Tema 5.1 - Room

Descargar estos apuntes pdf o html

Índice

Introducción

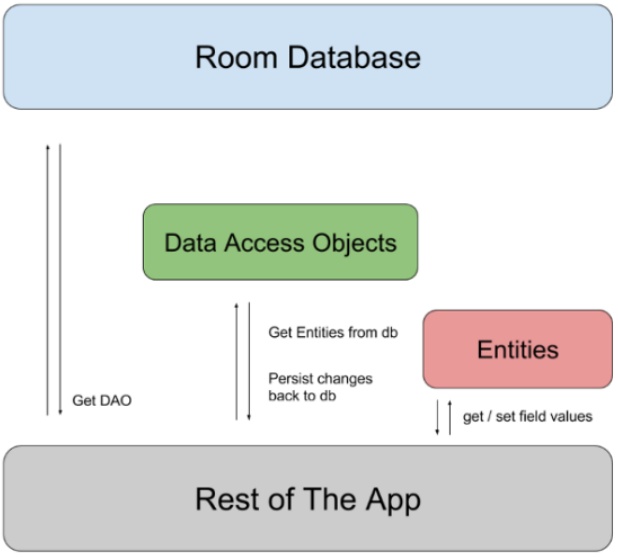
La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers**.

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 una capa de abstracción que facilita enormemente el proceso de codificación La biblioteca que nos va a permitir esta abstracción se distribuye como **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 utilizán 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.

✋ **Importante:** para poder usar la funcionalidad de Room y las anotaciones, deberemos añadir una serie de dependencias a build.gradle del módulo.
En las **dependencias**:

- Para las anotaciones deberemos añadir
  `kapt("androidx.room:room-compiler:$room_version")`
  No podemos olvidar añadir en plugins `id 'kotlin-kapt'`
- Para poder acceder a la funcionalidad de Room y enlazarla con las anotaciones
  `implementation "androidx.room:room-ktx:$room_version"`
  `implementation "androidx.room:room-runtime:$room_version"`
- La versión en el momento en que se editaron los apuntes era:
  `def room_version = "2.4.3" // En el script de Groovy`

Por tanto al `build.gradle` del módulo deberemos añadir algo similar a ...

```
plugins {
    ...
    id 'kotlin-kapt'
}

...

dependencies {
    ...
    def room_version = "2.4.3"
    kapt "androidx.room:room-compiler:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    implementation "androidx.room:room-runtime:$room_version"
    ...
}
```

Crear las entidades que definen nuestro modelo

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. Crearemos una clase en Kotlin que etiquetaremos como **data class**.

   > 📌 **Nota:** Recuerda que los objetos de este tipo de clases son inmutables y por tanto sus propiedades son de solo lectura o `val`. Además, generan automáticamente los métodos `copy`, `equals` y `toString`.

   Añadiremos a la anotación `@Entity` la propiedad `tableName` con el nombre de la tabla: `@Entity(tableName = "idTabla")`. Si no lo hicieramos, 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, // Clava primaria
       val nombre: String,
       val apellidos: String)
   ```

   Si necesitaramos una clave primaria compuesta por más de un campo, tendríamos que indicarlo con la propiedad primaryKeys de la etiqueta `@Entity`.
   De igual manera lo hariamos en el caso de necesitar una **clave ajena**, pero en este caso con la propiedad foreignKeys. Por ejemplo, si tuvieramos otra tabla `pedidos` en la que relacionaramos 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`.

```
// Expresariamos 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
                childColumns = arrayOf("dni_cliente"),
                onDelete = CASCADE)))
data class PedidoEntity(...)
```

No obstante más adelante veremos como definir relaciones entre objetos en lugar de tablas. No obstante recuerda 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. 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***'

   > 📌 **Nota:** Podemos hacer una analogía entre los '***objetos incorporados***' con los que definimos en las BDOR (Bases de datos objetos relacionales)

   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
    @Embedded val direccion: Direccion?)
```

La tabla que representa un objeto `ClienteEntity` contiene columnas con los siguientes nombres: `dni`, `nombre`, `apellidos`, `calle`, `ciudad`, `pais` y `codigo_postal`.

