Tema 5.1 - Room

Descargar estos apuntes pdf o html

Índice

Introducción

- **Persistencia local con Room**
  - Documentación oficial: **Save data in a local database using Room**
  - Documentación oficial versiones librería: **Room**
  - Video Tutorial (Inglés): **Philipp Lackner**
  - Video Tutorial (Castellano): **Martin Kiperszmid**
  - Video Tutorial (Castellano): **AristiDevs**
  - Video Tutorial (Castellano): **Gibrán García**

La plataforma Android proporciona **dos herramientas** principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers**.

> **Nota**
>
> Existen otras opciones de terceros como **SQLDelight** (**SQLite**) pero no están tan integradas Android Studio como Room. Aunque, es una opción muy válida otros SO **Kotlin Multipltaform**. Incluso si nos decidimos por usar **Firestore**, existe la posibilidad de **usarlo de forma offline**

Nosotros vamos a centrarnos en **SQLite**, aunque no entraremos ni en el diseño de BBDD relacionales ni en el uso avanzado de la BBDD. Para conocer toda la funcionalidad de SQLite se recomienda usar la documentación oficial. **El problema de trabajar directamente con esta API es que es un trabajo a bajo nivel que puede llevar a errores en tiempo de ejecución**.

Por este motivo Android Jetpack ha proporcionado ORM a modo de capa de abstracción que facilita enormemente el proceso de codificación La biblioteca que nos va a permitir esta abstracción y se distribuye como **Room**.

Dependencias

Para poder usar la funcionalidad de Room y las anotaciones, deberemos añadir una serie de dependencias y plugins.

Al igual que sucedía con **Hilt**, en **Room** vamos a necesitar algún tipo de procesador de anotaciones. Las primeras versiones usaban
KAPT, pero como con Hilt, **Room** ya está migrado a **KSP**. Por tanto, si ya estamos usando **Hilt** no necesitaremos añadir el plugin de **KSP**. No obstante vamos a recordar los pasos...

En el catálogo de versiones `lib.versions.toml` deberemos comprobar que hemos definido tener:

```
[versions]
kotlin = "2.0.20"
ksp = "2.0.20-1.0.25"
room = "2.6.1"

[libraries]
androidx-room-ktx = { group = "androidx.room", name = "room-ktx", version.ref = "room" }
androidx-room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "room" }

[plugins]
devtools-ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

En el `build.gradle.kts` raíz del proyecto añadiremos el siguiente plugin:

```
plugins {
    alias(libs.plugins.devtools.ksp) apply false
}
```

En el `build.gradle.kts` del **módulo de la aplicación** (app) añadiremos:

