

**ISRAEL LUCAS TORRIJOS IES AZARQUIEL**

**CFGS DAM CURSO 25/26 Proyecto: Digital Recipes**

**Fecha: 08/12/2025 Tutor: D. Francisco J. Pulido Moya**

Tabla de contenido

[Proyecto intermodular: Digital Recipes 3](#_Toc213598851)

[Capítulo 1. Introducción y objetivos 3](#_Toc213598852)

[Capítulo 2. Especificación de Requisitos 4](#_Toc213598853)

[Capítulo 3. Planificación Temporal y Evaluación de Costes 5](#_Toc213598854)

[3.1. Planificación temporal 5](#_Toc213598858)

[3.2. Evaluación de costes 6](#_Toc213598859)

[Capítulo 4. Tecnologías Utilizadas 7](#_Toc213598860)

[Capítulo 5. Desarrollo e Implementación 11](#_Toc213598861)

[8.1. Model 11](#_Toc213598867)

[8.2. View 20](#_Toc213598868)

[8.3. ViewModel 25](#_Toc213598869)

[Capítulo 6. Conclusiones y líneas futuras 25](#_Toc213598870)

[Capítulo 7. Bibliografía. 25](#_Toc213598871)

[10.1. Libros 25](#_Toc213598874)

[10.2. Páginas web 26](#_Toc213598875)

# 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:

|  |  |
| --- | --- |
| **Fase** | **Descripción** |
| 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.



### 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 **MVVM**. | 5 |
| Diseño de la interfaz | Bocetos de pantallas, selección de colores, tipografía, iconografía y estructura de navegación. | 4 |
| Implementación del modelo y repositorio (**Room**) | Codificación de entidades, **DAOs** y repositorio, con prueba de operaciones **CRUD**. | 5 |
| Implementación de **ViewModel** y **LiveData** | Conexión entre la lógica de datos y la interfaz. Observadores, comunicación entre fragments y estados de vista. | 5 |
| Desarrollo de **Activities** y **Fragments**  Integración de interfaz y lógica  Pruebas y corrección de errores  Documentación del proyecto | Creación de pantallas principales, vistas de detalle, formularios y navegación mediante **ViewPager2** y **Navigation Components**. | 12 |
| Enlace de layouts con **ViewBinding**, adaptadores, carga de imágenes (**Glide**) 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 **MVVM**. | 5 |

Duración estimada: 48 días de trabajo efectivo.

### 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.

1. ***Patrón Repository***

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

1. ***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.

1. ***Métodos Factory***

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

1. ***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.

1. ***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 un implementación más limpia que la empleada con la **API** de **SQLite**.

1. ***DAO***

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

1. ***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**.

1. ***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**.

1. ***LiveData***

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

* **Interfaz de usuario**

Las tecnologías empleadas para esta capa son las siguientes:

1. ***ViewBinding***

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

1. ***RecyclerView***

Se utiliza para mostrar listas de objetos en los que el usuario tendrá un nivel de interactuación, dando un agradable aspecto visual y correcta gestión de rendimiento respecto a **ListViews**.

1. ***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.

1. ***ViewPager2***

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

1. ***BottomSheet***

Se utiliza para presentar acciones sobre otros objetos seleccionados.

1. ***NavigationUI***

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

1. ***Glide***

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

1. ***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().

* **Documentación**

Para la realización de diagramas se ha empleado **PlantUML**.

## 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.

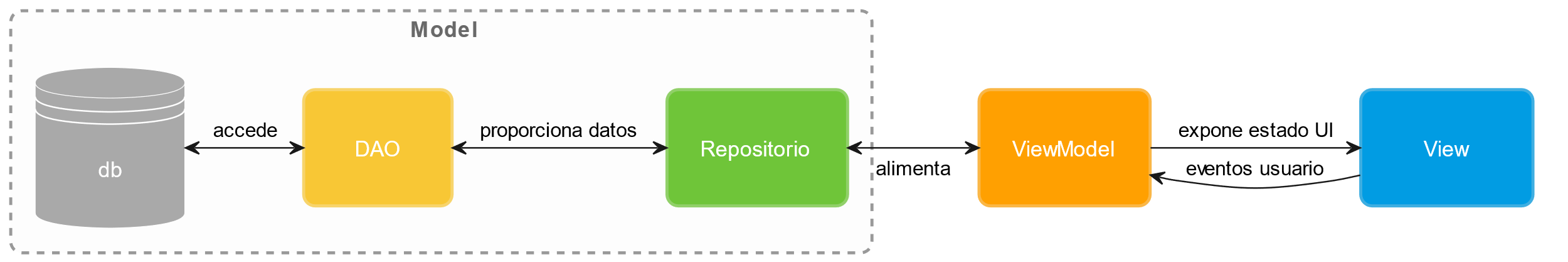


Figura 1. Arquitectura MVVM



### 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:



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.

Por eso tengo que crear tres clases java que representan **entidades**:

* **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.

@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.** 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.

@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**. 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.

@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;   
 ...

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.

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.** Permite realizar operaciones **CRUD** sobre los pasos vinculados a una receta.

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

* **RecetaCompleta.** Permite realizar operaciones **CRUD** con los datos completos de la receta.

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;  
}

#### 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.

**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);  
}

#### 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](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase?hl=es-419).
* La clase debe tener una anotación [@Database](https://developer.android.com/reference/kotlin/androidx/room/Database?hl=es-419). 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:

@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;  
 }  
  
}

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 datos.

Al implementar un objeto **RoomDatabase.Callback** definimos datos **dummy** en el método onCreate() mediante un hilo secundario o worker:

*// 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(Recetario 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());  
 }  
}

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

#### 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*.

A continuación, se muestra parte de la clase implementada para el **Repositorio**. En ella 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.

