

Apuntes

Descarregar aquests apunts [pdf](#) y [html](#)

Tema 11.1. Persistencia de Dades I Room

Índice

1. [Accés a bases de dades locals. SQLite amb Room](#)
 1. [Introducció](#)
 2. [Components de Room](#)
 3. [Crear les entitats que defineixen el nostre model](#)
 1. [Definint relacions entre Objectes](#)
 1. [Relacions Un a Molts entre objectes](#)
 4. [Crear els Objectes d'Accés a la Base de dades](#)
 1. [Mètodes de conveniència](#)
 2. [Mètodes de cerca](#)
 1. [Selecció d'un subconjunt de columnes](#)
5. [Crear la Base de dades Room i consumir-la](#)
6. [Instanciar la RoomDatabase per al seu posterior funcionament](#)

Accés a bases de dades locals. SQLite amb Room

Introducció

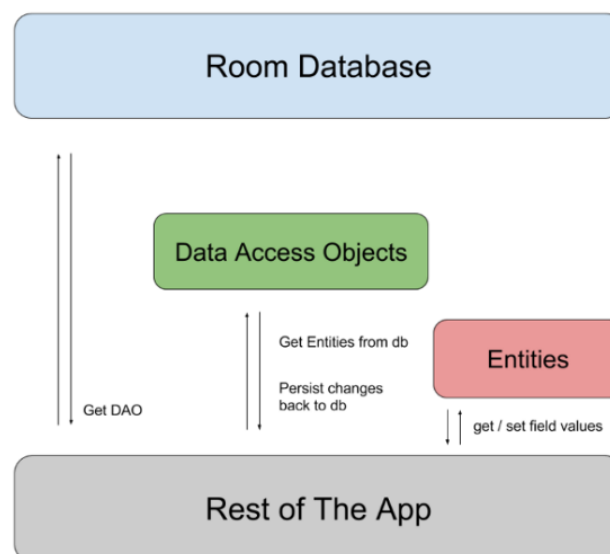
La plataforma Android proporciona dues eines principals per a l'emmagatzematge i consulta de dades estructurades: la base de dades **SQLite** i **Content Providers** (ho treballarem en temes posteriors).

Ens centrarem en **SQLite**, encara que no entrarem ni en el disseny de BBDD relacionals ni en l'ús avançat de la BBDD. Per a conèixer tota la funcionalitat de SQLite es recomana usar la documentació oficial. El problema de treballar directament amb aquesta API és que és un treball a baix nivell que pot portar a errors en temps d'execució.

Per aquest motiu Android Jetpack ha proporcionat una capa d'abstracció que facilita enormement el procés de codificació. La biblioteca que ens permetrà aquesta abstracció es distribueix com **Room**.

Components de Room

Per a treballar amb **Room** hem de familiaritzar-nos amb la seua arquitectura que està composta per tres parts principals:




- **Entity** (Entities), seran les classes que definiran les entitats de la nostra Base de dades, que corresponen amb cada taula que la formen.

- **Daos** (Data Access Objects), en aquest cas seran les classes abstractes o interfícies on estaran definits els mètodes que ens permetran l'operativitat (inserció, consulta, etc) amb cadascuna de les taules de la Base de dades.
- **RoomDatabase** (Room Database), conté la Base de dades i l'accés a aquesta, alhora que uneix els dos anteriors conceptes.

Una vegada que es creuen aquests tres elements, l'app podrà accedir als Doneu-vos a través de les instàncies de **Room Database** que creem, de forma més estable que amb l'accés directe a la API de *QLite.

L'ús de Room es fa a través de **Anotacions** afegides a les classes i interfícies, les principals són:

- **@Entity**, indica que la classe a la qual s'atribueix ha de ser tractada com una entitat.
- **@Dao**, ha d'afegir-se a les interfícies que volem que siguin nostres Doneu-vos. Dins d'aquestes interfícies es utilitzen altres anotacions que veurem més endavant.
- **@Database**, s'afegirà a les Room Database i indicarà que és una Database, aquestes classes han de ser abstractes i heretar de **RoomDatabase**.