```
plugins {
    alias(libs.plugins.devtools.ksp)
}
...
dependencies {
    implementation(libs.androidx.room.ktx)
    ksp(libs.androidx.room.compiler)
}
```

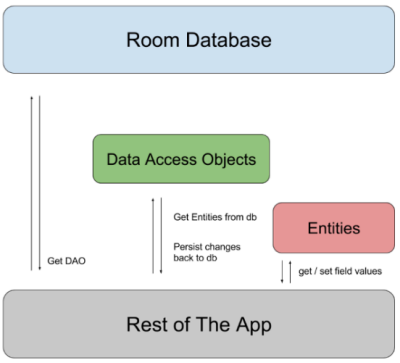
La versión de la librería puede cambiar, pero puedes ver los últimos releases de la misma en el siguiente enlace: **Room**

Componentes de Room

Para trabajar con Room debemos familiarizarnos con su arquitectura que está compuesta por tres partes principales:

- **Entity** (Entities), serán las clases que definirán las entidades de nuestra Base de datos, que corresponden con cada tabla que la forman.

- **DAOs** (Data Access Objects), en este caso serán las clases abstractas o interfaces donde estarán definidos los métodos que nos permitirán la operatividad (inserción, consulta, etc) con cada una de las tablas de la Base de Datos.
- **RoomDatabase** (Room Database), contiene la Base de Datos y el acceso a esta, a la vez que une los dos anteriores conceptos.



Una vez que se creen estos tres elementos, la app podrá acceder a los DAOs a través de las instancias de Room Database que creemos, de forma más estable que con el acceso directo a la API de SQLite.

El uso de Room se hace a través de **Anotaciones** añadidas a las clases e interfaces, las principales son:

- `@Entity`, indica que la clase a la que se atribuye debe ser tratada como una entidad.
- `@Dao`, debe de añadirse a las interfaces que queremos que sean nuestras Daos. Dentro de estas interfaces se utilizan otras anotaciones que veremos más adelante.
- `@Database`, se añadirá a las Room Database he indicará que es una Database, estas clases deben ser abstractas y heredar de RoomDatabase.

Añadiendo paquete room al proyecto

Recuerda que en la distribución de paquetes que se propuso a principio de curso. Dentro del paquete `.data` vamos a crear un paquete por cada una de las fuentes de datos que tengamos. En este caso, como vamos a usar Room, crearemos un paquete denominado `.data.room` y dentro del él vamos a crear los componentes que hemos mencionado y que concretaremos a continuación.

> **Código de los ejemplos**
>
> Si te surge alguna duda o tienes dificultades para completar este tema. Puedes descargar el proyecto con el código de mismo del siguiente enlace: **Proyecto ejemplos**

Crear las entidades que definen nuestro modelo

Para definir la persistencia con Room, utilizaremos lo que denominaremos *'Code First'*, *'Model First'* o *'Entity First'*, es decir, primero definiremos las entidades que queremos persistir y sus relaciones. Posteriormente, a partir de estas definiciones, se creará la base de datos.

Para ello pues, el primer paso a realizar será el de crear las distintas tablas que tendrá nuestra base de datos. Cada tabla corresponderá con una clase **Entity** etiquetada como `@Entity` en la que añadiremos los campos correspondiente a la tabla. Vamos a usar un ejemplo sencillo para entender el funcionamiento:

1. Por ejemplo, si queremos crear una tabla de `clientes` con los campos `dni`, `nombre` y `apellidos`. Crearemos una `data class` denominada `ClienteEntity` dentro del paquete `.data.room.ClienteEntity.kt`.
   A esta, le añadiremos a la anotación `@Entity` la propiedad `tableName` con el nombre de la tabla: `@Entity(tableName = "idTabla")`. Si no lo hiciéramos, la tabla tomaría el nombre de nuestra entidad.

   ```
   @Entity(tableName = "clientes")
   data class ClienteEntity(
       val dni: String,
       val nombre: String,
       val apellidos: String)
   ```

2. Deberemos decidir cual será nuestra **clave primaria** para añadir la anotación `@PrimaryKey` a la propiedad que decidamos.

   ```
   @Entity(tableName = "clientes")
   data class ClienteEntity(
       @PrimaryKey val dni: String,
       val nombre: String,
       val apellidos: String)
   ```

   Si necesitáramos una clave primaria compuesta por más de un campo, tendríamos que indicarlo con la propiedad **primaryKeys** de la anotación `@Entity`. De igual manera lo haríamos en el caso de necesitar una **clave ajena**, pero en este caso con la propiedad **foreignKeys**.

Por ejemplo, si tuviéramos otra tabla `pedidos` en la que relacionáramos el `dni` de la tabla `clientes` con el `dni` ajeno de los registros de esta tabla, `parentColumns` se referiría al `dni` de `clientes` mientras que `dni_cliente` en `childColumns` sería al de `pedidos`.

```
// Expresaríamos que un pedidos define una clave ajena dni en clientes.
@Entity(tableName = "pedidos",
        foreignKeys = arrayOf(
            ForeignKey(
                entity = ClienteEntity::class,
                parentColumns = arrayOf("dni"),
                // Nombre de la columna en la tabla pedidos
                childColumns = arrayOf("dni_cliente"),
                onDelete = CASCADE)
            )
        )
data class PedidoEntity(...)
```

No obstante, más adelante veremos como definir relaciones entre objetos en lugar de tablas. Además, ten en cuenta que la definición correcta de índices nos va a proporcionar mucha velocidad.

3. Es muy **buena práctica** usar la etiqueta `@ColumnInfo` **para que futuras modificaciones en la BD no afecten a mis entidades**. Esta propiedad sirve para personalizar la columna de la base de datos del atributo asociado. Podemos indicar, entre otras cosas, un nombre para la columna diferente al del atributo. Esto nos permite hacer más independiente la app de la BD. Además nos permite pasar del ***camelCasing*** que el como tendremos los nombres de propiedades compuestos en Kotlin a ***snake_casing*** que es el convenio estándar en las bases de datos. Por ejemplo, la clave ajena anterior de pedidos sería ...

