UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

GRADO DE INGENIERÍA EN INFORMÁTICA

NutriPlan

Aplicación para la Gestión de Recetas

Autor
Aissa Rouk El Masoudi

Directores
CARLOS RODRIGUEZ DOMINGUEZ

Facultad de Educación, Tecnología y Economía de Ceuta

Granada, 15 de Junio de 2025

Dedicado a
...

Resumen

OMS cita que en 2022 el 43 % de los adultos en el mundo tenía sobrepeso, y el 16 % padecía obesidad, lo que equivale a unos 890 millones de adultos obesos[17]. Desde 1990, el número de personas con obesidad se han más que duplicado entre los adultos y se han cuadruplicado entre los adolescentes[3].
Esta situación afecta a personas de todas las edades, regiones y tiene graves consecuencias para la salud, economía y los sistemas sanitarios, hace cada vez más urgente fomentar hábitos alimentarios saludables y estrategias que promuevan una alimentación consciente, eficiente y sostenible.
De este contexto surge NutriPlan, una aplicación de Meal Planning y gestión de recetas cuyo objetivo es fomentar un estilo de vida saludable, económico y sostenible, reduciendo el desperdicio de alimentos. Esta idea surgió en mi experiencia como estudiante Erasmus, dónde descubrí que al realizar la lista de la compra manualmente y verificar cuáles ingredientes tenía, se ahorraba más dinero en la compra, desperdiciaba menos comida y lo más importante, comía sano.

Abstract

This project will proceed with the design and development of a recipe application...

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

Siempre quise crear algo relacionado con mis dos vocaciones, ayudar a las personas y la programación, cada vez que tengo un momento libre intento utilizarlo para pensar cómo hacer el mundo un sitio mejor, qué ideas de aplicaciones podrían aportar algo a las necesidades de la sociedad, esto me llevó a crear varias aplicaciones desde crear una aplicación de escritura reflexiva para mejora personal, hasta la creación de un software administrador de tareas.

Esta idea de proyecto surgió cuando estaba en un programa de movilidad estudiantil, después de varios meses realizando la compra e intentando reducir su gasto sin que baje la calidad de mi dieta, noté al principio que cuando realizaba una lista de la compra, gastaba mucho menos de lo que debía, porque no compraba cosas innecesarias o duplicadas. Al cabo de otro tiempo, descubrí que si realizaba mi plan dietético de manera específica antes de realizar la compra semanal, este gasto era mucho menor porque en la lista añadía las cantidades de cada ingrediente que necesitaba, esto no solo redujo mis compras innecesarias, sino que me motivaba a comprar las cantidades exactas necesarias de cada producto o ingrediente, reduciendo el coste de mi compra semanal.

Además de este beneficio económico, percibí que la realización de esta práctica me impulsaba a respetar las cantidades planificadas en mi dieta, haciendo que, esta no solo se volviera más saludable, sino también más variada. La Librería Nacional de Medicina Estadounidense realizó un estudio dónde se concluyó que *"la planificación dietética está asociada a una dieta más saludable y una menor prevalencia de obesidad"*[16], y que *"la práctica de esta podría ser potencialmente relevante para la prevención de la obesidad"*[16].

Una publicación de la Comisión Europea afirma que los hogares generan más de la mitad del desperdicio alimentario existente y que una de sus causas es la insuficiente planificación de las compras y de las comidas, lo que no solo perjudica la economía familiar, sino que también tiene un impacto ambiental significativo.

Todo esto me llevó a crear NutriPlan, una aplicación que satisface todas estas necesidades. Un software que ayuda a los usuarios mediante la creación de una lista de la compra con las recetas ya registradas por el usuario, lleva un seguimiento de su despensa recordando que ingredientes aún tiene disponibles, y calculando cuándo cada ingrediente que se acaba.

1.2. Objetivos

El objetivo de este proyecto es la creación de una aplicación móvil que facilite la planificación dietética, haciéndola más sencilla y eficiente promoviendo todos los beneficios discutidos anteriormente en la motivación; lleve un seguimiento de la despensa del usuario y modifique esta cada vez que se realice la compra, planificación de la dieta semanal mediante la adición o eliminación de ingredientes en la despensa.

El resultado será una aplicación con las siguientes funcionalidades:

- Gestión de recetas
  - Adición de una nueva receta mediante la introducción del nombre, enlace, tiempo de preparación y la cantidad de porciones de la receta; la adición de los ingredientes correspondientes a la receta.
  - Eliminación o edición de una receta ya existente.
- Gestión de ingredientes
  - Adición de nuevos ingredientes a una receta con sus cantidades.
  - Posibilidad de buscar (mediante un motor de búsqueda) ingredientes nuevos para añadirlos a una receta que se esté creando.
- Gestión de dieta semanal
  - El usuario podrá asignar una o varias recetas para cada comida del día (desayuno, comida, cena), esta asignación es modificable.
- Gestión de la lista de la compra
  - El programa será el encargado de la creación de una lista de la compra. Esto se realizará sumando las cantidades de los ingredientes similares de todas las recetas y restando a esta suma las cantidades que haya en la despensa.

1.3. Estructura de la memoria

En este trabajo la estructura de la memoria sigue un orden lógico que facilita la lectura y evita repeticiones. En la presente *Introducción* se describe la motivación personal para el proyecto, se formulan los objetivos y se resumen las funcionalidades principales de NutriPlan. El capítulo 2 contiene una revisión del estado del arte: se exponen los fundamentos del *meal planning*, se analizan tendencias actuales, se justifican las decisiones de diseño de interfaz y se comparan las tecnologías y bibliotecas empleadas.

En el capítulo dedicado al diseño y descripción del sistema se detalla la arquitectura de la aplicación, incluyendo las decisiones sobre la lógica de la aplicación, la organización en capas (pantallas, componentes, servicios, contexto y tipos), los casos de uso y el modelado de datos. Este capítulo también recoge la explicación de cada una de las pantallas de la app, su interfaz y comportamiento, así como los componentes reutilizables que se han desarrollado.

El capítulo de *Prototipos y desarrollo* explica la evolución del proyecto a través de tres prototipos sucesivos. Para cada versión se describe qué funcionalidades se implementaron, cómo evolucionó la interfaz y qué problemas técnicos se resolvieron.

Finalmente, el capítulo de *Conclusiones y mejoras futuras* recoge una síntesis de los resultados obtenidos, una reflexión sobre las decisiones adoptadas y las dificultades encontradas, así como una enumeración de posibles mejoras y ampliaciones que se podrían abordar en trabajos futuros. Al final de la memoria se incluye la bibliografía que sustenta las afirmaciones del texto.

1.4. Recursos utilizados

- **Hardware.** El desarrollo se llevó a cabo en un ordenador portátil con procesador Intel i7 y 16 GB de memoria RAM. Para las pruebas se emplearon emuladores de Android (API 33) y un teléfono Android real para verificar el funcionamiento en un dispositivo físico.

- **Software.** Se utilizó el entorno de desarrollo `ReactNative` con TypeScript a través de la CLI oficial. Las dependencias del proyecto se gestionaron con `npm` y `Node.js`. El editor de código principal fue Visual Studio Code con extensiones de ayuda a la escritura de React y TypeScript. Para el backend y la autenticación se usó Firebase (`@react-native-firebase/app`, `auth` y `firestore`). El control de versiones se realizó con Git y GitHub. La documentación de la memoria se redactó en LaTeX empleando la clase `report` y se generaron diagramas UML con PlantUML.

- **Bibliotecas y librerías de la app.** En la capa de presentación se empleó `@rneui/themed` para los componentes de interfaz y `react-native-vector-icons` para la iconografía. La navegación entre pantallas se implementó con `@react-navigation/native`. Para

la búsqueda local se utilizó `react-minisearch`, que permite indexar y filtrar ingredientes en memoria con baja latencia. La gestión del estado global se resolvió mediante la API de contexto de React. Para la carga de imágenes se usaron bibliotecas nativas de `react-native-image-picker`. La aplicación se escribió íntegramente en TypeScript, y el tipado de los datos se definió en `src/Types/Types.tsx`.

- **Referencias y documentación.** Las bases teóricas del *meal planning* y de la usabilidad provienen de estudios recientes y obras de referencia citadas en la bibliografía, como los trabajos de la Organización Mundial de la Salud sobre obesidad y sobrepeso, las investigaciones de la Biblioteca Nacional de Medicina sobre planificación dietética382000137647362†L372-L383 y los principios de usabilidad de Nielsen382000137647362†L410-L418. Además se consultaron la documentación oficial de React Native, Firebase y MiniSearch.

1.5. Planificación temporal

1. **Fase de análisis y requisitos (semanas 1–2).** Se definieron los objetivos del proyecto, se identificaron las necesidades de los usuarios y se recopilaron requisitos funcionales y no funcionales. Durante esta fase se investigaron las aplicaciones existentes de planificación de comidas y se revisó bibliografía para comprender mejor el problema.

2. **Diseño de la interfaz y arquitectura (semanas 3–4).** Se elaboraron bocetos de las pantallas y diagramas UML de la arquitectura del sistema, se definieron las entidades de datos y se seleccionaron las tecnologías y librerías a utilizar.

3. **Implementación del prototipo 1 (semanas 5–8).** Se construyó una primera versión de NutriPlan que permitía el registro e inicio de sesión del usuario, la creación y visualización de recetas básicas y una pantalla principal con selección de día y tipo de comida. En esta etapa se configuró la base de datos en Firebase y se probaron las operaciones CRUD de recetas.

4. **Implementación del prototipo 2 (semanas 9–12).** Se añadieron la funcionalidad de planificación semanal, la despensa y la generación de la lista de la compra. Se integró MiniSearch para la búsqueda de ingredientes y se ajustó la estructura de datos para permitir la consolidación de cantidades.

5. **Implementación del prototipo 3 (semanas 13–15).** Se completaron las funcionalidades pendientes: subida de imágenes para las recetas, edición de ingredientes desde la despensa, marcaje de elementos comprados y mejoras de usabilidad. También se realizaron pruebas de integración y se refinaron los estilos y la navegación.

6. **Pruebas y documentación (semanas 16–17).** Se efectuaron pruebas unitarias y de integración para validar el correcto funcionamiento de los servicios y componentes. Posteriormente se elaboró la documentación de la aplicación y se redactó la memoria final.

Capítulo 2

Estado del arte

2.1. Fundamentos

La planificación dietética o Meal prepping

Hoy en día es difícil tener un estilo de vida saludable dada la falta de tiempo que causan las responsabilidades como el trabajo, familia, sueño, etc; varios datos muestran que los jóvenes consumen mucha menos fruta y verdura de la cantidad recomendada diaria, y que estos consumen mucho más comida basura de la que se recomienda [10].
Otro análisis indica que la mayor barrera que impide a las personas de seguir una dieta saludable es la falta de tiempo, el no querer abandonar alimentos favoritos y los elevados precios de las comidas sanas [10]. A mucha gente le cuesta mantener ese balance entre trabajar y tener una vida sana y he de ahí donde surge la planificación de dietas o meal planning, una práctica que consiste en planificar las comidas de un periodo futuro determinado (generalmente una semana). La mayor parte de las personas planifican tres comidas por día para los días laborales (de lunes a viernes), pero esta práctica es completamente personalizable.[18].

El método de uso de esta práctica puede variar dependiendo de la persona, pero la metodología más común es la siguiente.

1. Se elabora una tabla en la que se escriben todos los días de la semana en las columnas y los tipos de comida (desayuno, comida, cena, y en su caso merienda o aperitivo) en las filas (Figura 2.1). En cada celda se anotan las recetas o alimentos que se consumirán en el día y la comida correspondiente. A pesar de que este proceso se puede realizar en papel, resulta preferible hacerlo en un ordenador ya que facilita la edición y corrección de posibles errores. La tabla resultante será similar a la de la Figura 2.2.

Días / Comidas	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado	Domingo
Desayuno							
Comida							
Cena							
Aperitivos							

Figura 2.1: Plantilla planificación de comidas

Días / Comidas	**Lunes**	**Martes**	**Miércoles**	**Jueves**	**Viernes**	**Sábado**	**Domingo**
Desayuno	Avena con plátano y leche	Tostadas integrales con aguacate y café	Yogur con granola y frutos rojos	Huevos revueltos con pan integral	Tostadas con mermelada y queso fresco	Tortitas de avena con miel y fruta	Croissant + café con leche
Comida	Pollo a la plancha con arroz y ensalada	Lentejas estofadas con verduras y pan	Pasta integral con salsa de tomate y albóndigas	Pechuga de pavo con quinoa y ensalada	Paella de verduras o mixta	Hamburgues a casera con patatas al horno	Asado de pollo con patatas y ensalada
Cena	Crema de calabacín y tortilla francesa	Ensalada de atún con garbanzos	Sopa de verduras + pescado a la plancha	Ensalada caprese con pan tostado	Pizza casera con masa integral	Crema de zanahoria + ensalada ligera	Sopa de pollo + ensalada de pasta
Aperitivos	Yogur natural con frutos secos	manzana	Un puñado de almendras	Batido de plátano con leche	Batido de plátano con leche	Uvas	Yogur griego con miel

Figura 2.2: Plantilla planificación de comidas

2. Una vez completada la tabla con las recetas de la semana, se procede a listar los ingredientes necesarios para su preparación, incluyendo las cantidades específicas de cada uno (ver Tablas 2.1 y 2.2).

Cuadro 2.1: Ejemplo de recetas con ingredientes comunes

Receta	**Ingredientes**	**Cantidad (para 2 personas)**
Pollo a la plancha con arroz y ensalada	Pechugas de pollo, arroz, lechuga, tomate, aceite de oliva, sal, pimienta, limón	Pollo 300 g; Arroz 120 g; Lechuga 100 g; Tomate 100 g; Aceite 10 ml
Asado de pollo con patatas y verduras	Pollo, patatas, zanahoria, cebolla, aceite de oliva, sal, pimienta	Pollo 500 g; Patatas 400 g; Zanahoria 200 g; Cebolla 100 g; Aceite 20 ml

Cuadro 2.2: Lista de la compra combinada (optimizada)

Ingredientes	**Cantidad total**
Pollo	800 g
Arroz	120 g
Lechuga	100 g
Tomate	100 g
Patatas	400 g
Zanahoria	200 g
Cebolla	100 g
Aceite de oliva	30 ml
Condimentos (sal, pimienta, romero, limón)	Al gusto

3. Al obtener la lista de ingredientes necesarios para preparar las recetas planificadas, se revisan los productos disponibles (en el caso de un usuario común, el hogar). Esto permite la identificación de ingredientes que ya se poseen, haciendo que se puedan eliminar de la lista y ajustar la cantidad de los que se tengan parcialmente, evitando compras innecesarias y reduciendo el desperdicio alimentario. En nuestro ejemplo se supondrá que la despensa tiene 500 gramos de pollo y 100 de arroz (Tabla 3), dando a la tabla de ingredientes final (Tabla 3))

Cuadro 2.3: Proceso de cálculo de ingredientes según la despensa

Ingredientes	**Necesario**	**En despensa**	**A comprar**
Pollo	800 g	500 g	300 g
Arroz	120 g	100 g	20 g
Lechuga	100 g	0	100 g
Tomate	100 g	0	100 g
Patatas	400 g	0	400 g
Zanahoria	200 g	0	200 g
Cebolla	100 g	0	100 g
Aceite de oliva	30 ml	0	30 ml
Condimentos	Al gusto	0	Al gusto

Cuadro 2.4: Lista de la compra final optimizada

Ingredientes	**Cantidad a comprar**
Pollo	300 g
Arroz	20 g
Lechuga	100 g
Tomate	100 g
Patatas	400 g
Zanahoria	200 g
Cebolla	100 g
Aceite de oliva	30 ml
Condimentos (sal, pimienta, romero, limón)	Al gusto

4. Ahora que se ha obtenido la lista definitiva, es recomendable revisar la lista para ver si falta o se ha duplicado algún ingrediente, este paso no es necesario, pero ayuda a prevenir fallos. Después de esto faltaría solo realizar la compra.

En los últimos años la planificación dietética ha ganado popularidad gracias a diversos factores sociales y culturales. De entre ellos destacan el auge de las redes sociales y la creciente influencia del culturismo y el deporte en la vida cotidiana. Plataformas como Instagram, YouTube o TikTok han facilitado el acceso a información sobre estos tópicos, motivando a muchos a estructurar sus dietas de forma más consciente y personalizada. Esta tendencia se observa también en el ámbito deportivo, como por ejemplo en el culturismo, donde el *meal planning* es una práctica habitual y está estrechamente ligada a la preparación física. Varios estudios recientes muestran que los practicantes del culturismo adoptan esta práctica como parte fundamental de su entrenamiento, y que su fuente de información principal son las redes sociales y la comunidad digital [8, 13, 1].
Este fenómeno no solo se refleja en ámbitos deportivos como el culturismo, sino también en la población general. Una investigación francesa descubrió que el 57,4 % de los habitantes planifican sus comidas al menos de manera ocasional, lo que evidencia un interés extendido en esta práctica como parte de la vida diaria [5]. De manera similar, en los Estados Unidos se estima que alrededor del 37 % organizan sus comidas con uno o dos días de antelación, lo que muestra el aumento del uso de la programación de menús fuera del ámbito deportivo [7]. Esta convergencia entre el impacto social de las plataformas digitales, las prácticas del culturismo y la creciente adopción de *meal planning* en la población aporta una razón y contexto sólidos para el desarrollo de herramientas digitales como *NutriPlan*, cuyo objetivo es facilitar la organización alimentaria de manera sencilla, eficiente y accesible.

El mercado de aplicaciones de planificación dietética está en constante expansión. Du-

rante 2024 su valor alcanzó los 2,21 millones de dólares y se proyecta que crecerá al valor de 5,53 millones en 2033 [2]. Esta expansión viene impulsada por las nuevas tendencias como la de integración de la inteligencia artificial para la personalización de menús, la incorporación de prácticas sostenibles y la creación de comunidades digitales en torno a la alimentación. Además, la pandemia Covid-19 aceleró esta adopción al fomentar la preparación de comidas en casa y la planificación de las compras. Si se añade a la perspectiva el informe a nivel global de McKinsey, donde se afirma que el 50 % de los consumidores prioriza una alimentación saludable y más del 70 % desea mejorar su dieta, se demuestra un gran interés en este tópico [14].

Los beneficios que aporta la planificación dietética son los que generan el interés por este movimiento, caracterizada por la falta de tiempo para funciones básicas como el sueño, la práctica de deporte o la cocina. La organización de comidas no sólo asegura una ingesta calórica y nutricional adecuada, sino que también contribuye en diversos aspectos de la vida diaria.

Ahorro de tiempo. Una de las principales dificultades para mantener hábitos alimenticios saludables es la falta de tiempo. Según análisis hechos por la Biblioteca Nacional de Medicina de EE. UU., esta práctica reduce las compras improvisadas durante la semana y el tiempo empleado en decidir qué cocinar cada día, mejorando la eficiencia y disminuyendo la improvisación.

Reducción del estrés. El meal planning ayuda a disminuir la tensión y el estrés derivados de la indecisión diaria sobre qué comer. Muchas personas deciden qué cocinar o comer en momentos de fatiga o poca motivación, lo que les conduce a optar por opciones menos saludables. Disponer de un plan y de los ingredientes necesarios facilita la elección de comidas más equilibradas y reduce la carga mental asociada a esta tarea.

