

ISRAEL LUCAS TORRIJOS

CFGS DAM CURSO 25/26

Fecha: 08/12/2025

IES AZARQUIEL

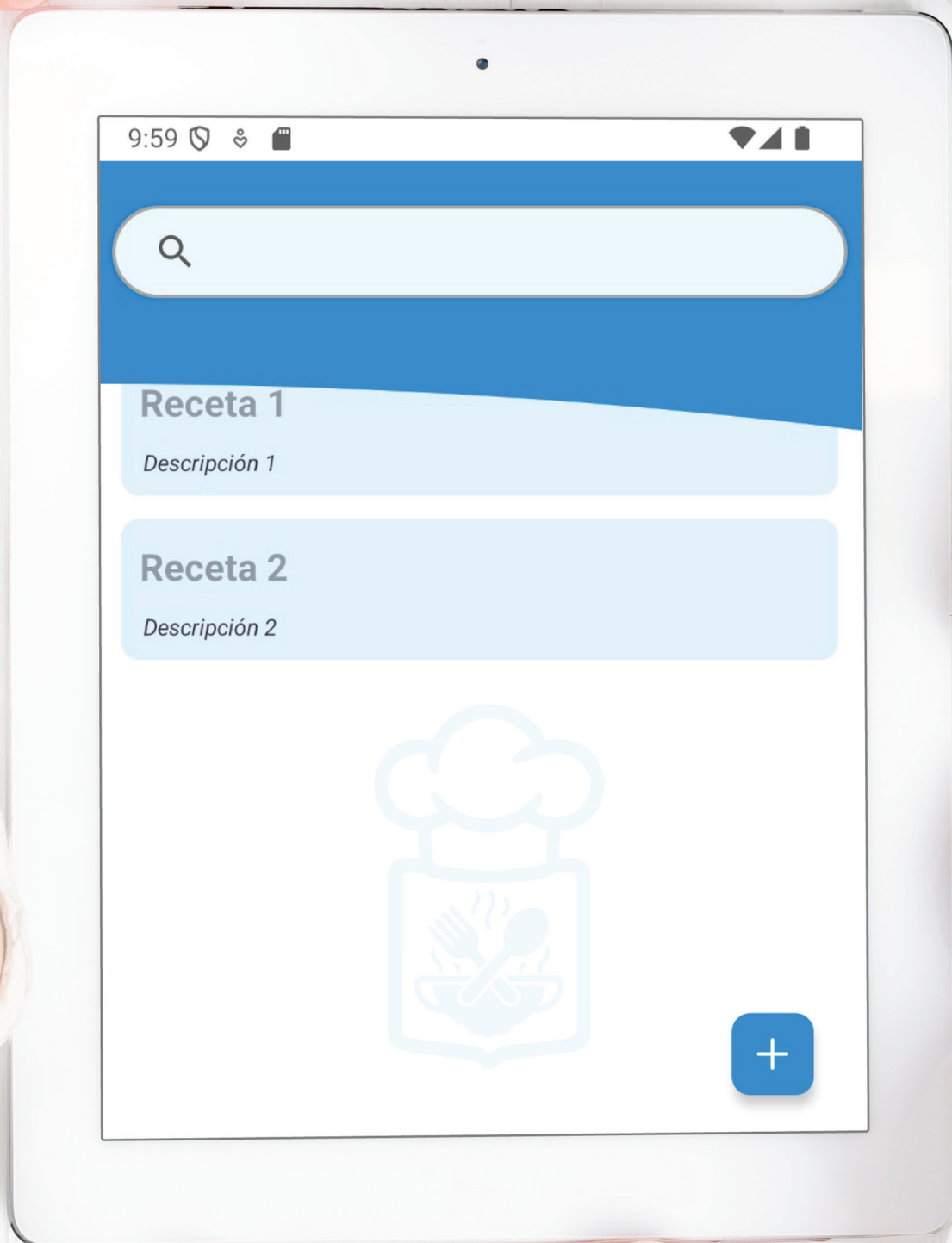
Proyecto: Digital Recipes

Tutor: D. Francisco J. Pulido Moya

digital



recipes



## Tabla de contenido

Proyecto intermodular: Digital Recipes .....	3
Capítulo 1. Introducción y objetivos.....	3
Capítulo 2. Especificación de Requisitos .....	4
Capítulo 3. Planificación Temporal y Evaluación de Costes .....	5
3.1. Planificación temporal .....	5
3.2. Evaluación de costes .....	6
Capítulo 4. Tecnologías Utilizadas .....	7
Capítulo 5. Desarrollo e Implementación.....	11
5.1. Model .....	11
5.2. View.....	16
5.3. ViewModel .....	43
Capítulo 6. Conclusiones y líneas futuras.....	46
Capítulo 7. Bibliografía. ....	47
7.1. Libros.....	47
7.2. Páginas web .....	47
Anexos .....	48
Anexo I. Entidades Room.....	48
Anexo II. Relaciones Room .....	49
Anexo III. Implementación DAO .....	50
Anexo IV. Base de datos .....	51
Anexo V. Poblar base de datos.....	52
Anexo VI. Repositorio .....	53

## Proyecto intermodular: Digital Recipes

### Capítulo 1. Introducción y objetivos

**Digital Recipes** (en español, *Recetas Digitales*) es una aplicación móvil para dispositivos con sistema operativo **Android 8.0** (Oreo) o superior.

Su principal objetivo es facilitar al usuario la **gestión de recetas de cocina** mediante un sencillo sistema de acceso y organización.

No resulta extraño que tengamos nuestras recetas dispersas en distintos formatos, como: documentos impresos, notas en el móvil, capturas de pantalla, etc. Emplear varios sistemas de organización y almacenamiento, nos dificulta recordar dónde las habíamos guardado o bajo qué nombre.

Esta aplicación pretende **eliminar estas barreras**, para que los usuarios puedan almacenar y consultar fácilmente sus recetas.

Entre otros **objetivos** del proyecto destacan:

- Reunir todas las recetas en un único lugar unificando el sistema de almacenamiento.
- Facilitar la consulta de recetas mediante una interfaz clara y estructurada.
- Organizar los ingredientes y los pasos de elaboración de forma intuitiva.
- Garantizar que aplicación sea sencilla de utilizar para abarcar distintos segmentos de mercado.
- Asentar las bases para futuras implementaciones, como: clasificación por categorías, favoritos o exportación.
- Ser socialmente responsable. Es necesario impulsar el formato digital para evitar el uso del formato físico.

En definitiva, **Digital Recipes** pretende convertirse en una herramienta práctica para el uso cotidiano en la cocina.

## Capítulo 2. Especificación de Requisitos

La **especificación de requisitos** forma parte de la **fase de análisis** en el desarrollo de software. Esta fase permite identificar y describir los requisitos o capacidades que el sistema debe ofrecer para dar respuesta a las necesidades de los usuarios.

Habitualmente, la obtención de requisitos se realiza mediante la comunicación con el cliente/usuario. En el caso de un proyecto intermodular, será el propio desarrollador quién asumirá ambos roles:

- Detectar las necesidades que motivan la aplicación, explicado en el punto anterior.
- Definir los requisitos **funcionales** y **no funcionales** del sistema.

En la siguiente tabla se detallan los requisitos:

Funcionales	No funcionales
El usuario puede ver un listado de recetas.	La aplicación funcionará en dispositivos con Android 8.0 (Oreo) o superior.
El usuario puede ver una receta con su portada, lista de ingredientes y pasos de elaboración.	La interfaz será sencilla y clara, adecuada para cualquier tipo de usuario.
El usuario puede buscar una receta por su nombre.	La aplicación no dependerá de conexión a Internet.
El usuario puede añadir nuevas recetas con imagen opcional.	La estructura interna permitirá añadir mejoras en futuras versiones (categorías, favoritos, exportación, etc.).
El usuario puede especificar ingredientes indicando cantidad y unidad.	Los textos y elementos visuales respetarán tamaños y contrastes adecuados para facilitar la lectura.
El usuario puede redactar los pasos de elaboración de cada receta.	La aplicación funcionará mediante interacción táctil.
El usuario puede modificar una receta existente (portada, ingredientes y pasos).	Se recomienda disponer de al menos 1 GB de almacenamiento libre.
El usuario puede eliminar una receta de forma controlada.	Se recomienda 1 GB de memoria RAM disponible para un funcionamiento fluido.
Las recetas se guardan mediante persistencia local en la base de datos.	Los datos se conservarán, aunque se cierre la aplicación o se reinicie el dispositivo.
Las operaciones internas hacen uso de concurrencia, evitando bloquear la interfaz.	Las operaciones de carga, listado y búsqueda deberán ejecutarse de manera fluida.

## Capítulo 3. Planificación Temporal y Evaluación de Costes

La planificación se ha realizado siguiendo una estructura secuencial. El trabajo se ha dividido en fases y, para cada una, se ha establecido una duración aproximada.

### 3.1. Planificación temporal

El desarrollo del proyecto se ha dividido en las siguientes etapas:

Fase	Descripción	Días
Análisis y definición de requisitos	Identificación de las necesidades del usuario, definición de los objetivos del proyecto y delimitación del alcance.	2
Diseño de la base de datos y arquitectura	Diseño del modelo de datos (entidades y relaciones) y elección de la arquitectura <b>MVVM</b> .	5
Diseño de la interfaz	Bocetos de pantallas, selección de colores, tipografía, iconografía y estructura de navegación.	4
Implementación del modelo y repositorio ( <b>Room</b> )	Codificación de entidades, <b>DAOs</b> y repositorio, con prueba de operaciones <b>CRUD</b> .	5
Implementación de <b>ViewModel</b> y <b>LiveData</b>	Conexión entre la lógica de datos y la interfaz. Observadores, comunicación entre fragments y estados de vista.	5
Desarrollo de <b>Activities</b> y <b>Fragments</b> Integración de interfaz y lógica Pruebas y corrección de errores Documentación del proyecto	Creación de pantallas principales, vistas de detalle, formularios y navegación mediante <b>ViewPager2</b> y <b>Navigation Components</b> .	12
	Enlace de layouts con <b>ViewBinding</b> , adaptadores, carga de imágenes ( <b>Glide</b> ) y control de eventos.	4
	Verificación funcional, interfaz, rendimiento y usabilidad.	2
	Elaboración de la memoria y anexos.	10
Análisis y definición de requisitos	Identificación de las necesidades del usuario, definición de los objetivos del proyecto y delimitación del alcance.	2
Diseño de la base de datos y arquitectura	Diseño del modelo de datos (entidades y relaciones) y elección de la arquitectura <b>MVVM</b> .	5

Duración estimada: **48 días de trabajo efectivo.**

### 3.2. Evaluación de costes

El desarrollo del proyecto se ha realizado íntegramente con **herramientas gratuitas**, por lo que no existen costes asociados al uso de hardware o software.

En la siguiente tabla se desglosan los recursos empleados y su coste:

Recurso	Tipo	Coste (€)
Android Studio	Entorno de desarrollo	0
Java (JDK)	Lenguaje y compilador	0
Android SDK	Herramientas de compilación	0
Librerías Jetpack (Room, ViewModel, LiveData)	Framework oficial	0
Glide	Librería para carga de imágenes	0
Dispositivo Android	Uso personal	0

Para la estimación del coste del tiempo invertido se toman como referencia las **40 horas** de distribución recogidas en el **Decreto 252/2011**, de 12/08/2011, de Castilla-La Mancha, aunque el tiempo real de desarrollo ha sido mayor.

Supondré una contraprestación económica de **20 €/hora**, incluyendo los gastos aparejados como consumo eléctrico. Se estimaría un coste total de: **800 euros**.

Esta cifra no supone un gasto real, sino una valoración del esfuerzo de desarrollo.



## Capítulo 4. Tecnologías Utilizadas

Para el desarrollo de la aplicación se han empleado distintas tecnologías y patrones de diseño propios del entorno **Android**. Se ha querido realizar una base sólida que permita ampliar funcionalidades en futuras versiones, siguiendo las recomendaciones de **Android Developer**.

- **Lenguaje de programación.**

Se ha optado por Java, ya que ha sido el lenguaje empleado durante el desarrollo del ciclo formativo. **Android** ha declarado a **Kotlin** como lenguaje oficial, la migración a este lenguaje sería una funcionalidad futura a implementar.

- **Entorno de desarrollo.**

Android Studio se ha utilizado como IDE principal, dado que integra un editor visual, emulador y administrador de dependencias **Kotlin DSL**.

Los plugin **Github** y **PlantUML**, entre otros, han sido necesarios para el desarrollo del proyecto.

- **Arquitectura.**

Los patrones de diseño empleados han sido múltiples:

1. **Patrón MVVM (Model-View-ViewModel)**

Este patrón permite separar la lógica de negocio de la interfaz gráfica de manera que las **View** gestionan la parte visual, los **ViewModel** administran la persistencia durante el ciclo de vida de la actividad y el **Model** que se encarga de implementar la persistencia local.

2. **Patrón DAO**

La implementación de **Room** implica la creación de clases **DAO** por cada entidad que definimos. Aunque se pueden aglutinar en una clase por practicidad en proyectos pequeños, no se recomienda en aplicaciones de relevancia dado que ocasionan un alto grado de acoplamiento.

3. **Patrón Repository**

La clase **Repositorio** actúa como intermediario entre la base de datos (**Room**) y los **ViewModels**. Esta clase se englobaría dentro del **Model**.

#### 4. Patrón Singleton

Se ha aplicado el patrón **Singleton** en la creación de la base de datos **Room**, para asegurar que la aplicación mantiene una única instancia activa.

