de lanzamiento y mejora la rapidez, que si se junta con una mayor testabilidad da unas buenas prácticas y una seguridad sobre el proyecto y los despliegues a otro nivel.

Por lo general, siempre voy a preferir los microservicios porque además de que son potencialmente más escalables, se percibe que lo raro en los tiempos que corren es tener aplicaciones monolíticas y que todos los proyectos serios y grandes son con microservicios en menor o mayor medida. Demostrar todo esto y la convivencia entre tecnologías ha sido la razón por la cual se decidió usar tantas tecnologías tan diversas y ver que se puede hacer una comunicación entre ellas si se hace de la forma correcta, ya que al fin y al cabo son entes independientes que se mandan mensajes entre sí.

5.2. Arquitectura y estructura del proyecto

5.2.1. Modelo Vista Controlador

El patrón de arquitectura **MVC** (Modelo-Vista-Controlador) es muy conocido y utilizado en el desarrollo software debido a sus ventajas significativas en términos de escalabilidad y mantenibilidad. Este enfoque organiza el código en tres componentes, el modelo (que maneja la lógica de datos y de negocio), la vista (que se encarga de la presentación), y el controlador, que actúa como intermediario entre el modelo y la vista. Esta separación de responsabilidades facilita la gestión y actualización del código, reduciendo la complejidad y permitiendo una mayor flexibilidad y testabilidad.

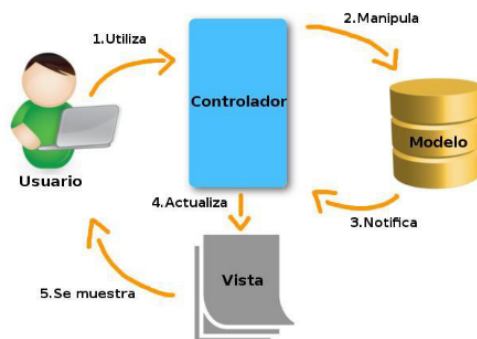


Figura 5.4: Esquema básico de MVC

Una de las mayores ventajas de MVC es que permite una mejor organización del código gracias a su estructura dividida, lo que resulta en aplicaciones más fáciles de leer y con mayor trazabilidad. La separación de responsabilidades entre el modelo, la vista y el controlador asegura que los cambios en una parte de la aplicación no afecten directamente a las otras, promoviendo un diseño más modular y menos acoplado.

Ventajas

- **Buena escalabilidad y mantenibilidad:** La separación de responsabilidades facilita el mantenimiento y la escalabilidad de la aplicación.
- **Más flexible y menos acoplado:** Permite realizar cambios en un componente sin afectar a los demás.
- **Más testable:** Cada componente se puede probar de manera independiente, mejorando la calidad del software.
- **Mejor organizado:** El código está más estructurado y es más fácil de entender y mantener.
- **Separación de responsabilidades:** Cada componente (M-V-C) tiene una responsabilidad única (principio SOLID).

Desventajas

- **La separación no es total:** Todavía puede haber acoplamiento entre el modelo, la vista y el controlador.
- **Código legacy:** Sin una organización adecuada, el código puede volverse desordenado y difícil de gestionar.
- **Sigue habiendo acoplamiento:** Aunque reducido, el acoplamiento no se elimina completamente.

- **Requiere disciplina:** Mantener la separación de responsabilidades requiere un esfuerzo constante y disciplina en el desarrollo.

Sin embargo, MVC no es una solución perfecta y presenta algunos problemas, ya que la separación entre los componentes no es absoluta, lo que suele llevar a cierto acoplamiento entre el modelo, la vista y el controlador. Además, si no se siguen estrictamente las buenas prácticas de diseño, es fácil que el código vuelva a una situación de desorden similar a la de las arquitecturas monolíticas. Para proyectos de gran tamaño, existen alternativas que pueden ofrecer mejores resultados, como el patrón Modelo-Vista-Adaptador (MVA), que soluciona algunos de los problemas inherentes a MVC.

A pesar de sus contras, con una organización adecuada y siguiendo reglas estrictas, el patrón MVC puede ser una mejora considerable para el desarrollo de software con respecto a una aplicación tradicional. Es crucial mantener una disciplina en la separación de responsabilidades para evitar que el código se vuelva inmanejable e ilegible con el tiempo.

Este patrón se utilizó en todo el frontend en Angular, ya que el propio framework de alguna forma quiere que lo utilices al ser sus componentes pequeños MVC separados.

5.2.2. DDD

El **DDD** (Domain Driven Development) es uno de los patrones de diseño más conocidos y estandarizados en el sector, se presentó por Eric Evans en su libro “Domain-Driven Design — Tackling Complexity in the Heart of Software.”^{en} 2004 y supuso un cambio de mentalidad. El DDD propone centrarse sistemáticamente en el dominio de la aplicación, ya que esta corriente del desarrollo piensa que si no se aborda de esta forma, dará igual que la infraestructura o la tecnología elegida sean buenas.

La mentalidad de DDD radica en que el ingeniero o arquitecto de software que desarrolla la aplicación no es un **experto del dominio**, este es un término que se repite mucho en referencia a DDD, se menciona en todos los libros al respecto. Significa que si estamos desarrollando un proyecto sobre transporte de mercancías el programador lo más probable es que no sea un experto en la materia, pero la clave es la colaboración entre los expertos de dominio y el equipo técnico. Esta colaboración asegura que el software construido refleje con precisión las reglas de negocio, lo que es fundamental para el éxito del proyecto y la satisfacción del cliente.

Otro de los conceptos centrales es el uso del **ubiquitous language** (lenguaje ubicuo), que se refiere a un lenguaje común y compartido por los desarrolladores y los expertos del dominio para mayor entendimiento, ya que el experto no estará habituado al lenguaje técnico ni el programador al lenguaje de negocio. El lenguaje ubicuo se refleja en el código, la documentación y el día a día del desarrollo, por lo que es importante definirlo bien [35].

Además de lo anterior, en DDD se suelen definir los siguientes patrones fundamentales para ordenar bien el dominio y que haya cohesión y bajo acoplamiento:

- **Entidades:** Son objetos que tienen una identidad definida que se mantiene constante a lo largo del tiempo. Por ejemplo, en el sistema de gestión de usuarios, un "Usuario" sería una entidad, ya que cada usuario tiene una identidad única que lo distingue de otros usuarios y sus propios atributos. Las entidades se suelen implementar a través de clases y casi todos los lenguajes y frameworks modernos tienen su propia abstracción para trabajar de forma más fácil [36].
- **Agregados:** Son grupos de objetos que se tratan como una sola unidad compuesta bien cohesionada. Un agregado tiene una **aggregate root**, que es la entidad principal, y todos los demás objetos están relacionados a través de esta raíz. La aggregate root garantiza la consistencia del agregado. Por ejemplo, en el sistema de usuarios, un "Usuario" podría ser un agregado con "Dirección" como entidad relacionada [37].

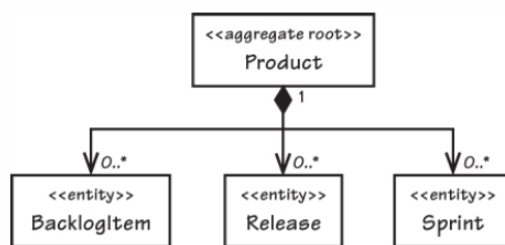


Figura 5.5: Ejemplo de agregado de SCRUM

Además de las entidades y agregados, DDD introduce la idea de **bounded contexts** (Contextos Delimitados) [38], que son secciones del dominio que tienen un contexto claro definido, en la figura 5.6 se puede ver un ejemplo que separa dos áreas de contexto de una aplicación. Esto ayuda a gestionar la complejidad dividiendo el sistema en partes más manejables, cada una con su propio modelo y lógica de negocio.

Los contextos delimitados facilitan la escalabilidad del proyecto, permitiendo que diferentes equipos trabajen en paralelo sin interferencias en contextos distintos. Dentro de cada contexto, el dominio y el lenguaje ubicuo es específico y preciso para el mismo.

En el proyecto no ha sido necesario utilizar los contextos delimitados ni los agregados, pues la propia separación de microservicios hace que haya pocas entidades en cada servicio y los agregados no tengan del todo sentido.

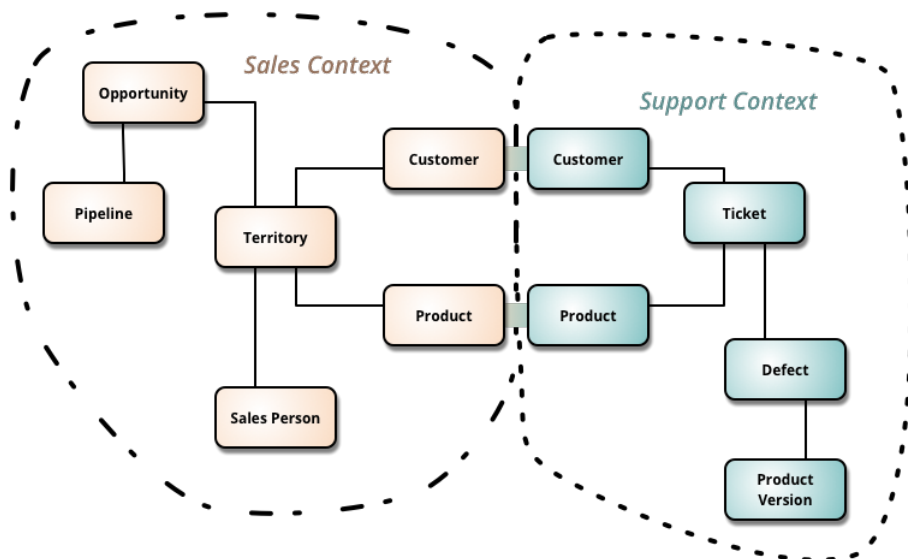


Figura 5.6: Aplicación con dos *bounded context*

5.2.3. Arquitecturas por capas

5.2.3.1. Arquitectura limpia

Presentada por el conocido Robert C. Martin (Uncle Bob) en su libro *Clean Architecture*.^{en} 2017, esta arquitectura proporciona unas bases de diseño que permiten conseguir una alta cohesión y un bajo acoplamiento entre los sistemas y el código del proyecto, haciendo así que la forma de escalar sea mucho más orgánica y que la mantenibilidad sea muy buena, siendo clave sobre todo en proyectos grandes y que se esperan utilizar durante un largo periodo de tiempo. Las principales características de la arquitectura limpia son la organización de la aplicación por capas y la regla de dependencia.

La Arquitectura Limpia propone tener una capa de Interfaces Externas, otra capa de Infraestructura donde estarán los controladores, gateways y puntos de entrada, una capa de Casos de Uso y, por último, una capa de Entidades. Esta separación clara define en qué parte del código debe ir cada elemento y, gracias a la regla de dependencia, la aplicación está mucho más desacoplada y las dependencias están jerarquizadas, haciendo que sea muy mantenible y tolerante a cambios. La regla de dependencia establece que las capas solo deben tener acceso a la capa inmediatamente inferior, es decir, la capa de Infraestructura solo tendría acceso a la de Aplicación.

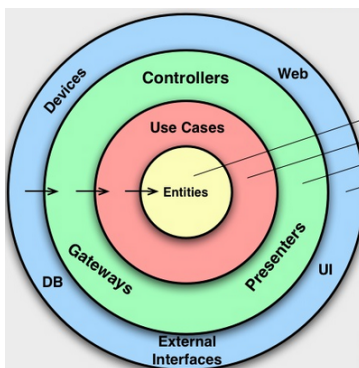


Figura 5.7: Esquema de la arquitectura limpia

Cuando se conoce bien esta arquitectura y se está habituado a trabajar con ella, proporciona numerosos beneficios, incluyendo un alto grado de modularidad que permite que diferentes equipos trabajen en distintas partes del código sin colisiones. Facilita la comunicación entre servicios de la capa de aplicación, permite nuevos desarrollos con poco impacto en otras implementaciones y reduce los costos de mantenimiento de una arquitectura tradicional. Además, estandariza la forma de trabajar de un equipo o de una empresa, lo que facilita la transición de los desarrolladores entre proyectos sin un gran impacto en el tiempo de adaptación. Muchos frameworks de backend y API ya vienen preparados para trabajar con este tipo de arquitecturas, haciendo que incluso cambiar de lenguaje y framework sea más sencillo.

5.2.4. Arquitectura hexagonal

La Arquitectura Hexagonal, también conocida como Arquitectura de Puertos y Adaptadores, fue propuesta por Alistair Cockburn [39]. Esta arquitectura se enfoca en mejorar la testabilidad del software mediante la implementación del principio SOLID de Inversión de Dependencias. La principal diferencia entre la Arquitectura Hexagonal y otras arquitecturas de capas es la manera en que se maneja la dependencia entre módulos.

El principio de Inversión de Dependencias sugiere que los módulos con lógica compleja no deberían depender directamente de módulos de utilidad. Para lograr esto, se introducen abstracciones de los servicios de utilidad en forma de interfaces. En el contexto de la Arquitectura Hexagonal, esto significa que los servicios de utilidad, como el acceso a datos, se definen como interfaces que pueden ser implementadas y cambiadas sin afectar la lógica del negocio. Esta abstracción permite que las implementaciones sean intercambiables y, más importante, que se puedan mockear en los tests. Esto facilita enormemente la prueba de unidades y la prueba de integración, ya que no es necesario utilizar una base de datos real, sino que se pueden usar mocks que simulen el comportamiento de la base de datos.

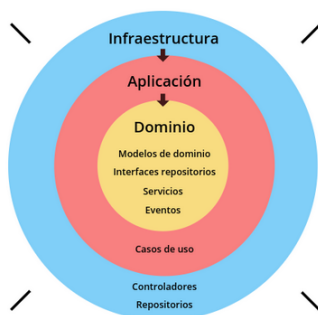


Figura 5.8: Esquema de la arquitectura hexagonal

Como se puede observar en la figura 5.8 hay pequeñas diferencias con respecto a la arquitectura limpia, pero los fundamentos y las ventajas son muy parecidas. Para llevar a cabo todos los beneficios comentados sobre la inversión de dependencias viene muy bien utilizar la inyección de dependencias de la forma que te lo permita el framework en cuestión.

5.2.5. Comparación y Elección de la Arquitectura

Aunque tanto la Arquitectura Limpia como la Hexagonal comparten muchos beneficios, incluyendo la alta mantenibilidad y escalabilidad, se eligió la Arquitectura Hexagonal debido a su enfoque en la testabilidad. En la situación actual del sector, donde el testing real en las empresas es limitado, la posibilidad de facilitar y mejorar las pruebas fue un factor decisivo. La Arquitectura Hexagonal, mediante la Inversión de Dependencias, permite crear un entorno de pruebas más controlado y eficiente, agilizando la ejecución de tests y asegurando una mayor robustez en el software.

De esta manera, mis tres backends están divididos por capas y siguiendo esta arquitectura sumado a unos patrones de diseño que comentaré en la siguiente sección. La estructura básica ha sido utilizar los controladores como punto de entrada de la aplicación y patrones repositorio como acceso a datos ambos en la capa de Infraestructura, luego, en la capa de Aplicación hay servicios con la lógica de implementación y en Dominio están las entidades con la lógica de negocio.

5.3. Patrones de diseño utilizados

5.3.1. DTO

El **DTO**[40] o Data Transfer Object es un patrón muy conocido en el diseño de software, estos objetos se utilizan para transferencia de datos entre capas, servicios o procesos y únicamente llevan datos, es decir, los DTO no tienen que llevar la lógica de negocio ya que

esto es responsabilidad del dominio. Lo cómodo de este patrón es que te permite olvidarte de la serialización de los objetos de dominio y tenerlo separado, que en conjunto al siguiente patrón han ahorrado mucho tiempo de hacer validaciones.

Sus ventajas y utilidades son las siguientes [41]:

Ventajas

- **Reducción de llamadas:** Al agrupar datos en un único objeto, se reduce la cantidad de llamadas necesarias entre el cliente y el servidor.
- **Encapsulación:** Con DTO se tiene centralizada toda la lógica para serializar las entidades y también para validación como se verá en la sección 6.2.2.
- **Desacoplamiento de los modelos:** Permite que el modelo de dominio y la capa de presentación evolucionen de manera independiente.
- **Facilidad de testing:** Al no contener lógica de negocio, los DTO son fácilmente testables.

Desventajas

- **Código adicional:** La implementación de DTO introduce clases y código adicionales.

Para el proyecto se aplicaron los DTO en el backend como punto de entrada a los endpoints en los controllers, estos controllers esperan DTO y responden con DTOs y son los servicios de la capa de aplicación quien lo convierte a objetos de dominio.

5.3.2. Data Mapper

El patrón **Mapper** se utiliza junto con los DTO para convertir datos entre diferentes capas de una aplicación. Un mapper toma un objeto de una clase (una entidad) y lo convierte en un DTO, y viceversa. Este patrón es esencial para mantener el desacoplamiento, la limpieza del código y el principio de responsabilidad única de SOLID. Las ventajas y desventajas son muy similares que los DTO, ya que al fin y al cabo es otro pequeño patrón introducido para separar responsabilidades y reutilizar código:

Ventajas

- **Desacoplamiento:** Mantiene la lógica de mapeo fuera de la lógica de negocio, es decir, fuera de las entidades.
- **Reutilización:** Los mappers pueden ser reutilizados en diferentes partes de la aplicación.
- **Facilidad de testing:** Los mappers son fácilmente testables, igual que los DTO.

Desventajas

- **Complejidad adicional:** Implementar y mantener mappers puede añadir complejidad al proyecto.
- **Rendimiento:** Si está mal implementado puede dar problemas de rendimiento.

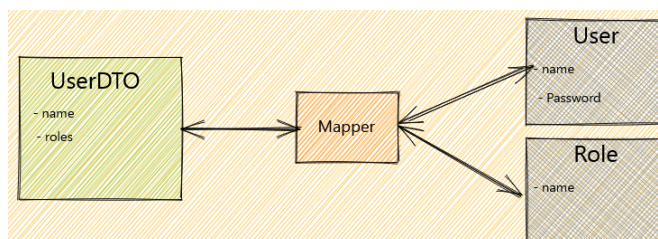


Figura 5.9: Esquema de los patrones DTO y Mapper

Utilizando este patrón en mis servicios ha sido muy sencillo el cambio entre DTO y dominio y la verdad que ha sido muy cómodo, si que es cierto que es un paso más y más código pero una vez está implementado es de agradecer, se puede ver un ejemplo sencillo de uso en la figura 5.9.

5.3.3. Repositorio

Repositorio es un patrón de diseño que proporciona una capa de abstracción entre la lógica de negocio y la capa de acceso a datos en una aplicación. Es especialmente útil para separar las responsabilidades del acceso a datos del resto de la aplicación, promoviendo la mantenibilidad, testabilidad y escalado del código. Con el se ha podido juntar todas las consultas y acceso a persistencia en un mismo sitio.

Además de la separación de responsabilidades, los métodos encapsulados en repositorios pueden ser reutilizados en diferentes partes de la aplicación, reduciendo la duplicación de código y mejorando la eficiencia del desarrollo. Uno de los grandes problemas de algunos proyectos mal estructurados es que se hacen consultas ad-hoc para una parte del código y luego se llega a la situación de que si cambia la estructura de la base de datos hay que cambiarlo en muchos sitios, pero con el patrón repositorio esto no ocurre.

Como comentamos anteriormente, es más fácil escribir pruebas unitarias y de integración gracias a la inversión de dependencias y a tener una abstracción en forma de interfaz de estos repositorios. Los repositorios pueden ser fácilmente mockeados, permitiendo probar la lógica de negocio sin necesidad de acceder a la base de datos real. En una aplicación tan pequeña como la mía puede no ser un problema, pero en productos grandes si testamos directamente contra una base de datos (aunque sea en local o en pre-producción) y hay muchos tests que

ejecutar, la comprobación de los mismos se puede alargar incluso a muchos minutos, haciendo que los despliegues sean muy tediosos y perdiendo agilidad.

Este uso de interfaces que venimos hablando no solo mejora la testabilidad, sino que si por ejemplo en el futuro hay que cambiar una implementación del repositorio utilizando SQL para un MySQL y se decide que ahora hay que trabajar con un ORM en postgres solo haya que cambiar un repositorio por otro sin preocuparse de nada ya que se llamaría de la misma forma gracias a la interfaz.

