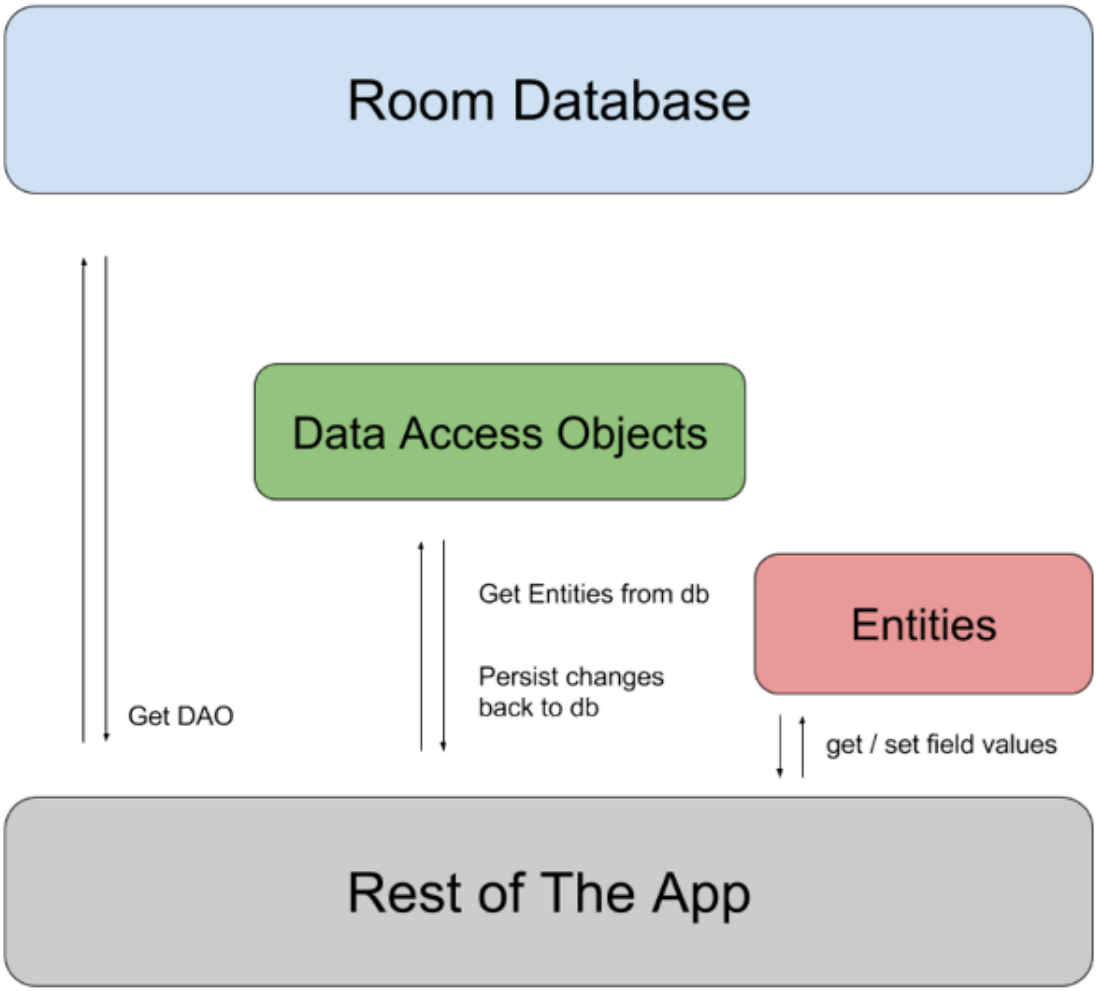
UA04. COMUNICANDO...

4.5. Room

Room es un ORM (Object- Relational mapping) que nos permitirá trabajar de una forma más sencilla con bases de datos SQL.



La imagen anterior nos muestra el funcionamiento de dicha herramienta que, aunque parezca complicado al principio, es muy fácil de entender cuando nos pongamos a ello. Tendremos una base de datos que le devolverá a nuestra app los **Data Access Objects** (DAO) estos son los encargados de persistir la información en la base de datos y de

devolvernos las **entities**, que serán las encargadas de devolvernos la información que hemos ido almacenando.