#### 5. Métodos Factory

Son utilizados y requeridos para la creación de instancias de **Fragments**.

#### 6. Patrón Observer

El patrón **Observer** permite crear una escucha activa de los cambios o modificaciones en los datos de las tablas de **SQLite**. Será **Room** quien notifique internamente dichos cambios a las instancias **LiveData**, para que los observadores puedan actualizar los datos mostrados por las vistas o **Views**.

- **Persistencia local.**

La capa **Model** implica la utilización de las siguientes tecnologías:

##### 1. Room

Se ha usado la **API Room** para gestionar la persistencia local, ya que nos permite disponer de una capa de abstracción **ORM** e implementar acciones **CRUD** con verificación en tiempo de diseño. Es necesario realizar anotaciones en las entidades, así como de definir clases para relaciones entre entidades.

Sin embargo, resulta en una implementación más limpia que la empleada con la **API de SQLite**.

```
implementation("androidx.room:room-runtime:2.8.3")  
annotationProcessor("androidx.room:room-compiler:2.8.3")
```

##### 2. Repository

Esta clase actúa como intermediario entre la capa **ViewModel** y **Model**. Android recomienda que sea en esta capa donde las operaciones **DML** sean llamadas al **DAO** mediante la creación de **hilos secundarios** o workers.

Esta clase no realizará ningún tipo de transformación de los datos en los métodos que declara, ya que esa es responsabilidad de los **ViewModels**.



### 3. DAO

Interfaz necesaria para la implementación de **ROOM**, y mediante anotaciones se declaran métodos que permiten realizar acciones **CRUD**.

### 4. ViewModel

Los **ViewModel** tienen muchas ventajas asociadas, y la más importante es crear una persistencia temporal resistente a cambios de configuración de la aplicación, como puede ser la rotación del dispositivo.

Además, garantiza el acceso a dichos datos entre **Fragments** que dependen de la misma **Activity**.

```
implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.9.4")
```

### 5. LiveData

Permite que los datos mostrados en pantalla se actualicen automáticamente cuando cambia el contenido de la base de datos.

```
implementation("androidx.lifecycle:lifecycle-livedata-ktx:2.9.4")
```

- **Interfaz de usuario**

Las tecnologías empleadas para esta capa son las siguientes:

#### 1. RecyclerView

Se utiliza para mostrar listas de objetos en los que el usuario tendrá un nivel de interacción, dando un agradable aspecto visual y correcta gestión de rendimiento respecto a **ListViews**. Tenemos que añadir la dependencia de **RecyclerView** en el archivo `build.gradle` (Module :app)

```
implementation("androidx.recyclerview:recyclerview:1.4.0")  
implementation("androidx.cardview:cardview:1.0.0")
```

#### 2. API Result Launcher

Conjunto de mecanismos que permiten gestionar las operaciones que requieren obtener un resultado entre **Activities** o **Fragments**, como la selección de una imagen de la galería o la solicitar de permisos.

Esta **API** sustituye al método `startActivityForResult()` y el `callback onActivityResult()`.

### 3. *ViewBinding*

Permite vincular las vistas con código Java sin necesidad de usar el método `findViewById()`, lo que mejora la legibilidad y reduce errores.

Para habilitar esta característica, debemos modificar el archivo **build.gradle.kts** de la aplicación y especificar:

```
buildFeatures {  
    viewBinding = true  
}
```

### 4. *ItemTouchHelper*

API que permite añadir la función de deslizar, para descartar elementos, y arrastrar y soltar a **RecyclerView**.

**ItemTouchHelper.Callback** es el contrato entre **ItemTouchHelper** y su aplicación. Le permite controlar qué comportamientos táctiles están habilitados para cada **ViewHolder** y también recibir devoluciones de llamada cuando el usuario realiza estas acciones.

### 5. *ViewPager2*

Empleado para mostrar diferentes secciones dentro de la vista de una receta (portada, ingredientes y pasos) mediante pestañas deslizables.

```
implementation("androidx.viewpager2:viewpager2:1.1.0")
```

### 6. *BottomSheet*

Se utiliza para presentar acciones sobre otros objetos seleccionados.

### 7. *NavigationUI*

Gestiona la navegación entre pantallas de manera estructurada y respetando el ciclo de vida de los **Fragments**.

```
implementation("androidx.navigation:navigation-fragment:2.9.5")  
implementation("androidx.navigation:navigation-ui:2.9.5")
```

### 8. *Glide*

Librería externa que permite cargar y mostrar imágenes de forma eficiente, optimizando consumo de memoria.

```
implementation("com.github.bumptech.glide:glide:5.0.5")
```

## Capítulo 5. Desarrollo e Implementación

La recomendación de **Android Developer** para la implementación aplicaciones es emplear el patrón de arquitectura **MVVM** (Model-View-ViewModel). En base a estas tres capas, se estructurará la explicación de como se ha desarrollado la aplicación.

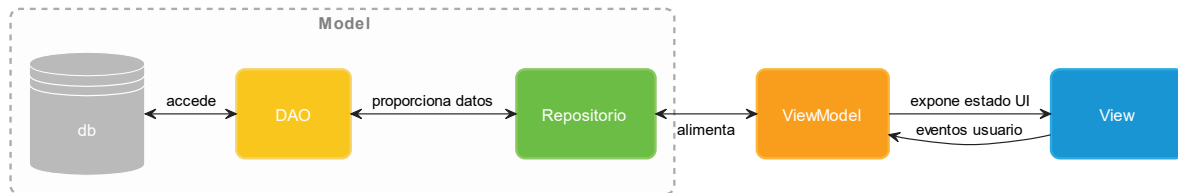


Figura 1. Arquitectura MVVM

### 5.1. Model

La etapa inicial del proyecto aborda la definición del modelo de datos en **Room**. Mediante el plugin **PlantUML** se ha generado el diagrama **Entidad/Relación** que define la estructura de la base de datos:

Diagrama E/R [Recetario]

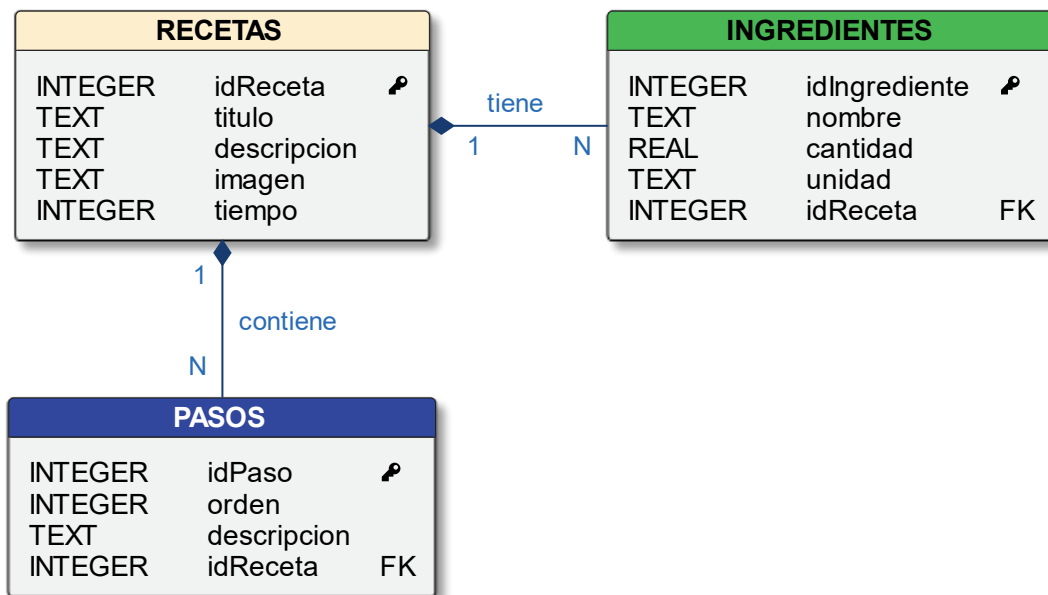


Figura 2. Diagrama Entidad-Relación del Recetario

La **API** de **Room** es una capa de abstracción **ORM** para acceder a la información de las bases de datos **SQLite**. En Android, tenemos que definir clases **Java** que se correspondan con las entidades/tablas en **Room** mediante la anotación **@Entity**.

Se crean tres clases java que representan **entidades** ([anexo I](#)):

- **Receta.** Representa la tabla *Recetas*. Incluye un campo autonumérico **idReceta** como clave primaria. El resto de campos permiten almacenar el título, la descripción, la imagen asociada y el tiempo de preparación.
- **Ingrediente.** Representa la tabla *Ingredientes*. Contiene un campo autonumérico **idIngrediente** como clave primaria. La propiedad **idReceta** se mapea a la columna `idReceta_fk` mediante la anotación `@ColumnInfo`. Lo que implica declarar la clave foránea en la anotación `@Entity` para garantizar la integridad referencial.
- **Paso.** Representa la tabla *Pasos*. Incluye el campo autonumérico **idPaso** como clave primaria. La propiedad **idReceta** se mapea a la columna `idReceta_fk` mediante la anotación `@ColumnInfo` como en la clase anterior.

Para consultar los datos entre **dos entidades** con una relación **1:N**, es necesario **modelar** dicha **relación** mediante una clase Java específica.

El modelado implica crear una clase que contenga una instancia de la clase padre anotada con `@Embedded` y otra instancia de la clase hija anotada con `@Relation`. Asignamos a `parentColumn` el nombre de la clave primaria de la entidad fuerte y a `entityColumn` el nombre de la entidad débil que hace referencia a la clave primaria de la entidad fuerte.

A tal efecto, se crean las siguientes clases Java que representan las **relaciones**:

- **RecetaIngredientes.** Permite realizar operaciones **CRUD** sobre los ingredientes asociados a una receta.
- **RecetaPasos.** Permite realizar operaciones **CRUD** sobre los pasos vinculados a una receta.
- **RecetaCompleta.** Permite realizar operaciones **CRUD** con los datos completos de la receta.

En el [anexo II](#) están las definiciones de las clases `@relation`.

### 5.1.1. Interfaces DAO

Las interfaces **DAO** son las responsables de definir los métodos de acceso a la base de datos. En **SQLite** empleamos los objetos **Cursor**. Con la **API Room**, no necesitamos todo el código relacionado con **Cursor**, y simplemente definimos nuestras consultas usando anotaciones en la clase **DAO**.

Definimos tantos métodos como sean necesarios para abarcar las operaciones **CRUD**. Cada método tiene aparejado una anotación con la operación que realiza: **@Insert**, **@Update** y **@Delete**. Estos métodos pueden retornar un valor numérico referente al **rowId** afectado.

Las consultas emplean la anotación **@Query** que declara la consulta **SQL** asociada, además para implementar correctamente el patrón **MVVM** los valores que retornan deben estar encapsulados con el tipo **LiveData**.

Los métodos que retornan instancias de clases que **modelan una relación** deben anotarse con **@Transaction**, ya que los campos anotados con **@Relation** se consultan por separado. Es recomendable emplear una clase **DAO** por cada entidad definida, ya que permite aumentar la modularidad.

En el [anexo III](#) están las definiciones de las interfaces.

### 5.1.2. Base de datos

Para definir la base de datos en **Room**, es necesario crear una clase que debe cumplir con las siguientes condiciones:

- Debe ser una clase abstracta que extienda de **RoomDatabase**.
- La clase debe tener una anotación **@Database**. En su atributo **entities** se declara un array de entidades, clases anotadas con **@Entity**.
- Para cada clase **DAO**, hay que definir un método abstracto con cero argumentos y muestre una instancia de la clase **DAO**.

Dentro de la clase se llama al método estático **Room.databaseBuilder()** para obtener la referencia a la base de datos.

Una práctica común consiste en implementar el patrón **Singleton** en la clase que gestiona la conexión a la base de datos. Este diseño garantiza la existencia de una única instancia del cliente de base de datos durante todo el ciclo de vida de la aplicación.

Para optimizar el manejo de operaciones concurrentes, se recomienda complementar esta arquitectura con un **pool** de hilos que administre las solicitudes de manera eficiente.

En el [anexo IV](#) se puede ver parte de la implementación.

### Poblar base de datos

Para comprobar que hemos creado el esquema de la base de datos correctamente y que Room crea las tablas, podemos usar el método `.addCallback(RoomDatabase.Callback)` de `Room.databaseBuilder` para capturar los métodos `onCreate()` y `onOpen()` del ciclo de vida de Room durante la creación de la base de datos.

Al implementar un objeto `RoomDatabase.Callback` definimos datos **dummy** en el método `onCreate()` mediante un hilo secundario o `worker`. En el [anexo V](#) se muestra como poblar la base de datos.

La implementación del **callback** se disparará cuando efectivamente se  **Cree la base de datos**, lo que ocurrirá cuando se ejecute la **primera operación de acceso** desde la capa de negocio.

Para forzar la creación de la base de datos podemos usar este método en durante la creación de la instancia de nuestra base de datos:

```
private static void inicializar() {  
    //Fuerza la creación de la base de datos  
    servicioExecutor.execute(() -> {  
        INSTANCIA.getOpenHelper().getWritableDatabase();  
    });  
}
```

En la **Activity**, creamos una instancia de la base de datos:

```
Recetario basedatos = Recetario.getInstance(getApplicationContext());
```

Para verificar la creación de la base de datos, en **Android Studio** abrimos el **App Inspection** desde el menú **View** → **Tool Windows** → **App Inspection**:



Figura 3. App Inspection de Android Studio

### 5.1.3. Repositorio

El último componente dentro de la capa **Modelo** es el **Repositorio**. Su función es centralizar y abstraer el acceso a los datos, actuando como intermediario entre el **ViewModel** y las diferentes fuentes de información.