Pero este patrón también tiene algunos problemas debido a las malas prácticas, y es que un repositorio en estadios tempranos del proyecto o en proyectos pequeños es permisible que tengan muchos métodos, pero cuando el proyecto crece se acaban agrandando demasiado, por esto, la buena práctica aquí es crear la interfaz para que su contrato sea un CRUD básico y un patrón Critería o Especificación, lo cual discutiré en el capítulo de [Conclusiones y trabajo futuro](#) ya que en este caso no era tan necesario de implementar.

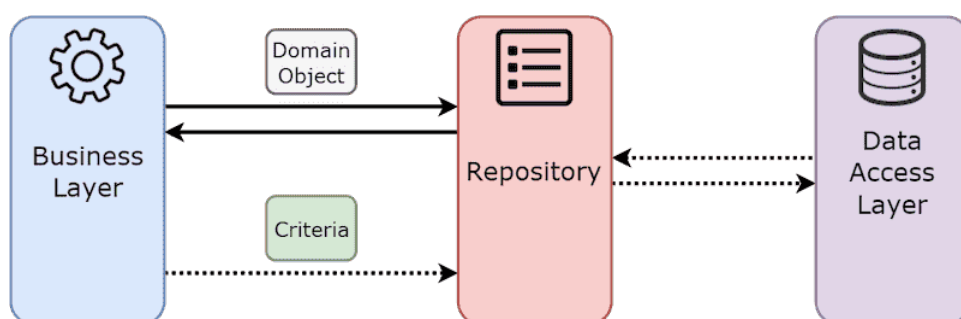


Figura 5.10: Funcionamiento general de Repository y Criteria.

Por esto, todo el backend está implementado con repositorio como acceso a datos y con la dependencia inyectada en los servicios de aplicación precisamente para que cuando se desarrolle el testing en un futuro se pueda mockear.

5.3.4. Singleton

El patrón Singleton es probablemente el patrón de diseño creacional más conocido, lo que hace es asegurar que una clase tenga una sola instancia y proporciona un punto global de acceso a esa instancia. Este patrón es ampliamente utilizado en situaciones donde es necesario controlar el acceso a recursos compartidos, como conexiones a bases de datos, gestores de configuración, o sistemas de logging.

Las ventajas pueden estar bien, ya que por ejemplo tener una conexión lista a la base de datos global es muy beneficioso, sin embargo, el patrón Singleton es también conocido por ser abusado, ya que hay muchas situaciones que se utiliza por pura rutina en lugares que simplemente no interesaría. Por ello, es un patrón muy polémico y que trae consigo un montón de problemas.

Estos problemas son sobre todo de acoplamiento no intencionado, ya que sin quererlo, se puede hacer que todas las clases dependientes del Singleton tengan problemas a la hora de cambiar este, creando una dependencia. También otro de los problemas es que los Singleton suelen ser malos para testing, ya que el estado global de la instancia hace que su estado persista entre los tests, por lo que si se usa testing donde no se debe, ese estado global podría hacer que los tests fallasen por no estar lo suficientemente aislados. Por último, hay que tener cuenta que este patrón crea objetos que van a estar instanciados durante mucho tiempo, y si se implementa mal podrían ir acumulando referencias con el tiempo hasta el punto de dar problemas de rendimiento.

En este caso, creo que estos posibles problemas no se van a materializar, ya que para lo que utilizo este patrón es para las conexiones a la base de datos en el backend y para los servicios en Angular, ya que los servicios en Angular son singleton por defecto y no se encontró ninguna referencia de que causen problemas en este aspecto. Pero por ello la intención fue la de no sobreutilizarlo y únicamente implementarlo para lo que está pensado.

5.4. Separación en microservicios

Para empezar la arquitectura de microservicios, tuve que separar el dominio original en servicios más pequeños y por lo tanto en contextos más pequeños. Partiendo del diagrama de clases iniciales de la sección 4.6, lo primero que se puede ver es que hay varios contextos que se pueden separar y son los de Comunidades, Usuarios y reservas. En realidad, se podría dividir solo entre Comunidades con reservas incluidas y Usuarios, pero para fines educativos decidí dividirlo en tres.

Lo primero en lo que hay que pensar para separar en un microservicio es si tiene sentido, y es que en el caso de Usuarios es un servicio que podría usarse por prácticamente cualquier parte de la aplicación si se extendiese en un futuro, así que es un servicio claro. Además, el servicio de Reservas con sus incidencias también tiene sentido separarle, ya que la ampliación es clara, ya que se podría tener un sistema de problemas no solo de espacios compartidos, sino de toda una comunidad.

Para hacer esta separación, se tienen que cortar irremediamente algunas de las relaciones que existían en el diagrama inicial para plasmarlas en forma de relaciones por identificador. Las relaciones que se han cortado son las siguientes:

- **User:** Esta clase deja de tener relaciones con Community y con Reservation y se le añade un campo Id de tipo string que se utilizará en Community y Reservation.
- **Reservation:** Deja de tener relación con Space y se le añade un campo spaceId para apuntar una reserva a un espacio concreto.
- **Space:** Debido al anterior, debe tener un nuevo campo Id para la identificación.
- **Community:** Se le añaden dos campos userIdList de tipo array y adminId, el primero será una lista de todos los usuarios que pertenecen a esa comunidad y el segundo indica quien es el administrador actual.

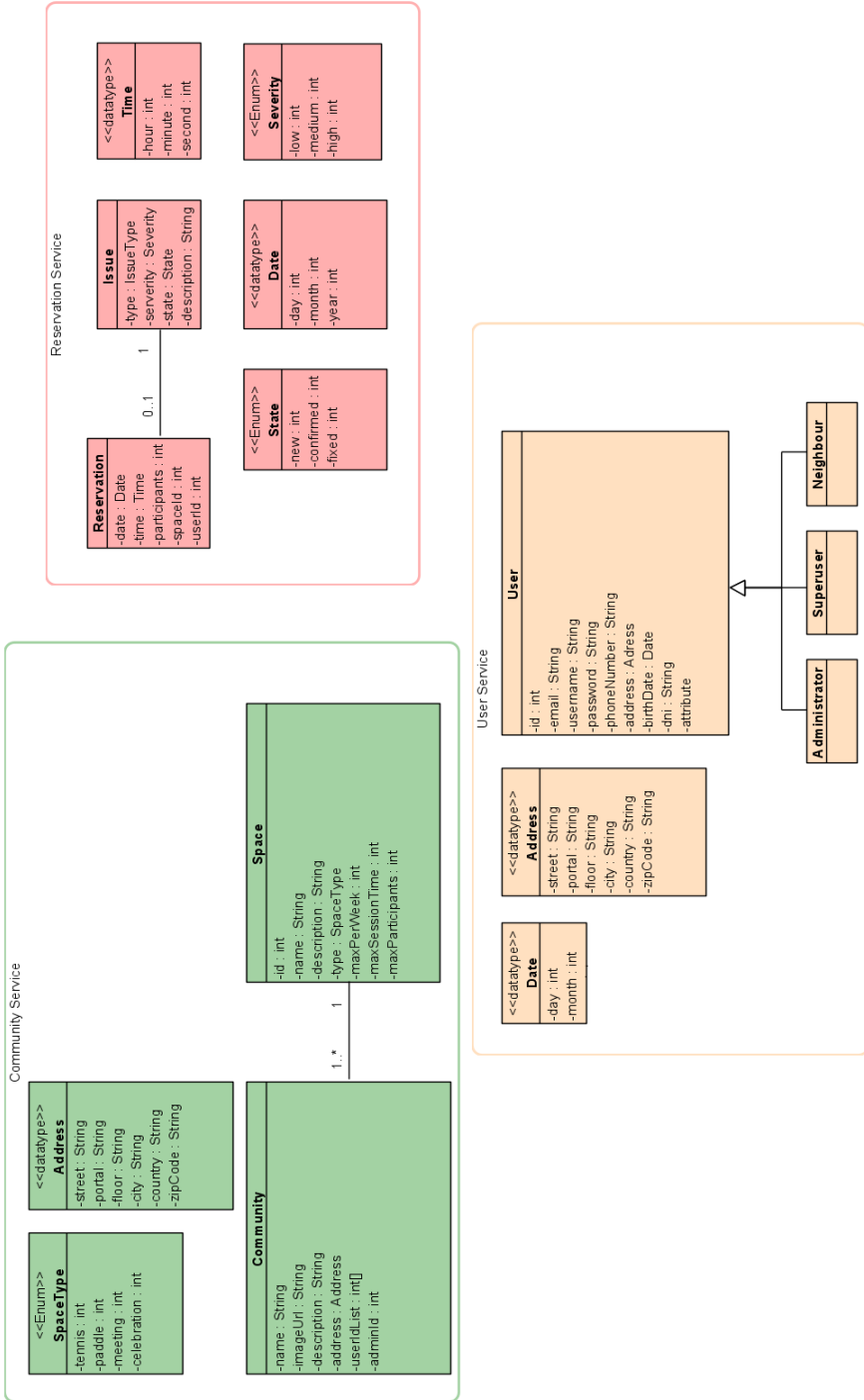


Figura 5.11: Diagrama de clases por microservicios

En la figura 5.11 se observan claramente la separación en cada uno de los tres microservicios y también se puede ver como algunos datatype se han tenido que implementar en varios servicios, como **Date** y **Address**

Uno se puede dar cuenta de que esas relaciones que antes estaban claramente definidas ahora son más difusas y podría haber redundancias de datos, colisiones y algún tipo de incoherencia, pero de la forma en que se ha separado es poco probable y tampoco se ha encontrado ninguna una vez estaban todos los servicios hechos y se estaba haciendo el frontend.

La complejidad se puede ver que es mayor, porque ahora estos tres servicios tienen que cruzar datos (aunque luego se verá que solo uno lo necesita) y ser llamados desde el frontend, pero a la larga esta es la decisión más inteligente si se llegase el punto de ampliar mucho la aplicación.

5.5. Diagrama de despliegue

El siguiente diagrama de figura 5.12 de despliegue muestra cómo los diferentes módulos de la aplicación están desplegados y cómo interactúan entre sí a través de APIs RESTful.

En la parte izquierda del diagrama, se puede ver un dispositivo cliente que no tendría por que ser un ordenador ya que la web también es totalmente funcional en otros dispositivos. En el hay un navegador web accediendo a la aplicación web "SM Community". La aplicación está compuesta por tres módulos principales: el Módulo de Reservas, el Módulo de Comunidad y el Módulo de Usuarios, los cuales serían cada uno de los servicios del frontend. Cada uno de estos módulos se comunica con los servicios específicos para cada una de las partes de la web.

El servidor 1 aloja el Servicio de Reservas y su correspondiente base de datos, así como la propia web. El flujo de datos se maneja de tal manera que las solicitudes del Módulo de Reservas del cliente son procesadas por el Servicio de Reservas en el servidor, que a su vez interactúa con la Base de Datos de Reservas. De manera similar, el servidor 2 gestiona tanto el Servicio de Comunidad como el Servicio de Usuarios, junto con sus respectivas bases de datos. Este diseño asegura que cada componente de la aplicación está desacoplado y es escalable de manera independiente asegurando también una buena mantenibilidad.

Obviamente el diseño ahora mismo no es perfecto y esto lo discutiré en el capítulo de 9 con un diagrama un poco más completo, pero el ecosistema en general funciona perfectamente y tiene un buen rendimiento.

Lo único fuera de lo común que se puede ver es que el microservicio de reservas consume la api del microservicio de comunidades, y es que es simplemente porque para realizar una reserva y ver si es válida la api de reservas necesita conocer algunas informaciones y configuraciones del espacio en el que se quiere reservar

5.6. Funcionamiento de las APIs

El funcionamiento general de las APIs es el mismo en todas, cada una con sus propios detalles del lenguaje pero todas similares. El flujo de ejecución es el siguiente, se hace una petición a la API, la cual a través del módulo de routing correspondiente envía la solicitud al **Controller** pertinente. Este Controller recibe los datos en forma de DTO, comprueba que no falten datos a gracias a los validadores de DTO y se le manda la información al **Servicio de aplicación** encargado de la acción que se llevará a cabo.

El servicio de aplicación convierte el DTO a una Entity a través del Mapper correspondiente a esa entidad para luego comprobar que todos los requisitos se cumplen. Una vez comprobado, se llama al **Repository**, que accederá a la base de datos para hacer la consulta y devolverá el mensaje en cuestión. Si se devuelven datos, el servicio de aplicación se encargará de volver a convertirlos en un DTO con el mismo Mapper y devolverse al controller para que lo devuelva en forma de HttpRequest. En la figura 5.13 se observa un pequeño esquema del funcionamiento en general de forma un poco más visual.

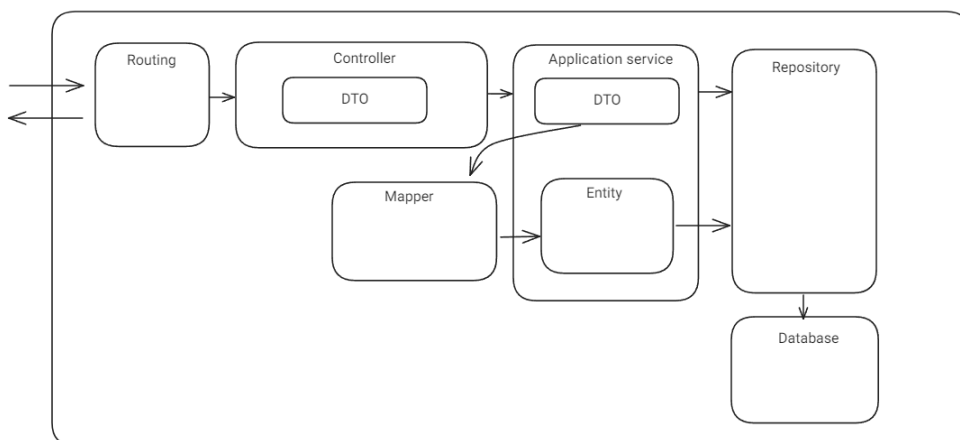


Figura 5.13: Esquema de flujo de ejecución API

Esta forma de trabajar y de ejecutar las peticiones a las APIs REST se basa en todas las arquitecturas y patrones mencionadas en el mismo capítulo y lo que se busca es una buena legibilidad y que el código se pueda escalar fácilmente. Al ser arquitectura hexagonal, la inversión de dependencias se utiliza en cada uno de los elementos para inyectar a través de inyección las dependencias inferiores y así poder trabajar con datos impostados en los tests.

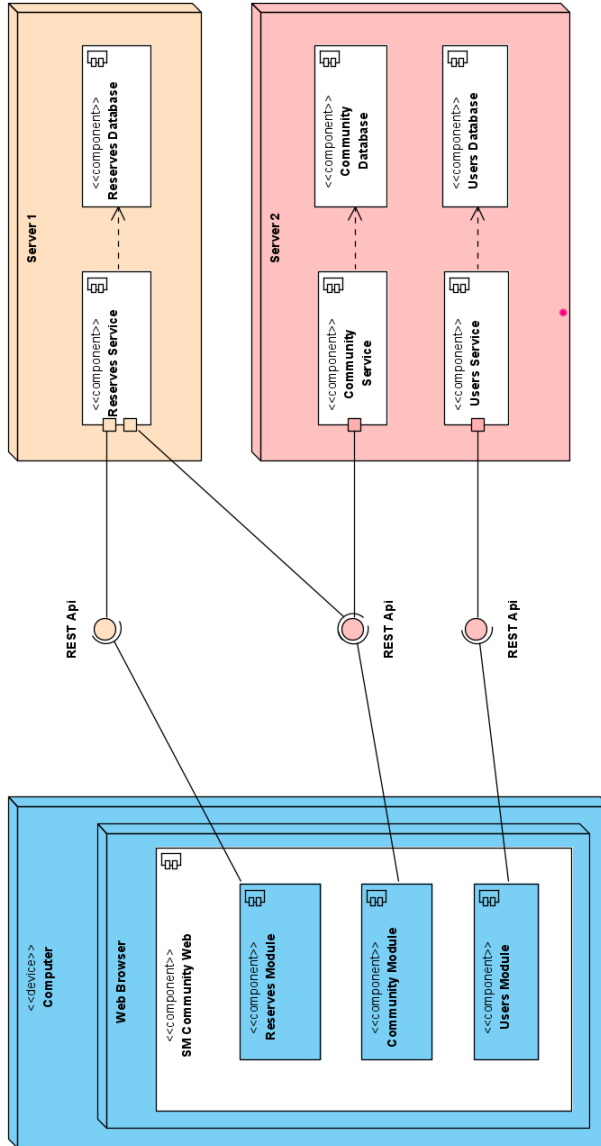


Figura 5.12: Diagrama despliegue y comunicación entre servicios.

Capítulo 6

Implementación

6.1. Trabajo previo

Previo a la implementación se realizó un trabajo a conciencia de investigación [42] de las mejores prácticas para estructurar la aplicación de forma que el despliegue posterior se haga de forma más fácil y limpio. Para ello, se ha generado una estructura de cuatro carpetas, tres para las APIs y una para el front-end, y se han creado variables globales en archivos `.env`[43] o `environment` para cada una de las partes de la web.

Me encargué de implementar los contenedores Docker desde el principio, así como scripts para el reparto de archivos y variables esenciales para cada servicio. Con esto, evito muchos problemas posteriores y genero una estructura limpia para posibles despliegues por otros desarrolladores. Todas las APIs y el frontend llaman a variables de entorno en todo lo que pueda ser necesario: urls, variable de desarrollo/producción, conexiones a bases de datos, credenciales, etc...

También se incluyen instrucciones claras y completas para la instalación y el despliegue de la aplicación, lo cual se explicara en una sección posterior. Además, tuve que buscar información sobre como implementar un entorno de producción y otro de desarrollo en Docker que también se verá en la sección de despliegue.

6.2. Implementaciones en común

En esta sección voy a hablar de implementaciones comunes a todas las APIs, ya que hay ciertas prácticas que se han realizado de una forma idéntica en las tres, salvando por supuesto los pequeños cambios entre lenguajes y frameworks, pero los conceptos son los mismos. Explicando aquí lo común, en las secciones concretas de cada servicio se explicarán las implementaciones especiales de cada uno de ellos.

6.2.1. Orquestación de cookies

Antes de empezar a hablar sobre los microservicios de la aplicación, hay que explicar como se ha gestionado la autenticación en las APIs y en el frontend, ya que se entiende mejor si se explica en conjunto. Básicamente, se ha optado por la aplicación de cookies de sesión con **JWT**(Json Web Tokens) firmados con claves publica/privada. Sobre esto hay bastantes opiniones y una alternativa de implementación.

El funcionamiento es el siguiente, un servicio de autenticación, que en este caso está en la API de usuarios, cifra gracias a la clave privada la información del usuario con una librería que cree JWT válidos [44]. En este token van incluidos el id de usuario, los roles, la expiación y otra información relevante, hasta aquí la forma de implementarlo es común a las dos alternativas.

Una opción es utilizar ese token como header, se envía como respuesta a la petición de login en el servicio de usuarios y el frontend lo guarda en memoria y como cookie o en localStorage. Con este token guardado, se manda como header *Authentication* en todas las peticiones que se realicen a las APIs para que estas con un **middleware** (o software intermedio) se compruebe la validez del token antes de que los controladores reciban la respuesta.

Esta opción está bien y es viable, pero tiene un problema principal y es por la que se prefirió no utilizarlo, y es que al guardar ese token en localStorage o como cookie sin **httpOnly**, es vulnerable a un robo de cookie o **cookie hijacking** a través de XSS o CSRF, uno de los ataques más peligrosos ya que el atacante puede directamente agregar la cookie a su navegador y saltarse todo tipo de login o doble factor. La propiedad mencionada httpOnly hace que a cookie solo pueda ser accedida a través de peticiones http, es decir, que solo el backend de origen podría coger la cookie.

Antes de seguir quiero explicar los ataques XSS y los CSRF. Un ataque **XSS**[45] (cross site scripting) se basa en realizar una inyección en una página web a través de scripts maliciosos añadidos en el código de la web en el lado del cliente. Esto es, por ejemplo, inyectar un script malicioso en un HTML a través de un formulario. Esto se haría de la siguiente manera, si un sitio es vulnerable a ataques XSS y hay sitios donde se pueda inyectar código JS (en un input de un formulario) que luego se vaya a visualizar en la web (en una tabla por ejemplo), el atacante podría añadir un script que coja la cookie de sesión del navegador y la envíe a un servidor suyo. Esto haría que el navegador de cualquier persona que viese esa información mandaría la cookie al atacante sin darse cuenta y le robarían su sesión.