 **Important:** per a poder usar la funcionalitat de Room i les anotacions, haurem d'afegir una sèrie de dependències a **build.gradle** del mòdul.
En les **dependències**:

- Per a les anotacions haurem d'afegir

```
kapt("androidx.room:room-compiler:$room_version")
```

 No podem oblidar afegir en *plugins `id 'kotlin-kapt'`
- Per a poder accedir a la funcionalitat de *Room i enllaçar-la amb les anotacions

```
implementation "androidx.room:room-ktx:$room_version"
implementation "androidx.room:room-runtime:$room_version"
```
- La versió en el moment en què es van editar les anotacions era:

```
def room_version = "2.4.3" // Al script de Groovy
```

Per tant al **build.gradle** del mòdul haurem d'afegir una cosa 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 les entitats que defineixen el nostre model

El primer passe a realitzar serà el de crear les diferents taules que tindrà la nostra base de dades. Cada taula correspondrà amb una classe `Entity` etiquetada com `@Entity` en la qual afegirem els camps corresponent a la taula. Usarem un exemple senzill per a entendre el funcionament:

1. Crearem una classe en Kotlin que etiquetarem com **data class**.

✦ **Nota:** Recorda que els objectes d'aquesta mena de classes són immutables i per tant les seues propietats són de només lectura o `val`. A més, generen automàticament els mètodes `copy`, `equals` i `toString`.

Afegirem a l'anotació `@Entity` la propietat `tableName` amb el nom de la taula: `@Entity(tableName = "idTabla")`. Si no ho fem, la taula prendria el nom de la nostra entitat.

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

2. Haurem de decidir com serà la nostra **clau primària** per a afegir l'anotació `@PrimaryKey` a la propietat que decidim.

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

Si necessitem una clau primària composta per més d'un camp, hauríem d'indicar-ho amb la propietat `primaryKeys` de l'etiqueta `@Entity`.

D'igual manera ho faríem en el cas de necessitar una **clau aliena**, però en aquest cas amb la propietat `foreignKeys`. Per exemple, si tinguérem una altra taula `pedidos` en la qual relacionariem la columna `dni` de la taula `clientes` amb el `dni` alie dels registres d'aquesta taula, `parentColumns` es referiria al `dni` de `clientes` i `dni_cliente` a `childColumns` seria el de `pedidos`.

```
// Expressariem que un pedidos define una clave aliena dni en clientes.
@Entity(tableName = "pedidos",
    foreignKeys = arrayOf(ForeignKey(entity = ClienteEntity::class,
        parentColumns = arrayOf("dni"),
        // Nom de la columna a la taula pedidos a la bd
        childColumns = arrayOf("dni_cliente"),
        onDelete = CASCADE)))
data class PedidoEntity(...)
```

No obstant això més endavant veurem com definir relacions entre objectes en lloc de taules. No obstant això recorda que la definició correcta d'índexs ens proporcionarà molta velocitat.

3. És molt **bona pràctica** usar l'etiqueta `@ColumnInfo` perquè futures modificacions en la BD no afecten les meues entitats. Aquesta propietat serveix per a personalitzar la columna de la base de dades de l'atribut associat. Podem indicar, entre altres coses, un nom per a la columna diferent del de l'atribut. Això ens permet fer més independent l'app de la BD. En el nostre exemple, al final la `Entity` quedaria de la següent 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. És possible que, a vegades, vulgues expressar una entitat o un objecte de dades com un sol element integral en la lògica de la base de dades, fins i tot si l'objecte conté diversos camps. En aqueixes situacions, pots usar l'anotació `@Embedded` per a representar **un objecte que els seus sub-camps vulgues desglossar en una taula**. Després, pots buscar els camps integrats tal com ho faries amb altres columnes individuals. A aquesta mena d'objectes, els denominarem '**objectes incorporats**'

✦ **Nota:** Podem fer una analogia entre els '**objectes incorporats**' amb els quals definim en les BDOR (Bases de dades objectes relacionals)

Per exemple, la classe `ClienteEntity` pot incloure un camp de tipus `Direccion`, que representa una composició de propietats anomenades `calle`, `ciudad`, `pais` i `codigoPostal`. Per a emmagatzemar les **columnes compostes per**

separat en la taula, inclou un camp **Direccion** en la classe **ClienteEntity** amb anotacions **@Embedded**, com es mostra en el següent fragment de codi:

```
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,  
    // Marquem com @Embedded el camp a descompondre  
    @Embedded val direccion: Direccion?)
```

La taula que representa un objecte **ClienteEntity** conté columnes amb els següents noms: **dni**, **nombre**, **apellidos**, **calle**, **ciudad**, **pais** i **codigo_postal**.

5. A vegades, necessites que l'app emmagatzeme un tipus de dades personalitzades en una sola columna de base de dades. Per a admetre tipus personalitzats, has de proporcionar convertidors de tipus, que són mètodes que indiquen a Room com convertir tipus personalitzats en tipus coneguts i a partir d'ells que Room pot conservar. Per a identificar els convertidors de tipus, pots usar l'anotació **@TypeConverter**.

Suposem que necessites conservar instàncies del tipus **Date** de Kotlin per a saber la data en què es va fer una comanda. Però la base de dades per a guardar dates ho fa mitjançant un **Long** que representa el **TIMESTAMP**.