De este modo, el **ViewModel** no necesita conocer los detalles de acceso o almacenamiento, simplemente solicita los datos al **Repositorio**.

En nuestro caso, el **Repositorio** se limita a **delegar** las operaciones sobre la base de datos a los métodos del **DAO**, encapsulándolos y evitando que otras capas accedan directamente a ellos.

Además, según las recomendaciones de **Android**, el **Repositorio** debe encargarse de **manejar los hilos de ejecución** en las operaciones de acceso a datos: *insert*, *update* y *delete*.

En la implementación del **Repositorio** se declaran como miembros las **interfaces DAO** necesarias, una **referencia a la base de datos**, que permite realizar las operaciones solicitadas, y una instancia del **contexto** de la aplicación.

En el [anexo VI](#) se muestra la implementación del **Repositorio**.



## 5.2. View

La capa View en **MVVM** corresponde a la interfaz gráfica de la aplicación. Sus componentes se definen en archivos de **layout XML**, los cuales se convierten en una jerarquía de objetos **View** durante el proceso de inflado.

Mediante **ViewBinding**, se genera automáticamente una clase asociada a cada **layout**, lo que permite acceder a sus vistas de forma segura y sin necesidad de realizar búsquedas manuales (`findViewById`), evitando errores de codificación y referencias nulas.

Las **Activities** representan las pantallas principales de la aplicación. No obstante, Android recomienda apoyarse en **Fragments** para lograr una mayor modularidad y reutilización de componentes dentro de una misma **Activity**, facilitando el desacoplamiento de la lógica de interfaz de usuario.

En esta aplicación se han creado cuatro **Activities**:

1. **RecetasActivity**
2. **VerRecetaActivity**
3. **CrearRecetaActivity**
4. **EditarRecetaActivity**

### 5.2.1. RecetasActivity

Esta actividad es la vista principal que se muestra al abrir la aplicación. Su finalidad es mostrar el listado de **Recetas** creadas por el usuario. Al capturar el evento **clic**, en cada ítem de la lista, la aplicación nos permitirá tres operaciones:

- **Ver Receta.** Permite visualizar el contenido de la receta.
- **Modificar.** Obtiene los detalles de la receta y los muestra para ser modificados.
- **Eliminar.** Borra la receta de la base de datos.

En el archivo **AndroidManifest.xml** se declara junto a un elemento `<intent-filter>`:

```
<activity
    android:name=".view.lista_recetas.RecetasActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Aloja varios elementos siendo los más relevantes:

## 1. SearchView

La **SearchView** es un componente que permite filtrar dinámicamente el contenido mostrado en el **RecyclerView**.

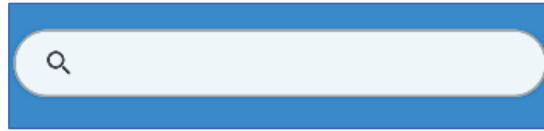


Figura 4. Elemento SearchView

Para configurar este elemento, se asigna un objeto **SearchView.OnQueryTextListener()** con el método **setOnQueryTextListener()**:

```
binding.svRecetas.setOnQueryTextListener(new SearchView.OnQueryTextListener() {  
    @Override  
    public boolean onQueryTextSubmit(String query) {  
        viewModel.setFiltroBusqueda(query.trim());  
        return true;  
    }  
  
    @Override  
    public boolean onQueryTextChange(String newText) {  
        viewModel.setFiltroBusqueda(newText.trim());  
        return true;  
    }  
});
```

Este **listener** nos obliga a sobrescribir dos métodos:

- **onQueryTextSubmit()** se ejecuta cuando el usuario envía la consulta.
- **onQueryTextChange()** se llama cada vez que el usuario modifica el texto de búsqueda.

En ambos casos, el texto introducido se asigna al **MutableLiveData filtroBusqueda** del **ViewModel**. Esto permite observar los cambios y actualizar el adaptador del **RecyclerView**, que se encargará de refrescar los elementos visibles mediante una llamada a **notifyDataSetChanged()**.

```
// Observamos los cambios al filtrar recetas  
viewModel.recetasFiltradas.observe(this, recetas -> {  
    adapter.actualizarDatos(recetas);  
});
```

Es importante devolver **true** en ambos métodos para indicar que el evento ha sido gestionado y evitar su propagación a otros componentes.

## 2. RecyclerView

Es un contenedor de listas que permite mostrar de forma eficiente una lista de objetos **Receta**.

En la **Activity** tenemos que configurar el **RecyclerView** asignando una instancia del **Adapter**.

```
adapter = new RecetasAdapter(new ArrayList<>(), getOnClickRecetaListener());  
binding.rvRecetas.setAdapter(adapter);
```

### ViewHolder

Cada ítem de la lista está representado por una instancia de la clase **RecetasViewHolder**. Este objeto actúa como contenedor de las referencias de las vistas definidas en el **layout** para cada ítem del **RecyclerView**.

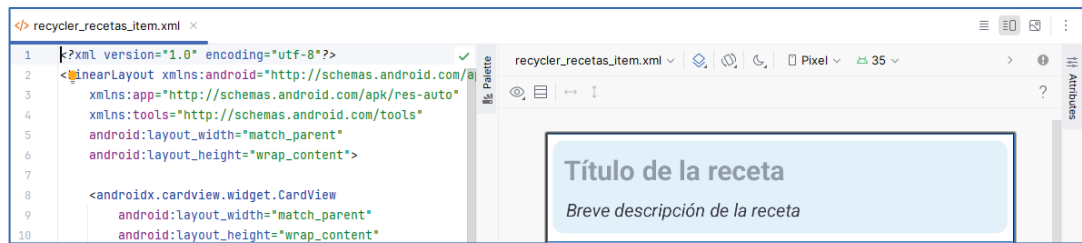


Figura 5. Layout del ítem del RecyclerView

Esta clase se encarga de vincular los datos del objeto **Receta** con las vistas del **ítem**. En mi caso, solo usamos dos **TextViews** **tvTituloReceta** y **tvDescripcion**.

Por lo tanto, esta clase conoce todos los elementos declarados en el layout **recycler\_recetas\_item.xml** y accedemos a ellos mediante **ViewBinding**:

```
private RecyclerViewRecetasItemBinding binding;
```

El **ViewHolder** debe reaccionar a los eventos de cada una de ellas, será en esta clase donde debemos configurar los listeners.

```
public RecetasViewHolder(@NonNull RecyclerViewRecetasItemBinding binding,  
    RecetasAdapter.OnClickItemClickListener listener) {  
    ...  
    binding.getRoot().setOnClickListener(v -> {  
        int position = getBindingAdapterPosition();  
  
        if (listener == null || position == RecyclerView.NO_POSITION) {return; }  
  
        listener.onClickReceta(position);  
    });  
}
```

## Adapter

La clase **RecetasAdapter** extiende de la clase abstracta **RecyclerView.Adapter** y declaramos las siguientes variables miembro:

```
private List<Receta> recetas;  
private OnClickListener listener;
```

Y serán instanciados en el constructor:

```
public RecetasAdapter(List<Receta> recetas, OnClickListener listener) {  
    this.recetas = recetas;  
    this.listener = listener;  
}
```

Además, nos obliga a implementar estos [3 métodos](#):

- **getItemCount(): RecyclerView** utiliza este método para conocer la cantidad de elementos que debe mostrar.
- **onCreateViewHolder(): RecyclerView** invoca a este método siempre que necesita crear un nuevo objeto **ViewHolder**. Y sólo lo inicializa y vincula las vistas.

A medida que el usuario se desplaza por la lista, los **ViewHolders** se reutilizan, por lo que rara vez se vuelve a llamar a **onCreateViewHolder()** a menos que se agote la caché o cambien los tipos de vista.

- **onBindViewHolder(): RecyclerView** llama a este método para asociar los datos del modelo con las vistas del **ViewHolder**.

Se llama cada vez que **RecyclerView** necesita mostrar un elemento, ya sea que el **ViewHolder** se haya creado recientemente o se haya reciclado al hacer un **scroll** en la lista.

En el siguiente [artículo](#) demuestran como al desplazarnos por una lista de 100 elementos, el método **onBindViewHolder()** se ejecuta **100 veces**, mientras que **onCreateViewHolder()** solo es llamado unas 15 veces, gracias al reciclaje.

El objetivo de esa demostración es concienciar a los programadores a no sobrecargar con operaciones costosas el método **onBindViewHolder()**.

Por este motivo, he refactorizado todos los **adaptadores** y **viewholders**, para alinear la implementación a las *mejores prácticas*.

### 3. BottomSheet

Al hacer clic en un ítem de la lista, se mostrará un **BottomSheet** con las opciones a realizar sobre el mismo: **Ver receta, modificar y eliminar**.

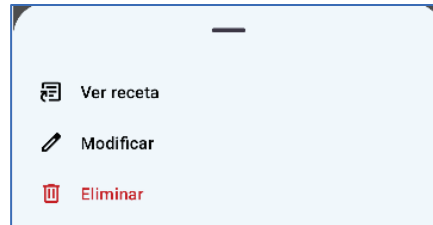


Figura 6. Opciones del BottomSheet

La clase **RecetasBottomSheet** extiende de **BottomSheetDialogFragment**, y definimos como miembros las siguientes variables:

```
private BottomsheetVerRecetasBinding binding;  
private OnClickOpcionListener listener;  
private RecetasViewModel viewModel;  
private Receta recetaSeleccionada;
```

En los eventos **clic** de cada elemento, propagamos los métodos del **listener**:

```
private void configurarListeners() {  
    binding.opcionVerReceta.setOnClickListeners(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            if (listener != null) listener.onVerReceta(recetaSeleccionada);  
            dismiss();  
        }  
    });  
  
    binding.opcionModificarReceta.setOnClickListeners(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            if (listener != null) listener.onModificarReceta(recetaSeleccionada);  
            dismiss();  
        }  
    });  
  
    binding.opcionEliminarReceta.setOnClickListeners(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            if (listener != null) listener.onEliminarReceta(recetaSeleccionada);  
            dismiss();  
        }  
    });  
}
```

Cada método invoca un método de la interfaz **OnClickOpcionListener**, para que la **Activity** sea la responsable de manejar las acciones.

```
public interface OnClickOpcionListener {  
    void onVerReceta(Receta receta);  
    void onModificarReceta(Receta receta);  
    void onEliminarReceta(Receta Receta);  
}
```

Android, prohíbe constructores con parámetros en **Fragments**, ya que al recrear el fragmento se rompe el ciclo de vida. Por este motivo, el **listener** será establecido mediante un método setter `setOnClickOpcionListener()`.

```
public void setOnClickOpcionListener(OnClickOpcionListener listener) {  
    this.listener = listener;  
}
```

En el método `onCreate()` del ciclo de vida del **Fragment** se realiza la inicialización del **ViewModel** compartido, para acceder a la receta seleccionada.

#### 4. FloatingActionButton

Al pulsar este objeto, abre **CrearRecetaActivity** para crear una **Receta**.



Figura 7. Botón FloatingActionButton