Mejora de la dieta. Diversos estudios han demostrado que cocinar en casa con mayor frecuencia se asocia a un consumo menor de carbohidratos, azúcares y grasas [9]. Además, la planificación dietética incentiva a cumplir objetivos calóricos y nutricionales mediante la compra anticipada de ingredientes. La evidencia científica indica que esta práctica está relacionada con una mejor calidad de dieta y una menor prevalencia de obesidad, lo que la hace potencialmente relevante en estrategias de prevención [5].

Ahorro económico. El uso de una lista de la compra estructurada se asocia con una mayor calidad de la dieta y, al mismo tiempo, un gasto más eficiente, ya que evita la adquisición de productos innecesarios [4], haciendo que ésta práctica beneficie tanto a la salud como a la economía.

Reducción del desperdicio alimentario. Más de 59 millones de toneladas de residuos alimentarios, con un valor estimado en 132 millones; son generados anualmente en la Unión Europea. Una de las principales causas identificadas es la falta de planificación en la compra de alimentos en los hogares [6]. El *meal planning* contribuye en la mitigación de este problema, al fomentar la compra de cantidades ajustadas a las necesidades reales.

2.2. Diseño de la interfaz

El diseño de la inteefaz de usuario juega un papel muy importante en el uso y la aceptación de la aplicación. De hecho el diseño y usabilidad son requisitos esenciales para el éxito de este tipo de apps [11]. Varios estudios señalan que los usuarios tienden a aceptar apps con facilidad de uso, utilidad percibida y calidad de contenido que ofrece la aplicación [12]. Todo esto demuestra que una interfaz agradable, intuitiva y adaptada a las necesidades del usuario mejora la experiencia de uso de esta y crea un sentimiento positivo hacia la aplicación. Por ello se escogieron varios principios de diseño de interfaz y experiencia de usuario como el diseño minimalista, el uso de colores específicos, etc.

2.2.1. Diseño minimalista

El disño minimalista es una corriente estética y funcional que se basa en reducir los elementos de una interfaz o producto al mínimo posible sin quitar lo esencial, lo que elimina la información innecesaria y destaca la fundamental. Esta corriente sigue la premisa de "menos es más", frase popular del arquitecto Ludwig Mies van der Rohe, que subraya la importancia de la simplicidad y la funcionalidad en el diseño [19]. Jakob Nielsen, experti en usabilidad enfatiza que las interfaces deberían evitar la sobrecarga de información, mostrando solo lo esencial [15]. Este diseño se basa en los siguientes principios:

1. **Simplicidad:** mantiene solo los elementos necesarios para comunicar la información.
2. **Funcionalidad:** enfatiza que cada componente en el diseño debe tener un propósito claro haciendo que se eliminen elementos decorativos que no aportan ningún propósito claro haciendo que se elmininen elementos decorativos que no aportan ningún valor funcional.
3. **Uso de espacio blanco o "negativo":** se usa en la creación de un equilibrio entre los componentes y la redirección de la atención del usuario.
4. **Jerarquía visual:** el uso de una jerarquía visual que favorezca y organice la información de manera que los elementos más importantes se destaquen, creando una interfaz navegable y comprensible.

2.3. Análisis de librerías y frameworks para el frontend y la lógica

En este apartado se explican qué son los frameworks y librerías más relevantes utilizados en el proyecto, se presentan las alternativas principales del mercado con sus ventajas e inconvenientes, y se justifica la elección final. Para facilitar la lectura se aborda con detalle React Native y Firebase (pues condicionan arquitectura y despliegue).

2.3.1. ¿Qué es un framework / librería en este contexto?

En el ámbito del desarrollo móvil, un framework es un conjunto estructurado de herramientas, APIs y convenciones que facilita la creación de aplicaciones; una librería es un módulo reutilizable que proporciona funcionalidad concreta (UI, comunicación, almacenamiento, etc.). La elección de framework condiciona la arquitectura, el flujo de trabajo y las opciones de despliegue, mientras que la selección de librerías define aspectos concretos de la experiencia de desarrollo y del producto final.

2.3.2. React Native (análisis profundo)

React Native es un framework de código abierto propuesto por Meta que permite construir aplicaciones móviles utilizando JavaScript/TypeScript. A diferencia de enfoques híbridos que renderizan UI en un WebView, React Native mapea componentes declarativos a widgets nativos de cada plataforma.

Alternativas principales

- **Flutter (Google).** Framework que usa Dart y un motor propio de renderizado; dibuja toda la UI con su propia capa gráfica.
- **Desarrollo nativo (Kotlin/Java para Android, Swift/Objective-C para iOS).** Máximo control y optimización, código específico por plataforma.
- **Soluciones híbridas (Ionic/Cordova, Capacitor).** Aplicaciones web empaquetadas con acceso a APIs nativas mediante puentes.

Ventajas y desventajas de cada alternativa

- **React Native**
  - Ventajas: reutilización de código entre plataformas, ecosistema maduro de librerías JS/TS, integración sencilla con módulos nativos, curva de entrada baja para desarrolladores web.
  - Inconvenientes: en casos de UI muy compleja o rendimiento extremo puede requerir módulos nativos; la coordinación de versiones entre runtime y librerías exige mantenimiento.
- **Flutter**
  - Ventajas: rendimiento cercano a nativo; control total del rendering; experiencia UI homogénea entre plataformas.
  - Inconvenientes: requiere aprender Dart y reescribir la UI; menor interoperabilidad directa con bibliotecas JS/TS del ecosistema web existente.

- **Desarrollo nativo**
  - Ventajas: máximo rendimiento y acceso completo a APIs; ideal para apps con requisitos muy específicos.
  - Inconvenientes: doble esfuerzo de desarrollo y mantenimiento por plataforma; mayor coste y tiempo.
- **Ionic / Cordova / Capacitor**
  - Ventajas: desarrollo rápido con tecnologías web; buena opción para apps sencillas.
  - Inconvenientes: limitaciones de rendimiento y experiencia nativa; comportamiento divergente en gestos y componentes nativos.

Razones para elegir React Native en este proyecto La elección de React Native responde a un equilibrio entre productividad y control técnico: permite desarrollar rápidamente con TypeScript, reutilizar lógica en toda la aplicación y aprovechar un amplio ecosistema (UI kits, navegación, integración con Firebase). Dado el alcance funcional de NutriPlan (formularios, listas, modales, acceso a cámara/galería y persistencia en la nube) React Native ofrece la combinación adecuada de eficiencia de desarrollo y capacidad para escalar con módulos nativos cuando sea necesario.

2.3.3. Otras librerías y componentes

Gestión de estado y contexto

Dada la complejidad de las aplicaciones y la variedad de componentes es complicado compartir datos entre distintas partes del programa. Un contexto se encarga de pasar información entre diferentes partes y niveles de una aplicación. Existen varias alternativas disponibles para esta función como:

- **React Context:** módulo de React que permite compartir funciones y estados entre componentes como si fueran variables globales. Es un componente nativo de React y no requiere de dependencias externas, fácil de añadir. Su único inconveniente es que afecta al rendimiento del software si no se planifica bien y no es compatible con estados o efectos complejos.
- **Redux:** librería de código abierto para el manejo de estados en JavaScript que permite mantener un estado globalizado en un componente llamado tienda (*store*) y actualizarlo cuando sea necesario mediante las acciones guardadas en las funciones llamadas *reducers*. Redux permite tener un estado predecible, escalable, fácil de depurar y consistente, pero al mismo tiempo no es un componente nativo de React, es

complejo y por ende requiere experiencia y más código de lo usual para configuraciones básicas y excesivo para proyectos pequeños.

- **Zustand:** concepto similar a los anteriores solo que tiene una API muhco más simple que la de redux y un enfoque minimalista. Su simplicidad, flexibilidad, rapidez y no necesidad de *wrappers* como `React Context` le hacen eficiente y fácil de utilizar.

Se ha usado `React Context` para el uso del contexto gracias a su facilidad de instalación, que se realiza en un sólo comando; su simplicidad y al tener una aplicación no tan compleja, esta librería no afectaría en el rendimiento.

Navegación

La navegación en este caso es necesitada para la conexión entre pantallas permitiendo al usuario acceder a todo el contenido de la aplicación en un orden específico. Existen varias alternativas como:

- **React Native Navigation**: librería creada por *React Native* que permite la navegación e implementación de pilas (*stack*), pestañas y bandejas de navegación, se integra con gestos y la navegación nativa de React Native, es fácil de utilizar, pero menos eficiente en grandes proyectos.

- **React Native Navigation(Wix)**: a diferencia de *React Native Navigation* que usa JavaScript, esta usa las API nativas de navegación de iOS y Android, es más eficiente en grandes proyectos pero difícil de configurar.

Al estar trabajando en un proyecto simple que no requiere de muchas páginas ni tampoco utiliza pestañas o bandejas de navegación se ha utilizado *React Native Navigation* para controlar las pantallas y su acceso dada su simplicidad de uso, fama y eficiencia en programas pequeños.

`@rneui/themed` (React Native Elements) y UI kits

Conjunto de componentes de interfaz listos para usar (botones, inputs, cards, etc.). Alternativas: React Native Paper, NativeBase, UI Kitten. Ventajas: aceleración en la construcción de pantallas, tematización central, componentes coherentes. Inconvenientes: peso adicional en la app y limitaciones si se requiere personalización extrema. Razón de elección: `@rneui/themed` proporciona los componentes que mejor encajan con el estilo requerido y permite personalizaciones sencillas sin reescribir elementos base; se usa en varios componentes del proyecto.

Búsqueda local: `react-minisearch` (MiniSearch)

Motor de búsqueda en memoria ligero para texto indexa documentos y permite búsquedas por prefijo y ranking. Alternativas: Fuse.js (búsqueda difusa), Lunr.js (índice invertido similar a motores más grandes). Ventajas: MiniSearch es muy ligero, simple de integrar y suficiente para listas pequeñas/medianas como el catálogo de ingredientes. Inconvenientes: no diseñado para grandes volúmenes o búsqueda distribuida. Por qué se escogió: en `IngredientSearchSelectorComponent` en la app se requiere latencia mínima y funcionamiento offline; MiniSearch satisface estos requisitos con bajo coste.

Iconos: `react-native-vector-icons` (Ionicons)

Colección de iconos vectoriales para React Native con adaptadores para múltiples familias (Ionicons, Material, etc.). Alternativas: usar SVGs con `react-native-svg`, o iconos del sistema. Ventajas: fácil uso, consistencia visual y escalabilidad; amplia disponibilidad de glyphs. Inconvenientes: usado en cabeceras y botones.

Pickers y selectores: `CustomPicker` y bibliotecas

Es un componente local `CustomPicker` que abre un `Modal` con opciones para seleccionar el `quantityType`(unidad de métrica del ingrediente). Alternativas: `@react-native-picker/picker` (nativo) o `react-native-dropdown-picker` (más características). Ventajas del enfoque actual: control total sobre la apariencia del modal y la interacción; comportamiento consistente en iOS/Android. Inconvenientes: requiere implementar accesibilidad y animaciones manualmente. Razón de elección: necesidad de un comportamiento visual concreto integrado en `IngredientComponent` y de un control simple sobre apertura/cierre mediante las props `isPickerOpen` / `setIsPickerOpen`.

Imágenes y selección de medios

Librería de gestión de imágenes mediante bibliotecas nativas (por ejemplo `react-native-image-picke`
Alternativas: Expo Image Picker, soluciones servidoras. Ventajas: acceso nativo a cámara/-galería y control de permisos. Inconvenientes: configuración nativa inicial. Razón: se eligió la solución que ofrece más control sobre permisos y procesado de imágenes en la ejecución nativa, no se puede utilizar la opción de *Expo* ya que le proyecto no fué programado de manera nativa.

2.4. Análisis de librerías y frameworks para back-end

2.4.1. Base de datos

Es el software encargado de guardar todos los datos necesarios del usuario y tenerlos al alcance en cualquier momento. Existen varias opciones que cubren nuestras necesidades,

desde bases de datos locales hasta en la nube.

- **SQLite.** Es una base de datos relacional autónoma y ligera que es usada en proyectos locales y en línea; utiliza SQL como su lenguaje de programación, su instalación en *React Native* es fácil e intuitiva. A pesar de que SQLite tiene esos beneficios, su integración en aplicaciones que necesitan bases de datos en la nube resulta complicado y costoso dado que SQLite no incorpora de forma nativa la sincronización de datos online.

- **Firebase.** Plataforma creada por Google que integra servicios de autenticación, bases de datos en tiempo real, almacenamiento de ficheros, y herramientas auxiliares para análisis y despliegue. La mayor parte de sus servicios son gratuitos(hasta superar un límite establecido), a pesar de que su base de datos no sea relacional, no resulta difícil usarla como tal; su integración con *React Native* se realiza en varios pasos y es muy simple; y por último tiene una comunidad muy grande que aporta documentación y *feedback* extenso. En cuento a sus desventajas, estas son el no ser óptimo en relaciones o búsquedas complejas y su pago.

- **SupaBase.** Base de datos de código abierto que utiliza *PostgreSQL* haciéndola muy robusta y potente, tiene atuenticación almacenamiento y funciones serverless(no requieren de servidor) y es gratuito gracias a su filosofía de código abierto. En cambio, no es tan maduro ni ampliamente probado en aplicaciones móviles (no tanto como Firebase), tiene una configuración e uso complejo y técnico y su documentación no es tan extensa.

- **MongoDb.** Base de datos local que contiene sincronización nativa con la nube, resulta óptima para aplicaciones que requieren de almacenamiento en y sin línea (online y offline) y el tipo de base de datos es similar al de *Firebase*(no relacional). Sin embargo, su uso resulta complejo al añadir lógica personalizada al backend, no tiene una comunidad tan madura con React Native.

Para el backend se necesita una base de datos para una aplicación que no consume, no contiene muchos datos; que la base de datos sea minimalista en el aspecto del uso e instalación, pensada para app móviles y no tenga ningún coste económico. Firebase cumple con todas estas condiciones y además contiene servicios de autenticación y reglas de seguridad, por ello se escogió este como software de almacenamiento.

2.5. Proyectos actuales

Aplicaciones móviles existentes de meal planning

El interés creciente por planificar la alimentación doméstica ha dado lugar a aplicaciones con enfoques bien diferenciados: desde soluciones orientadas a la rapidez y conveniencia

hasta plataformas centradas en el control nutricional o en la compra integrada. A continuación se exponen las diferencias esenciales entre las propuestas comerciales más conocidas y NutriPlan.

- ***Mealime*** apuesta por la simplicidad y la velocidad: menús semanales preconfigurados y listas de compra automáticas reducen la fricción para el usuario que busca solucionar comidas sin preocuparse por detalles. Esa filosofía simplifica la adopción inicial pero sacrifica flexibilidad; las recetas y porciones estandarizadas limitan la capacidad de ajustar ingredientes, sustituir productos o conciliar con el inventario real del hogar. Para una herramienta cuyo valor diferencial es precisamente optimizar la lista de la compra en función de la despensa, esa pérdida de precisión es significativa.

- ***Yazio*** sitúa el foco en el seguimiento nutricional: ofrecer conteo calórico y trazabilidad de macronutrientes requiere una base de datos nutricional rica y una interfaz que guíe al usuario en la introducción y el ajuste de datos. Es una excelente opción para objetivos de salud, pero su complejidad y el uso extendido de funciones premium añaden barreras de entrada. En el contexto de este proyecto, donde la prioridad es validar un flujo práctico (receta → planificación → despensa → lista) la profundidad nutricional extrema no aporta tanto valor inmediato como la fiabilidad en el cálculo de la compra y la experiencia off-line.

- ***Instacart*** se orienta hacia la transacción: conecta recetas y menús con la compra directa en supermercados asociados. Su fortaleza es la integración comercial y la comodidad para comprar en pocos pasos; su limitación es la dependencia de cobertura regional y acuerdos con comercios, además de que el objetivo principal es la venta más que la personalización dietética o la reducción del desperdicio.

Frente a estos enfoques, NutriPlan se diseñó con prioridades claras y deliberadas: ofrecer control granular sobre las recetas, garantizar la conciliación precisa con la despensa del usuario, funcionar con baja latencia (incluso sin conexión) y mantener la propiedad y privacidad de los datos. Estas prioridades se tradujeron en decisiones técnicas y de producto concretas:

- Precisión y granularidad: cada receta modela ingredientes con cantidad y unidad (`RecipeIngredient` en `src/Types/Types.tsx`), lo que permite cálculos exactos para la lista de la compra. Aunque exige mayor entrada de datos por parte del usuario, reduce el desperdicio y mejora la utilidad práctica del sistema.

- Disponibilidad y rendimiento: la indexación local (MiniSearch) y la persistencia de Firestore proporcionan búsquedas instantáneas y operación offline para tareas frecuentes, mejorando la experiencia en contextos reales como tiendas con cobertura limitada.

- Privacidad y propiedad de datos: los datos se aíslan por usuario (`uid`) y las reglas de seguridad de Firestore protegen la información personal; esta opción refuerza la confianza del usuario y facilita la portabilidad de los datos.
- Modelado equilibrado para BaaS: el diseño de datos combina referencias y elementos embebidos para minimizar lecturas innecesarias en Firestore y mantener pantallas reactivas (p. ej. `MainScreen`, `GroceryListScreen`), reduciendo costes operativos y latencia.
- Onboarding progresivo y usabilidad: la experiencia se organiza para que el usuario comience con tareas esenciales (añadir recetas, planificar, generar lista) y pueda acceder a funcionalidades más complejas solo si lo desea; la interfaz usa modales y componentes reutilizables (`src/Components/`) para minimizar el esfuerzo cognitivo.
- Modularidad y evolución: la separación en capas (Components, Screens, Services, Context, Types) permite validar la idea con rapidez y mantener la puerta abierta a integraciones futuras (APIs de supermercados, motores de recomendación, o servicios avanzados de nutrición) sin reescribir la interfaz.

En conjunto, NutriPlan se sitúa entre las aplicaciones sencillas orientadas a la conveniencia y las plataformas complejas centradas en la nutrición o en el comercio electrónico. La elección de priorizar precisión en las recetas, conciliación con la despensa y operación offline responde a un objetivo práctico: reducir desperdicio y ofrecer una herramienta útil en el día a día del usuario. Estas decisiones, aunque aumentan la complejidad técnica inicial, maximizan el valor real entregado y facilitan evolucionar la aplicación hacia funcionalidades adicionales cuando la validación con usuarios lo aconseje.

2.6. Conclusiones

Tras analizar las distintas tecnologías y herramientas disponibles para el desarrollo de aplicaciones móviles, especialmente en el contexto de la planificación de comidas, podemos extraer conclusiones relevantes sobre las decisiones tomadas en este proyecto:

Frameworks de desarrollo móvil

React Native ha demostrado ser la elección más adecuada frente a alternativas como Flutter o desarrollo nativo puro. Sus principales ventajas radican en:

- La capacidad de mantener una única base de código para iOS y Android, reduciendo significativamente el tiempo de desarrollo.
- Un ecosistema maduro de librerías que facilita la implementación de funcionalidades comunes.

- La posibilidad de aprovechar el conocimiento previo de JavaScript/TypeScript y React.