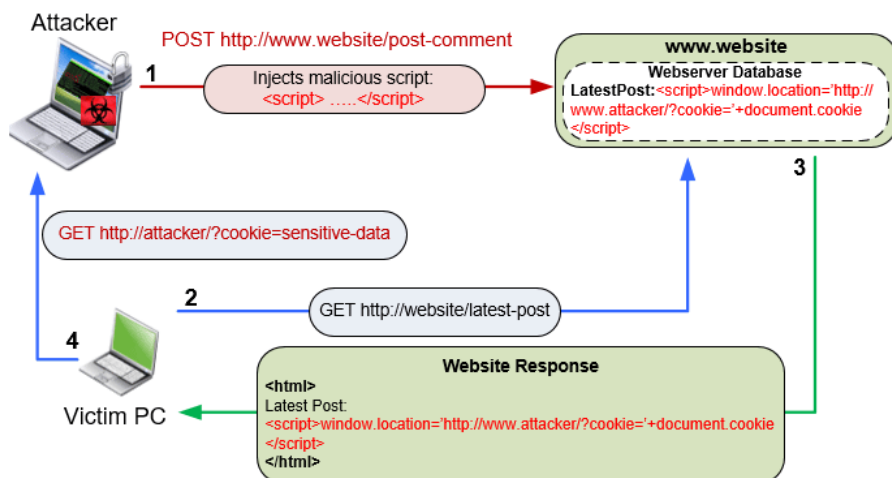


Figura 6.1: Esquema de un ataque XSS

Por otro lado, un ataque **CSRF** [46] (Cross Site Request Forgery) se produce cuando un usuario autenticado es inducido, a realizar una solicitud maliciosa en un sitio web en el que está autenticado. Si la web no verifica adecuadamente la validez de la solicitud, el atacante puede realizar acciones como transferir dinero o cambiar la configuración del usuario. Las cookies de sesión son enviadas automáticamente con cada solicitud al servidor, por lo que, si un atacante puede obtener acceso a estas cookies, puede enviar solicitudes autenticadas en nombre del usuario sin su conocimiento.

En esta web hay numerosos formularios y tablas donde se muestra información, y aunque Angular tenga protección nativa contra XSS que ya explicaré en su sección, prefería evitar problemas y posibles vectores de ataque [47].

Sabiendo todo esto, la implementación final ha sido utilizar cookies de sesión que lleven el JWT y las propiedades `httpOnly` y `secure` (hace que las cookies solo sean accesibles con el protocolo https). El funcionamiento es que el backend firma el JWT y se añade como cookie (header `Set-Cookie` [48]) con las propiedades comentadas, con lo cual la cookie estará en todas las peticiones al mismo dominio de origen el cual se ha puesto como `.sergiomotrel.com` para que todos los subdominios sean válidos. Con esto nos aseguramos de que el frontend no pueda acceder a la cookie y enviarla y de que siempre las cookies estén insertadas en las peticiones, haciendo que simplemente las APIs tengan que ver la validez del JWT que hay en ella y actuar en consecuencia.

Para validar este JWT, las tres APIs incluyen la clave pública y solo la pública correspondiente a la privada que se utilizó para firmarle. Cada API utiliza su propia librería correspondiente para hacer esta verificación pero todas con la misma clave.

6.2.2. Validación de peticiones

Como ya comenté en el capítulo de [Diseño](#), todas las APIs implementan el patrón de diseño **DTO** para el paso de información entre capas y como entrada a los controladores. Además, las tres implementan una librería cada una de validación de esos DTO que hacen mucho más fácil responder a malas peticiones o argumentos que faltan.

Los validadores de DTO simplifican el mantenimiento del código al centralizar la lógica de validación en un único lugar, se evita la duplicación de código y se facilita la actualización de las reglas de validación, haciendo simple nuevos requisitos de negocio sobre validaciones. Si en algún momento es necesario cambiar los requisitos de validación, estos cambios pueden realizarse en los validadores de DTO sin tener que modificar múltiples partes de la aplicación.

Otro beneficio clave es la mejora en la seguridad de la aplicación. Los validadores de DTO ayudan a prevenir ataques comunes, como la inyección de SQL o XSS, al validar y limpiar los datos de entrada de manera consistente. Esto proporciona una capa adicional de defensa que protege la aplicación contra datos maliciosos, complementando otras medidas de seguridad comentadas en la sección anterior.

Al principio es un poco tedioso pero prácticamente todas las librerías funcionan de la misma forma así que al final se ahorró tiempo. Qué librerías son y como se implementaron se explicará en la sección de cada API.

6.2.3. Control de errores y respuesta de las APIs

Una vez que los DTO han sido validados y se ha asegurado que los datos cumplen con los requisitos, se puede hacer una validación adicional en la capa de servicio para confirmar que los datos recibidos son correctos. Esta segunda capa de validación es crucial para verificar van más allá de la estructura de los datos, como la consistencia de los mismos, la existencia las ids a las que se referencia y el cumplimiento de reglas de negocio específicas. Por ejemplo, aunque un DTO pueda indicar que una fecha es válida, la capa de servicio debe asegurarse de que esta fecha esté dentro de un rango permitido o que no sea una fecha pasada si se quiere que sea futura (por ejemplo en las reservas).

Para gestionar estas validaciones adicionales, la capa de servicio verifica cada uno de los requisitos específicos del negocio. Cuando se encuentra un dato que no cumple con estas reglas, se lanzan excepciones descriptivas que puedan ser manejadas adecuadamente. Por ejemplo, si un usuario intenta registrar una reserva con un identificador de espacio compartido que no existe, la capa de servicio debe lanzar una excepción indicando que el producto no se encuentra en la base de datos. Estas excepciones proporcionan información útil para saber que es lo que salió mal.

Una vez que las excepciones son lanzadas desde la capa de servicio, el controlador las captura y las maneja de manera adecuada, capturando estas excepciones y traduciéndolas en respuestas HTTP significativas y con mensajes acordes. Para el ejemplo anterior, si la capa de servicio lanza la excepción, el controlador debería capturar esta excepción y devolver una

respuesta HTTP 400 (Bad Request) con un mensaje claro que describa que el espacio no existe.

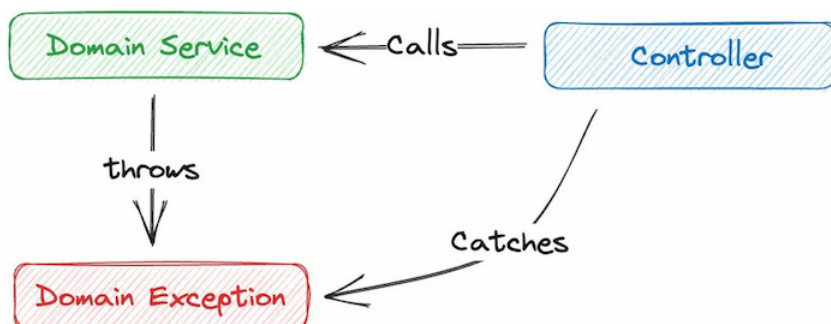


Figura 6.2: Esquema de manejo de excepciones en arquitectura por capas.

Este enfoque de manejo de excepciones mejora la robustez y la fiabilidad de la aplicación y de las APIs, además de poder dar una mejor experiencia de usuario mostrándole los errores correspondientes desde el frontend. Además, esta práctica permite mantener un código más limpio y legible, separando claramente las responsabilidades de validación, lógica de negocio y presentación de errores e incluso sirviendo como propia documentación, ya que se ve claramente por que y dónde se produce el error. Esta práctica se ha realizado en las tres APIs cada una de la forma en la que dicta su lenguaje o framework, así que en las secciones correspondientes obviaré el tema ya que se implementa de forma muy similar siguiendo el esquema de la figura 6.2 con try-catch.

6.2.4. Estructura de carpetas

Para seguir el diseño expuesto en el capítulo de 5 y la arquitectura hexagonal, se ha seguido una estructura de carpetas en la que se distingue claramente a que capa de la arquitectura corresponde cada fichero y cada clase. En la figura 6.3 se puede ver que la API tiene en su raíz tres carpetas correspondiendo a las capas de **Dominio**, **Aplicación** e **Infraestructura**. Dentro de Aplicación se ordenan los patrones DTO y los Mapper junto a los servicios de aplicación. En Dominio, tenemos las Excepciones, los Modelos y las abstracciones de los patrones Repositorio. Por último, en la capa de Infraestructura están los Controllers y las implementaciones de los Repositorios si fuesen necesarias.

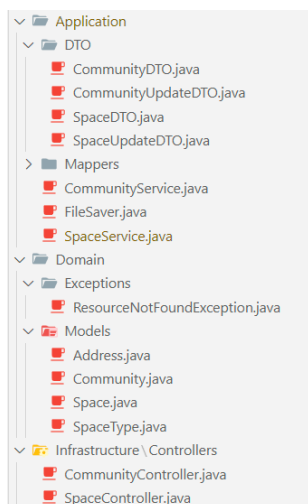


Figura 6.3: Estructura de carpetas de las APIs

Esta estructura de carpetas es común para los tres servicios del backend, dando una consistencia a través de ellos y haciendo que todo sea mucho más visual y organizado. Con esta estructura de carpetas se pueden realizar proyectos muy grandes, pero si fuese necesario, se podría ampliar montando este sistema de carpetas (Aplicación, Dominio e Infraestructura) como subsistema de carpetas para cada caso de uso. Por ejemplo, dos carpetas *Comunidades* y *Espacios* en las que estuviese un subsistema para cada uno.

6.2.5. Middleware JWT

En cada una de las APIs se ha montado un middleware de autenticación de tokens JWT que intercepta cada petición que se recibe para comprobar que lleve la cookie mencionada en la sección [Orquestación de cookies](#). Además de esto, se hacen las siguientes comprobaciones:

1. Si la cookie está presente, se extrae el token JWT de la misma.
2. Con la clave pública, se comprueba que el token JWT sea válido.
3. Si es válido, se extraen las *claims*, que contienen id de usuario, rol, expiración y el momento en el que el token fue firmado.
4. Si el token no ha expirado, se almacena la información de usuario y rol en la petición.
5. Se deja continuar a la petición (ahora con la información de la cookie) al controlador correspondiente.

Con este sencillo middleware, me aseguro que todas las peticiones a endpoints privados sean correctas y el usuario esté autenticado, además de obtener sus datos que más tarde se utilizarán para saber si el usuario tiene permiso sobre cierto endpoint.

6.3. API de usuarios

El servicio de usuarios se centra principalmente en manejar el registro, login y autenticación de usuarios, siendo el responsable de la firma de los tokens y de hacer de protección contra la parte pública de la web.

6.3.1. Estructura básica

Esta api es la primera que fue implementada, esto fue así debido a que es el núcleo de la información sobre los usuarios y hace las funciones de autenticación y generación de los tokens utilizados por toda la aplicación.

Como tecnología, está desarrollada con Nest.js. Al ser una tecnología muy similar a Angular, opté por utilizarla para esta API para tener las funciones más básicas y esenciales de la web implementadas de forma sólida y limpia.

La base de datos es una MongoDB de la que ya hablé en la sección 3.1.17, elegida por su capacidad de escalado y simplicidad de codificar con ella. También, como esta base de datos solo albergará usuarios y roles, la robustez de redundancia que teóricamente se podría perder frente a bases de datos relacionales debido a la pérdida de la integridad relacional no será preocupante, ya que una base de datos relacional equivalente constaría de solo dos tablas[49].

Para la integración de estas dos tecnologías, tuve que añadir al backend la librería Mongoose, una librería de modelado de datos para JavaScript[50]. Con Mongoose tenemos una alternativa a la opción nativa de Nest para aplicar el modelo **ORM** (Object Relational Mapping) de forma más específica a Mongo.

ORM nos permite mapear de forma directa los objetos que estamos manejando en nuestro código con las estructuras de nuestra base de datos[51]. Esto nos permite una codificación mucho más ágil ahorrando tediosos procesos para transformar la información entre la capa de datos y la capa aplicación. En este caso con Mongoose necesitamos codificar algunas clases e interfaces para que todo funcione correctamente y se genere ese mapeo, esto se puede ver en la figura 6.4.


```
1   @Schema()
2   export class User {
3     id: string;
4
5     @Prop({ required: true })
6     username: string;
7
8     @Prop({ required: true })
9     name: string;
10    @Prop({ required: true })
11    lastname: string;
12
13    @Prop({ required: true })
14    email: string;
15
16    @Prop({ required: true, minlength: 8 })
17    password: string;
18
19    @Prop({ required: true, default: Role.neighbour })
20    role: Role;
21  }
22  export const UserSchema = SchemaFactory.createClass(User);
23
```

Figura 6.4: User Entity

Según dicta la documentación de Nest^[52] para el uso con Mongoose, se deben definir el modelo de la clase que nos interese como una **entidad** o entity, que es la representación de un objeto del dominio, en este caso User.

El decorador **@Schema** marca la clase como un esquema, mapeándolo directamente a Mongo. **@Prop** define que el siguiente atributo es una propiedad del schema y se le pueden dar distintos validadores como el requerimiento, la longitud, etc... Por último a través de **SchemaFactory** definimos como una constante el esquema anterior para su uso en la aplicación.

También tuve que añadir la conexión a la base de datos a través de la importación de MongooseModule y hacer otras pequeñas configuraciones, todo evitando el uso de credenciales en texto claro y utilizando las variables de entorno ya mencionadas.

```
1     import * as dotenv from "dotenv";
2
3     dotenv.config();
4
5     @Module({
6       imports: [
7         ConfigModule.forRoot(),
8         UsersModule,
9         MongooseModule.forRoot(`mongodb://${process.env.MONGO_USER}:
10        ${process.env.MONGO_PASS}@${process.env.MONGO_URI}/
11        ${process.env.MONGO_DB}?authSource=admin&directConnection=true`),
12        ],
13       controllers: [AppController],
14       providers: [AppService],
15     })
16     export class AppModule {}
17
```

Figura 6.5: Inicialización de Mongo con variables de entorno

Para proteger algunas rutas de accesos indeseados implementé un Guard de rutas que solo permitiese el acceso cuando el token adjunto a la petición sea correcto, es decir, que el usuario este logeado y el token se encuentre dentro de la caducidad. Esta gestión de los tokens, de crearles y de comprobarles, se ha realizado a través de la librería de JWT de Nest. En estos JWT se incuye tanto una caducidad, como la id y el rol del usuario.

En lo que a gestión de los datos se refiere, implementé **DTOs** (Data Transfer Object) para todos los tipos de datos que se gestionan en la aplicación, y generé todos los tipos de datos necesarios para que la información sea gestionada de la mejor forma posible y que así cualquier modificación o mantenimiento sea fácil de realizar.

6.3.2. Endpoints

Las operaciones o **endpoints** de la Api codificadas hasta la fecha son las necesarias para poder implementar un CRUD (create, read, update y delete) pero un poco ampliado y modificado, ya que para algunas funcionalidades necesitaba algunas cosas específicas ya sea por seguridad o por comodidad a la hora de gestionar las llamadas, se pueden encontrar en el apéndice D. Todos los endpoint estan configurados en una clase controladora con el decorador `@Controller`, que hace llamadas a un servicio de aplicación para la gestión de la información y las llamadas a la base de datos, intentando siempre mantener una buena organización y buenas prácticas.

Las contraseñas se encriptan con Bcrypt a la hora de registrar y se comprueba contra esa

cadena encriptada una vez el usuario intenta hacer login, pero se ha intentado bajo ningún concepto devolver las contraseñas ni las encriptaciones.

Como dato, el endpoint en la ruta /login es el que genera la cookie y la añade a la respuesta para obtener los beneficios comentados en el apartado [Orquestación de cookies](#). Las propiedades domain y secure se controlan por variable de entorno ya que en local deberían tener valores distintos al no tener protocolo https.

```
1     response.cookie('token', token, {
2         httpOnly: true,
3         secure: Boolean(process.env.SECURE_COOKIE),
4         maxAge: 3600000,
5         domain: process.env.DOMAIN
6     })
7
```

Figura 6.6: Añadido de cookie a la respuesta con variables de entorno.

Todos los endpoint sensibles se han protegido detrás de una comprobación del token JWT adjuntado como cookie comentado anteriormente, esto se comprueba a través de. Además, si el endpoint en concreto tiene algún tipo de riesgo para la aplicación o para los datos del usuario, se ha protegido también comprobando el rol del usuario de ese JWT, permitiendo solo a los Administradores o a los Superusuarios realizar la llamada.

6.4. API de comunidades y espacios

La segunda API (a partir de ahora API de comunidades) maneja todas las Comunidades, Espacios y todas las relaciones entre ambas, siendo la API central y más importante de las tres.

6.4.1. Estructura básica

El servicio de Comunidades está implementado con Spring Boot, el framework basado en Java, el cual trae muchas funcionalidades predefinidas y orientadas para un trabajo por capas. Como para este servicio se está utilizando Maven, todas las dependencias van añadidas en el archivo **pom.xml**, entre las más importantes que se han añadido están las siguientes:

- **Springframework Security:** Añade la cadena de seguridad de Spring al proyecto, de la cual se hablará más adelante debido a su relevancia.
- **Jsonwebtoken:** Para la implementación del middleware JWT.

- **Mysql:** Habilita las conexiones y el driver para bases de datos Mysql.
- **Básicos de Spring:** Spring config, Spring Starter Data y Spring starter validation son algunas de las dependencias básicas para el acceso a datos, utilización de repositorios y validación de DTO

6.4.2. Servicio de almacenamiento

Cuando estaba implementando el microservicio de comunidades, rápidamente surgió la necesidad de implementar algún tipo de almacenamiento de imágenes, ya que tanto la comunidad como los espacios tienen una foto identificativa. Esto podría haber sido otro microservicio totalmente ajeno, pero dado el tamaño de la aplicación se decidió no hacerlo de esta manera y que directamente fuese parte del microservicio de comunidades.

Para empezar con esto, se tuvo que decidir donde irían alojadas las imágenes, y en este caso es en la carpeta `/statics` que además se tuvo que añadir al `.gitignore` para que esa carpeta no se repositara. La implementación se hizo ayudado de la propia documentación de Spring [53] que da una buena idea de como hacerlo. Para empezar, hay que crear un archivo de configuración para la ruta que utiliza el decorador `@ConfigurationProperties` y que luego hay que habilitar en el archivo `main` de la aplicación con el decorador `@EnableConfigurationProperties` indicando la clase de las propiedades. Además, hay que iniciar el servicio también en el `main` de la aplicación apuntando a un método de iniciación del servicio de almacenamiento.

El servicio en sí se ha implementado a través de una interface `StorageService` (figura 6.7) para que si en un futuro se quisiera añadir otro tipo de implementación, por ejemplo para ficheros PDF, sea mucho más cómodo. En la capa de aplicación hice una implementación de esta interface llamada `ImageStorageService` que se encarga de recuperar, guardar y tratar los errores de las imágenes, asegurando además de que exclusivamente se guardan imágenes y se sirvan como recurso en vez de descarga.

Como extra, se crearon las excepciones pertinentes y una clase para devolver la respuesta correctamente, indicando el nombre de la foto y la url para un correcto tratamiento. Con todo esto preparado quedaba implementar el controlador para manejar el enrutado teniendo de base `/images` y como endpoints un GET de una imagen, un GET de una lista de imágenes con `/all` como subruta y un POST para subir con `/upload`.

Después surgió el problema de que las imágenes no se mostraban de forma correcta disparando un error 404 (No encontrado) y después 403 (Prohibido). El error 404 resultaba ser debido a que por una mala configuración estaba apuntando a la ruta `/images` del sistema de archivos y no a `/statics` como se mencionó anteriormente, por lo que evidentemente no las encontraba al no haber nada.

```
1 public interface StorageService {
2     void init();
3
4     void store(MultipartFile file);
5
6     Stream<Path> loadAll();
7
8     Path load(String filename);
9
10    Resource loadAsResource(String filename);
11
12    void deleteAll();
13 }
14
```

Figura 6.7: Interfaz StorageService.

6.4.3. Spring Security

La cadena de filtros de SpringWebSecurity [54] se encarga de aplicar ciertos filtros de seguridad, cada uno con responsabilidad única, para asegurar que el servicio es seguro y, además, dando la posibilidad de prohibir o permitir rutas dependiendo de lo que se requiera.