```
    @ColumnInfo(name = "dni_cliente")
    val dniCliente: String,
```

En nuestro ejemplo, al final la `Entity` quedaría de la siguiente manera:

```
@Entity(tableName = "clientes")
data class ClienteEntity(
    @PrimaryKey
    @ColumnInfo(name = "dni")
    val dni: String,
    @ColumnInfo(name = "nombre")
    val nombre: String,
    @ColumnInfo(name = "apellidos")
    val apellidos: String)
```

4. Es posible que, a veces, quieras expresar una entidad o un objeto de datos como un solo elemento integral en la lógica de la base de datos, incluso si el objeto contiene varios campos.

En esas situaciones, puedes usar la anotación `@Embedded` para representar **un objeto cuyos subcampos quieras desglosar en una tabla**. Luego, puedes buscar los campos integrados tal como lo harías con otras columnas individuales. A este tipo de objetos, los denominaremos '***objetos incorporados***'
Podemos hacer una analogía entre los '***objetos incorporados***' con los que definimos en las BDOR (Bases de Datos Bbjeto-Relacionales de Oracle o PostgreSQL)
Por ejemplo, la clase `ClienteEntity` puede incluir un campo de tipo `Direccion`, que representa una composición de propiedades llamadas `calle`, `ciudad`, `pais` y `codigoPostal`. Para almacenar las **columnas compuestas por separado** en la tabla, incluye un campo `Direccion` en la clase `ClienteEntity` con anotaciones `@Embedded`, como se muestra en el siguiente fragmento de código:

```
data class Direccion(
    val calle: String?,
    val ciudad: String?,
    val pais: String?,
    @ColumnInfo(name = "codigo_postal")
    val codigoPostal: String?
)

@Entity(tableName = "clientes")
data class ClienteEntity(
    @PrimaryKey
    @ColumnInfo(name = "dni")
    val dni: String,
    @ColumnInfo(name = "nombre")
    val nombre: String,
    @ColumnInfo(name = "apellidos")
    val apellidos: String,
    // Marcamos como embebe @Embedded el campo a descomponer en la tabla
    @Embedded val direccion: Direccion?
)
```

Del código anterior, la tabla que representa un objeto `ClienteEntity` contendrá columnas con los siguientes nombres:

dni	nombre	apellidos	calle	ciudad	pais	codigo_postal

5. A veces, necesitas que la app almacene un tipo de datos personalizado en una sola columna de base de datos. Para admitir tipos personalizados, debes proporcionar convertidores de tipo, que son métodos que indican a Room cómo convertir tipos personalizados en tipos conocidos que Room pueda tratar. Para identificar los conversores de tipo, puedes usar la anotación `@TypeConverter`.

Supongamos que necesitas conservar instancias del tipo `LocalDate` definido en `org.jetbrains.kotlinx:kotlinx-datetime` para saber la **fecha** en que se hizo un pedido. Pero la base de datos para guardar fechas lo hace mediante un `Int` que representa el **TIMESTAMP**. Definiremos primero una clase denominada `RoomConverters` dentro del paquete `.data.room` con todos los métodos convertidores. Ojo, no importa el nombre del método, **sino su signatura** (parámetro de entrada y de salida). Por ejemplo para fecha podría ser:

```
// Estamos presuponiendo que nuestras fechas nunca son nulas.
class RoomConverters {
    @TypeConverter
    fun fromTimestamp(value: Int): LocalDate {
        return LocalDate.fromEpochDays(value)
    }

    @TypeConverter
    fun dateToTimestamp(date: LocalDate): Int {
        return date.toEpochDays()
    }
}
```

Importante

Cuando definamos la BD ya veremos como indicarle que aplique todas las conversiones definidas.

Ahora ya podemos definir nuestra clase `PedidoEntity` dentro del paquete `.data.room` ...

```
@Entity(tableName = "pedidos",
        foreignKeys = arrayOf(
            ForeignKey(
                entity = ClienteEntity::class,
                parentColumns = arrayOf("dni"),
                childColumns = arrayOf("dni_cliente"),
                onDelete = CASCADE)
            )
        )
data class PedidoEntity(
    // El id será autogenerado insertando un 0.
    @PrimaryKey (autoGenerate = true)
    @ColumnInfo(name = "id")
    val id: Int,

    @ColumnInfo(name = "dni_cliente")
    val dniCliente: Int,

    // Indicamos que siempre debe tener una fecha.
    // Esta en la DB se guardará como un Int.
    @NonNull
    @ColumnInfo(name = "fecha")
    val fecha: LocalDate
)
```

Definiendo relaciones entre Objetos

Usando los '***objetos incorporados***' podremos hacerlo de forma sencilla.

Relaciones Uno a Muchos entre objetos

Supongamos que queremos definir un nuevo objeto a recuparar que no es una entidad en la BD, pero **quiero que contenga un cliente con todos sus pedidos**. De forma análoga a los que tenemos en las entidades de **JPA**.

Deberemos definir la clase que exprese la relación y que se completará *'mapeará'* automáticamente cuando la usemos en nuestro **DAO**.

