Universidad de Cantabria

Facultad
de
Ciencias

DESARROLLO DE UNA APLICACIÓN MÓVIL PARA LA MONITORIZACIÓN DE ACTIVIDADES DEPORTIVAS
(DEVELOPMENT OF A MOBILE APPLICATION FOR SPORTS ACTIVITY MONITORING)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERIA INFORMÁTICA

Autor: Aitor Angulo Salas

Director: Juan María Rivas Concepción

Julio – 2025

Resumen

Tener hábitos saludables es una tarea que se le dificulta a muchas personas en la actualidad, especialmente cuando se enfrentan a barreras como la falta de motivación o la dificultad de encontrar compañía para hacer ejercicio. En la actualidad el teléfono móvil es una herramienta de uso continuo en la mayoría de personas, por ello es una herramienta que puede servir de ayuda en estos casos.

Este Trabajo Fin de Grado tiene como objetivo crear una aplicación móvil que promueva la práctica deportiva y el tracking del estado físico. Con esta aplicación los usuarios podrán registrar sus entrenamientos, consultar su historial, compartir actividades y conectar con otras personas que practiquen los mismos deportes. Además, incluye funciones sociales que buscan reforzar la motivación y fomentar la creación vínculos entre usuarios.

A través de esta aplicación se pretende no solo promover el deporte, sino también disfrutar del deporte en compañía.

Palabras clave: salud, actividad física, comunidad, aplicación móvil, motivación.

Abstract

Maintaining healthy habits is a task that many people find difficult today, especially when faced with barriers such as lack of motivation or difficulty finding a companion to exercise with. Nowadays, the mobile phone is a device used continuously by most people, making it a useful tool in addressing these challenges.

This bachelor's thesis is all about developing a mobile app that encourages people to get active and keep track of their fitness journeys. With this app, users can log their workouts, look back at their progress, share their activities, and connect with others who enjoy the same sport. Plus, it comes packed with social features aimed at boosting motivation and building a sense of community. Among users.

This app aims to not just promote sports, but also to make exercising a fun experience with friends.

Key words: health, physical activity, community, mobile app, motivation.

Índice

Índice de ilustraciones

Índice de tablas

1. Introducción

En este apartado se justificará tanto por que se ha decidido hacer este trabajo como aquellas tecnologías usadas en su desarrollo.

1.1 Motivación

Practicar regularmente deporte es una de las claves para mantener un estilo de vida saludable, tanto a nivel físico como mental. Sin embargo, muchas personas son las que encuentran barreras a la hora de practicar deporte. Entre los principales obstáculos se encuentran la falta de motivación, la inseguridad a entrenar en solitario o la dificultad para encontrar compañeros con los que practicar deporte.

La aplicación TrackFit se plantea como una solución integral para quienes desean registrar sus actividades físicas, realizar un seguimiento de su progreso, compartir sus entrenamientos con otros usuarios e incluso organizar quedadas deportivas para practicar deporte en compañía. A través de estas funcionalidades, se busca crear una red social enfocada en el deporte, en la que la interacción con otros pueda servir de motivación.

Para mostrar una situación común que la aplicación pretende cubrir, imaginemos a María, una persona que quiere empezar a correr, pero no se siente cómoda saliendo sola. María descarga TrackFit, explora las actividades compartidas en su zona y encuentra una quedada para correr los domingos en un parque cercano. Decide unirse y, tras su primera salida en grupo, se siente más motivada a seguir entrenando. Además, puede registrar sus entrenamientos y ver su evolución con el tiempo, lo cual aumenta su motivación.

El formato elegido para la implementación ha sido una aplicación móvil Android, dado que los teléfonos inteligentes permiten un acceso rápido, geolocalización y notificaciones, lo que facilita la interacción y el seguimiento del progreso deportivo de forma cómoda y constante.

1.2 Decisiones tecnológicas

El desarrollo de TrackFit, una aplicación para el registro de actividades, ha requerido una cuidadosa selección de tecnologías que equilibren rendimiento, escalabilidad, mantenibilidad y experiencia de usuario.

En el desarrollo de la aplicación móvil se ha optado por utilizar tecnologías nativas de Android, empleando Kotlin [2] como lenguaje principal junto con Jetpack Compose [1] para la construcción de la interfaz de usuario. Su integración con otras bibliotecas del ecosistema como Navigation, ViewModel o Hilt, ha permitido construir una arquitectura modular y escalable.

Para el desarrollo backend se ha utilizado Spring Boot, un framework robusto y ampliamente utilizado. Spring Boot permite construir APIs RESTful de manera

rápida y estructurada, con una configuración mínima y una amplia variedad de dependencias ya integradas.

Como base de datos se ha elegido MySQL [11] [12], ideal para mantener la consistencia e integridad de los datos de los usuarios, actividades y quedadas. Para el almacenamiento de imágenes, se ha optado por Cloudinary [3], un servicio en la nube especializado en la gestión y optimización de imágenes y videos.

Finalmente, en relación con la calidad del software, se han implementado pruebas unitarias, de integración y de interfaz de usuario utilizando herramientas como JUnit, Mockito y el framework de pruebas de Jetpack Compose. Estas pruebas aseguran que tanto la lógica de negocio como la experiencia de usuario se mantengan consistentes a lo largo del tiempo.

1.3 Objetivos

El objetivo principal del proyecto TrackFit será desarrollar una aplicación móvil destinada a motivar y facilitar el seguimiento del estado físico y los entrenamientos de los usuarios.

TrackFit contará con un único tipo de usuario: el usuario registrado. Este tendrá acceso a todas las funcionalidades de la aplicación, lo que garantizará una experiencia completa y personalizada desde el primer acceso.

En lo referente a la gestión de cuenta, el usuario podrá modificar su imagen de perfil como única opción de personalización. También se implementará un historial de actividades donde se almacenarán las publicaciones realizadas por el usuario. Este historial podrá ser filtrado por diferentes criterios para facilitar la localización de entrenamientos anteriores o publicaciones específicas.

Una de las funcionalidades clave será el feed público de actividades, en el que se mostrarán de forma cronológica las publicaciones compartidas por todos los usuarios. Será posible interactuar con cada publicación (por ejemplo, mediante "me gusta" o comentarios) y acceder a una vista detallada que incluirá información específica de cada actividad: distancia, duración, tipo de ejercicio, nivel de esfuerzo, entre otros datos.

Además, TrackFit incorporará un sistema de quedadas. Cualquier usuario podrá organizar una quedada, especificando detalles como el nombre del evento, lugar, descripción, fecha y modalidad deportiva. Opcionalmente, se podrá adjuntar un archivo GPX con la ruta que se seguirá durante la actividad, lo que permitirá a los participantes visualizarla antes del evento. También se ofrecerá la posibilidad de unirse a quedadas organizadas por otros usuarios desde el propio feed.

La aplicación también permitirá buscar y seguir a otros usuarios con el objetivo de visualizar sus actividades y poder seguir a otros usuarios.

Por último, para registrar una actividad en tiempo real, TrackFit dispondrá de un HUD (Head-Up Display) que mostrará en todo momento métricas relevantes como la velocidad, la distancia recorrida, el tiempo total de la actividad y las

calorías estimadas quemadas. Esta funcionalidad convertirá el móvil en una herramienta activa durante el entrenamiento, permitiendo al usuario tener un mayor control sobre su rendimiento.

En resumen, el propósito de TrackFit es combinar el seguimiento físico, la motivación personal y la interacción social en una sola aplicación, ofreciendo así una solución integral para quienes deseen mantenerse activos y compartir su progreso con otros.

1.4 Organización

La presente memoria se estructura en las siguientes secciones:

- Sección 2. Tecnologías y herramientas: se describen la metodología de desarrollo empleada, así como las tecnologías y herramientas seleccionadas para el proyecto.

- Sección 3. Análisis y especificaciones de requisitos: se detallan los requisitos funcionales y no funcionales que guían el desarrollo de la aplicación.

- Sección 4. Diseño e implementación del Servicio REST: se explica la arquitectura del backend, describiendo sus principales capas y recursos.

- Sección 5. Diseño e implementación de la Aplicación Android: se expone la arquitectura de la aplicación móvil y se muestran ejemplos de implementación en sus diferentes capas.

- Sección 6. Pruebas: se presentan las pruebas realizadas, incluyendo casos de prueba para el servicio REST y la aplicación Android.

- Sección 7. Despliegue del backend: se describe el flujo de despliegue y la infraestructura utilizada para publicar el servicio en la nube.

- Sección 8. Conclusiones: se exponen las conclusiones obtenidas y se plantean posibles trabajos futuros.

- Sección 9. Bibliografía: recoge las fuentes utilizadas durante el desarrollo del proyecto.

2. Tecnologías y herramientas

Este apartado va a ser el encargado de explicar la metodología de desarrollo seleccionada además de las herramientas y tecnologías utilizadas para realizar el proyecto.

2.1 Metodología de desarrollo software

Para este proyecto se ha optado por una metodología Scrum, pero algo modificada debido a que iba a ser un trabajo individual y no colectivo.

En primer lugar, según esta metodología, se debían de extraer los requisitos de

la aplicación. Para extraer los requisitos, aparte de mis propias aportaciones como usuario potencial de la aplicación, se entrevistó a otra serie de usuarios potenciales de la aplicación muy heterogéneos para tener una visión amplia de lo que puede querer cada tipo de usuario.

Posteriormente los requisitos se almacenaron en un backlog. Un backlog es una lista que recoge todas las historias de usuario que se recojan. Una historia de usuario es una explicación desde el punto de vista del usuario final de una funcionalidad que se implementará en el sistema. Las historias de usuario tienen el siguiente formato: “Yo, como usuario de la aplicación, quiero poder *insertar tarea*, de manera que *insertar objetivo que cumple dicha tarea*”. Por ejemplo: “Yo como usuario, quiero poder crear una quedada, de manera que pueda organizar una quedada con otros usuarios para practicar deporte.”.

Para crear el backlog se consideraron diferentes herramientas, incluyendo Trello o Scrumdesk, pero finalmente se optó por GitHub [4] Issues, ya que de esa manera se podía tener integrado todo en el mismo repositorio [5] además de ser fácil y rápida su utilización. El backlog en GitHub se muestra en la *Ilustración 1.* Se pueden observar las distintas historias con sus pesos y prioridades organizadas en las 4 secciones: Backlog, Sprint Tasks, In progress y Done.

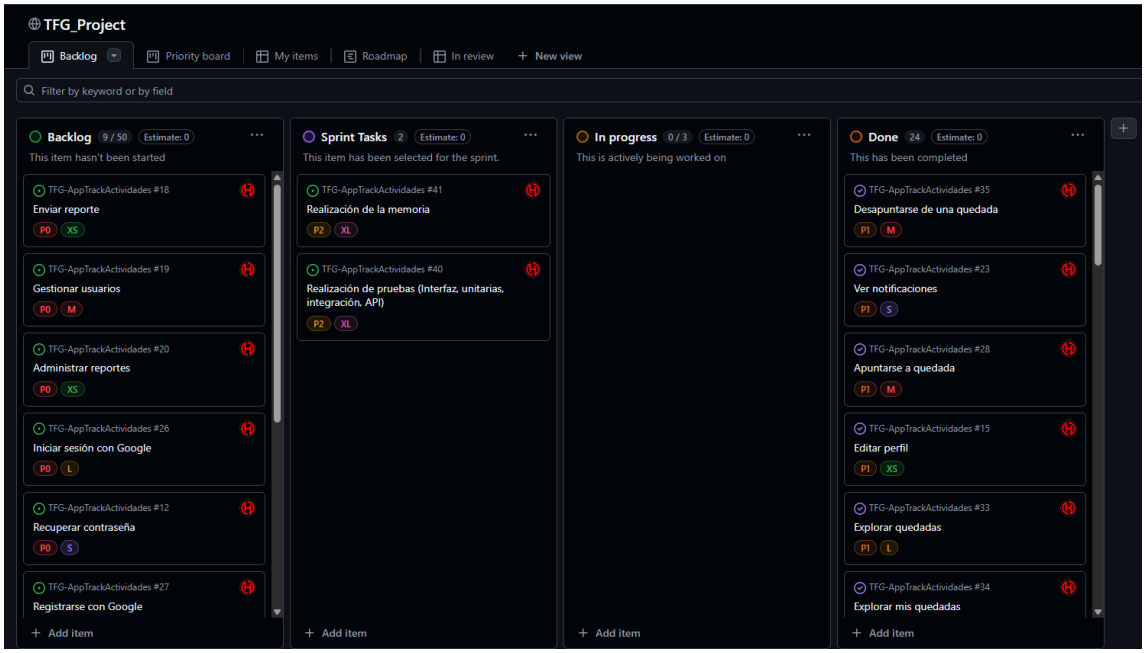


Ilustración 1 - Backlog

En Scrum existe el concepto de sprint, el cual es un periodo de tiempo definido en el que se desarrollan historias de usuario. Para este proyecto se definieron sprints de 2 semanas, ya que se creía que era un tiempo suficiente para poder desarrollar con tranquilidad las funcionalidades acordadas.

Para el sistema de control de versiones se ha utilizado Git [6] junto a la plataforma GitHub para almacenar el repositorio del proyecto, en el cual se puede encontrar tanto el código de la aplicación y el backend como los documentos adjuntos a él. Se ha creado una rama main de la cual sale la rama develop donde se van

haciendo las actualizaciones intermedias antes de hacer la actualización del main, y a su vez hay las ramas feature que siguen con el formato feature/<nombre historia usuario> las cuales cada una se enlaza con una sola historia de usuario.

2.2 Tecnologías

Antes de iniciar el proyecto se evaluaron distintas alternativas tecnológicas tanto para el desarrollo de la aplicación móvil como para el backend. En el caso de la aplicación móvil, se decidió trabajar con Android ya que abarca gran parte del mercado móvil. Se planteó un debate inicial entre utilizar Java o Kotlin como lenguaje de programación, así como entre XML tradicional y Jetpack Compose para la construcción de interfaces gráficas. Finalmente, se optó por Kotlin, dado que es el lenguaje recomendado por Google desde 2019 para el desarrollo de aplicaciones Android, por su sintaxis moderna, segura y concisa (Google, 2019)[1].

Asimismo, se eligió Jetpack Compose como framework para la interfaz de usuario debido a su enfoque declarativo, su fuerte integración con Kotlin y su capacidad para simplificar el desarrollo, mejorar la reutilización de componentes y facilitar el mantenimiento del código (Android Developers, 2023)[2].

Para el desarrollo del backend, se optó por Spring Boot, un framework ampliamente adoptado en la industria por su robustez, extensibilidad y facilidad de integración. Esta elección también respondió a criterios de experiencia previa, ya que se había trabajado con él anteriormente en la carrera, lo que permitió reducir la curva de aprendizaje y acelerar el desarrollo sin sacrificar calidad ni escalabilidad. Por último, para la base de datos se ha optado por MySQL, ya que es una tecnología conocida, rápida y de fácil uso e integración.