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();  
 }  
...

AQUÍ FALTA CHICHA

### View

La capa **View** en el modelo **MVVM** corresponde a la interfaz gráfica de la aplicación. Sus elementos se definen en archivos **layout XML**, que posteriormente se transforman en una jerarquía de objetos 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), simplificando así el trabajo con la interfaz.

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. **ModificarRecetaActivity**

#### 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**.

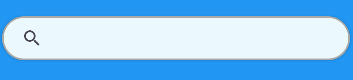


Figura 4. Elemento SearchView

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 almacena en una variable **MutableLiveData** 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.

1. **RecyclerView**

En 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**.

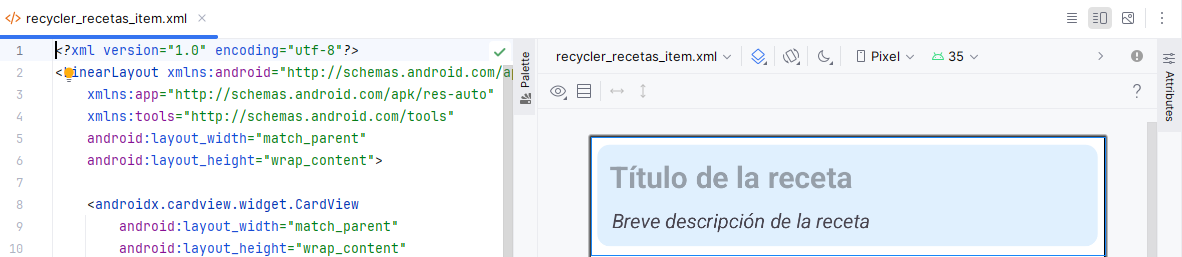


Figura 5. Layout del ítem del RecyclerView

Será responsabilidad de esta clase vincular los datos de cada objeto **Receta** con las vistas definidas para cada 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 RecyclerRecetasItemBinding 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 RecyclerRecetasItemBinding binding, RecetasAdapter.OnClickItemListener 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 OnClickItemListener listener;

Y serán instanciados en el constructor:

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

Además, nos obliga a implementar estos [3 métodos](https://developer.android.com/develop/ui/views/layout/recyclerview?hl=es-419#implement-adapter):

* getItemCount(): **RecyclerView** llama a este método para determinar el tamaño de la lista y cuándo no hay más elementos que se puedan 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](https://medium.com/@kumargaurav.xqf/mastering-recyclerview-optimizing-oncreateviewholder-and-onbindviewholder-for-large-datasets-a05d9249a1b1) 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 tenido que refactorizar todos los adaptadores y viewholders, para alinear la implementación con las *mejores prácticas*.

1. **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**.

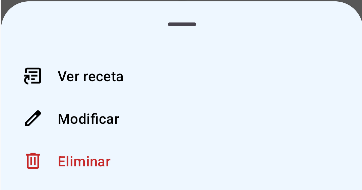


Figura 6. Opciones del BottomSheet

Para implementar la funcionalidad de este ventana flotante ha sido necesario declarar la clase **RecetasBottomSheet** que extiende de **BottomSheetDialogFragment**.

Esta clase tendrá como miembros:

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

La clase **BottomsheetVerRecetasBinding** es un implementación interna de Android como consecuencia de haber activado **ViewBinding** en **build.gradle.kts** de la aplicación:

*android* **{** ...buildFeatures **{** viewBinding = true  
 **}  
}**

Esta clase debe ser capaz de reaccionar a los eventos clic sobre cada uno de los elementos que componen su interfaz.

Por este motivo, se ha creado una interfaz que define tres métodos, uno por cada operación realizable:

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

Será la **Activity** quien implemente la funcionalidad de estos métodos, conforme a las recomendaciones de **Android**. Para poder escuchar los eventos se ha declarado en la clase un setter para que desde la **Activity** se pueda añadir dicha implementación:

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

1. **FloatingActionButton**

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



Figura 7. Botón FloatingActionButton

Un ejemplo de visualización de esta actividad puede verse en la siguiente captura de imagen de emulador Android:



Clic en receta

Figura 8. Ejemplo de visualización de RecetasActivity

#### 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](https://developer.android.com/guide/navigation), [ViewPager2](https://developer.android.com/develop/ui/views/animations/screen-slide-2) y [TabLayout](https://developer.android.com/reference/com/google/android/material/tabs/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 él se indica el **grafo de navegación** en el atributo app:navGraph, y se establece app:defaultNavHost="true" para que el **NavController** gestione el botón *Up* (atrás) del sistema:

<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**.

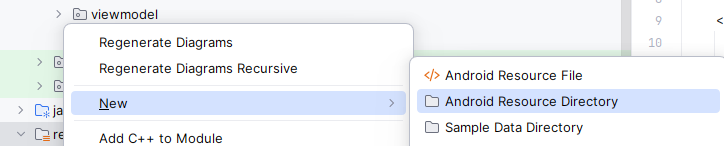


Figura 10. Añadir carpeta de recursos

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

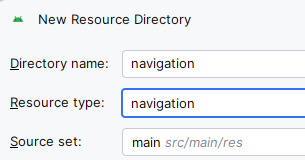


Figura 11. Carpeta Navigation

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, del elemento **navigation**, indica el **Fragment** inicial. El atributo android:name, del fragmento, especifica la clase **Java** que contiene el **ViewPager2** responsable de coordinar la navegación. Ambos atributos son de obligada declaración para que funcione.

#### CrearRecetaActivity

### ViewModel

## Capítulo 6. Conclusiones y líneas futuras

## Capítulo 7. Bibliografía.



### Libros

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

### Páginas web

|  |  |
| --- | --- |
| **URL** | **Fecha** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |