

Apuntes

Descargar estos apuntes [pdf](#) y [html](#)

Tema 11.1. Persistencia de Datos I Room

Índice

1. [Acceso a bases de datos locales. SQLite con Room](#)
 1. [Introducción](#)
 2. [Componentes de Room](#)
 3. [Crear las Entidades](#)
 1. [Métodos de conveniencia](#)
 2. [Métodos de búsqueda](#)
 1. [Selección de un subconjunto de columnas](#)
 4. [Crear la Base de Datos Room](#)
 5. [Instanciar la RoomDatabase para su posterior funcionamiento](#)

Acceso a bases de datos locales. SQLite con Room

Introducción

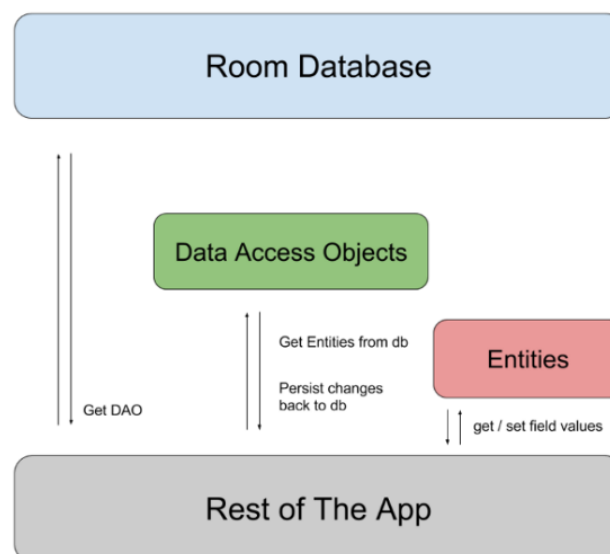
La plataforma Android proporciona dos herramientas principales para el almacenamiento y consulta de datos estructurados: la base de datos **SQLite** y **Content Providers** (lo trabajaremos en temas posteriores).

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 creamos, 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.

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

En los **plugins** deberemos añadir `id 'kotlin-kapt'` que nos permitirá trabajar con la herramienta de anotaciones para Kotlin.

En las **dependencias**:

- para las anotaciones deberemos añadir
`kapt("androidx.room:room-compiler:$room_version")`
- Para poder acceder a la funcionalidad de Room y enlazarla con las anotaciones
`implementation "androidx.room:room-ktx:$room_version"`
`implementation "androidx.room:room-runtime:$room_version"`
`annotationProcessor "androidx.room:room-compiler:$room_version"`

Crear las Entidades

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**. *Este tipo de clases están formadas por el constructor con los atributos que se etiquetaran como **val** para los inmutables y **var** para los mutables. Este tipo de clases genera internamente las propiedades y métodos básicos necesarios incluido el **toString***

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

Añadiremos a la Anotación **@Entity** la propiedad **tableName** con el nombre de la tabla: `@Entity(tableName = "CLIENTES")`

2. Deberemos decidir cual será nuestra clave primaria para añadir la anotación correspondiente al atributo que decidamos.

```
@PrimaryKey
val dni: String,
```

Si necesitáramos una clave primaria compuesta por más de un campo, tendríamos que indicarlo con la propiedad **primaryKeys** de la etiqueta **@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 **Compras** en la que relacionáramos el **dni** de la tabla **clientes** con el **dni** de los registros de esta tabla, **parentColumns** se referiría al **dni** de **Cientes** mientras que **childColumns** sería al de **Compras**.*

```
@Entity(
    tableName = "COMPRAS",
    foreignKeys = @ForeignKey(entity = Cliente.class,
                             parentColumns = "dni",
                             childColumns = "dni",
                             onDelete = CASCADE)
)
```

3. Es muy buena práctica usar la etiqueta `@ColumnInfo` 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,
)
```

Crear los Objetos de Acceso a la Base de Datos

Los **DAO** serán elementos de tipo Interfaz mayoritariamente 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 las 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 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.

```

@Dao
interface ClienteDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(cliente: ClienteEntity)
    @Delete
    suspend fun delete(cliente: ClienteEntity)
    @Update
    suspend fun update(cliente: ClienteEntity)
}

```

📌 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 {

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

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

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(cliente: ClienteEntity)
    @Delete
    suspend fun delete(cliente: ClienteEntity)
    @Update
    suspend fun update(cliente: 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 información 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>
```

Crear la Base de Datos Room

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],
    version = 1
)
abstract class ClienteDB: RoomDatabase() {
    abstract fun clienteDao(): ClienteDao
}


```

Si quisieramos añadir más de una tabla en la BD tendríamos que realizar algo como lo siguiente (ejemplo de developer):

```

// Song and Album are classes annotated with @Entity.
@Database(version = 1, entities = [Song::class, Album::class])
abstract class MusicDatabase : RoomDatabase {
    // SongDao is a class annotated with @Dao.
    abstract fun getSongDao(): SongDao
    // AlbumDao is a class annotated with @Dao.
    abstract fun getArtistDao(): ArtistDao
    // SongAlbumDao is a class annotated with @Dao.
    abstract fun getSongAlbumDao(): SongAlbumDao
}

```

 **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 haremos lo siguiente:

```

val db = Room.databaseBuilder(
    applicationContext,
    ClienteDB::class.java, "database-name"
).build()

```

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

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

 **Compón la aplicación de los ejemplos anteriores y prueba su funcionamiento**

 **Ejercicio propuesto cursor con recycler**