Definirem primer una classe amb tots els mètodes convertidors. Ull, no importa l'aneu del mètode si no la seua signatura (paràmetre d'entrada i d'eixida) Per exemple per a data podria ser:

```
// Estem pressuposant que les nostres dates mai són nul·les.
class Converters {
    @TypeConverter
    fun fromTimestamp(value: Long): Date { // Convertix de Long a Date
        return value.let { Date(it) }
    }


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

Ara ja podem definir la nostra classe **PedidoEntity** ...

```
@Entity(tableName = "pedidos",
    foreignKeys = arrayOf(ForeignKey(entity = ClienteEntity::class,
        parentColumns = arrayOf("dni"),
        // Nom de la columna a la taula
        childColumns = arrayOf("dni_cliente"),
        onDelete = CASCADE)))
data class PedidoEntity(
    @PrimaryKey (autoGenerate = true) // El id serà auto-generat.
    @ColumnInfo(name = "id")
    val id: Int,

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

    @NonNull
    @ColumnInfo(name = "fecha")
    // Indiquem que sempre ha de tindre una data.
    val fecha: Date)
```

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

Definint relacions entre Objectes

Usant els '**objectes incorporats**' podrem fer-ho de manera senzilla.

Relacions Un a Molts entre objectes

Suposem que volem definir un nou objecte a obtindre que no és una entitat a la BD, però vull que continga un client amb totes les seues comandes. Anàlogament als que tenim en les entitats de JPA.

Haurem de definir la classe que expresse la relació i que es completarà '*mapejarà*' automàticament quan la vaja'm a usar en el nostre **DAO**.

```
data class ClienteConPedidos(  
    // Sap obtindre el objecte embegut  
    @Embedded val cliente: ClienteEntity,  
    @Relation(  
        parentColumn = "dni",  
        // Nom de la columna a la taula  
        entityColumn = "dni_cliente"  
    )  
    val pedidos: List<PedidoEntity>  
)
```

👉 **Important:** Pots saber més sobre com definir altres tipus de **relacions entre objectes** amb room en el següent [enllaç](#)

Crear els Objectes d'Accés a la Base de dades

Els **DAO** seran elements de tipus Interfície majoritàriament en els quals inclourem els mètodes necessaris d'accés i gestió a les entitats de la Base de dades. En temps de compilació, Room generarà automàticament la implementacions de Doneu-vos que hàgem definit.

És bona pràctica, **definir un DAO per cada entitat que tinguem**, i en aquest definir les funcionalitats associades a aquesta entitat.

Perquè una interfície siga gestionada com DAO, caldrà etiquetar-la d'aquesta manera, seguint el nostre exemple tindriem:

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

Per a poder operar amb la funcionalitat de Room, es necessitarà fer les crides fora del fil principal ja que les instàncies de **RoomDatabase** són costoses quant a temps, per la qual cosa serà necessari llançar aquestes crides mitjançant '*corrutines*'. **La manera recomanada és la de crear els mètodes de la interfícies DAO com a mètodes de suspensió.**

Dins d'un DAO podem crear dos tipus diferents de mètodes, de conveniència i de cerca.

Mètodes de conveniència

Aquests mètodes ens permeten realitzar les operacions bàsiques d'inserció, modificació i eliminació de registres en la BD sense haver d'escriure cap mena de codi SQL. A aquests mètodes se li passa l'entitat sobre la qual es vulga treballar i és la pròpia llibreria Room l'encarregada de crear la sentència SQL usant la clau primària per a la identificació del registre sobre el qual es vol operar. Hauran d'anar precedits per les anotacions **@Insert**, **@Delete** o **@Update** depenent de la necessitat.

```

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

    @Delete
    suspend fun delete(cliente : ClienteEntity)

    @Update
    suspend fun update(cliente : ClienteEntity)

    // Sino passem l'entitat i el que tenim
    // és la PK haurem de fer una consulta amb @Query
    @Query("DELETE FROM clientes WHERE dni = :dni")
    suspend fun deleteByDni(dni: String)
}

```

📌 **Nota:** Com es pot veure en l'anterior codi, és una manera molt senzilla de realitzar les operacions bàsiques usant les anotacions corresponents i passant com a paràmetre al mètode l'Entitat sobre la qual volem operar. En el cas de la inserció podem veure que es pot utilitzar la propietat **onConflict** per a indicar si volem reemplaçar l'element existent pel nou en cas que coincidisca la clau primària (en aquest cas el *dni).

Mètodes de cerca

Aquests mètodes seran els que crearem per a realitzar consultes sobre la BD i hauran d'anar precedits per l'anotació **@Query**. Aquesta anotació permet que s'afija com a paràmetre una cadena amb la sentència SQL per a la consulta.

