UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

recipEat
Exploración y planificación automática de recetas

Presentado por Jimena Arnaiz González
en Universidad de Burgos
a 9 de junio de 2025
Tutora: Ana Serrano Mamolar

Resumen

En este proyecto, se ha desarrollado *recipEat*, una aplicación móvil Android nativa enfocada en la gestión de recetas y la planificación de comidas, diseñada para ayudar a los usuarios a descubrir, organizar y personalizar sus platos con los ingredientes disponibles en su hogar. La aplicación permite realizar búsquedas por nombre o por ingredientes, generar planes semanales, ver un historial de recetas cocinadas en los últimos 7 o 30 días. La pantalla de inicio se actualiza automáticamente para mostrar recetas recomendadas basadas en las preferencias del usuario. Además, los usuarios pueden crear sus propias recetas, guardarlas como favoritas y editar su perfil.

Una característica destacada de *recipEat* es su capacidad de funcionar en modo *offline*, permitiendo a los usuarios acceder a sus recetas favoritas, las recetas creadas por ellos y a las últimas 15 recetas vistas, incluso cuando no tienen conexión a Internet. Los datos se sincronizan con *Firebase* cuando se restablece la conexión, asegurando que toda la información esté siempre actualizada. La *app* utiliza la API de *Spoonacular* para obtener recetas y detalles adicionales, proporcionando una experiencia fluida y personalizada para cada usuario.

Descriptores

aplicación móvil, gestión de recetas, Android, *Kotlin*, *Firebase*, API, *Jetpack Compose*, búsqueda por ingredientes, autenticación de usuarios, recetas personalizadas, planificación de comidas, instrucciones, recetas favoritas, interfaz de usuario, almacenamiento local, modo *offline*, usabilidad ...

Abstract

In this project, *recipEat* has been developed, an Android mobile application focused on recipe management and meal planning, designed to help users discover, organise and customise their dishes using the ingredients available at home. The app allows users to search by name or ingredients, generate weekly plans, and view a history of recipes cooked in the last 7 or 30 days. The home screen automatically updates to show recommended recipes based on the user's preferences. Additionally, users can create their own recipes, save them as favorites, and edit their profile.

A standout feature of *recipEat* is its ability to operate in offline mode, allowing users to access their favourite recipes, recipes they have created and the last 15 recipes viewed, even when they are not connected to the Internet. Data is synchronised with Firebase when the connection is re-established, ensuring that all information is always up to date. The app uses the Spoonacular API to fetch recipes and additional details, providing a seamless and personalised experience for each user...

Keywords

mobile application, recipe management, Android, Kotlin, Firebase, API, Jetpack Compose, search by ingredients, user authentication, personalized recipes, meal planning, steps, favorite recipes, user interface, local storage, offline, usability...

Índice general

Índice general iii

Índice de figuras v

1. Introducción 1

2. Objetivos del proyecto 3

3. Conceptos teóricos 7
- 3.1. Desarrollo de Aplicaciones Móviles . . . 7
- 3.2. APIs . . . 8
- 3.3. Arquitectura de *Software*: MVVM . . . 8
- 3.4. Programación Asíncrona con corrutinas . . . 9
- 3.5. Gestión del Ciclo de Vida . . . 9
- 3.6. Persistencia de Datos . . . 9
- 3.7. Sincronización en Segundo Plano . . . 10

4. Técnicas y herramientas 11
- 4.1. Técnicas . . . 11
- 4.2. Herramientas . . . 12
- 4.3. Bibliotecas y frameworks . . . 16
- 4.4. APIs y servicios externos . . . 19
- 4.5. Herramientas para escribir la memoria . . . 21

5. Aspectos relevantes del desarrollo del proyecto 25
- 5.1. Inicio del Proyecto . . . 25
- 5.2. Desarrollo del *frontend* . . . 25

5.3. Desarrollo del *backend* 28
5.4. Despliegue 37
5.5. Testing 38

6. Trabajos relacionados **43**
6.1. Cookpad 43
6.2. Ekilu 43
6.3. Yummly 44

7. Conclusiones y Líneas de trabajo futuras **45**
7.1. Conclusiones 45
7.2. Líneas de trabajo futuras 46

Bibliografía **49**

Índice de figuras

4.1. Tablero Kanban *GitHub Projects* 12
4.2. Entorno de desarrollo *Android Studio* 14
4.3. Resumen general del análisis de calidad del código 15
4.4. Resumen *Report Tests* 15
4.5. Métrica de cobertura 16
4.6. Firebase Firestore 20
4.7. Firebase Authentication 20
5.1. Fragmento de código que gestiona la navegación en la *app* . . . 26
5.2. *UI Login* Vertical 27
5.3. *UI Login* Horizontal 28
5.4. Llamadas a la API Spoonacular 30
5.5. Función para mapear el modelo *ApiReceta* a *Receta* 31
5.6. Tabla Recetas de *Room* 35
5.7. Tabla Recientes de *Room* 35
5.8. Tabla Favoritos de *Room* 36
5.9. Pantalla Home en modo *offline* 37
5.10. Pantalla de perfil en modo *offline* 37
5.11. Test de *Login* con credenciales válidas 40
5.12. Test de *Login* con contraseña vacía 40

1. Introducción

Decidir qué cocinar cada día puede ser una tarea complicada. Muchas personas se enfrentan a la falta de ideas, el desperdicio de ingredientes que no saben cómo aprovechar o la dificultad de encontrar recetas que se adapten a lo que tienen disponible. *recipEat* nace con el objetivo de facilitar esta tarea al proporcionar una manera rápida y eficiente de encontrar recetas basadas en ingredientes, simplificando la planificación de comidas y reduciendo el desperdicio de alimentos.

La aplicación permite a los usuarios explorar recetas de la API *Spoonacular*, gestionar su lista de recetas favoritas y registrar un historial de cocina. También se incluye un planificador semanal de comidas que facilita la organización del menú diario.

El trabajo está compuesto por dos documentos principales: la memoria del proyecto y un conjunto de anexos. Ambos se complementan con el objetivo de ofrecer una documentación completa y detallada del desarrollo realizado.

Memoria

1. **Introducción**: Breve presentación del proyecto y de la estructura.
2. **Objetivos del proyecto**: Explicación de las metas a alcanzar con el desarrollo del proyecto.
3. **Conceptos teóricos**: Explicación de fundamentos teóricos para la comprensión y desarrollo del proyecto.
4. **Técnicas y herramientas**: Tecnologías y herramientas utilizadas en el proyecto.

5. **Aspectos relevantes del desarrollo**: Elementos clave y desafíos del proceso de desarrollo.
6. **Trabajos relacionados**: Referencias a proyectos similares utilizados como guía.
7. **Conclusiones y líneas de trabajo futuras**: Resultados finales y posibles mejoras futuras.

Anexos

A. **Plan de Proyecto Software**: Evalúa la viabilidad económica y legal del proyecto, y establece la planificación temporal.

B. **Especificación de Requisitos**: Identifica y describe los requisitos funcionales y no funcionales del proyecto.

C. **Especificación de Diseño**: Detalla las estructuras de datos, la arquitectura y las decisiones técnicas del sistema.

D. **Documentación Técnica de Programación**: Conjunto de recursos que facilita la comprensión y mantenimiento del código por futuros desarrolladores.

E. **Documentación de Usuario**: Manual que explica al usuario final cómo utilizar la aplicación correctamente.

F. **Sostenibilización curricular**: Reflexión personal sobre la integración y aplicación de competencias relacionadas con la sostenibilidad a lo largo del desarrollo de la aplicación.

2. Objetivos del proyecto

El presente proyecto tiene como finalidad el desarrollo de una aplicación móvil Android nativa que ayude a los usuarios en la gestión diaria de sus comidas, permitiendo descubrir recetas basadas en los ingredientes que ya poseen en casa. Bajo el nombre de *recipEat*, esta solución pretende mejorar la organización en la cocina, fomentar una alimentación variada y reducir el desperdicio de alimentos.

Para lograrlo, se establecen dos líneas principales de objetivos: por un lado, los funcionales, que responden a las necesidades que debe cubrir la aplicación desde el punto de vista del usuario; y por otro, los técnicos, centrados en los desafíos y decisiones tecnológicas que hacen posible su desarrollo e implementación.

Objetivos funcionales

- **Autenticación de usuarios:** Permitir a los usuarios registrarse con un nombre de usuario, correo y contraseña, así como iniciar sesión mediante sus credenciales. Este mecanismo es fundamental para asociar datos personales o personalizados a cada usuario, adaptar el contenido mostrado según su perfil y acceder a funcionalidades específicas como la gestión de recetas propias o el seguimiento de actividad dentro de la aplicación.

- **Gestión de recetas:** Ofrecer la posibilidad de explorar recetas en la pantalla de inicio, buscar recetas por ingredientes o por título, visualizar los detalles de cada receta, marcarlas como favoritas y añadirlas al historial de cocina.

- **Historial de cocina:** Registrar las recetas cocinadas por el usuario y permitir visualizar las recetas preparadas en los últimos 7 o 30 días, facilitando el acceso a las más recientes.

- **Búsqueda de ingredientes:** Incluir una funcionalidad de búsqueda de ingredientes con autocompletado, permitiendo seleccionar varios ingredientes y obtener recetas que los contengan.

- **Creación y gestión de recetas personalizadas:** Permitir a los usuarios crear sus propias recetas incluyendo imagen, tiempo de preparación, raciones, ingredientes, pasos y ocasión, así como editarlas o eliminarlas posteriormente.

- **Modo offline:** Permitir ante una pérdida de conexión el acceso a recetas propias, recetas favoritas, quince recetas predeterminadas y las últimas quince recetas visualizadas por el usuario (que haya entrado a ver los detalles de la receta), almacenando los datos localmente y sincronizándolos al recuperar la conexión.

- **Gestión personal:** Permitir a los usuarios modificar su imagen, nombre de usuario y visualizar su correo, así como acceder a sus recetas favoritas y al historial de recetas cocinadas.

- **Visualización de pasos para cocinar a la par:** Mostrar los pasos de cada receta de forma secuencial y clara, permitiendo navegar entre ellos y, al llegar al último paso, marcar la receta como cocinada para mejorar la experiencia de cocinado.

- **Plan semanal:** Generar un plan semanal de comidas asignando recetas a desayuno, almuerzo y cena para cada día, actualizándolo automáticamente cada lunes mediante un *Worker*, asegurando variedad y evitando repeticiones.

- **Personalización del contenido en pantalla de inicio:** Adaptar dinámicamente las recetas mostradas en la pantalla principal según los gustos del usuario, analizando sus recetas favoritas y aplicando filtros como tipo de plato, tiempo o preferencias alimentarias (vegana, vegetariana, sin gluten) en función de su frecuencia.

- **Interacción intuitiva:** Diseñar una interfaz sencilla y fluida que facilite la navegación entre las distintas pantallas de la aplicación.

Objetivos técnicos

- **Desarrollo nativo para Android con *Kotlin* y *Jetpack Compose*:** Utilizar *Kotlin* y *Jetpack Compose* para construir una interfaz moderna y eficiente.
- **Integración con *Firebase*:** Usar *Firebase* para la autenticación de usuarios y como *backend* para almacenar información relevante del usuario, recetas y funcionalidades clave como favoritos, historial o plan semanal.
- **Consumo de API *Spoonacular*:** Obtener recetas y sus detalles a través de la API de *Spoonacular*, almacenarlas en *Firebase* y reducir la dependencia directa de la API para mejorar la autonomía de la aplicación.
- **Optimización del rendimiento:** Garantizar tiempos de respuesta rápidos, especialmente en la carga de recetas y búsquedas de ingredientes.
- **Modo *offline* con *Room*:** Implementar almacenamiento local con *Room*, permitiendo visualizar contenido sin conexión y sincronizar los datos al detectar conectividad.
- **Filtrado inteligente de ingredientes con *Gemini*:** Utilizar la API de *Gemini* para filtrar los ingredientes de las listas de la compra, eliminando elementos que no son realmente ingredientes y asegurando precisión.
- **Diseño de arquitectura modular y mantenible:** Estructurar el proyecto en capas bien diferenciadas como *data.model*, *ui.screens*, *ui.viewmodel* y *utils*, facilitando la escalabilidad y mantenibilidad del código.
- **Gestión del estado con *ViewModel* y *Compose*:** Aplicar un enfoque reactivo usando *ViewModel*, *StateFlow* y *remember*, asegurando una interfaz coherente y actualizada ante los cambios de estado.
- **Interfaz visual atractiva y funcional:** Desarrollar una interfaz visualmente atractiva y adaptativa mediante *Jetpack Compose*, que se ajuste a distintos tamaños y resoluciones de pantalla, teniendo en cuenta así la fragmentación del ecosistema Android.