2.2.1 Kotlin

Kotlin es un lenguaje de programación orientado a objetos creado por JetBrains [7] el cual es actualmente soportado oficialmente por Google para el desarrollo de aplicaciones móviles en Android. Kotlin es un lenguaje que simplifica mucho la lectura del código y su desarrollo. Una característica importante de este lenguaje es que el código Kotlin se puede combinar fácilmente con código Java, lo cual lo hace todavía más modular.

Uno de sus puntos fuertes es que Kotlin elimina el código redundante, además de ser compacto y conciso, lo que optimiza y aligera el tiempo de desarrollo. Gracias a las corrutinas se simplifican las llamadas a la red y el acceso a bases de datos, permitiendo además dejar atrás los 'callbacks'. Por último, Kotlin estos últimos años está avanzando en el desarrollo multiplataforma para poder conseguir que un solo código se pueda utilizar para la misma aplicación en distintas plataformas aligerando así el tiempo de desarrollo.

[1] https://developer.android.com/kotlin?hl=es-419
[2] https://developer.android.com/compose

2.2.2 Jetpack Compose

Jetpack Compose es un framework moderno que nos permite desarrollar interfaces de usuario de aplicaciones Android de manera rápida y reutilizable. A diferencia de XML, con Jetpack Compose se permite construir la interfaz de forma declarativa, por lo cual esto simplifica su gestión y legibilidad.

2.2.3 Spring Boot

Spring Boot[3] es un framework de código abierto basado en Spring diseñado para facilitar al programador la creación de backend en Java de manera rápida y estructurada. Con Spring Boot se ha desarrollado el servicio REST (REpresentational State Transfer), el cual ofrece una interfaz clara basada en recursos. Estos servicios son los que les permiten a los clientes acceder mediante métodos HTTP (GET, POST, PUT, DELETE y PATCH) a los recursos.

2.2.4 MySQL

MySQL[4] es una base de datos relacional (RDBMS) de código abierto que ofrece fiabilidad, rendimiento y facilidad de uso. Está basada en SQL (Structured Query Language) el cual nos permite realizar consultas eficientes, mantener la consistencia mediante restricciones como claves primarias y foráneas, además de asegurar la integridad y seguridad de los datos.

2.2.5 Git

Git es una herramienta de control de versiones de código. Es gratuita, rápida, potente y permite gestionar el historial de cambios lo cual es importante en proyectos de programación. Para gestionar el repositorio se ha utilizado GitHub, que es una plataforma web que permite visualizar y almacenar repositorios Git.

Además, GitHub nos ofrece GitHub Actions para realizar integración continua. En el caso del repositorio de este proyecto[5], se ha configurado para que se ejecute GitHub Actions cada vez que se suban cambios a la rama develop o main. Por otro lado, GitHub ofrece los Projects que son almacenes donde se puede organizar las tareas al igual que haría un tablero Kanban.

2.3 Herramientas

Durante todo el desarrollo de la aplicación se han utilizado una serie de herramientas que van a ser descritas a continuación.

2.3.1 Android Studio

Android Studio[6] es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones Android. Basado en IntelliJ IDEA [8], Android Studio proporciona herramientas avanzadas de edición de código, emulación de dispositivos, análisis de rendimiento, gestión de recursos y diseño de interfaces mediante Jetpack Compose o XML. En este proyecto se ha utilizado como

[3] https://spring.io/projects/spring-boot
[4] https://www.mysql.com/
[5] https://github.com/HERRERA99/TFG-AppTrackActividades
[6] https://developer.android.com/studio?hl=es-419

entorno principal para el desarrollo móvil, permitiendo una integración completa con Kotlin, Compose y las bibliotecas del ecosistema Jetpack.

2.3.2 IntelliJ IDEA

IntelliJ IDEA[7]es un entorno de desarrollo profesional ampliamente utilizado para el desarrollo en Java y otros lenguajes como Kotlin. En este proyecto se ha empleado para el desarrollo del backend utilizando Spring Boot, gracias a sus potentes herramientas de refactorización, autocompletado inteligente, navegación rápida y soporte nativo para Maven, Gradle y Spring.

2.3.3 MySQL Workbench

MySQL[8] Workbench es una herramienta visual oficial de MySQL que permite diseñar, modelar, gestionar y consultar bases de datos. Ha sido utilizada en este proyecto en su mayor parte para inspeccionar el estado de las tablas según se iban rellenando.

2.3.4 Microsoft Azure

Microsoft Azure[9] es una plataforma en la nube que proporciona múltiples servicios para el desarrollo, despliegue y monitoreo de aplicaciones. En este proyecto se ha utilizado para el despliegue del backend REST desarrollado con Spring Boot, aprovechando sus capacidades de escalado, disponibilidad y seguridad. Azure App Service ha permitido publicar la API de forma sencilla, gestionando automáticamente el entorno de ejecución y las actualizaciones.

2.3.5 GitHub Projects

GitHub Projects[10] es una herramienta integrada en GitHub que permite gestionar tareas de desarrollo mediante tableros Kanban. Se ha utilizado para organizar las historias de usuario, asignar prioridades y hacer seguimiento del estado de cada funcionalidad (pendiente, en desarrollo, completada). Su integración con los issues del repositorio facilita la trazabilidad y el control del avance del proyecto.

2.3.6 MockFlow

MockFlow [10] es una herramienta online para la creación de wireframes y prototipos de interfaces. En este proyecto se ha utilizado para diseñar los primeros bocetos de las pantallas de la aplicación móvil, definiendo la estructura general y el flujo de navegación entre pantallas antes de pasar al desarrollo con Jetpack Compose. Se eligió por su facilidad de uso y su plan gratuito. Aunque durante el desarrollo se modificaron las pantallas, esto sirvió para tener una estructura inicial de cómo iba a verse la aplicación.

[7] https://www.jetbrains.com/idea/
[8] https://www.mysql.com/products/workbench/
[9] https://azure.microsoft.com/es-es/
[10]https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects

2.3.7 Postman

Postman[11] es una herramienta ampliamente utilizada para el diseño, prueba y documentación de APIs. En este proyecto se ha empleado para realizar pruebas de integración sobre el servicio REST, verificando que los endpoints funcionaban correctamente, enviaban y recibían los datos esperados, y respetaban las validaciones definidas.

2.3.8 Cloudinary

Cloudinary[12] es un servicio en la nube especializado en la gestión de imágenes y archivos multimedia. En este proyecto se ha utilizado su API gratuita para almacenar y servir las imágenes subidas por los usuarios de sus fotos de perfil. Cloudinary proporciona URLs optimizadas para cada recurso, además de una fácil integración con Java.

3. Análisis y especificaciones de requisitos

Este apartado documenta todos los requisitos capturados para el desarrollo del proyecto TrackFit. Los objetivos de más alto nivel que se identificaron fueron «fomentar la actividad física como hábito» y «facilitar la práctica colectiva de deporte al aire libre». Con ello se busca dar respuesta a las barreras de entrada que muchas personas enfrentan, ofreciendo una plataforma donde puedan encontrar a otros usuarios en su misma situación y superar juntos obstáculos como la falta de motivación o de compañía.

3.1 Especificación de requisitos funcionales

Los requisitos funcionales son aquellos que describen qué funcionalidades debe tener la aplicación desarrollada. Estos requisitos se han descrito en historias de usuario y almacenado en el tablero Kanban tal y como se describe en la metodología Scrum. Cada historia de usuario tiene un identificador único, un título y una descripción que sigue la estructura de “Yo, como <rol>, quiero poder <necesidad>, de manera que <proposición>, además de unos criterios de aceptación.

En total se identificaron 34 requisitos funcionales, de los cuales se pudieron llegar a implementar en la aplicación 25. Durante el desarrollo de la aplicación, se identificaron unos ciertos problemas a solventar que se tradujeron en 2 tickets de cambio (TF32 y TF33). La *Tabla 1* muestra todas las historias de usuario con su identificador (TF<número>), titulo y descripción.

Identificador	Historia de usuario	Descripción
TF00	Preparar entorno de desarrollo	Yo como desarrollador, quiero preparar el entorno de trabajo, de manera que tenga listo y configurado las librerías, versiones y entorno de trabajo antes de empezar a desarrollar la aplicación.
TF01	Crear boceto de la aplicación	Yo como desarrollador, quiero tener un boceto de la aplicación, de manera que pueda probar el funcionamiento básico de esta.

[11] https://www.postman.com/
[12] https://cloudinary.com/

TF02	Inicio de sesión	Yo como usuario, quiero iniciar sesión con mi correo y contraseña de manera que pueda acceder a mi cuenta.
TF03	Registro de usuario	Yo como usuario, quiero registrarme en la aplicación de manera que pueda disfrutar de todas las funcionalidades de la aplicación.
TF04	Registrar una actividad	Yo como usuario, quiero poder registrar mis actividades deportivas de manera que pueda analizar datos como distancia, tiempo y ritmo para hacer un seguimiento de mi progreso.
TF05	Publicar actividad	Yo como usuario, quiero poder publicar mis actividades, de manera que pueda compartir con mis seguidores mis entrenamientos.
TF06	Feed de publicaciones	Yo como usuario, quiero ver publicaciones de otras personas con sus entrenamientos de manera que pueda interactuar con ellas (comentar, dar «me gusta» o compartir) y ver sus detalles.
TF07	Ver detalles de una actividad	Yo como usuario, quiero poder ver los detalles de una actividad, de manera que pueda visualizar los parámetros recogidos en esta.
TF08	Ver perfil	Yo como usuario, quiero poder ver mi perfil y el de otros usuarios de manera que pueda consultar información relevante acerca de mi u otros usuarios, además de poder seguirlos.
TF09	Buscar usuario	Yo como usuario, quiero poder buscar usuarios, de manera que pueda buscar usuarios para ver sus perfiles y seguirlos.
TF10	Ver mi historial de actividades	Yo como usuario, quiero ver un listado con mis actividades anteriores de manera que pueda analizar mi progreso y comparar mis entrenamientos pasados además de poder aplicar filtros de datos o nombre.
TF11	Buscar actividad por palabra clave	Yo como usuario, quiero poder buscar actividades en mi historial por palabra clave, de manera que pueda encontrar la actividad que busco de manera rápida y sencilla.
TF12	Seguir usuario	Yo como usuario, quiero seguir a usuarios, de manera que pueda seguir a personas que me interesen.
TF13	Crear quedada	Yo como usuario, quiero poder crear una quedada, de manera que pueda organizar una quedada con otros usuarios para practicar deporte.
TF14	Explorar quedadas	Yo como usuario, quiero poder explorar quedadas, de manera que pueda encontrar todo tipo de quedadas a las que pueda apuntarme.
TF15	Apuntarse a quedada	Yo como usuario, quiero apuntarme a una quedada, de manera que pueda practicar deporte con otros usuarios.
TF16	Desapuntarse de una quedada	Yo como usuario, quiero poder desapuntarme de una quedada, de manera que no me aparezca más en mis quedadas.
TF17	Eliminar quedada	Yo como usuario, quiero poder eliminar la quedada, de manera que ya no le aparezca más a los usuarios en el feed.
TF18	Explorar mis quedadas	Yo como usuario, quiero poder explorar mis quedadas, de manera que pueda observar a que quedadas estoy apuntado.
TF19	Editar perfil	Yo como usuario, quiero poder editar mi perfil, de manera que pueda personalizar mi información personal y fotos de perfil.
TF20	Ver notificaciones	Yo como usuario, quiero poder ver las notificaciones, de manera que pueda estar informado acerca de las ultimas interacciones con mi perfil o actividades.

TF21	Registrarse con Google	Yo como usuario, quiero poder registrarme con Google, de manera que me sea más fácil y rápido crearse una cuenta.
TF22	Iniciar sesión con Google	Yo como usuario, quiero poder iniciar sesión con Google, de manera que sea más fácil y rápido iniciar sesión.
TF23	Visualizar objetivos	Yo como usuario, quiero poder visualizar mis objetivos, de manera que pueda visualizar su progreso y estado.
TF24	Administrar reportes	Yo como administrador, quiero poder administrar los reportes, de manera que pueda resolver los problemas relacionados con la aplicación o los usuarios.
TF25	Gestionar usuarios	Yo como administrador, quiero poder gestionar usuarios, de manera que pueda ejercer acciones sobre ellos como banearlos.
TF26	Enviar reporte	Yo como usuario, quiero poder enviar reportes de errores, publicaciones o usuarios, de manera que estos problemas sean solucionados o los usuarios y las publicaciones investigados.
TF27	Recuperar contraseña	Yo como usuario, quiero poder recuperar mi contraseña, de manera de que si la he olvidado pueda cambiarla.
TF28	Visualizar mi progreso	Yo como usuario quiero ver estadísticas y gráficos sobre mis entrenamientos de manera que pueda analizar mi evolución y detectar áreas de mejora.
TF29	Añadir objetivos personales	Yo como usuario quiero establecer objetivos como distancia semanal o número de entrenamientos de manera que pueda motivarme a mejorar y alcanzar mis metas deportivas.
TF31	Desplegar backend en Azure	Yo, como usuario, quiero que el backend este desplegado en Azure, de manera que pueda acceder en cualquier momento y lugar a la aplicación.
TF32	Ticket modificar distribución de la Bottom Bar	Yo como usuario de la aplicación, quiero que el botón de mapas sea sustituido por un botón de historial y que se añada un acceso directo al perfil, de manera que pueda acceder fácilmente a mi historial de actividades y a mi perfil desde la barra de navegación inferior.
TF33	Ticket verificar correo electrónico	Yo como usuario, quiero poder verificar mi correo, de manera que solo puedan interactuar conmigo cuentas verificadas.

Tabla 1. Historias de usuario

Las historias que no se han podido llegar a implementar son registrarse con Google, iniciar sesión con Google, visualizar objetivos, administrar reportes, gestionar usuarios, enviar reporte, recuperar contraseña, visualizar mi progreso y añadir objetivos personales.

3.2 Requisitos no funcionales

Los requisitos no funcionales especifican los estándares y cualidades que debe cumplir un sistema para funcionar de manera eficaz. Estos abarcan aspectos como el rendimiento, la escalabilidad, la seguridad, la usabilidad y la mantenibilidad, que influyen directamente en la experiencia del usuario y en la sostenibilidad del software a largo plazo. Para su definición, se ha tomado como referencia el estándar internacional ISO/IEC 25010, el cual establece un modelo de calidad del software basado en ocho características principales: adecuación

funcional, eficiencia del rendimiento, compatibilidad, usabilidad, fiabilidad, seguridad, mantenibilidad y portabilidad. Este marco ha servido como guía para asegurar que la aplicación cumpla con los niveles de calidad esperados en todas sus dimensiones clave. En la *Tabla 2* observamos todos los requisitos no funcionales de la aplicación.