```
private void configurarFab() {  
    binding.fabCrearReceta.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            //Abrir activity para crear receta  
            Intent intent = new Intent(RecetasActivity.this,  
                                     CrearRecetaActivity.class);  
            startActivity(intent);  
        }  
    });  
}
```

A continuación, se muestra un ejemplo de cómo se visualiza esta **Activity**:

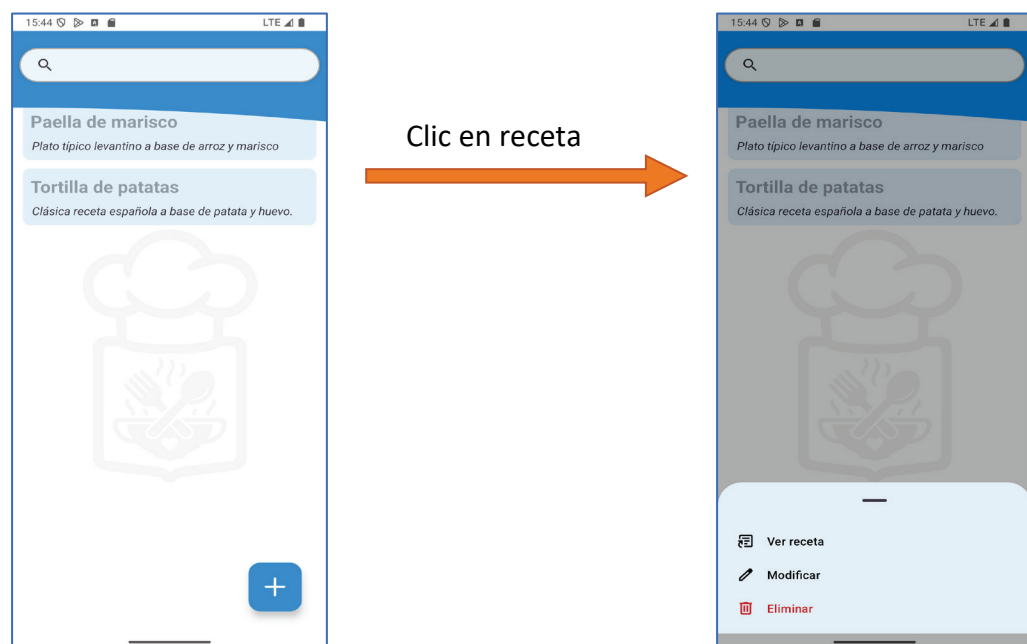


Figura 8. Ejemplo de visualización de RecetasActivity

### 5.2.2. VerRecetaActivity

Esta actividad permite visualizar una receta completa, incluyendo su **portada**, **ingredientes** y **pasos** de elaboración.

Para su implementación se han utilizado los componentes [Navigation](#), [ViewPager2](#) y [TabLayout](#), tal y como se muestra en el siguiente diagrama:

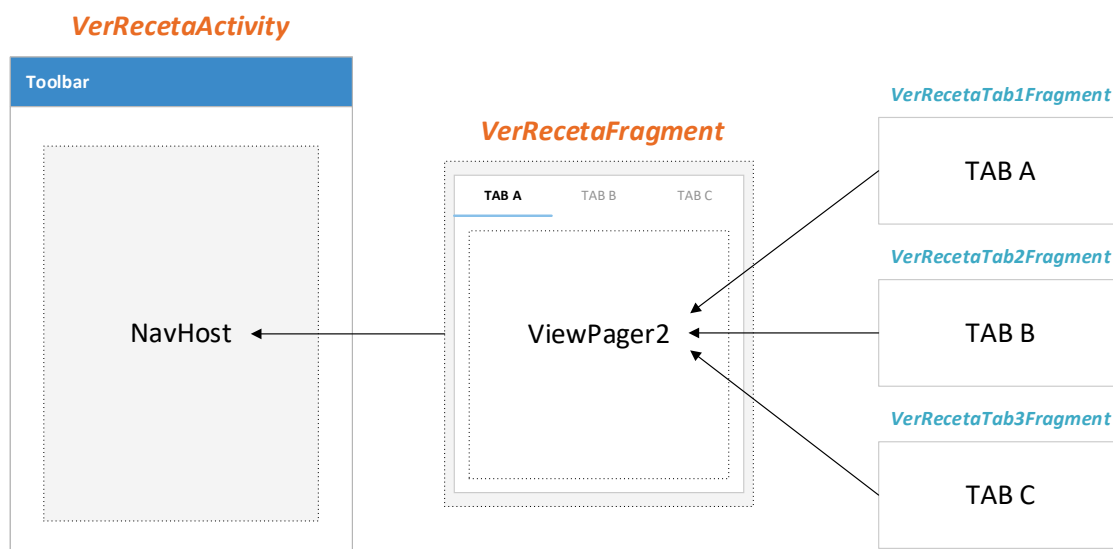


Figura 9. Diagrama de elementos gráficos de la actividad

El componente **Navigation** se compone de tres elementos principales:

- **Navigation Graph**. Es un recurso **XML** que define la estructura de navegación de la aplicación. En él se especifican los **destinos** (*Fragments* o *Activities*) y las rutas entre ellos, representadas mediante **acciones** (*flechas*).
- **NavController**. Es el objeto **Java** responsable de gestionar la navegación entre los distintos **destinos** o **Fragments**. Se encarga de mostrar, en el **NavHostFragment**, los destinos seleccionados por el usuario.
- **NavHostFragment**. Es un contenedor declarado en el **layout** de la **Activity** en donde se mostrarán los destinos definidos en el **Navigation Graph**. Es decir, es el área en la que el **NavController** irá cargando y reemplazando los diferentes **Fragments** durante la navegación.

Por lo tanto, el **layout** de la actividad debe incluir una **toolbar** y un **FragmentContainerView**, necesario para declarar el **NavHostFragment**.



En el atributo `app:navGraph` se indica el **grafo de navegación**, y para que el **NavController** gestione el botón *Up* (atrás) se establece `app:defaultNavHost="true"`:

```
<androidx.fragment.app.FragmentContainerView
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:id="@+id/nav_host_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/ver_receta_graph"
    tools:layout="@layout/fragment_ver_receta_tab1"/>
```

## TabLayout y ViewPager2

Para controlar la navegación entre fragmentos se ha utilizado el **ViewPager2**, que permite crear vistas deslizantes, junto con un **TabLayout** para mostrar pestañas en la parte superior.

Se crea un **Fragment** contenedor que implementa ambos componentes, denominado **VerRecetaFragment**.

Además, se define un **Fragment** por cada pestaña:

- *VerRecetaTab1Fragment*: controla las vistas de la portada de la receta.
- *VerRecetaTab2Fragment*: gestiona una vista de los ingredientes.
- *VerRecetaTab3Fragment*: permite visualizar los pasos de elaboración.

El **ViewPager2** requiere de un adaptador de tipo **FragmentStateAdapter**, que se encargará de crear los **Fragments** según la posición seleccionada.

Para vincular el **TabLayout** y el **ViewPager2** se utiliza un **TabLayoutMediator**, el cual recibe ambos objetos. En su último parámetro, se implementa la interfaz **TabConfigurationStrategy**, para sobrescribir el método `onConfigureTab()`, que nos permite establecer el título de cada pestaña en tiempo de ejecución.

## Grafo de navegación

Para crear el grafo de navegación nos situamos en la carpeta **res** del proyecto, y con el botón derecho del ratón, hacemos clic en: **New → Android Resource Directory**.

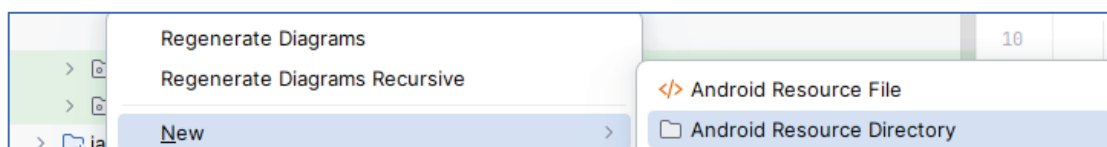


Figura 10. Añadir carpeta de recursos

Y seleccionamos **navigation** en **Resource type**:

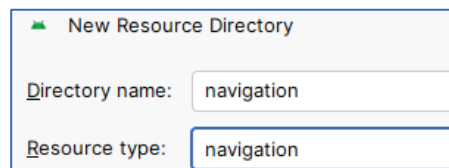


Figura 11. Carpeta Navigation

En la carpeta creada, hacemos clic con el botón derecho del ratón para crear un nuevo archivo de navegación, que se denominará: **ver\_receta\_graph**.

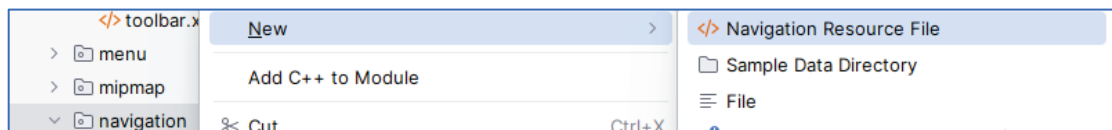


Figura 12. Archivo de navegación

En este grafo, únicamente incluimos el fragmento **VerRecetaFragment**, ya que los Fragments de cada pestaña serán gestionados directamente por el **ViewPager2**.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    app:startDestination="@id/verReceta_Fragment">
    <fragment
        android:id="@+id/verReceta_Fragment"
        android:name="org.overdrive.recetasdigitales.view.ver_receta.VerRecetaFragment"
        android:label="fragment_ver_receta"
        tools:layout="@layout/fragment_ver_receta" />
</navigation>
```

El atributo **app:startDestination** indica el **Fragment** inicial, y el atributo **android:name** especifica la clase **Java** que contiene el **ViewPager2**, ambos obligatorios.



Figura 13. ViewPager2+TabLayout

### 5.2.3. CrearRecetaActivity

Esta **Activity** permite al usuario crear una receta desde cero mediante un sistema de navegación de tipo **wizard** o asistente por pasos.

Aloja **tres Fragments** y utiliza el componente **Navigation** para gestionar la navegación entre ellos.

#### ViewModel

En el método `onCreate()`, se instancia el **ViewModel** compartido **CrearRecetaViewModel**, que almacena los datos introducidos en cada fragmento y garantiza su persistencia durante cambios de configuración de la **Activity**:

```
this.viewModel = new ViewModelProvider(this)
    .get(CrearRecetaViewModel.class);
```

#### Toolbar

Se utiliza una **Toolbar** común para todos los **Fragments**, lo que proporciona una interfaz coherente durante todo el proceso:

```
setSupportActionBar(binding.appBarLayout.toolbar);
```

La **Toolbar** se ha definido en un **layout** independiente para facilitar su reutilización:

```
<com.google.android.material.appbar.AppBarLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/appBarLayout"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />
</com.google.android.material.appbar.AppBarLayout>
```

La forma de reutilizar este diseño es mediante el elemento `<include>`:

```
<include
    android:id="@+id/appBarLayout"
    layout="@layout/toolbar" />
```

Es habitual ver este código dentro de un **CoordinatorLayout**, junto con **ConstraintLayout**, usando el atributo `app:layout_behavior`.

## Grafo de navegación

En esta **Activity**, la gestión de la navegación no recae sobre **ViewPager2**, sino en el **NavController**.

Lo primero es crear el grafo de navegación de forma similar a lo explicado en el apartado anterior. Sin embargo, en este caso tendremos que añadir todos los **Fragments** que conformarán el asistente.

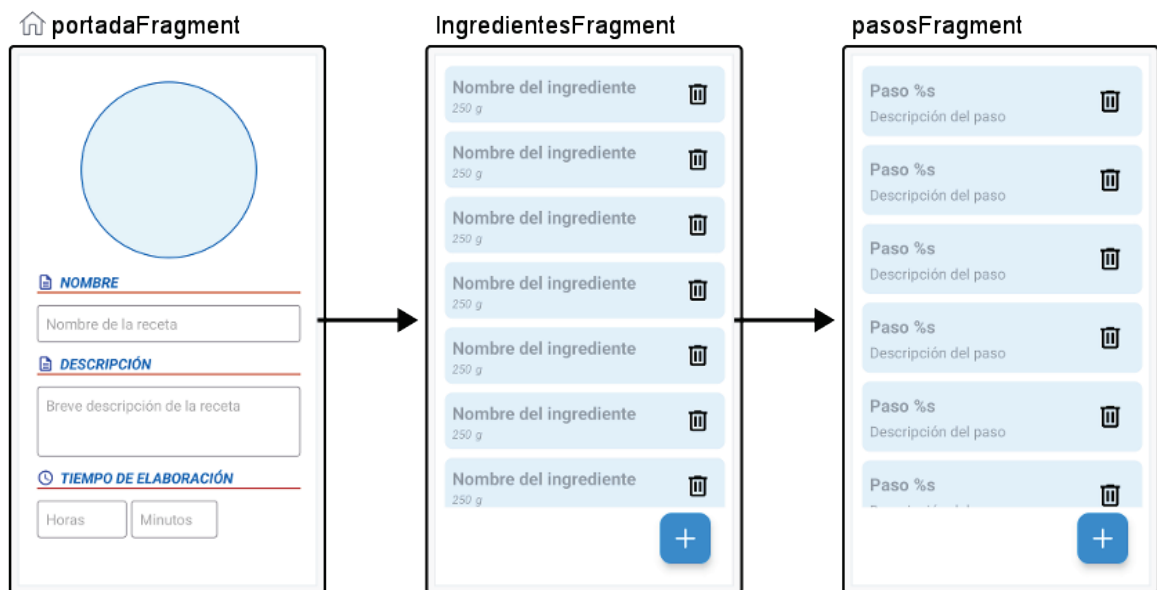


Figura 14. Previsualización de los destinos de navegación

La navegación entre pasos se define en un **grafo XML** donde cada **Fragment** aparece como un **destino**. Por ejemplo:

```
<fragment
    android:id="@+id/portadaFragment"
    android:name="org.overdrive.recetasdigitales.view.crear_receta.portada.PortadaFragment"
    android:label="Portada"
    tools:layout="@layout/fragment_portada">
    <action
        android:id="@+id/action_portada_a_ingredientes"
        app:destination="@id/IngredientesFragment" />
</fragment>
```

Para poder referenciar este objeto le asignamos un **android:id**, el atributo **android:name** indica la clase **Java** del **Fragment**, y **android:label** define el texto mostrado en la **Toolbar**.

Es recomendable personalizar el **android:id** de la **action**, ya que lo emplearemos para configurar la navegación con nombres más explícitos y abreviados. El atributo **app:destination** especifica el **Fragment** al que se debe navegar.

## Configuración del NavController

Al utilizar **FragmentManager**, la creación del **NavHostFragment** se realiza mediante una transacción interna del **FragmentManager**, la cual puede no haberse completado todavía cuando se ejecuta `onCreate()` de la **Activity**.

Por ese motivo, si se intenta obtener el **NavController** demasiado pronto, es posible obtener valores nulos o errores en tiempo de ejecución.

Para evitar este problema, **Android** recomienda recuperar primero la instancia del **NavHostFragment**, y después su **NavController**:

```
NavHostFragment navHostFragment = (NavHostFragment) getSupportFragmentManager()
    .findFragmentById(binding.navHostFragmentCrear.getId());

NavController navController = navHostFragment.getNavController();
```

## Vincular Toolbar con Navigation

La **Toolbar** se vincula con el sistema de navegación mediante **AppBarConfiguration** y **NavigationUI**. Esto le permite gestionar los eventos de navegación.

```
AppBarConfiguration appBarConfiguration = new AppBarConfiguration.Builder().build();
NavigationUI.setupActionBarWithNavController(this, navController, appBarConfiguration);
```

El primer fragmento deberá permitir cancelar la creación de la receta y volver a la **Activity** principal. Para implementar este comportamiento, se invoca al constructor **Builder()** sin especificar argumentos.