3. Conceptos teóricos

Este capítulo proporciona una descripción general de los conceptos teóricos clave que son relevantes para el desarrollo y comprensión de la aplicación. Estos conceptos abarcan áreas como el desarrollo de aplicaciones móviles, la gestión de datos, la integración de APIs y la arquitectura de las aplicaciones nativas.

3.1. Desarrollo de Aplicaciones Móviles

El desarrollo de aplicaciones móviles consiste en la creación de *software* que se ejecuta en dispositivos portátiles como teléfonos inteligentes y tabletas. Las plataformas predominantes en este ámbito son Android e iOS, cada una con su propio ecosistema de herramientas, lenguajes de programación y marcos de desarrollo.

Entre los aspectos técnicos más relevantes se encuentran:

- **Lenguajes de programación:** *Kotlin* [27] es un lenguaje moderno que ofrece características como seguridad contra nulidades, sintaxis concisa y compatibilidad total con *Java*, lo que lo convierte en una opción popular para el desarrollo Android.

  Junto a *Kotlin*, también se utiliza *Java* [28], el lenguaje tradicional para Android, ampliamente soportado y con una gran base de código existente. Además, existen alternativas multiplataforma como *Flutter* [18], que utiliza el lenguaje *Dart* [21], y permite desarrollar aplicaciones nativas tanto para Android como para iOS desde una única base de código.

- ***Frameworks* de interfaz de usuario:** Las interfaces modernas se diseñan frecuentemente de manera declarativa, lo que mejora la legibilidad y el mantenimiento del código. Ejemplos de este enfoque son *Jetpack Compose* [13], que permite construir *UIs* dinámicas basadas en estados, y *Flutter* [18], que facilita el desarrollo multiplataforma con una arquitectura declarativa similar.
- **Conectividad con servicios externos:** Muchas aplicaciones móviles dependen de servicios *backend* para funcionalidades como autenticación, almacenamiento en la nube y sincronización de datos. Soluciones como *Firebase* ofrecen una infraestructura escalable y en tiempo real que cubre estas necesidades.

3.2. APIs

Una API [24] (*Application Programming Interface*) es un conjunto de reglas que permite la comunicación entre distintas piezas de *software*. Las APIs web permiten que las aplicaciones móviles accedan a servicios remotos para obtener o enviar datos.

- **APIs REST:** *Representational State Transfer* (REST) [31] es un estilo arquitectónico que se basa en el uso de métodos HTTP como *GET*, *POST*, *PUT* y *DELETE* para manipular recursos. Las APIs REST son ampliamente utilizadas por su simplicidad y compatibilidad con servicios web modernos.
- **Autenticación y claves API:** Para acceder a muchos servicios, es necesario autenticar las peticiones mediante *tokens* o claves API, lo que garantiza la seguridad y el control del acceso a los recursos.

3.3. Arquitectura de *Software*: MVVM

El patrón de arquitectura MVVM (*Model-View-ViewModel*) [23] se ha consolidado como un patrón de diseño eficaz en el desarrollo de aplicaciones móviles. Su objetivo es separar responsabilidades y facilitar la escalabilidad y mantenibilidad del código.

- ***Model*:** Representa la capa de datos y lógica de negocio.
- ***View*:** Corresponde a la interfaz de usuario, encargada de mostrar la información al usuario.

- ***ViewModel*:** Actúa como intermediario, transformando y gestionando los datos antes de presentarlos en la vista.

Este patrón mejora la organización del código y permite un desarrollo más limpio y desacoplado.

3.4. Programación Asíncrona con corrutinas

La programación asíncrona es esencial en aplicaciones que interactúan con servicios externos o ejecutan operaciones de larga duración. *Kotlin* introduce las corrutinas [17], una herramienta que permite manejar tareas asincrónicas de manera secuencial y no bloqueante.

Las corrutinas permiten realizar operaciones como llamadas HTTP, acceso a bases de datos o sincronización de datos en segundo plano sin interferir con la experiencia del usuario. Gracias a su integración con bibliotecas como *Retrofit* y *Room*, el uso de corrutinas se ha vuelto estándar en el desarrollo moderno.

3.5. Gestión del Ciclo de Vida

Las aplicaciones móviles deben responder correctamente a cambios en el estado del sistema, como rotaciones de pantalla, cambios de conectividad o navegación entre pantallas. Para ello, las plataformas móviles ofrecen herramientas que permiten observar y reaccionar a eventos del ciclo de vida de forma segura.

Componentes como *ViewModel* y *LiveData*, disponibles en la arquitectura *Android Jetpack*, permiten conservar y gestionar el estado de la *UI* de manera eficiente, evitando pérdidas de datos y comportamientos inesperados [10].

3.6. Persistencia de Datos

La persistencia de datos en una aplicación implica la capacidad de conservar información relevante del usuario o del sistema entre sesiones, ya sea mediante almacenamiento local o remoto. En este proyecto se emplea una estrategia híbrida: por un lado, el almacenamiento local permite que la aplicación funcione sin conexión a Internet, manteniendo accesibles datos

esenciales como favoritos, historial o información temporal. Para ello, Android proporciona la librería *Room* [14], que facilita la gestión de bases de datos *SQLite* mediante el uso de anotaciones y consultas seguras.

Por otro lado, también se utiliza persistencia remota a través de *Firebase*, lo que permite sincronizar datos en la nube, ofrecer respaldo y compartir información entre dispositivos o sesiones del mismo usuario. Esta combinación mejora la experiencia de usuario al equilibrar rendimiento, disponibilidad y sincronización.

3.7. Sincronización en Segundo Plano

Ciertas tareas deben ejecutarse de forma periódica o en momentos concretos, incluso cuando la aplicación no se encuentra en uso. Para ello, se puede recurrir a herramientas como *WorkManager* [3], una librería de Android que facilita la programación de trabajos en segundo plano, permitiendo establecer condiciones como la disponibilidad de conexión, el nivel de batería o una hora determinada para su ejecución.

WorkManager es útil para sincronizar datos, actualizar información periódicamente o ejecutar procesos tras reinicios del dispositivo.

4. Técnicas y herramientas

Este capítulo presenta las técnicas, herramientas y bibliotecas empleadas durante el desarrollo del proyecto. Se detallan las decisiones técnicas tomadas, junto con una breve comparativa de alternativas consideradas, justificando las elecciones realizadas en cada caso.

4.1. Técnicas

SCRUM

Scrum [5] es una metodología ágil ampliamente utilizada en el desarrollo de software, especialmente en contextos donde los requisitos pueden cambiar rápidamente y se necesita una entrega incremental de valor. Está diseñada para fomentar la colaboración entre los miembros del equipo, mejorar la transparencia del proceso y facilitar la adaptación constante a nuevas necesidades.

El marco de trabajo de Scrum se organiza en iteraciones llamadas *sprints*, que suelen durar entre una y cuatro semanas. Durante cada *sprint* se planifican tareas específicas, se llevan a cabo reuniones diarias de seguimiento (*daily stand-ups*), y se realiza una revisión al final del ciclo (*sprint review*) junto con una retrospectiva para identificar mejoras.

En este proyecto, se adoptaron varias prácticas de Scrum como la planificación de sprints quincenales y la revisión de las tareas completadas al final de cada ciclo. Esta organización permitió mantener un enfoque constante en los objetivos más importantes, mejorar la productividad y asegurar una evolución continua del proyecto.

Kanban

Kanban [4] es una técnica de gestión visual del trabajo que permite controlar el flujo de tareas mediante un tablero dividido en columnas que representan los distintos estados del proceso (por ejemplo, Pendiente, En progreso, Completado). Su objetivo es optimizar la eficiencia, reducir el tiempo de entrega y evitar la sobrecarga de trabajo.

Para complementar la planificación mediante Scrum, se adoptó el uso de un tablero Kanban en *GitHub Projects* . Las tareas se distribuyeron en las columnas *Backlog*, *In Progress* y *Done*, lo que permitió obtener una visión clara y actualizada del estado del proyecto. Para crear las tareas, se generaron *issues* en el repositorio, donde se detallaron el título y una descripción. Las tareas comenzaron en la columna *Backlog* y, a medida que se trabajaba en ellas, se movían a *In Progress*. Una vez completadas, se trasladaban a *Done*, proporcionando una representación visual del avance. El tablero utilizado durante el proyecto se puede ver en la figura 4.1.

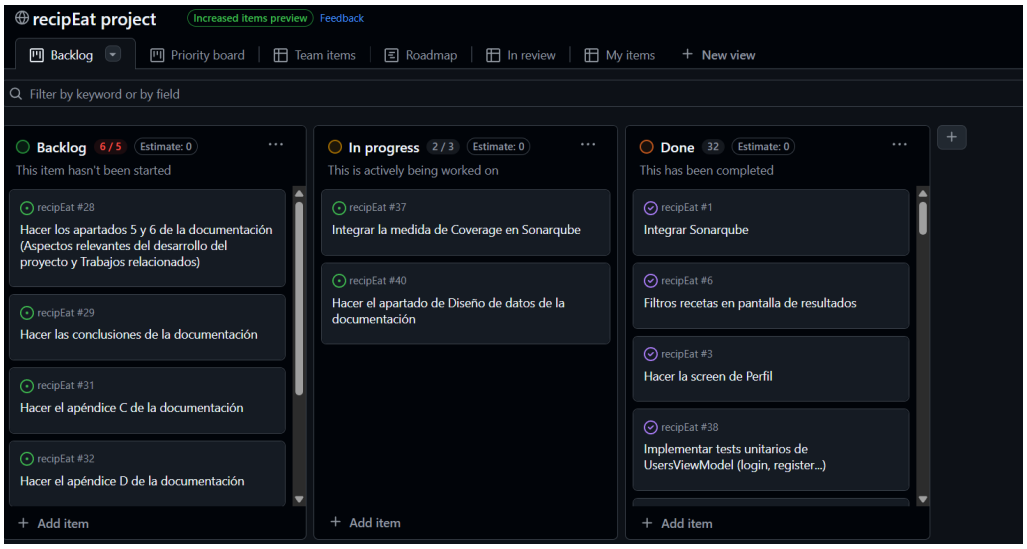

Figura 4.1: Tablero Kanban *GitHub Projects*

4.2. Herramientas

Android Studio

El entorno de desarrollo oficial para Android, *Android Studio* [1], fue utilizado como base para todo el desarrollo de la aplicación. Este entorno

ofrece una integración completa con *Kotlin*, *Jetpack Compose*, emuladores y herramientas de análisis, lo que facilita un flujo de trabajo eficiente y optimizado para el desarrollo de aplicaciones Android.

Además, para el desarrollo de la aplicación, se optó por Android como plataforma debido a la familiaridad con la misma y el ecosistema robusto que ofrece. Dado que el proyecto se centra en la creación de una aplicación móvil nativa, Android se presenta como una opción natural, ya que es ampliamente utilizado y proporciona un gran número de herramientas y recursos.

Aunque *Flutter*[18] es una alternativa popular para el desarrollo multiplataforma, se descartó su uso por las siguientes razones:

- Curva de aprendizaje: La curva de aprendizaje de *Flutter* es mayor en comparación con el desarrollo nativo para Android, especialmente al trabajar con tecnologías específicas del sistema operativo, como la autenticación o la integración con *Firebase*.
- Compatibilidad y optimización: Al trabajar de manera nativa, se obtiene un mayor control sobre las optimizaciones y el rendimiento de la aplicación, lo cual es especialmente importante cuando se manejan datos de manera eficiente o se requiere una integración profunda con servicios como *Firebase* o APIs externas.
- Aprovechamiento de capacidades específicas del sistema operativo: A diferencia de las soluciones multiplataforma, donde ciertas funcionalidades deben adaptarse o dependen de capas de abstracción adicionales, el desarrollo nativo permite explotar al máximo las capacidades propias del sistema operativo, como componentes de interfaz de usuario optimizados, acceso directo a sensores del dispositivo y una gestión más precisa del ciclo de vida de la aplicación. Esto se traduce en una experiencia de usuario más fluida, tiempos de respuesta más rápidos y una mayor estabilidad general.