Como se ha comentado, hubo una mala implementación que llevó a ciertos quebraderos de cabeza con la implementación de **SpringWebSecurity** y su cadena de filtros. El error originó en la falta de conocimiento, ya que se entendió que la cadena de filtros funcionaba de la siguiente manera: las peticiones pasan por la cadena de filtros, primero autorizando las peticiones http con `authorizeHttpRequests` y luego dándole a esta función las configuraciones de los tipos de peticiones que están permitidas. En el caso de la figura 6.8 serían todas las peticiones `OPTIONS` para permitir hacer el apretón de manos previo y luego la carpeta `/images` entera menos `/upload` estarían permitidas, para que por último las peticiones no especificadas fuesen autenticadas. Después de esto, se pensó que las peticiones que requiriesen autenticación pasaban por el filtro `JWTFilter`, el cual dejaba pasar o no según si se cumplían sus requisitos.

```
1  @Bean
2  public SecurityFilterChain filterChain(HttpSecurity http) {
3      http
4          .cors(Customizer.withDefaults())
5          .csrf(csrf -> csrf.disable())
6          .authorizeHttpRequests((authz) -> authz
7              .requestMatchers(HttpMethod.OPTIONS, "/*").permitAll()
8              .requestMatchers("/images/upload").authenticated()
9              .requestMatchers("/images/*").permitAll()
10             .requestMatchers("/spaces/get/*").permitAll()
11             .anyRequest().authenticated())
12             .addFilterBefore(new JwtFilter(),
13                 UsernamePasswordAuthenticationFilter.class);
14
15     return http.build();
16 }
17
```

Figura 6.8: Cadena de filtros de SpringWebSecurity.

Cuando se encontró el fallo de las imágenes, se observó que algo estaba mal implementado, y es que la idea de funcionamiento de la cadena era errónea, ya que lo que ocurre es que el `JWTFilter` ocurre previo al filtro de autenticación con usuario y contraseña y a su vez este filtro ocurre antes de la configuración del tipo de peticiones y rutas, por lo que el filtro estaba directamente rechazando las peticiones que no traían JWT (porque no fuesen necesarias) y no permitiendo continuar con la cadena. El comportamiento correcto es el siguiente: Se recibe una petición y se comprueba si lleva JWT y si es válido, si es así, se incluye una `UsernamePasswordAuthenticationToken` en el contexto de la cadena de filtros y se deja pasar la petición, si el caso es el contrario, simplemente hay que dejar pasar la petición pero sin el token en el contexto, para que luego el filtro `UsernamePasswordAuthenticationFilter` se encargue de comprobar si las rutas que necesitan estar autenticadas tienen el token, mientras que las que no lo necesitan simplemente se permite.

Haciendo esos cambios en la lógica del filtro JWT ahora se puede controlar de forma mucho más sencilla y sin tener que añadir código al filtro que peticiones deben asegurarse simplemente añadiendo una línea a la cadena de filtros si se quiere que un endpoint concreto se permita.

Además de todo esto, `WebSecurity` se encarga de los `Cross Origin Resource Sharing` (CORS), admitiendo o no compartir con ciertos dominios y orígenes según mis intereses y aplicando los headers correspondientes, que en este caso caso se permitió todo tipo de operaciones del origen del entorno local y al que luego se añadió el origen del servidor de producción del frontend para que las peticiones que vienen de ahí se permitan.

6.4.4. Endpoints

Al igual que se implementó la API de usuarios, la de comunidades implementa un CRUD algo ampliado para las entidades Comunidad y Espacio, todo esto utilizando DTOs y mappers como se viene explicando en capítulos anteriores, se puede encontrar en el apéndice C una guía de los mismos.

Estos endpoints dan una base sólida para realizar todas las operaciones que el frontend necesita y son ejecutados por los servicios de aplicación y por un **JpaRepository**, que es un repositorio ya implementado de Spring, que gracias a una convención de nombres [55], permite simplemente hacer consultas a la base de datos a través de un nombre. Por ejemplo *findByAdminIdOrNeighbourIds* hace una búsqueda de comunidad por Administrador o Vecino sin tener que hacer la implementación, algo que es muy potente y conveniente.

```
1 interface CommunityRepository extends JpaRepository<Community, Long> {
2
3     List<Community> findByAdminIdOrNeighbourIds(@Param("userId") String id);
4
5     Optional<Community> findByInvitationCode(String invitationCode);
6
7 }
8
```

Figura 6.9: Implementación de repositorio con JpaRepository.

6.5. API de reservas

La última API maneja la gestión de Reservas de la aplicación y es la única que necesita comunicarse con otra de las APIs (API de comunidades) para el caso específico de comprobar ciertos parámetros del espacio, como el tiempo máximo de reserva por usuario.

6.5.1. Estructura básica

El servicio está implementado con Slim Framework en su versión 4 en PHP 8 y una base de datos PostgreSQL como base de datos. Al contrario que las otras dos, no utiliza ningún tipo de ORM para el acceso a datos ni la persistencia, sino que hace consultas SQL.

Gracias a composer, la instalación de dependencias es muy sencilla y se han instalado **Guzzle** [56], y el **Validator de Symfony** (el framework de PHP) para validar DTOs. En Slim, es muy común utilizar ciertos paquetes de Symfony, ya que simplifican mucho las cosas y ya vienen implementadas, como por ejemplo el Validator que se ha utilizado o incluso el

ORM nativo de Symfony Doctrine. En este caso no se instaló Doctrine ya que la integración era complicada y no lo vi necesario.

Además de lo anterior, Slim y otros frameworks de PHP utilizan un contenedor de dependencias como ya mencione en la sección 3.1.11, que sirve para manejar la inyección de dependencias en nuestras clases. Como se puede ver en la figura 6.10 vemos como por ejemplo en el servicio de comunidades se está inyectando el cliente de Guzzle o la configuración de la url de la API.

```
1 Client::class => function () {
2     return new Client();
3 },
4 CommunityService::class => function (ContainerInterface $container) {
5     $settings = $container->get(SettingsInterface::class)->get('services');
6     return new CommunityService($container->get(Client::class),
7         $settings['community_service_url']);
8 },
9 ValidatorInterface::class => function () {
10    return Validation::createValidatorBuilder()
11        ->enableAttributeMapping()
12        ->getValidator();
13 },
14 ReservationRepository::class => function (ContainerInterface $c) {
15    return new PosgresReservationRepository($c->get(PDO::class));
16 },
17
```

Figura 6.10: Parte del contenedor de dependencias de Slim.

Al respecto de esta configuración, Slim tiene un servicio de configuración muy conveniente que permite cargar ahí todos los datos necesarios y que sean accesibles desde cualquier lugar de la aplicación. En la figura 6.11 se muestra como se utiliza la interfaz **SettingsInterface** propia de Slim y se cargan todos los datos que se utilizarán a través de variables de entorno que vienen del archivo ya conocido **.env**.

Por lo demás, el resto está implementado exactamente de la misma forma que se comentó en las secciones **API de usuarios** y **API de comunidades y espacios**, incluyendo routing, controllers, servicios de aplicación y entidad de dominio(Reserva).


```
1 SettingsInterface::class => function () {
2     return new Settings([
3         'displayErrorDetails' => true,
4         'logError' => true,
5         'logErrorDetails' => true,
6         'logger' => [
7             'name' => 'slim-app',
8             'path' => isset ($_ENV['docker']) ?
9                 'php://stdout' : __DIR__ . '/../logs/app.log',
10            'level' => Logger::DEBUG,
11        ],
12        'connection' => [
13            'driver' => $_ENV['DB_DRIVER'],
14            'host' => $_ENV['DB_HOST'],
15            'port' => $_ENV['DB_PORT'],
16            'dbname' => $_ENV['POSTGRES_DB'],
17            'user' => $_ENV['POSTGRES_USER'],
18            'password' => $_ENV['POSTGRES_PASSWORD'],
19        ],
20        'services' => [
21            'community_service_url' => $_ENV['COMMUNITY_SERVICE_URL'],
22            'user_service_url' => $_ENV['USER_SERVICE_URL'],
23        ]
24    ]);
25 }
26
```

Figura 6.11: Parte del contenedor de dependencias de Slim.

6.5.2. Comunicación con API de comunidades

Para la comunicación más sencilla con la otra API, se ha implementado un servicio de aplicación para hacer la comunicación, DTO para las respuestas y la librería Guzzle para realizar las peticiones. Esta decisión de utilizar Guzzle se ha tomado porque la forma tradicional haciendo cURL es muy propensa a errores en PHP, ya que es mucho más arcaica.

En la figura 6.12 se puede ver lo simple que queda gracias a Guzzle, que se asemeja mucho a como se hacen las peticiones en otros frameworks como por ejemplo Angular, utilizando un httpClient.

```
1 class CommunityService{
2
3 private Client $httpClient;
4 private string $apiUrl;
5
6 public function __construct(Client $httpClient, string $apiUrl)
7 {
8     $this->httpClient = $httpClient;
9     $this->apiUrl = $apiUrl;
10 }
11
12 public function getCommunityParametersBySpaceId(int $spaceId)
13 : CommunityConfigDTO
14 {
15     $response = $this->httpClient->get("{$this->apiUrl}/{$spaceId}");
16     $data = json_decode($response->getBody()->getContents(), true);
17
18     return new CommunityConfigDTO(
19         $data['maxPerWeek'],
20     );
21 }
22 }
23
```

Figura 6.12: Servicio de comunicación con la API de comunidades.

6.6. Frontend con Angular

6.6.1. Estructura básica

El desarrollo con Angular fue muy fácil y rápido debido a que es la tecnología a la que más habituado estoy. Para empezar el proyecto, generé la estructura de carpetas, que consta de varias carpetas raíz que tienen un subsistema de carpetas dentro con los elementos básicos de angular como módulos, componentes, servicios, interfaces, directivas y guards. Para una guía más a fondo de uso, se puede consultar el apéndice D, pero la plantilla básica de la parte interna de la aplicación es la que se puede ver en la figura 6.13.



Figura 6.13: Plantilla básica de la web.

Estas carpetas raíz que se comentaban tienen distintos propósitos y están separadas en:

- **Shared:** Una carpeta compartida con su módulo correspondiente donde se metieron todos los componentes que fuesen a ser reutilizados y compartidos por varios módulos.
- **Login:** En ella se agrupa todo lo que está relacionado con el login y el registro.
- **Home:** Todo lo relacionado con la plantilla, el menú lateral y el contenedor central dinámico.
- **Manage:** El módulo y componentes de toda la parte de administración y gestión para los usuarios Superuser y Administrador.

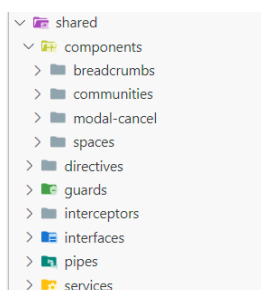


Figura 6.14: Estructura de carpetas básico en el frontend.

El subsistema de carpetas comentado se asemeja a la figura 6.14, siendo esta la parte de *shared* que es la más grande junto a la de Manage. Como se puede observar, ha una parte de

guards, y es que las guards han sido imprescindibles en este frontend, ya que todo el sistema de permisos está implementado con la ayuda de tres guards y una directiva.

Los tres guards sirven para proteger rutas de no estar identificado, por rol y la última para gestionar que partes puede el usuario editar en la parte de administración, ya que un usuario Administrador no debe poder editar las mismas cosas que un Superuser. La guard de identificación, por otra parte, sirve para que si el usuario no tiene un token correcto, ha expirado o no está identificado, no pueda entrar a ninguna ruta que no sea login o registro.

La directiva, por otra parte, sirve para mostrar ciertos elementos según el rol, de forma que es muy sencillo hacer que un elemento HTML aparezca o no sin tener que llenar la parte HTML de los componentes de comprobaciones al uso. En la figura 6.15 se ve un enlace a otra sección que solo aparecerá si el usuario es Administrador o Superusuario, además de que esa ruta en su propio routing está también protegida por rol

```
<a *appHasRole="['superuser','administrator']" [routerLink]="['/administrar/comunidades']" type="button"
```

Figura 6.15: Directiva de control por roles.

```
{
  path: 'comunidades',
  component: CommunitiesListComponent,
  canActivate: [AdministratorGuard],

  data: {
    type: 'manage',
    breadcrumb: 'Administrar comunidades',
    allowedRoles: ['administrator', 'superuser'],
  },
},
```

Figura 6.16: Ruta protegida por un guard de rol administrativo.

6.6.2. Servicio de autenticación

Para el sistema de autenticación y mantener la sesión, se implementó un servicio que mantiene los datos del usuario, el cual se encarga de comprobar la información del token, si es válido y si ha expirado cada 5 minutos. El servicio tiene un sistema de cache para no sobrecargar el backend que hace que si un token ha sido validado se mantenga durante esos 5 minutos sin volver a hacer la petición.

Teniendo en cuenta que la cookie del backend tiene una hora de duración, el usuario podrá mantener la sesión durante ese tiempo, aunque siempre se podría aumentar la duración tanto de la caché como de la duración de la cookie en el backend.

Este servicio ha resultado muy útil para recuperar la información del usuario en muchos sitios de la web, ya que es muy normal cruzar datos con la API de usuarios para diversas operaciones, por lo que teniendo este pequeño servicio se puede aprovechar para evitar llamadas al backend simplemente para buscar información. Obviamente, no se mantiene ningún tipo de dato sensible como la contraseña.

6.6.3. Formularios

La parte más importante del frontend han sido los formularios, ya que la web consta de más de seis formularios cada uno con sus propios requisitos. Para esto, Angular brinda una herramienta perfecta a través de los FormGroup, FormControl y los Validators[57], que son clases del modulo ReactiveForms que simplifican mucho la gestión de formularios y como validarlos. Esta validación es tan facil como crear un FormGroup con un control para cada input y los validadores que se le quiere aplicar, en la figura 6.17 se puede ver un ejemplo de ello.

```
this.spacesForm = this.fb.group({
  name: ['', [Validators.required, Validators.minLength(2)]],
  description: ['', [Validators.minLength(2)]],
  type: ['', [Validators.required]],
  image: [null, [Validators.required]],
  maxParticipants: [1, [Validators.required]],
  maxPerWeek: [1, [Validators.required]],
  communityId: ['', [Validators.required]],
})
```

Figura 6.17: Ejemplo de validación de formularios en Angular.

Gracias a esta sencilla validación, solo quedaba ocuparse de hacer las peticiones HTTP correspondientes a los backend, y para eso se han utilizado servicios separados para cada uno de los Modelos en cuestión, es decir, comunidades, reservas, espacios y usuarios. Estos servicios inyectan el HttpClient de Angular[58] para hacer peticiones y gracias a los Observables de la librería RxJs[59] que utiliza Angular el control de las respuestas ha sido muy sencillo.

En una futura actualización de la web, se podría intentar hacer un componente de formularios genérico para no tener varios componentes formularios específicos para cada cosa, pero dado que es más tardado y delicado, se prefirió realizar de esta forma en un inicio.

6.6.4. Maquetado

Como ya se comentó en la sección 3.1.18, se ha utilizado la librería de componentes TailwindUI que brinda componentes HTML de todo tipo para utilizar con una estética moderna y muy fáciles de personalizar, por lo que excepto algunas cosas concretas como el login, registro y el componente de reservas, el resto se ha delegado a esta librería de componentes y después se han modificado detalles para dar una estética más uniforme.

Como es el más interesante, se va a detallar el componente de reservas, que aunque parezca sencillo ha sido un pequeño reto, ya que tiene muchas peticiones por detrás y tiene que recuperar datos de todas las APIs. Este componente necesita las siguientes informaciones:

- **Comunidad:** La comunidad actual para mostrar su información.
- **Espacio:** El espacio en el que se hace la reserva, de el se consulta la información que se ve en el cuadro verde.

- **Usuario:** Aunque no se muestre nada, se mantiene la información del usuario en un servicio, ya que se necesita para realizar la reserva.
- **Reservas:** De este servicio se recuperan todas las reservas del espacio de hoy para mostrar la disponibilidad y luego se insertan todas las reservas realizadas.

Sabiendo esto, se puede concluir con facilidad que el cuello de botella de la aplicación seguramente esté en este componente, ya que el intercambio de información y cruce de datos, además de que los servidores utilizados son de gama baja, podrían producir un pequeño retardo.

Como este componente es el punto intermedio de toda la información, el cruce de datos se realiza en el frontend y ciertas comprobaciones antes de enviar la información también, como por ejemplo si el usuario tiene horas disponibles, que aunque luego se vuelva a comprobar en el backend, evita llamadas innecesarias al mismo.

6.7. Experiencia de usuario

Esta sección se centrará en la experiencia de usuario y en pequeñas buenas prácticas que se han intentado aplicar o que se aplicarán en un futuro. Aunque en muchas ocasiones se mencione el SEO, se refiere a las dos cosas, pues lo que busca Google con el posicionamiento de las webs es ordenarlas por utilidad y buena experiencia para el usuario.

6.7.1. Enlaces y botones

Los enlaces de la web deben ser siempre elementos tipo ancla con una propiedad href o `<a>`, esto es básico, pero importante. Sobre todo en este caso haciendo uso de Angular y en concreto la propiedad `routerLink` [60] que es utilizable en todas las etiquetas HTML, hay muchas veces que se tiende a en un `<div>` lo cual es un error.

También, es aconsejable utilizar el texto de enlace que sea lo más descriptivo posible para que tanto Google como los usuarios tengan un mejor contexto de donde van a ser redirigidos. Además, en Angular la propiedad `routerLink` ya mencionada se transforma en compilación al href correspondiente, así que no hay problema [61].

Respecto a los botones ocurre algo muy parecido, están totalmente desaconsejados de utilizar como un enlace y la función que deberían hacer es realizar una acción dentro de la misma página, es decir, no hacer redirección ni navegación. Se pueden, sin embargo, utilizar en cualquier acción del usuario que implicase una interacción con la web, o la ejecución de un script. Por ejemplo, un botón que muestre un modal, o un botón que interacciones con un formulario.

Estas cosas, aunque parezcan poca cosa, si se van sumando errores de SEO puede acabar habiendo problemas. Además, lo mencionado en este punto también aplica a la accesibilidad, lo cual amplío en el siguiente punto [62].

6.7.2. Accesibilidad

La **accesibilidad** web se refiere a hacer que los sitios web sean fácilmente utilizables por el mayor número posible de personas, incluyendo las que tienen condiciones especiales y asegurando que puedan percibir e interactuar con la web [63].

Este tema puede llegar a ser extenso y es algo en lo que no se ha hecho mucho incapié, salvo algunas cosas básicas recomendadas para los elementos más repetidos e importantes de la web. En concreto, se ha realizado lo siguiente:

- **Etiquetas en general:** Se deben utilizar las etiquetas en un contexto apropiado. Si se quiere hacer un footer, la etiqueta a utilizar debería ser `<footer>`, y así, con el resto de elementos. A esta práctica se le llama **HTML semántico** y es la tendencia hacia la que está virando el maquetado web.
- **Imágenes:** Utilización de la propiedad *alt* que debe describir lo que se ve en la imagen. Evitar completamente los textos que forman parte de la propia imagen, si es necesario texto, con HTML y CSS. Además, de ser obligado, el texto debe ser claramente visible, pudiendo utilizar técnicas como una pequeña sombra en el texto para generar contraste.
- **Formularios:** Muchas veces utilizar solo placeholder en los input de los formularios no es suficiente para describir que es el campo, así que se ha planteado para que dentro de lo posible se apliquen etiquetas o *labels*.
- **Colores:** Los colores tienen que estar bien contrastados para que sea fácilmente legible y para esto se han utilizado las herramientas de desarrollador de Firefox y Google Chrome que tienen una ayuda para comprobar si el contraste es suficiente. Los colores de la web al ser contrastes de grises tienen una buena legibilidad.

Como cosa a mayores que podría implementar es la propiedad *tabindex* de las etiquetas HTML [64], que indica si una etiqueta se ve afectada al utilizar la tecla tabulador del teclado, ya que muchas personas con discapacidades utilizan esta forma de navegación.

6.7.3. Core Web Vitals

Las Core Web Vitals (CWV) [65] de Google, son un conjunto de métricas diseñadas para la evaluación y análisis de la experiencia del usuario en la web. En esta categoría se incluyen oficialmente cuatro métricas cada una enfocada en aspectos distintos como la velocidad de carga, la interactividad y la estabilidad visual

6.7.4. Herramientas

Para la medición y la detección de problemas en las webs, existen múltiples herramientas que analizan diferentes aspectos de una web. Para mí, no son competencia unas de otras,

sino que cada una aporta distintos puntos de vista y haciendo un conglomerado de todos lo análisis te puedes formar una idea de lo que está pasando y investigar en como enfocar cada problema. Algunas de las utilizadas para las mediciones son:

- **Google Lighthouse:** Extensión para navegador concebida para el análisis en tiempo real del rendimiento, SEO, accesibilidad y prácticas de una URL. Estos cuatro puntos como se ha explicado van todos de la mano, así que tener un buen valor en todas ayuda al indexado. La forma de medir es en una escala de cero a cien en cada punto, teniendo tres rangos distintos de "gravidad" separados por colores, los cuales indican aproximadamente en que rango debería estar la web para indexar correctamente.
Además de la puntuación, una de los grandes beneficios de utilizar Lighthouse es una página es cada una de las categorías que te indica que problemas ha encontrado a la hora de ejecutar la web y de donde podrían venir. Aunque no es certero al cien por cien, puede dar muy buenas pistas para encontrar errores.
- **PageSpeed Insights:** Parecida a la anterior pero en forma de web, da métricas y puntuaciones a tu web. Lo mejor es combinar ambas y hacer una síntesis de los problemas que muestran ambas, ya que se complementan muy bien. En el caso de esta su fuerte es un análisis profundo de las CWV.