Al ejecutar la **Activity**, el fragmento inicial mostrará la flecha **Up** (←) que permitirá al usuario salir del asistente.

## Configurar botón Up

Para capturar el evento de retroceder en **CrearRecetaActivity**, y delegar la acción al controlador de navegación, **NavController**; tenemos que sobrescribir el método `onSupportNavigateUp()` de la clase **AppCompatActivity**.

```
@Override
public boolean onSupportNavigateUp() {
    NavDestination destination = navController.getCurrentDestination();

    if (destination != null && destination.getId() == R.id.portadaFragment) {
        new AlertDialog.Builder(this)
            .setTitle("Cancelar creación de receta")
            .setMessage("¿Deseas salir del asistente?")
            .setNegativeButton("No", null)
    }
```

```
        .setPositiveButton("Si", (dialog, which) -> finish())  
        .show();  
  
        return true; //Evitamos que se siga propagando el evento  
    }  
    return navController.navigateUp() || super.onSupportNavigateUp();  
}
```

Mediante un condicional, evaluamos si el actual destino es el fragmento **PortadaFragment**. Si es el caso, se mostrará un cuadro de dialogo para confirmar si quiere cancelar la creación de la receta o no. Es necesario devolver **true** para informar al sistema que se ha controlado el evento y no es necesario seguir propagándolo.

Y en cualquier otro caso, se delega el evento al **NavController**.

#### 5.2.3.1. PortadaFragment

Este **fragmento** solicita los datos básicos para crear la receta: **nombre**, **descripción**, **imagen** y **tiempo** de elaboración.

### Personalización de la Toolbar

Para añadir un botón *Siguiente* se ha creado un **archivo XML** de **menú**. El diseño es muy sencillo, contiene un único item que muestra el icono >:

```
<?xml version="1.0" encoding="utf-8"?>  
<menu xmlns:android="http://schemas.android.com/apk/res/android"  
      xmlns:app="http://schemas.android.com/apk/res-auto">  
    <item  
        android:id="@+id/action_siguiente"  
        android:enabled="true"  
        android:icon="@drawable/outline_navigate_next_36"  
        android:title="@string/siguiente"  
        android:visible="true"  
        app:iconTint="?attr/colorOnPrimary"  
        app:showAsAction="always|withText" />  
    </item>  
</menu>
```

Este icono se muestra siempre, y en el caso de disponer de espacio suficiente, también muestra el texto del atributo **android:title**.



Figura 15. Toolbar para el fragmento Portada

En el Fragment, se utiliza **MenuProvider**, que es la forma recomendada de personalizar la **Toolbar** en lugar de **setHasOptionsMenu(true)**, ya que está mejor integrado con el ciclo de vida.

La implementación sería la siguiente:

```
requireActivity().addMenuProvider(new MenuProvider() {  
  
    @Override  
    public void onCreateMenu(@NonNull Menu menu,  
                             @NonNull MenuInflater menuInflater) {  
        menuInflater.inflate(R.menu.menu_siguiente, menu);  
    }  
  
    @Override  
    public boolean onMenuItemSelected(@NonNull MenuItem menuItem) {  
  
        if (menuItem.getItemId() == R.id.action_siguiente) {  
  
            // Validar nombre obligatorio  
            if (binding.etNombreReceta.getText().toString().trim().isEmpty()) {  
                binding.etNombreReceta.setError("El nombre de la receta " +  
                                                  "es obligatorio");  
                binding.etNombreReceta.requestFocus();  
                return true;  
            }  
  
            if (contenidoCorrecto()) {  
                setRecetaViewModel();  
                navController.navigate(R.id.action_portada_a_ingredientes);  
                return true; // Le dice que ha consumido el evento  
            }  
        }  
        return false;  
    }  
}, getViewLifecycleOwner(), Lifecycle.State.RESUMED);
```

La interfaz **MenuProvider** obliga a implementar:

- `onCreateMenu()`: infla el layout del menu.
- `onMenuItemSelected()`: reacciona al evento de hacer clic sobre un item del menú. En este método, se ha implementado una *simple validación de datos*, ya que lo único que **no** puede estar **vacío** es el **nombre** de la receta.

Además, se evalúa que los **EditText** no tengan ningún error. Si todo es correcto, creamos un objeto **Receta** que lo añadimos al **ViewModel** y navegamos al siguiente **Fragment**.

### Configuración de TextWatchers

Los **TextWatchers** se han empleado para validar dinámicamente los campos que almacenan el tiempo de elaboración (**horas y minutos**). Cada **EditText** registra un **TextWatcherSimple**, una clase que extiende **TextWatcher** y sobrescribe únicamente el método `afterTextChanged()`, evitando código innecesario.



Para confirmar los valores introducidos, se implementa el método genérico `validarTiempo()`, que evalúa si el número se encuentra dentro de un rango:

```
private boolean validarTiempo(int tiempo, int min, int max) {  
    return tiempo >= min && tiempo <= max;  
}
```

Si el valor es incorrecto, se establece un error en el **EditText** para informar al usuario y así impedir la navegación al siguiente **Fragment**.

```
binding.etHoras.addTextChangedListener(new TextWatcherSimple() {  
    @Override  
    public void afterTextChanged(android.text.Editable s) {  
  
        horas = StringToInt(s.toString(), 0);  
  
        if (!validarTiempo(horas, horaMin, horaMax)) {  
            binding.etHoras.setError("Valor entre " + horaMin + " y " + horaMax);  
            binding.etHoras.requestFocus();  
        } else {  
            binding.etHoras.setError(null);  
        }  
    }  
});  
  
binding.etMinutos.addTextChangedListener(new TextWatcherSimple() {  
  
    @Override  
    public void afterTextChanged(android.text.Editable s) {  
  
        minutos = StringToInt(s.toString(), 0);  
  
        if (!validarTiempo(minutos, minutoMin, minutoMax)) {  
            binding.etMinutos.setError("Valor entre " + minutoMin +  
                " y " + minutoMax);  
            binding.etMinutos.requestFocus();  
        } else {  
            binding.etMinutos.setError(null);  
        }  
    }  
});
```

## Configurar listeners

Se establece un `OnClickListener` en el **ImageView** del fragmento para permitir que el usuario seleccione una imagen para la receta.

```
binding.ivImagenReceta.setOnClickListener(v -> {  
    lanzadorImagen.launch("image/*");  
});
```

Siguiendo las mejores prácticas, se utiliza la **API** de [Activity Result](#), que proporciona componentes para **registrar un resultado**, **iniciar la actividad** que produce el resultado, y **manejar el resultado** una vez son enviados por el sistema.

Se declara un miembro de clase del tipo **ActivityResultLauncher<String>**, que se inicializa mediante el método `registerForActivityResult()`, el cual requiere **dos parámetros**:

### 1. Contract (contrato)

Define el tipo de dato de **entrada** y **salida**. En este caso, `ActivityResultContracts.GetContent()`, que recibe un **String** y devuelve un objeto **Uri**.

### 2. Callback

Se implementa la lógica que maneja el resultado devuelto. Aquí se asigna la **URI** a una variable **MutableLiveData<Uri>** del **ViewModel**.

El objetivo es almacenar temporalmente la **URI** temporal de acceso a la imagen seleccionada. Durante el guardado definitivo, la imagen se copia al directorio interno de la aplicación y la nueva **URI** será la que se guarde en la base de datos.

```
// Launcher para obtener la imagen
private ActivityResultLauncher<String> lanzadorImagen = registerForActivityResult(
    new ActivityResultContracts.GetContent(),
    new ActivityResultCallback<Uri>() {
        @Override
        public void onActivityResult(Uri uri) {

            if (uri != null) {
                // Guardar URI en ViewModel
                viewModel.setImagenUri(uri);
                //La imagen se almacena cuando se guarde la receta
            }
        }
    }
);
```

**NOTA:** La instancia de la `ActivityResultLauncher` debe **registrarse antes** de que la actividad alcance el estado **STARTED**. Lo recomendado es registrarlo en el método `onCreate()`, aunque también podría hacerse en `onStart()`, lo que evitaría problemas derivados del ciclo de vida.

## Configurar observers

Los observadores permiten que la interfaz gráfica de la aplicación sea **reactiva**, es decir, que se actualice automáticamente cuando cambian los datos gestionados por el **ViewModel**. Nos permite sincronizar las vistas con los datos actuales cuando se reanude la **Activity** o **Fragment**.

En primer lugar, se observa la variable **MutableLiveData<Receta>**, que almacena los datos principales de la receta. Cuando el **ViewModel** modifica su contenido, se lo notifica al observador y se actualizan las vistas del fragmento.



Figura 16. Diagrama de componentes MVVM

También se observa la **URI** de la imagen seleccionada. Cuando cambia, al seleccionar una imagen de la galería, se actualiza el **ImageView** mediante **Glide**:

```
viewModel.getReceta().observe(getViewLifecycleOwner(), receta -> {
    binding.etNombreReceta.setText(receta.getTitulo());
    binding.etDescripcionReceta.setText(receta.getDescripcion());

    int horas = GestorTiempo.getHoras(receta.getTiempo());
    if (horas != 0) {
        binding.etHoras.setText(String.valueOf(horas));
    }

    int minutos = GestorTiempo.getMinutos(receta.getTiempo());
    if (minutos != 0) {
        binding.etMinutos.setText(String.valueOf(minutos));
    }

    this.horas = horas;
    this.minutos = minutos;
});

viewModel.getImagenUri().observe(getViewLifecycleOwner(), uri -> {
    if (uri != null) {
        Glide.with(requireContext())
            .load(uri)
            .into(binding.ivImagenReceta);
    }
});
```

Se ha creado la clase **GestorTiempo** que permite encapsular las operaciones de conversión de datos a partir de un dato de tipo **long**.

```
public class GestorTiempo {
    private final long horas;
    private final long minutos;
    ...

    public static int getHoras(long tiempoEnMinutos) {
        return (int) tiempoEnMinutos / 60;
    }

    public static int getMinutos(long tiempoEnMinutos) {
        return (int) tiempoEnMinutos % 60;
    }
    ...
}
```

### 5.2.3.2. IngredientesFragment

Este fragmento constituye el segundo paso del asistente para crear una receta. Permite al usuario **añadir, modificar y eliminar ingredientes** de una lista.

Gestiona varios elementos, siendo los más relevantes los siguientes:

#### Personalización de la Toolbar

Para añadir un botón *Siguiente*, se reutiliza el **archivo XML** de **menú** descrito en el fragmento anterior.

En el método `onViewCreated()` se recupera el **NavController**, y se declara un **MenuProvider** para gestionar la navegación hacia el último fragmento:

```
private void configurarMenuProvider() {
    requireActivity().addMenuProvider(new MenuProvider() {

        @Override
        public void onCreateMenu(@NonNull Menu menu,
                                @NonNull MenuInflater menuInflater) {
            menuInflater.inflate(R.menu.menu_siguiente, menu);
        }

        @Override
        public boolean onMenuItemSelected(@NonNull MenuItem menuItem) {

            if (menuItem.getItemId() == R.id.action_siguiente) {
                navController.navigate(R.id.action_ingredientes_a_pasos);
                return true;
            }
            return false;
        }
    }, getViewLifecycleOwner(), Lifecycle.State.RESUMED);
}
```

#### Configurar RecyclerView

La configuración del **RecyclerView** requiere un **Adapter** (*IngredientesAdapter*) y un **ViewHolder** (*IngredientesViewHolder*).

La implementación de ambos elementos es, en esencia, la misma que la realizada en **ListarRecetasActivity**. Por ello, en esta sección únicamente las partes específicas que afectan a este fragmento.

El **ViewHolder** vincula los datos de cada objeto **Ingrediente**, proporcionados por el **Adapter**, con las vistas que componen cada item del **RecyclerView**.

Además, define los **listeners** que afectan al propio **item** como al **ImageButton** que permite eliminar un ingrediente.

En ambos casos, se propaga la posición del **item** a través de los métodos definidos en la interfaz **OnClickIngredienteListener**, encapsulada en el **Adapter**.

```
public class IngredientesViewHolder extends RecyclerView.ViewHolder {
    ...
    itemView.setOnClickListener(v -> {
        int posicion = getBindingAdapterPosition();
        if (posicion != RecyclerView.NO_POSITION) {
            listener.onClickIngrediente(posicion);
        }
    });

    binding.ibBorrarIngredienteItem.setOnClickListener(v -> {
        int posicion = getBindingAdapterPosition();
        if (posicion != RecyclerView.NO_POSITION) {
            listener.onEliminarIngrediente(posicion);
        }
    });
    ...
}

public class IngredientesAdapter extends
    RecyclerView.Adapter<IngredientesViewHolder> {
    ...
    public interface OnClickIngredienteListener {
        void onClickIngrediente(int posicion);
        void onEliminarIngrediente(int posicion);
    }
}
```

El **Fragment** implementará los métodos de esta interfaz al instanciar el objeto **IngredientesAdapter**. Si el usuario hace **clik** en el **item**, se abrirá un **BottomSheet** con los datos del **Ingrediente**, recuperados del **ViewModel**.