Id	Tipo	Descripción
RNF1	Compatibilidad	La aplicación debe ser compatible con versiones de Android desde la 8.1 (Oreo) hasta las más recientes, asegurando su funcionamiento en una amplia variedad de dispositivos.
RNF2	Rendimiento	La aplicación debe contar con tiempos de carga rápidos (menos de 3 segundos[13][14]) y una respuesta fluida (menos de 400 milisegundos) al interactuar con botones, menús y funcionalidades clave.
RNF3	Usabilidad	La interfaz debe ser intuitiva, seguir las guías de diseño de Android y facilitar la navegación por las distintas secciones como registro de actividades, quedadas y perfil.
RNF4	Usabilidad	La aplicación debe adaptarse correctamente al modo oscuro, garantizando una visualización adecuada de todos los elementos sin comprometer la legibilidad.
RNF5	Rendimiento	La arquitectura de la aplicación debe permitir el crecimiento en el número de usuarios y actividades sin afectar el rendimiento ni la experiencia de uso.
RNF6	Seguridad	La contraseña de los usuarios tiene que ser almacenada de forma cifrada para mayor seguridad.
RNF7	Rendimiento	Las acciones como seguir o dejar de seguir a otro usuario deben responder en menos de un segundo para ofrecer una interacción fluida.

Tabla 2. Requisitos no funcionales

4. Diseño e implementación del Servicio REST

Esta va a ser la sección encargada de describir el diseño e implementación del servicio REST, explicando qué arquitectura se ha utilizado y por qué, detallando cada una de sus capas.

4.1 Arquitectura

Lo primero que hay que especificar para entender la arquitectura elegida es que el servicio REST se ha desarrollado con Spring Boot, lo que nos lleva a desarrollarlo con la arquitectura típica de Spring Boot, que se centra en la separación de responsabilidades. A continuación, se describen las principales capas de la arquitectura y en la *Ilustración 2* se pueden ver gráficamente:

- *Controller layer:* Esta capa es la encargada de recibir las peticiones HTTP y procesarlas para llamar al servicio correspondiente y devolver la respuesta adecuada al cliente. En esta capa se definen los endpoints de la aplicación, así como la ruta a cada recurso.

[13] https://jakobnielsenphd.substack.com/p/time-scale-ux
[14] https://www.browserstack.com/guide/how-fast-should-a-website-load

- *Service Layer:* Esta capa es la encargada de la lógica de negocio. Procesa y transforma los datos coordinándose tanto como con la capa superior (Controlador) como con la capa inferior (Repositorio). Aquí es donde se implementan las funcionalidades principales de la aplicación.

- *Repository Layer:* Esta capa es parte de la capa de persistencia de los datos independientemente de la tecnología usada. Los repositorios nos proporcionan métodos para realizar las operaciones CRUD (Create, Read, Update y Delete) en las entidades de la aplicación. En este proyecto se ha utilizado JPA (Java Persistence API), el cual nos proporciona automáticamente métodos básicos como puede ser el findBy para buscar endidades por su identificador, pero también da soporte para crear nuestras propias consultas SQL personalizadas. JPA es una especificación de Java que permite mapear objetos Java a tablas de una base de datos relacional, facilitando así la gestión y persistencia de datos de forma transparente.

- *Entities:* Representan los objetos del dominio de la aplicación y encapsulan los datos y comportamientos asociados. Estas entidades se mapean directamente en el sistema de almacenamiento.

- *Data Base*: Es la encargada de almacenar todos los datos de manera persistente.

El diseño de todas las capas mencionadas será detallado en las siguientes secciones.

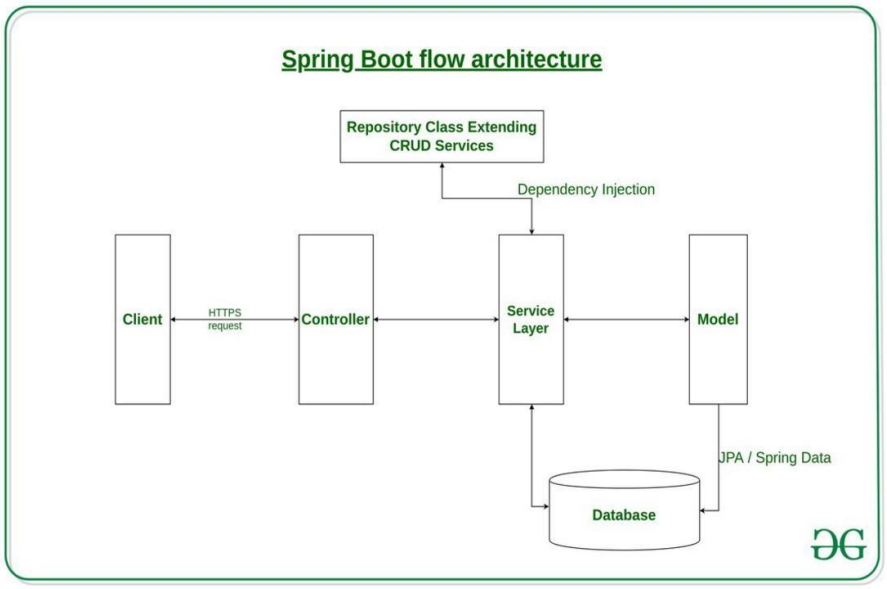


Ilustración 2. Arquitectura Spring Boot [16]

El modelo de dominio de la aplicación se muestra en la *Ilustración 3*. En él podemos observar cómo cada usuario tiene asociado tanto unas publicaciones,

[16] https://www.geeksforgeeks.org/springboot/spring-boot-architecture/

como actividades, comentarios, likes, follows y quedadas. A su vez cada publicación tiene asociada una actividad la cual tendrá una serie de LatLng que es un objeto que sirve para almacenar las coordenadas de cada punto de la ruta.

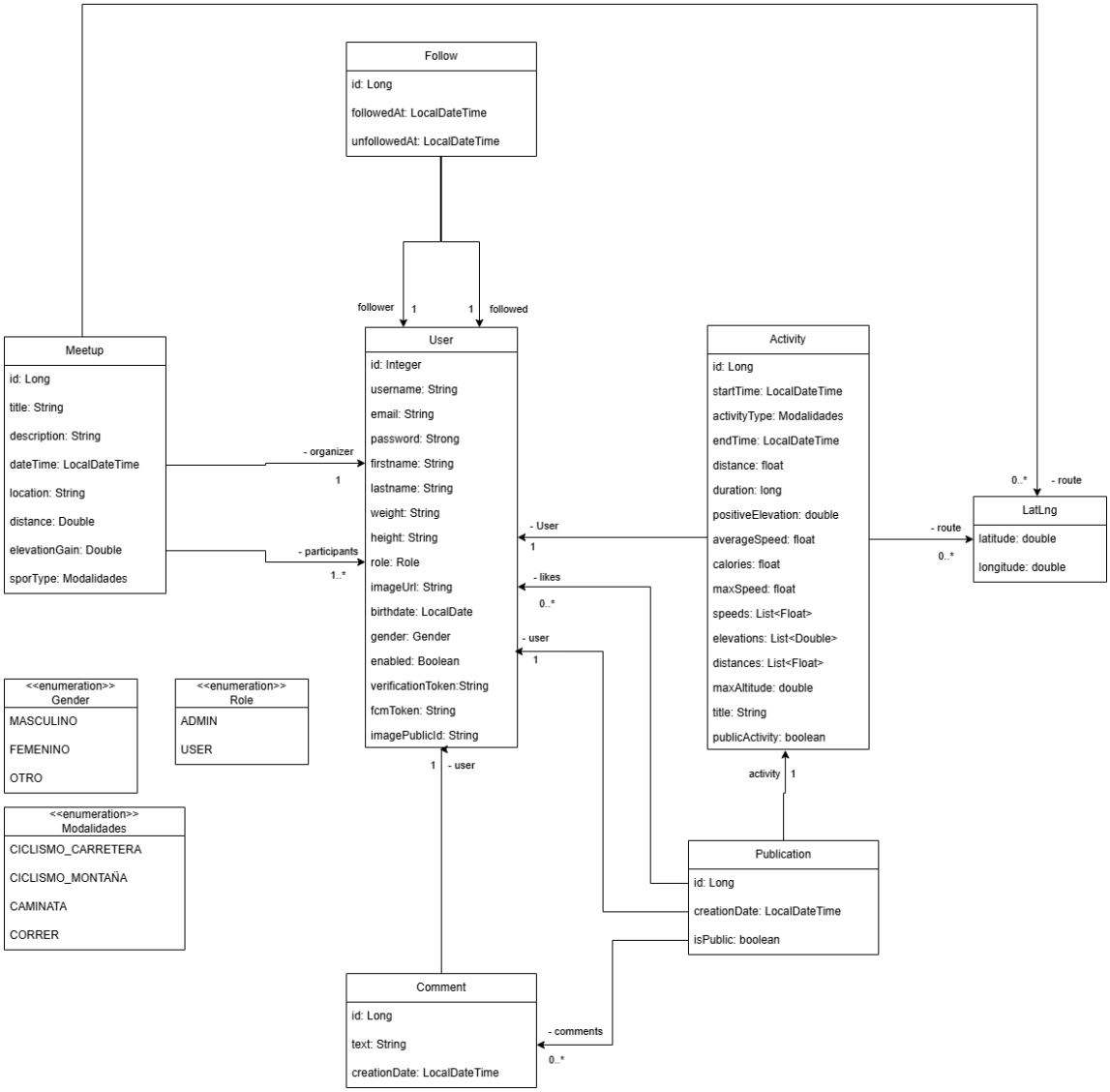


Ilustración 3. Modelo de dominio TrackFit

4.2 Controller Layer

Lo primero que se tiene que hacer para desarrollar un servicio REST es definir los recursos que componen cada controlador, sus plantillas URI, los códigos HTTP que retornan y las representaciones del recurso tanto del cliente como del servidor.

Adicionalmente, el acceso a los endpoints se controlará mediante el uso de JWT (JSON Web Token), de manera que únicamente los usuarios autenticados puedan realizar peticiones. JWT es un mecanismo de autenticación que permite verificar la identidad del usuario mediante un token firmado que se envía en cada petición, evitando mantener sesiones en el servidor. Este token debe ser incluido en cada petición a los endpoints protegidos, generalmente a través del

encabezado Authorization. El servidor valida el token para garantizar que la solicitud proviene de un usuario legítimo y autorizado.

En el backend se han definido los siguientes controladores: ActivityController, AuthController, MeetupController, PublicationController y UserController.

4.2.1 Recursos servicio REST

Vamos a dividir en diferentes tablas en función del controlador para que sea más legible.
En la *Tabla 3* podemos observar los endpoints del AuthController.

Recurso	URI	Query Params	Método	Códigos HTTP
Registrarse	/auth/register		POST	200 (OK) 400 (Bad Request)
Iniciar sesión	/auth/login		POST	200 (OK) 400 (Bad Request)
Validar token	/auth/validateToken		POST	200 (OK) 400 (Bad Request)
Verificar cuenta	/auth/verify		POST	200 (OK) 400 (Bad Request)

Tabla 3. Endpoints AuthController

En la *Tabla 4* podemos observar los endpoints del ActivityController.

Recurso	URI	Query Params	Método	Códigos HTTP
Obtener actividades	/activities		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Crear actividad	/activities		POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener flujo actividades	/activities/stream		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)

Tabla 4. Endpoints ActivityController

En la *Tabla 5* podemos observar los endpoints del PublicationController.

Recurso	URI	Query Params	Método	Códigos HTTP
Crear publicación	/publications		POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener publicación	/publications/{id}		GET	200 (OK) 401 (Unauthorized) 404 (Not Found)
Obtener publicaciones publicas	/publications/public	page, size	GET	200 (OK) 401 (Unauthorized) 404 (Not Found)

Añadir like a publicación	/publications/{id}/like	userId	POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Eliminar like de publicación	/publications/{id}/removeLike	userId	POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Añadir comentario	/publications/{id}/comment	userId, text	POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener comentarios	/publications/{id}/comment		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener publicaciones usuario	/publications/user/{id}	page, size	GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener publicaciones filtradas		page, size, nombre, activityType, distanciaMin, distanciaMax, positiveElevationMin, positiveElevationMax, durationMin, durationMax, averageSpeedMin, averageSpeedMax	GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)

Tabla 5. Endpoints PublicationController

En la *Tabla 6* podemos observar los endpoints del MeetupController.

Recurso	URI	Query Params	Método	Códigos HTTP
Crear quedada	/meetups	meetupJson, gpxFile	POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found) 500 (Internal server error)
Obtener quedada	/meetups/{id}		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener quedadas cercanas	/meetups/all	Lat, lgn, page, size	GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener mis quedadas	/meetups/my	page, size	GET	200 (OK) 400 (Bad Request) 401 (Unauthorized)

				404 (Not Found)
Unirse a quedada	/meetups/{id}/join		POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Desapuntarse de la quedada	/meetups/{id}/leave		POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Borrar quedada	/meetups/{id}		DELETE	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)

Tabla 6. Endpoints MeetupController

En la *Tabla 7* Podemos observar los endpoints del UserController.

Recurso	URI	Query Params	Método	Códigos HTTP
Obtener mi información	/user/me		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener información usuario	/user/{iduser}		GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Buscar usuario	/user/search	text, page, size	GET	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Modificar foto perfil	/user/{idUser}/profile-picture	image	PATCH	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Seguir usuario	/user/{followedId}/follow		POST	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Dejar de seguir usuario	/user/{followedId}/unfollow		DELETE	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)
Obtener fcm token	/user/fcm-token		PUT	200 (OK) 400 (Bad Request) 401 (Unauthorized) 404 (Not Found)

Tabla 7. Endpoints UserController

Para definir la estructura y contenido de los datos intercambiados entre el cliente y el servidor, se establecen las representaciones de cada recurso. Esto nos permite saber con claridad cómo va a ser el formato de la llamada y de la petición. En este caso se va a utilizar JSON para las representaciones. En la *Tabla 8* y *Tabla 9* podemos observar dos representaciones JSON de la representación del cliente y del servidor.

Representación iniciar sesión cliente

{
"identifier": "hererra99" ,
"password": "001970Herrera"
}

Tabla 8. Representación cliente

Representación iniciar sesión servidor

{
"token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIadqwwdqwdqwdsadasdasdasdas
"
}

Tabla 9. Representación servidor

En el servicio, a la hora de devolver recursos, no siempre nos tienen que interesar todos los atributos de una entidad, por ello se utilizan DTOs (Data Transfer Objects), los cuales son objetos con solo los atributos que nos interesa devolver. Por ejemplo, en el caso de obtener un usuario para mostrar su perfil no nos interesan campos como su fecha de nacimiento, su token o si está o no habilitado. A modo de ejemplo se muestra una representación de lo que devuelve el endpoint para obtener un usuario por id, en este caso UserDTO, en la *Tabla 10*.