Así, la elección de *Android Studio* y la plataforma Android permitió un desarrollo más enfocado y optimizado, con un control total sobre las características específicas del sistema operativo.

A continuación, se muestra una imagen del entorno de desarrollo *Android Studio*, utilizada durante el desarrollo de la aplicación (Figura 4.2).

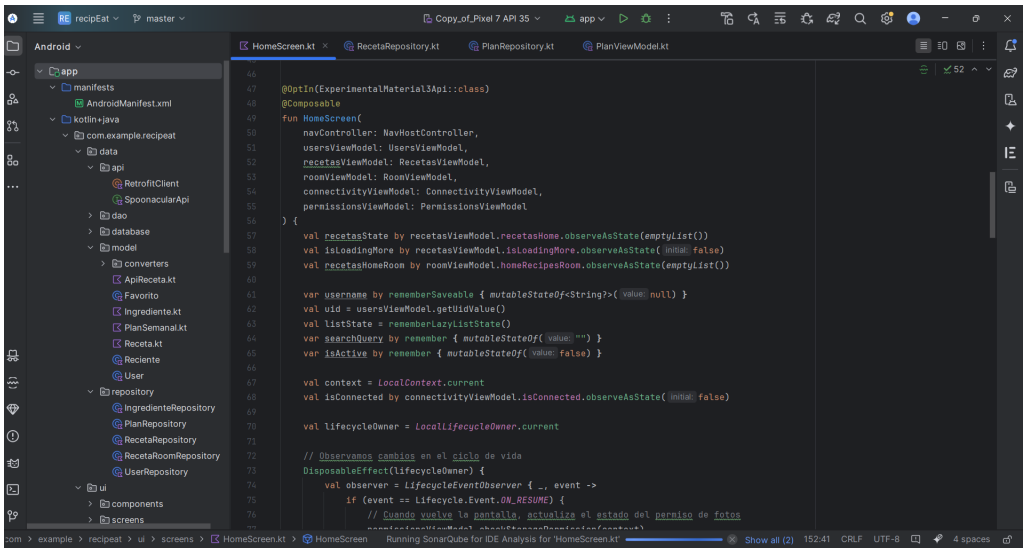

Figura 4.2: Entorno de desarrollo *Android Studio*

GitHub

Github [19] se usó como plataforma para alojar el repositorio y se empleó el sistema de *issues* para organizar tareas y seguimiento del desarrollo.

SonarQube

Se integró *SonarQube* [34] en el proceso de desarrollo para realizar análisis estáticos del código. Esta herramienta permitió identificar problemas relacionados con la calidad, mantenibilidad, duplicación de código, seguridad y estándares de buenas prácticas, fomentando una mejora continua del código fuente. En la Figura 4.3 se muestra el resumen general del análisis realizado por *SonarQube* sobre la rama principal del proyecto, donde se visualizan métricas clave, como la fiabilidad, seguridad, duplicidad y mantenibilidad.

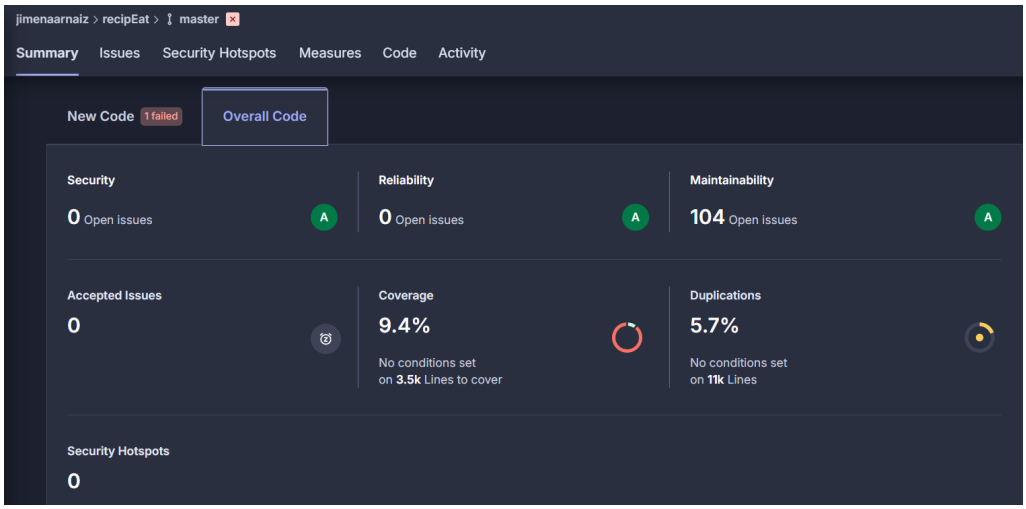

Figura 4.3: Resumen general del análisis de calidad del código

Los resultados reflejan que la mayoría de las métricas presentan una calificación de nivel A, lo que indica un buen estado general de la calidad del código en esos aspectos. Sin embargo, la métrica de *coverage* (cobertura de pruebas) no alcanza el 10 %, a pesar de contar con varios tests implementados (ver figura 4.4) que cubren parte de la lógica de los repositorios y *ViewModels*. Esto se debe a que *SonarQube* analiza también otros archivos, como las pantallas o componentes de *Jetpack Compose* del paquete UI (ver figura 4.5), que no cuentan con pruebas automatizadas.

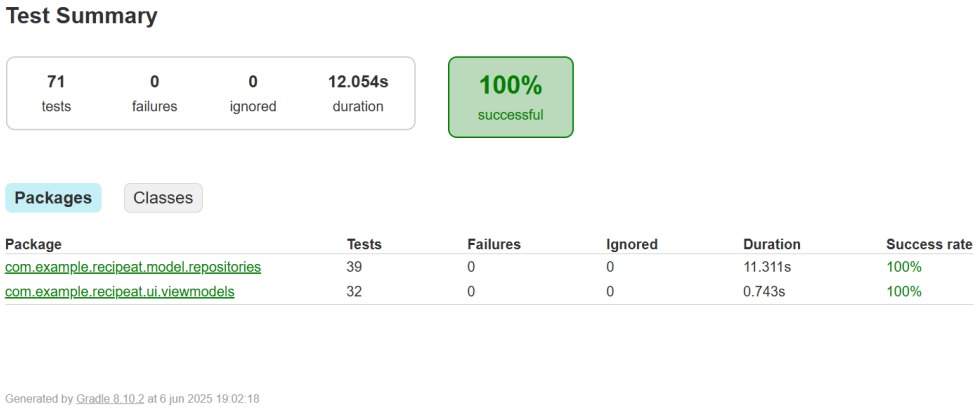

Figura 4.4: Resumen *Report Tests*

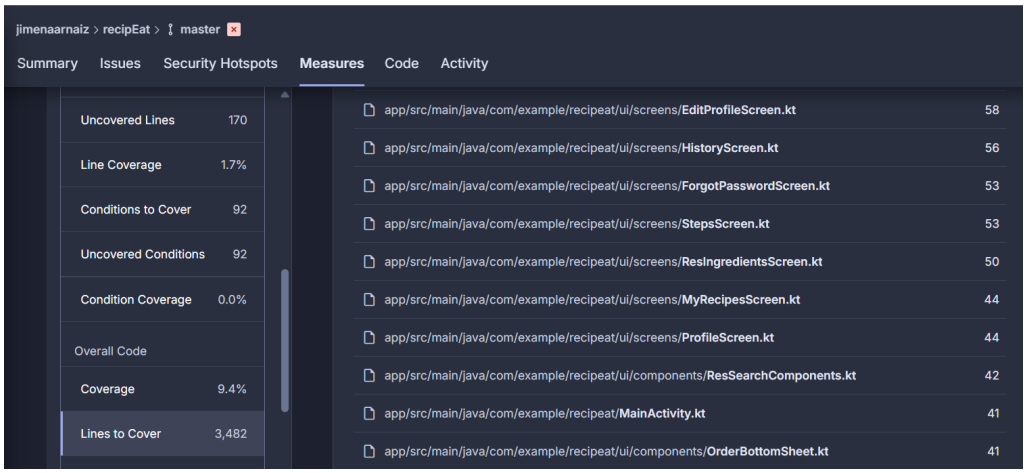

Figura 4.5: Métrica de cobertura

Las pruebas de interfaces en *Jetpack Compose* suelen ser más complejas de implementar como tests unitarios debido a la naturaleza declarativa y visual de estos componentes, que requieren herramientas específicas para pruebas instrumentadas o de interfaz de usuario (UI *testing*), como *Espresso* o *Compose Testing Framework*.

Por otro lado, la métrica de duplicaciones aparece en color naranja porque *SonarQube* solo la marca en verde cuando el porcentaje de código duplicado es menor o igual al 3 %. A pesar de los esfuerzos por reducir esta duplicación, no se ha logrado bajar ese umbral debido a la naturaleza repetitiva de algunas funcionalidades que, por su diseño, presentan fragmentos de código similares.

4.3. Bibliotecas y frameworks

Jetpack Compose

Jetpack Compose [13] fue elegido como el *framework* para la interfaz de usuario de la aplicación, en lugar de usar XML tradicional. Este moderno marco de trabajo permite un diseño declarativo, más legible y mantenible, y se integra de forma nativa con *Kotlin*, lo que optimiza la productividad del desarrollo. Algunas de las razones por las que se eligió *Jetpack Compose* frente a XML son las siguientes:

- Código más conciso: *Jetpack Compose* permite escribir menos código, lo que hace que la base del proyecto sea más limpia y fácil de mantener. A diferencia de XML, que requiere una mayor cantidad de código para definir la interfaz, *Compose* simplifica este proceso.
- Mejor rendimiento frente a XML tradicional: *Jetpack Compose* mejora el rendimiento al ser más eficiente en el manejo de los componentes visuales y las interacciones en tiempo real, lo que es más difícil de lograr con XML.
- Declarativo y Reactivo: *Compose* facilita la creación de interfaces de usuario dinámicas mediante un enfoque declarativo, lo que mejora la legibilidad y la facilidad de mantenimiento del código. Esto evita los problemas comunes en XML, como la complejidad de la actualización de vistas y el manejo de cambios de estado.
- Mejor integración con *Kotlin*: Al estar completamente diseñado para *Kotlin*, *Compose* aprovecha las características de este lenguaje, como la *null-safety* y las extensiones de funciones, simplificando el desarrollo. Esto no es tan directo en XML, que requiere configuraciones y adaptaciones adicionales.
- Facilidad de mantenimiento y escalabilidad: Al ser un *framework* más moderno, *Jetpack Compose* tiene una arquitectura más flexible que facilita la extensión y modificación de las interfaces sin los problemas comunes de escalabilidad que ocurren al trabajar con XML, donde los cambios pueden volverse engorrosos a medida que la aplicación crece.

Room

Room [14] es una librería de persistencia de datos proporcionada por Google que facilita el almacenamiento local en aplicaciones Android. Se utiliza para gestionar bases de datos SQLite de manera más eficiente y con una interfaz de acceso simplificada. En este proyecto, *Room* se empleó para gestionar el modo *offline*, permitiendo almacenar de forma local las recetas favoritas y las recetas creadas por el usuario. Esto garantiza que el usuario pueda acceder a sus datos incluso sin conexión a Internet, mejorando la experiencia y la fiabilidad de la aplicación.

Kotlin Coroutines

Kotlin Coroutines [25] es una biblioteca de *Kotlin* que permite escribir código asíncrono de manera secuencial y más legible. Son utilizadas para

operaciones asíncronas como llamadas a la API y acceso a bases de datos, mejorando la eficiencia sin bloquear el hilo principal.

Coil

Coil [6] es una biblioteca de carga de imágenes para Android basada en *Kotlin*, diseñada para ser rápida y eficiente. *Coil* aprovecha las características de *Kotlin* y está completamente integrada con las APIs modernas de Android, como las corrutinas, para ofrecer una experiencia de carga asíncrona sin bloquear el hilo principal.