Si hace **clik** en el botón de **eliminar**, se mostrará un dialogo de confirmación, y dependiendo de la elección, se eliminará el ingrediente o se cancelará la acción.

```
adapter = new IngredientesAdapter(new ArrayList<>(),
    new IngredientesAdapter.OnClickIngredienteListener() {

        @Override
        public void onClickIngrediente(int position) {

            viewModel.setPosicionIngredienteEditando(position);
            IngredientesBottomSheet bottomSheet = new IngredientesBottomSheet();
            bottomSheet.show(IngredientesFragment.this.getParentFragmentManager(),
                IngredientesBottomSheet.TAG);
        }

        @Override
        public void onEliminarIngrediente(int posicion) {
            new AlertDialog.Builder(requireContext())
                .setTitle("Eliminar")
                .setMessage("¿Está seguro de borrar el ingrediente?")
                .setNegativeButton("No", null)
                .setPositiveButton("Si", (dialog, which) ->
                    viewModel.eliminarIngrediente(posicion))
                .show();
        }
    });
```

## Configurar observers

En este fragmento se observa la lista de ingredientes para pasársela al **adapter** mediante un método de actualización de datos, en el que se notifican los cambios con `notifyDataSetChanged()`.

```
viewModel.getIngredientes().observe(getViewLifecycleOwner(), ingredientes -> {  
    if (ingredientes != null) {  
        adapter.actualizarDatos(ingredientes);  
    }  
});
```

## Configurar FloatingActionButton

Al pulsar este botón flotante, se crea una instancia de **IngredientesBottomSheet**:

```
private void configurarFab() {  
    binding.fabIngredientes.setOnClickListener(  
        new View.OnClickListener() {  
            @Override  
            public void onClick(View view) {  
                IngredientesBottomSheet bottomSheet = new IngredientesBottomSheet();  
                bottomSheet.show(getParentFragmentManager(),  
                    IngredientesBottomSheet.TAG);  
            }  
        });  
}
```

## Configurar BottomSheet

La clase **IngredientesBottomSheet**, en su método `onCreate()`, establece el estilo, inicializa **CrearRecetaViewModel**, y configura la ventana de dialogo como **modal**.

```
setCancelable(false);
```

A diferencia de **RecetasBottomSheet**, no permite cerrar la ventana sin hacer clic en alguno de los botones declarados en su **layout**, **aceptar** o **cancelar**.



Figura 17. Diseño IngredientesBottomSheet

Para controlar la elección del usuario, se añade a cada botón un **listener** que gestiona el evento **OnClick**.

El botón **Cancelar** cierra el fragmento, momento en el que se dispara el método `onDismiss()`, donde se resetean tanto el flag de control **esEdicion** como la variable **posicionIngredienteEditando** del **CrearRecetaViewModel**:

```
@Override
public void onDismiss(@NonNull DialogInterface dialog) {
    super.onDismiss(dialog);

    // Resetear estado de edición
    viewModel.setPosicionIngredienteEditando(-1);
    esEdicion = false;
}
```

La variable **esEdicion** nos permite dividir el flujo de ejecución del **BottomSheet**. Si se trata de un ingrediente nuevo, las vistas del **layout** no mostrarán datos.

Si la variable **posicionIngredienteEditando** contiene una posición válida, el **BottomSheet** cargará los datos del ingrediente correspondiente para su modificación.

El botón **Aceptar** realiza una validación del campo **nombre**, que es requerido. Además, utiliza el flag **esEdicion** para determinar si debe invocar a `actualizarIngrediente()` o `agregarIngrediente()`.

La actualización del ingrediente implica llamar al método del **ViewModel** `actualizarIngrediente(ingredienteEditando)`, el cual crea una nueva lista y la establece en el **LiveData** de la lista de ingredientes, provocando así la notificación automática a los observadores del **IngredientesFragment**.

La creación de un nuevo ingrediente lo añade directamente a la lista del **ViewModel**, lo que igualmente notifica a los observadores del fragmento.

Finalmente, esta clase observa la variable **posicionIngredienteEditando** del **ViewModel**, que determina si el modo actual es de edición y establece el valor del flag de control. En la implementación del observador se realiza además la vinculación de los datos del ingrediente con las vistas del **BottomSheet**.

Se ha configurado un campo de autocompletado para las unidades. Aunque es un campo libre, sugiere las más habituales mediante un `string-array`.



### 5.2.3.3. PasosFragment

Este fragmento es el último paso del asistente para crear una receta, y permite al usuario **añadir, modificar y eliminar** pasos dentro de una lista.

Dado que la entidad **Paso** tiene un campo **orden**, la lista debe tener en cuenta dicho valor. Por este motivo, se ha implementado la capacidad de **reordenar los ítems** del **RecyclerView**, aplicando un **reindexado automático** tras cada movimiento.

### Personalización de la Toolbar

Para añadir un botón *Guardar* se crea un nuevo **archivo XML de menú**.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_guardar"
        android:title="@string/guardar"
        android:icon="@drawable/outline_save_36"
        app:iconTint="?attr/colorOnPrimary"
        app:showAsAction="always|withText" />
</menu>
```

En el método `onViewCreated` se recupera el **NavController**, y se declara un **MenuProvider** para iniciar el proceso de persistencia de la receta completa:

```
private void configurarMenuProvider() {
    requireActivity().addMenuProvider(new MenuProvider() {

        @Override
        public void onCreateMenu(@NonNull Menu menu,
                                @NonNull MenuInflater menuInflater) {
            menuInflater.inflate(R.menu.menu_guardar, menu);
        }

        @Override
        public boolean onMenuItemSelected(@NonNull MenuItem menuItem) {
            // Item: guardar
            if (menuItem.getItemId() == R.id.action_guardar) {
                guardarReceta();
                return true;
            }
            return false;
        }
    }, getViewLifecycleOwner(), Lifecycle.State.RESUMED);
}
```

### Configurar RecyclerView

La configuración del **RecyclerView** requiere de un **Adapter** (*PasosAdapter*) y un **ViewHolder** (*PasosViewHolder*). La estructura de ambos elementos es prácticamente la misma que la implementada en **IngredientesFragment**.

Cuando el usuario hace clic sobre un ítem, se establece la posición del paso que se va a editar en la variable **posicionPasoEditando** del **ViewModel**, y a continuación se muestra una instancia de **PasosBottomSheet**.

Si el usuario pulsa sobre el **ImageButton** de eliminación del ítem, se muestra un diálogo de confirmación que permite eliminar el paso o cancelar la acción.

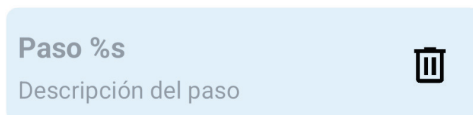


Figura 18. Item de un paso

### Configurar observers

En este fragmento se observa la lista de pasos, para asignarla al **adaptador** mediante un método de actualización, que notifica los cambios al adaptador mediante `notifyDataSetChanged()`.

```
viewModel.getPasos().observe(getViewLifecycleOwner(), pasos -> {  
    if (pasos != null) {  
        adapter.actualizarDatos(pasos);  
    }  
});
```

Al tratarse del último fragmento del asistente, aquí se realiza la persistencia de la receta completa en la base de datos **Room**. Para ello, se declara un segundo observador sobre el método `getRecetaGuardada()` del **CrearRecetaViewModel**:

```
viewModel.getRecetaGuardada().observe(getViewLifecycleOwner(), guardada -> {  
    if (Boolean.TRUE.equals(guardada)) {  
        Toast.makeText(getContext(), "Receta guardada", Toast.LENGTH_SHORT).show();  
        requireActivity().finish();  
    }  
});
```

Cuando se notifica que la receta ha sido guardada, la **Activity** se cierra, regresando a la actividad principal de la aplicación. De este modo, se espera a que Room finalice su proceso y se disparen los eventos de notificación que permiten visualizar, sin retardos, la nueva receta en el **RecyclerView** de **VerRecetasActivity**.

En el **ViewModel** se realiza una llamada a un método distinto del **Repositorio**, dependiendo de si se está creando o modificando la receta. En ambos métodos se ha implementado un **callback** que establece en true la variable **recetaGuardada**, la cual está siendo observada en **PasosFragment**.

## Configurar FloatingActionButton

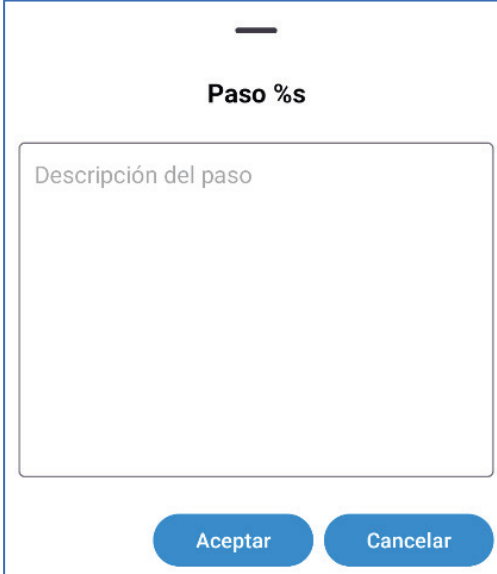
Al pulsar el botón flotante, se crea una instancia del **PasosBottomSheet**:

```
private void configurarFab() {  
    binding.fabPasos.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
  
            PasosBottomSheet bottomSheet = new PasosBottomSheet();  
            bottomSheet.show(getParentFragmentManager(), PasosBottomSheet.TAG);  
        }  
    });  
}
```

## Configurar BottomSheet

La clase **PasosBottomSheet**, en su método `onCreate()`, establece el estilo, inicializa **CrearRecetaViewModel**, y configura la ventana como **modal**.

A diferencia de **RecetasBottomSheet**, no permite cerrar la ventana sin hacer clic en alguno de los botones declarados en su **layout**, **aceptar** o **cancelar**.



La imagen muestra un diseño de interfaz de usuario para un BottomSheet. En la parte superior, hay un título "Paso %s". Debajo del título, hay un campo de texto con el placeholder "Descripción del paso". En la parte inferior, hay dos botones azules con el texto "Aceptar" y "Cancelar".

Figura 19. Diseño PasosBottomSheet

La estructura de implementación de esta clase es prácticamente idéntica a la descrita en **IngredientesBottomSheet**.

La gestión de los flujos de creación y edición del **Paso** utiliza la misma estrategia: conservar en el **ViewModel** la posición del **paso** a editar (**posicionPasoEditando**) con un observador que impide agregar un paso vacío y vincula los datos del objeto **Paso** con las vistas del **BottomSheet** para su edición.

## Configurar TouchHelper

Para permitir al usuario **ordenar ítems del RecyclerView** mediante **arrastre táctil**, se ha creado la clase **PasosItemTouchHelper**, que extiende de [ItemTouchHelper.Callback](#).

```
private void configurarTouchHelper() {  
    touchCallback = new PasosItemTouchHelper(  
        new PasosItemTouchHelper.OrdenarPasos() {  
            @Override  
            public void mover(int fromPos, int toPos) {  
                viewModel.moverPaso(fromPos, toPos);  
            }  
        });  
  
    touchHelper = new ItemTouchHelper(touchCallback);  
    touchHelper.attachToRecyclerView(binding.rvPasos);  
}
```

La vinculación con el **RecyclerView** se realiza mediante `attachToRecyclerView()`.

La clase **PasosItemTouchHelper** define la interfaz **OrdenarPasos**, con el método `mover()` que es implementado en **PasosFragment**:

```
public interface OrdenarPasos {  
    void mover(int fromPos, int toPos);  
}
```

En el fragmento, la implementación del **listener** propaga al **ViewModel** el índice original y el nuevo índice del ítem en el **Adapter**. De este modo, se modifica la lista de **CrearRecetaViewModel** con el orden actualizado de los elementos.

Los métodos obligatorios a implementar son:

- `getMovementFlags`: mediante diversas banderas se especifica qué movimientos están permitidos para los ítems del **RecyclerView**, tanto en arrastre como en deslizamiento lateral.

La construcción de los valores se realiza mediante `makeMovementFlags()`, donde el primer parámetro define los movimientos de arrastre, y el segundo los de deslizamiento.

```
@Override  
public int getMovementFlags(@NonNull RecyclerView rv, @NonNull  
    RecyclerView.ViewHolder vh) {  
  
    return makeMovementFlags(ItemTouchHelper.UP | ItemTouchHelper.DOWN, 0);  
}
```

En este caso, se permite el movimiento vertical para arrastrar y soltar.

- `onMove`:

Este método se ejecuta cuando se mueve un item. Permite determinar la posición inicial y las posiciones intermedias durante el desplazamiento:

```
@Override
public boolean onMove(@NonNull RecyclerView rv,
                      @NonNull RecyclerView.ViewHolder viewHolder,
                      @NonNull RecyclerView.ViewHolder target) {

    int from = viewHolder.getBindingAdapterPosition();
    int to = target.getBindingAdapterPosition();

    if (from == RecyclerView.NO_POSITION || to == RecyclerView.NO_POSITION)
        return false;

    rv.getAdapter().notifyItemMoved(from, to);

    if (posicionInicial == RecyclerView.NO_POSITION) posicionInicial =
        from;

    posicionFinal = to;

    return true;
}
```

Este método ofrece acceso a los **ViewHolder** para actualizar el adaptador y visualizar los cambios de la lista en tiempo real.

Devuelve `true` para indicar que el evento ha sido manejado.

- `onSwiped`: este método no se implementa, ya que no se permite el desplazamiento lateral.

Además de los métodos anteriores, se definen los siguientes:

- `isLongPressDragEnabled`: habilita la pulsación larga para iniciar el movimiento, devolviendo `true`.
- `isItemViewSwipeEnabled`: deshabilita el deslizamiento lateral con `false`.
- `onSelectedChanged`: este método permite cambiar color de fondo del item cuando el usuario lo mantiene pulsado:

```
@Override
public void onSelectedChanged(RecyclerView.ViewHolder viewHolder,
                              int actionState) {
    super.onSelectedChanged(viewHolder, actionState);

    if (actionState == ItemTouchHelper.ACTION_STATE_DRAG &&
        viewHolder != null) {
        viewHolder.itemView.setActivated(true); // Aplica color de arrastre
    }
}
```

Los valores de color se han definido en un archivo **XML** de tipo **selector**:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- Estado arrastrando (activated=true) -->
    <item android:color="@color/dragCardView"
          android:state_pressed="true" />

    <item android:color="@color/dragCardView"
          android:state_activated="true" />

    <!-- Estado normal -->
    <item android:color="@color/fondoCardView" />
</selector>
```

- **clearView**: se ejecuta cuando el usuario deja de pulsar el ítem y se finaliza el movimiento. En este instante, se restablece el estado visual del ítem y se evalúan las posiciones inicial y final del movimiento.