Representación obtener usuario por id con DTO

{
"id": 2,
"image": "https://res.cloudinary.com/edv5d.jpg",
"nombre": "Aitor",
"apellidos": "Angulo Salas",
"email": "aitor001970@gmail.com",
"peso": 89.0,
"altura": 187,
"genero": "MASCULINO",
"followersCount": 1,
"followingCount": 1,
"username": "hererra99",
"following": false
}

Tabla 10. Representación Usuario (DTO)

La implementación de las clases Controller se ha realizado mediante la utilización de clases POJO (Plain Old Java Object) anotadas con su respectivo @RestController y @RequestMapping("<Endpoint>"). Un POJO es una clase Java simple que no extiende ni implementa ninguna otra clase o interfaz.

La clase que implementa el controlador AuthController se presenta en la *Ilustración 4*. Esta clase es la encargada de gestionar el inicio de sesión, registrar los usuarios, validar los tokens y comprobar que el usuario ha verificado su email. Se puede observar que este controlador delega en el AuthService para realizar su lógica de negocio. En la siguiente sección se describen estas clases de servicio.

AuthController.java

```java
@RestController  Aitor Angulo Salas *
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthService authService;

    @PostMapping(value = "login")  Aitor Angulo Salas
    public ResponseEntity<AuthResponse> Login(@RequestBody LoginRequest request) {
        return ResponseEntity.ok(authService.login(request));
    }

    @PostMapping(value = "register")  Aitor Angulo Salas
    public ResponseEntity<AuthResponse> Register(@RequestBody RegisterRequest request) {
        return ResponseEntity.ok(authService.register(request));
    }

    @PostMapping(value = "validateToken")  Aitor Angulo Salas *
    public ResponseEntity<ValidResponse> validateToken(HttpServletRequest request) {
        ValidResponse response = authService.validateToken(request);
        return ResponseEntity.ok(response);
    }

    @GetMapping("/verify")  Aitor Angulo Salas
    public ResponseEntity<String> verifyAccount(@RequestParam("token") String token) {
        return authService.verifyUserAccount(token);
    }
}
```

Ilustración 4. Clase AuthController

4.3 Service Layer

La capa de servicio como ya hemos dicho es la encargada de implementar la capa de negocio del sistema. Esta capa se podría omitir si las operaciones que van a llevar a cabo fueran sencillas, pero en este caso se ha optado por desarrollarla explícitamente, ya que le da una visión más modular a cada capa. Adicionalmente, a la hora de entender el código, el tener separado cada acción en su respectiva capa mejora la legibilidad. Además, si en un futuro se quiere seguir expandiendo la aplicación agregándole lógica de negocio más compleja, el tener separado el código en capas hará mucho más rápida y sencilla su refactorización.

En esta capa encontramos los siguientes servicios: ActivityService, AuthService, EmailService, MeetupService, NotificacionesService, PublicationService, UserService. Al igual que los controladores, las clases Service se basan en clases POJO anotadas con su respectivo @Service.

Un fragmento de la clase AuthService se muestra en la *Ilustración 5*. Esta clase está asociada al AuthController visto previamente. Se puede observar que el controlador solo tiene la obligación de procesar la petición HTTP, y delega en el Service la acción que tiene que realizar. Esta clase AuthService solo tiene los métodos asociados a cada endpoint, y, además, un método auxiliar que genera un HTML para indicar si el correo ha podido o no ser verificado.

AuthService.java

```java
@Service  2 usages  Aitor Angulo Salas *
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository userRepository;
    private final JwtService jwtService;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;
    private final EmailService emailService;

    public AuthResponse login(LoginRequest request) {  1 usage  Aitor Angulo Salas *
        if (request.getIdentifier() == null || request.getPassword() == null) {
            throw new IllegalArgumentException("El identificador y la contraseña son obligatorios");
        }

        User user = userRepository.findByUsernameOrEmail(request.getIdentifier())
                .orElseThrow(() -> new UsernameNotFoundException("Usuario no encontrado"));

        if (!user.isEnabled()) {
            throw new IllegalStateException("Debes verificar tu correo electrónico antes de iniciar sesión");
        }

        try {
            authenticationManager.authenticate(
                    new UsernamePasswordAuthenticationToken(user.getUsername(), request.getPassword())
            );
        } catch (BadCredentialsException e) {
            throw new BadCredentialsException("Contraseña incorrecta");
        }

        String token = jwtService.getToken(user);
        return AuthResponse.builder()
                .token(token)
                .build();
    }

    public AuthResponse register(RegisterRequest request) {  1 usage  Aitor Angulo Salas *
        if (request.getEmail() == null || request.getUsername() == null || request.getPassword() == null) {
            throw new IllegalArgumentException("El email, usuario y contraseña son obligatorios");
        }

        if (userRepository.existsByUsername(request.getUsername())) {
            throw new IllegalArgumentException("El nombre de usuario ya está en uso");
        }

        if (userRepository.existsByEmail(request.getEmail())) {
            throw new IllegalArgumentException("El correo electrónico ya está registrado");
        }

        String token = UUID.randomUUID().toString();

        User user = User.builder()
                .username(request.getUsername())
                .email(request.getEmail())
                .password(passwordEncoder.encode(request.getPassword()))
                .firstname(request.getFirstname())
                .lastname(request.getLastname())
                .height(request.getHeight())
                .weight(request.getWeight())
                .role(Role.USER)
                .birthdate(request.getBirthdate())
                .gender(request.getGender())
                .imageUrl("https://i.postimg.cc/RFSkJZtg/462076-1g-CSN462076-MG3928385-1248x702.webp")
                .enabled(false)
                .verificationToken(token)
                .build();

        userRepository.save(user);
        emailService.sendVerificationEmail(user.getEmail(), token);

        return AuthResponse.builder()
                .token(null)
                .build();
    }
}
```

Ilustración 5. Clase AuthService

4.4 Repository Layer

La capa Repository es la capa responsable de la interacción con la capa de persistencia de los datos y se basa en el framework JPA.

Como tecnología para almacenar los datos en este proyecto se ha seleccionado MySQL, ya que era una tecnología familiar, de fácil implementación y con gran potencia. Además, su excelente integración con JPA nos permite gestionar la persistencia de datos de manera eficiente directamente desde el código.

Con JPA, la definición de las tablas y el mapeado entre entidades y las tablas se realiza mediante anotaciones en las clases del dominio. Esto es posible ya que JPA nos ofrece una serie de anotaciones como @Id, @ManyToMany, @Column entre otras que nos permite mapear entre entidades y tablas, así como especificar aspectos adicionales que se podrían especificar con SQL nativo.

Se ha definido el modelo de la base de datos, el cual es muy parecido al de dominio, pero en este caso se incluyen todas las relaciones con tablas intermedias o claves foráneas. El modelo de datos se presenta en la *Ilustración 6.*

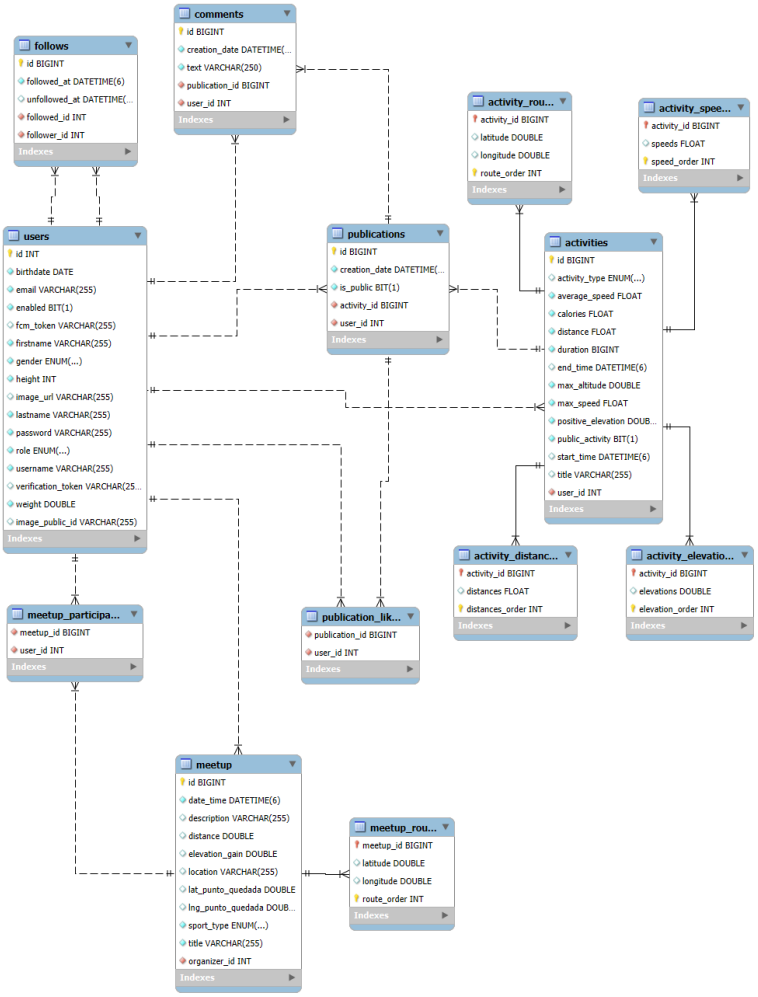


Ilustración 6. Modelo de la base de datos

Ya hemos definido como se van a mapear y representar las entidades, pero ahora hace falta operar con ellas (guardar, borra y obtener). Para ello se utiliza la capa del repositorio, que es la encargada de hacer de puente entre el servicio y la base de datos. Para ello JPA nos ofrece JpaRepository, que son una serie de interfaces que nos ofrecen ya por defecto métodos para operaciones CRUD sin necesidad de declararlas nosotros. Adicionalmente, también nos permite definir métodos de acceso personalizados con consultas SQL.

En la *ilustración 7* se muestra un ejemplo a través de la clase UserRepository, la cual nos muestra métodos para realizar operaciones sobre los usuarios. Entre los métodos definidos en UserRepository, encontramos consultas personalizadas que permiten buscar usuarios por nombre de usuario, correo electrónico, nombre completo o fragmentos de texto, así como validar la existencia de ciertos datos o gestionar procesos como la verificación de cuenta. Esto demuestra cómo JPA permite extender fácilmente el comportamiento del repositorio mediante anotaciones como @Query, sin necesidad de escribir implementaciones manuales.

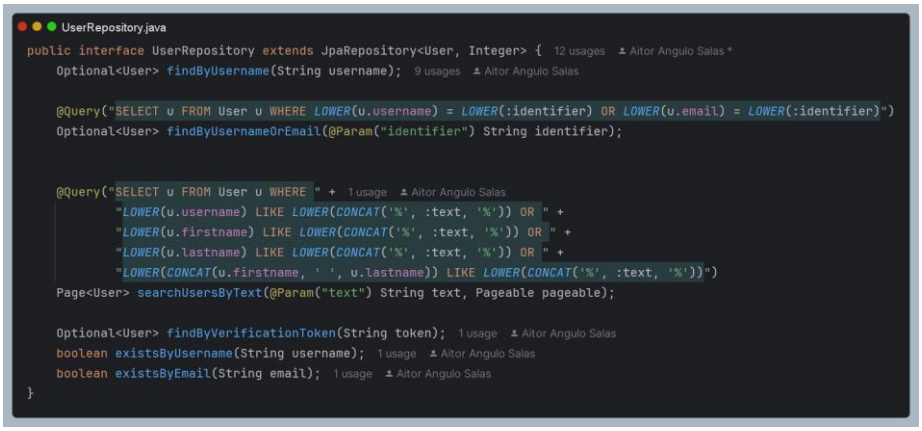


Ilustración 7. Interfaz UserRepository

5. Diseño e implementación de la Aplicación Android

Esta sección detalla el diseño y la implementación de la aplicación Android desarrollada. Para su construcción se ha optado por una arquitectura basada en el patrón MVVM (Model-View-ViewModel), complementado con los principios de Clean Architecture[18], lo que permite una separación clara de responsabilidades y mejora de la mantenibilidad, escalabilidad y facilidad de prueba del sistema. La arquitectura de la aplicación se representa en la *Ilustración 8*.

[18] Clean Architecture es un patrón de diseño que organiza el código en capas separando la lógica de negocio de los detalles de implementación, facilitando el mantenimiento y la escalabilidad.

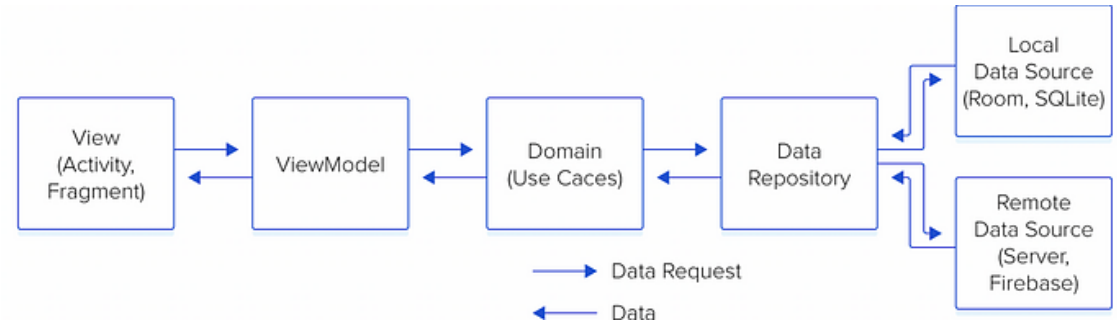


Ilustración 8. Arquitectura de la aplicación[19]

5.1 Arquitectura

La arquitectura de la aplicación Android se basa en el patrón MVVM. Este patrón garantiza una clara separación de responsabilidades entre las distintas capas, mejorando la mantenibilidad, testeabilidad y escalabilidad del código. Sin embargo, la estructura del proyecto no solo se limita al MVVM tradicional, sino que adopta la organización en capas bien diferenciadas (data, domain y presentation), en línea con los principios de la Clean Architecture.

La capa de presentación contiene los componentes de la interfaz (Composables) y los ViewModels. Las vistas observan un estado expuesto por los ViewModels, los cuales gestionan la lógica de interfaz, coordinan eventos del usuario y se comunican con los casos de uso definidos en la capa de dominio.

- Los ViewModel son los encargados de coordinar el flujo de datos desde el modelo a la vista. Además, son los encargados de gestionar los eventos que produzca el usuario al interactuar con la vista e invocar los casos de uso definidos.

- La vista no implementa lógica, sino que sirve para representar el estado actual de la interfaz y detectar los eventos que se realicen en ella, para delegar las respuestas en el ViewModel.

La capa de dominio representa el puente para la comunicación entre la capa de datos y la de presentación. Esta capa contiene los casos de uso, los cuales son llamados por la capa de presentación para ordenarle a la capa de datos operaciones.