5. A veces, necesitas que la app almacene un tipo de datos personalizados 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 y a partir de ellos que Room puede conservar. Para identificar los conversores de tipo, puedes usar la anotación `@TypeConverter`.
   Supongamos que necesitas conservar instancias del tipo `Date` de Kotlin para saber la **fecha** en que se hizo un pedido. Pero la base de datos para guardar fechas lo hace mediante un `Long` que representa el TIMESTAMP.
   Definiremos primero una clase con todos los métodos convertidores. Ojo, no importa el id del método si no su signatura (parámetro de entrada y de salida) Por ejemplo para fecha podría ser:

```
// Estamos presuponiendo que nuestras fechas nunca son nulas.
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Long): Date {  // Convierte de Long a Date
        return value.let { Date(it) }
    }

    @TypeConverter
    fun dateToTimestamp(date: Date): Long {  // Convierte de Date a Long
        return date.time
    }
}
```

Ahora ya podemos definir nuestra clase `PedidoEntity` ...

```
@Entity(tableName = "pedidos",
        foreignKeys = arrayOf(ForeignKey(entity = ClienteEntity::class,
                parentColumns = arrayOf("dni"),
                // Nombre de la columna en la tabla
                childColumns = arrayOf("dni_cliente"),
                onDelete = CASCADE)))
data class PedidoEntity(
    @PrimaryKey (autoGenerate = true) // El id será autogenerado.

    @ColumnInfo(name = "id")
    val id: Int,

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

    // Indicamos que siempre debe tener una fecha.
    @NonNull
    @ColumnInfo(name = "fecha")
    val fecha: Date)
```

> ✋ **Importante:** Cuando definamos la BD ya veremos como indicarle que aplique todas las conversiones definidas.

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 tabla
          entityColumn = "dni_cliente"
    )
    val pedidos: List<PedidoEntity>
)
```

> ✋ **Importante:** 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 mayoritariament, 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 automaticamente 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 etiquetarla de esta manera, siguiendo nuestro ejemplo tendriamos:

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

Para poder operar con la funcionalidada 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.

```
@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
    @Query("DELETE FROM clientes WHERE dni = :dni")
    suspend fun deleteByDni(dni: String)
}
```

> 📌 **Nota:** 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.

```
@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
}
```

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 se puede ver en el código, 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. 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 TuplaNombreApellido(
    @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<TuplaNombreApellido>
```

Por ejemplo, si quisieramos 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>
```

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 Entities 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. Date ⇔ Long o Bitmap ⇔ byte[]
@TypeConverters(Converters::class)
abstract class TiendaDB: RoomDatabase() {
    abstract fun clienteDao() : ClienteDao
    abstract fun pedidoDao() : PedidoDao
}
```

> ✋ **Importante:** Cada Database tendrá un método abstracto que devolverá su tipo `DAO` correspondiente para posteriormente acceder a los métodos de este.

Instanciar la RoomDatabase para su posterior funcionamiento

Una vez creados todos los componentes necesarios para el funcionamiento de Room Database, podremos crear instancias en el lugar donde las necesitemos para la gestión de la base de datos, para ello crearemos primero un método estático `fun getDatabase(context: Context): AgendaDb` que me devolverá una única instancia de la BD de forma síncrona (como se ve de la línea 5 a la 19).

```
@TypeConverters(Converters::class)
abstract class AgendaDb: RoomDatabase() {
    abstract fun contactoDao() : ContactoDao

    companion object {
        @Volatile
        private var db: AgendaDb? = null

        fun getDatabase(context: Context): AgendaDb {
            return db ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                        context,
                        AgendaDb::class.java, "agenda"
                ).build()
                db = instance
                instance
            }
        }
    }
}
```

Como necesitamos el contexto podemos definir un objeto **`TiendaApp`** que herede de **`Application()`** para crear la instancia de forma global y accesible desde el **`MainActivity`**.

```
class TiendaApp : Application() {
    val database: AgendaDb by lazy { AgendaDb.getDatabase(this) }
}
```

Para que se cree una instancia del objeto **`TiendaApp`** deberemos modificar nuestro **`AndroidManifest.xml`** para indicarle que está asociado a nuestra App añadiendo el siguiente atributo a la etiqueta **`<application>`**

```
...
<application
    android:name="paquete_donde_este_definido.TiendaApp"
    ...>

...
</application>
```

Para obtener una instancia del DAO en algún punto de nuestra aplicación podemos hacer.

```
// Suponiendo que estamos definiendo un ViewModel

private val dao = (applicationInstance as TiendaApp).database.contactoDao()

// Un posible uso de este objeto para realizar una
// consulta con paso de parámetro sería de la siguiente manera...

viewModelScope.launch {
    if (dao.getFromDni(cliente.dni)?.dni != cliente.dni)
        dao.insert(cliente)
    else
        withContext(Dispatchers.Main) {
            Toast.makeText(
                            this@MainActivity,
                            "El registro ya existe",
                            Toast.LENGTH_LONG
                        ).show()
        }
}
```