```
data class ClienteConPedidos(
    // Sabe recuperar el objeto embebido.
    @Embedded val cliente: ClienteEntity,
    @Relation(
          parentColumn = "dni",
          // Nombre de la columna en la parte del muchos
          entityColumn = "dni_cliente"
    )
    val pedidos: List<PedidoEntity>
)
```

Más adelante, veremos su utilidad y que hay formas más '*simples*' de hacerlo.

> **Información**
>
> Puedes saber más sobre como definir otros tipos de **relaciones entre objetos** con room en el siguiente **enlace**

Crear los Objetos de Acceso a la Base de Datos

Los **DAO** serán elementos de tipo **Interfaz**, en los cuales incluiremos los métodos necesarios de acceso y gestión a las entidades de la Base de Datos. En tiempo de compilación, Room generará automáticamente la implementaciones de DAOs que hayamos definido.

Es buena práctica, **definir un DAO por cada entidad que tengamos**, y en este definir las funcionalidades asociadas a esta entidad.

Para que una interfaz sea gestionada como DAO, habrá que añadir la anotación `@Dao`, siguiendo nuestro ejemplo tendríamos:

```
@Dao
interface ClienteDao
{
    ...
}
```

Para poder operar con la funcionalidad de Room, se necesitará hacer las llamadas fuera del hilo principal ya que las instancias de `RoomDatabase` son costosas en cuanto a tiempo, por lo que será necesario lanzar estas llamadas mediante corrutinas. **La manera recomendada es la de crear los métodos de la interfaces DAO como métodos de suspensión**.

Dentro de un DAO podemos crear dos tipos distintos de métodos, de conveniencia y de búsqueda.

Métodos de conveniencia

Estos métodos nos permiten realizar las operaciones básicas de **inserción**, **modificación** y **eliminación** de registros en la BD sin tener que escribir ningún tipo de código SQL. A estos métodos se le pasa la entidad sobre la que se quiera trabajar y es la propia librería Room la encargada de crear la sentencia SQL usando la clave primaria para la identificación del registro sobre el que se quiere operar. Deberán ir precedidos por las anotaciones **@Insert, @Delete o @Update** dependiendo de la necesidad. Por ejemplo, en nuestro caso, para la entidad `ClienteEntity` crearemos `ClienteDao` en su mismo paquete de la siguiente manera...

```
@Dao
interface ClienteDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(cliente : ClienteEntity)

    @Delete
    suspend fun delete(cliente : ClienteEntity)

    @Update
    suspend fun update(cliente : ClienteEntity)

    // Si no pasamos la entidad y lo que tenemos
    // es la PK deberemos hacer una consulta con @Query
    // pasando la PK como parámetro.
    @Query("DELETE FROM clientes WHERE dni = :dni")
    suspend fun deleteByDni(dni: String)
}
```

Como se puede ver en el anterior código, es una manera muy sencilla de realizar las operaciones básicas usando las anotaciones correspondientes y pasando como parámetro al método la Entidad sobre la que queremos operar. En el caso de la inserción podemos ver que se puede utilizar la propiedad `onConflict` para indicar si queremos reemplazar el elemento existente por el nuevo en caso de que coincida la **clave primaria** (en este caso el dni).

Métodos de búsqueda

Estos métodos serán los que crearemos para realizar consultas sobre la BD y tendrán que ir precedidos por la anotación **@Query**. Esta anotación permite que se añada como parámetro una cadena con la sentencia SQL para la consulta. Por ejemplo, podemos completar nuestro DAO de la siguiente manera....