La capa de datos se encarga de toda la lógica relacionada con la obtención, almacenamiento y gestión de datos. Aquí se encuentran los repositorios, que actúan como intermediarios entre las fuentes de datos (API remota o base de datos local) y el resto de la aplicación. Las comunicaciones con la API se hacen mediante interfaces ApiService, definidas con Retrofit[20], donde se especifican las llamadas a los endpoints del backend.

---

[19] https://github.com/Mohamed-Fadel/clean-mvvm-flutter-sample

[20] Biblioteca de cliente HTTP de código abierto para Android y Java que facilita la interacción con servicios web RESTful. Simplifica el proceso de realizar solicitudes de red al transformar interfaces Java en llamadas HTTP.

A continuación, se va a detallar desde más bajo nivel a más alto cada capa con cada una de las posibles clases que tiene para entender mejor el porqué de la distribución en 3 capas.

5.1.1 Capa de datos

Como ya hemos dicho, esta capa es la encargada de gestionar todo lo relacionado con el acceso a los datos proporcionados por la API, ya que en esta aplicación no se utilizan datos locales más allá del token de autentificación de la API o los datos personales del usuario.

El acceso a la API se realiza a través de las interfaces denominadas ApiService, definidas utilizando Retrofit. Estas interfaces definen el acceso a los endpoints del backend, los métodos HTTP asociados y los parámetros necesarios. Un ejemplo de un método definido en un ApiService se presenta en la *Ilustración 9*. Este método utiliza las siguientes etiquetas: @POST, que es la encargada de definir que se va a realización una operation HTTP POST en la Uri especificada, @Header que se encarga de definir una cabecera para la petición, en este caso "Authorization" y por último @Path, que se encarga de definir cuál va a ser el valor de la Uri en {activityId}.

PublicationsApiService.kt

```kotlin
@POST("/publications/{activityId}")
suspend fun createPublication(
    @Header("Authorization") token: String,
    @Path("activityId") activityId: Long
): PublicationResponse
```

Ilustración 9. Método ApiService

El elemento más básico que nos vamos a encontrar en esta capa son los DTOs. Los DTOs hacen la misma función que en la API del backend, son entidades más simples que contienen solo los atributos que se quieren enviar/recibir de la API.

Para gestionar todos estos métodos, existen los repositorios, los cuales se encargan de interactuar con los ApiService. Debido a que TrackFit es una red social con el potencial de tener que gestionar grandes volúmenes de datos, se optó por utilizar paginación. La paginación se basa en obtener los datos de la API en secciones o páginas, esto es, si hay 20 publicaciones y se hace una petición paginada de tamaño 10, la API solo devuelve las 10 primeras, y es la aplicación la encargada de pedir las 10 siguientes. Debido a esto, a parte de los repositorios ha sido necesario crear clases PagingSource, que son las encargados de la carga paginada de los elementos. Un ejemplo del PublicationPagingSource se muestra en la *Ilustración 10*.

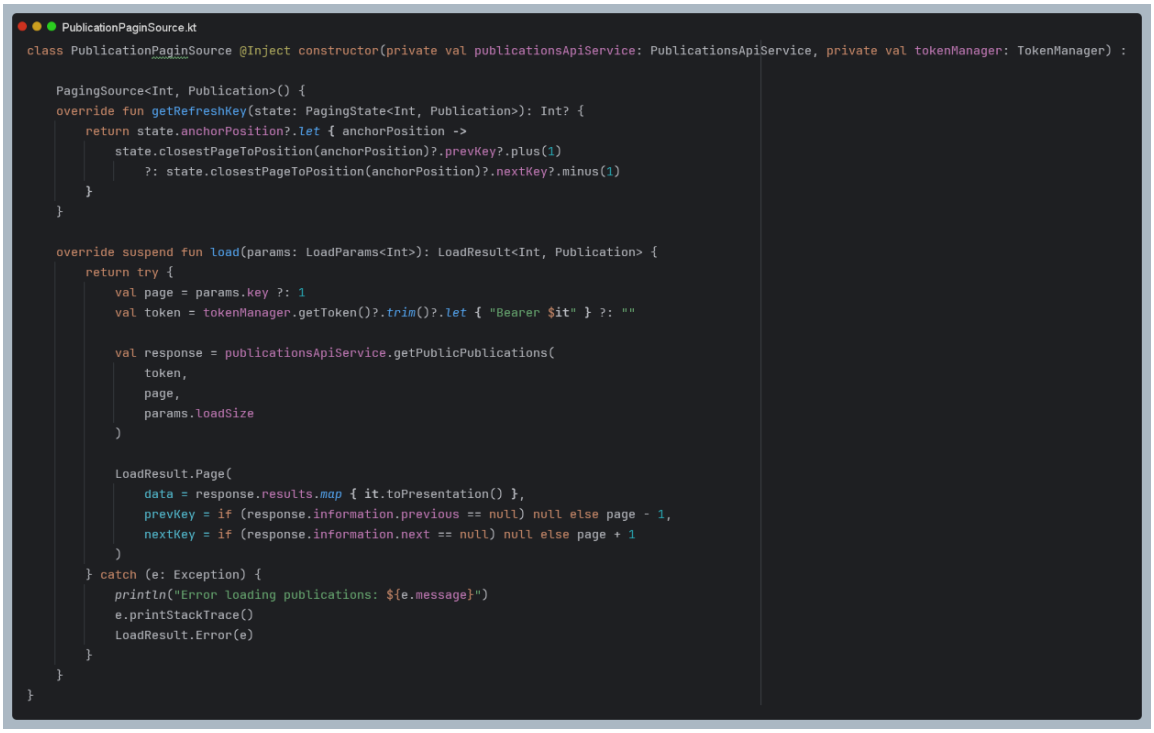


Ilustración 10. Clase PublicationPagingSource

Como podemos observar, la clase PublicationPagingSource extiende PagingSource, e implementa los métodos getRefreshKey() y load(). Esta clase forma parte del sistema de paginación automática que proporciona la librería Paging 3 de Android. Es decir, no somos nosotros quienes solicitamos manualmente cada página de datos, sino que el sistema de paginación se encarga automáticamente de solicitar nuevas páginas a medida que el usuario hace scroll en la interfaz.

El método load() es el encargado de cargar los datos de una página específica. En él se recupera el token de autenticación local y se realiza la llamada a la API con los parámetros adecuados (número de página y tamaño). Luego, los datos obtenidos se transforman al modelo de presentación necesario.

Por su parte, el método getRefreshKey() determina cuál es la clave (es decir, el número de página) más adecuada para refrescar los datos cuando ocurre una actualización o invalidación del contenido, como por ejemplo si el usuario recarga la vista.

La implementación del método getPublications() puede verse en la *Ilustración 11*. Este método, configura y devuelve un flujo reactivo (Flow[21]) de datos paginados (PagingData<Publication>) mediante la clase Pager de la librería Paging 3. En su configuración se especifican tanto el tamaño de página como la distancia de prefetch, y se proporciona una factoría que crea instancias del PublicationPagingSource. Este flujo es el que finalmente se expone al resto de

[21] Flow es un tipo asíncrono de Kotlin que permite emitir múltiples valores secuencialmente, y es ideal para representar flujos de datos que pueden cambiar con el tiempo, como actualizaciones en una base de datos o respuestas paginadas desde una API.

la aplicación para ser consumido de forma eficiente desde la interfaz de usuario, permitiendo así una carga progresiva y optimizada de las publicaciones.

Ilustración 11. Método clase PublicationsRepository

5.1.2 Capa de dominio

La capa de dominio es aquella encargada de hacer de puente entre la de datos y la capa de presentación. Esta capa define qué se puede hacer en la aplicación, es decir, los casos de uso que encapsulan operaciones concretas y específicas del sistema.

Una de sus principales ventajas es que esta capa es completamente independiente del resto de capas, esto es, que da igual el framework que se utilice tanto en la capa de presentación como en la de datos. Esto hace que la implementación sea reutilizable, desacoplada y fácilmente testeable.

El diagrama de casos de uso de la aplicación se muestra en la *Ilustración 13*. Los casos de uso son las acciones del sistema que el usuario puede llevar a cabo. Cada uno encapsula una operación concreta como lo podría ser “obtener publicaciones”. Estos casos de uso reciben los parámetros necesarios y devuelven los resultados a la capa de presentación. Un ejemplo del caso de uso ‘obtener publicaciones públicas’ se presenta en la *Ilustración 12*. Este método es el encargado de devolver un Flow con las publicaciones.

Ilustración 12. Caso de uso obtener publicaciones publicas

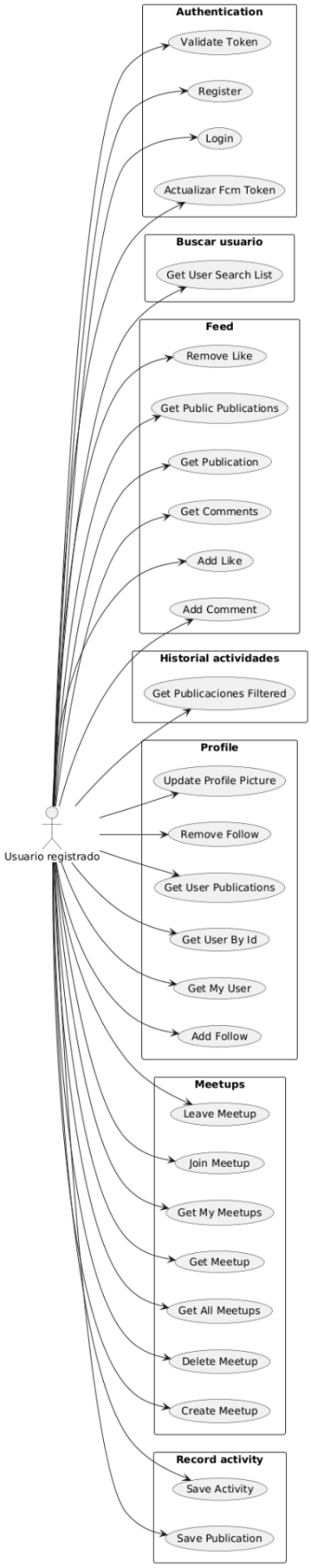


Ilustración 13. Diagrama de casos de uso

5.1.3 Capa de presentación

La capa de presentación es la encargada de la gestión de la interfaz de usuario y de representar visualmente el estado actual de la aplicación. Esta es la capa más cercana al usuario, y su función principal es mostrar datos y capturar interacciones, delegando la lógica al ViewModel.

Vamos a empezar con los composables. Los composables son funciones anotadas con @Composable que permiten implementar interfaces de forma declarativa. En los composables, en vez de definir cómo cambia la UI paso a paso, se define qué debe mostrarse en función de su estado. Por otro lado, tiene recomposición automática, esto es, si cualquier valor del composable cambia, este se recompone automáticamente. Un ejemplo de composable se presenta en la *Ilustración 14*.

ActivityScreen.kt

```kotlin
@Composable
fun ContenidoDetalladoActivity(
    publication: Publication?,
    modifier: Modifier = Modifier,
    activityViewModel: ActivityViewModel
) {
    Column(
        modifier = modifier.verticalScroll(rememberScrollState()),
        verticalArrangement = Arrangement.spacedBy(32.dp)
    ) {
        if (publication == null) {
            Text("No se ha podido cargar la actividad")
        } else {
            CabeceraActividad(
                imagenPerfil = publication.user?.imageUrl!!,
                nombreUsuario = publication.user.username!!,
                fecha = publication.startTime!!,
                tipoActividad = publication.activityType!!
            )
            TituloActividad(titulo = publication.title!!)
            DetallesActividad(publication = publication)
            GraficoVelocidad(publication = publication, activityViewModel = activityViewModel)
            GraficoAltitud(publication = publication, activityViewModel = activityViewModel)
        }
        Log.d("Distancias", publication?.distances.toString())
    }
}
```

Ilustración 14. Ejemplo composable

La función ContenidoDetalladoActivity es una función composable que se encarga de mostrar de forma estructurada la información detallada de una publicación de actividad física (Publication). Si la publicación es nula, se muestra un mensaje de error; en caso contrario, se renderizan distintos componentes como la cabecera con la imagen y nombre del usuario, la fecha, el tipo de actividad, el título, los detalles, y dos gráficos (uno de velocidad y otro de altitud), utilizando los datos de la publicación. Todo el contenido se organiza dentro de una columna con desplazamiento vertical y espaciado entre elementos, y se emplea un ActivityViewModel para gestionar los datos mostrados en los gráficos.

Por otro lado, en la capa de presentación tenemos a los ViewModel, que son los encargados de exponer el estado necesario para la interfaz y de gestionar eventos provenientes de ella. El ViewModel está etiquetado gracias al uso de Hilt con la anotación @HiltViewModel. De esa manera, Hilt se encarga de crear e inyectar el ViewModel de manera automática, lo cual hace el desarrollo mucho más eficiente. A su vez, Hilt permite inyectar otras dependencias al propio ViewModel. Como hemos comentado, lo ideal es que los Composables no tengan ni lógica de negocio ni ninguna variable sensible, por lo tanto, esas variables son almacenadas en el ViewModel, y se permite acceder al composable mediante la clase LiveData. Esta última clase es un observable que sigue el ciclo de vida de los componentes de la UI como Activity, Fragment o Composable. De esta manera se asegura que solo se actualice la interfaz cuando esté activa, evitando errores comunes como NullPointerException o actualizaciones innecesarias.

5.1.4 Herramientas de soporte a la arquitectura

Ya hemos definido las 3 capas de la arquitectura, pero todavía nos faltan dos aspectos clave a tratar, el primero es que sabemos cómo funcionan las vistas de la aplicación, pero no sabemos cómo navegar entre ellas, y el siguiente es que no hemos explicado como acceder detalladamente a la API desde nuestra aplicación.

Primero vamos a tratar la navegación. En la navegación hemos tenido que implementar 2 clases, la primera es “Screens”. Esta clase define todas las ventanas que posee la aplicación. Las ventanas simples se anotan con @Serializable, mientras las ventanas más complejas se definen como data class. Lo que diferencia una ventana simple de una compleja es que la ventana compleja es aquella que debe tener definida una ruta, ya que cada ventana es única. Esto ocurre en aquellas ventanas destinadas a las quedadas, publicaciones, perfiles de usuario y quedadas. En la Ilustración 15 se muestra una pantalla simple (FormularioQuedadas), y una compleja (DetallesQuedada).