Navigation Compose

Navigation Compose [2] es una biblioteca de *Jetpack* diseñada específicamente para gestionar la navegación entre pantallas dentro de aplicaciones desarrolladas con *Jetpack Compose*. Proporciona una forma declarativa de definir rutas, pasar argumentos entre pantallas y controlar el flujo de la aplicación desde una única fuente.

En este proyecto, se utilizó para implementar la navegación entre las diferentes pantallas de forma fluida, estructurada y totalmente integrada con el paradigma de desarrollo basado en *Compose*. Gracias a esta biblioteca, se logró una gestión del *back stack* eficiente y una arquitectura más limpia y mantenible.

WorkManager

WorkManager [3] es una biblioteca de *Android Jetpack* diseñada para gestionar tareas en segundo plano de manera eficiente y confiable, incluso si la aplicación se cierra o el dispositivo se reinicia. Está especialmente recomendada para tareas que deben garantizarse en algún momento, como sincronización de datos o actualizaciones periódicas. En el contexto de esta aplicación, se configuró para actualizar automáticamente el plan semanal cada lunes, garantizando que los usuarios cuenten con una nueva propuesta de recetas adaptadas a sus preferencias sin necesidad de intervención manual.

Retrofit

Retrofit [36] es una biblioteca cliente de tipo REST para Android y Java desarrollada por Square. Permite realizar solicitudes HTTP de forma estructurada, segura y sencilla, facilitando la conversión de respuestas JSON

en objetos de *Kotlin* o Java mediante anotaciones y adaptadores. Su diseño modular permite integrarse fácilmente con bibliotecas como *Gson*, *OkHttp* y *Coroutines*, lo que la convierte en una opción muy utilizada para el consumo de APIs en aplicaciones Android.

En esta aplicación, *Retrofit* se utilizó para realizar llamadas a la API de *Spoonacular* [35], permitiendo obtener recetas, ingredientes y otra información relevante de forma eficiente y segura. Se complementa con:

Gson Converter

Gson [20] es una librería que facilita la conversión de datos entre objetos *Kotlin* y su representación en formato JSON, permitiendo que *Retrofit* transforme las respuestas de la API en objetos *Kotlin* de manera sencilla y eficiente.

MockK

MockK [32] es una biblioteca de *mocking* para *Kotlin* pensada específicamente para pruebas unitarias. Se caracteriza por ofrecer una sintaxis intuitiva y una integración fluida con *coroutines*, objetos suspendidos y clases finales.

En este proyecto, *MockK* fue utilizada para probar los repositorios encargados de la lógica de acceso a datos, simulando las dependencias como *Firebase Firestore* o fuentes locales. Esto permitió verificar el comportamiento esperado de las clases bajo diferentes condiciones sin depender de servicios externos, mejorando así la fiabilidad y cobertura de las pruebas.

4.4. APIs y servicios externos

Firebase

Firebase [22] se utilizó como plataforma *backend* en la nube para gestionar diversas funcionalidades esenciales de la aplicación. A través de su módulo de *Authentication*, se implementó un sistema seguro de autenticación de usuarios, permitiendo el acceso personalizado a los datos de cada perfil. Además, se usó *Cloud Firestore* como base de datos principal, donde se almacenan los ingredientes, las recetas obtenidas de la API *Spoonacular* y las recetas que crean los propios usuarios, entre otros datos importantes.

A continuación, se muestran capturas del uso de *Firebase Authentication* y *Firestore* en el proyecto, donde se gestionan tanto los usuarios como los datos de las recetas (Figuras 4.6 y 4.7).

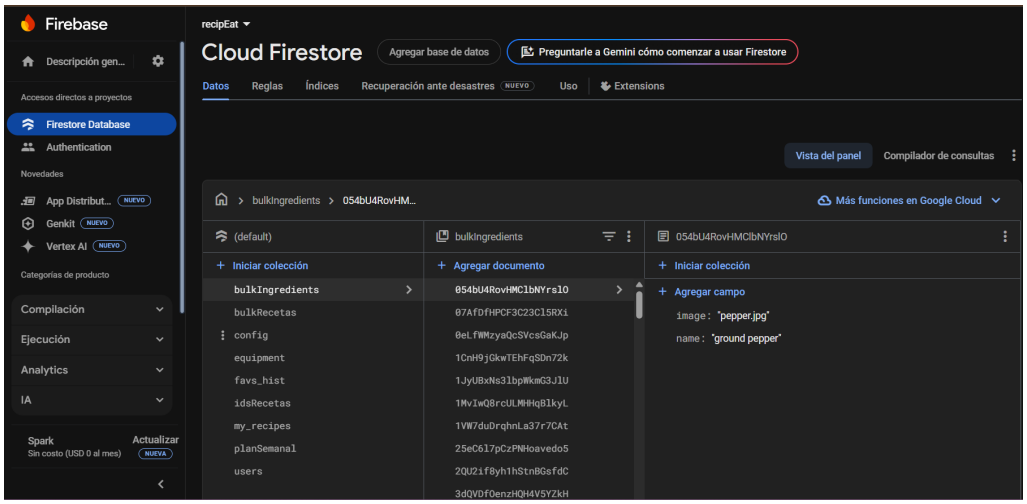

Figura 4.6: Firebase Firestore

Figura 4.7: Firebase Authentication

API *Spoonacular*

Durante el desarrollo del proyecto, se investigaron diversas fuentes de datos para obtener recetas, pero se encontraron varias limitaciones con los datasets disponibles. Muchos de estos conjuntos de datos no se ajustaban a las necesidades específicas del proyecto, como la disponibilidad de ingredientes, imágenes de ingredientes y recetas, o no eran lo suficientemente completos para cubrir todos los requisitos de la aplicación.

Por este motivo, se decidió utilizar la API *Spoonacular*, una solución robusta y bien documentada que proporciona acceso a una amplia base de datos de recetas con información detallada sobre ingredientes, pasos de preparación, imágenes y más. Esta API permitió obtener recetas de manera eficiente y adaptada a las necesidades de la aplicación, garantizando que los datos estuvieran completos.

Además, la integración de la API con el *backend* de *Firebase* facilitó el almacenamiento y la gestión de los datos dentro de la aplicación, mejorando la experiencia del usuario y optimizando la accesibilidad a la información.

Google Gemini (*Generative AI*)

Google Gemini [8] es un modelo de inteligencia artificial generativa desarrollado por Google DeepMind, diseñado para comprender y generar texto de forma contextual y precisa. Se incluyó para filtrar de forma inteligente los ingredientes de la lista de la compra generada al crear el plan semanal, ya que algunas recetas de *Spoonacular* incluyen instrucciones o partes de los pasos como si fuesen ingredientes.

4.5. Herramientas para escribir la memoria

Durante el desarrollo de esta memoria se emplearon diversas herramientas digitales que facilitaron la redacción, edición colaborativa y creación de diagramas técnicos. A continuación, se describen tres herramientas clave utilizadas en este trabajo junto con algunas de sus ventajas más destacadas.

Overleaf

Overleaf [29] es una plataforma colaborativa en línea que permite escribir, editar y compilar documentos en LaTeX directamente desde el navegador. Esta herramienta ha ganado popularidad entre estudiantes, investigadores y profesionales debido a su enfoque en la simplicidad y la colaboración. Uno de sus principales puntos fuertes es la posibilidad de trabajar con otras personas en tiempo real, como si se tratara de un documento compartido, manteniendo al mismo tiempo la potencia del lenguaje LaTeX. Además, ofrece control de versiones, historial de cambios y posibilidad de sincronizar con GitHub o Dropbox, lo que facilita la integración con flujos de trabajo ya establecidos.

Ventajas:

- **Colaboración en tiempo real:** varios usuarios pueden trabajar simultáneamente, visualizando los cambios al instante.
- **No requiere instalación local:** todo se gestiona en la nube, evitando problemas de configuración o compatibilidad.
- **Historial de versiones:** permite ver y restaurar versiones anteriores del documento fácilmente.
- **Amplia colección de plantillas:** desde artículos académicos hasta currículos y tesis, lo que acelera el inicio del trabajo.
- **Compilación automática:** actualiza el PDF automáticamente al guardar, facilitando la visualización del resultado final.

draw.io

draw.io [15] (actualmente *diagrams.net*) es una herramienta en línea gratuita para la creación de diagramas. Su interfaz visual permite diseñar diagramas de flujo, mapas mentales, organigramas, diagramas UML y otros modelos gráficos con facilidad. Ofrece integración con servicios en la nube como Google Drive o OneDrive, lo que permite guardar y compartir diagramas directamente. Además, se puede utilizar de manera local sin conexión, mediante su versión de escritorio. En este proyecto se utilizó para elaborar diagramas de flujo que describen procesos y la navegación entre pantallas de la aplicación.

Ventajas:

- **Facilidad de uso:** su sistema de arrastrar y soltar permite crear diagramas de forma rápida sin conocimientos técnicos previos.
- **Versatilidad de formatos:** permite exportar los diagramas en PNG, JPEG, PDF, SVG, entre otros, lo que facilita su inclusión en documentos.
- **Sincronización con la nube:** guarda automáticamente en servicios como Google Drive o GitHub, lo que favorece la colaboración y el respaldo.
- **Amplia biblioteca de formas:** ofrece símbolos adaptados a distintas disciplinas, desde redes hasta arquitectura de software.

- **Uso sin conexión:** se puede descargar como aplicación de escritorio, útil en entornos sin conexión constante.

PlantUML

PlantUML [30] es una herramienta de código abierto diseñada para crear diagramas UML y otros tipos de diagramas a partir de descripciones en texto. Utiliza una sintaxis específica que permite generar diagramas de clases, de secuencia, de actividades, de casos de uso, entre otros. Esta herramienta es especialmente útil en proyectos de desarrollo de software porque permite integrar los diagramas directamente en el flujo de desarrollo, ya que el código fuente del diagrama puede mantenerse junto al resto del proyecto.

Ventajas:

- **Basada en texto:** ideal para mantener bajo control de versiones, facilitando la trazabilidad y revisión de los cambios.
- **Automatización:** al estar basado en *scripts*, se pueden regenerar diagramas automáticamente si cambian los modelos o procesos.
- **Integración con LaTeX:** se puede compilar desde archivos externos o mediante herramientas como *pdfLaTeX*, facilitando su inclusión en documentos académicos.
- **Multiplataforma:** al ser de código abierto, funciona en Windows, macOS y Linux, y puede ejecutarse tanto en línea como localmente.
- **Flexibilidad:** soporta múltiples tipos de diagramas, no solo UML, sino también organigramas o diagramas de Gantt.

dbdiagram.io

`dbdiagram.io` [33] es una herramienta en línea que permite diseñar y visualizar esquemas de bases de datos de forma rápida y sencilla mediante una sintaxis textual llamada DBML (*Database Markup Language*). Esta plataforma está orientada principalmente al modelado de bases de datos relacionales, aunque también resulta útil como recurso para representar estructuras de datos NoSQL adaptadas a formato relacional, como es el caso de bases de datos en *Firestore*.

Ventajas:

- **Sintaxis declarativa clara:** mediante DBML se pueden definir tablas, campos, tipos de datos y relaciones de forma estructurada y legible.
- **Visualización automática:** la herramienta genera automáticamente un diagrama entidad-relación (ER) a partir del código, facilitando la comprensión del modelo de datos.
- **Colaboración en línea:** permite compartir diagramas mediante enlaces, facilitando el trabajo en equipo y la revisión entre miembros de un proyecto.
- **Exportación:** se pueden exportar los diagramas como imágenes (PNG, SVG) o incluso generar *scripts* SQL para distintas bases de datos como PostgreSQL, MySQL o SQLite.

5. Aspectos relevantes del desarrollo del proyecto

5.1. Inicio del Proyecto

El proyecto nació de la necesidad cotidiana de decidir qué comer, una tarea que a menudo se convierte en un dilema por la falta de ideas o la escasez de tiempo. La intención era crear una herramienta que simplificara el qué preparar para comer, brindando opciones basadas en ingredientes disponibles.

Con el objetivo de ofrecer una experiencia de usuario óptima y aprovechar al máximo las capacidades del sistema operativo, se decidió desarrollar la aplicación de forma nativa para Android. Este enfoque permitiría una integración más profunda con las funcionalidades del dispositivo, garantizando un rendimiento eficiente y una interfaz adaptada a las necesidades del usuario.

