

**DIGITAL RECIPES**

**Israel Lucas Torrijos Digital Recipes**

**DAM Grupo del profesor D. Francisco J. Pulido Moya**

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Tabla de contenido

[Proyecto intermodular 3](#_Toc211708795)

[Capítulo 1. Introducción y objetivos 3](#_Toc211708796)

[Capítulo 2. Especificación de Requisitos 3](#_Toc211708797)

[Capítulo 3. Planificación Temporal y Evaluación de Costes 3](#_Toc211708798)

[Capítulo 4. Tecnologías Utilizadas 3](#_Toc211708799)

[Capítulo 5. Desarrollo e Implementación 3](#_Toc211708800)

[5.1. Model 3](#_Toc211708806)

[5.2. ViewModel 10](#_Toc211708807)

[5.3. View 10](#_Toc211708808)

[Capítulo 6. Conclusiones y líneas futuras 11](#_Toc211708809)

[Capítulo 7. Bibliografía. 11](#_Toc211708810)

[7.1. Páginas web 11](#_Toc211708813)

# Proyecto intermodular

## Capítulo 1. Introducción y objetivos

## Capítulo 2. Especificación de Requisitos

## Capítulo 3. Planificación Temporal y Evaluación de Costes

## Capítulo 4. Tecnologías Utilizadas

ViewBinding

RecyclerView

Patrón arquitectura MVVM

Patrón de diseño Repository, Singleton

Room

BottomSheets

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.

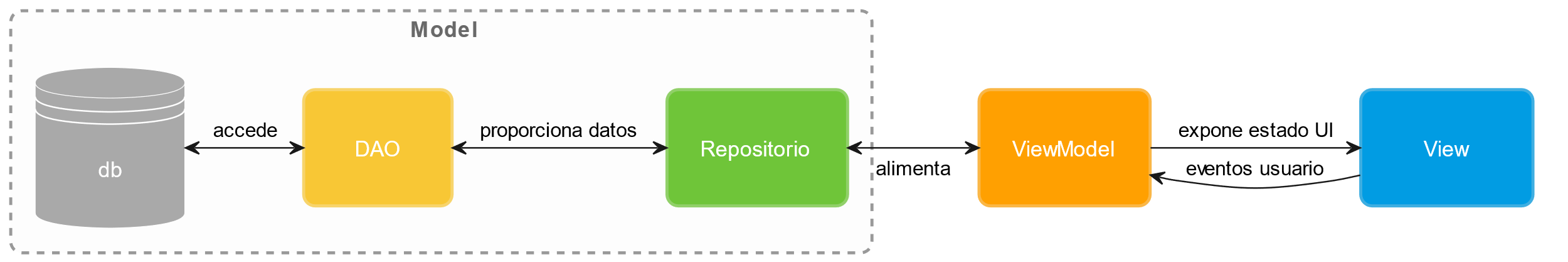


Figura 1. Arquitectura MVVM



### 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 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 int idReceta;  
  
 @ColumnInfo(defaultValue = "Sin título")  
 private String titulo;  
  
 private String descripcion;  
  
 @ColumnInfo(name = "imagen")  
 private String imagenUri;  
  
 private int 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 int idIngrediente;  
  
 private String nombre;  
  
 @ColumnInfo(defaultValue = "0")  
 private double cantidad;  
  
 private String unidad;  
  
 @NonNull  
 @ColumnInfo(name = "idReceta\_fk", index = true)  
 private int 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 int idPaso;  
  
 private int orden;  
 private String descripcion;  
  
 @ColumnInfo(name = "idReceta\_fk", index = true)  
 private int 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" )  
  
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 númerico 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.

#### 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, que conforman la base de datos.
* 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 metodo 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";  
  
 *// Exponer DAO* 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*;  
 }  
  
 *// Metodo para crear la instancia* private static Recetario crearInstancia(Context context) {  
 Log.*d*(*TAG*, "Creando instancia de la base de datos");  
 return Room.*databaseBuilder*(context, Recetario.class, Constantes.*BASEDATOS*)  
 .allowMainThreadQueries()  
 .addCallback(*accionesCicloVida*)  
 .build();  
 }  
  
}

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:

*// Metodo para crear la instancia*private static Recetario crearInstancia(Context context) {  
 Log.*d*(*TAG*, "Creando instancia de la base de datos");  
 return Room.*databaseBuilder*(context, Recetario.class, Constantes.*BASEDATOS*)  
 .allowMainThreadQueries()  
 .addCallback(*accionesCicloVida*)  
 .build();  
}  
  
*// Interceptar el ciclo de vida BBDD mediante callback*static RoomDatabase.Callback *accionesCicloVida* = new RoomDatabase.Callback() {  
  
 @Override  
 public void onCreate(@NonNull SupportSQLiteDatabase db) {  
 super.onCreate(db);  
 Log.*d*(*TAG*, "Base de datos creada, poblando datos...");  
  
 *// Poblar la base de datos en un hilo worker  
 // para evitar bloquear el hilo principal  
 servicioExecutor*.execute(() -> {  
 *poblarBaseDatos*(Recetario.*INSTANCIA*);  
 });  
 }  
  
 @Override  
 public void onOpen(@NonNull SupportSQLiteDatabase db) {  
 super.onOpen(db);  
 Log.*d*(*TAG*, "Base de datos abierta");  
 }  
};  
  
private static void poblarBaseDatos(Recetario database) {  
 if (database == null) return;  
  
 try {  
 */\*\* Primero entidades fuertes. Recetas \*\*/* Receta r1 = new Receta("Receta 1", "Descripción 1", "Sin imagen", 0);  
 Receta r2 = new Receta("Receta 2", "Descripción 2", "Sin imagen", 0);  
  
 *// 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, "ud", 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 componente final de la capa **Modelo** es la clase **Repositorio**. Su función es centralizar la lógica de negocio y abstraer el acceso a los datos, funcionando como un punto único de coordinación entre el **ViewModel** y las distintas fuentes de datos.

Por lo tanto, veremos como simplemente propaga las llamadas a los métodos de los **DAO**.

### ViewModel

### View

## Capítulo 6. Conclusiones y líneas futuras

## Capítulo 7. Bibliografía.



### Páginas web

|  |  |
| --- | --- |
| URL | Fecha |
| <https://developer.android.com/training/data-storage/room?hl=es-419> | 14/10/2025 |
| <https://gerardfp.github.io/dam/mobils/room/> | 14/10/2025 |
| <https://www.geeksforgeeks.org/android/modal-bottom-sheet-in-android-with-examples/> | 21/10/2025 |
| <https://stackoverflow.com/questions/37104960/bottomsheetdialog-with-transparent-background> | 21/10/2025 |
| <https://www.javaoneworld.com/2020/05/creating-custom-bottom-sheet-dialog.html> | 21/10/2025 |