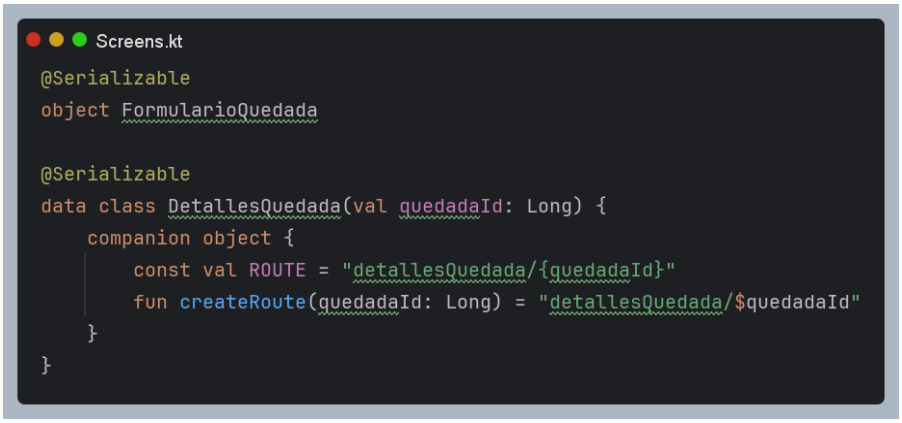


Ilustración 15. Screen simple y compleja

Por otro lado, tenemos la clase NavigationWrapper, que es la encargada de montar toda la pila de navegación. En ella se define un NavHost, el cual contiene

todas las navegaciones posibles desde esa ventana y sus respectivos composables. Además, dentro de cada composable, se recupera el ViewModel correspondiente mediante hiltViewModel(), y se pasan las funciones de navegación y datos necesarios al Composable que representa esa pantalla. Un ejemplo de NavHost simple se muestra en la *Ilustración 16*.

NavigationWrapper.kt

```
composable<Splash> {
    val splashViewModel: SplashViewModel = hiltViewModel()
    SplashScreen(
        navigateToHome = {
            navController.navigate(Home) {
                popUpTo(0)
            }
        },
        navigateToFeed = {
            navController.navigate(Feed) {
                popUpTo(0)
            }
        },
        splashViewModel = splashViewModel
    )
}
```

Ilustración 16. NavHost simple

Sin embargo, cuando una pantalla requiere recibir parámetros, como el ID de una publicación o un usuario, se utiliza navArgument para extraerlos del BackStackEntry. Un ejemplo de NavHost con argumentos se presenta en la *Ilustración 17*.

NavigationWrapper.kt

```
composable<Feed> {
    val feedViewModel: FeedViewModel = hiltViewModel()
    val postInteractionViewModel: PostInteractionViewModel = hiltViewModel()
    FeedScreen(
        navigateToStartRecordActivity = { navController.navigate(RecordActivity) },
        navigateToActivity = { publicationId ->
            navController.navigate("activity/$publicationId")
        },
        feedViewModel = feedViewModel,
        navigateToProfile = { profileId ->
            navController.navigate("profile/$profileId")
        },
        navigateToSearch = { navController.navigate(Search) },
        postInteractionViewModel = postInteractionViewModel,
        navigateToHistorial = { navController.navigate(Historial) },
        navigateToQuedadas = { navController.navigate(Quedadas) }
    )
}
```

Ilustración 17. NavHost con argumentos

6. Pruebas

Las pruebas son una parte esencial del desarrollo software, ya que te permiten comprobar si tu software realiza las tareas necesarias de manera correcta. En este apartado se detallarán las distintas pruebas que se han realizado, tanto en el servicio REST como en la aplicación Android.

Primero vamos a explicar los principales tipos de pruebas que se han realizado:

- *Pruebas unitarias*: Este tipo de pruebas se centran en verificar el correcto funcionamiento de unidades pequeñas de código como pueden ser métodos, funciones o clases de manera aislada. Las pruebas unitarias permiten detectar errores de lógica muy temprano en el desarrollo y facilitan la refactorización y el mantenimiento del código.

- *Pruebas de integración*: Este tipo de pruebas se realizan para verificar la interacción y funcionamiento en conjunto de los diferentes módulos y componentes del sistema. Este nivel de pruebas es fundamental para garantizar la cohesión y el funcionamiento conjunto de los subsistemas.

- *Pruebas de interfaz de usuario (UI)*: Estas pruebas son las encargadas de validar el correcto funcionamiento de la capa visual de la aplicación asegurándose que brinde una experiencia de usuario adecuada. Se comprueban aspectos como la correcta visualización de elementos, la respuesta a interacciones, la navegación entre pantallas y la validación de datos introducidos. Son fundamentales para garantizar la usabilidad y la accesibilidad del producto.

- *Pruebas de sistema*: Estas pruebas se realizan para verificar el funcionamiento global del sistema, se enfocan en validar que el sistema cumpla los requisitos no funcionales establecidos.

- *Pruebas de aceptación:* Estas pruebas se realizan para saber si el sistema cumple con las expectativas y requisitos planteados por el cliente o usuario final.

6.1 Pruebas servicio REST

Debido a limitaciones temporales, la validación del correcto funcionamiento del servicio REST se ha llevado a cabo combinando el uso de pruebas unitarias con comprobaciones manuales mediante la herramienta Postman. Esta aplicación permite simular peticiones HTTP a los distintos endpoints de la API, facilitando la verificación de su comportamiento en tiempo de ejecución.

Se ha realizado con Postman la comprobación de todos los endpoints de la API, validando así los formatos de entrada y salida JSON, garantizando que la API cumple con las especificaciones definidas en el diseño. En la *Ilustración 18* se muestra el uso de Postman para el probar el endpoint “GET user”.

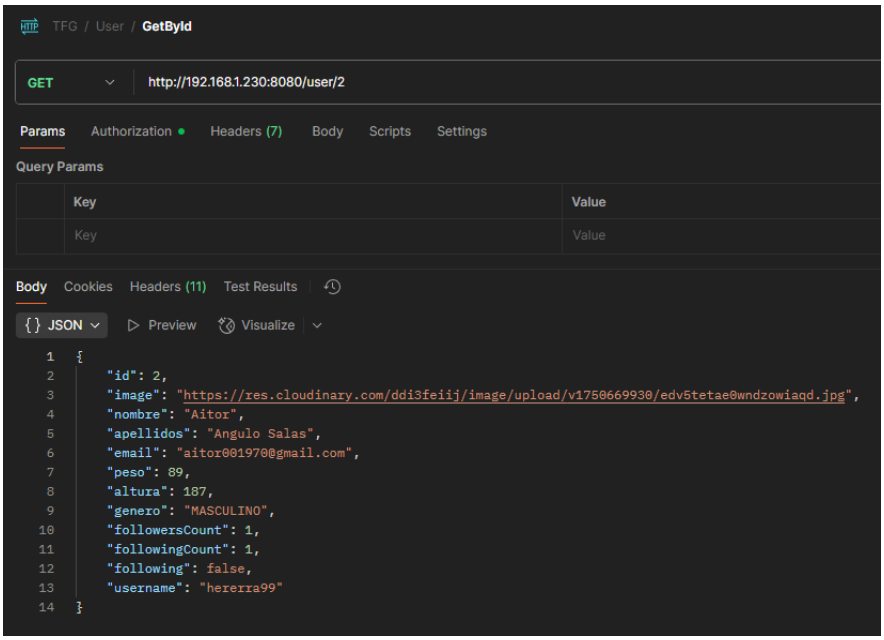


Ilustración 18. Uso de Postman

Además, se han desarrollado pruebas unitarias para varios servicios de la aplicación, lo que ha permitido verificar de forma automatizada la lógica de negocio de manera aislada, mejorando así la robustez del sistema.

A continuación, tanto en la *Tabla 11* como en la *Tabla 12* se presentan detalladamente los casos de prueba diseñados para el ActivityService, donde se describen tanto las entradas como salidas esperadas. Un ejemplo de un test creado se muestra en la *Ilustración 19*.

+createActivity(activityDTO: ActivityDTO, username: String): ActivityDTO		
Identificador	**Entrada**	**Salida esperada**
UAP.1a	ActivityDTO válido y nombre de usuario existente	Objeto ActivityDTO con los datos de la actividad creada
UAP.1b	ActivityDTO válido y nombre de usuario inexistente ("fantasma")	Excepción UsernameNotFoundException

Tabla 11. Casos de prueba método createActivity(), clase ActivityService

+getActivities(): List<ActivityDTO>		
Identificador	**Entrada**	**Salida esperada**
UAP.2a	Existen actividades en el sistema	Lista con objetos ActivityDTO
UAP.2b	No existen actividades	Excepción NoSuchElementException

Tabla 12. Casos de prueba método getActivities(), clase ActivityService

Ilustración 19. Test ActivityService

6.2 Pruebas aplicación Android

En este apartado las pruebas unitarias que se han desarrollado en la aplicación corresponden con todos los métodos de las clases Utils, que son aquellos más propensos a este tipo de pruebas. En total se han realizado pruebas para 5 clases Utils y 39 métodos en total. A continuación, presentan detalladamente los casos de prueba para la clase ValidationUtils. Un ejemplo de un test de la clase ValidationUtils se presenta en la *Ilustración 20*. En la *Tabla 13* y *Tabla 14* se muestran los casos de prueba del método isPasswordSecure() e isUsernameValid.

+ isPasswordSecure(password: String): Boolean		
Identificador	**Entrada**	**Salida**
U1.a	"Password123"	True
U1.b	"pass123"	False
U1.c	"PASSWORD123"	False
U1.d	"Password"	False
U1.e	"Pass1234"	True

Tabla 13. Caso de prueba método isPasswordSecure, clase ValidationUtils

+ isUsernameValid(username: String): Boolean		
Identificador	**Entrada**	**Salida**
U2.a	"user_name"	True
U2.b	"user-name"	True
U2.c	"user name"	False
U2.d	"us"	False
U2.e	"asdasdadadasdasdadasad"	False
U2.f	"user@name"	False
U2.g	"user123"	True

Tabla 14. Caso de prueba método isUsernameValid, clase ValidationUtils

ValidationUtilsTest.kt

```
@Test
fun contrasenaConMayusculasMinusculasYDigitosEsValida() {
    Assertions.assertTrue(isPasswordSecure("Password123"))
}
```

Ilustración 20. Test clase ValidationUtils

El método de prueba contrasenaConMayusculasMinusculasYDigitosEsValida tiene como objetivo verificar que la función isPasswordSecure valida correctamente una contraseña que cumple con los criterios mínimos de seguridad. En este caso, se utiliza la contraseña "Password123", la cual contiene letras mayúsculas, minúsculas y dígitos, cumpliendo así con los requisitos básicos de seguridad. La prueba utiliza el método assertTrue de JUnit para asegurar que el resultado devuelto por isPasswordSecure sea true, confirmando que la contraseña proporcionada es considerada segura.

Para las pruebas de interfaz se ha optado por comprobar la vista de registro de usuario, ya que es la parte más crítica de toda la aplicación puesto que se trabaja con datos sensibles. Además, en esta vista hay una gran cantidad de comprobaciones de errores que deben ser verificadas. Se han realizado 6 test

en este apartado. Las pruebas de interfaz consisten en simular el comportamiento de una persona utilizando la aplicación, interactuando con los distintos elementos visuales (como campos de texto o botones) de forma automatizada. Estas pruebas, que también pueden considerarse pruebas de integración, permiten comprobar que la lógica del sistema y la interfaz funcionan correctamente de manera conjunta. Para su implementación se ha utilizado el framework de testing de Jetpack Compose, que permite interactuar y verificar mediante código los componentes de la interfaz de usuario de forma precisa y eficiente. Los casos de prueba se pueden observar en la *Tabla 15* y una implementación en la *Ilustración 21*.

Identificador	Entrada	Resultado
UI1.a	Username: “aitor” Email: “test@gmail.com” Password1: “12345678Aa” Password2: “12345678Aa” Name: “Aitor” Surname: “Angulo” Date: “27/04/2003” Gender: “Male” Weight: “70” Height: “170”	"Verifica la cuenta con el email recibido."
UI1.b	Username: “aitor” Email: “test@.com” Password1: “12345678Aa” Password2: “12345678Aa” Name: “Aitor” Surname: “Angulo” Date: “27/04/2003” Gender: “Male” Weight: “70” Height: “170”	"El correo electrónico no es válido.”
UI1.c	Username: “a” Email: “test@.com” Password1: “12345678Aa” Password2: “12345678Aa” Name: “Aitor” Surname: “Angulo” Date: “27/04/2003” Gender: “Male” Weight: “70” Height: “170”	"El nombre de usuario no es válido."
UI1.d	Username: “aitor” Email: “test@.com” Password1: “12345678Aa” Password2: “kdfbheha129631A” Name: “Aitor” Surname: “Angulo” Date: “27/04/2003” Gender: “Male”	"Las contraseñas no coinciden.”

	Weight: “70” Height: “170”	
UI1.e	Username: “aitor” Email: “test@.com” Password1: “12” Password2: “12” Name: “Aitor” Surname: “Angulo” Date: “27/04/2003” Gender: “Male” Weight: “70” Height: “170”	"Formato de contraseña incorrecto"
UI1.f	Username: “aitor” Email: “test@.com” Password1: “12345678Aa” Password2: “12345678Aa” Name: “Aitor” Surname: “Angulo” Date: “27/04/2150” Gender: “Male” Weight: “70” Height: “170”	"La fecha de nacimiento no puede ser futura.”
UI1.g	Username: Email: Password1: Password2: Name: Surname: Date: Gender: Weight: Height:	"Todos los campos son obligatorios."

Tabla 15. Casos de prueba UI registrar usuario

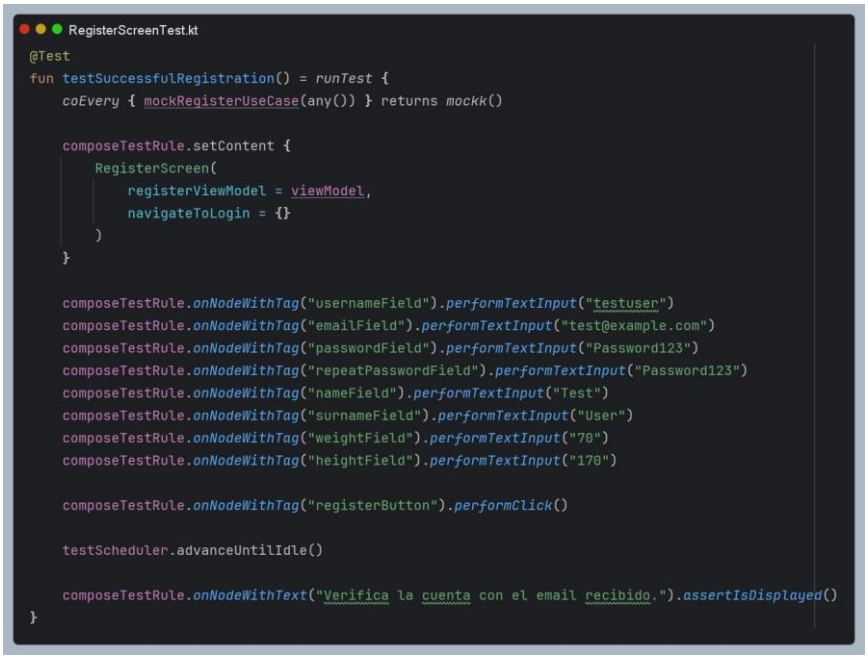


Ilustración 21. Test clase RegisterScreenTest

El método testSuccessfulRegistration verifica el flujo completo de un registro exitoso en la interfaz de usuario utilizando Jetpack Compose. Simula la interacción de un usuario rellenando correctamente todos los campos del formulario de registro y pulsando el botón de registro para enviar la información. Se utiliza un mockRegisterUseCase para simular la lógica de registro sin depender de la lógica de la API en el backend. Finalmente, se comprueba que el mensaje mostrado es el correcto indicando que se debe verificar la cuenta por correo electrónico.