Sin embargo, es importante reconocer que Flutter podría ofrecer mejor rendimiento en aplicaciones con interfaces muy complejas o animaciones elaboradas. No obstante, para las necesidades de NutriPlan, donde prima la funcionalidad práctica sobre efectos visuales complejos, React Native proporciona el balance óptimo entre velocidad de desarrollo y rendimiento.

Backend as a Service (BaaS)

Firebase ha resultado ser superior a alternativas como Supabase o un backend personalizado para este proyecto específico por:

- Facilidad de integración con React Native mediante SDKs oficiales.
- Capacidades de sincronización offline que mejoran la experiencia del usuario.
- Reducción significativa en tiempo de desarrollo al no requerir infraestructura propia.

Aunque las demás opciones ofrecen ventajas en términos de código abierto y control sobre los datos, Firebase proporciona una solución más madura y probada para las necesidades actuales de la aplicación. La decisión de usar Firestore sobre Realtime Database se justifica por su modelo de datos más flexible y mejor soporte para consultas complejas.

Gestión de estado

La elección de React Context sobre alternativas como Redux o MobX se fundamenta en:

- Simplicidad en la implementación y menor curva de aprendizaje.
- Suficiente para la escala actual de la aplicación.
- Integración nativa con React sin dependencias adicionales.

Si bien Redux ofrece ventajas en aplicaciones más grandes, especialmente en términos de depuración y manejo de estados complejos, la sobrecarga que introduce no se justifica para el alcance actual de NutriPlan.

Navegación y ruteo

React Navigation ha demostrado ser superior a alternativas como React Native Navigation (Wix) en nuestro caso por:

- API más intuitiva y documentación más completa.

- Mejor integración con el ecosistema React Native.
- Soporte nativo para gestos y animaciones en iOS y Android.

UI Components

La combinación de React Native Elements (@rneui/themed) con componentes personalizados ha resultado más efectiva que usar soluciones completas como Native Base o UI Kitten porque:

- Permite mayor flexibilidad en el diseño de la interfaz.
- Reduce el tamaño final de la aplicación al incluir solo los componentes necesarios.
- Facilita la personalización y consistencia visual.

Búsqueda y rendimiento

La implementación de búsqueda local con MiniSearch ha demostrado ser más eficiente que alternativas como Fuse.js o búsquedas en servidor por:

- Mejor rendimiento en búsquedas sobre conjuntos de datos medianos.
- Funcionamiento offline sin depender de la red.
- Menor consumo de recursos del dispositivo.

Conclusión general

Las decisiones tecnológicas tomadas priorizan tres aspectos fundamentales:

1. Velocidad de desarrollo y prototipado rápido.
2. Experiencia de usuario fluida con buen rendimiento offline.
3. Mantenibilidad y escalabilidad del código.

Aunque existen alternativas que podrían ofrecer ventajas en aspectos específicos (rendimiento puro, control total sobre el backend, capacidades avanzadas de estado), las elecciones realizadas proporcionan el mejor compromiso para los objetivos del proyecto: crear una aplicación funcional, mantenible y con buena experiencia de usuario, permitiendo iteraciones rápidas y validación de funcionalidades con usuarios reales.

Capítulo 3

Diseño y descripción del sistema

3.1. Lógica de la aplicación

3.1.1. Decisiones de diseño: estructura, clases y casos de uso

En esta sección se explica de forma explícita y concisa qué decisiones de diseño se adoptaron en la aplicación y por qué se tomó cada una de ellas.

3.1.2. Casos de uso

En esta sección se enumeran y justifican los casos de uso principales de la aplicación *NutriPlan*. Cada caso de uso incluye la motivación para su inclusión en el sistema, la forma en que se materializó en la implementación (referencias a módulos y ficheros relevantes) y una discusión sobre alternativas de diseño que se valoraron durante el desarrollo, explicando por qué la solución adoptada resultó la más adecuada para los objetivos del proyecto.

Criterio de selección de los casos de uso

Los casos de uso se seleccionaron atendiendo a los requisitos funcionales básicos y a las prioridades del proyecto: permitir al usuario planificar sus comidas semanalmente, gestionar recetas e ingredientes, mantener un inventario de despensa y obtener una lista de la compra optimizada. Se dio preferencia a casos que maximizaran el beneficio al usuario final y que fueran implementables con un coste razonable de desarrollo.

1. Registro e inicio de sesión

La mayoría de aplicaciones permiten al usuario a crear una cuenta, iniciar sesión y mantener esta activa en el dispositivo. Esto es esencial para la separación de datos entre diferentes cuentas desde recetas privadas, despensas e planificaciones.
Para ello la manera más sencilla y eficiente de implementar este caso de uso es mediante la

creación de las pantallas de inicio y registro de sesión (*LoginScreen* e *RegisterScreen*) que permitan al usuario registrarse e iniciar sesión y mantener la cuenta activa en su dispositivo.
Su implementación es con Firebase que proporciona *Firebase Auth* que permite controlar el inicio de sesión y la autenticación.
Existen varias alternativas que satisfacerían estos casos de us como la autenticación mediante un backend propio (REST/API) o servicios como Auth0. Estos se descartaron por la sobrecarga operativa y de configuración; Firebase ofrece integración directa con React Native, SDKs maduros y un tiempo de puesta en marcha muy inferior, adecuado para un TFG cuyo objetivo es priorizar la funcionalidad de cara al usuario.

2. Gestión de recetas (CRUD)

El componente más importante de nuestra aplicación es la receta y como en cada lista, el usuario tiene el derecho de crear, editar, eliminar, programar y listar las recetas con sus ingredientes y metadatos.
Para ello se ha creado una arquitectura en la base de datos que separa entidades, funciones CRUD que comunican con la base de datos; pantallas y modales que muestran y permiten la edición de estos

3. Búsqueda de ingredientes y selección rápida

Ofrecer una búsqueda rápida de ingredientes para añadir a recetas o planificaciones. Para ello se creó una barra de búsqueda y se usó *mini-search* como motor de búsqueda. Una búsqueda rápida y disponible offline mejora notablemente la experiencia de creación de recetas. Indexar en cliente reduce latencia y dependencia de la red en tareas frecuentes. Hay alternativas como búsquedas remotas (consultas a Firestore, o un servicio externo) o usar Fuse.js. Se prefirió MiniSearch por su rendimiento y simplicidad cuando los volúmenes de datos son pequeños/medios; la búsqueda remota hubiera incrementado latencias y coste de lecturas.

4. Planificación semanal de comidas

Como lo usual en cada planificación de menús, se tiene que poder asignar recetas o ingredientes a días y comidas (desayuno/comida/cena) y visualizar la semana con las asignaciones. Para la Implementación de este caso se usan vistas (*View*) en `MainScreen` como orquestador, `HeaderComponent` para selección del día y servicios `weeklyMeals-db-services.ts` para conectar con Firebase.
Centralizar la planificación en `MainScreen` permite ofrecer una vista de contexto (lista de recetas previstas, botones de acción) y mantener la interacción rápida mediante modales para planificar. Se diseñó la entidad `WeeklyMeal` con campos mínimos que permiten tanto enlazar a una receta como a un ingrediente suelto. Alternativas: sistema de planificación

basado en reglas (p. ej. sugerencias automáticas por variedad nutricional) o planificación mediante calendarios externos. Estas ideas quedaron fuera del alcance inicial por aumentar la complejidad; se mantienen como mejoras futuras.

5. Gestión de despensa (inventario)

Uno de los beneficios de esta aplicación es la característica de mantener un inventario local con los ingredientes disponibles, actualizarlos en cualquier momento y hacer el cálculo automático de los ingredientes necesarios después de restar los presentes. Para ello, la implementación de `PantryScreen`, `ingredientPantry-db-services.ts` y componentes de tarjeta (`PlannedIngredientCard`, `IngredientCard`) para la visualización y el cálculo de los ingredientes planificados. La despensa es clave para calcular la lista de la compra optimizada; por ello se optó por un modelo sencillo (`PantryItem`) que almacena cantidad y unidad por usuario. Las operaciones de suma/resta se realizan en servicios para mantener lógica centralizada, otras alternativas serían la sincronización exclusiva con supermercado (APIs externas) o almacenamiento solo en servidor. Se optó por persistencia con Firestore y caching local para permitir uso offline y reducir latencias.

6. Generación y gestión de la lista de la compra

Calcular la cantidad a comprar sumando ingredientes planificados y restando existencias en la despensa; marcar artículos como comprados.
Implementación mediante la creación de `GroceryListScreen` para la muestra de los ingredientes a comprar, `groceryBought-db-services.ts` para la conexión con Firebase y `GroceryItem` que determina la estructura de los datos. Automatizar el cálculo de la lista de la compra es el valor diferencial central del producto. Al implementar este cálculo en la capa de servicios se asegura consistencia y evita duplicación de lógica en la UI. Marcar como comprado actualiza la despensa mediante operaciones atómicas o lotes para evitar desajustes. Se podía haber delegado el cálculo en un backend propio o en funciones serverless para mayor control. Se valoró, pero se priorizó la simplicidad operativa aprovechando Firestore y la capacidad de operaciones por lotes en clientes.

7. Gestión de imágenes y multimedia

Descripción: permitir asociar imágenes a recetas y gestionar permisos de cámara/galería. Implementación: subida a Firebase Storage y almacenamiento de `imageUrl` en documentos de receta; componentes que usan `react-native-image-picker`. Justificación: imágenes mejoran la usabilidad y ayudan al reconocimiento visual de recetas; Firebase Storage facilita el almacenamiento y entrega CDN-like. Alternativas: almacenar imágenes como base64 en documentos o usar servicios externos de CDN; se descartaron por coste y complejidad de integración.

Resumen y coherencia de los casos de uso

Los casos de uso seleccionados cubren el flujo completo esperado por el usuario: registrar y acceder a su cuenta, crear y organizar recetas, planificar menús, verificar la despensa y generar la lista de la compra. Cada caso se implementó priorizando latencia baja, disponibilidad offline y simplicidad de mantenimiento: decisiones que se reflejan en la estructura de ficheros (`src/Services/`, `src/Screens/`, `src/Components/`) y en el modelado de datos (`src/Types/Types.tsx`). Las alternativas más complejas (backend propio, normalización extrema, trazabilidad con Redux) se consideraron pero se descartaron por coste de desarrollo o por no aportar suficiente ventaja en la fase inicial del proyecto. Estas decisiones dejan, no obstante, margen para futuras migraciones y escalado cuando los requisitos lo exijan.

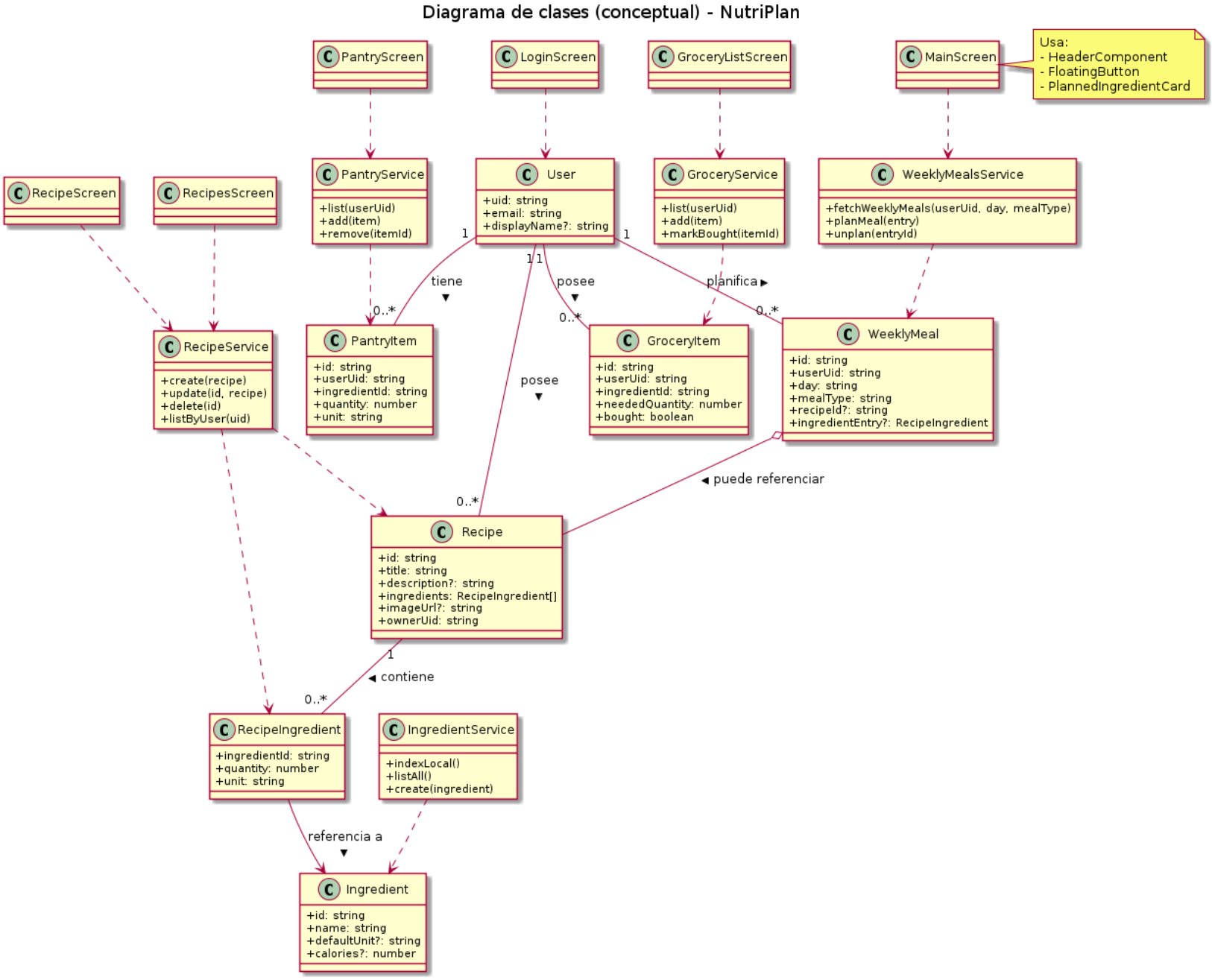


Diagrama de clases (conceptual) - NutriPlan
PantryScreen
LoginScreen
GroceryListScreen
MainScreen
Usa:
- HeaderComponent
- FloatingButton
- PlannedIngredientCard
RecipeScreen
RecipesScreen
PantryService
+list(userUid)
+add(item)
+remove(itemId)
User
+uid: string
+email: string
+displayName?: string
GroceryService
+list(userUid)
+add(item)
+markBought(itemId)
WeeklyMealsService
+fetchWeeklyMeals(userUid, day, mealType)
+planMeal(entry)
+unplan(entryId)
1
tiene
0..*
1
posee
0..*
1
planifica
0..*
RecipeService
+create(recipe)
+update(id, recipe)
+delete(id)
+listByUser(uid)
PantryItem
+id: string
+userUid: string
+ingredientId: string
+quantity: number
+unit: string
posee
GroceryItem
+id: string
+userUid: string
+ingredientId: string
+neededQuantity: number
+bought: boolean
WeeklyMeal
+id: string
+userUid: string
+day: string
+mealType: string
+recipeId?: string
+ingredientEntry?: RecipeIngredient
puede referenciar
0..*
Recipe
+id: string
+title: string
+description?: string
+ingredients: RecipeIngredient[]
+imageUrl?: string
+ownerUid: string
1
contiene
0..*
RecipeIngredient
+ingredientId: string
+quantity: number
+unit: string
IngredientService
+indexLocal()
+listAll()
+create(ingredient)
referencia a
Ingredient
+id: string
+name: string
+defaultUnit?: string
+calories?: number

3.1.3. Justificación del diagrama de clases y decisiones de diseño

En este apartado se justifican las decisiones que motivaron la separación representada en el diagrama de clases conceptual y se explica por qué cada componente, clase y relación fue diseñada de la manera mostrada.

Criterio general de separación

La arquitectura sigue un principio de separación por responsabilidades (single responsibility, bajo acoplamiento y alta cohesión). Se decidió distinguir claramente entre:

- Entidades del dominio llamadas tipos en *TypeScript* que son similares a las clases (tipos que modelan los datos): representan la forma y semántica de los objetos (User, Recipe, Ingredient, RecipeIngredient, WeeklyMeal, PantryItem, GroceryItem).
- Servicios por dominio (capa Services): encapsulan la lógica de persistencia y las operaciones con la base de datos de Firebase.
- Vistas/controladores (Pantallas y Componentes): orquestan flujos de interacción y presentan datos al usuario.
- Contexto y tipado (Context / Types): comparten estado global y contratos tipados entre capas.

Esta separación permite modificar la fuente de datos (por ejemplo migrar Firebase por otro BaaS) sin cambiar la UI, facilita la escritura de pruebas unitarias y simplifica la localización de la lógica de negocio.

Entidades del dominio: elección de atributos

Las entidades modeladas en `Types.tsx` recogen únicamente los atributos necesarios para las operaciones de la aplicación, siguiendo criterios de minimalismo y practicidad.

- *User* (usuario): uid, email, displayName. Razonamiento: el identificador único (uid) es la clave primaria en Firebase y es suficiente para asociar propiedad y seguridad. Los metadatos se reducen al mínimo imprescindible (email y nombre para mostrar). Otras opciones consideradas: añadir perfil extenso (roles, preferencias nutricionales). Se descartó por simplicidad y porque esas extensiones sólo se exigirán si el caso de uso lo demanda.
- *Recipe* (Receta): id, title, description, ingredients[], imageUrl, ownerUid. Razonamiento: una receta requiere un identificador, metadatos visibles (título, descripción), referencia al propietario para control de acceso y lista de ingredientes embebidos (RecipeIngredient) para renderizado rápido. ImageUrl se almacena como referencia a Firebase Storage. Alternativas: normalizar completamente los ingredientes (referencias

sólo a ids) para evitar duplicación. Se escogió un balance: almacenar RecipeIngredient con referencia al ingredientId y cantidades, lo que permite reconstruir la receta sin múltiples consultas complejas y mantiene integridad semántica.

- *Ingredient* (Ingrediente): id, name, defaultUnit, calories. Razonamiento: ingredientes tienen nombre y unidad por defecto; valores nutricionales se guardan opcionalmente. Mantener metadata ligera facilita indexación local (MiniSearch). Alternativas: modelo extensible con tablas de nutrición completas. Se reservó para iteraciones futuras.

- *RecipeIngredient* (Ingrediente de receta): ingredientId, quantity, unit. Razonamiento: estructura que refleja la relación entre receta e ingrediente con cantidad y unidad. Este tipo evita ambigüedades y permite operaciones aritméticas (sumas para la lista de la compra). Alternativas: modelo por ítem sin referencia (texto libre). Se desechó por perder control en cálculos y agregaciones.

- *WeeklyMeal / PantryItem / GroceryItem*: campos mínimos para identificar usuario, entidad referenciada, cantidad, unidad y estado. Razonamiento: el foco es soportar planificación, inventario y compras; por ello se incluyeron únicamente atributos necesarios para esas operaciones.

En todos los casos se priorizó la claridad semántica y la operatividad (facilidad de cálculo de listas de compra, conciliación con la despensa y sincronización con Firebase).

Relaciones entre entidades: multiplicidad y tipo

Las decisiones sobre multiplicidad (p. ej. User 1 – 0..* Recipe) reflejan la realidad del dominio:

- Un usuario puede poseer muchas recetas; una receta pertenece a un único propietario (ownerUid).