5.2. Desarrollo del *frontend*

En el desarrollo del *frontend* de la aplicación, se decidió seguir las buenas prácticas recomendadas por la documentación oficial de Android, utilizando A*ndroid Studio* como entorno de desarrollo y *Jetpack Compose* para la creación de la interfaz de usuario. *Jetpack Compose* facilitó la construcción de una interfaz declarativa, moderna y flexible, permitiendo un desarrollo más rápido y con un código más limpio y fácil de mantener. Con *Compose*, se podía construir la interfaz directamente en *Kotlin*, evitando la tradicional

separación entre XML y código Java/*Kotlin*, lo que simplificaba el proceso de diseño y desarrollo.

Una de las primeras decisiones tomadas fue el uso de *Navigation Component* [2] de Android, que permite gestionar de manera eficiente la navegación entre las diferentes pantallas de la aplicación. *Navigation* no solo facilita la transición entre pantallas, sino que también organiza y centraliza el flujo de navegación de la aplicación, asegurando que todas las rutas y transiciones estén claramente definidas. Esta herramienta fomenta las buenas prácticas de desarrollo al garantizar que las pantallas estén correctamente estructuradas y al evitar problemas de duplicación de código en las transiciones entre actividades o fragmentos. Además, *Navigation* facilita la gestión del estado de la navegación, lo que resulta especialmente útil para manejar la acción de volver atrás sin perder el contexto de la pantalla actual. La Figura 5.1 muestra un fragmento del código del *Navigation Graph* utilizado para definir las rutas y la estructura de navegación de la aplicación.

```
NavigationGraph.kt
com.example.recipeat (androidTest)
com.example.recipeat (test)
model.repositories
IngredienteRepositoryTest
PlanRepositoryTest
RecetaRoomRepositoryTest
UsersRepositoryTest.kt
ui.viewmodels
FiltrosViewModelTest
IngredientesViewModelTest
UsersViewModelTest2
ExampleUnitTest
ava (generated)
es
es (generated)
dle Scripts
build.gradle.kts (Project: recipEat)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules for
gradle.properties (Project Properties)
gradle-wrapper.properties (Gradle Versi
bs.versions.toml (Version Catalog)
ocal.properties (SDK Location)

118            }
119            composable(BottomNavItem.Home.route) {
120                HomeScreen(navController, usersViewModel, recetasViewModel, roomViewModel, connectivityViewModel, permissions
121                onBottomBarVisibilityChanged(true)
122            }
123            composable(BottomNavItem.MyRecipes.route) {
124                MyRecipesScreen(navController, recetasViewModel, roomViewModel, usersViewModel, connectivityViewModel)   //
125                onBottomBarVisibilityChanged(true)
126            }
127            composable(BottomNavItem.Profile.route) {
128                ProfileScreen(navController, usersViewModel, connectivityViewModel) // Perfil
129                onBottomBarVisibilityChanged(true)
130            }
131            composable( route: "resultados/{query}") { backStackEntry ->
132                val query = backStackEntry.arguments?.getString( key: "query") ?: ""
133                ResNameScreen(nombreReceta = query, navController = navController, recetasViewModel, filtrosViewModel, usersV
134                onBottomBarVisibilityChanged(false)
135            }
136            composable( route: "detalles/{idReceta}/{deUser}") { backStackEntry ->
137                val idReceta = backStackEntry.arguments?.getString( key: "idReceta") ?: ""
138                val deUser = backStackEntry.arguments?.getString( key: "deUser")?.toBoolean() ?: false
139
140                // Verifica que se haya obtenido el valor correctamente
141                Log.d( tag: "DetailsScreen",  msg: "Valor de deUser: $deUser")
142
143                // Ahora pasamos el parámetro 'deUser' a la pantalla de detalles
144                DetailsScreen(idReceta = idReceta, navController = navController, recetasViewModel = recetasViewModel, esDeUs
app > src > main > java > com > example > recipeat > NavigationGraph.kt > NavigationGraph
65:5  CRLF  UTF-8  4 spac
```

Figura 5.1: Fragmento de código que gestiona la navegación en la *app*

Otro aspecto importante durante el desarrollo del *frontend* fue la compatibilidad con la rotación de pantalla. La implementación de la rotación sin perder el estado se hizo posible mediante el uso de *SavedStateHandle* junto con los *ViewModels*, lo cual almacenó temporalmente el estado de la pantalla y lo restauró automáticamente cuando la orientación cambiaba. Esto permitió mantener la coherencia y fluidez de la interfaz, sin tener que recargar los datos o interrumpir la experiencia.

Además, fue necesario diseñar composiciones diferentes para orientación vertical y horizontal en pantallas como *Login*, *Register* o *Forgot Password*.

Por ejemplo, en la pantalla de inicio de sesión (*LoginScreen*), se detecta la orientación usando *LocalConfiguration.current.orientation* y se decide, en función de ello, si se deben colocar los elementos en una columna (*portrait*) o en una fila (*landscape*). Esto permitió mantener una distribución limpia y cómoda, asegurando que la imagen y los formularios no se solapen ni se vean comprimidos en orientaciones amplias. En las Figuras 5.2 y 5.3 se muestra cómo varía el diseño de esta pantalla según la orientación del dispositivo.

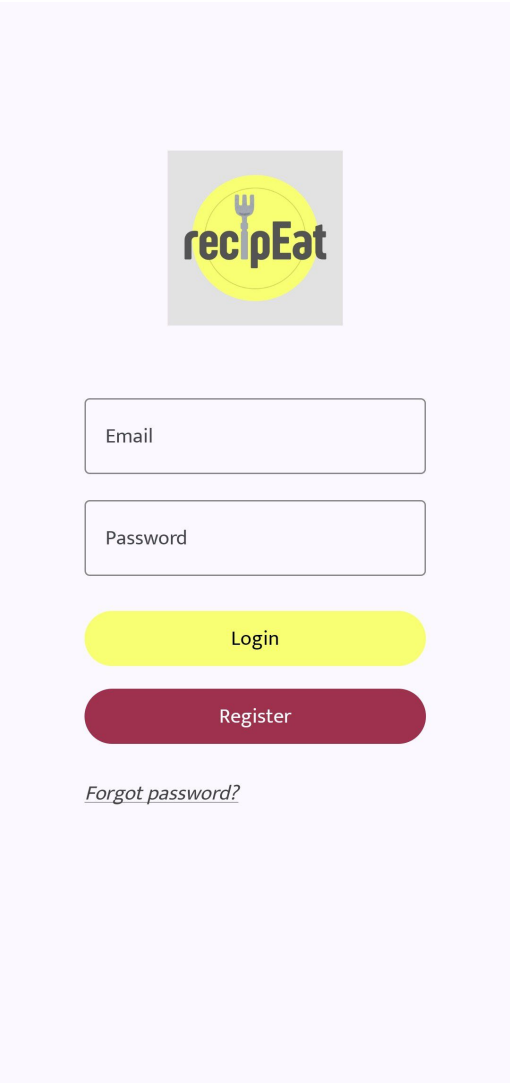

Figura 5.2: *UI Login* Vertical

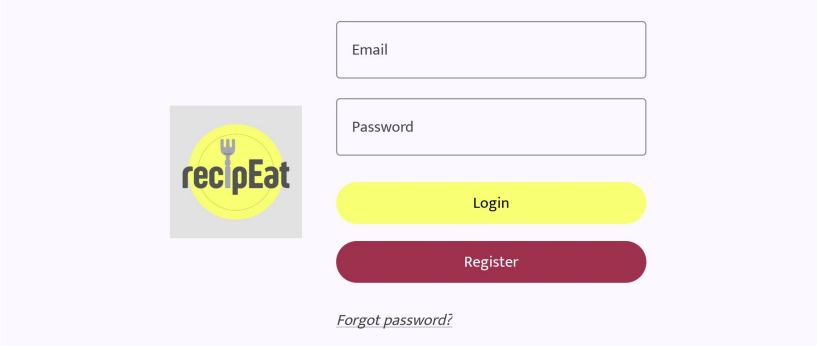

Figura 5.3: *UI Login* Horizontal

Por otro lado, también se pensó en optimizar la experiencia del usuario, no solo desde la parte funcional, sino también en cuanto a la fluidez de la interfaz. Las pantallas fueron cuidadosamente diseñadas para ser intuitivas y fáciles de navegar, con el mínimo de interacción necesaria para realizar tareas como agregar ingredientes a las recetas o consultar los detalles de una receta.

Un reto importante en el desarrollo fue implementar un sistema de *scrolling* eficiente para las listas de recetas, que ofreciera una navegación fluida sin comprometer el rendimiento del dispositivo. Para ello, se aplicaron dos estrategias complementarias: por un lado, se implementó paginación tanto en la pantalla principal (*Home*) como en la de recetas creadas por los usuarios, cargando los datos en bloques de 15 en 15. La primera carga trae los primeros 15 elementos y, si el usuario continúa desplazándose, se utiliza el documento más reciente como punto de referencia (*startAfter*) para obtener el siguiente bloque. Esta estrategia permite dar la sensación de *scroll* infinito sin derrochar los recursos del dispositivo (memoria, CPU, consumo de datos). Por otro lado, también se hizo uso de *LazyColumn* [9] en *Compose*, que renderiza únicamente los elementos visibles en pantalla, reduciendo el consumo de memoria y mejorando aún más el rendimiento al mostrar grandes cantidades de datos. Con ello se logró que las pantallas fueran dinámicas, ágiles y adaptables a todo tipo de dispositivos y tamaños de pantalla.

5.3. Desarrollo del *backend*

Durante el desarrollo del *backend*, hubo un cambio en la arquitectura y una reestructuración significativa del diseño inicial.

En un primer momento, la arquitectura planteada combinaba dos fuentes de datos: por un lado, la API externa de *Spoonacular*, y por otro, *Firebase* como *backend* propio. Esta decisión buscaba aprovechar al máximo la información que proporcionaba la API para alimentar funcionalidades como la búsqueda de recetas por nombre o por ingredientes, mostrar recetas aleatorias en el inicio y más. Paralelamente, se utilizaba *Firebase* para almacenar datos personalizados del usuario, como sus recetas creadas, favoritos o historial de cocina.

Sin embargo, durante la revisión del segundo *sprint*, se identificaron varios problemas importantes:

- El uso intensivo de la API provocaba el riesgo de alcanzar el límite diario de llamadas, afectando al funcionamiento de la *app*.
- La coexistencia de dos fuentes de datos generaba una lógica fragmentada y difícil de mantener, propensa a inconsistencias.
- La dependencia directa de la API en tiempo real aumentaba la fragilidad del sistema frente a caídas o cambios en el servicio.

Ante esta situación, se decidió rediseñar el *backend* con un enfoque más limpio y eficiente, delegando toda la gestión de datos a *Firebase* y utilizando la API únicamente para obtener los datos que sirvan de base para la base de datos en *Firebase*. Esta decisión tuvo consecuencias positivas a nivel de organización, rendimiento y simplicidad del sistema.