6.3 Pruebas de sistema de la Aplicación Android

En este apartado se detallarán las pruebas realizadas para cumplir los diversos requisitos no funcionales establecidos inicialmente para la aplicación.

6.3.1 Pruebas de portabilidad

El requisito de portabilidad que debía cumplir la aplicación para ser compatible con diferentes versiones de Android desde la 8.1 Oreo hasta la actual.

Para comprobarlo se ha utilizado un emulador con Android 8.1 y otro con Android 15.0. Se verificó que la aplicación funcionaba correctamente en ambos entornos, sin presentar ningún fallo. Con esto se consideró que el requisito de portabilidad se cumplía.

6.3.2 Pruebas de usabilidad

El requisito de usabilidad era que, aparte de ser intuitiva, siguiera las guías de diseño de Android y facilitara la navegación por las distintas secciones como registro de actividades, quedadas y perfil, debía adaptarse bien al modo oscuro. Como podemos ver en las siguientes imágenes, el modo oscuro se adapta e integra bien en la aplicación, además de que se han aplicado principios de diseño para componer una interfaz que se adhiera a los principios de diseño de Android.

Todo esto ha sido mejorado gracias a los comentarios de retroalimentación de los usuarios que han utilizado la aplicación. En la *Ilustración 22* podemos observar la aplicación tanto en modo claro como oscuro.

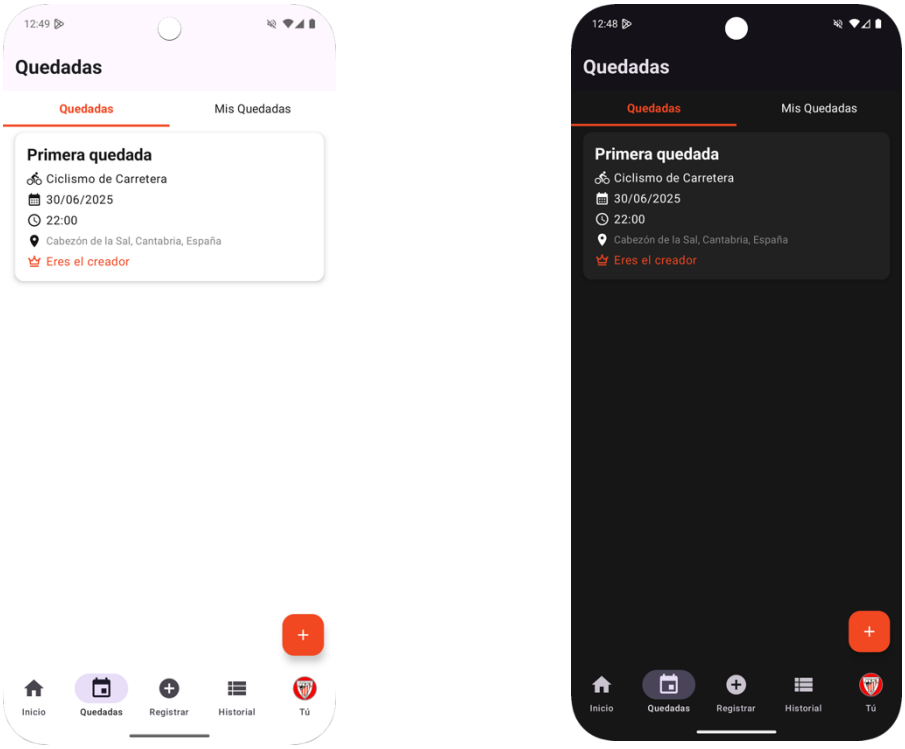


Ilustración 22. Interfaz de la aplicación modo claro/oscuro

6.4 Integración continua con GitHub Actions

Para garantizar la calidad del software y automatizar la validación de cada cambio, se configuró un flujo de integración continua con GitHub Actions. De esta manera, cada vez que se realiza un push o un pull request a main o develop se ejecutan las pruebas. Para ello se ha configurado un archivo workflow en el cual se definen los pasos que se tienen que ejecutar. Primeramente, se descarga el repositorio y se configura la máquina virtual de Ubuntu. Una vez configurado todo se ejecutan los test unitarios y posteriormente las pruebas de UI. Una vez ejecutado todo se muestra el resultado de la ejecución en el repositorio, en caso de que pase todo satisfactoriamente se muestra un check (✔) en el commit y en el caso contrario se muestra una cruz (✘) en el commit y se indica en que test ha fallado.

7. Despliegue del backend

Para garantizar que el backend esté siempre disponible y accesible desde cualquier parte del mundo, se ha decidido desplegar la API en la nube utilizando Microsoft Azure. Microsoft Azure es una plataforma en la nube que destaca por su capacidad de escalado y sus opciones de seguridad avanzadas. La organización del despliegue la podemos observar en la *Ilustración 23* que a continuación será explicada detalladamente.

Los servicios utilizados en Microsoft Azure son la máquina virtual Ubuntu[23] para desplegar la aplicación Spring Boot y Azure SQL Database para generar la base de datos MySQL.

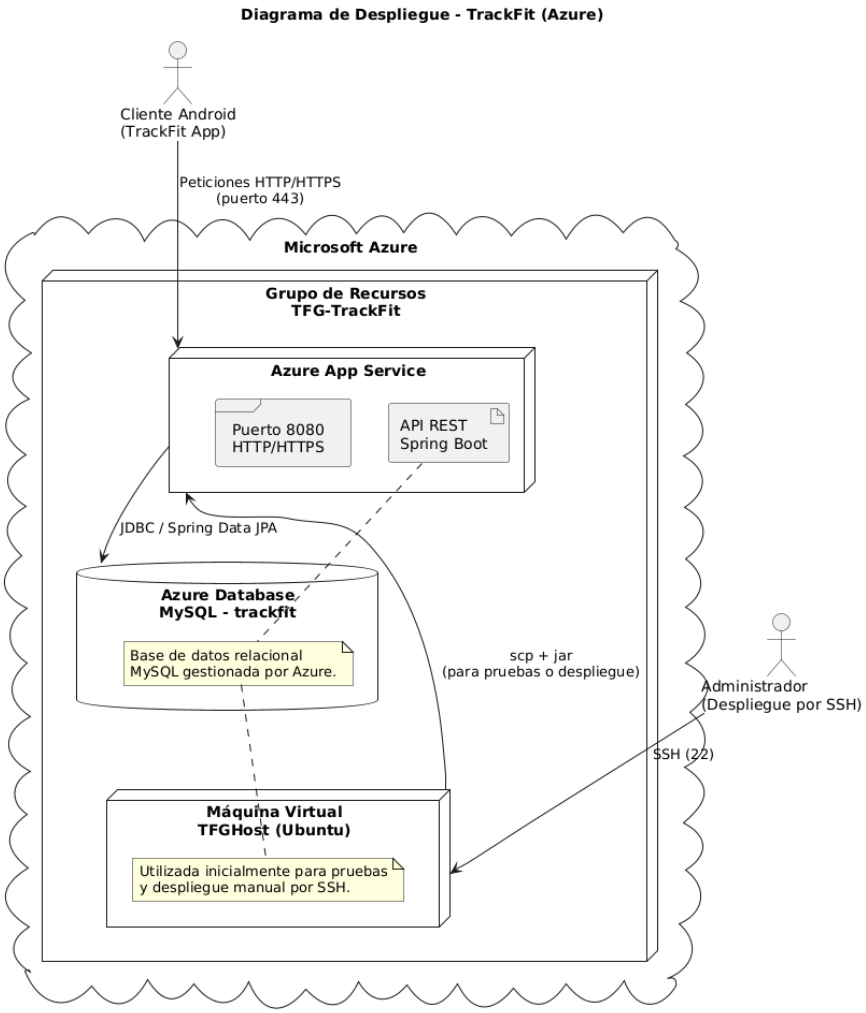


Ilustración 23. Diagrama de despliegue TrackFit

7.1 Flujo de despliegue

El primero paso del despliegue fue crear un grupo de recursos, denominado “TFG-TrackFit”, en el que se creó la máquina virtual. Para su configuración se asignó el nombre “TFGHost”, se seleccionó un servidor Ubuntu con 1 CPU, se configuró la clave SSH y se definieron las reglas de entrada para los puertos SSH (22) y HTTPS (443). Una vez creada la máquina, se configuró el acceso al puerto 8080, como se puede observar en la *Ilustración 24.*

[23] Ubuntu es una distribución GNU/Linux basada en Debian GNU/Linux, que incluye principalmente software libre y de código abierto.

300	HTTP	80	TCP	Cualquiera	Cualquiera	Allow	
320	HTTPS	443	TCP	Cualquiera	Cualquiera	Allow	
1000	AllowAnyCustom22Inbound	22	TCP	Cualquiera	Cualquiera	Allow	
1010	AllowAnyCustom8080Inbound	8080	TCP	Cualquiera	Cualquiera	Allow	

Ilustración 24. Reglas de red de la máquina virtual

Tras la reación de la máquina virtual, fue necesario conectarse a ella para desplegar el servicio REST. Para esta tarea, se ejecutaron una serie de comandos que se describen la *Tabla 16.*

Comando	**Función**
ssh -i <PathClavePrivada> azureuser@<IP>	Inicia sesión en la máquina virtual a través de SSH usando la clave privada y la IP pública asignada.
sudo apt install openjdk-17-jre-headless	Instala Java 17 en la máquina virtual, necesario para ejecutar aplicaciones .jar.
scp -i <PathClavePrivada> <archivo.jar> azureuser@<IP>:/home/azureuser	Copia el archivo .jar desde tu máquina local al directorio del usuario en la máquina virtual, utilizando scp (secure copy).
java -jar <archivo.jar>	Ejecuta el archivo .jar en la máquina virtual usando Java.
http://<IP_Pública>:8080	Dirección donde estará disponible el servicio una vez ejecutado el archivo .jar. Asegúrate de que el puerto 8080 esté habilitado en el grupo de seguridad de red (NSG) de Azure.

Tabla 16. Comandos máquina virtual Azure

Una vez desplegado el servicio, y dado que este hacía uso de una base de datos Azure SQL Database, se procedió a crear el recurso correspondiente. Para ello, se asignó un nombre a la BBDD y al servidor, se configuró la autenticación SQL con su usuario y contraseña, y en la configuración de red se habilitó un punto de acceso público permitiendo el acceso desde los servicios de Azure y la IP del cliente local. Finalmente, se modificó el archivo application.properties del servicio para reflejar esta configuración. El archivo application.properties se muestra en la *Ilustración 25* y el backend desplegado en Microsoft Azure puede verse en la *Ilustración 26*.

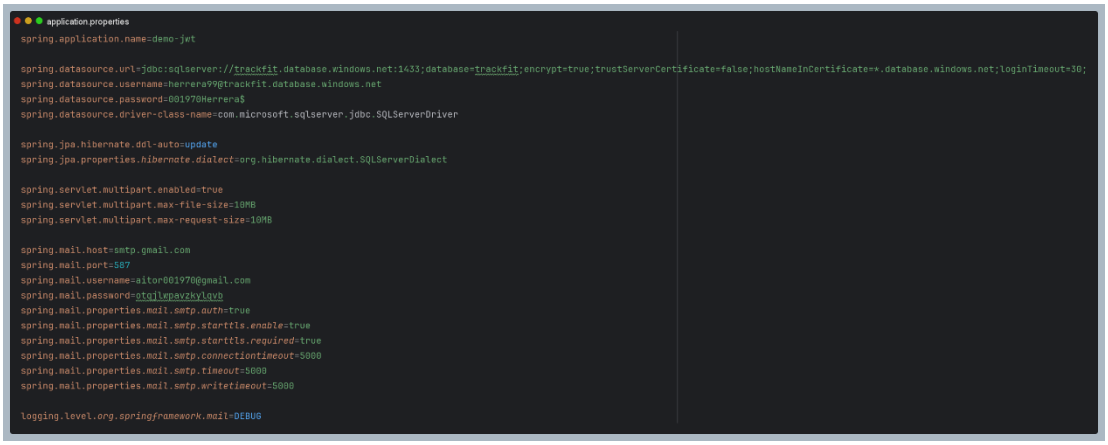


Ilustración 25. Archivo application.properties

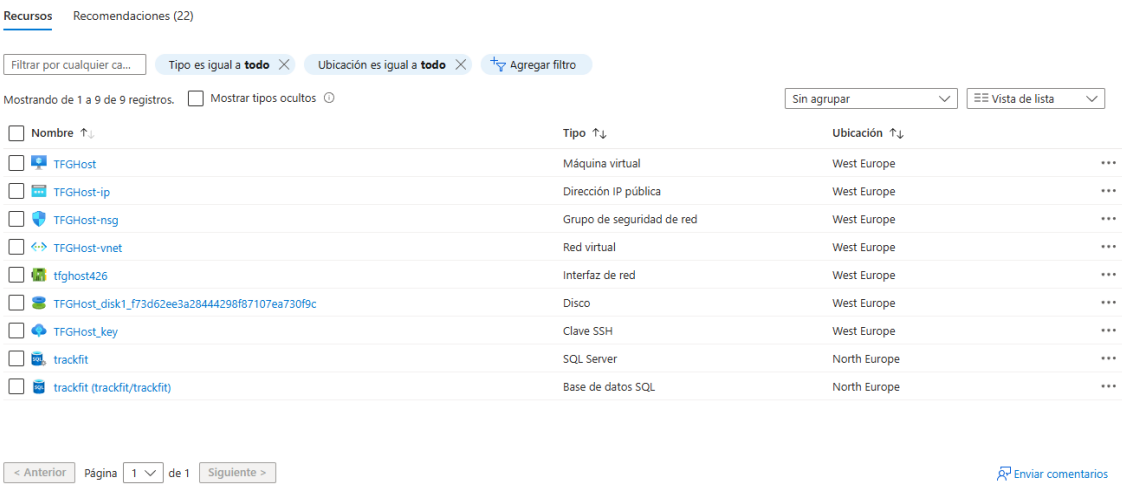


Ilustración 26. Backend desplegado en Microsoft Azure

8. Conclusiones

El desarrollo de esta aplicación ha supuesto un reto desde concebir la idea hasta la implementación tanto del backend como del frontend. El objetivo principal del proyecto fue combinar los conocimientos adquiridos durante el grado junto con nuevas tecnologías y herramientas, para acometer el desarrollo de una aplicación compleja, inspirada en plataformas comerciales de referencia como Strava. De esta forma se pudieron abordar retos realistas de carácter técnico y de diseño que una solución de estas características implica.