- Una receta contiene múltiples RecipeIngredient; cada RecipeIngredient referencia un Ingredient.

- WeeklyMeal puede referenciar una Recipe o un RecipeIngredient (soporte tanto para planificar por receta como para planificar por ingrediente suelto).

Se eligieron relaciones que favorecen consultas eficientes en Firestore (documentos con colecciones anidadas o referencias) y disminuyen la necesidad de joins costosos en el cliente.

Alternativas de modelado consideradas:

- Modelado completamente normalizado (separar por completo recetas/ingredientes y usar solo referencias). Pro: evita duplicación; Contra: aumenta número de lecturas y complejidad en tiempo de ejecución en un BaaS tipo Firestore.

- Desnormalización agresiva (duplica datos para lecturas más rápidas). Pro: lecturas simples y rápidas; Contra: mayor coste de mantenimiento y riesgo de inconsistencia.

Se adoptó un modelo mixto —referencias cuando interesa (ingredientId) y documentación embebida cuando favorece la eficiencia de lectura (detalles de receta necesarios en pantallas principales).

Servicios por dominio: motivos y alternativas

La capa `Services/` agrupa la lógica de acceso y transformación de datos desde Firebase por entidad, es decir, se crea un archivo por cada entidad. Esto centraliza el manejo de errores, facilita la futura sustitución del backend sin afectar a los componentes y permite pruebas unitarias. También existen varias alternativas, la más famosa y simple sería poner todo el código en un sólo archivo, lo que lo centraliza, pero al mismo tiempo dificulta su futura mejora dada su dificultad de comprensión (archivo muy extenso) lo que también dificultaría su futura mejora o implementación.

Por tanto se escogió la primera estrategia de servicios separados por entidad y dominio gracias a su claridad y testabilidad.

Screens y Components: razones de la componetización

La separación entre Screens (orquestadores de flujo) y Components (presentacionales y reutilizables) responde a:

- Reutilización visual: componentes como AppHeader, FloatingButton o IngredientCard se usan en múltiples pantallas.
- Testabilidad: los componentes puros son fáciles de testear aislados.
- Simplicidad en la navegación: screens combinan componentes y delegan la lógica de datos a services/context.

Alternativas: pantallas monolíticas con lógica embebida; se evitó por dificultar mantenimiento y proliferación de código duplicado.

Consideraciones sobre diseño de API y persistencia (Firestore)

Se optó por una estrategia acorde con Firestore:

- Minimizar número de lecturas necesarias para las vistas más frecuentes.
- Aprovechar la persistencia local y sincronización automática de Firestore.
- Diseñar reglas de seguridad basadas en ownerUid para proteger datos por usuario.

Alternativas como un backend propio permitirían consultas relacionales complejas y control total del esquema, pero aumentarían el coste de desarrollo y operación; por tanto se eligió Firebase para acelerar el desarrollo y delegar la operativa del backend.

Conclusión y balance de alternativas

La separación y el modelado adoptados son coherentes con los requisitos funcionales: soportar CRUD de recetas, planificación semanal, inventario y lista de la compra con latencia reducida y posibilidad de funcionamiento offline. Las alternativas valoradas (normalización completa, backend propio, Redux) aportan ventajas en ciertos escenarios; no obstante, el equilibrio entre rapidez de desarrollo, mantenibilidad y experiencia de usuario condujo a la decisión final reflejada en el diagrama. Las elecciones dejan además margen para evolución: la arquitectura modular facilita migraciones (por ejemplo sustituir Context por Redux o Firebase por Supabase) cuando la escala o los requisitos lo exijan.

Punto de entrada y control de sesión

El archivo *App.tsx* actúa como punto de entrada único y responsable de la inicialización de la navegación y del control de sesión. Esta decisión centraliza la lógica de autenticación (escucha del estado de Firebase Auth) y simplifica la condición de rutas (pantallas disponibles para usuario autenticado versus no autenticado), evitando dispersión de la lógica de sesión por la aplicación.

Separación en capas (Components / Screens / Services / Context / Types)

La aplicación se organizó en capas claramente diferenciadas:

- Components: componentes atómicos y reutilizables; su propósito es aislar la presentación y mantenerlos independientes de la lógica.
- Screens: Pantallas que tienen el rol de contenedores de flujo que orquestan varios componentes y traducen casos de uso en interfaz.
- Services: capa de acceso a datos que encapsula todas las interacciones con Firebase; su objetivo es proteger al resto de la aplicación del detalle de la persistencia y facilitar pruebas y cambios futuros.
- Context: proveedor global para estado compartido (usuario, flags de render, configuraciones), elegido para reducir el acoplamiento entre pantallas y para exponer acciones globales sin códigos excesivo.
- Types: definición de contratos mediante TypeScript para garantizar coherencia entre UI y backend.

Esta separación favorece mantenibilidad, pruebas unitarias y la posibilidad de sustituir o adaptar una capa sin modificar las demás.

3.1.4. Justificación de los tipos y colecciones empleadas

Motivación general de los tipos definidos en `src/Types/Types.tsx`

Los tipos incluidos en `Types.tsx` responden a un modelado orientado al dominio y a las restricciones/ventajas técnicas de la plataforma: se ha buscado definir un conjunto mínimo y suficiente de entidades que represente recetas, ingredientes, planificación y despensa sin introducir campos superfluos que compliquen consultas o índices en Firestore. Las decisiones clave fueron: usar identificadores como `string` por compatibilidad con Firestore; cantidades como `number` para permitir operaciones aritméticas; enums para impedir valores erróneos y facilitar autocompletado; y campos opcionales para mantener compatibilidad con documentos antiguos y permitir evolución. Se priorizó un equilibrio entre normalización (referencias a `ingredientId` para evitar duplicación) y desnormalización mínima cuando mejora el rendimiento de lectura en pantallas críticas. Por alcance y coste de desarrollo se dejaron fuera atributos secundarios (p. ej. perfiles nutricionales extensos o datos de caducidad) que pueden añadirse en iteraciones posteriores sin romper el esquema básico. Esta selección facilita consultas eficientes, soporte offline y una evolución controlada del modelo de datos.

Tipos definidos

Ingredient (Ingrediente)
Es el componente principal del programa de menús dado que cada receta está compuesta de ingredientes. En la base de datos se ha creado la colección Ingredientes que dispone de los documentos.

```
export type Ingredient = {
 id: string;
 name: string;
 category: string;
};
```

- *id:* el identificador del ingrediente.
- *name:* contiene el nombre del ingrediente. Es de tipo string.
- *category:* la categoría del ingrediente (Lácteo, fruta, verdura, ...), dado que es un texto, su tipo será string

Recipe (Receta)

```
export type Recipe = {
 id: string;
 name: string;
 userId: string;
 link?: string;
 preparationTime?: number;
 servingSize: number;
 image?: string;
};
```

Se diseñó priorizando:

- `id` como string para compatibilidad con Firestore y facilitar referencias.
- `userId` para aislamiento de datos entre usuarios (seguridad).
- `link` y `preparationTime` opcionales porque no todas las recetas los necesitan.
- `servingSize` obligatorio para cálculos precisos de cantidades.
- `image` opcional como URL, evitando almacenar binarios en documento.

Alternativas consideradas:

- Incluir ingredientes embebidos: descartado por dificultar actualizaciones y aumentar duplicación.
- Añadir campos nutricionales: reservado para futuras iteraciones.
- Usar números para IDs: preferido string por compatibilidad Firebase.

RecipeIngredient

```
export type RecipeIngredient = {
 id: string;
 recipeId: string;
 ingredientId: string;
 quantity: number;
 quantityType: QuantityType;
};
```

Decisiones clave:

- Tipo unión entre Recipe e Ingredient para mantener normalización.
- `quantity` como number para permitir decimales en medidas.

- `quantityType` como enum para validación y UI consistente.

Alternativas evaluadas:

- Embeber información completa del ingrediente: aumentaría duplicación.
- Cantidades como strings: complicaría cálculos automáticos.

WeeklyMeal

```
export type WeeklyMeal = {
 id: string;
 day: DaysOfWeek;
 mealType: MealType;
 userId: string;
 entryType?: WeeklyEntryType;
 recipeId?: string;
 ingredientId?: string;
 quantity?: number;
 quantityType?: QuantityType;
 createdAt?: number | Date;
};
```

Se diseñó para:

- Soportar tanto recetas completas como ingredientes sueltos (`entryType`).
- Mantener flexibilidad en planificación (campos opcionales).
- Facilitar consultas por día/comida sin joins complejos.

Otras opciones consideradas:

- Separar en dos colecciones (recetas/ingredientes): aumentaría complejidad de queries.
- Incluir datos completos: preferida referencia para consistencia.

IngredientPantry

```
export type IngredientPantry = {
 id: string;
 ingredientId: string;
 quantity: number;
 quantityType: QuantityType;
};
```

Diseño enfocado a:

- Seguimiento preciso de inventario por usuario.
- Soporte para diferentes unidades de medida.
- Facilitar cálculos de lista de compra.

Alternativas:

- Modelo más complejo con fechas caducidad: reservado para futuro.
- Cantidades como rangos: innecesario para v1.

GroceryBought.
Colección que guarda los ingredientes ya comprados de la lista de la compra. Este dato es guardado cuando el usuario compra un ingrediente de la lista.

```
export type GroceryBought = {
 id: string;
 ingredientId: string; // ID of the ingredient that was bought
 timestamp: number; // Timestamp when the ingredient was bought
};
```

- *id:* identificador.
- *ingredientId:* ingrediente ya comprado.
- *timestamp:* tiempo de compra.

Enumeraciones La decisión de usar enums en lugar de strings libres o constantes globales es la prevención de errores humanos, es frecuente que el programador confunda valores que resultan ser ambiguos como el caso de gramos que su abreviatura podría ser "gr"tanto como "g", por ello se han creado los siguientes enums para que a la hora de programar se usen las variables globales. También al estar todo centralizado, facilita la futura actualización o mejora de estos valores.

- **QuantityType.** Define los posibles tipos de unidades de medida para los ingredientes.

```
export enum QuantityType {
 UNIT = 'unit',
 GRAM = 'gram',
 KILOGRAM = 'kilogram',
 MILLILITER = 'milliliter',
 LITER = 'liter',
 CUP = 'cup',
```

```
 TABLESPOON = 'tablespoon',
 TEASPOON = 'teaspoon',
}
```

- **WeeklyEntryType.** Define si el *WeeklyMeal* guardado es un ingrediente o una receta como es comentado en la sección **??**.

```
export enum WeeklyEntryType {
 RECIPE = 'RECIPE',
 INGREDIENT = 'INGREDIENT',
}
```

- **MealType.** Tipo de comida a la que se asigna una receta.

```
export enum MealType {
 BREAKFAST = 'Breakfast',
 LUNCH = 'Lunch',
 DINNER = 'Dinner',
}
```

- **DaysOfWeek.** Días de la semana

```
export enum DaysOfWeek {
 MONDAY = 'Mon',
 TUESDAY = 'Tue',
 WEDNESDAY = 'Wed',
 THURSDAY = 'Thu',
 FRIDAY = 'Fri',
 SATURDAY = 'Sat',
 SUNDAY = 'Sun',
}
```

Decisiones generales sobre tipos

- Uso de TypeScript para garantizar tipo seguro.
- IDs como strings por compatibilidad Firestore.
- Referencias vs embedding según caso de uso.
- Campos opcionales (?) solo cuando realmente necesario.

- Tipos sin ID (`WithoutId`) para creación de documentos.

Esta estructura de tipos permite un balance entre normalización (para consistencia) y desnormalización (para rendimiento), mientras mantiene la flexibilidad necesaria para evolucionar la aplicación según feedback de usuarios.

3.1.5. Diseño y lógica de las páginas

Creación de páginas

Como en cada aplicación, hay que crear páginas para mostrar toda la información necesaria al usuario, pero sin sobrecargarlas de información y al mismo tiempo satisfaciendo los casos de uso. En nuestro caso, se han tomado varios factores en la decisión del diseño de las páginas. Como en todas las aplicaciones, se tiene que crear una página de inicio de sesión y registro para permitir el acceso al contenido específico de cada usuario, seguido de ello se tiene que crear una página principal, que es considerada el punto intermedio de todas las páginas, es decir, desde ella se puede acceder a todas las páginas de la aplicación.

Pagina de inicio (LoginScreen) En la página de inicio se ha decidido introducir lo esencial para satisfacer el caso de uso de "*inicio de sesión*". Esta se trata de:

- Un título con el texto "*Login*"que en inglés significa inicio de sesión para poner al usuario en contexto e indicarle la funcionalidad de la página.
- Dos campos de texto situados uno encima del otro y que el primero capta el correo y el segundo la contraseña. Se han insterado textos y marcadores de posición encima de los campos para mostrar qué contenido es introducido en cada uno.
- Debajo de los campos de texto se encuentra un botón de inicio de sesión.
- Como último componente se encuentra un texto que al pulsarlo redirige a la página de registro.
- Una alerta mostrando un texto indicando que los campos deben estar rellenados antes de proceder al inicio de sesión. Esta sólo es visible cuando el usuario no rellena un campo.
- Otra alerta del mismo estilo solo que esta es visible cuando falla el inicio de sesión por cualquier motivo.

Lógica de implementación

La página utiliza varios conceptos clave de React y React Native:

- **Estado local:** Mediante `useState` se gestionan:

- `email`: almacena el correo introducido
- `password`: guarda la contraseña

- **Manejo de autenticación:** La función `handleLogin` async:
  - Valida que ambos campos estén completos
  - Limpia el email con `trim()` para evitar errores de espacios
  - Utiliza `auth().signInWithEmailAndPassword()` de Firebase
  - Captura y muestra errores mediante el sistema de alertas

- **Adaptación a plataforma:** Usa `KeyboardAvoidingView` con comportamiento específico por plataforma:
  - iOS: modo 'padding' para ajustar la vista cuando aparece el teclado
  - Android: comportamiento por defecto

Estilos y diseño visual

Los estilos se definen mediante `StyleSheet.create()` y siguen principios de diseño consistentes:

- **Layout:** Uso de `flex: 1` y centrado mediante `justifyContent` y `alignItems`.
- **Inputs:** Bordes suaves (`borderRadius: 5`) y padding consistente.
- **Botón:** Destacado visual mediante color naranja y padding vertical.
- **Tipografía:** Tamaños y pesos de fuente definidos para jerarquía clara.

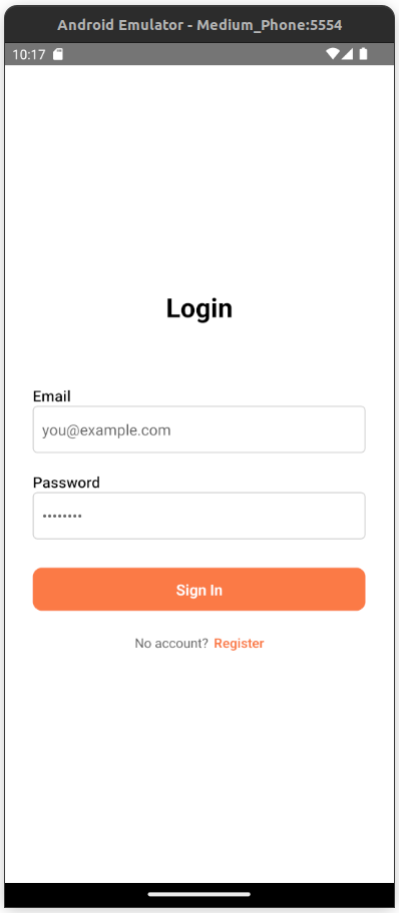


Figura 3.1: Página de inicio de sesión LoginScreen

Página de registro (RegisterScreen) En el caso de esta página, su función es crear un usuario no existente y para realizar ello, el sistema necesita un correo electrónico y contraseña del usuario. El diseño de la página es el siguiente:

- Un título mostrando al usuario su posición actual.
- Dos campos de texto similares al de la página de inicio de sesión pero en este caso recogen los datos de *email* (correo electrónico) y *password* (contraseña).
- Un botón como último componente y que contiene el texto de registrarse"(*"sign up"* en inglés).

Lógica de implementación

La página de registro utiliza una estructura similar a LoginScreen pero orientada a la creación de cuenta:

- **Estado local:** Se gestionan dos estados mediante `useState`:
  - `email`: almacena el correo del nuevo usuario
  - `password`: guarda la contraseña elegida
- **Gestión del registro:** La función `handleRegister` async implementa:
  - Validación de campos completos antes de proceder
  - Limpieza del email con `trim()`
  - Llamada a Firebase Auth mediante `createUserWithEmailAndPassword()`
  - Manejo de errores con alertas del sistema
- **Adaptación multiplataforma:** Al igual que en login, se usa `KeyboardAvoidingView` para:
  - Ajustar la vista cuando aparece el teclado en iOS
  - Mantener comportamiento nativo en Android

Estilos y diseño visual

Los estilos se mantienen consistentes con LoginScreen mediante `StyleSheet.create()`:

- **Contenedor:** Usa `flex: 1` con fondo `screensBackgroundColor` y padding uniforme
- **Formulario:** Ancho del 90 % y margen superior para separación visual
- **Campos de entrada:** Mantienen el diseño con bordes suaves y espaciado consistente
- **Botón de registro:** Destaca visualmente con el color naranja corporativo y tipografía clara

Esta implementación sigue el principio minimalista mencionado anteriormente, manteniendo solo los elementos esenciales para el registro mientras se asegura una experiencia de usuario fluida y coherente con el resto de la aplicación.

En estas dos páginas (*LoginScreen e RegisterScreen*) era posible introducir más componentes, pero dado el principio del minimalismo se decidió mantenerse al menor número de componentes posibles para no sobrecargar las páginas de manera visual y en rendimiento. Por otra parte, no era necesario añadir algo adicional dado que no se requieren datos adicionales.