```
@Dao
interface ClienteDao {
    // ... aquí van los métodos de conveniencia CRUD

    @Query("SELECT * FROM clientes")
    suspend fun get(): List<ClienteEntity>

    @Query("SELECT * FROM clientes WHERE dni IN (:dni)")
    suspend fun getFromDni(dni : String): ClienteEntity

    @Query("SELECT COUNT(*) FROM clientes")
    suspend fun count(): Int
}
```

Los dos primeros métodos de nuestro DAO, son dos métodos de consulta, el primero devuelve una **lista de Clientes** mientras que el segundo devuelve **un solo cliente**. Como ya hemos visto, Room permite pasar un parámetro o una lista de parámetros dentro de la propia consulta siempre que lo precedamos con `:` y que coincida en nombre con el parámetro que le llega al método.

Selección de un subconjunto de columnas

En muchas ocasiones no será necesaria toda la información de la tabla, sino que solo necesitaremos recuperar algunas de las columnas a modo de **DTO**. Aunque se puede recuperar toda la informació y posteriormente realizar un filtrado de esta, para ahorrar recursos es interesante consultar solamente los campos que se necesitan. Para ello, necesitaremos crear un objeto del tipo de columnas que queramos devolver, esto se podrá hacer usando una `data class` nueva para esos datos:

```
data class NombreApellidosDTO(
    @ColumnInfo(name = "nombre")
    val nombre: String,
    @ColumnInfo(name = "apellidos")
    val apellidos: String
)
```

Y solamente se tendrá que indicar en el Dao que el método correspondiente devolverá los datos de este tipo:

```
@Query("SELECT nombre, apellidos FROM clientes")
suspend fun getNombreApellido(): List<NombreApellidosDTO>
```

Por ejemplo, si quisiéramos recuperar el mapeo de la clase `ClienteConPedidos` que definimos en la relación uno a muchos entre objetos. Deberíamos marcar el método con anotación `@Transaction`, pues en su interior hay una anotación `@Relation` que se deberá completar antes de devolver la instancia del objetos.

```
@Transaction
// Fíjate que no indicamos la relación en la consulta ya está definida
// en el objeto a recuperar.
@Query("SELECT * FROM clientes")
// Fíjate que devuelve una lista de objetos ClienteConPedidos
suspend fun getPedidos(): List<ClienteConPedidos>
```

Sin embargo si la relación ya la definimos al definir las entidades podemos hacerlo mediante un `JOIN` sin necesidad de definir el objeto `ClienteConPedidos` ni ningún tipo de DTO de la siguiente manera.

```
@Query("SELECT * FROM clientes JOIN pedidos ON clientes.dni = pedidos.dni_cliente")
suspend fun obtenerClientesConPedidos(): Map<ClienteEntity, List<PedidoEntity>>
```

Fíjate que en este caso, el método devuelve un `Map` en el que la clave es un `ClienteEntity` y el valor todos los `PedidoEntity` asociados a este.

Crear la Base de Datos Room y consumirla

El último elemento que quedaría por crear sería la RoomDatabase. Este elemento es el que **se encargará de crear la base de datos a partir de las entidades definidas** y las operaciones que se realizarán sobre estas a partir de los DAOs de nuestra app. Por tanto, es el elemento que enlaza a los dos anteriores. Para ello crearemos una clase abstracta que deberá de heredar de `RoomDatabase` y a la que etiquetaremos con la anotación `@Database` con las propiedades versión y entities. La primera propiedad especificará la versión de la BD, mientras que en entities indicaremos la entidad o entidades asociadas a esta.

```
@Database(
    entities = [ClienteEntity::class, PedidoEntity::class],
    version = 1
)
// Indicaremos las conversiones de tipos que hemos definido para
// nuestra base de datos si las hay. Ej. LocalDate ↔ Long o Bitmap ↔ byte[]
@TypeConverters(RoomConverters::class)
abstract class TiendaDB: RoomDatabase() {

    // Cada RoomDatabase definirá métodos abstractos que devolverá
    // los tipos de **`DAO`** definidos y cuando instanciemos
    // la base de datos, obtendremos una instancia de
    // estos DAOs a través de ellos.
    abstract fun clienteDao() : ClienteDao
    abstract fun pedidoDao() : PedidoDao
}
```

Instanciar la RoomDatabase para su posterior funcionamiento

Una vez creados todos los componentes necesarios para el funcionamiento de Room Database, deberemos poder crear una instancias de la misma. Para ello, podemos definir un método de clase denominado `fun getDatabase(context: Context)` que usaremos posteriormente para que **Hilt** sepa crear una instancia del mismo e inyectarlo al obtener el DAO como veremos más adelante.