A lo largo del proyecto, se ha conseguido implementar una solución funcional y coherente, basado siempre en una arquitectura limpia y modular que permitirá su progreso en el futuro en caso de ser necesario.

Entre las funcionalidades más destacadas de la aplicación se encuentran el registro y seguimiento de entrenamientos, la creación y participación en quedadas, el feed de publicaciones públicas, la posibilidad de seguir a otros usuarios, recibir notificaciones, y la gestión del perfil.

En definitiva, los objetivos planteados al inicio del proyecto se han cumplido satisfactoriamente, resultando en una aplicación útil, atractiva y con potencial para evolucionar en el futuro.

Por último, para mostrar la apariencia visual de la aplicación finalizada se ha incluido un Anexo 1 en el cual se muestran las capturas del resultado final de la aplicación.

8.1 Trabajos futuros

Si bien TrackFit ya nos ofrece una base sólida y funcional, existe margen de mejora y expansión. A continuación se van a describir algunas de las posibles futuras actualizaciones:

1. *Implementar objetivos del usuario*: Una de las funcionalidades que quedó pendiente fue que el usuario pudiera establecer unos objetivos semanales/mensuales/anuales y así analizar su progreso.

2. *Sincronización con otras plataformas deportivas*: Se contempla la integración con aplicaciones populares como Garmin Connect o Wahoo, permitiendo importar actividades directamente desde estos servicios y enriquecer así el historial de entrenamientos del usuario.

3. *Subida de actividades mediante archivos GPX*: Actualmente esta funcionalidad se limita a las quedadas, pero sería interesante permitir a los usuarios registrar entrenamientos subiendo directamente un archivo .gpx, lo cual facilitaría la portabilidad entre plataformas y dispositivos.

4. *Mayor personalización del perfil*: Se pretende ampliar las opciones de personalización del perfil de usuario más allá de la imagen, incorporando aspectos como biografía, intereses deportivos, estadísticas destacadas o logros personales.

5. *Conexión con sensores externos*: Una mejora relevante sería la posibilidad de conectar sensores (como pulsómetros o podómetros) mediante Bluetooth o ANT+, de forma que durante el registro de una actividad se puedan capturar métricas adicionales como la frecuencia cardíaca o la cadencia.

6. *Agregar la opción de compartir*: Se pretende implementar la funcionalidad de compartir tanto publicaciones, perfiles como quedadas, permitiendo a los usuarios difundir contenido dentro y fuera de la aplicación.

7. Aumento del número de pruebas complementando así las pruebas unitarias, permitiendo validar la correcta comunicación entre componentes del sistema. De este modo, se podrá garantizar una cobertura más completa del comportamiento de la API, especialmente dentro de un entorno de integración y despliegue.

9. Bibliografía

[1] Android Developers. (2023). Jetpack Compose. Recuperado de: https://developer.android.com/jetpack/compose

[2] Android Developers. (2019). Kotlin para desarrolladores de Android. Recuperado de: https://developer.android.com/kotlin

[3] Cloudinary. (s.f.). Cloudinary: Image and Video Management. Recuperado de: https://cloudinary.com/

[4] GitHub. (s.f.). Projects - GitHub Features. Recuperado de: https://github.com/features/issues

[5] GitHub. (s.f.). Repositorio del proyecto TrackFit. Recuperado de: https://github.com/HERRERA99/TFG-AppTrackActividades

[6] Git SCM. (s.f.). Git - Distributed version control system. Recuperado de: https://git-scm.com/

[7] JetBrains. (s.f.). Kotlin Programming Language. Recuperado de: https://kotlinlang.org/

[8] JetBrains. (s.f.). IntelliJ IDEA. Recuperado de: https://www.jetbrains.com/idea/

[9] Microsoft Azure. (s.f.). Microsoft Azure: Cloud Computing Services. Recuperado de: https://azure.microsoft.com/es-es/

[10] MockFlow. (s.f.). MockFlow - Wireframe Tools. Recuperado de: https://www.mockflow.com/

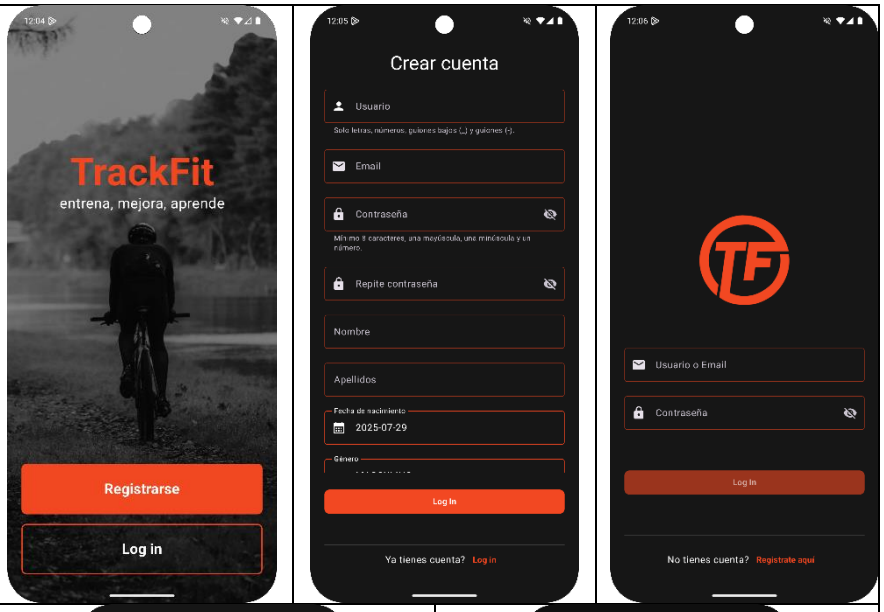
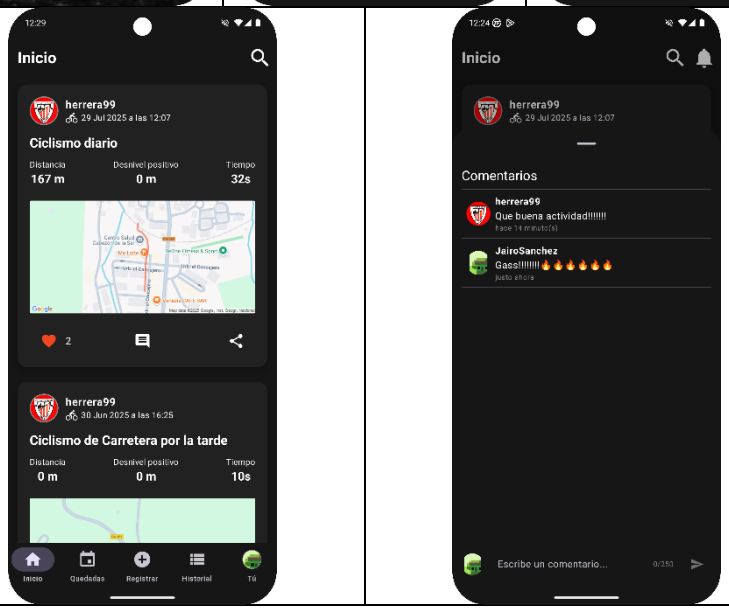
[11] MySQL. (s.f.). MySQL Workbench. Recuperado de: https://www.mysql.com/products/workbench/

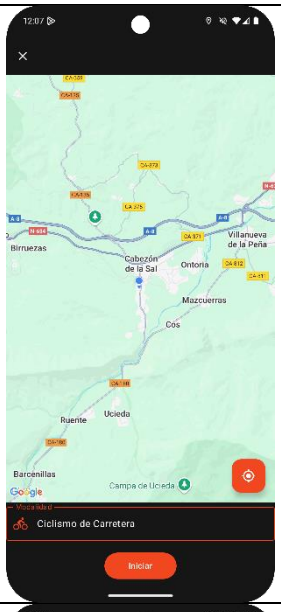
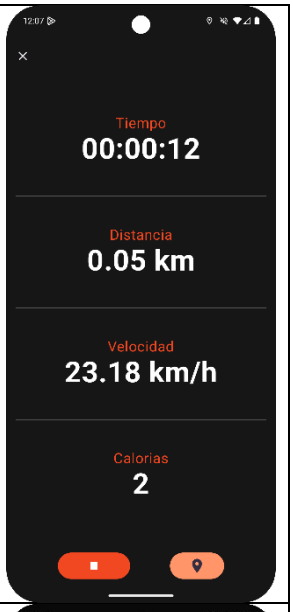
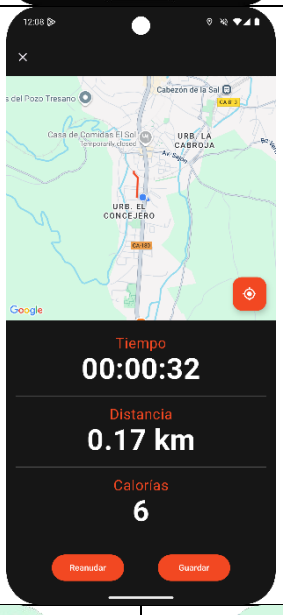
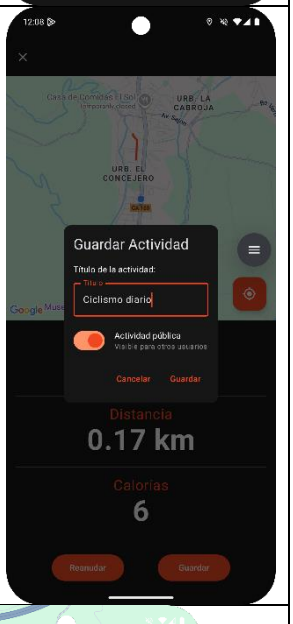
[12] MySQL. (s.f.). MySQL Database. Recuperado de: https://www.mysql.com/

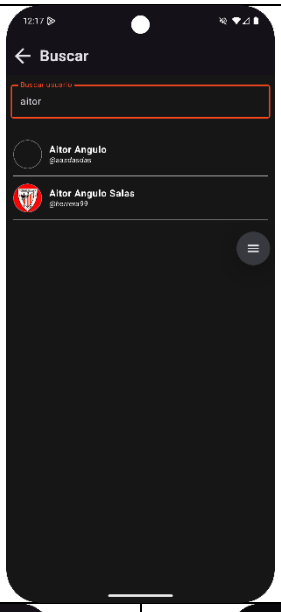
[13] Postman. (s.f.). Postman API Platform. Recuperado de: https://www.postman.com/

[14] Spring. (s.f.). Spring Boot Project. Recuperado de: https://spring.io/projects/spring-boot

Anexo 1: Resultado final de la aplicación

<table>
<tr>
<td>Ingreso a la aplicación</td>
<td colspan="3"> 
</td>
</tr>
<tr>
<td>Feed</td>
<td colspan="3"> 
</td>
</tr>
</table>

Registro actividad			
			
Detalles de la actividad			

Buscar usuario		
Historial		
Perfil		

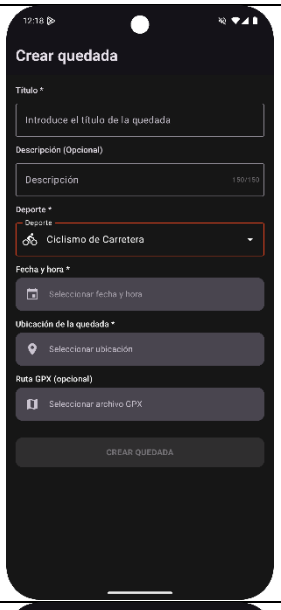
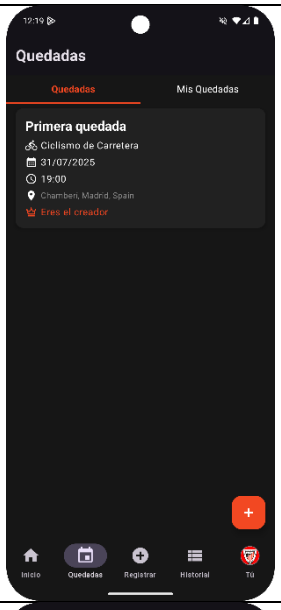
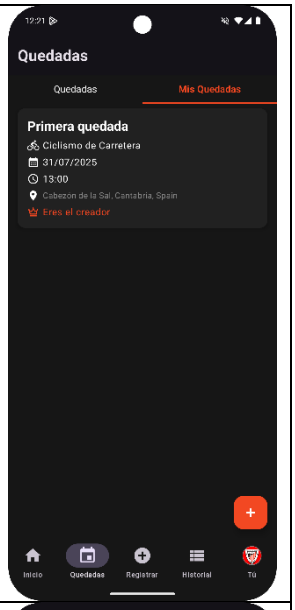
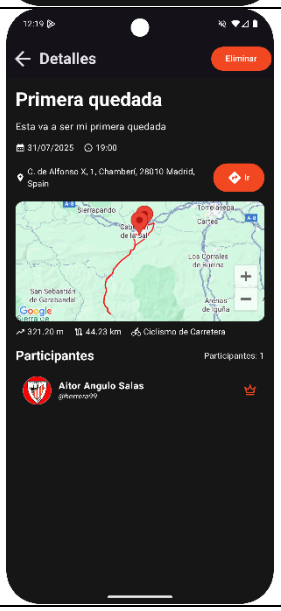
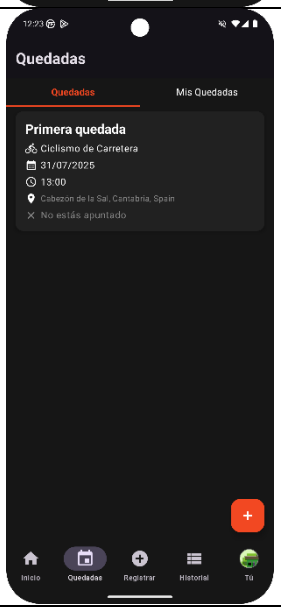
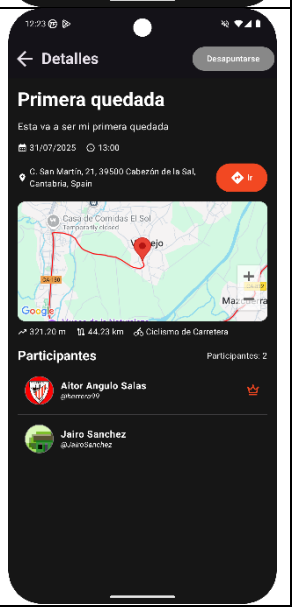
Quedadas			
			

Tabla 17. Imágenes de la aplicación final