En palabras propias de Google, *„Esto, junto con otros aspectos de la experiencia en la página, se corresponde con lo que nuestros sistemas de posicionamiento principales buscan recompensar”*. Sabiendo esto, es muy importante que por lo menos se consiga una puntuación decente y que muchas de las runas de la web sean categorizadas en Search Console como rutas que cumplen las métricas.

Sin embargo, muchas veces no es tan fácil, ya que los robots de Google y otras herramientas que hacen estas mediciones lo hacen con ciertas configuraciones para simular a usuarios de perfil bajo, como por ejemplo móviles con 3G u ordenadores con baja velocidad de internet. Para ser realista, es muy complicado que todo el mundo tenga una experiencia de usuario perfecta, pero hay que intentar hacer todo lo que se pueda.

- **First Contentful Paint (FCP)** - Mide el momento en que el navegador renderiza el primer elemento de contenido en la pantalla.
- **Largest Contentful Paint (LCP)** - Mide el contenido más grande que se renderiza en la página dentro del display port del dispositivo. Es posible que el FCP y el LCP sean las dos métricas más difíciles de mejorar debido a que dependen mucho del dispositivo del usuario y en este caso son los dos cuellos de botella debido a que mis servidores son muy humildes. Se hablará más de optimización de imágenes en la siguiente sección.
- **INP (Input Latency)** - Se ha convertido en marzo de este año en una Core Web Vital, evaluando todas las latencias de click.
- **CLS (Cumulative Layout Shift)** - Es la considerada como más desesperante, midiendo la cantidad de saltos de elementos interactivos en pantalla.

- **TTFB (Time to First Byte)** - El tiempo hasta el primer byte en realidad no es una Web Vital, pero es muy importante porque afecta al resto de ellas.

Por lo general se ha probado por las distintas páginas de la web y se ha observado que la puntuación es muy buena y que por lo general el único factor limitante es que hay veces que la api tarda un poco más en responder debido a que los servidores en los que está desplegado son de perfil bajo y a que no hay server side rendering aplicado.

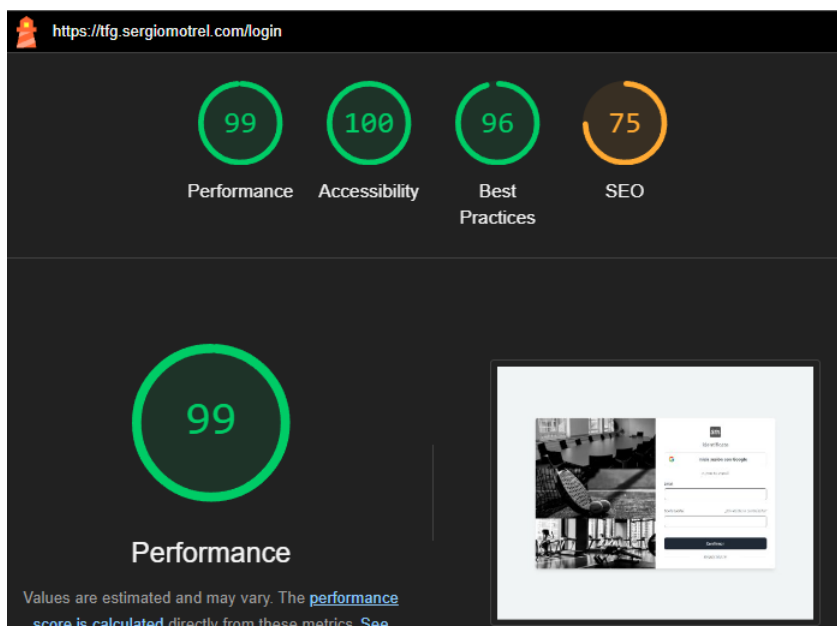


Figura 6.18: Puntuación lighthouse de una página de la web.

En la única parte de la web que soy consciente de que hay un problema de experiencia de usuario es en el formulario de registro, ya que al mostrar los errores de cada campo la aparición del error provoca un salto, lo que hace que sea frustrante. Este formulario es la primera pantalla que hice y tendría que retocarla en un futuro, simplemente cambiando que los espacios para los errores ocupasen el espacio de forma predeterminada estaría arreglado.

También, como explicaba antes, haciendo una simulación muy estricta sobre la conectividad de un posible usuario con 3g, se puede ver que la puntuación baja debido a que las imágenes tardan en descargarse, pero las mejoras para eso las veremos a continuación.

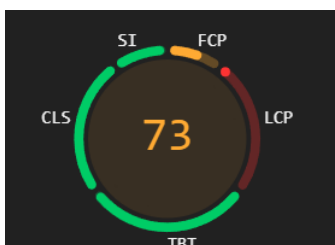


Figura 6.19: Puntuación baja debido al 3g.

6.7.5. Imágenes

Las imágenes de la web están tratadas con conversión a webP, un formato preparado y optimizado para la web que ya utilizan todas las webs modernas. Esta desarrollado por Google y se puede operar de dos formas distintas, "lossless"(sin pérdida) y "lossy"(con pérdida), adaptándose a distintas situaciones, aunque por lo general se opta por el formato lossy con la optimización justa para que la pérdida no sea notable pero reduciendo mucho el archivo . Al hacer esta mejora se puede reducir entre un 25 % y un 35 % el tamaño, dependiendo del color, las dimensiones, el formato de origen, etc...

Aunque parezca poca cosa, esto se nota mucho sobre todo en dispositivos móviles que cuentan con una conexión peor que un ordenador tradicional y, si se hace en toda la web, la velocidad de descarga de esas imágenes mejora mucho.

En el único sitio en el que esta conversión y compresión no está hecha es en las imágenes que suben los usuarios, aunque es una práctica muy aconsejable y que en el futuro tendría que realizar, ya que ahora mismo se permite que un usuario suba imágenes grandes.

Hay otros formatos de optimización como AVIF y webP2, pero a la fecha de la realización de este proyecto todavía hay algunos navegadores que no lo soportan. Se podría implementar una imagen en estos formatos con un fallback (respaldo de contingencia) para navegadores antiguos, pero es una cosa mínima y que para un proyecto de este carácter no merece la pena hacer. Para esta conversión y compresión de imágenes realizada incluso para la memoria se ha utilizado Squoosh [66] , una página muy conocida para esto.

Además, hay dos prácticas comunes que se ejecutan en el caso de las imágenes que por su coste se han considerado para este caso, pero merecen la pena de comentar para un futuro, estas son la utilización de un **CDN** (Content Delivery Network) y el ajuste de los tamaños de las imágenes a su contenedor.

El ajuste de los tamaños puede parecer obvio, y se refiere a que si en un contenedor de 400px haces referencia a una imagen 1080px, estarías cargandola innecesariamente grande, ya que el navegador lo que está haciendo es descargar la imagen y adaptarla al contenedor sin realizar ninguna corrección así que el tamaño del archivo es completo. Por esto, merece la pena utilizar imágenes del tamaño correcto para cada contenedor. Para el diseño responsivo en el que los contenedores varían de tamaño según el dispositivo, se pueden ejecutar estas soluciones:

- Sistema de recorte y modificación de imágenes .“al vuelo”self-hosted en el propio servidor de producción donde se almacenan las imágenes como por ejemplo ImgProxy, picfit o Thumbor. Estas herramientas dependen mucho de la potencia y conexión del servidor que las esté ejecutando, ya que es crucial que la conversión sea lo más rápida posible.
- Utilizar distintas imágenes y aplicar unas u otras dependiendo del dispositivo. Esto se consigue con la propiedad `srcset` de la etiqueta `img`, a la que se puede acompañar de la etiqueta `sizes` para un mejor comportamiento. Estas imágenes se generarían previamente, es decir, en la carpeta `assets` o donde fuese necesario habría cuatro o cinco imágenes de distintos tamaños por cada imagen original y luego se cargaría la que mejor se ajustase. Por ejemplo, si tuviésemos originalmente `image.png`, optimizaríamos esta imagen a `webp` y generaríamos `image320.webp`, `image480.webp`, `image700.webp` y `image1000.webp` y luego el navegador elegiría la imagen correcta.

6.7.6. Tiempo de bloqueo

En herramientas de medición de rendimiento y experiencia de usuario es común ver como consejo de rendimiento la reducción de bloqueo de JavaScript o “blocking time”. Esto se debe a que el navegador ejecuta el código de javascript de forma secuencial y por tanto mientras esta ejecutando lógicas pesadas, el navegador no puede trabajar en otras cosas, haciendo que muchos de los recursos que necesitamos para cargar los contenidos más importantes se ven retrasados por el bloqueo. Esto en una aplicación muy grande, con muchos componentes y mucha lógica detrás es un problema ya que las imágenes que necesitamos o mismamente el componente que lleva el contenedor de la imagen puede retrasar su carga hasta que el resto de componentes hayan hecho todo lo necesario.

Para arreglar este problema tenemos varias técnicas y buenas prácticas: Detección de cambios

Deferrable views: Desde Angular 17, se ha introducido una nueva sintaxis para la parte del `template(html)` y algunas funcionalidades. Entre ellas están las Deferrable Views o vistas diferidas, que lo que hacen es aplazar la carga de su contenido según algunas opciones que podemos elegir. Hay bastantes opciones, pero las que más destaco para lo que yo busco son *on viewport* y *on idle*. El primero aplaza la carga hasta que el bloque de código se muestra en el viewport del navegador, haciendo que todos los componentes que no son esenciales solo carguen si así fuese necesario, reduciendo mucho la carga de la web. *On idle* es un concepto similar, solo que el aplazamiento se produce hasta que el navegador está en un estado de reposo. Este último de por sí no evitaría ejecución de código, pero sí que podemos hacer que lo que no sea esencial tenga una menor prioridad.

Como tal, esto solo evitaría renderizado de html y no reduciría el bloqueo por javascript, pero teniendo en cuenta que dentro del `defer` podemos poner componentes Angular enteros, la mejora en rendimiento es más que notable, haciendo que toda la inicialización del componente y su detección de cambios se aplace o incluso se evite. Como estrategia general, se ha aplicado *on viewport* en todo lo que estaría *under the fold* (comúnmente llamado a los elementos que no se ven en el viewport nada más cargar la web). Para la utilización de esta estrategia, el framework te obliga a implementar la etiqueta `@placeholder` y añadir un `div` del elemento que

quieres que muestre hasta que se cumpla la condición del `defer`, solucionando así un posible problema de *layout shift* debido al salto que habría si no hubiese nada y de pronto apareciese el componente.

- **Máxima reutilización del código:** Si se reutilizan componentes, mucha de la lógica no tendrá que volver a ejecutarse.
- **Lazy loading de componentes y módulos:** Angular nos brinda la herramienta perfecta para reducir la carga de javascript.
- **Tree shaking:** que es un proceso que ejecuta el compilador al generar la build de producción, eliminando el código repetido y muerto y liberando un poco el bloqueo producido, ya sea porque el fichero `.js` ahora ocupa menos como al haber menos código.
- **Code splitting:** Una herramienta muy poderosa que nos permite hacer carga perezosa de módulos enteros y componentes. Por ejemplo si sabemos que un componente no es parte de la portada de la web, no tendría sentido cargarlo ni tener que descargarlo. Esto lo hace el compilador de Angular a través del enrutador interno, mirando que rutas usan que componentes y con esa información, separando el código en distintos trozos `.js` que luego solo se ejecutan en las páginas que sean necesarias.

Aquí quiero introducir el concepto de **bundle size**, que es el tamaño total del javascript de la compilación generada. Por ejemplo, si el trozo principal (`main.js`) ocupa medio MB y los trozos separados con `code splitting` suman 500kb, el bundle size sería de 1,5MB. Para aplicaciones medianas-grandes, el bundle size suele ser de entre 300kb y 1MB. Si lo pensamos, 1MB para una conexión actual no es nada, pero si pensamos en que la web también se ejecuta en un móvil, esto puede ser excesivo, por lo que hacer `code splitting` es extremadamente importante.

- **Utilización de Web Workers:** Los Web Workers (o trabajadores web), realizan funciones fuera del hilo principal, descargando a este de mucho trabajo y bloqueo. Aunque es una técnica avanzada, se puede utilizar en sitios donde se ejecute una lógica muy demandante, como por ejemplo cálculos y filtrado de tablas. El Web Worker ejecutaría el código en un hilo paralelo y luego le comunicaría la información a Angular, que solo tendría que mostrarla.

Capítulo 7

Despliegue

7.1. Configuración de dominio

Como dominio de todo el ecosistema de la web se ha utilizado el proveedor de servicios **Cloudflare**, mundialmente conocido por dar uno de los mejores servicios en cuanto a DNS y dominios. Este dominio ya estaba en propiedad con anterioridad, así que solo se tuvo que configurarle para los microservicios de la web.

Una vez identificado en la web del proveedor, hay que elegir el dominio en el que se quieren realizar las modificaciones, que en este caso el dominio es **sergiomotrel.com**. Como se puede ver en la figura 7.1, figura en una lista de mis propiedades.

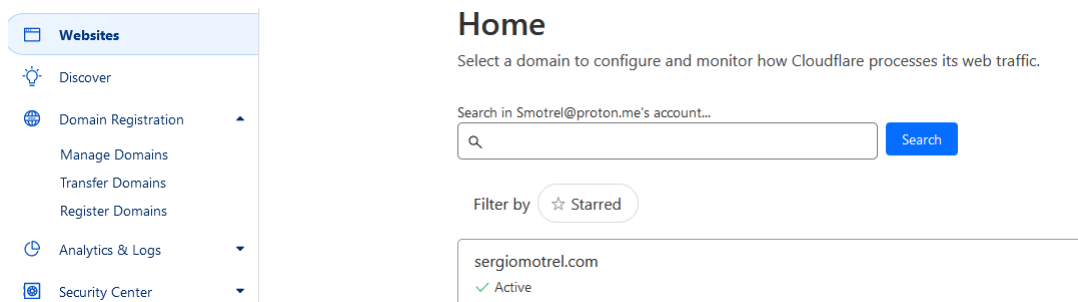


Figura 7.1: Dominios registrados en Cloudflare.

Una vez seleccionado el dominio correspondiente, se registraron los subdominios necesarios para el despliegue, que se eligieron **tfg** para el frontend y **users**, **reserves** y **community** para las APIs. Para ello, hay que hacer una redirección de DNS con el nombre de los subdominios de tipo A de **address** (*dirección*) a la ip de los servidores. En la figura 7.2 se pueden ver las cuatro redirecciones con la ip borrada por seguridad. Además de la redirección, se

puede apreciar que está seleccionada la opción de **proxy**, y es que Cloudflare ofrece un caché muy potente a través del mismo, el cual viene configurado de serie si se activa esta opción. Esto es muy útil para que los archivos estáticos y los archivos js resultantes de la compilación se sirvan de forma más rápida.





Type ▲	Name	Content	Proxy status
A	community	51.75	 Proxied
A	reserves	51.68	 Proxied
A	tfg	51.68	 Proxied
A	users	51.75	 Proxied

Figura 7.2: Redirecciones DNS en Cloudflare

Por último, ya que el proveedor firma certificados SSL de forma gratuita con la propiedad del dominio, se han utilizado para servir la web a través de **https**. Para ello, se configuraron los certificados SSL para el dominio `sergiomotrel.com` y para `*.sergiomotrel.com`, es decir, todos sus subdominios. Estos certificados se verán más en profundidad en la sección siguiente a la hora del despliegue.

7.2. Configuración de servidores

Previo a empezar con el despliegue, hubo que configurar los dos servidores (se realizaron los mismos pasos en los dos servidores) del proveedor OVH que se mencionaron anteriormente. Para desplegar el proyecto, solo es necesario instalar **Docker** y algún tipo de proxy inverso. El proxy inverso elegido ha sido **Nginx** por una cuestión de comodidad. También se podía haber desarrollado el entorno de contenedores docker para que Nginx fuese aplicado como un contenedor, sin embarbo, dado que la forma en la que se despliegue el proyecto depende mucho de las necesidades, se ha dejado a libre elección el proxy inverso a elegir y el número de proxys inversos. Esta configuración inicial y todo el proceso de despliegue se puede ver paso por paso en el anexo [B.1](#), pero se explicará más en detalle a continuación.

Como primer paso, ya que los servidores venían de fábrica, se hizo una actualización limpia del repositorio de paquetes de Ubuntu y se actualizaron los paquetes instalados, esto es importante realizarlo en todo servidor Linux de fábrica para asegurarse de que se tiene todo en las últimas versiones estables.

Luego, la siguiente instalación que se hizo para esta configuración es instalar Docker para Ubuntu 22.04. Docker no tiene un paquete directo en Ubuntu con el que realizar la instalación, así que siguiendo la documentación de DigitalOcean (otro proveedor de servicios) se realizaron todos los pasos necesarios para descargar los archivos de origen y añadirlos a nuestro repositorio de software Linux y que así el paquete de docker estuviese disponible para la instalación.

Después de la instalación limpia de Docker y de comprobar que funcionase, se instaló y configuró Nginx, que si que dispone de paquete propio para Ubuntu y se realizaron configuraciones en ambos Nginx para redireccionar el tráfico entrante por los subdominios a los servicios adecuados, para ello, antes de generar los ficheros de configuración de Nginx, se tuvieron que borrar los ajustes predeterminados y añadir las claves de los certificados SSL aportados por CloudFlare en la sección anterior, para después importar la configuración básica de Nginx que se puede ver en las figuras B.1 y B.2, al cual se le cambio *domain* por sergiomotrel.com y la localización de los archivos SSL. La configuración hace lo siguiente:

- Se escucha por el puerto 80(http) y 443(https) y redirigir el tráfico del puerto 80 al 443 para forzar la utilización de https.
- En el primer servidor:
 - Redirección de todo el tráfico entrante por **tfg.sergiomotrel.com** a la carpeta donde estará alojado el frontend Angular.
 - Redirección de todo el tráfico entrante por **reserves.sergiomotrel.com** al puerto 8081, en el que opera Slim Framework con la API de reservas.
- En el segundo servidor:
 - Redirección de todo el tráfico entrante por **users.sergiomotrel.com** al puerto 3000, en el que opera NestJs con la API de usuarios.
 - Redirección de todo el tráfico entrante por **community.sergiomotrel.com** al puerto 8081, en el que opera Spring Boot con la API de comunidades.

Ahora Nginx ya estaba funcionando y redirigía el tráfico a los servicios necesarios, pero faltaba desplegar cada uno de los servicios en los puertos correspondientes. Para ello, en la raíz del proyecto se encuentran los archivos **docker-compose.prod1.yml** y **docker-compose.prod2.yml** con la separación de los servicios que tenían que ir en cada uno de los servidores y los puertos correspondientes.

Para descargar este proyecto en los servidores, hubo que clonar el repositorio con **git clone**, el cual pidió una clave GPG de GitHub que se puede encontrar bajo la configuración de la cuenta en la página web de GitHub.

Una vez hecho esto, antes de poner en funcionamiento los contenedores Docker, hay que configurar todas las variables de entorno, ya que las mismas no están repositadas por seguridad. Para hacer esto, hay que realizar una copia de los siguientes ficheros, eliminar la extensión *.template* y rellenar los datos y credenciales de cada uno, los cuales no expondré aquí por seguridad:

- /angular-front/.env.template
- /microservicio-reservas/.env.template
- /microservicio-comunidades/.env.template

- /microservicio-usuarios/.env.template

Además de credenciales, hay que poner ciertos datos como entornos de ejecución a producción, rellenar las urls donde estarán ciertos servicios y cambiar el modo de las cookies a **secure**.