Como ya no se contaba con el *backend* de recetas e ingredientes directamente proporcionado por la API, fue necesario construir uno propio en *Firebase*. Para ello, se desarrolló una función para el almacenamiento de recetas provenientes de la API de *Spoonacular* en *Firebase Firestor*e. Esta función ejecutaba una petición para obtener recetas aleatorias y, por cada receta, realizaba una segunda llamada a la API para obtener las instrucciones detalladas en formato de pasos separados.

```
interface SpoonacularApi {

    // Obtener recetas random
    @GET("recipes/random")
    suspend fun obtenerRecetasRandom(
        @Query("number") number: Int = 90,
        @Query("apiKey") apiKey: String = API_SPOONACULAR_KEY
    ): RandomRecipesResponse

    // Obtener instrucciones en diferentes strings de una receta por ID
    @GET("recipes/{id}/analyzedInstructions")
    suspend fun obtenerInstruccionesReceta(
        @Path("id") recetaId: Int,
        @Query("stepBreakdown") stepBreakdown: Boolean = true,
        @Query("apiKey") apiKey: String = API_SPOONACULAR_KEY
    ) : List<Map<String, Any>>
```

Figura 5.4: Llamadas a la API Spoonacular

Cada receta obtenida se transformaba utilizando la función `mapApiRecetaToReceta`, que convertía el modelo de datos de la API (`ApiReceta`) al modelo de dominio utilizado por la aplicación (`Receta`). Esta transformación incluía un procesamiento adicional de las instrucciones (*analyzedInstructions*), de las cuales se extraían exclusivamente las descripciones textuales de los pasos, facilitando así su presentación en la interfaz de usuario.

```
// Función para mapear ApiReceta a Receta
fun mapApiRecetaToReceta(apiReceta: ApiReceta, uid: String, analyzedInstructions: List<Map<String, Any>>): Receta {

    // Extraer solo los valores de "step" en una lista de Strings
    val pasos = analyzedInstructions.flatMap { instruction ->
        val steps = instruction["steps"] as? List<Map<String, Any>> ?: emptyList()
        steps.mapNotNull { step ->
            step["step"] as? String // Extraemos solo la descripción del paso
        }
    }

    return Receta(
        id = apiReceta.id.toString(),
        title = apiReceta.title,
        image = apiReceta.image,
        servings = apiReceta.servings,
        ingredients = apiReceta.extendedIngredients,
        steps = pasos,
        time = apiReceta.readyInMinutes,
        dishTypes = apiReceta.dishTypes,
        userId = uid,  // ID del usuario que crea la receta
        glutenFree = apiReceta.glutenFree,
        vegan = apiReceta.vegan,
        vegetarian = apiReceta.vegetarian,
        date = System.currentTimeMillis(),
        unusedIngredients = emptyList(),
        missingIngredientCount = 0,
        unusedIngredientCount = 0
    )
}
```

Figura 5.5: Función para mapear el modelo *ApiReceta* a *Receta*

Una vez transformada, la receta se convertía en un *HashMap* que contenía sus propiedades. Esta estructura se almacenaba dentro de la colección recetas, bajo un subdocumento correspondiente al usuario autenticado en la colección recetas_aleatorias.

En cuanto a los ingredientes, a medida que se obtenían y almacenaban nuevas recetas, se implementó una función personalizada que recorría todas las recetas guardadas en *Firebase* con el objetivo de extraer ingredientes únicos. Esta función fue desarrollada específicamente para construir una colección centralizada en *Firestore* denominada *ingredientes*, que actuaría como base de datos propia. Dicha colección se utilizaba para alimentar funcionalidades esenciales de la aplicación, como la búsqueda por ingredientes disponibles.

Sin embargo, con el paso del tiempo se observó una limitación importante en este enfoque inicial: si bien se lograba construir una base amplia de ingredientes, muchos de ellos estaban vinculados a muy pocas recetas, ya que estas se obtenían de manera aleatoria. Esto generaba un repositorio de datos

desequilibrado, con una gran variedad de ingredientes poco representativos en términos de utilidad para el usuario.

Ante esta situación, se optó por redirigir la estrategia de almacenamiento: en lugar de partir de recetas aleatorias y extraer sus ingredientes, se generó una lista completa de todos los ingredientes presentes en la base de datos. A partir de esta lista, se realizó una selección manual de los ingredientes más relevantes, priorizando aquellos más comunes y útiles para los usuarios y, a partir de ella, se realizaron llamadas específicas a la API para obtener recetas relacionadas con esos ingredientes concretos.

Este cambio permitió construir un *backend* más coherente y orientado a la experiencia del usuario, asegurando que cada ingrediente almacenado estuviese respaldado por múltiples recetas relevantes. Como resultado, se mejoró la eficiencia de las búsquedas por ingrediente y se redujo el ruido en la base de datos.

En fases más avanzadas del desarrollo, se incorporó la funcionalidad de generar un plan semanal personalizado para cada usuario. Para ello, fue necesario actualizar las recetas almacenadas en *Firebase* añadiendo el campo `aisle` (pasillo) a cada ingrediente. Este campo resultó fundamental tanto para construir listas de la compra organizadas por secciones, como para implementar un sistema inteligente de selección de recetas. Durante la generación del plan semanal, el algoritmo se apoya en la información del *aisle* para asegurar la diversidad de ingredientes y evitar el uso excesivo de ciertos tipos de alimentos (por ejemplo, carne o pasta).

Generación inteligente del plan semanal

Una de las funcionalidades más destacadas de la aplicación es la generación automática de un plan semanal de comidas, que incluye desayuno, almuerzo y cena para cada día de la semana. Esta funcionalidad se desarrolló en fases avanzadas del proyecto, y requirió una lógica cuidadosa para garantizar que el plan ofreciera variedad y utilidad real para el usuario.

Para su implementación, se diseñó un algoritmo que, a partir de las recetas almacenadas en *Firebase*, construye un plan semanal evitando repeticiones recientes y promoviendo una alimentación variada. El proceso sigue los siguientes principios:

- **Evitar repeticiones:** Se consulta el plan de la semana anterior y se evita seleccionar recetas ya utilizadas, lo que promueve la variedad en las comidas semanales.

- **Clasificación por tipo de plato:** Las recetas se clasifican según si corresponden a desayuno, almuerzo o cena. Esta clasificación se basa, en primer lugar, en el campo *dishTypes* de cada receta. En caso de que este campo esté vacío, se analiza el *aisle* principal de sus ingredientes para inferir el tipo de comida. Por ejemplo, para los desayunos se consideran válidos los pasillos de panadería, lácteos o cereales; mientras que para los almuerzos y cenas se priorizan ingredientes como carnes, vegetales, arroces o alimentos étnicos.
- **Filtrado adicional para desayunos:** Además del tipo de plato o pasillo, las recetas candidatas para el desayuno deben tener un tiempo de preparación reducido (igual o inferior a 30 minutos), lo que refleja una característica común de este tipo de comida.
- **Diversificación de ingredientes:** Se analiza el pasillo (*aisle*) principal de cada receta (por ejemplo: lácteos, carnes, vegetales, etc.) y se limita la repetición de pasillos similares dentro de la misma semana para evitar planes redundantes.
- **Relajación progresiva de restricciones:** Debido a que no se cuenta con una gran base de datos que asegure que siempre se encuentren recetas, si tras aplicar todos los filtros no se logra generar un plan completo, el sistema flexibiliza progresivamente las restricciones para garantizar siempre un resultado válido.

La generación del plan semanal se realiza durante el proceso de inicio de sesión, cuando se detecta que no existe un plan previo para el usuario, ya sea porque es la primera vez que accede o porque ha cambiado de usuario. A partir de entonces, la generación del plan se automatiza mediante un ***`Worker`*** local configurado con la librería ***`WorkManager`***, que ejecuta la tarea cada lunes a las 00:00. Esto garantiza que el plan se actualice semanalmente incluso si la aplicación no está en primer plano o tras un reinicio del dispositivo.

Este *Worker* se encarga de verificar si hay un usuario autenticado antes de proceder, ya que el plan se genera en base al identificador del usuario activo. Si no hay una sesión iniciada en el momento de la ejecución (por ejemplo, si el usuario ha cerrado sesión), la generación del plan no se lleva a cabo. Por ello, al iniciar sesión, se comprueba si el usuario ya tiene un plan correspondiente a la semana actual; si no lo tiene, se lanza la generación de dicho plan.

Para organizar los planes generados, se utiliza un identificador basado en la fecha en formato `día-mes`, donde el día corresponde al lunes de la

semana actual. Además, la tarea periódica se configura para ejecutarse con un *delay* calculado hasta el próximo lunes a medianoche.

Una vez generado el plan, se guarda también una **lista de la compra**, obtenida a partir de los ingredientes de las recetas seleccionadas. Para ello, los ingredientes son procesados, agrupados y filtrados mediante la API de Gemini de Google, eliminando elementos que en realidad son instrucciones y no ingredientes reales.

Implementación del modo *offline*

La implementación del modo *offline* representó uno de los mayores desafíos del desarrollo, al implicar que los usuarios pudieran seguir accediendo a sus recetas favoritas, recientes y creadas, incluso sin conexión a Internet. Este objetivo conllevó varios retos técnicos: en primer lugar, lograr que la aplicación reaccionara de forma autónoma ante la pérdida de conectividad, adaptando su comportamiento sin requerir intervención del usuario; en segundo lugar, diseñar una lógica que distinguiera de forma clara entre los datos almacenados localmente y aquellos obtenidos desde el backend remoto, garantizando en todo momento una experiencia coherente; y por último, establecer un mecanismo de sincronización eficiente que permitiera actualizar los datos en la nube una vez restablecida la conexión, evitando conflictos o pérdida de información durante el proceso.

Para ello, se diseñó una arquitectura de almacenamiento local basada en *Room*, con un repositorio específico denominado `RecetaRoomRepository`, que actúa como intermediario entre la capa de presentación y la base de datos local.

Las recetas se almacenan localmente en la entidad `Receta`, que contiene toda la información relevante, incluyendo ingredientes, pasos, tipo de plato... Sin embargo, para representar acciones del usuario como marcar una receta como favorita o haberla consultado recientemente, se definieron las entidades `Favorito` y `Reciente`, respectivamente. Estas entidades no almacenan los datos completos de la receta, sino referencias al ID de la receta existente en `Receta`, junto con el `userId` y una marca temporal (*Timestamp*) que permite ordenar y controlar estas listas.

Se implementó una lógica de control para mantener un máximo de 15 recetas en la lista de recientes, eliminando automáticamente la entrada más antigua cuando se supera este umbral. Cada tipo de interacción se gestiona mediante su respectivo DAO, asegurando una separación clara

de responsabilidades y facilitando las operaciones de inserción, consulta y eliminación.

Para la detección de la conectividad, se utilizó la clase `NetworkConnectivityManager`, que emplea el `ConnectivityManager` de Android [11] para monitorizar cambios en el estado de red. Esta clase ofrece una interfaz reactiva, permitiendo a la aplicación adaptarse dinámicamente: accediendo al contenido local cuando no hay conexión y sincronizando los datos cuando esta se restablece.

Gracias a esto, la aplicación mantiene su funcionalidad esencial en todo momento, proporcionando una experiencia fluida y sin interrupciones para el usuario.

Para ilustrar la estructura y funcionamiento del almacenamiento *offline*, en las Figuras 5.6, 5.7 y 5.8, se presentan capturas del *App Inspector* de *Android Studio* [12], donde se visualiza el contenido de las tablas locales `Receta`, `Favorito` y `Reciente` en la base de datos *Room*. Esta herramienta permitió verificar que las recetas se almacenan correctamente y que las relaciones entre entidades se mantienen según lo previsto.

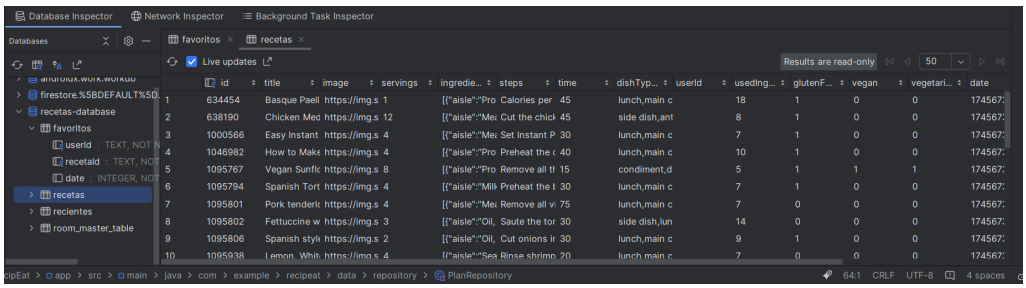

Figura 5.6: Tabla Recetas de *Room*

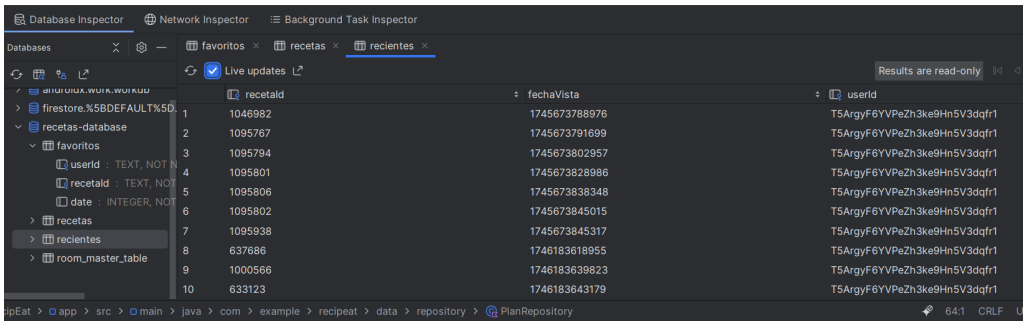

Figura 5.7: Tabla Recientes de *Room*

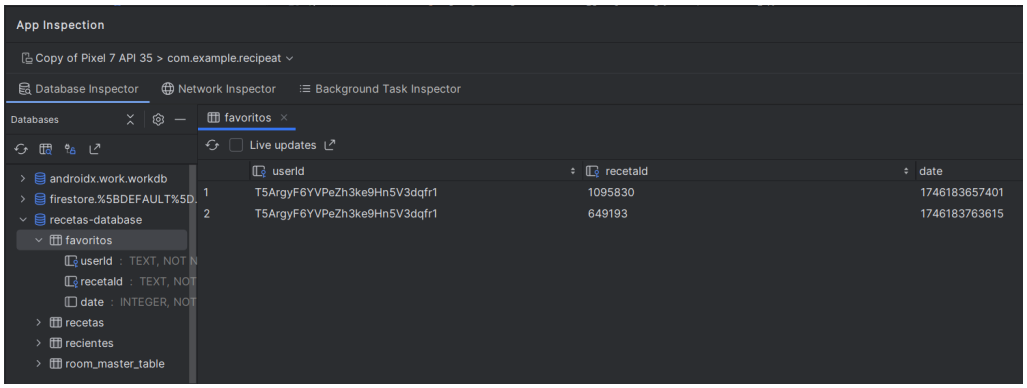

Figura 5.8: Tabla Favoritos de *Room*

A continuación, en las figuras 5.9 y 5.10, se muestran capturas de pantalla de la aplicación funcionando en modo *offline*.