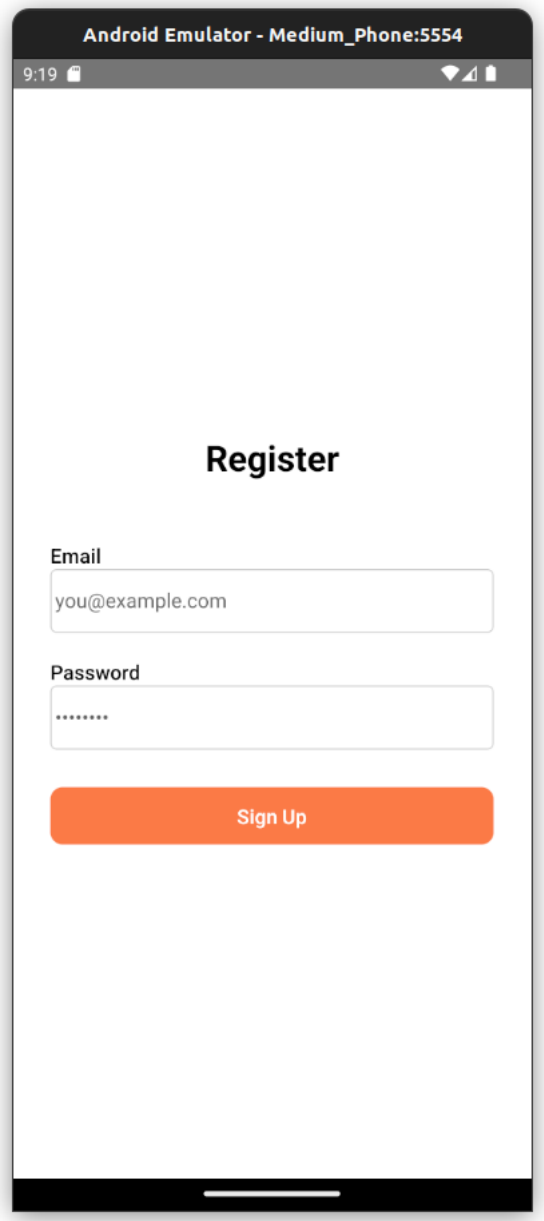


Figura 3.2: Página de registro (RegisterScreen)

Página principal (MainScreen) En la planificación del diseño se decidió mostrar el calendario semanal de las recetas planificadas para cada día, pero no era posible mostrar todas las comidas planificadas para toda la semana en una sola pantalla móvil y tampoco era muy minimalista mostrar toda esa información en una sola página; por ello se crearon dos componentes de selección, el primero permite al usuario elegir el día de la semana y el segundo el tipo de comida (desayuno, comida o cena), esto hace que sólo se muestre la lista de una comida de un día específico, simplifica y evita la sobrecarga del diseño de la app. La página contiene los componentes en este orden:

- Una cabecera que en su izquierda contiene el texto "*Daily Meals*"(comidas diarias) y a su izquierda dos botones uno cierra la sesión y el situado a su derecha abre la página de recetas (RecipesScreen).
- Un menú deslizable horizontalmente que permite al usuario seleccionar el día deseado y que muestra el día seleccionado en naranja.
- Otro menú similar al anterior, pero en este caso no es deslizable y sólo tiene la opción de escoger entre desayuno, comida y cena.
- Una vista deslizable verticalmente que muestra todos las recetas e ingredientes planificados para el día y comida seleccionados anteriormente.
- En la zona inferior derecha hay cuatro botones con diferentes iconos, cada icono muestra la función del botón y estas son.
  - Ir a lista de la compra (*GroceryListScreen*).
  - Ir a la despensa (*PantryScreen*).
  - Abrir el menú responsable de planificar recetas.
  - Acceder al menú para añadir nuevas recetas.

Lógica de implementación

La página principal implementa una lógica más compleja que las anteriores debido a su naturaleza central en la aplicación.

Esta página tienen varias funciones y procesos que se ejecutan dependiendo de la etapa. Al acceder a *MainScreen* por primera vez (después de haber iniciado sesión) el sistema busca y recoge los ingredientes, recetas y *WeeklyMeals* relevantes para el uso de la aplicación y de mientras muestra un icono de carga. Una vez obtenida la información relevante, se guarda en las variables de *state* de React Native (para permitir futuras actualizaciones) y se muestran en la lista de MainScreen las recetas planificadas del día y el tipo de comida especificado. En la zona inferior derecha se encuentran botones de acceso creados para abrir modales para crear o planificar una receta y acceder a las páginas de despensa (*PantryScreen*) y lista de compra (*GroceryListScreen*).

Variables y funciones utilizadas

- **Estado local:** Gestiona múltiples estados mediante `useState`:
  - `selectedMeal`: controla el tipo de comida seleccionada
  - `selectedDay`: mantiene el día de la semana actual
  - `weeklyMeals`: almacena las comidas planificadas
  - `currentWeeklyMealsRecipes`: guarda las recetas actuales
  - `currentWeeklyMealsIngredients`: mantiene los ingredientes planificados
- **Gestión de datos:** Implementa funciones asíncronas para:
  - `fetchWeeklyMeals`: obtiene las comidas planificadas (*WeeklyMeals*) en el día y comida elegidos en los menús superiores.
  - `fetchRecipes`: recupera las recetas completas.
  - `handlePlanRecipe`: gestiona la planificación de recetas
  - `handleUnplanRecipe`: elimina recetas del calendario
- **Efectos y ciclo de vida:** Utiliza `useEffect` para:
  - Cargar datos iniciales al montar el componente
  - Actualizar las comidas cuando cambia el día o tipo de comida
  - Sincronizar las recetas con las comidas planificadas

Estilos y diseño visual

Los estilos se definen mediante `StyleSheet.create()` siguiendo un diseño consistente:

- **Contenedor principal:** Utiliza `flex: 1` con fondo definido en `screensBackgroundColor`
- **Botones flotantes:** Posicionamiento absoluto con sombras y elevación para efecto visual
- **ScrollView:** Implementado para el desplazamiento vertical suave de las recetas
- **Indicadores de carga:** Centrados y con colores corporativos

Esta implementación sigue un enfoque modular que permite una gestión eficiente de los datos y una experiencia de usuario fluida, manteniendo la consistencia visual con el resto de la aplicación. La separación de responsabilidades entre componentes facilita el mantenimiento y la escalabilidad del código.

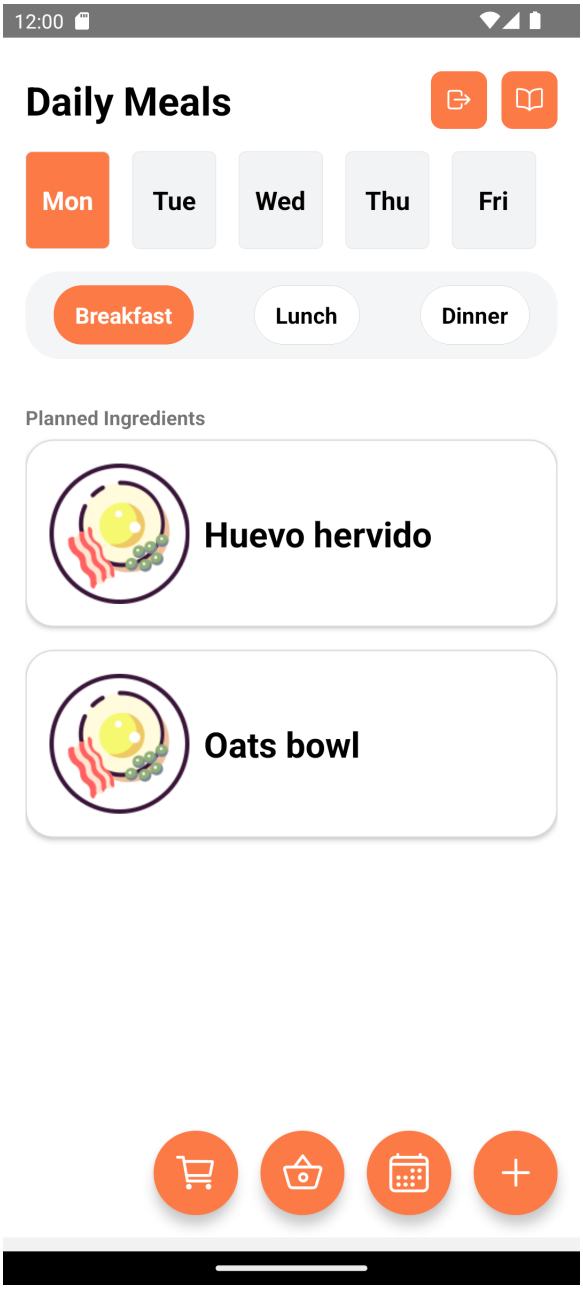


Figura 3.3: Página principal (MainScreen)

Página de recetas (RecipesScreen) En esta página se muestran todas las recetas creadas por el usuario. Para mantener el minimalismo y la simplicidad, se ha diseñado de

la siguiente manera:

- Una cabecera con el texto *Recipes* (Recetas) y un botón de retorno a la página principal.
- Una lista deslizable verticalmente que contiene todas las recetas del usuario, cada una representada por un `RecipeCardComponent` que muestra:
  - El título de la receta.
  - Una imagen si esta existe.
- Un botón flotante en la esquina inferior derecha que permite añadir una nueva receta.

Cada tarjeta de receta permite interacción mediante pulsación prolongada, mostrando un menú de opciones para editar o eliminar la receta.

Lógica de implementación

La página de recetas implementa una lógica centrada en la visualización y edición de recetas individuales:

- **Gestión de estado global:** Utiliza el contexto de la aplicación (`useAppContext`) para acceder y modificar la lista de recetas, permitiendo que los cambios se reflejen en toda la app de manera automática.
- **Navegación:** Emplea el sistema de navegación de React Native para permitir al usuario acceder a la pantalla de detalle de una receta al pulsar sobre una tarjeta, pasando la información relevante como parámetro.
- **Renderizado eficiente:** La lista de recetas se muestra mediante un `ScrollView`, asegurando un desplazamiento fluido y una experiencia visual agradable incluso con muchas recetas.
- **Interacción con tarjetas:** Cada `RecipeCardComponent` incluye eventos de pulsación para navegar, y de pulsación prolongada para mostrar opciones adicionales (editar/eliminar), manteniendo la interfaz limpia y funcional.
- **Actualización reactiva:** Al añadir, editar o eliminar una receta, el estado global se actualiza y la lista se re-renderiza automáticamente, garantizando que el usuario siempre vea la información más actual.

Estilos y diseño visual

Los estilos siguen los principios de consistencia y minimalismo presentes en el resto de la aplicación:

- **Contenedor principal:** Fondo definido por `screensBackgroundColor` y padding uniforme para mantener la coherencia visual.

- **Cabecera:** Espaciado inferior para separar visualmente la cabecera de la lista de recetas.

- **Tarjetas de receta:** Bordes suaves, sombras ligeras y separación entre tarjetas para facilitar la lectura y la interacción.

- **Botón flotante:** Posicionado en la esquina inferior derecha, con color destacado y sombra para resaltar la acción principal de añadir receta.

Esta página mantiene la filosofía de diseño minimalista, mostrando únicamente la información relevante y permitiendo al usuario gestionar sus recetas de forma intuitiva y eficiente.

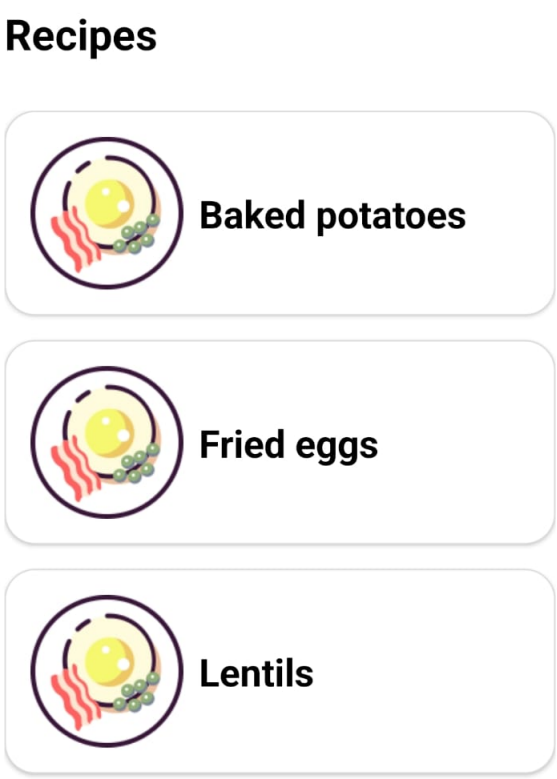


Figura 3.4: Página de recetas (RecipesScreen)

Página de receta individual (RecipeScreen)
Esta pantalla muestra el detalle completo de una receta específica y permite su edición.

Se ha estructurado de la siguiente forma:

- Una cabecera con el título de la receta y un botón de retorno.
- La imagen de la receta si existe, con un botón para añadir/cambiar imagen.
- Una sección de descripción que muestra el detalle de la receta.
- Una lista de ingredientes donde cada uno muestra:
  - Nombre del ingrediente
  - Cantidad y unidad de medida
  - Botón para eliminar el ingrediente
- Un botón flotante para añadir nuevos ingredientes a la receta.

Lógica de implementación

La página de receta individual implementa una lógica orientada a la visualización, edición y gestión de los ingredientes de una receta:

- **Gestión de estado local y global:** Utiliza `useState` para controlar el modo de edición, los cambios realizados y la visibilidad de los modales. El acceso al contexto global (`useAppContext`) permite modificar la receta y sus ingredientes en toda la aplicación.
- **Edición de receta:** El usuario puede alternar entre modo visualización y edición. En modo edición, los campos de la receta (nombre, enlace, tiempo de preparación, raciones) se muestran como inputs editables. Al guardar, se valida la información y se actualiza el estado global.
- **Gestión de ingredientes:** Los ingredientes se obtienen dinámicamente y se muestran en una lista. En modo edición, cada ingrediente puede modificarse (cantidad, unidad) o eliminarse mediante botones específicos. La adición de ingredientes se realiza a través de un modal de búsqueda y selección.
- **Sincronización y actualización:** Los cambios en los ingredientes o en la receta se sincronizan automáticamente con la base de datos y el estado global, garantizando que la información esté siempre actualizada.
- **Interfaz reactiva:** El diseño permite alternar entre visualización y edición de forma intuitiva, mostrando los botones y campos adecuados según el estado actual.

Estilos y diseño visual

Los estilos mantienen la coherencia con el resto de la aplicación y favorecen la claridad y la usabilidad:

- **Contenedor principal:** Fondo definido por `screensBackgroundColor` y padding uniforme.

- **Cabecera:** Espaciado inferior y tipografía destacada para el título.

- **Imagen de receta:** Bordes redondeados y tamaño fijo para mantener la estética.

- **Campos de edición:** Inputs con bordes suaves y espaciado consistente.

- **Lista de ingredientes:** Separación clara entre ingredientes, con botones de acción visibles en modo edición.

- **Botón flotante:** Posicionado en la esquina inferior derecha, con color destacado y sombra para resaltar la acción principal de añadir ingrediente.

- **Botón de guardar/editar:** Cambia de icono y texto según el estado, facilitando la interacción.

Esta pantalla permite al usuario consultar y modificar todos los detalles de una receta de forma sencilla y eficiente, manteniendo la filosofía de diseño minimalista y funcional de la aplicación.

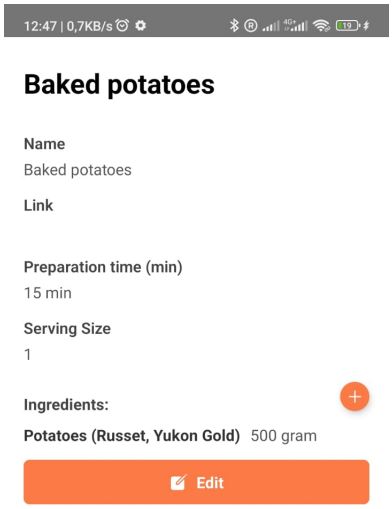


Figura 3.5: Página de receta individual (RecipeScreen)

Página de despensa (PantryScreen) La despensa es una parte crucial de la aplicación ya que permite al usuario mantener un registro de los ingredientes disponibles. Su diseño se compone de:

- Una cabecera con el texto *Pantry* (Despensa) y botón de retorno.
- Una lista deslizable de ingredientes disponibles, donde cada `IngredientCard` muestra:
  - Nombre del ingrediente
  - Cantidad disponible y unidad
  - Opción para editar la cantidad
  - Opción para eliminar el ingrediente de la despensa
- Un botón flotante para añadir nuevos ingredientes a la despensa mediante el `IngredientSearchModal`.

Diseño de la lógica La página de la despensa es la encargada de mostrar y permitir al usuario introducir los ingredientes en su despensa. Al abrirse la pantalla, esta muestra el icono de carga para indicar que la app está en proceso de carga y al mismo tiempo pide a la base de datos todos los *PantryIngredients* que este usuario dispone para mostrarlos mediante la función `getAllIngredientPantries`. Una vez obtenidos se muestran en una lista (ver Figura 3.6) que permite al usuario ver, cambiar los datos (cantidad, medida) o eliminar el ingrediente. Cada vez que el usuario edita añade o elimina un ingrediente la página se encarga de actualizarlo en la base de datos de Firebase.

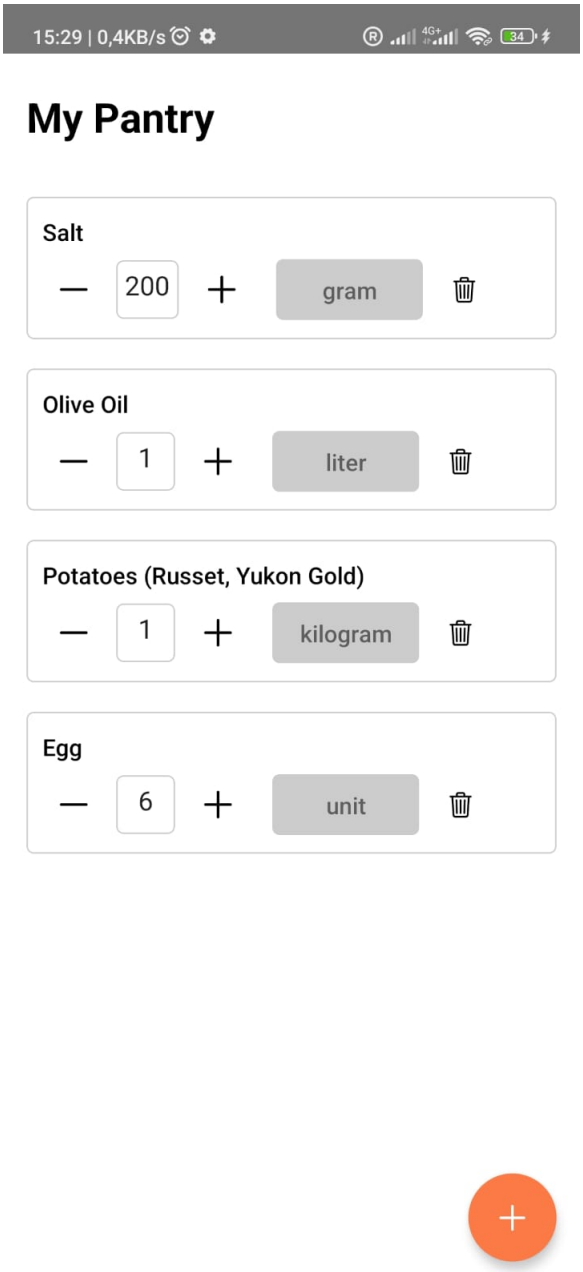


Figura 3.6: Página de la despensa (PantryScreen)

Página de lista de la compra (GroceryListScreen) Esta página muestra los ingredientes necesarios para las recetas planificadas, calculando automáticamente las cantidades

requeridas después de considerar lo disponible en la despensa. Su estructura es:

- Una cabecera con el título Grocery List (*Lista de la compra*), un botón de retorno y un botón de actualización que recalcula la lista.
- Un texto que muestra la última hora de actualización de la lista.
- Una lista deslizable de ingredientes necesarios, donde cada elemento muestra:
  - Nombre del ingrediente
  - Cantidad a comprar y unidad
  - Un checkbox para marcar como comprado
- Una sección colapsable que muestra los ingredientes ya marcados como comprados, permitiendo desmarcarlos si es necesario.
- Un mensaje indicando que no hay nada que comprar cuando la lista está vacía.