Con esto hecho, solo quedaba ejecutar ambos archivos de docker compose para desplegar los servicios y asegurarse de que todo estaba funcionando.

```
docker compose -d -f docker-compose.prod1.yml up --build
```

```
docker compose -d -f docker-compose.prod2.yml up --build
```

Esto desplegó los contenedores del frontend y la API reservas en el servidor 1 y la API usuarios y la API comunidades en el servidor 2.

Además de todo esto, utilicé la herramienta Filldb.info [67] para rellenar las bases de datos de información *dummy* (o falsa) para la realización de pruebas. Con la herramienta se poblaron todos los datos menos las imágenes, ya que tiene la limitación de no generarlas, por lo que se realizaron las inserciones de las imágenes utilizando la web y así, de paso, realizando pruebas de todos los formularios y funciones.

Todo este proceso parece sencillo, pero la orquestación necesaria para realizar todo esto al principio no fue fácil y se encontraron algunos problemas iniciales con las cookies resultando en logins incorrectos que se arreglaron una vez se detectó que se estaban aplicando mal las propiedades **httpOnly** y **secure**. Por lo demás, el proceso de despliegue no fue demasiado complicado.

Capítulo 8

Resumen de la realización del proyecto

8.1. Problemas

A la hora de ejecutar el proyecto, se han cometido algunos errores por desconocimiento o por aplicar mal algunos conceptos o prácticas, aunque muchos de ellos se han podido solventar, es de agradecer haberse enfrentado a ellos ya que en un futuro no se volverán a cometer y han aportado una valiosa experiencia, la siguiente lista engloba los más destacados con una breve explicación de cada uno.

- **.env repositado** - Desde el principio se ha tenido muy en cuenta no dejar nada en texto claro ni tener credenciales o datos sensibles en ninguna parte del proyecto, pero por lo visto en algún momento se creó un `.env` que no se añadió a `.gitignore`, con lo que sin quererlo se acabó repositando.

Aunque parece poco grave, el problema es que en muchas ocasiones se puede filtrar la carpeta `.git` del proyecto, dejando expuesto el historial de commits, por lo que se podrían ver los cambios que se realizaron en ese `.env`, haciendo que las contraseñas y credenciales que tenía dejen de ser seguras. Para solucionarlo, se ejecutó el comando `git rm -cache .env` que quita el `.env` del repositorio sin eliminar el archivo local, dejando solo rastro en el historial del repositorio.

Para eliminarlo del historial bastó con ejecutar el siguiente comando, el cual elimina de todas las ramas cualquier rastro del mismo:

```
git filter-branch -force -index-filter "git rm -cached -ignore-unmatch .env" -prune-empty -tag-name-filter cat - -all
```

- **Mala implementación inicial de las APIs** - Cuando se empezó este proyecto, había una idea bastante correcta de como implementar la API y que patrones utilizar (Controller, servicios de aplicación, dominio, etc...). Sin embargo, con el tiempo se fue

viendo que no se conocía bien de donde venía todo esto ni de las bases, por lo que cuando se estudió la arquitectura hexagonal, se vio que estaba aplicando algunas cosas incorrectamente. Para arreglarlo, prácticamente se tuvo que rehacer dos APIs, no por completo, pero sí que hubo que reestructurar todo el sistema de carpetas, renombrar muchas funciones y clases y separar muchas responsabilidades (por ejemplo en los servicios) para cumplir mejor con el SRP (Principio de responsabilidad única) de SOLID.

También, no había un buen entendimiento del poder de **DIP** (Principio de inversión de dependencias) ni como podía beneficiar al testing, por lo que también hubo que cambiar en varios sitios las implementaciones y aplicar interfaces en su lugar. Todo esto llevó bastante trabajo y se perdió mucho tiempo del total del proyecto evitando poder enfocarse más en otras cosas, como en el testing o tener que dejar para el futuro el sistema de incidencias, pero creo que al final ha aportado un enfoque muy útil para el futuro, entendiendo mucho mejor las arquitecturas por capas, el uso de interfaces y como estructurar de una forma mucho más limpia y mantenible un proyecto, así que creo que ha merecido la pena rehacerlo.

- **Docker** - El potencial de Docker está claro y visto en perspectiva, es muy útil, pero las complicaciones que ha habido en este proyecto con esta tecnología, han ralentizado mucho. Incluso sabiendo que es algo que se tenía que hacer desde el principio y llevando los docker-compose y Dockerfile al día y a rajatabla, las versiones de Docker que tenía Windows 11 en los primeros dos tercios del desarrollo eran muy inestables, haciendo que incluso se tuviese que reiniciar de fábrica el ordenador en el que se realizó el proyecto debido a corrupciones de registro y la imposibilidad siquiera de iniciar Docker Desktop. Por fortuna, las versiones actuales ya no tienen nada que ver y funcionan a la perfección, pero sin duda este ha sido una de las cosas que más tiempo han quitado y que, además, ha sido un problema muy injusto, porque por mucho que se intentara resolver, no estaba en un problema de implementación sino del propio programa.
- **Git** - No ha sido un problema como tal, pero no se ha dado un uso profesional. Se podría haber elegido alguno de los flujos de trabajo como GitFlow o GitHubFlow, montar un entorno de preproducción y hacer las cosas mucho más profesionales con un ciclo de DevOps y Integración continua/Despliegue continuo (CI/CD), pero habiendo solo una persona en el proyecto no se sintió la necesidad de hacerlo y para ha sido más fácil simplemente utilizar una o dos ramas.
- **Elección de tecnologías** - Además de querer utilizar diversas tecnologías para comprobar que la arquitectura en microservicios da como ventaja poder desarrollar con más libertad, se buscaba la idea de aprender diversos frameworks de APIs y lenguajes de programación ya que interesaba tener conocimiento de los mismos. Desde luego, se ha aprendido bastante de las tecnologías utilizadas y ha aportado muchísimo valor a la hora de tener las ideas más estructuradas sobre todo en cuanto a backend.

Pero siendo realista, si se hubiese hecho todo en la misma tecnología el desarrollo habría sido más rápido. Todas las APIs comparten arquitectura y en ese aspecto la experiencia de comprobar que una vez conoces una arquitectura el framework solo hace variar la forma en la que se implementa debido a los pequeños detalles ha sido interesante. Son estos detalles los que han restado tiempo, pero hay que pensar que la causa real es

la falta de *expertise* técnica y, aun así, ha sido gratificante porque siempre está bien informarse de nuevas herramientas y tecnologías.

8.2. Riesgos materializados

Se van a englobar todos los riesgos materializados, ya que los cuatro han sido principalmente por el mismo motivo, y los riesgos han sido **RSK-1** (Optimismo), **RSK-2** (Mala estimación), **RSK-3** (Retras) y **RSK-4** (Abandono del proyecto).

Cuando hice la evaluación de riesgos iniciales del proyecto, no pensaba que pudiesen ocurrir, pero el riesgo de desmotivarse y semi-abandonar el proyecto se acabó convirtiendo en una realidad. Y no es porque el proyecto o las tecnologías no llamasen la atención, sino porque a la vez que se empezó el proyecto tuve la oportunidad de entrar en un puesto de desarrollador fullstack centrado en Angular. Este nuevo trabajo limitó mucho mi tiempo y prácticamente todo el tiempo libre que he tenido en este último año lo he dedicado a estudiar algunos ámbitos del desarrollo web dejando un poco más de lado el proyecto.

Además, la experiencia laboral me ha proporcionado una perspectiva práctica que no habría obtenido únicamente con el desarrollo del proyecto. El trabajo diario con Angular, PHP, NestJS y otras tecnologías han consolidado mis habilidades y me ha permitido aplicar soluciones más eficientes y efectivas al proyecto. Esta experiencia ha enriquecido tanto mi formación profesional como la calidad del proyecto en sí.

Es muy fácil caer en pensar que todo va a funcionar a la primera en un proyecto e intentar abarcar demasiado. En este caso es lo que ha pasado varias veces, de la idea inicial a la planteada para este trabajo ya se avisó por parte de mi tutora que era demasiado grande y dio una guía de cuanto acabar abarcando y se eligió añadir un poco más que parecía pequeño. Pues claramente se fue optimista y por ello se ha tenido que dejar para futuro toda la funcionalidad de las incidencias y de la API Gateway, ya que se sobrepasó el tiempo límite de horas del proyecto.

Sin embargo, no hay arrepentimiento, ya que todo lo aprendido en el trabajo y en el estudio ha valido para volver al proyecto y reconocer todos los errores de la lista de este capítulo y, además, despuntar en mi lugar de trabajo. Por esto, aunque hay que reconocer que el desarrollo del proyecto se ha alargado innecesariamente, creo que se ha tomado la decisión correcta.

Por otro lado, el desafío de gestionar el tiempo entre el trabajo y el proyecto académico ha enseñado lecciones valiosas sobre la gestión del tiempo y la priorización de tareas. Aunque al principio se subestimara el impacto de asumir ambos compromisos, esta experiencia ha dado una preparación mejor para enfrentar situaciones similares en el futuro, aunque hay que reconocer que la planificación se podría haber hecho mejor.

Capítulo 9

Conclusiones y trabajo futuro

9.1. Conclusiones

A la finalización del Trabajo Final de Grado se puede concluir que el producto es funcional y con pocos retoques y adiciones que se comentarán más tarde podría ser un producto funcional y utilizable por usuarios reales.

Las horas de investigación sobre todas las tecnologías y prácticas utilizadas han dado sus frutos y al final y pese a todos los problemas encontrados y que el proyecto se haya alargado, el proyecto ha salido adelante y se han conseguido muchos objetivos que al principio parecían difíciles.

Algunos objetivos relacionados con el proyecto que se han conseguido:

- Se ha conseguido completar un producto mínimo utilizable que implementa gestión de reservas, de usuarios y de comunidades.
- Se ha conseguido una interfaz fácil de utilizar y accesible para multitud de usuarios
- Se ha conseguido un diseño responsivo utilizable en distintos dispositivos.
- Se ha conseguido la coordinación de distintas tecnologías
- Se ha conseguido una arquitectura de microservicios funcional y con una buena comunicación entre los mismos.
- Se ha conseguido aplicar buenas prácticas de seguridad, buena arquitectura y patrones y prácticas modernas.
- Se ha conseguido un despliegue con contenedores Docker que aporta un traspaso sencillo de servicios.

- Se ha conseguido un despliegue real en más de un servidor y la comunicación entre ambos, además de una protección básica de ambos.

Objetivos personales conseguidos en este trabajo:

- Aprendizaje de tres frameworks backend distintos, de sus diferencias y de como implementar APIs funcionales en ellos.
- Aprendizaje superficial del uso básico de tres tipos de bases de datos.
- Aprendizaje de como aplicar la Arquitectura Hexagonal.
- Una profundización de Angular aplicada a un proyecto que requiere un conocimiento más completo del mismo.
- Se ha aprendido como coordinar los microservicios, algo que era una carencia antes de empezar este proyecto.
- Profundización en Docker, comunicación entre contenedores y despliegue de los mismos.
- Aprendizaje de cómo configurar servidores desde cero y como acabar desplegando en ellos.

Por lo general, el objetivo de conseguir una perspectiva más global del desarrollo web se ha cumplido y ha sido todo un éxito, ya que antes de empezar el proyecto no se conocía en profundidad el backend y mucho menos la parte de administración de servidores.

En general, la exhaustiva investigación de diversas tecnologías y prácticas que ha necesitado este proyecto ha satisfecho todas las aspiraciones personales de aprendizaje y los objetivos iniciales de como tenía que ser el producto final.

Al final, uno de los objetivos principales era conocer muchas herramientas asentadas en el sector y tenerlas a la mano en el futuro de mi carrera profesional, y es algo que se ha cumplido con creces.

9.2. Trabajo futuro

En esta sección se pasa a detallar algunas mejoras que se pueden aplicar en el futuro en caso de ampliar la aplicación.

9.2.1. SEO

Aunque es cierto que la web esta casi toda protegida por el login y por tanto el SEO solo afectaría para la parte no protegida, la intención es hacer una demo para demostrar el producto que si que se verá afectada y una landing page. Un resumen de las prácticas que estaría bien implementar a futuro al respecto son:

- **Metadata:** Etiquetas HTML para dar contexto a los bots.
- **Etiquetas básicas SEO:** Aplicación de las etiqueta básicas HTML de título[68], descripción[69], viewport[70], canonical[71] y autor de forma dinámica al cambiar de ruta.
- **Datos estructurados:** Los datos estructurados de tipo Schema son un estándar que ha adoptado Google para que los bots entiendan mejor como navegar las webs.
- **Sitemap:** Archivo que se incluye en una ruta de la aplicación con el objetivo de facilitar la navegación de los bots, indicando un mapa de como está la web estructurada.
- **Robots.txt:** Archivo de texto plano que se suele ubicar en la raíz de un sitio web y se utiliza para prohibir rutas a los bots que no interesen.
- **Opengraph:** Inclusión de etiquetas para que las redes sociales (Facebook y Twitter) incrusten los enlaces de forma más atractiva.

9.2.2. Patrón Criteria

Como se comentó en la sección 5.3.3 relativa al patrón repositorio, el patrón **Criteria** se utiliza muy a menudo junto con el Repositorio para realizar consultas flexibles y dinámicas sobre datos. Este patrón permite construir consultas complejas sin necesidad de modificar la implementación del repositorio. Es decir, que si por ejemplo hay un User del que se necesita encontrar todos los usuarios que vivan en Valladolid y en otra consulta se necesitan todos los usuarios que sean administradores, no haría falta construir dos consultas y dos métodos en el repositorio, ya que a través del patrón Criteria se podrían hacer ambas. Ventajas del Patrón Criteria

Ventajas

- **Flexibilidad en las consultas:** Permite construir consultas complejas de manera dinámica.
- **Reutilización de código:** Se reutilizaría el método que implemente el patrón Criteria para todas las consultas de búsqueda independientemente de sus requisitos.
- **Desacoplamiento de la lógica:** El funcionamiento del patrón Criteria se mantendría separado del repositorio, eso si es que el propio framework no trae ya una implementación del mismo

El uso conjunto de los patrones Repositorio y Criteria permite una organización del código y una escalabilidad a otro nivel, por ejemplo se conoce que Amazon utiliza este patrón para el filtrado de sus productos, solo imaginar la de consultas distintas para todas las combinaciones de su filtro que harían falta para este patrón, sería simplemente abrumante. Por ello, si se sigue ampliando el proyecto sería muy beneficioso hacer una migración de las APIs a que sus repositorios tuviesen un crud básico y un método *filter* que implementase este patrón.

9.2.3. Implementación de incidencias

Ya se ha comentado en la sección 8.2 que por falta de tiempo no se pudo implementar la parte de incidencias. Se implementaría de forma muy similar a las reservas, estando en el mismo servidor y en la misma API (en PHP). La arquitectura y patrones son los mismos, así que tampoco llevaría mucho más tiempo.

Paralelamente, también se crearían los formularios pertinentes en Angular y la gestión de las mismas para todos los tipos de usuarios. Es decir, se implementarían los requisitos comentados en la sección 4.2 referentes a las reservas.

9.2.4. Añadido de API Gateway

Otra de las cosas que se ha quedado pendiente es utilizar una API Gateway, que mejora la comunicación y gestión de los microservicios en una arquitectura moderna. La elegida sería Kong, y actúa como un punto de entrada único para todas las peticiones de clientes hacia los microservicios, simplificando la interacción y reduciendo la complejidad del sistema. Esta capa adicional dirige el tráfico a los microservicios correctos sin que el cliente tenga que saber hacia que servicio apuntar, pero también maneja tareas transversales como autenticación, autorización, *rate limiting*, análisis y monitoreo. Al centralizar estas funciones, Kong permitiría escalar más rápido e incluso separar los microservicios en distintos dominios si fuese necesario.

Además, ofrece una infraestructura que facilita la integración continua y el despliegue automatizado de microservicios, además del despliegue **Canary**, que es un tipo de despliegue progresivo para detectar errores tempranos. Puede también manejar grandes volúmenes de tráfico y su flexibilidad para integrarse con múltiples servicios de backend hacen que sea una opción ideal para entornos de producción que requieren alta disponibilidad y rendimiento.

Añadido a esto se podría también añadir plugins de kong para comunicación con Prometheus(un sistema de monitoreo) y Grafana(un sistema de gráficas), pero eso ya es para un futuro más lejano

9.2.5. Renderizado en el lado del servidor(SSR)

El **SSR** es un proceso que en la actualidad es un estándar en la web. En el pasado, muchos lenguajes eran directamente procesados en el servidor, como PHP, pero actualmente la mayoría de frameworks con más cuota de mercado están basados en javascript, que como ya se sabe, se ejecuta en el lado del cliente. Esto unido a que las webs cada vez son más demandantes, debido a las necesidades complejas de algunas aplicaciones web y la reactividad, hace que muchos dispositivos de gama baja o un poco antiguos no sean capaces en muchas ocasiones de ejecutarlo en tiempos aceptables.

Para ello, casi todos los frameworks basados en javascript dan la opción de aplicar un renderizado en el lado del servidor, que lo que hace es ejecutar todos los scripts necesarios en

el servidor para luego mandarle esta información al cliente y lo único que tenga que hacer es aplicar el html y los estilos, que son mucho menos demandantes. En Angular existe un paquete llamado Angular Universal que gestiona a través de un servidor Express en Node.js toda la ejecución y luego el proceso de paso de información se realiza a través de la **hidratación**. El proceso es el siguiente, el servidor ejecuta el código javascript en un DOM virtual, después, le manda el DOM virtual junto con el estado de la aplicación al cliente, el cual "hidrata" el DOM básico que ha creado con el html y el css y lo deja en el estado final.

Todo esto, por supuesto, añade complejidad a la aplicación, haciendo que tenga que haber un despliegue extra de la parte Express que además utiliza recursos del servidor, teniendo que mantener esta parte y asegurarse de que funcione correctamente. Sin embargo, las mejoras son increíbles, la web cargaría visualmente instantáneamente ya que cuando el cliente recupera los datos del servidor la web ya está cargada, y además seguiríamos teniendo la funcionalidad y la rapidez de una SPA. También las mejoras en SEO son muy beneficiosas, ya que todas las métricas comentadas anteriormente serían mucho más rápidas, añadidas a que los bots de Google por fin tendrían una versión de la web sin código que ejecutar. Lo único que tendría que hacer el navegador cliente es descargar la información, las imágenes y renderizar estas.

La adición del SSR es una forma de dar una experiencia inmejorable al cliente y es algo que una vez se sabe como funciona no supone un desarrollo ni un esfuerzo extra.

9.2.6. Integración y despliegue continuos

La **Integración Continua** (CI) y el **Despliegue Continuo** (CD) son prácticas que en el sector del desarrollo están a la orden del día para mejorar la eficiencia y la calidad del proceso de desarrollo, haciendo el despliegue y la forma de trabajar más ágil. CI implica la integración frecuente de cambios de código en un repositorio compartido, seguido de la ejecución automática de pruebas para detectar errores de manera temprana. CD amplía CI al automatizar el despliegue de cambios de código a entornos de producción.

Aprovechando que el repositorio está en GitHub y que ofrece una herramienta CI/CD gratuita para repositorios personales llamada **GitHub Actions**, podría automatizar, personalizar y ejecutar flujos de trabajo como el compilado de Angular y el despliegue de los contenedores para agilizar aún más el desarrollo y preocuparse menos por los despliegues.

También se podrían utilizar herramientas similares open-source como Jenkins o OpenShift, que son muy interesantes y completas y permiten la ejecución de scripts de CI/CD a través de distintos servidores y contenedores.

Apéndice A

Manual de arranque en local

A.1. Requisitos

- NodeJS 16.14 o superior[72].
- Instalación de Angular CLI 16 o superior.
- Docker Desktop en Windows (necesita WSL 2) o Docker en MacOS/Linux.

A.2. Clonado del repositorio

Desgarga del repositorio Git al servidor, Git pedirá la clave **GPG** de **GitHub** asociada a una cuenta con permisos:

```
git clone https://github.com/smotrel/sm-community-tfg
```

Una vez el repositorio esté clonado, las plantillas de variables globales (**.env.template**) de cada servicio deben copiarse y renombrarse a **.env** y se deben rellenar estos últimos con las credenciales que se prefiera que se creen los servicios y las urls correspondientes de los servicios. **Es importante que estas credenciales no se introduzcan en la plantilla y que no se compartan.** Estos archivos se encuentran en :

- /angular-front/.env.template
- /microservicio-reservas/.env.template
- /microservicio-comunidades/.env.template
- /microservicio-usuarios/.env.template

A.3. Despliegue de contenedores

Una vez todos los datos estén listos, solo queda desplegar los contenedores, para desplegar los contenedores Docker hay que lanzar el siguiente comando:

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

Esto desplegará los contenedores de todos los backend con sus respectivas bases de datos.