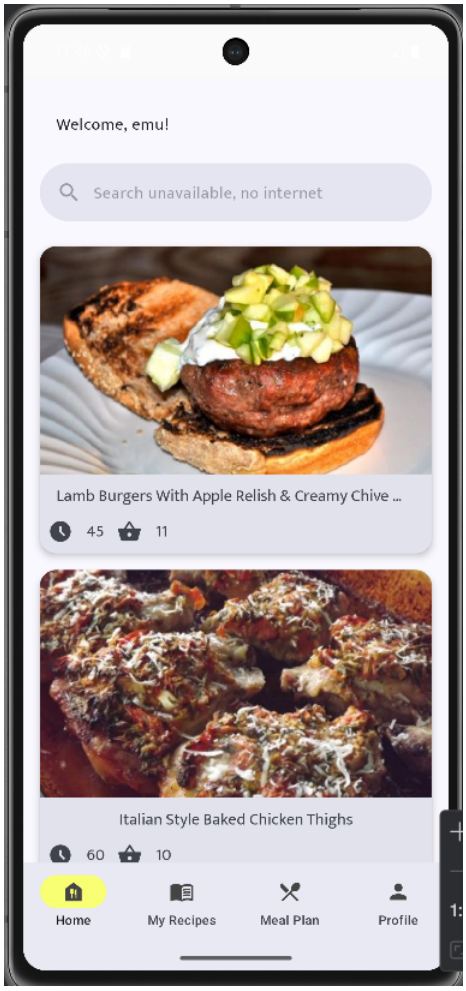

Figura 5.9: Pantalla Home en modo *offline*

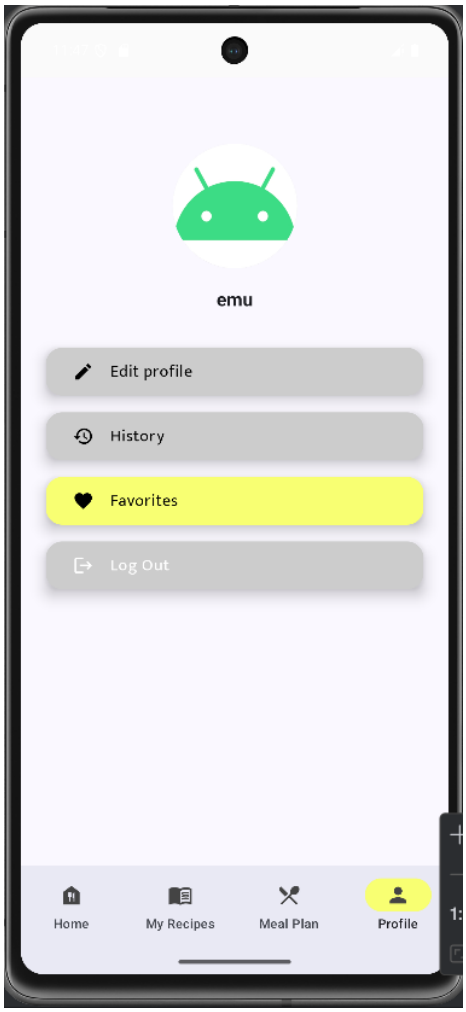

Figura 5.10: Pantalla de perfil en modo *offline*

5.4. Despliegue

Durante el desarrollo de la aplicación, se llevaron a cabo pruebas exhaustivas para validar su funcionamiento tanto en entornos simulados como reales. Inicialmente, se utilizó el emulador de *Android Studio*, que permitió simular múltiples configuraciones de dispositivo y versiones del sistema operativo Android. Esto facilitó la detección temprana de errores, así como la verificación de la correcta adaptación de la interfaz de usuario a distintos tamaños de pantalla y condiciones de rendimiento.

No obstante, con el objetivo de aproximarse a escenarios más realistas, también se realizaron pruebas en dispositivos físicos. Estas pruebas permitieron evaluar aspectos como la fluidez de navegación, los tiempos de respuesta, el comportamiento ante pérdida de conectividad, la persistencia de datos en modo sin conexión y la estabilidad general de la aplicación frente a eventos comunes, como cambios de orientación.

Aunque la aplicación no ha sido publicada en la Google Play Store, se encuentra disponible para su descarga a través de la sección de *releases* del repositorio recipEat en GitHub. Asimismo, para facilitar su evaluación sin necesidad de instalación local, se ha habilitado una versión accesible desde el navegador mediante la plataforma *Appetize.io* [1], disponible en el siguiente enlace: appetize recipEat. Ahora bien, esta plataforma presenta ciertas limitaciones asociadas a su uso gratuito. En concreto, cada sesión de uso está restringida a un máximo de tres minutos, tras los cuales la aplicación se cierra automáticamente, siendo necesario volver a iniciarla e iniciar sesión de nuevo para continuar con la evaluación. A pesar de estas restricciones, se ha considerado la mejor alternativa disponible para permitir una revisión rápida y sin necesidad de instalación local.

También es importante destacar que el rendimiento de la aplicación varía en función del entorno de ejecución. La versión más limitada corresponde a la plataforma *Appetize.io*, que, al ejecutarse en remoto dentro de un navegador, presenta ciertas restricciones de rendimiento. El emulador de *Android Studio* ofrece una experiencia más fluida, pero es en dispositivos físicos donde la aplicación muestra su funcionamiento óptimo, con tiempos de carga reducidos y navegación más ágil.

5.5. Testing

El proceso de *testing* es una etapa esencial en el desarrollo de *software*, ya que permite asegurar que los distintos componentes del sistema funcionen correctamente y de forma coherente ante posibles cambios futuros. En este proyecto, se han implementado pruebas unitarias con el objetivo de verificar la lógica interna de los repositorios y otras clases clave de la aplicación.

[1] Para acceder a la aplicación, puede utilizar el siguiente usuario de prueba: `demo@gmail.com` con contraseña `123456`

Pruebas Unitarias

Las pruebas unitarias fueron desarrolladas utilizando el *framework* `MockK` [32], ampliamente utilizado en proyectos *Kotlin* para la creación de objetos simulados (*mocks*). Estas pruebas se centraron principalmente en los repositorios, que encapsulan la lógica de acceso a datos, incluyendo la interacción con *Firebase Firestore*.

Durante el proceso de *testing*, fue necesario realizar modificaciones en el código de producción para facilitar las pruebas. Algunas de estas modificaciones incluyeron:

- **Inyección de dependencias:** Se pasó `Firebase Firestore` como parámetro al constructor de los repositorios, permitiendo así sustituirlo por una versión simulada durante las pruebas.
- **Uso de funciones suspendidas:** Para aprovechar el soporte de *coroutines* en las pruebas y garantizar un flujo controlado y asíncrono de las operaciones.
- ***Mock* del sistema de *logs* de *Android*:** Al ser funciones estáticas, su presencia generaba errores en el entorno de test, por lo que fue necesario *mockearlas* explícitamente para evitar fallos durante la ejecución de las pruebas.

Gracias a estas modificaciones, fue posible escribir pruebas unitarias eficaces que abarcan tanto escenarios positivos (como la recuperación exitosa de datos desde *Firebase*) como escenarios negativos, por ejemplo:

```
@Test
fun `login with valid credentials should return success`() = runTest {
    val validEmail = "test@example.com"
    val validPassword = "password123"

    val mockAuthResult = mockk<AuthResult>()
    val mockUser = mockk<FirebaseUser>()
    val mockDocumentSnapshot = mockk<DocumentSnapshot>()
    val mockUserRef = mockk<DocumentReference>()

    // Simular que el login con FirebaseAuth funciona y devuelve un usuario
    every { mockAuthResult.user } returns mockUser
    every { mockUser.uid } returns "mockUid"
    every { auth.signInWithEmailAndPassword(validEmail, validPassword) } returns Tasks.forResult(mockAuthResult)

    // Simular referencia al documento del usuario en Firestore
    every { db.collection( collectionPath: "users").document( documentPath: "mockUid") } returns mockUserRef
    every { mockUserRef.get() } returns Tasks.forResult(mockDocumentSnapshot)

    // Simular que el documento existe en Firestore
    every { mockDocumentSnapshot.exists() } returns true

    // Act
    val result = mockUserRepository.login(validEmail, validPassword)

    // Assert
    assertEquals( expected: "success", result)
    verify { auth.signInWithEmailAndPassword(validEmail, validPassword) }
    verify { db.collection( collectionPath: "users").document( documentPath: "mockUid") }
}
```

Figura 5.11: Test de *Login* con credenciales válidas

```
31  class UserRepositoryTest {
104     }
105
106     @Test
107     fun `login with empty password should return error message`() = runTest {
108
109         val email = "test@example.com"
110         val emptyPassword = ""
111         val exception = FirebaseAuthInvalidCredentialsException("ERROR_INVALID_CREDENTIAL", "Password is empty")
112
113         every {
114             auth.signInWithEmailAndPassword(email, emptyPassword)
115         } returns Tasks.forException(exception)
116
117         // Act
118         val result = mockUserRepository.login(email, emptyPassword)
119
120         // Assert
121         assertEquals( expected: "Invalid credentials. Please, check your email or password.", result)
122         verify { auth.signInWithEmailAndPassword(email, emptyPassword) }
123     }
124
Tests passed: 13 of 13 tests – 5 sec 876 ms
```

Figura 5.12: Test de *Login* con contraseña vacía

Ejecución de pruebas

La ejecución de las pruebas unitarias se realiza principalmente desde el entorno de desarrollo *Android Studio*, aprovechando la integración con el sistema de compilación *Gradle*.

Beneficios

Este enfoque de pruebas aporta múltiples beneficios al desarrollo:

- **Verificación del comportamiento esperado:** Se asegura que las funciones devuelvan los datos correctos ante entradas conocidas.
- **Confianza durante el *refactor*:** Las pruebas sirven como red de seguridad al modificar el código.
- **Mejor comprensión del código:** Actúan también como documentación viva del comportamiento esperado de cada clase.
- **Detección temprana de errores:** Al validar los componentes de manera aislada, los errores se detectan en etapas tempranas, reduciendo el coste de corrección.

En definitiva, el proceso de *testing* mediante pruebas unitarias con `MockK` ha sido clave para garantizar la estabilidad, calidad y mantenibilidad del código en este proyecto.

6. Trabajos relacionados

En este capítulo se presenta un análisis de trabajos previos y aplicaciones existentes relacionadas con el seguimiento de recetas y la gestión de alimentos en el hogar, destacando aplicaciones como Cookpad, Ekilu y Yummly. Estas plataformas comparten un enfoque similar de gestión de recetas, con funcionalidades como la recomendación de platos basados en ingredientes disponibles y la integración de características para facilitar la vida de los usuarios en la cocina.