Si las ambas son válidas, se notifica el cambio al **ViewModel**, y se reinicia la variable de control **posicionInicial**.

```
@Override
public void clearView(@NonNull RecyclerView rv, @NonNull
RecyclerView.ViewHolder vh) {
    super.clearView(rv, vh);

    vh.itemView.setActivated(false); // Restablece estado al soltar

    // Si las posiciones son válidas
    if (posicionInicial != RecyclerView.NO_POSITION &&
        posicionFinal != RecyclerView.NO_POSITION &&
        posicionInicial != posicionFinal) {

        // Comunicamos al viewModel que debe efectuar los cambios
        listener.mover(posicionInicial, posicionFinal);
    }

    // Reinicializamos las posiciones para el siguiente movimiento.
    posicionInicial = RecyclerView.NO_POSITION;
}
```

#### 5.2.4. EditarRecetaActivity

Se emplea la herencia para reutilizar tanto la **Activity** como los **Fragments** diseñados para la creación de recetas.

```
public class EditarRecetaActivity extends CrearRecetaActivity
```

En su método **onCreate()**, se comprueba si el **Intent** recibido contiene el identificador de una receta. En tal caso, se invoca al método **cargarRecetaParaEditar()** del **CrearRecetaViewModel**, solicita al repositorio los datos de la receta y actualiza las variables que notificarán los cambios a los observadores.

### 5.3. ViewModel

El **ViewModel** actúa como intermediario entre la vista y el modelo, desacoplando la lógica de negocio de la interfaz de usuario al exponer los datos de forma observable.

En el patrón **MVVM**, las clases del **ViewModel** mantienen una instancia del **Repositorio**, a través del cual solicitan los datos a la base de datos. Esta capa no gestiona directamente hilos de ejecución, aunque sí resulta idónea para transformar datos mediante utilidades como **Transformation.SwitchMap()**, ya que administra objetos encapsulados en **LiveData** de manera segura y reactiva.

#### 5.3.1. VerRecetaViewModel

En el constructor se obtiene una referencia al repositorio de la aplicación mediante:

```
repo = new RecetarioRepositorio(application);
```

Si el módulo de estado guardado (**SavedStateHandle**) conserva una referencia al identificador de una receta, el **ViewModel** se encarga de recuperar sus datos:

```
long idReceta = savedStateHandle.get(RECETA_ID);  
recetaCompleta = repo.getRecetaCompleta(idReceta);
```

De este modo, el **ViewModel** puede restaurar un estado previo ante cambios de configuración u otras circunstancias que provoquen la recreación de la actividad o los fragmentos.

Además, se define el método `init(recetaId)`, invocado desde la **Activity**, con el fin de inicializar el **ViewModel** y asociar un estado recuperable al ciclo de vida:

```
public void init(long recetaId) {  
    if (recetaCompleta != null) {  
        return;  
    }  
  
    savedStateHandle.set(RECETA_ID, recetaId);  
    recetaCompleta = repo.getRecetaCompleta(recetaId);  
}
```

El método `getRecetaCompleta()` devuelve un objeto **LiveData<RecetaCompleta>**, que es observado desde los fragmentos que muestran los apartados de la receta. Esto permite que cada fragmento actualice automáticamente sus vistas si se producen cambios en los datos.

### 5.3.2. RecetasViewModel

Este **ViewModel** se encarga de comunicar los cambios de datos al **RecyclerView** de **RecetasActivity**. En su constructor, se utiliza **Transformations.switchMap()**, que permite obtener un nuevo **LiveData** a partir de otro mediante una **función** de conversión.

En esta implementación, **switchMap()** se utiliza para filtrar el listado de recetas según el texto que el usuario introduce en el **SearchView**.

```
public RecetasViewModel(@NonNull Application application) {  
    ...  
    recetasFiltradas = Transformations.switchMap(filtroBusqueda,  
                                                new Function1<String, LiveData<List<Receta>>>() {  
            @Override  
            public LiveData<List<Receta>> invoke(String texto) {  
                return repo.buscarPorTitulo(texto);  
            }  
        }  
    );  
}
```

Cada vez que cambia el valor del **LiveData filtroBusqueda**, se ejecuta la función definida, que solicita al repositorio las recetas con título coincidente.

Por último, se añaden métodos **getter** y **setter** de los miembros de clase, lo que permite observar los cambios en los datos y actualizar la interfaz.

### 5.3.3. CrearRecetaViewModel

Este **ViewModel** es utilizado por **CrearRecetaActivity** y los fragmentos que esta contiene, así como por **EditarRecetaActivity** mediante herencia.

La clase declara diversas variables **LiveData**, como: **receta**, lista de **ingredientes** y **pasos**. Son los elementos principales que permiten actualizar, de forma reactiva, las vistas de los fragmentos de **CrearRecetaActivity**.

La variable **imagenUriTemporal**, almacena la **URI** de la imagen seleccionada por el usuario, que puede cambiar hasta que la receta completa no se persista.

Las variables **posicionIngredienteEditando** y **posicionPasoEditando** permiten a los **BottomSheet** identificar si se ha hecho clic en un ítem y mostrar sus datos para que el usuario los pueda modificar.



La variable **idRecetaEditando**, es un flag de control, almacena el identificador de la receta seleccionada para modificar, de modo que pueda cargarse en los fragmentos de **EditarRecetaActivity**.

Se han declarado los métodos **setter** y **getter** de los miembros de la clase, así como métodos específicos para **agregar**, **actualizar** y **eliminar** elementos (ingredientes o pasos).

En dichos métodos, se crea siempre una nueva lista a partir de la existente, asegurando que **LiveData** notifique los cambios a los observadores cuando se vincule la nueva lista mediante el método `setValue()`. Por ejemplo:

```
public void setIngrediente(Ingrediente ingrediente) {  
    List<Ingrediente> lista = new ArrayList<>(ingredientes.getValue());  
    lista.add(ingrediente);  
    ingredientes.setValue(lista);  
}
```

Para finalizar, se han implementado varios métodos que gestionan las llamadas al repositorio para persistir la información en **Room**.

En el método `guardarRecetaCompleta()`, empleamos la variable **idRecetaEditando** para determinar si debemos persistir o reemplazar la receta completa. Se realizan llamadas al **Repositorio** con los datos necesarios para persistir los datos. El último parámetro de estas llamadas es un **callback** que nos permite retener el cierre de la **Activity** hasta que la persistencia haya finalizado.

```
public void guardarRecetaCompleta() {  
    Receta r = receta.getValue();  
    List<Ingrediente> ing = ingredientes.getValue();  
    List<Paso> ps = pasos.getValue();  
    Uri uri = imagenUriTemporal.getValue();  
  
    if (idRecetaEditando == null) {  
        repo.insertarRecetaCompleta(r, ing, ps, uri, () -> {  
            recetaGuardada.postValue(true);  
        });  
    } else {  
        repo.reemplazarRecetaCompleta(idRecetaEditando, r, ing, ps, uri, () -> {  
            recetaGuardada.postValue(true);  
        });  
    }  
}
```

El método `cargarRecetaParaEditar()`, se invoca desde **EditarRecetaActivity**, permite consultar al **Repositorio** por la **RecetaCompleta** con el identificador pasado por parámetro. El **ViewModel** observa los cambios en los datos que retorna el **Repositorio** y notifica a los **Fragments** mediante los **LiveData**.

## Capítulo 6. Conclusiones y líneas futuras

Se ha desarrollado una aplicación siguiendo las **APIs** más actuales y las mejores prácticas recomendadas por Android, lo que permite una fácil ampliación gracias a su grado de modularidad.

Como líneas futuras de implementación, se proponen las siguientes mejoras:

- Migración a **Kotlin** para emplear **Jetpack Compose**.
- Persistencia en la nube mediante **Retrofit**, **Supabase** o **Firebase Storage**.
- Importación y exportación de recetas.
- Autenticación de usuarios mediante **Firebase Authentication**.
- Exportación de recetas a **PDF** usando plantillas.
- Ampliación de imágenes adjuntas a cada paso.
- Filtrado multicriterio.
- Categorización de recetas.
- Uso de **DiffUtil** y **ListAdapter** en **RecyclerView** para optimizar rendimiento.
- Transiciones y animaciones en la navegación entre fragmentos.

## Capítulo 7. Bibliografía.

### 7.1. Libros

- Ramos Martín, A. y Ramos Martín, M. J. Entornos de desarrollo (2ª ed.). Editorial Garceta (Madrid, 2014). Págs.: 13-15.

### 7.2. Páginas web

URL	Fecha
<a href="https://gerardfp.github.io/dam/mobils/room/">https://gerardfp.github.io/dam/mobils/room/</a>	14/10/2025
<a href="https://www.geeksforgeeks.org/android/modal-bottom-sheet-in-android-with-examples/">https://www.geeksforgeeks.org/android/modal-bottom-sheet-in-android-with-examples/</a>	21/10/2025
<a href="https://www.javaoneworld.com/2020/05/creating-custom-bottom-sheet-dialog.html">https://www.javaoneworld.com/2020/05/creating-custom-bottom-sheet-dialog.html</a>	21/10/2025
<a href="https://www.geeksforgeeks.org/android/shared-viewmodel-in-android/">https://www.geeksforgeeks.org/android/shared-viewmodel-in-android/</a>	21/10/2025
<a href="https://developer.android.com/guide/fragments/communicate">https://developer.android.com/guide/fragments/communicate</a>	22/10/2025
<a href="https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-savedstate">https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-savedstate</a>	24/10/2025
<a href="https://www.geeksforgeeks.org/kotlin/android-tablelayout/">https://www.geeksforgeeks.org/kotlin/android-tablelayout/</a>	25/10/2025
<a href="https://www.develou.com/implementar-el-navigation-component/">https://www.develou.com/implementar-el-navigation-component/</a>	28/10/2025
<a href="https://gerardfp.github.io/dam/mobils/fragments/">https://gerardfp.github.io/dam/mobils/fragments/</a>	28/10/2025
<a href="https://medium.com/@auvehassan/android-menu-provider-aa663150aca8">https://medium.com/@auvehassan/android-menu-provider-aa663150aca8</a>	30/10/2025
<a href="https://proandroiddev.com/menuprovider-api-android-208465c80a54">https://proandroiddev.com/menuprovider-api-android-208465c80a54</a>	30/10/2025
<a href="https://www.youtube.com/watch?v=fGVgQNqHoTk">https://www.youtube.com/watch?v=fGVgQNqHoTk</a>	30/10/2025
<a href="https://medium.com/@fitareq/best-practices-for-efficient-recyclerview-performance-in-android-b405eedc6d34">https://medium.com/@fitareq/best-practices-for-efficient-recyclerview-performance-in-android-b405eedc6d34</a>	02/11/2025
<a href="https://www.develou.com/recyclerview-en-android/">https://www.develou.com/recyclerview-en-android/</a>	02/11/2025
<a href="https://www.geeksforgeeks.org/android/how-to-add-drag-and-drop-feature-in-android-recyclerview/">https://www.geeksforgeeks.org/android/how-to-add-drag-and-drop-feature-in-android-recyclerview/</a>	03/11/1025
<a href="https://medium.com/@appdevinsights/difference-between-id-and-id-4b4a95e90f54">https://medium.com/@appdevinsights/difference-between-id-and-id-4b4a95e90f54</a>	03/11/1025
<a href="https://developer.android.com/reference/androidx/recyclerview/widget/ItemTouchHelper.Callback">https://developer.android.com/reference/androidx/recyclerview/widget/ItemTouchHelper.Callback</a>	05/11/2025
<a href="https://www.c-sharpcorner.com/article/how-to-make-a-drag-and-drop-recyclerview-in-android/">https://www.c-sharpcorner.com/article/how-to-make-a-drag-and-drop-recyclerview-in-android/</a>	05/11/2025
<a href="https://medium.com/@Codeible/swipe-or-slide-and-drag-and-drop-items-in-recyclerview-6dbe4871f87">https://medium.com/@Codeible/swipe-or-slide-and-drag-and-drop-items-in-recyclerview-6dbe4871f87</a>	05/11/2025
<a href="https://www.digitalocean.com/community/tutorials/android-recyclerview-drag-and-drop">https://www.digitalocean.com/community/tutorials/android-recyclerview-drag-and-drop</a>	05/11/2025

## Anexos

### Anexo I. Entidades Room

#### Receta

```
@Entity(tableName = Constantes.TABLA_RECETAS)
public class Receta {

    @PrimaryKey(autoGenerate = true)
    private long idReceta;

    @ColumnInfo(defaultValue = "Sin título")
    private String titulo;

    private String descripcion;

    @ColumnInfo(name = "imagen")
    private String imagenUri;

    private long tiempo;
    ...
}
```