A.4. Ejecución de Angular

Se ha decidido que la ejecución de Angular en local se la clásica y recomendada por Angular, ya que si se introduce en un contenedor se pierde el llamado **Hot Module Replacement** o **Watch Mode**, que recarga al instante el servido de la aplicación en local para ver los cambios al instante. Para realizar la ejecución, solo hay que ponerse en la carpeta del frontend (/angular-front) y ejecutar uno de los siguientes comandos:

```
ng serve
```

```
npm run build
```

Apéndice B

Manual de despliegue en producción

B.1. Requisitos

- Uno o más servidores con Ubuntu 22+. (En este manual se utilizarán dos)
- Un dominio y certificados SSL correspondientes.
- Software de subida de archivos FTP. (También se puede utilizar el comando **scp**)
- Tener el tráfico redireccionado de tu dominio a los subdominios que se pretendan utilizar a la ip de los servidores.

B.2. Configuración previa de los servidores

(Los pasos siguientes se realizarán en cada uno de los servidores)

Actualización limpia del repositorio de paquetes de Ubuntu y actualización de los paquetes instalados:

```
sudo apt-get update
```

```
sudo apt-get upgrade
```

Descarga de orígenes de paquetes oficiales de Docker:

B.2. CONFIGURACIÓN PREVIA DE LOS SERVIDORES

1. Instalación de dependencias necesarias, tanto para la descarga de Docker como para la posterior aplicación de la clave *GPG*, una clave pública criptográfica necesaria para ciertas operaciones.

```
sudo apt install apt-transport-https ca-certificates curl
software-properties-common
```

2. Descarga y aplicación de la clave pública *GPG* oficial de Docker para Ubuntu.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

3. Descarga y aplicación de los listados de archivos fuente de Docker en el llavero de Ubuntu.

```
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs)
stable" | sudo tee /etc/apt/sources.list.d/docker.list >
/dev/null
```

4. Actualización del repositorio de software de Ubuntu para aplicar los cambios y los nuevos orígenes.

```
sudo apt update
```

5. Descarga e instalación del paquete de Docker.

```
sudo apt install docker-ce
```

Instalación de Nginx:

```
sudo apt install nginx
```

Creación de estructura de carpetas y subida de certificados SSL para Nginx:

1. Borrado de los archivos de configuración predeterminados que se crean con la instalación de Nginx.

```
rm /etc/nginx/sites-available/default && rm
/etc/nginx/sites-enabled/default
```

2. Creación de la carpeta en la que irán los certificados SSL.

```
mkdir /etc/nginx/ssl
```

B.2. CONFIGURACIÓN PREVIA DE LOS SERVIDORES

3. Se subieron los archivos al servidor a través de FTP con el programa Termius, un fichero **sergiomotrel.crt** y otro **sergiomotrel.key**.
4. Aplicación de permisos de solo lectura para dueño y grupo de la carpeta y los certificados y cambio de dueño de los mismos al usuario de Nginx.

```
sudo chown www:www /etc/nginx/ssl && sudo chown www:www
/etc/nginx/ssl/*
sudo chmod 440 /etc/nginx/ssl && sudo chmod 440 /etc/nginx/ssl/*
```

5. Creación del nuevo archivo de configuración de Nginx y de un link simbólico en la carpeta sites-enabled.

```
touch /etc/nginx/sites-available/sergiomotrel.com
ln -s /etc/nginx/sites-available/sergiomotrel.com
/etc/nginx/sites-enabled/
```

Realizar la configuración de nginx con el código de la siguiente sección.

B.3. Código de configuración de Nginx

```
server {
    listen 80;
    server_name frontend.domain.com;
    return 301 https://$host$request_uri;
}
server {
    listen 443 ssl;
    server_name frontend.domain.com;
    ssl_certificate /etc/nginx/ssl/domain.crt;
    ssl_certificate_key /etc/nginx/ssl/domain.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        root
            /var/sm-community-tfg/angular-front/dist/angular-front;
        index index.html;
        try_files $uri $uri/ /index.html;
    }
}
server {
    listen 80;
    server_name reserves.domain.com;
    return 301 https://$host$request_uri; # Redireccionar todo el
        tráfico HTTP a HTTPS
}
server {
    listen 443 ssl;
    server_name reserves.domain.com;
    ssl_certificate /etc/nginx/ssl/domain.crt;
    ssl_certificate_key /etc/nginx/ssl/domain.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        proxy_pass http://localhost:8081;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Figura B.1: Archivo de configuración Nginx, **Servidor 1**.

```
server {
    listen 80;
    server_name users.domain.com;
    return 301 https://$host$request_uri; # Redireccionar todo el
        tráfico HTTP a HTTPS
}
server {
    listen 443 ssl;
    server_name users.domain.com;
    ssl_certificate /etc/nginx/ssl/domain.crt;
    ssl_certificate_key /etc/nginx/ssl/domain.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        proxy_pass http://localhost:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
server {
    listen 80;
    server_name community.domain.com;
    return 301 https://$host$request_uri; # Redireccionar todo el
        tráfico HTTP a HTTPS
}
server {
    listen 443 ssl;
    server_name community.domain.com;
    ssl_certificate /etc/nginx/ssl/domain.crt;
    ssl_certificate_key /etc/nginx/ssl/domain.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Figura B.2: Archivo de configuración Nginx, **Servidor 2**.

Es necesario sustituir los **subdominios** y la ruta a los **certificados SSL** para redirigir el tráfico de los subdominios configurados previamente en el proveedor del dominio.

B.4. Despliegue de contenedores

El código de ejemplo de [B.1](#) y [B.2](#) está pensado para que el repositorio se clone en la carpeta `/var`. Si el repositorio se clona en otra carpeta, se debe cambiar la ruta destino del frontend en la configuración del servidor uno.

Desgarga del repositorio Git al servidor, Git pedirá la clave **GPG** de **GitHub** asociada a una cuenta con permisos:

```
git clone https://github.com/smotrel/sm-community-tfg
```

Una vez el repositorio esté clonado, las plantillas de variables globales (**.env.template**) de cada servicio deben copiarse y renombrarse a **.env** y se deben rellenar estos últimos con las credenciales que se prefiera que se creen los servicios y las urls correspondientes de los servicios. **Es importante que estas credenciales no se introduzcan en la plantilla y que no se compartan.** Estos archivos se encuentran en :

- `/angular-front/.env.template`
- `/microservicio-reservas/.env.template`
- `/microservicio-comunidades/.env.template`
- `/microservicio-usuarios/.env.template`

Una vez todos los datos estén listos, solo queda desplegar los contenedores, para desplegar los contenedores Docker hay que lanzar los siguientes comandos en el servidor 1 y en el servidor 2, respectivamente:

```
docker compose -d -f docker-compose.prod1.yml up --build
```

```
docker compose -d -f docker-compose.prod2.yml up --build
```

Esto desplegará los contenedores del frontend y la API reservas en el servidor 1 y la API usuarios y la API comunidades en el servidor 2 y pondrá ambas en segundo plano.

Apéndice C

Manual de usuario

C.1. Registro e Identificación

Para utilizar en la web, es necesario disponer de una cuenta de usuario previamente registrada y realizar la identificación en el servicio. A continuación, se describen los procesos de registro e identificación.

C.1.1. Registro

1. Dirigirse a la página web de la aplicación, lo cual lleva a la página de login si no hay una sesión iniciada, que aparece en la figura [C.1](#).
2. Una vez en la página principal, pulsar el enlace *REGISTRATE*, debajo del botón de *Confirmar*.
3. Esto llevará al formulario de registro, el cual tiene que ser rellenado y ser válido, rellenando email, usuario, nombre, apellidos y una contraseña de mínimo 8 caracteres. Una vez que el formulario sea válido, cumplimentarlo pulsando el botón *Crear Cuenta* que se ve en la figura [C.2](#).
4. Si todo ha ido bien y los datos se validan correctamente, se mostrará un mensaje de éxito y se redirigirá a la página de identificación otra vez y la cuenta se habrá creado.

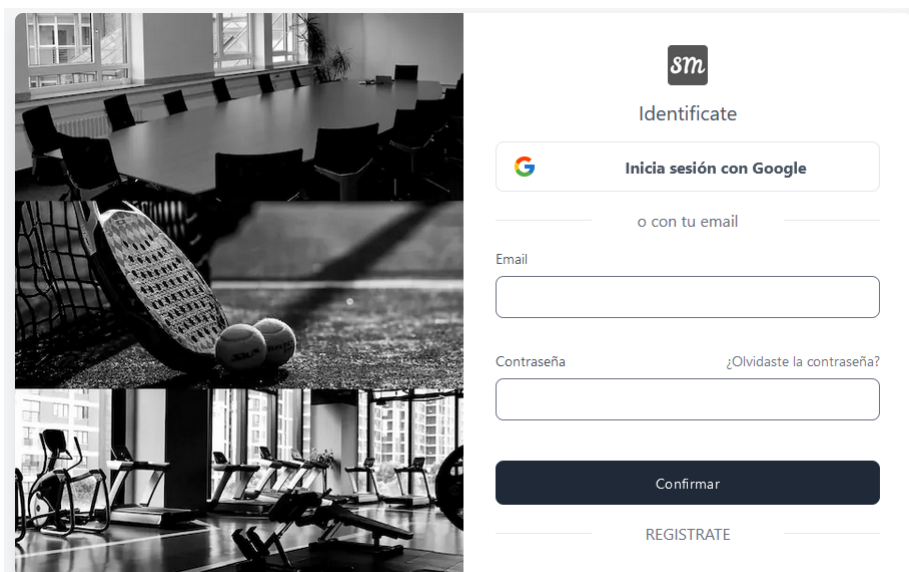


Figura C.1: Formulario de identificación.

Volver

Regístrate

Nombre de usuario

Email

Nombre

Apellidos

Contraseña

Crear Cuenta

Figura C.2: Formulario de registro.

C.1.2. Identificación

1. Dirigirse a la página web de la aplicación para ver la vista principal de identificación de la figura C.1.
2. Una vez en la página principal, se debe rellenar el formulario de identificación con correo y contraseña válidos.
3. Si el correo tiene el formato correcto, la contraseña cumple los mínimos de seguridad y el servidor confirma que las credenciales son correctas, se redirigirá dentro de la aplicación. Si hay algún error, se mostrará un mensaje con el error correspondiente.

C.2. Estructura de la web

La aplicación web, una vez se está identificado, tiene una estructura clásica de navegación lateral con un contenido dinámico cambiante en la parte central. Como se puede ver en la figura C.3, hay diferentes partes que corresponden a:

- En verde, el contenido dinámico cambiante según la ruta de la web a la que se ha navegado.
- En rojo, un acceso rápido a las comunidades a las que se pertenece.
- En naranja, los enlaces de navegación a las partes disponibles de la web.
- En amarillo, un botón con un menú contextual para editar el perfil y desconectarse.



Figura C.3: Estructura general de la web.

C.3. Vecino

Un usuario de tipo **Vecino** tiene los permisos para unirse a comunidades con código de invitación, crear reservas sobre espacios compartidos y para ver sus reservas y borrarlas, así como de modificar su perfil si fuese necesario. A continuación, se describe como hacer cada una de estas acciones.

C.3.1. Unirse a comunidades

Si se tiene un código de invitación a una comunidad que haya sido compartido por el Administrador de la comunidad de vecinos, unirse a la comunidad es muy sencillo. Solo se tiene que introducir el código en el campo que se ve en la página principal de comunidades que se puede ver en la figura C.4 marcado en rojo.

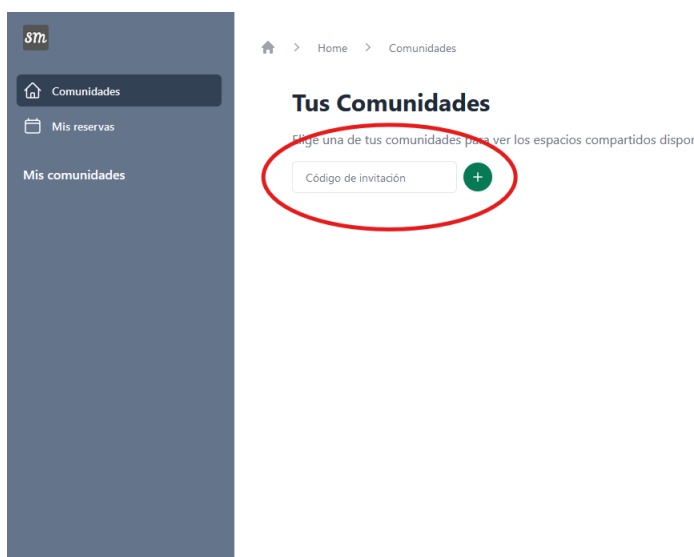


Figura C.4: Campo de unión a una comunidad con código de invitación

Si la comunidad existe y el código proporcionado por el Administrador es correcto, se unirá a una comunidad y la comunidad aparecerá en el listado de comunidades de la página principal y en el listado de acceso rápido del menú lateral, como se puede ver en la figura C.5.

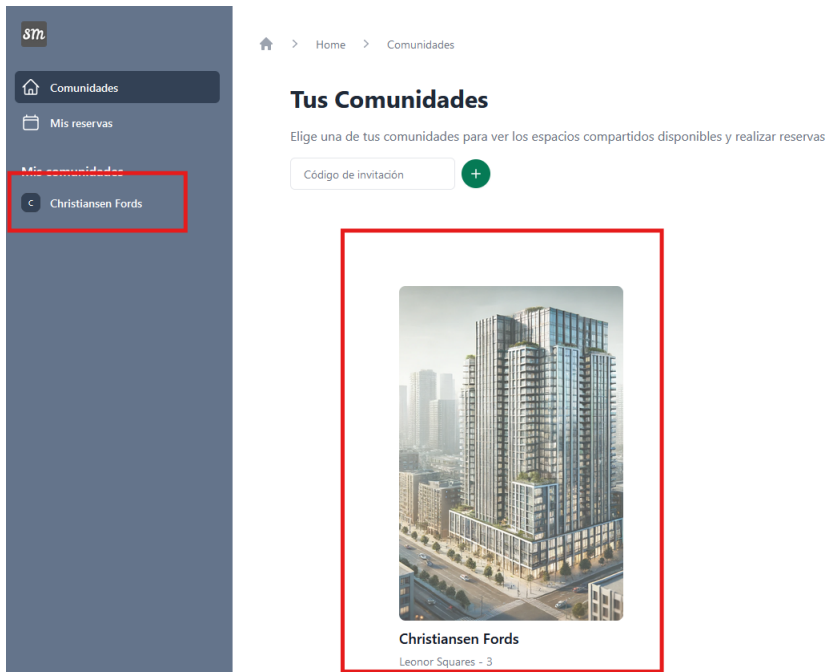


Figura C.5: Comunidad a la que se ha unido con el código de invitación.

C.3.2. Realizar una reserva

Para realizar una reserva, se debe seleccionar la comunidad en la que se quiere reservar en el menú de acceso rápido o en la página de comunidades. A continuación, se navegará a la página con los espacios disponibles de esa comunidad en los que se puede reservar. En la figura C.6 se puede ver esta lista y para reservar, solo se tiene que hacer click en el espacio deseado.

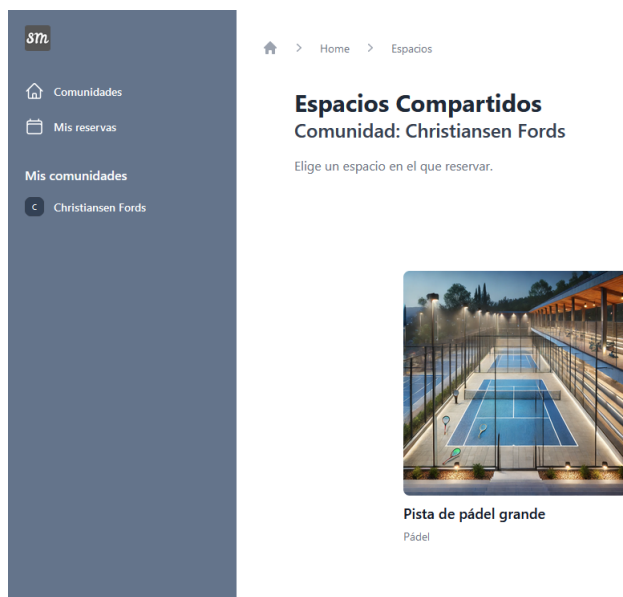


Figura C.6: Lista de espacios disponibles para reservar.

Por último, se navegará a la página de reserva mostrada en la figura C.7 en la que se podrá hacer la reserva del espacio en cuestión. En verde, se muestra el tiempo máximo semanal del espacio, el tiempo disponible y la información de cuantos participantes puede haber, aunque solo tiene que reservar un usuario. En azul y rojo, un selector de fecha y un selector de horas respectivamente, en las que aparecerán las horas ocupadas con el botón deshabilitado y las horas disponibles que se pueden seleccionar.

Pista de pádel grande - Comunidad Christiansen Fords

Selecciona una fecha y un horario para reservar. Asegúrate de cumplir con las reglas de uso.

9h Max. horas semanales	6h Horas disponibles	4 Max. participantes
-----------------------------------	--------------------------------	--------------------------------

<p>< Junio de 2024 ></p> <p>L M X J V S D</p> <p>27 28 29 30 31 1 2</p> <p>3 4 5 6 7 8 9</p> <p>10 11 12 13 14 15 16</p> <p>17 18 19 20 21 22 23</p> <p>24 25 26 27 28 29 30</p> <p>1 2 3 4 5 6 7</p>	<p>martes, 25 de junio</p> <table border="1"><tr><td>7:00</td><td>8:00</td><td>9:00</td></tr><tr><td>10:00</td><td>11:00</td><td>12:00</td></tr><tr><td>13:00</td><td>14:00</td><td>15:00</td></tr><tr><td>16:00</td><td>17:00</td><td>18:00</td></tr><tr><td>19:00</td><td>20:00</td><td></td></tr></table>	7:00	8:00	9:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00	17:00	18:00	19:00	20:00	
7:00	8:00	9:00														
10:00	11:00	12:00														
13:00	14:00	15:00														
16:00	17:00	18:00														
19:00	20:00															

Confirmar Reserva

Figura C.7: Página para reservar un espacio.

Cuando se este satisfecho, se puede pulsar el botón de *Confirmar reserva* y si todo está bien, se han seleccionado horas y se tiene tiempo disponible, se realizará la reserva.

C.3.3. Listado de reservas

Si se pulsa en el enlace del menú lateral *Reservas*, se llegará al menú mostrado en la figura C.8 en el que se puede ver las reservas actuales en el listado *Reservas actuales* y las pasadas en *Historial anterior*. En esta página se podrán borrar las reservas venideras y cancelarlas pulsando el cubo de basura.

C.4. Administrador

Como usuario **Administrador**, se pueden realizar todas las acciones para las que tiene permiso un vecino. Además, aparecerá un botón disponible en el menú lateral llamado *Administrar* como se puede ver en la figura C.9. Este botón llevará a una página de selección de tipos de contenido que se quiere administrar.

Mis reservas

Reservas actuales

<input type="checkbox"/>	Espacio	Participantes	Fecha/Hora	Acciones
<input type="checkbox"/>	Pista de pádel grande	1	26/6/2024 13:00	
<input type="checkbox"/>	Pista de pádel grande	1	26/6/2024 15:00	
<input type="checkbox"/>	Pista de pádel grande	1	26/6/2024 16:00	

Historial anterior

Espacio	Participanes	Fecha/Hora
---------	--------------	------------

Figura C.8: Listado de reservas.



Figura C.9: Menú lateral de un usuario Administrador.

Administrar

Elige que categoría quieres administrar



Figura C.10: Menú de selección de administrador.

Esta página de administración mostrada en C.10 sirve para seleccionar sobre que se quiere administrar, solo hace falta pulsar en la tarjeta del contenido que se quiera y llevará directamente a un listado de ese contenido, como ejemplo, veremos como administrar comunidades. En las figuras C.11, C.12 y C.13 se pueden ver el listado de visualización de comunidades, el formulario para editar comunidades y el formulario para crear comunidades.