6.1. Cookpad

Cookpad [7] es una aplicación popular a nivel mundial que permite a los usuarios compartir recetas de cocina. Su enfoque principal es facilitar la creación y el intercambio de recetas entre los usuarios, brindando un espacio para la comunidad y el aprendizaje. A diferencia de otras plataformas, Cookpad pone un fuerte énfasis en la interacción social, permitiendo a los usuarios compartir fotos, comentarios y sugerencias sobre las recetas publicadas. Además, la aplicación ofrece recomendaciones basadas en las preferencias de los usuarios y permite personalizar las búsquedas según los ingredientes disponibles, lo que facilita la creación de platos con lo que se tiene en casa.

6.2. Ekilu

Ekilu [16] es una aplicación que se enfoca en la recomendación de recetas a partir de los ingredientes que el usuario ya tiene disponibles en su hogar. Esta app ofrece una interfaz intuitiva donde los usuarios pueden seleccionar

los ingredientes que tienen en su despensa, y la aplicación les sugiere recetas que pueden realizar con esos productos. Además, Ekilu incorpora opciones para crear listas de compras y gestionar la disponibilidad de los ingredientes. Uno de sus puntos fuertes es la integración con opciones de recetas saludables y una variedad de filtros para ajustarse a diferentes preferencias dietéticas, como recetas vegetarianas, sin gluten, etc.

6.3. Yummly

Yummly [37] es otra plataforma destacada en el ámbito de las recetas de cocina. Similar a Ekilu, Yummly permite a los usuarios buscar recetas a partir de ingredientes específicos. Su algoritmo avanzado sugiere recetas no solo basadas en los ingredientes, sino también teniendo en cuenta las preferencias dietéticas, el tiempo de preparación y las restricciones alimentarias. Yummly también se destaca por su funcionalidad de "listas de compras", en la que los usuarios pueden agregar los ingredientes necesarios para una receta y organizarlos en función de lo que ya tienen disponible en su hogar. Además, Yummly ofrece recomendaciones personalizadas mediante su sistema de aprendizaje automático, adaptándose a los gustos y hábitos culinarios de los usuarios.

Estas aplicaciones representan algunas de las mejores soluciones disponibles actualmente en el mercado para gestionar recetas de cocina y optimizar el uso de los ingredientes disponibles.

7. Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

A continuación se detallan las conclusiones de este proyecto:

Este Trabajo de Fin de Grado supone el cierre de una etapa universitaria en la que no solo he adquirido conocimientos técnicos sobre desarrollo de *software*, sino también una visión más completa y crítica sobre el ciclo de vida de un proyecto, desde la idea inicial hasta su implementación y entrega final.

Aunque el desarrollo de aplicaciones móviles no forma parte del contenido principal del grado, durante mi estancia Erasmus tuve la oportunidad de cursar una asignatura centrada en esta área. Aquella experiencia despertó mi interés por el desarrollo móvil y fue una de las razones por las que decidí enfocar este proyecto en la creación de una aplicación Android.

El desarrollo de la *app* desde cero ha supuesto un reto importante a nivel técnico y personal. A lo largo del proyecto, he tenido que aprender y aplicar tecnologías como *Jetpack Compose*, *Firebase*, *Room* y la integración con APIs REST. También ha sido clave seguir buenas prácticas de arquitectura y diseño para asegurar que la aplicación sea funcional, clara y fácil de mantener.

Uno de los mayores retos ha sido organizar el proyecto de forma eficiente, tomar decisiones tecnológicas adecuadas y asegurar que la aplicación se comporta correctamente en situaciones reales. Todo esto ha implicado inves-

tigar, probar diferentes soluciones y adaptarme constantemente a nuevos aprendizajes.

La elaboración de la documentación técnica ha requerido bastante tiempo y dedicación, pero ha sido fundamental para dejar constancia del trabajo realizado, justificar las decisiones tomadas y facilitar que otras personas puedan entender el proyecto o incluso continuarlo en el futuro.

En resumen, el objetivo principal del trabajo se ha cumplido de forma satisfactoria, incluso superando las expectativas iniciales al añadir funcionalidades adicionales. Este proyecto no solo me ha permitido aplicar lo aprendido durante la carrera, sino también explorar nuevas áreas que han ampliado mis conocimientos y reforzado mi interés por el desarrollo móvil.

7.2. Líneas de trabajo futuras

A continuación, se presentan posibles líneas de trabajo futuras que permitirán seguir evolucionando *recipEat* y mejorar la experiencia del usuario:

- **Interacción social:** Permitir a los usuarios compartir recetas creadas o favoritas con otros, fomentando la creación de una comunidad activa. Esto podría incluir la opción de seguir a otros usuarios, comentar recetas y colaborar en la creación de nuevas recetas. Esta característica ayudaría a generar un entorno de intercambio de conocimientos sobre cocina y aprovechamiento de alimentos.

- **Registro manual de alimentos y fechas de caducidad:** Incorporar la capacidad de registrar manualmente los alimentos disponibles en el hogar, junto con sus fechas de caducidad, permitiría al usuario llevar un control más preciso y personalizado de lo que tiene en su despensa. Además, se podrían generar alertas de caducidad, ayudando a reducir el desperdicio alimentario y optimizando la planificación de las comidas.

- **Mejorar la personalización del plan semanal:** Ampliar las opciones de personalización para el planificador semanal, permitiendo a los usuarios seleccionar el tipo de dieta (como vegana, sin gluten, etc.), sino también parámetros adicionales como el tiempo de cocinado estimado, la preferencia de ingredientes y la repetición de ingredientes durante la semana. Esto permitiría una experiencia aún más personalizada y adaptada a las necesidades de cada usuario.

- **Expansión multiplataforma:** Aunque el enfoque inicial ha sido Android, se podría considerar la expansión a iOS mediante *Kotlin Multiplatform* [26]. Esto permitiría llegar a un público más amplio, garantizando que los usuarios de ambas plataformas disfruten de una experiencia similar.

- **Estadísticas y análisis de hábitos alimenticios:** Añadir una sección de estadísticas en el perfil del usuario, que permita visualizar hábitos de consumo, estadísticas sobre los ingredientes más usados, número de recetas diferentes cocinadas y la reducción de desperdicio alimentario a lo largo del tiempo.

- **Integración con servicios de compra *online*:** Facilitar la compra de ingredientes directamente desde la aplicación, integrándose con servicios de compra *online* o supermercados locales. Esto simplificaría el proceso de obtención de los ingredientes necesarios para las recetas y permitiría realizar compras de manera más eficiente.

- **Mejora de la experiencia de usuario con IA:** Implementar inteligencia artificial para sugerir recetas basadas en las preferencias de los usuarios, las recetas que han cocinado previamente, o incluso los ingredientes que están por caducar. Esto permitiría un nivel de personalización aún mayor, proporcionando recomendaciones dinámicas que mejoren la experiencia culinaria.

Bibliografía

[1] Android Developers. Android studio. https://developer.android.com/studio, 2024. [Internet; accessed 02-May-2025].

[2] Android Developers. Navigation in jetpack compose. https://developer.android.com/jetpack/compose/navigation, 2024. [Internet; accessed 02-May-2025].

[3] Android Developers. Workmanager. https://developer.android.com/topic/libraries/architecture/workmanager, 2024. [Internet; accessed 02-May-2025].

[4] Asana. ¿Qué es el método Kanban? – Guía de gestión del trabajo. https://asana.com/es/resources/what-is-kanban, 2024. [Internet; accessed 23-Apr-2025].

[5] Asana. ¿Qué es Scrum? – Guía de gestión ágil de proyectos. https://asana.com/es/resources/what-is-scrum, 2024. [Internet; accessed 23-Apr-2025].

[6] Coil Contributors. Coil - kotlin image loader. https://coil-kt.github.io/coil/, 2024. [Internet; accessed 02-May-2025].

[7] Cookpad. Cookpad: The best recipe sharing app. https://cookpad.com, 2023. [Internet; accessed 02-May-2025].

[8] Google DeepMind. Gemini api. https://ai.google.dev/gemini, 2025. [Internet; accessed 25-Apr-2025].

[9] Android Developers. Lazy layouts in compose. https://developer.android.com/jetpack/compose/lists, 2024. [Internet; accessed 03-May-2025].

[10] Android Developers. Viewmodel overview. https://developer.android.com/topic/libraries/architecture/viewmodel, 2024. [Internet; accessed 03-May-2025].

[11] Android Developers. Connectivitymanager. https://developer.android.com/reference/android/net/ConnectivityManager, 2025. [Internet; accessed 04-May-2025].

[12] Android Developers. Inspect app databases with the app inspector. https://developer.android.com/studio/inspect/db-inspector, 2025. [Internet; accessed 04-May-2025].

[13] Android Developers. Jetpack compose. https://developer.android.com/compose, 2025. [Internet; accessed 25-Apr-2025].

[14] Android Developers. Room persistence library - android jetpack. https://developer.android.com/jetpack/androidx/releases/room?hl=es-419, 2025. [Internet; accessed 25-Apr-2025].

[15] diagrams.net. Draw.io - diagramas online gratuitos. https://www.diagrams.net, 2025. [Internet; accessed 02-May-2025].

[16] Ekilu. Ekilu: Tu receta con los ingredientes que tienes. https://www.ekilu.com, 2023. [Internet; accessed 02-May-2025].

[17] Roman Elizarov. Structured concurrency in kotlin. https://elizarov.medium.com, 2025. [Internet; accessed 04-May-2025].

[18] Flutter Team. Flutter. https://flutter.dev, 2024. [Internet; accessed 02-May-2025].

[19] GitHub, Inc. Github. https://github.com, 2024. [Internet; accessed 02-May-2025].

[20] Google. Gson. https://github.com/google/gson, 2024. [Internet; accessed 02-May-2025].

[21] Google. Dart programming language. https://dart.dev, 2025. [Internet; accessed 7-Jun-2025].

[22] Google. Firebase. https://firebase.google.com/?hl=es-419, 2025. [Internet; accessed 25-Apr-2025].

[23] Google. Guide to app architecture. https://developer.android.com/topic/architecture, 2025. [Internet; accessed 04-May-2025].

[24] IBM. What is an api (application programming interface)? https://www.ibm.com/think/topics/api, 2025. [Internet; accessed 04-May-2025].

[25] JetBrains. Kotlin coroutines. https://kotlinlang.org/docs/coroutines-overview.html, 2024. [Internet; accessed 02-May-2025].

[26] JetBrains. Kotlin Multiplatform documentation. https://kotlinlang.org/docs/multiplatform.html, 2025. [Internet; accessed 23-Apr-2025].

[27] JetBrains. Kotlin programming language. https://kotlinlang.org/, 2025. [Internet; accessed 25-Apr-2025].

[28] Oracle. Java programming language. https://www.oracle.com/java/, 2025. [Internet; accessed 7-Jun-2025].

[29] Overleaf. Overleaf, online latex editor. https://www.overleaf.com, 2025. [Internet; accessed 02-May-2025].

[30] PlantUML. Plantuml - diagrams as code. https://plantuml.com, 2025. [Internet; accessed 02-May-2025].

[31] RestfulAPI.net. Restful api design. https://restfulapi.net/, 2025. [Internet; accessed 25-Apr-2025].

[32] Ivan Shkuro. Mockk: Kotlin mocking library. https://mockk.io/, 2023. [Internet; accessed 04-May-2025].

[33] Holistics Software. dbdiagram.io — database relationship diagrams design tool. https://dbdiagram.io, 2025. [Internet; accessed 31-May-2025].

[34] SonarSource. Sonarqube. https://www.sonarsource.com/products/sonarqube/, 2024. [Internet; accessed 02-May-2025].

[35] Spoonacular. Spoonacular food api. https://spoonacular.com/food-api, 2025. [Internet; accessed 25-Apr-2025].

[36] Square, Inc. Retrofit. https://square.github.io/retrofit/, 2024. [Internet; accessed 02-May-2025].

[37] Yummly. Yummly: The recipe discovery and meal planning app. https://www.yummly.com, 2023. [Internet; accessed 02-May-2025].