Diseño de la lógica

Esta página implementa uno de los algoritmos más complejos de la aplicación, ya que debe calcular precisamente qué ingredientes necesita comprar el usuario. El proceso se realiza en varias etapas:

Primero, el sistema recopila todas las comidas planificadas para la semana y las procesa según su tipo. Si es una receta, obtiene todos sus ingredientes; si es un ingrediente individual planificado, lo añade directamente. Durante este proceso, el sistema agrega las cantidades de ingredientes idénticos que comparten la misma unidad de medida. Por ejemplo, si una receta requiere 300g de arroz y otra necesita 200g, estos se combinan en una única entrada de 500g.

Una vez agregados todos los ingredientes necesarios, el sistema consulta el inventario de la despensa del usuario. Para cada ingrediente en la lista de compra, verifica si existe en la despensa y, en caso afirmativo, resta la cantidad disponible de la cantidad necesaria. Por ejemplo, si se necesitan 500g de arroz pero hay 200g en la despensa, la lista de compra mostrará que se deben comprar 300g.

La lista resultante se ordena alfabéticamente y se divide en dos secciones: ingredientes pendientes de comprar e ingredientes ya comprados. Cuando un usuario marca un ingrediente como comprado, este se mueve automáticamente a la sección de comprados y se guarda este estado en la base de datos, permitiendo que la información persista entre sesiones.

El sistema también implementa una actualización en tiempo real: cada vez que se marca o desmarca un ingrediente, o cuando se pulsa el botón de actualizar, se recalcula toda la lista, asegurando que los datos mostrados están siempre sincronizados con el estado actual de la despensa y las comidas planificadas.

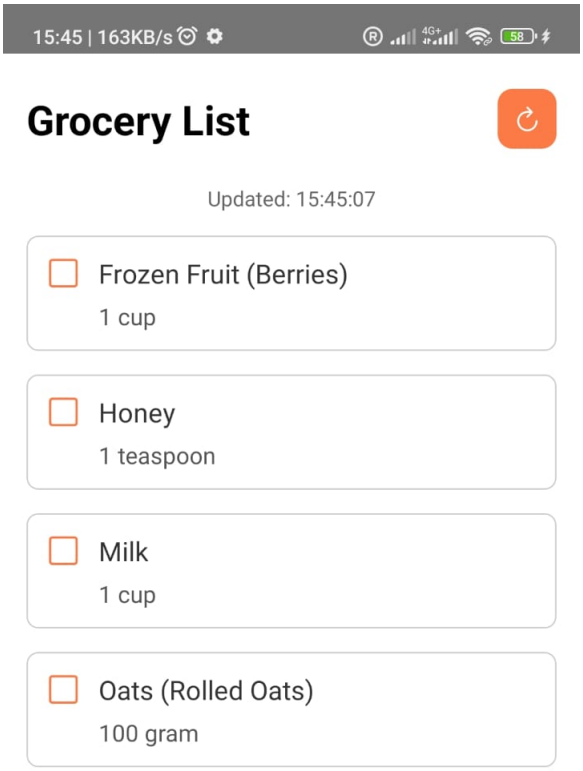


Figura 3.7: Página de la lista de la compra (*GroceryListScreen*)

Esta implementación mantiene el principio de diseño minimalista, mostrando solo la información esencial y manteniendo una jerarquía visual clara. La navegación es intuitiva

gracias a los botones de retorno consistentes y el feedback visual inmediato para las acciones del usuario, como el cambio de estado al marcar ingredientes como comprados.

Flujo de páginas y navegación

Nuestra aplicación comienza por la página de inicio de sesión (LoginScreen en el caso de no haber iniciado sesión previamente) que tiene acceso a la página de registro (RegisterScreen) en la que el usuario puede crear una cuenta nueva. Una vez registrado o iniciado sesión, se pasa directamente a la página principal(MainScreen) que desde ella se puede acceder a todo el contenido directa e indirectamente, en esta también se muestra la información más relevante y se tiene acceso a RecipesScreen, PantryScreen y GroceryListScreen; se tiene que saber que todas las páginas a las que se accede desde la página principal tienen permitido volver a esta. Al mismo tiempo RecipesScreen que es la página en donde se muestran todas las recetas creadas por el usuario; tiene acceso a RecipeScreen y esta tiene acceso de vuelta a la página de recetas.

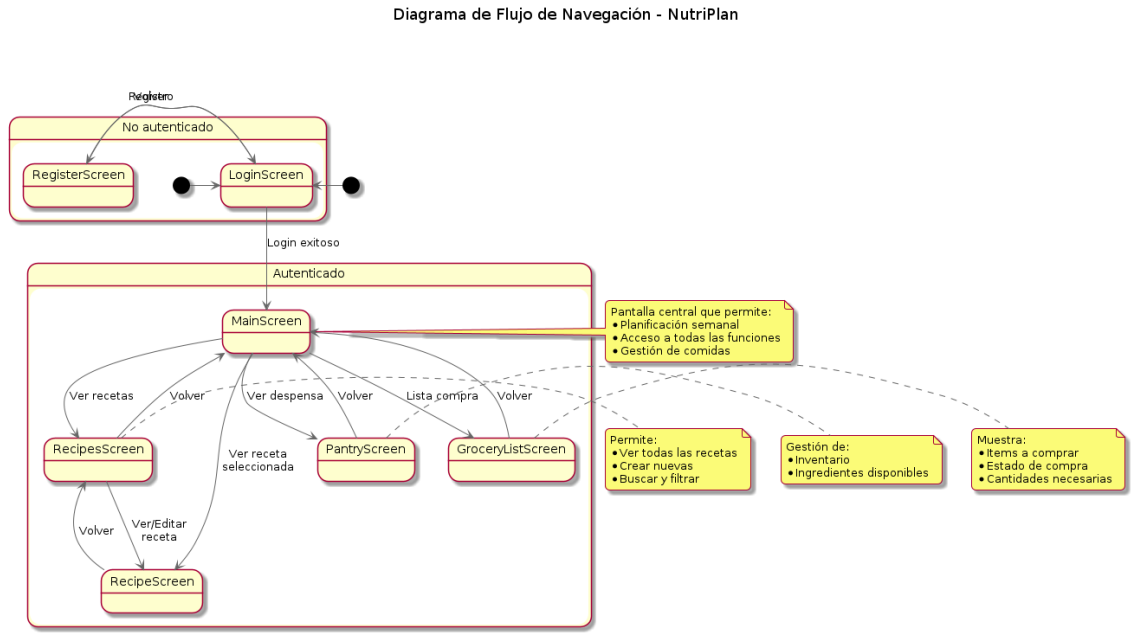


Figura 3.8: Diagrama de flujo de navegación de NutriPlan

Organización y estructura visual La interfaz se organizó priorizando la jerarquía visual y la facilidad de navegación:

- **Barra superior consistente:** Implementada en `AppHeader`, mantiene el contexto de navegación mostrando títulos en cada sección y provee acceso rápido a funciones

y páginas principales. Esta decisión responde al principio de consistencia y reduce la carga cognitiva del usuario al mantener puntos de referencia estables.

- **Botones flotantes de acción:** El componente `FloatingButton` se utiliza para acciones principales (añadir receta, planificar comida) siguiendo patrones de Material Design. Su posición fija en la esquina inferior derecha facilita el acceso con el pulgar, mejorando la ergonomía en uso móvil.
- **Tarjetas y listas:** Los componentes `IngredientCard` y `RecipeCardComponent` presentan información en bloques visuales claramente delimitados, facilitando el escaneo rápido y la comprensión del contenido.

Paleta de colores La selección de colores responde tanto a principios de accesibilidad como a la psicología del color en aplicaciones de alimentación:

- **Color primario:** Se eligieron como colores principales el blanco (#FFF) y el naranja (#FB7945). El blanco se utiliza principalmente para fondos de pantallas y de componentes; y los colores de las letras. El naranja se utiliza como color de fondo de los componentes(mayoritariamente botones) y también como color para resaltar elementos seleccionados.
- **Colores secundarios:** estos también son utilizados frecuentemente y son el gris claro (#CCC) utilizado para separación sutil entre elementos, color de letras o como fondo de pantalla de otros. Los colores negro y blanco han sido usados para la letra.

Esta paleta no solo es estéticamente agradable sino que cumple con ratios de contraste WCAG 2.1 para garantizar accesibilidad.

Interacción y feedback El diseño de interacciones prioriza la eficiencia y claridad:

- **Gestos intuitivos:** Se implementaron gestos naturales como el mantener pulsado en un el `RecipeCard` para mostrar el modal de opciones
- **Feedback visual inmediato:** Cada acción (marcar ingrediente como comprado, añadir a la planificación) tiene una respuesta visual clara que confirma la operación.

Tipografía y legibilidad Se utilizó una jerarquía tipográfica clara:

- **Títulos:** Fuente System Bold a 20pt para encabezados principales, garantizando visibilidad sin ser intrusiva.
- **Contenido:** System Regular a 16pt para el cuerpo, optimizando legibilidad en pantallas móviles.
- **Información secundaria:** System Light a 14pt para metadatos y detalles adicionales.

Adaptación a casos de uso El diseño se alineó específicamente con los casos de uso principales:

- **Planificación semanal:** La vista principal (`MainScreen`) organiza la información por días y comidas, con un selector superior (`HeaderComponent`) que facilita la navegación temporal.
- **Gestión de recetas:** La interfaz de recetas prioriza imágenes y datos clave, permitiendo acceso rápido a ingredientes y cantidades mediante `IngredientComponent`.
- **Lista de compra:** `GroceryListScreen` implementa una visualización clara de items pendientes vs. comprados, con actualización inmediata del estado.
- **Control de despensa:** `PantryScreen` organiza ingredientes en categorías visuales claras, facilitando el inventario rápido.

Consideraciones de accesibilidad El diseño incorpora elementos de accesibilidad básicos:

- Textos alternativos para imágenes
- Áreas táctiles generosas (mínimo 44x44 puntos)
- Contraste suficiente entre texto y fondo
- Soporte para el escalado de texto del sistema

Iconografía y motivos de elección La iconografía de la aplicación se seleccionó con dos objetivos claros: maximizar la legibilidad y la affordance (que el icono sugiera su acción) y mantener coherencia visual en toda la interfaz. A continuación se recogen los iconos relevantes que aparecen en la aplicación (según la estructura de componentes en `src/Components`) y la justificación de cada elección.

Librería usada La aplicación emplea una librería de iconos vectoriales (por ejemplo `react-native-vector-icons` con familias como Ionicons o MaterialCommunityIcons). Esta decisión se fundamenta en su:

- amplia colección de glifos adecuados para interfaces móviles,
- escalabilidad y nitidez en pantallas de distinta densidad (vectorial),
- facilidad de uso e integración con componentes y temas (color, tamaño, estilos).

Iconos principales y motivos

- Añadir / *Plus* (FloatingButton, AddRecipe, AddIngredient). Se usa el icono de "más" por ser el estándar universal para crear nuevos elementos. Está ubicado en botones flotantes para accesibilidad con el pulgar y destaca con el color primario para indicar la acción principal.
- Buscar / *Search* (IngredientSearchSelector, IngredientSearchModal). El icono de lupa comunica rápidamente la funcionalidad de búsqueda. Al incluir búsqueda local (MiniSearch) es crítico que el control sea reconocible e inmediato.
- Volver / Flecha atrás (HeaderComponent, navegación). La flecha dirige la navegación y mantiene la metáfora de pila de pantallas; su colocación consistente en la cabecera reduce la curva de aprendizaje.
- Menú / *Hamburger* o Más opciones (AppHeader, RecipeOptionsModal). Se utiliza un icono de "más/menú" para agrupar acciones contextuales sin saturar la interfaz principal.
- Editar / *Pencil* (RecipeScreen, ediciones rápidas). El lápiz es la convención para edición; su uso en modales y tarjetas sugiere edición in-place en lugar de navegación completa a otra pantalla.
- Eliminar / *Trash* (opciones de receta, lista de compra). El cubo de basura es inequívoco para borrar elementos; se presenta junto al color de alerta para minimizar acciones accidentales.
- Calendario / Planificación (MainScreen, MealsHeader). Un icono de calendario o planificación facilita asociar la acción a la gestión temporal (días/turnos) y refuerza la función de plan semanal.
- Carrito de la compra / Grocery (GroceryListScreen). Icono que remite a compra y lista de la compra; su uso ayuda a distinguir visualmente la vista de compras del inventario.
- Check / Marcar como comprado (GroceryListScreen, PantryScreen). La marca de verificación indica estado completado/comprado; su reversibilidad en la UI es clave para feedback inmediato.
- Cámara / Imagen (AddRecipeModal). El icono de cámara sugiere subir o tomar fotos de recetas; refuerza la posibilidad de asociar recursos multimedia a recetas.
- Usuario / Perfil (Login/Register). El icono de avatar sintetiza las acciones de acceso y perfil; facilita la identificación del flujo de autenticación.

Validación y mejoras futuras

El diseño actual establece una base sólida pero contempla evolución:

- Incorporación de temas oscuros para reducir fatiga visual
- Expansión de gestos y atajos para usuarios avanzados
- Mejora de la visualización de datos nutricionales
- Implementación de vistas adaptativas para tablets

Las decisiones de diseño tomadas buscan equilibrar simplicidad, eficiencia y satisfacción del usuario, creando una experiencia coherente que facilita la planificación alimentaria y reduce la fricción en tareas cotidianas. La adherencia a principios minimalistas y patrones establecidos de UX móvil contribuye a una curva de aprendizaje suave mientras se mantiene la potencia funcional necesaria.

3.1.6. Componentes de la aplicación

Componentes de la interfaz

FloatingButton

Componente `FloatingButton`

Floating Button es un botón flotante — componente redondo con sombra— diseñado para acciones primarias en pantalla. El componente ofrece una apariencia de flotación mediante elevación y sombra, y está implementado en `FloatingButton.tsx` como una función que devuelve el elemento táctil con un icono centrado.

Descripción funcional

- Botón circular con icono (Ionicons) centrado, pensado para acciones de alto nivel en la interfaz (por ejemplo: añadir, crear, o abrir un modal).
- Su posicionamiento por defecto está en la esquina inferior derecha de la pantalla, pero admite estilos personalizados para adaptar posición y apariencia.
- La sombra y la elevación proporcionan el efecto visual de estar "flotando" sobre el contenido.

Parámetros (props) El componente acepta los siguientes parámetros:

- `onPress: () =>void` — función que se ejecuta cuando el botón es pulsado.
- `iconName: IoniconName` — nombre del icono a mostrar (válido dentro de las opciones de Ionicons).

- `iconColor?: string` — color del icono (opcional; por defecto `'white'`).
- `iconSize?: number` — tamaño del icono en dp (opcional; por defecto `28`).
- `containerStyle?: ViewStyle` — estilos adicionales para el contenedor del botón (opcional; permite sobrescribir posición, tamaño, color, etc.).
- `iconContainerStyle?: ViewStyle` — estilos adicionales aplicables al icono (opcional).

Contrato y comportamiento

- Entrada: las props indicadas y la interacción del usuario (tap).
- Salida: invocación de `onPress` cuando el usuario pulsa el botón; no realiza otras mutaciones ni llamadas internas por defecto.
- Comportamiento por defecto: si no se proporcionan estilos, el botón se muestra en la esquina inferior derecha con fondo de color de acento, tamaño fijo (60×60) y borde redondeado completo.
- Extensibilidad: mediante `containerStyle` y `iconContainerStyle` es posible adaptar posición, tamaño y apariencia sin modificar la lógica interna.

Figura 3.9: *FloatingButton* — ejemplo de botón flotante

IngredientCard

El componente `IngredientCard` es una tarjeta de presentación de datos cuyo propósito es mostrar sin permitir su modificación la información relevante de un `RecipeIngredient` asociado a una receta. Está concebido como un componente puramente presentacional que recibe un único parámetro con la estructura del ingrediente (identificador, nombre, cantidad y unidad) y lo renderiza de forma clara para el usuario.

Descripción funcional

- `IngredientCard` muestra el nombre del ingrediente y la cantidad requerida por la receta; su diseño prioriza la legibilidad y la consistencia visual con el resto de la interfaz.

- No permite la edición directa de los campos que muestra; cualquier modificación debe realizarse a través de los flujos de edición correspondientes de la aplicación.
- Se utiliza en la página RecipeScreen y en el modal AddRecipeModal como elemento de la lista de ingredientes de una receta.

Interfaz pública (props) y contrato

- **Prop principal:** un objeto de tipo `RecipeIngredient` (o similar) que contiene al menos `id`, `name`, `quantity` y `quantityType`.
- **Contrato:** entrada = datos de un `RecipeIngredient`; salida = renderizado visual; efectos secundarios = ninguno. El componente garantiza no mutar los datos recibidos y no invocar callbacks de modificación.
- **Precondiciones:** el objeto recibido debe contener un identificador único y un nombre legible; la cantidad puede ser numérica o nula (el componente debe mostrar `N/A` o un marcador equivalente cuando falten datos).

Estructura y dependencias

- Es una función / componente funcional escrito en TypeScript/React Native.
- Depende únicamente de primitives de React Native para el renderizado (por ejemplo, `View`, `Text`, `StyleSheet`) y de los tipos definidos en `Types.tsx` para tipado robusto.
- No realiza llamadas a servicios ni accede al contexto de la aplicación; su lógica se limita a la presentación.

Comportamiento y flujo

- Al recibir un `RecipeIngredient`, renderiza en orden: título/etiqueta, nombre del ingrediente y la línea de cantidad/unidad.
- Si faltan campos (por ejemplo, `quantityType`), muestra un indicador textual coherente (por ejemplo, `N/A`).
- Diseñado para ser usado en listados; se recomienda emplear como `key` en mapeos el `id` del `RecipeIngredient` para evitar re-renderizados innecesarios.

Consideraciones de robustez y casos borde

- **Datos incompletos:** mostrar marcadores claros cuando la información sea parcial.
- **Longitud del texto:** truncar nombres excesivamente largos o usar `numberOfLines` para evitar desbordes en la UI del modal o la pantalla.

Ejemplo de uso A modo ilustrativo, en `RecipeScreen` o `AddRecipeModal` se utiliza típicamente dentro de un mapeo:

```
// Fragmento de ejemplo
import { IngredientCard } from '../Components/IngredientCard';

// dentro del render
{recipeIngredients.map(ri => (
 <IngredientCard key={ri.id} recipeIngredient={ri} />
 ))}
```

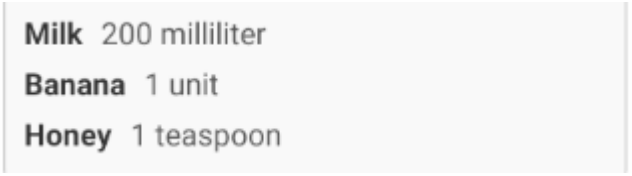


Figura 3.10: Ejemplos de varios componentes *IngredientCard*

AppHeader

El componente `AppHeader` se utiliza para mostrar las cabeceras en las distintas pantallas de la aplicación. Está diseñado como un componente funcional que recibe un texto para mostrar como título y, opcionalmente, un componente que se situará a la derecha del título (por ejemplo, un botón de acción o un icono). Su objetivo es proporcionar una presentación consistente de las cabeceras y facilitar la reutilización en todas las vistas.