#### Ingrediente

```
@Entity(tableName = Constantes.TABLA_INGREDIENTES,
        foreignKeys = {@ForeignKey(
            entity = Receta.class,
            parentColumns = "idReceta",
            childColumns = "idReceta_fk",
            onDelete = ForeignKey.CASCADE,
            onUpdate = ForeignKey.CASCADE)
    })
public class Ingrediente {

    @PrimaryKey(autoGenerate = true)
    private long idIngrediente;

    private String nombre;

    @ColumnInfo(defaultValue = "0")
    private double cantidad;

    private String unidad;

    @NonNull
    @ColumnInfo(name = "idReceta_fk", index = true)
    private long idReceta;
    ...
}
```

#### Paso

```
@Entity(tableName = Constantes.TABLA_PASOS,
        foreignKeys = {@ForeignKey(
            entity = Receta.class,
            parentColumns = "idReceta",
            childColumns = "idReceta_fk",
            onDelete = ForeignKey.CASCADE,
            onUpdate = ForeignKey.CASCADE)
    })
```

```
public class Paso {  
  
    @PrimaryKey(autoGenerate = true)  
    private long idPaso;  
  
    private int orden;  
    private String descripcion;  
  
    @ColumnInfo(name = "idReceta_fk", index = true)  
    private long idReceta;  
    ...  
}
```

## Anexo II. Relaciones Room

### RecetaIngredientes

```
public class RecetaIngredientes {  
    @Embedded  
    public Receta receta; // Es la entidad padre  
  
    @Relation(  
        parentColumn = "idReceta",  
        entityColumn = "idReceta_fk" // Es la clave foránea  
    )  
  
    public List<Ingrediente> ingredientes;  
}
```

### RecetaPasos

```
public class RecetaPasos {  
    @Embedded  
    public Receta receta; // Es la entidad padre  
  
    @Relation(  
        parentColumn = "idReceta",  
        entityColumn = "idReceta_fk"  
    )  
  
    public List<Paso> pasos;  
}
```

### RecetaCompleta

```
public class RecetaCompleta {  
  
    @Embedded  
    public Receta receta;  
  
    @Relation(  
        parentColumn = "idReceta",  
        entityColumn = "idReceta_fk"  
    )  
    public List<Ingrediente> ingredientes;  
  
    @Relation(  
        parentColumn = "idReceta",  
        entityColumn = "idReceta_fk"  
    )  
    public List<Paso> pasos;  
}
```

## Anexo III. Implementación DAO

### IngredienteDAO

```
@Dao
public interface IngredienteDAO {

    @Insert
    void insertarIngredientes(List<Ingrediente> ingredientes);

    @Transaction
    @Query("SELECT * FROM recetas WHERE idReceta = :id")
    LiveData<RecetaIngredientes> getRecetaConIngredientes(long id);

    @Insert
    void insertarIngrediente(Ingrediente ing);
}
```

### PasoDAO

```
@Dao
public interface PasoDAO {

    @Insert
    void insertarPasos(List<Paso> pasos);

    // Recuperar pasos de una receta
    @Transaction
    @Query("SELECT * FROM recetas WHERE idReceta = :id")
    LiveData<RecetaPasos> getRecetaConPasos(long id);

    @Insert
    void insertarPaso(Paso paso);
}
```

### RecetaDAO

```
@Dao
public interface RecetaDAO {

    @Transaction
    @Query("SELECT * FROM recetas WHERE idReceta = :id")
    LiveData<RecetaCompleta> getRecetaCompleta(long id);

    @Transaction
    @Query("SELECT * FROM recetas")
    LiveData<List<RecetaCompleta>> getTodasRecetasCompletas();

    @Query("SELECT * FROM recetas WHERE idReceta = :id")
    Receta getReceta(long id);

    @Query("SELECT * FROM recetas")
    LiveData<List<Receta>> getTodasRecetas();

    @Query("SELECT * FROM recetas " +
            "WHERE titulo LIKE '%' || :texto || '%" +
            "ORDER BY titulo ASC")
    LiveData<List<Receta>> buscarPorTitulo(String texto);

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    long insertarReceta(Receta receta);

    @Query("DELETE FROM recetas WHERE idReceta = :id")
}
```

```
void borrarRecetaPorId(long id);

@Delete
void borrarReceta(Receta receta);

@Query("SELECT imagen FROM RECETAS WHERE idReceta = :id")
String getUriImagen(long id);
}
```

## Anexo IV. Base de datos

```
@Database(
    entities = {Receta.class, Ingrediente.class, Paso.class},
    version = 1,
    exportSchema = false)
public abstract class Recetario extends RoomDatabase {
    private static final String TAG = "RecetarioBBDD";

    public abstract RecetaDAO recetaDAO();
    public abstract IngredienteDAO ingredienteDAO();
    public abstract PasoDAO pasoDAO();
    //Patrón Singleton con multithilo
    private static volatile Recetario INSTANCIA;
    private static final int NUMBER_OF_THREADS = 4;
    public static final ExecutorService servicioExecutor =
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);

    public static Recetario getInstance(final Context context) {

        if (INSTANCIA == null) {
            synchronized (Recetario.class) {
                if (INSTANCIA == null) {
                    INSTANCIA = crearInstancia(context);
                }
            }
        }

        return INSTANCIA;
    }

    // Método para crear la instancia
    private static Recetario crearInstancia(Context context) {
        Log.d(TAG, "Creando instancia de la base de datos");
        Recetario db = Room.databaseBuilder(context, Recetario.class,
            Constantes.BASEDATOS)
            .allowMainThreadQueries()
            .addCallback(crearCallback(context))
            .build();
        return db;
    }
}
```

## Anexo V. Poblar base de datos

```
// Interceptar el ciclo de vida BBDD mediante callback
private static RoomDatabase.Callback crearCallback(Context context) {

    @Override
    public void onCreate(@NonNull SupportSQLiteDatabase db) {
        super.onCreate(db);
        Log.d(TAG, "Base de datos creada, poblando datos...");

        servicioExecutor.execute(() -> {
            poblarBaseDatos(INSTANCIA, context);
        });
    }

    @Override
    public void onOpen(@NonNull SupportSQLiteDatabase db) {
        super.onOpen(db);
        Log.d(TAG, "Base de datos abierta");
    }
};

private static void poblarBaseDatos(RoomDatabase database, Context appContext) {
    if (database == null) return;

    try {
        /** Primero entidades fuertes. Recetas */
        String uriImg1 = "android.resource://" +
            appContext.getPackageName() + "/drawable/tortilla_patatas";

        Receta r1 = new Receta("Receta 1", "Descripción 1", uriImg1, 45);
        Receta r2 = new Receta("Receta 2", "Descripción 2", "Sin imagen", 75);

        // Insertar recetas
        long rowId1 = database.recetaDAO().insertarReceta(r1);
        long rowId2 = database.recetaDAO().insertarReceta(r2);

        /** Segundo entidades débiles. Ingredientes */
        List<Ingrediente> ingredientes = Arrays.asList(
            new Ingrediente("R1-Ingrediente 1", 100.0, "g", rowId1),
            new Ingrediente("R1-Ingrediente 2", 1.0, "uds", rowId1),
            new Ingrediente("R2-Ingrediente 1", 50.0, "ml", rowId2),
            new Ingrediente("R2-Ingrediente 2", 2, "cucharadas", rowId2)
        );

        // Insertar ingredientes
        database.ingredienteDAO().insertarIngredientes(ingredientes);

        /** Pasos */
        List<Paso> pasos = Arrays.asList(
            new Paso(1, "Mezclar ingredientes", rowId1),
            new Paso(2, "Hornear 30 minutos", rowId1),
            new Paso(1, "Batir los huevos", rowId2)
        );

        // Insertar pasos
        database.pasoDAO().insertarPasos(pasos);
    } catch (Exception e) {
        Log.e(TAG, "Error al poblar BD: " + e.getMessage());
    }
}
```



## Anexo VI. Repositorio

```
public class RecetarioRepositorio {
    private static final String TAG = RecetarioRepositorio.class.getSimpleName();

    //Guardar instancia de la base de datos
    private final Recetario mdb;
    private final IngredienteDAO mIngredienteDAO;
    private final RecetaDAO mRecetaDAO;
    private final PasoDAO mPasoDAO;

    private final Context appContext;

    public RecetarioRepositorio(Context contexto) {
        this.appContext = contexto;
        this.mdb = Recetario.getInstance(contexto);

        // Invocamos la implementación interna de los DAO por Room
        this.mRecetaDAO = mdb.recetaDAO();
        this.mIngredienteDAO = mdb.ingredienteDAO();
        this.mPasoDAO = mdb.pasoDAO();
    }

    public LiveData<List<Receta>> getTodasRecetas() {
        return mRecetaDAO.getTodasRecetas();
    }

    public LiveData<List<Receta>> buscarPortitulo(String texto) {
        return mRecetaDAO.buscarPortitulo(texto);
    }

    public void borrarReceta(Receta receta) {
        Recetario.servicioExecutor.execute(
            new Runnable() {
                @Override
                public void run() {
                    eliminarImagen(receta.getImagenUri());
                    mRecetaDAO.borrarReceta(receta);
                }
            }
        );
    }

    public LiveData<RecetaCompleta> getRecetaCompleta(long recetaId) {
        return mRecetaDAO.getRecetaCompleta(recetaId);
    }

    public long insertarReceta(Receta receta) {
        return mRecetaDAO.insertarReceta(receta);
    }

    private void persistirRecetaCompleta(Receta receta,
                                           List<Ingrediente> ingredientes,
                                           List<Paso> pasos,
                                           Uri imagenUriTemporal) {

        String rutaFinalImagen = copiarImagenAlmacenamientoInterno(imagenUriTemporal);
        receta.setImagenUri(rutaFinalImagen);

        long recetaId = mRecetaDAO.insertarReceta(receta);
        receta.setIdReceta(recetaId);

        for (Ingrediente ing : ingredientes) ing.setIdReceta(recetaId);
        mIngredienteDAO.insertarIngredientes(ingredientes);
    }
}
```

```
        for (int i = 0; i < pasos.size(); i++) {
            Paso paso = pasos.get(i);
            paso.setIdReceta(recetaId);
            paso.setOrden(i + 1);
        }
        mPasoDAO.insertarPasos(pasos);
    }

    public void insertarRecetaCompleta(Receta receta,
                                       List<Ingrediente> ingredientes,
                                       List<Paso> pasos,
                                       Uri imagenUriTemporal,
                                       Runnable onFinish) {
        Recetario.servicioExecutor.execute(() -> {
            mdb.runInTransaction(() -> persistirRecetaCompleta(receta, ingredientes, pasos,
imagenUriTemporal));

            if (onFinish != null) {
                new Handler(Looper.getMainLooper()).post(onFinish);
            }
        });
    }

    public void reemplazarRecetaCompleta(long idOriginal,
                                       Receta receta,
                                       List<Ingrediente> ingredientesNuevos,
                                       List<Paso> pasosNuevos,
                                       Uri imagenUriTemporal,
                                       Runnable onFinish) {
        Recetario.servicioExecutor.execute(() -> {
            mdb.runInTransaction(() -> {

                //1. Sacamos receta original solo para conocer su URI real
                Receta recetaOriginal = mRecetaDAO.getReceta(idOriginal);
                String imagenOriginal = recetaOriginal.getImagenUri();

                //2. Borrar la imagen solo si el usuario realmente eligió una nueva
                if (imagenUriTemporal != null && imagenOriginal != null) {
                    eliminarImagen(imagenOriginal);
                }

                //3. Borrarnos la receta completa
                mRecetaDAO.borrarRecetaPorId(idOriginal);

                //4. Insertamos la nueva con su imagen correspondiente
                persistirRecetaCompleta(receta, ingredientesNuevos, pasosNuevos,
imagenUriTemporal);

            });

            if (onFinish != null) {
                new Handler(Looper.getMainLooper()).post(onFinish);
            }
        });
    }

    //Gestion de imagenes
    private String copiarImagenAlmacenamientoInterno(Uri uriTemp) {
        if (uriTemp == null) return null;

        try {
            // Crear nombre único
            String timeStamp = new SimpleDateFormat("yyyyMMdd_HH:mm:ss", Locale.getDefault())
                .format(new Date());
            String nombreArchivo = "RECETA_" + timeStamp + ".jpg";
```

```
// Directorio privado de la app. Si no existe se crea.
File directorio = new File(appContext.getFilesDir(), "imagenes_recetas");
if (!directorio.exists()) {
    directorio.mkdirs();
}

//Creamos el archivo a persistir
File archivoDestino = new File(directorio, nombreArchivo);

// Copiar archivo con buffer. Usamos try-with-resources para cerrar archivos al
finalizar
try (InputStream in = appContext.getContentResolver().openInputStream(uriTemp);
    OutputStream out = new FileOutputStream(archivoDestino)) {
    byte[] buffer = new byte[1024];
    int length;

    while ((length = in.read(buffer)) > 0) {
        out.write(buffer, 0, length);
    }
}

// Devolvemos la uri
String ruta = archivoDestino.getAbsolutePath();
return "file://" + ruta;

} catch (Exception e) {
    Log.e(TAG, "Error copiando imagen: " + e.getMessage());
    return null;
}

}

// Método para eliminar imagenes de recetas modificadas
public void eliminarImagen(String imagenPath) {
    if (imagenPath == null) return;

    // Eliminamos prefijo file:// para que File encuentre el archivo
    if (imagenPath.startsWith("file://")) {
        imagenPath = imagenPath.substring("file://".length());
    }

    File imagenFile = new File(imagenPath);
    if (imagenFile.exists()) {
        imagenFile.delete();
    }
}
}
```