Administración de comunidades Añadir comunidad

<input type="checkbox"/> Borrar todo	Ciudad	Dirección	Acciones
<input checked="" type="checkbox"/>	Yundt Rest	East Kraig	Tremblay Bridge 8 ✎ 🗑
<input checked="" type="checkbox"/>	Christiansen Fords	South Emilybury	Leonor Squares 3 ✎ 🗑
<input checked="" type="checkbox"/>	Breitenberg Bypass	Krajcikstad	Dovie Summit 8 ✎ 🗑
<input type="checkbox"/>	Buster Knoll	North Van	McClure Knolls 4 ✎ 🗑
<input type="checkbox"/>	Oran Terrace	Franeckside	Balistreri Road 9 ✎ 🗑
<input type="checkbox"/>	Fay Creek	Russelstad	Dakota Rapid 6 ✎ 🗑
<input type="checkbox"/>	O'Hara Brooks	Schneiderbury	Conroy Vista 9 ✎ 🗑
<input type="checkbox"/>	Schroeder Islands	Johnstonview	Skiles Forges 4 ✎ 🗑
<input type="checkbox"/>	Alvah Island	Purdyland	Schoen Parks 3 ✎ 🗑

Figura C.11: Listado de administración de comunidades.

En el listado de comunidades, se pueden ver las diferentes acciones disponibles:

- En azul se pueden seleccionar las comunidades que se quieren borrar en grupo.
- Una vez se seleccione más de una comunidad, se podrán borrar en grupo con el botón *Borrar todo* marcado en rojo. También, se pueden seleccionar o deseleccionar todas con el checkbox a la izquierda del botón.
- En la marca morada, se puede editar una comunidad si se pulsa en el botón del lápiz y borrar esa comunidad si se hace click en el cubo de basura.
- En el botón marcado en verde, se redirige al formulario de adición de comunidades

Cualquier acción que signifique el borrado de una o más comunidades se tendrá que confirmar en un popup para evitar errores.

Editar comunidad

Información de la comunidad


Nombre	País
<input type="text" value="Yundt Rest"/>	<input type="text" value="Faroe Islands"/>
Ciudad	Calle
<input type="text" value="East Kraig"/>	<input type="text" value="Tremblay Bridge"/>
Portal	Piso
<input type="text" value="8"/>	<input type="text" value="6"/>
Código postal	
<input type="text" value="89257"/>	
Imagen	
 Subir una imagen o arrastrar PNG, JPG, JPEG o WEBP de hasta 2MB	

Figura C.12: Formulario de edición de comunidades.

Crear comunidad

Información de la comunidad


Nombre	País
<input type="text"/>	<input type="text"/>
Ciudad	Calle
<input type="text"/>	<input type="text"/>
Portal	Piso
<input type="text"/>	<input type="text"/>
Código postal	
<input type="text"/>	
Imagen	
	

Figura C.13: Formulario de adición de comunidades.

En los formularios de edición y añadir comunidades, solo hay que rellenar los campos o cambiarles en el caso de edición y confirmar el formulario. Si algún dato es incorrecto aparecerá un error.

La gestión de usuarios, espacios y reservas funciona exactamente igual que este ejemplo dado con comunidades. Los listados y formularios son similares pero con los campos indicados para cada tipo de contenido.

Por último, cabe destacar que el usuario Administrador solo tiene permisos para ver, editar y borrar las comunidades, reservas y espacios para los que es Administrador.

C.5. Superusuario

El usuario **Superusuario** está destinado para el soporte y gestión de la aplicación en general. Tiene todas las funciones de los Vecinos y Administradores solo que además:

- Tiene permiso sobre todos los usuarios para editar su perfil, crear y borrarles, además de poderles dar roles superiores a Vecino.
- Tiene permisos sobre todas las Comunidades, no solo de las que pertenece y puede cambiar quien es el Administrador.
- Tiene permisos sobre todos los Espacios, no solo de las que pertenece.
- Tiene permisos sobre todas las Reservas, no solo de las que pertenece.

La forma de administrar es al misma que la del usuario Administrador, para una referencia de la misma, en el anexo [Administrador](#) se puede ver una guía básica de uso.

Apéndice D

Manual de endpoints de las APIs

D.1. API de usuarios

D.1.0.1. Registro

Tipo: POST

Ruta: /users/register

Atributos:

- username: string
- name: string
- lastname: string
- email: string
- password: string
- role: Role

Devuelve: Un JSON que representa el objeto User creado, pero sin devolver la contraseña. Si falla, se devuelve una **HttpResponse** con código de estado **400** y un mensaje de error.

D.1.0.2. Login

Tipo: POST

Ruta: /users/login

Atributos:

- email: string
- password: string

Devuelve: Comprueba la contraseña introducida contra el Bcrypt almacenado y si todo esta correcto, genera un token JWT válido y lo devuelve junto a los detalles del usuario (sin contraseña). Si falla, se lanza una **UnauthorizedException** y se devuelve un error.

Como dato, este es el endpoint que genera una cookie y la añade a la respuesta para obtener los beneficios comentados en el apartado ???. Las propiedades domain y secure se controlan por variable de entorno ya que en local deberían tener valores distintos al no tener protocolo https.

```
1     response.cookie('token', token, {
2         httpOnly: true,
3         secure: Boolean(process.env.SECURE_COOKIE),
4         maxAge: 3600000,
5         domain: process.env.DOMAIN
6     })
7
```

Figura D.1: Añadido de cookie a la respuesta con variables de entorno.

D.1.0.3. Comprobación de token JWT

Tipo: GET

Ruta: /users/checkAuth

Atributos:

- token : cookie

Devuelve: Booleano true si el token JWT aportado es válido y dentro de la caducidad. Se lanza una UnauthorizedException con status 400 y se devuelve mensaje de error.

D.1.0.4. Petición de todos los usuarios

Ruta protegida por identificación y rol

Tipo: GET

Ruta: /users

Devuelve: Un JSON con la información de todos los usuarios de la base de datos, sin incluir contraseñas.

D.1.0.5. Información de un usuario con id

Ruta protegida por identificación y rol

Tipo: GET

Ruta: /users/:id

Atributos:

- token : cookie

Devuelve: Un JSON con la información del usuario perteneciente a la id aportada.

D.1.0.6. Información de un usuario con token

Ruta protegida por identificación

Tipo: GET

Ruta: /getUserInfo

Atributos:

- token: cookie

Devuelve: Un JSON con la información del usuario perteneciente al token jwt aportado.

D.1.0.7. Modificación de usuario

Ruta protegida por identificación y rol

Tipo: PATCH

Ruta: /users/:id

Atributos:

- token: cookie

Devuelve: Un JSON con la nueva información del usuario.

D.1.0.8. Borrado de usuario

Ruta protegida por identificación y rol

Tipo: DELETE

Ruta: /users/:id

Atributos:

- token: cookie

Devuelve: Un mensaje de éxito o de error.

D.2. API de comunidades

D.2.0.1. Añadir comunidad

Tipo: POST

Ruta: /community

Atributos:

- name : string
- address : Address
- image : string (url de la imagen)
- adminId : string (id del administrador)
- neighbourIds : string[] (id del administrador)

Devuelve: Un JSON con los datos de la comunidad y un código de estado 201 (creado).

D.2.0.2. Petición de comunidad con su id

Tipo: GET

Ruta: /community/:id

Devuelve: Un JSON con la información de la comunidad si existe.

D.2.0.3. Petición de todas las comunidades

Tipo: GET

Ruta: /community

Devuelve: Un JSON con una lista de todas las comunidades.

D.2.0.4. Petición de todas las comunidades de un usuario

Ruta protegida por identificación

Tipo: GET

Ruta: /community/byUser

Devuelve: Un JSON con las comunidades en las que está el usuario como vecino o administrador. La información del usuario se consigue del token.

D.2.0.5. Actualizar una comunidad

Ruta protegida por identificación y rol

Tipo: PATCH

Ruta: /:id

Atributos:

- name : string
- address : Address
- image : string (url de la imagen)
- adminId : string (id del administrador)
- neighbourIds : string[] (id del administrador)

Devuelve: Un JSON con los datos actualizados y código 200 (OK).

D.2.0.6. Unirse a una comunidad con código de invitación

Tipo: POST

Ruta: /join/inviteCode

Devuelve: Código 200 (OK).

D.2.0.7. Borrar comunidad

Ruta protegida por identificación y rol

Tipo: DELETE

Ruta: /:id

Devuelve: Código 200 (OK).

D.2.0.8. Añadir espacio

Tipo: POST

Ruta: /spaces

Atributos:

- name : string
- description : string
- type : string
- maxPerWeek : int (tiempo máximo de reservas semanales)
- image : string
- maxParticipants : int
- communityId : int

Devuelve: Un JSON con los datos del espacio y un código de estado 201 (creado).

D.2.0.9. Petición de espacio con su id

Tipo: GET

Ruta: /spaces/:id

Devuelve: Un JSON con la información del espacio si existe.

D.2.0.10. Petición de todos los espacios

Ruta protegida por identificación

Tipo: GET

Ruta: /spaces

Devuelve: Un JSON con una lista de todos los espacios.

D.2.0.11. Petición de todos los espacios de una comunidad

Ruta protegida por identificación

Tipo: GET

Ruta: /byCommunityId/:id

Devuelve: Un JSON con los espacios que tiene una comunidad.

D.2.0.12. Actualizar un espacio

Ruta protegida por identificación y rol

Tipo: PATCH

Ruta: /:id

Atributos:

- name : string
- description : string
- type : string
- maxPerWeek : int (tiempo máximo de reservas semanales)
- image : string
- maxParticipants : int
- communityId : int

Devuelve: Un JSON con los datos actualizados y código 200 (OK).

D.2.0.13. Borrar espacio

Ruta protegida por identificación y rol

Tipo: DELETE

Ruta: /:id

Devuelve: Código 200 (OK).

D.3. API de reservas

Bibliografía

- [1] *Trabajo fin de grado*. URL: <https://www.uva.es/export/sites/uva/2.estudios/2.06.oficinavirtual/2.06.01.tramitesacademicos/2.06.01.09.trabajofindegrado/>.
- [2] W3Techs. *Usage statistics of React vs. Angular for websites*. 2023. URL: <https://w3techs.com/technologies/comparison/js-angularjs,js-react>.
- [3] plusvecinos. *La mejor App para tu comunidad de propietarios*. URL: <https://plusvecinos.com>.
- [4] recuerda que debes citarnos. *Fotocasa - (c) 2024 Adevinta en caso de que redistribuyas o difundas nuestro contenido protegido. ¿A qué edad nos convertimos en propietarios de vivienda en España? - (c) 2024 Adevinta, en caso de que redistribuyas o difundas nuestro contenido protegido, recuerda que debes citarnos*. 2023. URL: <https://www.fotocasa.es/fotocasa-life/compraventa/a-que-edad-nos-convertimos-en-propietarios-de-vivienda-en-espana/>.
- [5] Ring. *Neighbors by Ring*. URL: https://play.google.com/store/apps/details?id=com.ring.neighborhoods&hl=en_US.
- [6] Fincapp. *La app para Comunidades de Vecinos más usada*. URL: <https://fincapp.es/>.
- [7] Android Developers. *WebView*. URL: <https://developer.android.com/reference/android/webkit/WebView>.
- [8] web.dev. *Progressive Web Apps*. URL: <https://web.dev/explore/progressive-web-apps?hl=es-419>.
- [9] Agile gurus. *The Agile Manifesto*. 2001. URL: <https://agilemanifesto.org/>.
- [10] Jefatura de estado Gobierno de España. *Ley Orgánica 7/2021, de 26 de mayo, de protección de datos personales*. URL: <https://www.boe.es/buscar/act.php?id=BOE-A-2021-8806>.
- [11] Unión Europea. *TypeScript: JavaScript With Syntax For Types*. 2016. URL: <https://www.typescriptlang.org/>.
- [12] Glassdoor. *Sueldo medio Junior Full Stack Developer en España*. 2024. URL: https://www.glassdoor.es/Sueldos/junior-full-stack-developer-sueldo-SRCH_K00,27.htm.
- [13] Factorial. *Coste de un trabajador para la empresa*. 2024. URL: <https://factorialhr.es/blog/coste-empresa-trabajador/>.

- [14] Tarifasgasluz. *¿Cuál es el precio de la luz hoy y las horas más baratas?* 2024. URL: <https://tarifasgasluz.com/comparador/precio-kwh>.
- [15] Naturgy. *¿Cuánto consume un ordenador?* 2023. URL: https://www.naturgy.es/hogar/blog/cuanto_consume_un_ordenador.
- [16] Apple. *Mac mini*. 2024. URL: <https://www.apple.com/es/shop/buy-mac/mac-mini>.
- [17] TypeScript Lang. *Reglamento (UE) 2016/679 relativo a la protección de las personas físicas*. 2016. URL: <https://www.boe.es/buscar/doc.php?id=DOUE-L-2016-80807>.
- [18] Mozilla. *Reglamento (UE) 2016/679 relativo a la protección de las personas físicas*. 2023. URL: <https://developer.mozilla.org/en-US/docs/Glossary/SPA>.
- [19] Mozilla. *Angular Documentation*. 2024. URL: <https://angular.dev/>.
- [20] Mozilla fundation. *Package management basics*. 2024. URL: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management.
- [21] Oracle. *Software Java*. URL: <https://www.oracle.com/es/java/>.
- [22] Oracle. *Oracle*. URL: <https://www.oracle.com/es/>.
- [23] Node.js. *Node.js — Run JavaScript Everywhere*. URL: <https://nodejs.org/en>.
- [24] JWT.io. *JSON Web Tokens - JWT.io*. URL: <https://jwt.io>.
- [25] Docker Docs. *Docker Overview*. URL: <https://docs.docker.com/guides/docker-overview/>.
- [26] Git. *Git Web*. 2024. URL: <https://git-scm.com/>.
- [27] Git. *What is git?* 2024. URL: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>.
- [28] Docker Docs. *Docker Overview*. URL: <https://www.linkedin.com/pulse/5-git-workflows-improve-your-version-control/>.
- [29] PostgreSQL.org. *Materialized Views*. URL: <https://www.postgresql.org/docs/current/rules-materializedviews.html>.
- [30] Ruihan Wang y Zongyan Yang. *SQL vs NoSQL: A Performance Comparison*. 2023. URL: <https://www.cs.rochester.edu/courses/261/fall2017/termpaper/submissions/06/Paper.pdf>.
- [31] JGraph. *draw.io*. URL: <https://www.draw.io/>.
- [32] Userlane. *When Legacy Becomes Loss: The True Costs of Legacy Systems*. URL: <https://www.userlane.com/blog/enterprise-legacy-systems/>.
- [33] Deloitte America. *Connect and extend: Mainframe modernization hits its stride*. 2022. URL: <https://www2.deloitte.com/us/en/insights/focus/tech-trends/2023/future-mainframe-technology-latest-trends.html>.
- [34] Netflix Technology Blog at Medium. *Rebuilding Netflix Video Processing Pipeline with Microservices*. URL: <https://netflixtechblog.com/rebuilding-netflix-video-processing-pipeline-with-microservices-4e5e6310e359>.
- [35] Martin Fowler. *Ubiquitous Language*. URL: <https://martinfowler.com/bliki/UbiquitousLanguage.html>.

- [36] Martin Fowler. *Evans Classification*. URL: <https://martinfowler.com/bliki/EvansClassification.html>.
- [37] Martin Fowler. *DDD: Aggregate*. URL: https://martinfowler.com/bliki/DDD_Aggregate.html.
- [38] Martin Fowler. *Bounded context*. URL: <https://martinfowler.com/bliki/BoundedContext.html>.
- [39] Alistair Cockburn. *Hexagonal architecture*. URL: <https://alistair.cockburn.us/hexagonal-architecture/>.
- [40] baeldung Michal Aibin. *The DTO Pattern (Data Transfer Object)*. URL: <https://www.baeldung.com/java-dto-pattern>.
- [41] Java design patterns. *Data Transfer Object Pattern in Java: Simplifying Data Exchange Between Subsystems*. URL: <https://java-design-patterns.com/patterns/data-transfer-object/#real-world-applications-of-dto-pattern-in-java>.
- [42] IBM. *¿Qué es Java Spring Boot?* URL: <https://www.ibm.com/es-es/topics/java-spring-boot>.
- [43] Redis Raja Rao. *JSON Web Tokens (JWT) are Dangerous for User Sessions*. 2022. URL: <https://dev.to/khalidk799/environment-variables-its-best-practices-1o1o>.
- [44] Kim Maida. *Signing and Validating JSON Web Tokens (JWT)*. URL: <https://dev.to/kimmaida/signing-and-validating-json-web-tokens-jwt-for-everyone-25fb>.
- [45] Itech Empires. *Best Practices for Working with Environment Variables*. 2023. URL: <https://www.itechempires.com/2023/03/best-practices-for-working-with-environment-variables-in-python/>.
- [46] OWASP KirstenS. *Cross Site Request Forgery (CSRF)*. URL: <https://owasp.org/www-community/attacks/csrf>.
- [47] OWASP KirstenS. *Cross Site Request Forgery (CSRF)*. URL: <https://redis.io/blog/json-web-tokens-jwt-are-dangerous-for-user-sessions/>.
- [48] Mozilla. *MDN - Set-Cookie*. URL: <https://developer.mozilla.org/es/docs/Web/HTTP/Headers/Set-Cookie>.
- [49] MongoDB Inc. *MongoDB vs MySQL: Comparing Databases*. 2023. URL: <https://www.mongodb.com/compare/mongodb-mysql>.
- [50] Mongoose. *Mongoose Documentation*. 2023. URL: <https://mongoosejs.com/>.
- [51] José Antonio Muro. *¿Qué es un ORM (Object-Relational Mapping) y para qué sirve?* 2023. URL: <https://www2.deloitte.com/es/es/pages/technology/articles/que-es-orm.html>.
- [52] NestJS. *MongoDB | NestJS*. URL: <https://docs.nestjs.com/techniques/mongodb>.
- [53] Spring.io. *Uploading Files*. URL: <https://spring.io/guides/gs/uploading-files>.
- [54] Spring.io. *The Security Filter Chain*. URL: <https://docs.spring.io/spring-security/site/docs/3.0.x/reference/security-filter-chain.html>.
- [55] Spring.io. *JPA Query Methods*. URL: <https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>.
- [56] GuzzlePHP. *Guzzle Documentation*. URL: <https://docs.guzzlephp.org/en/stable/>.

- [57] Angular Documentation. *Reactive Forms*. URL: <https://angular.dev/guide/forms/reactive-forms>.
- [58] Angular Documentation. *Setting up HttpClient*. URL: <https://angular.dev/guide/http/setup>.
- [59] RxJS. *RxJS*. URL: <https://rxjs.dev/>.
- [60] Angular. *Use RouterLink for Navigation*. URL: <https://angular.dev/tutorials/learn-angular/14-routerLink>.
- [61] Google Developers. *Prácticas recomendadas sobre enlaces de Google*. URL: <https://developers.google.com/search/docs/crawling-indexing/links-crawable?hl=es-419>.
- [62] Matt G. Southern. *Google SEO 101: Do's and Don'ts of Links and JavaScript*. URL: <https://www.searchenginejournal.com/google-seo-links-javascript/365246/>.
- [63] observatoriodelaaccesibilidad. *Guía todo en uno sobre la accesibilidad web en España*. URL: <https://observatoriodelaaccesibilidad.es/wp-content/uploads/2020/06/Accesibilidad-web.pdf>.
- [64] Mozilla MDN web docs. *Tabindex*. URL: https://developer.mozilla.org/es/docs/Web/HTML/Global_attributes/tabindex.
- [65] Google Developers. *Conceptos básicos sobre las Métricas web principales*. URL: <https://developers.google.com/search/docs/appearance/core-web-vitals?hl=es>.
- [66] Squoosh. *Squoosh*. URL: <https://squoosh.app/>.
- [67] Filldb.info. *Dummy data for MYSQL database*. URL: <https://filldb.info/>.
- [68] Lukas Rasmussen. *Why is My Title Tag not Showing up in Google? The Top 5 Reasons*. 2024. URL: <https://morningscore.io/title-tag-is-not-showing-up-in-google>.
- [69] Google Developers. *Apariencia en la Búsqueda: Snippets y los Resultados de Búsqueda*. URL: <https://developers.google.com/search/docs/appearance/snippet?hl=es-419>.
- [70] Gidmaster. *Viewport Meta Tag to Control Responsive Website*. 2020. URL: <https://gidmaster.medium.com/viewport-meta-tag-to-control-responsive-website-bd64f90ee3b6>.
- [71] Google Developers. *Consolidar URLs duplicadas*. URL: <https://developers.google.com/search/docs/crawling-indexing/consolidate-duplicate-urls?hl=es>.
- [72] Angular documentation. *Version compatibility*. URL: <https://angular.dev/reference/versions>.