Descripción funcional

- Muestra un título claramente legible, con estilo destacado, alineado a la izquierda.
- Permite la inserción de un componente auxiliar en la zona derecha de la cabecera (por ejemplo, un botón de cerrar sesión, acceso a un menú o un icono de ayuda).
- Mantiene un diseño sencillo y adaptable para diferentes tamaños de pantalla.

Parámetros

- `title: string` — cadena que contiene el texto a mostrar en la cabecera.
- `rightComponent?: React.ReactNode` — componente opcional que se renderiza a la derecha del título.

Estructura y uso

- Implementado con `View` y `Text` de React Native, y estilizado mediante `StyleSheet`.

- Está pensado para ser incluido en la parte superior de las pantallas principales de la aplicación, proporcionando uniformidad visual.
- La inclusión de componentes auxiliares a la derecha permite adaptarlo a distintos flujos funcionales sin alterar el diseño base.

Figura 3.11: *AppHeader*

HeaderComponent

El componente `HeaderComponent` se utiliza como cabecera específica de la pantalla principal (MainScreen). Está compuesto por una cabecera superior reutilizable (`AppHeader`) y una vista deslizable horizontal que presenta los días de la semana para permitir al usuario seleccionar el día deseado. Su propósito es ofrecer una navegación rápida entre los días y mantener coherencia visual con el resto de la interfaz.

Descripción funcional

- Muestra una fila superior con el título de la sección y un botón de acción a la derecha (gestionado por `AppHeader`).
- Debajo de la cabecera muestra un `ScrollView` horizontal con tarjetas representando cada día de la semana; el día seleccionado se resalta visualmente (color de acento) para indicar la elección del usuario.
- Está pensado exclusivamente para MainScreen y centraliza la lógica de selección del día en la UI de inicio.

Parámetros y contrato El componente se define como una función que recibe los siguientes parámetros:

- `selectedDay: DaysOfWeek` — día actualmente seleccionado por el usuario; se usa para aplicar el estilo activo a la tarjeta correspondiente.
- `setSelectedDay: (day: DaysOfWeek) =>void` — función que actualiza el día seleccionado en el estado del contenedor.
- `onButtonPress: () =>void` — callback que se ejecuta al pulsar el botón situado en la esquina superior derecha de la cabecera (comportamiento delegado al contenedor).

Contrato:

- Entrada: el día seleccionado actual, la función que lo modifica y la acción a ejecutar al pulsar el botón derecho.
- Salida: invocaciones a `setSelectedDay` cuando el usuario selecciona un día, y a `onButtonPress` al pulsar el botón auxiliar.
- Efectos secundarios: ninguno persistente; la mutación del estado se realiza en el contenedor mediante las funciones pasadas.

Estructura y dependencias

- Internamente combina `AppHeader` para la zona superior y un `ScrollView` horizontal para las tarjetas de los días.
- Utiliza el arreglo de días exportado (`daysOfWeekArray`) para renderizar las tarjetas de selección.
- Para la presentación emplea componentes nativos (`View`, `Text`, `TouchableOpacity`) y estilos mediante `StyleSheet`. La iconografía se obtiene de la librería de iconos que usa la aplicación.

Comportamiento de interacción

- Al tocar una tarjeta de día se ejecuta `setSelectedDay(day)`, provocando que la tarjeta pase a estado activo (cambio de color de fondo y texto).
- El botón ubicado en la parte superior derecha delega su acción a `onButtonPress`, permitiendo al contenedor decidir la operación (por ejemplo, navegar a otra pantalla o abrir un modal).
- La vista deslizable admite desplazamiento horizontal y oculta el indicador de scroll para una presentación limpia.

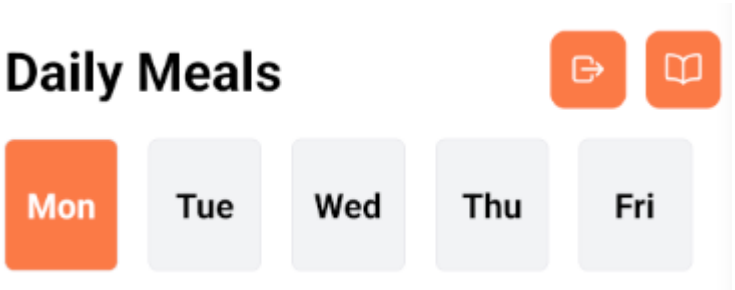


Figura 3.12: *HeaderComponent*

IngredientComponent

A continuación se explica, de manera concisa y práctica, el funcionamiento del componente `IngredientComponent`. Este componente es una unidad de interfaz que recibe como entrada un objeto `RecipeIngredient` (identificado por su `id`) junto con callbacks para actualizar cantidad, tipo de unidad y para eliminar el ingrediente. Su responsabilidad es presentar el nombre, la cantidad y la métrica del ingrediente y ofrecer controles de interacción (botones y campo numérico) que delegan la lógica de mutación en el componente contenedor.

Descripción funcional

- Muestra el nombre legible del ingrediente (resuelto a partir de la lista completa `ingredients` pasada como prop).
- Presenta un control de cantidad editable: botones de incremento/decremento y un campo de texto numérico con teclado decimal.
- Muestra un selector (Picker) para elegir el tipo de unidad asociado a la cantidad.
- Incluye un botón para eliminar el ingrediente; todas las acciones disparan callbacks proporcionados por el contenedor, por lo que la persistencia del estado se realiza fuera del componente.

Interfaz (props) y contrato El componente acepta las siguientes props (tipado en TypeScript, tal y como aparece en el código):

- `ingredients: Ingredient[]` — lista completa de ingredientes disponibles (usada para resolver el nombre por `id`).
- `id: string` — identificador único del ingrediente que este componente representa.
- `quantity: number` — cantidad actual asignada.
- `quantityType: QuantityType` — tipo de unidad asociado a la cantidad (p. ej. GRAM, ML, UNIT).
- `number: number` — índice/posición del componente en la lista (empleado únicamente para estilos como `zIndex`).
- `setQuantity: (quantity: number) =>void` — callback para actualizar la cantidad (la mutación real la realiza el contenedor).
- `setQuantityType: (quantityType: QuantityType) =>void` — callback para actualizar la unidad.

- `onDelete: (id: string) =>void` — callback para eliminar el ingrediente por su `id`.

Contrato:

- Entrada: props listadas arriba y la interacción del usuario (teclado, botones).
- Salida: invocaciones a `setQuantity`, `setQuantityType` y `onDelete` con los parámetros esperados.

Estructura interna y dependencias

- Implementado como componente funcional en TypeScript/React Native.
- Usa primitivas de React Native: `View`, `Text`, `TextInput`, `TouchableOpacity` y estilos mediante `StyleSheet`.
- Utiliza `Icon` (Ionicons) para los botones gráficos y un componente local `CustomPicker` para seleccionar el `quantityType`.
- Depende de los tipos definidos en `Types.tsx` (por ejemplo, `Ingredient` y `QuantityType`).

Comportamiento y detalles de UX

- El componente mantiene internamente `textValue` (string) para el campo de texto de la cantidad y sincroniza con `quantity` mediante `setQuantity` cuando el valor es válido.
- Al editar el campo numérico se valida con una expresión regular que permite dígitos y punto decimal; en `onBlur` se restablece el texto al valor numérico actual si la entrada no es válida.
- Los botones `remove` y `add` calculan el nuevo valor (asegurando un mínimo de 0), actualizan la cantidad mediante `setQuantity` y sincronizan el `textValue`.
- El `CustomPicker` controla el tipo de unidad a través de `pickerOpen` y `setQuantityType`; la lista de opciones usada es `quantityTypes`.
- Se aplica `numberOfLines` y `ellipsizeMode="tail"` al nombre para evitar desbordes y preservar la alineación en listas o modales.

Consideraciones de robustez y accesibilidad

- Validar que `ingredients.find(...)` devuelva un nombre antes de renderizar; mostrar un marcador (p. ej. `N/A`) si no se encuentra.
- Usar `recipeIngredient.id` como clave en los mapeos de lista; evitar índices como `key`.
- Añadir `accessible` y `accessibilityLabel` en elementos interactivos (botones) para mejorar compatibilidad con lectores de pantalla.
- Para colecciones grandes, considerar memoización del componente y usar `FlatList` con `getItemLayout` para optimizar el renderizado.

Figura 3.13: *IngredientComponent*

Componentes modales

AddIngredientModal **AddIngredientModal** ver Figura 3.14. Modal encargado de añadir ingredientes en la base de datos. Contiene dos campos de texto para insertar el nombre y la categoría del ingrediente y tres botones, uno superior encargado de cerrar el modal y dos inferiores, uno cancela el proceso (botón "*cancel*") y otro (botón "*Add Ingredient*") que al ser pulsado confirma que los datos insertados sean correctos, el nuevo ingrediente no exista en la base de datos para finalmente añadir el ingrediente, mostrar un mensaje de confirmación al usuario y cerrar el modal.

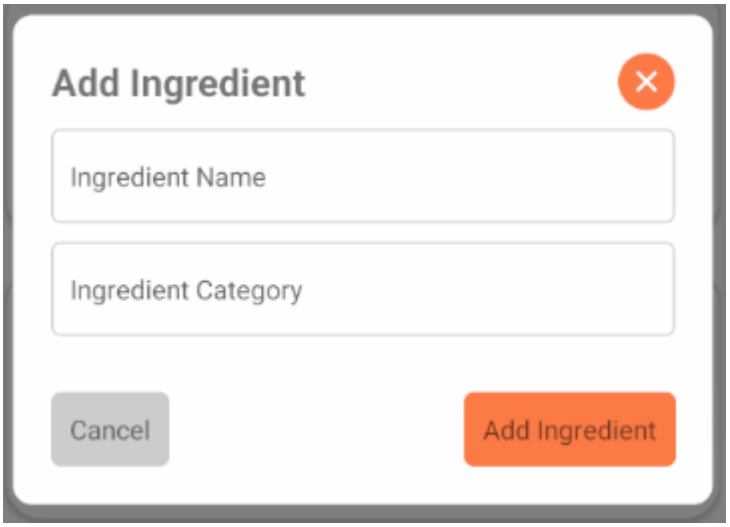
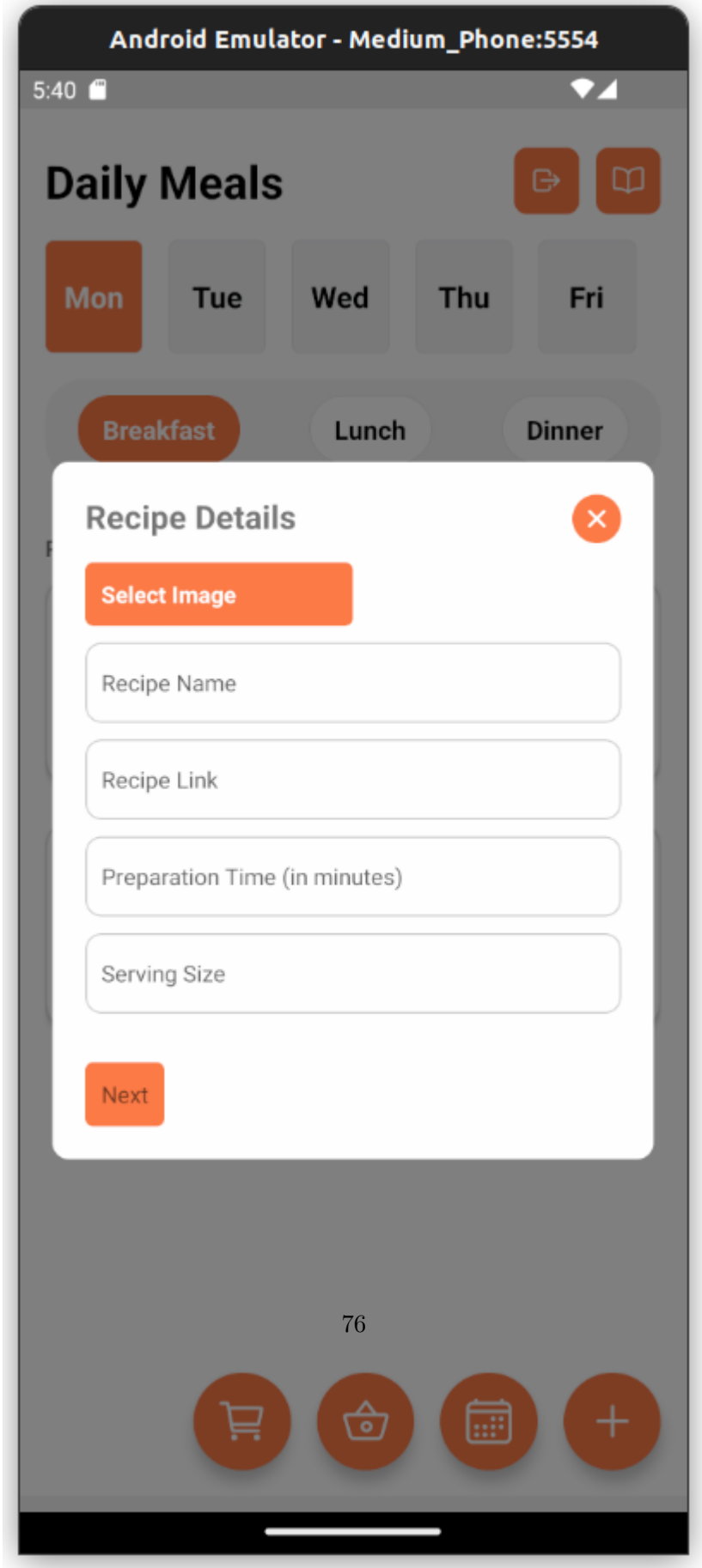
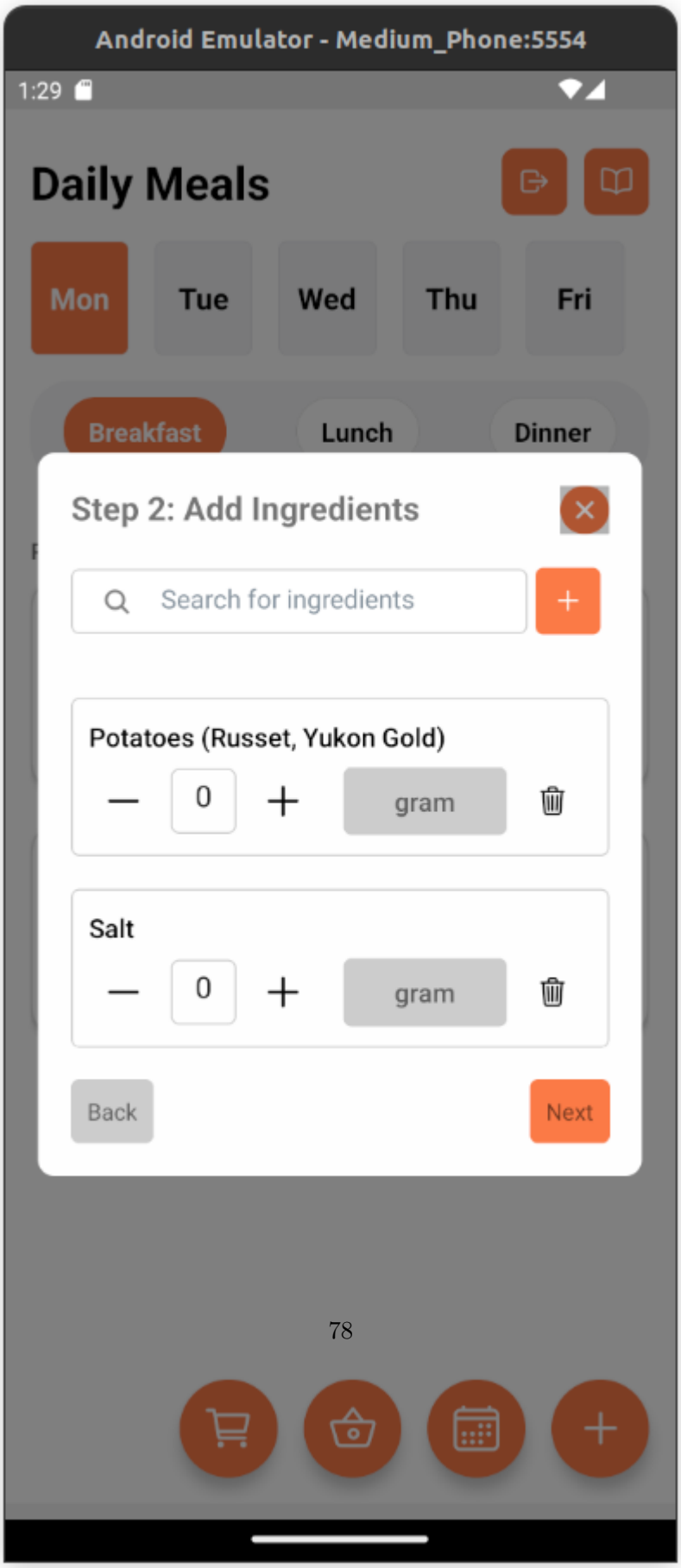


Figura 3.14: Modal de *AddIngredient*

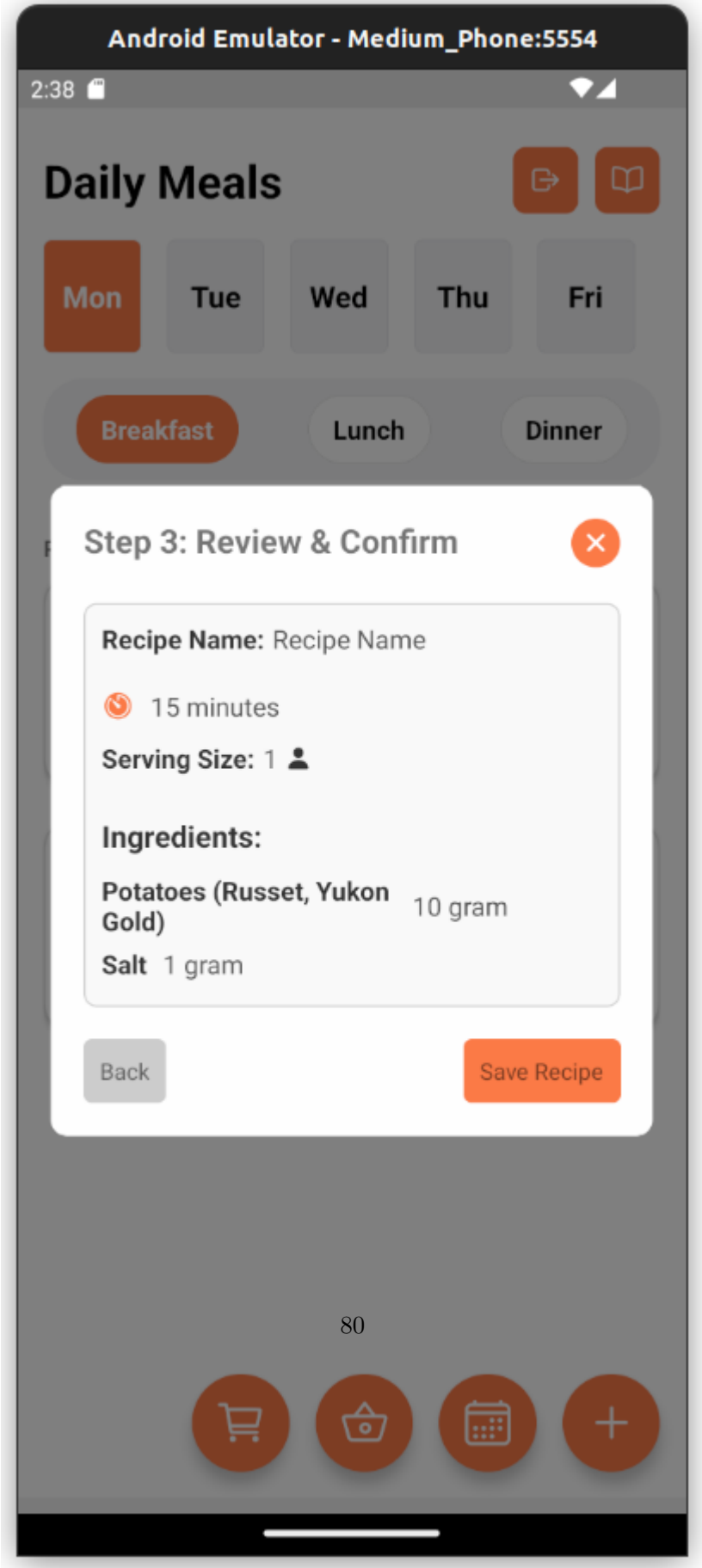
AddRecipeModal **AddRecipeModal (Ver Figura 3.15).** Es mostrado cuando el usuario pulsa en el botón de añadir receta situado en la parte inferior de MainScreen, está formado por tres secciones, la primera es la encargada de crear la colección Recipe, contiene un botón de selección de imagen, cuatro campos de texto (para el nombre, enlace, tiempo de preparación y raciones) y un botón para pasar a la siguiente fase. esta sección no permite al usuario pasar a la siguiente si no se han rellenado los datos necesarios.