```

@Dao
interface ClienteDao {

    // ... ací van el mètodes del CRUD

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

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

```

Els dos primers mètodes del nostre DAO, són dos mètodes de consulta, el primer retorna una llista de Clients mentre que el segon retorna un sol client. Com es pot veure en el codi, Room permet passar un paràmetre o una llista de paràmetres dins

de la pròpia consulta sempre que el precedim amb `:` i que coincidisca en nom amb el paràmetre que li arriba al mètode.

Selecció d'un subconjunt de columnes

En moltes ocasions no serà necessària tota la informació de la taula, sinó que només necessitarem recuperar algunes de les columnes. Encara que es pot recuperar tota la informació i posteriorment realitzar un filtrat d'aquesta, per a estalviar recursos és interessant consultar solament els camps que es necessiten. Per a això necessitarem crear un objecte del tipus de columnes que vulguem retornar, això es podrà fer usant una data class nova per a aqueixes dades:

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

I solament s'haurà d'indicar en el DAO que el mètode corresponent retornarà les dades d'aquest tipus:

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

Per exemple, si volguérem recuperar el mapatge de la classe `ClienteConPedidos` que definim en la relació un a molts entre objectes. Hauríem de marcar el mètode amb anotació `@Transaction`, perquè en el seu interior hi ha una anotació `@Relation` que es deurà completar abans de retornar la instància de l'objectes.

```
@Transaction  
// Fixa't que no indiquem la relació en la consulta ja està definida  
// en l'objecte a recuperar.  
@Query("SELECT * FROM clientes")  
// Fixa't que retorna una llista d'objectes ClienteConPedidos  
suspend fun getPedidos(): List<ClienteConPedidos>
```

Crear la Base de dades Room i consumir-la

L'últim element que quedaria per crear seria la [RoomDatabase](#). Aquest element és el que s'encarregarà de crear la base de dades a partir de les *'Entities'* definides i les operacions que es realitzaran sobre aquestes a partir dels *Doneu-vos* de la nostra app. Per tant és l'element que enllaça als dos anteriors. Per a això crearem una classe abstracta que haurà d'heretar de `RoomDatabase` i a la qual etiquetarem amb l'anotació `@Database` amb les propietats *versió* i *entities*. La primera propietat especificarà la versió de la BD, mentre que en *entities* indicarem l'entitat o entitats associades a aquesta.

```
@Database(
    entities = [ClienteEntity::class, PedidoEntity::class],
    version = 1
)
// Indicarem les conversions de tipus que hem definit per a
// la nostra base de dades si n'hi ha. Ex. Dona't ⇄ Long o Mapa de bits ⇄ byte[]
@TypeConverters(Converters::class)
abstract class TiendaDB: RoomDatabase() {
    abstract fun clienteDao() : ClienteDao
    abstract fun pedidoDao() : PedidoDao
}
```

👉 **Important:** Cada Database tindrà un mètode abstracte que retornarà el seu tipus `DAO` corresponent per a posteriorment accedir als mètodes d'aquest.

Instanciar la RoomDatabase per al seu posterior funcionament

Una vegada creats tots els components necessaris per al funcionament de Room Database, podrem crear instàncies en el lloc on les necessitem per a la gestió de la base de dades, per a això crearem primer un mètode estàtic

`fun getDatabase(context: Context): AgendaDb` que em retornarà una única instància de la BD de manera síncrona (com es veu de la línia 5 a la 19).

```

1  @TypeConverters(Converters::class)
2  abstract class AgendaDb: RoomDatabase() {
3      abstract fun contactoDao() : ContactoDao
4
5      companion object {
6          @Volatile
7          private var db: AgendaDb? = null
8
9          fun getDatabase(context: Context): AgendaDb {
10             return db ?: synchronized(this) {
11                 val instance = Room.databaseBuilder(
12                     context,
13                     AgendaDb::class.java, "agenda"
14                 ).build()
15                 db = instance
16                 instance
17             }
18         }
19     }
20 }

```

Com necessitem el context podem definir un objecte **TiendaApp** que herete de **Application()** per a crear la instància de manera global i accessible des del **MainActivity**.

```

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

```

Perquè es cree una instància de l'objecte **TiendaApp** deurem modificar el nostre **AndroidManifest.xml** per a indicar-li que està associat a la nostra App afegint el següent atribut a l'etiqueta **<application>**.

```

...
<application
    android:name="paquet_a_on_este_definit.TiendaApp"
    ...>

...
</application>

```

Per a obtenir una instància del DAO en algun punt de la nostra aplicació podem fer.

```
// Suposant que estam definint un ViewModel

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

// Un possible ús d'aquest objecte per a realitzar una
// consulta amb pas de paràmetre seria de la següent 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()
        }
}
```

 **Compon l'aplicació dels exemples anteriors i prova el seua funcionament**

 **Exercici proposat cursor amb recycler**