```
@Database(
    entities = [ClienteEntity::class, PedidoEntity::class],
    version = 1
)
@TypeConverters(RoomConverters::class)
abstract class TiendaDB: RoomDatabase() {
    abstract fun clienteDao() : ClienteDao
    abstract fun pedidoDao() : PedidoDao

    companion object {
        fun getDatabase(
            context: Context
        ) = Room.databaseBuilder(
            context,
            TiendaDB::class.java, "tienda"
        )
        .allowMainThreadQueries()
        .fallbackToDestructiveMigration()
        .build()
    }
}
```

Si te fijas `Room.databaseBuilder` precisa del **contexto** de la aplicación para poder crear el fichero `tienda` que contendrá la BD.

Preparando la Base de Datos y los DAOs con Hilt

En el fichero `AppModule.kt` dentro del paquete `.di` prepararemos la inyección de dependencias de la siguiente manera:

1. Para inyectar la BD definimos `provideTiendaDatabase` que llama al método estático `TiendaDB.getDatabase(context)` para instanciarla. Fíjate que el contexto de la aplicación se inyecta usando la anotación `@ApplicationContext` de Hilt.

   ```
   @Provides
   @Singleton
   fun provideTiendaDatabase(
       @ApplicationContext context: Context
   ): TiendaDB = TiendaDB.getDatabase(context)
   ```

2. Para inyectar los DAOs, definimos `provideClienteDao` y `providePedidoDao` que inyectan los DAOs de la BD. Fíjate que se inyecta la BD y se llama a los métodos abstractos que devuelven los DAOs.

```
@Provides
@Singleton
fun provideClienteDao(
    tiendaDB: TiendaDB
): ClienteDao = tiendaDB.clienteDao()

@Provides
@Singleton
fun providePedidoDao(
    tiendaDB: TiendaDB
): PedidoDao = tiendaDB.pedidoDao()
```

Inyectando los DAOs en los repositorios

Para inyectar los DAOs en los repositorios, deberemos añadirlos como propiedades privadas en los constructores de estos. Por ejemplo, en el caso del repositorio de `ClienteRepository` deberemos añadir el DAO de `ClienteDao` como parámetro del constructor.

```
class ClienteRepository @Inject constructor(
    private val clienteDao: ClienteDao
) {
    ...
}
```

> **Importante**
>
> Recuerda que, tal y como hicimos a principio de curso con las clases de prueba. Deberemos definir los métodos de extensión que transformen un `ClienteEntity` en un objeto `Cliente` del modelo para su uso en el ViewModel y viceversa.

Cargando datos de prueba en la BD

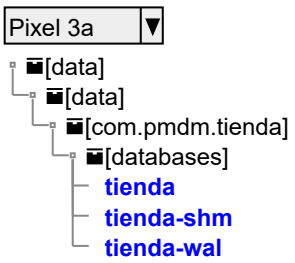
Este proceso se puede realizar de diferentes maneras, pero una forma de hacerlo de manera centralizada es en el momento de carga de la aplicación, por ejemplo invalidando el método `onCreate` de la clase `TiendaAplication` que recordemos hereda de `Application`.

Fíjate que a través de la anotación `@Inject` **Hilt** será capaz de crear una instancia de `ClienteDao` y por ende de `TiendaDB` en la propiedad `daoClientes` la primera vez que se use al ejecutar el método `onCreate` para cargar los datos de prueba.

```
@HiltAndroidApp
class TiendaAplication: Application() {

    @Inject
    lateinit var daoClientes: ClienteDao

    override fun onCreate() {
        super.onCreate()

        runBlocking {
            if (daoClientes.count() == 0) {
                daoClientes.insert(
                    ClienteEntity("12345678A", "Juan", "Pérez"))
                daoClientes.insert(
                    ClienteEntity("87654321B", "María", "García"))

        }
    }
}
```

Ubicación de la Base de Datos en el dispositivo

Si abrimos el **Device Explorer** de Android Studio, podremos ver que se ha creado un fichero `tienda` en la ruta `data/data/<nombre_paquete>/databases`. Este fichero es la base de datos que hemos creado y que se ha guardado en el dispositivo. Podremos ver su contenido si la copiamos con cualquier herramienta de gestión de bases de datos SQLite como **DBeaver**.



Aunque hay formas de mantener versiones de la BD y **migrar de una a otra**. La opción más sencilla es ante un cambio en alguna de las entidades sería eliminar la BD y volver a crearla. Para ello, podemos usar la propiedad `fallbackToDestructiveMigration()` en el método `databaseBuilder` que hemos usado para crear la BD o símplmente borrarla usando el **Device Explorer** y volver a ejecutar la app.

Inspeccionando y depurando la BD con Database Inspector

Existe la posibilidad de probar las consultas e inspeccionar la base de datos en tiempo de ejecución. Para ello, la herramienta de Android Studio **Database Inspector**.