Android Emulator - Medium_Phone:5554
5:40
Daily Meals
Mon
Tue
Wed
Thu
Fri
Breakfast
Lunch
Dinner
Recipe Details
Select Image
Recipe Name
Recipe Link
Preparation Time (in minutes)
Serving Size
Next

La segunda sección (ver Figure 3.16) es la encargada de permitir al usuario escoger todos los ingredientes necesarios para esta receta, añadir sus cantidades y confirmar que estas sean correctas. Consiste de una barra de búsqueda **??** que facilita al usuario encontrar un ingrediente determinado; en el caso de no encontrarlo, existe un botón a su derecha que abre el modal que crea un ingrediente (*AddRecipeModal*); debajo de ellos se encuentra una vista desplazable(*ScrollView*) que contiene la lista de los ingredientes que el usuario va escogiendo permitendo a este desplazarse verticalmente entre ellos. Los ingredientes escogidos son mostrados en la lista usando el componente *IngredientComponent* que permite visualizar, modificar o eliminar las cantidades del ingrediente escogido, seguido de un botón que comprueba que las cantidades de los ingredientes son correctas y lleva a la última sección; y otro botón que devuelve al usuario a la primera sección.


Android Emulator - Medium_Phone:5554
1:29
Daily Meals
Mon
Tue
Wed
Thu
Fri
Breakfast
Lunch
Dinner
Step 2: Add Ingredients
Search for ingredients
Potatoes (Russet, Yukon Gold)
0
gram
Salt
0
gram
Back
Next

En la tercera y última sección, se muestran todos los datos e ingredientes introducidos en las secciones anteriores para que el usuario las compruebe y confirme la adición de la receta en la aplicación. La confirmación se realiza al pulsar en el botón de "*Save Recipe* (Guardar receta)", esto hace que el sistema recoja todos los datos, cree la receta, los *RecipeIngredients* (Ingredientes de la receta) y añade estos en la base de datos. Una vez finalizados los procesos muestra un mensaje de confirmación al usuario y la receta es mostrada en la pantalla de recetas (*RecipeScreen*)


Android Emulator - Medium_Phone:5554
2:38
Daily Meals
Mon
Tue
Wed
Thu
Fri
Breakfast
Lunch
Dinner
Step 3: Review & Confirm
Recipe Name: Recipe Name
15 minutes
Serving Size: 1
Ingredients:
Potatoes (Russet, Yukon Gold) 10 gram
Salt 1 gram
Back
Save Recipe

IngredientSearchModal El modal `IngredientSearchModal` es la herramienta principal para buscar y seleccionar ingredientes existentes o añadir uno nuevo cuando no se encuentra. Se presenta como una ventana emergente con una barra de búsqueda en la parte superior que indexa los ingredientes locales usando MiniSearch, mostrando resultados en tiempo real y permitiendo ajustar la cantidad y la unidad antes de confirmar la inserción en la receta o en la despensa. Este modal está pensado para ser rápido, tolerante a entradas parciales y funcionar sin dependencia de la red, por lo que la experiencia es fluida incluso en condiciones de conectividad limitada. A nivel de interacción, el modal ofrece un flujo corto: búsqueda, selección, ajuste de cantidad/unidad y confirmación; si el usuario no encuentra el ingrediente, dispone de un acceso directo al modal de creación de ingrediente.

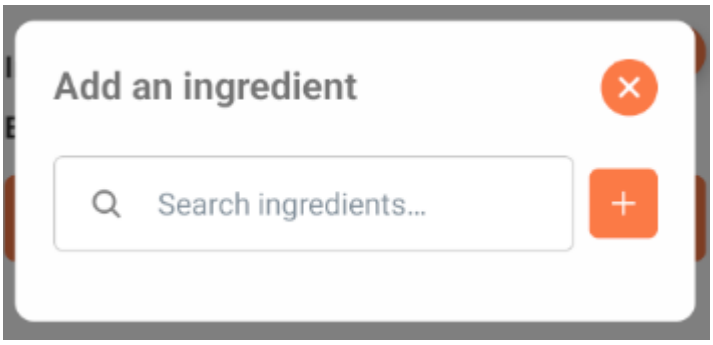


Figura 3.18: *IngredientSearch* modal

IngredientSearchSelectorComponent El componente `IngredientSearchSelectorComponent` es una versión compacta y reutilizable del selector de ingredientes usado dentro de otros modales y pantallas. Su propósito es encapsular la lógica de búsqueda, selección múltiple y visualización de las entradas seleccionadas en una forma que pueda integrarse dentro de otros formularios sin duplicar código. Desde el punto de vista del diseño, su comportamiento prioriza la claridad: muestra las coincidencias, permite seleccionar una entrada y ajustar su cantidad, y comunica los cambios al contenedor mediante callbacks. Esta separación facilita mantener la lógica de búsqueda aislada y reutilizable en diferentes contextos de la aplicación.

CustomPicker El `CustomPicker` es un selector personalizado que sustituye a pickers nativos para ofrecer un aspecto y comportamiento homogéneo entre plataformas. Se utiliza principalmente para escoger el `quantityType` (unidad de medida) en los componentes que requieren selección de unidades. Está implementado como un modal ligero que muestra una lista de opciones con feedback táctil y permite cerrar la selección sin cambiar el valor. Este componente asegura consistencia visual y evita diferencias de comportamiento entre iOS y Android.

Figura 3.19: *CustomPicker*

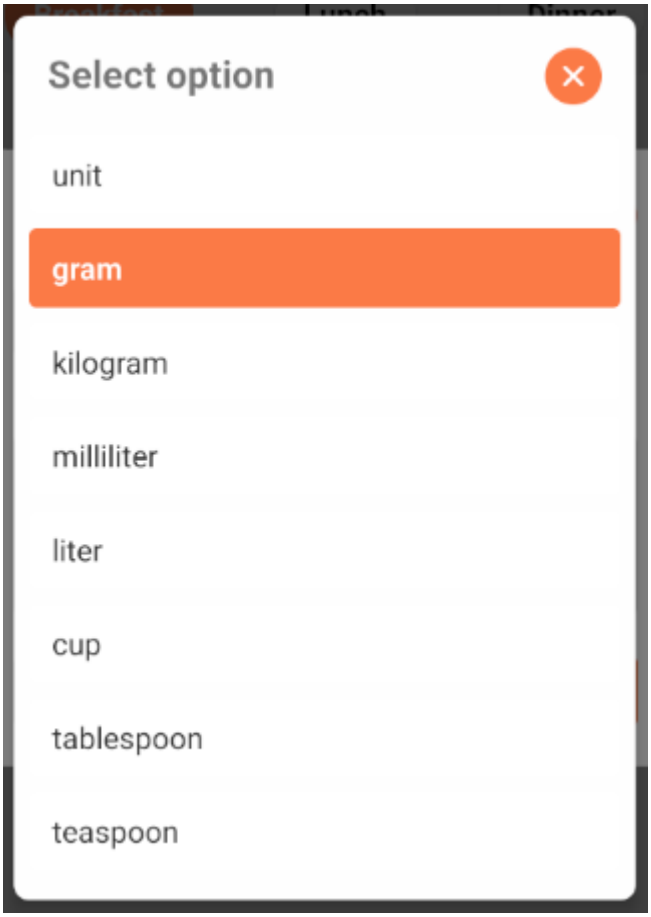


Figura 3.20: *CustomPickerModal*

AddIngredientButton El pequeño componente `AddIngredientButton` actúa como disparador de la creación rápida de ingredientes desde contextos donde el usuario está añadiendo recetas o editando la despensa. Su diseño es compacto y reconocible (icono -") y su

responsabilidad es abrir el modal de creación de ingrediente, centralizando la lógica de navegación y manteniendo la UI limpia. Su existencia evita repetir el mismo control en múltiples lugares y mejora la coherencia de la interfaz.

Figura 3.21: *AddIngredientButton* modal

MealsHeader extttMealsHeader es un componente de cabecera específico para la pantalla de planificación semanal. Combina elementos de navegación temporal (selección de día) con controles de contexto (tipo de comida), organizando la información de forma que el usuario pueda cambiar rápidamente el día o la comida a la que desea asignar recetas. Está diseñado para ser compacto y semántico: el día y el tipo de comida son los ejes que determinan el contenido mostrado en la pantalla principal.

MealTypeComponent El `MealTypeComponent` abstrae la selección del tipo de comida (desayuno, comida, cena). Presenta botones o fichas con el nombre de la comida y resalta la opción activa. Su mérito principal es separar la lógica de presentación del tipo de comida del resto de la pantalla, facilitando usar la misma pieza en diferentes vistas y simplificando pruebas y mantenimiento.

ModalHeareComponent El componente `ModalHeareComponent` (sic. nombre según el repositorio) actúa como cabecera reutilizable dentro de los modales complejos. Provee el título, el botón de cierre y, opcionalmente, atajos contextuales (por ejemplo, guardar o retroceder). Su uso mejora la coherencia entre los distintos modales y reduce la repetición de código para la gestión de la zona superior de estas ventanas.

PlannedIngredientCard extttPlannedIngredientCard es la representación compacta de un ingrediente incluido en la planificación semanal. A diferencia de `IngredientCard`, incluye información sobre el contexto de planificación (día, comida) y permite acciones rápidas como desasignar o ajustar la cantidad propuesta. Su papel es servir de elemento visual en listados de planificación, manteniendo un equilibrio entre detalle y compacidad.

RecipeCardComponent El `RecipeCardComponent` es la tarjeta utilizada en listados de recetas. Muestra el título, una miniatura de la imagen si existe y una breve descripción o metadato (tiempo de preparación, porciones). Las tarjetas son táctiles: un toque abre el detalle de la receta y una pulsación larga abre las opciones contextuales. La tarjeta está diseñada para priorizar la legibilidad y permitir una navegación rápida entre recetas.

RecipeOptionsModal El modal `RecipeOptionsModal` agrupa acciones contextuales sobre una receta: editar, eliminar, clonar o añadir a la planificación. Se invoca desde una pulsación larga sobre una tarjeta o desde un botón de opciones. La lógica del modal delega las operaciones al contexto o a los servicios correspondientes, mostrando confirmaciones cuando la acción es destructiva. Su objetivo es concentrar las operaciones avanzadas sin sobrecargar la vista principal.

PlanMealModal extttPlanMealModal es el modal encargado de asignar una receta o un ingrediente a un día y tipo de comida. En su interfaz el usuario elige el día, el tipo de comida y confirma la planificación; internamente valida conflictos sencillos (por ejemplo, evitar duplicados no deseados) y delega la persistencia a los servicios de planificación. Es el punto de conexión entre la exploración de recetas y la generación del calendario semanal.

PlannedIngredeintOptionsModal El modal `PlannedIngredeintOptionsModal` (nombre tal cual en el repositorio) provee opciones específicas para un ingrediente que ya forma parte de la planificación: modificar cantidad, cambiar unidad, mover a otro día o eliminar de la planificación. Centralizar estas acciones en un modal evita interfaces fragmentadas y facilita la confirmación de cambios que afectan a la lista de la compra resultante.

3.2. Mockups

3.3. Conclusiones

Capítulo 4

Prototipos y desarrollo

4.1. Prototipo 1

El primer prototipo se centró en construir la base de la aplicación. En esta etapa se implementó la autenticación de usuarios mediante Firebase, permitiendo el registro e inicio de sesión. Se diseñaron las pantallas *LoginScreen* y *RegisterScreen* con un estilo minimalista y se desarrolló la página principal (*MainScreen*) como punto central de navegación. También se creó la funcionalidad básica de gestión de recetas: el usuario podía añadir una nueva receta con su título, enlace y porciones, visualizar la lista de recetas en *RecipesScreen* y acceder al detalle en *RecipeScreen*. Se definieron las colecciones en Firestore y se validaron las operaciones CRUD básicas. Este prototipo sirvió para sentar las bases de la arquitectura en capas y de la navegación entre pantallas.

4.2. Prototipo 2

En el segundo prototipo se ampliaron notablemente las funcionalidades. Se añadió la planificación semanal, permitiendo asignar recetas o ingredientes a días y comidas mediante la entidad `WeeklyMeal`. Para ello se crearon componentes de selección de día y tipo de comida en la *MainScreen* y se desarrolló el *PlanMealModal*. Se incorporó la gestión de despensa (*PantryScreen*), donde el usuario puede visualizar y modificar los ingredientes disponibles, así como la generación automática de la lista de la compra (*GroceryListScreen*). Se implementó el algoritmo que suma las cantidades de ingredientes planificados y resta las existencias de la despensa, consolidando resultados en una lista ordenada382000137647362†L1773-L1794. Además se integró `react-minisearch` para la búsqueda local de ingredientes, mejorando la experiencia de añadir elementos a las recetas y a la despensa. Con este prototipo se validó la viabilidad de los casos de uso principales.

4.3. Prototipo 3

El tercer prototipo culminó el desarrollo incorporando funcionalidades avanzadas y optimizando la interfaz. Se añadió la posibilidad de asociar imágenes a las recetas utilizando bibliotecas nativas de selección de medios y almacenamiento en Firebase Storage. Se completó la lógica de edición de recetas e ingredientes, permitiendo modificar cantidades y unidades desde las diferentes pantallas. Se pulieron los componentes de navegación y se mejoró el diseño visual mediante la unificación de la paleta de colores y la creación de componentes reutilizables. Esta versión también implementó un sistema de notificación visual para marcar ingredientes como comprados y gestionó la actualización en tiempo real de la despensa y la lista de la compra. Finalmente se llevaron a cabo pruebas de integración y pruebas con usuarios que permitieron ajustar detalles de usabilidad antes de considerar la aplicación terminada.

4.4. Conclusiones

El desarrollo incremental a través de prototipos permitió validar conceptos y ajustar la funcionalidad en función de la retroalimentación. El primer prototipo demostró que la combinación de React Native y Firebase era adecuada para una aplicación multiplataforma y sentó la base de la arquitectura. El segundo prototipo integró los casos de uso esenciales (planificación, despensa y lista de la compra) y permitió comprobar la coherencia del modelo de datos y del flujo de navegación. El tercer prototipo refinó la experiencia de usuario, añadió funciones multimedia y culminó con una versión estable y testada. Esta metodología iterativa facilitó detectar y corregir errores de forma temprana y permitió concentrarse en mejorar la aplicación en cada ciclo.

Capítulo 5

Conclusiones y mejoras futuras

5.1. Conclusiones técnicas

5.2. Conclusiones personales

5.3. Futuras mejoras

Capítulo 6

Conclusions and future works

6.1. Technical conclusions

6.2. Personal conclusions

6.3. Future works

Bibliografía

[1] Benjamins, J., Hilkens, L., Cruyff, M., and Woertman, L. Social media, body image and resistance training: Creating the perfect 'me'. *Sports Medicine - Open 7*, 71 (2021).

[2] Business Research Insights. Meal planning app market report. https://www.businessresearchinsights.com/es/market-reports/meal-planning-app-market-113013, 2024.

[3] de las Naciones Unidas para la Alimentación y la Agricultura, O. El comercio de alimentos y la obesidad.

[4] Dimitri, C., and Rogus, S. The effects of consumer knowledge on the use of grocery lists. *Journal of Nutrition Education and Behavior 49*, 8 (2017), 745–753.

[5] Ducrot, P., Méjean, C., Allès, B., Fassier, P., Hercberg, S., and Péneau, S. Meal planning is associated with food variety, diet quality and body weight status in a large sample of french adults. *International Journal of Behavioral Nutrition and Physical Activity 14*, 12 (2017).

[6] European Commission. Food waste in the eu. https://food.ec.europa.eu/food-safety/food-waste_en, 2020.

[7] Food Marketing Institute. U.s. grocery shopper trends 2015, 2015.

[8] Helms, E. R., Prnjak, K., and Linardon, J. Towards a sustainable nutrition paradigm in physique sport: A narrative review. *Sports 7*, 7 (2019), 172.

[9] Johns Hopkins Bloomberg School of Public Health. Study suggests home cooking is main ingredient in healthier diet. https://clf.jhsph.edu/about-us/news/news-2014/study-suggests-home-cooking-main-ingredient-healthier-diet, 2014.

[10] Lappalainen, R. e. a. E. j. o. c. n. v. Difficulties in trying to eat healthier: descriptive analysis of perceived barriers for healthy eating.

[11] Liew, M., Zhang, J., See, J., and Ong, Y. Usability challenges for health and wellness mobile apps: Mixed-methods study among mhealth experts and consumers. *JMIR mHealth and uHealth 7*, 1 (2019), e12160.

[12] Lim, P., Lim, Y., Rajah, R., et al. Cuestionario de usabilidad para aplicaciones móviles de salud independientes o interactivas: una revisión sistemática. *BMC Digital Health 3* (2025), 11.

[13] Masoga, S. Nutrition information sources used by amateur bodybuilding athletes around polokwane municipality in limpopo province, south africa. *Research Square* (2021).

[14] McKinsey & Company. The state of consumer health and nutrition 2023. `https://www.businessresearchinsights.com/es/market-reports/meal-planning-app-market-113013`, 2023.

[15] Nielsen, J. Las diez heurísticas de usabilidad. `https://www.nngroup.com/articles/ten-usability-heuristics`, 1994. Consultado en octubre de 2025.

[16] of Medicine, N. L. Meal planning is associated with food variety, diet quality and body weight status in a large sample of french adults.

[17] Organización Mundial de la Salud, O. *Obesidad y Sobrepeso*. 2025.

[18] Overhoff, T. How to meal prep for the week.

[19] Wikipedia contributors. Less is more — wikipedia, the free encyclopedia, 2025. Consultado el 7 de octubre de 2025.