Necesitas incluir el plugin kotlin-kapt y las dos siguientes dependencias:

```
id 'kotlin-kapt'

implementation 'androidx.recyclerview:recyclerview:1.2.1'
implementation 'androidx.room:room-ktx:2.4.3'
kapt 'androidx.room:room-compiler:2.4.3'
```

Entidades

En Room, cada entidad (Entity <https://developer.android.com/training/data-storage/room/defining-data>) representa una tabla debe ser una clase. En nuestro caso, vamos a definir una tabla que nos permita almacenar los datos de un disco de música. En concreto, vamos a crear una clase llamada **TaskEntity** Y la anotaremos con **@Entity**.
Las anotaciones convierten clases normales en elementos interpretables por **Room**. Habrá que utilizarlas continuamente para configurar la base de datos y conseguir una estructura y propiedades a nuestro gusto.

```
@Entity(tableName = "task_entity")
data class TaskEntity (
        @PrimaryKey(autoGenerate = true)
        var id:Int = 0,
        var name:String = "",
        var isDone:Boolean = false
)
```

La anotación **@Entity** la utilizamos para añadirle un nombre a nuestra entidad como tabla de la base de datos. Cada base de datos puede contener una o varias tablas y cada una persiste un modelo diferente.

La anotación **@PrimaryKey (autoGenerate = true)** está diciendo que la variable **id** es un valor que se autogenera al crear un objeto de esta clase y que no podrá repetirse. Es un valor único con el cual podremos localizar un objeto concreto.

DAOs

También necesitaremos una **interface DAO** (Data Access Object), que será la que nos permita acceder a la tabla, **hay que crear una por cada entidad**. En nuestro caso, esta clase será **TaskDao.kt** y la anotaremos con **@Dao**.

Los objetos DAO facilitan mucho el acceso a la BD. Aquí se declararán los métodos que interactuarán con las tablas:

- **@Query**: Se hacen consultas directamente a la base de datos usando SQL. En este ejemplo haremos dos muy sencillas, pero se pueden hacer cosas impresionantes.
- **@Insert**: Se usará para insertar entidades a la base de datos, a diferencia de las **@Query** no hay que hacer ningún tipo de consulta, sino pasar el objeto a insertar.
- **@Update**: Actualizan una entidad ya insertada. Solo tendremos que pasar ese objeto modificado y ya se encarga de actualizarlo. ¿Cómo sabe que objeto hay que modificar? Pues por nuestro **id**, la **PrimaryKey**.
- **@Delete**: Como su propio nombre indica borra de la tabla un objeto que le pasemos.

```
@Dao
interface TaskDao {
    @Query("SELECT * FROM task_entity")
    fun getAllTasks(): MutableList    @Insert    ...    @Update
}
```

Android Studio permite **autocompletar** y hacer validaciones sencillas de las queries que definamos.

A las consultas también podemos pasarles argumentos. Sólo hay que indicar el argumento en la función:

```
@Query("SELECT * FROM task_entity WHERE id = :id")
fun findById(id: Int): TaskEntity
```

Gracias a **onConflict**, podemos definir la estrategia de inserción si hubiera conflictos: **IGNORE**, **ABORT** o **REPLACE**:

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
fun insert(tasks: List<TaskEntity>)
```

Base de datos

Por último, definiremos una clase que nos permitirá la creación de la base de datos como tal. Para ello crearemos el archivo **TaskDatabase.kt** y usaremos la anotación **@Database** .

Lo recomendado es que esta clase siga un patrón **singleton**. Aquí es donde se indicarán las entidades que formaran la base de datos, la forma de crearla/destruirla, la versión, etc.

```
@Database(entities = arrayOf(TaskEntity::class), version = 1)
    abstract class TasksDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao
}
```

Lo primero que debemos fijarnos es en la anotación **@Database**, que especifica que la entidad será una lista de **TaskEntity** (entidad que ya hemos creado) y que la **versión es 1**.

La clase extendida **RoomDatabase()**, es una clase que tenemos gracias a importar la dependencia de **Room** en nuestro **gradle**. Para finalizar tiene una sola función que hace referencia al **Dao** que hemos creado anteriormente, si tuviésemos más Dao's pues habría que implementarlos ahí también.

Para crear la instancia de Room, en el **Application** por ejemplo:

```
val room: TasksDatabase = Room
.databaseBuilder(this, TasksDatabase::class.java, "tasks")
.build()
```

Y para usarlo:

```
val tasks = room.TaskDao().getAll()
```

MisNotasApp

Ahora vamos a crear una clase algo diferente. Esta clase va a extender de **Application()** y eso significa que será lo primero en ejecutarse al abrirse la aplicación.

```
class MisNotasApp: Application() {
    companion object {
        lateinit var database: TasksDatabase
    }
    override fun onCreate() {
        super.onCreate()
        MisNotasApp.database =  Room.databaseBuilder(this, Task
    }
}
```

Aquí, la instancia de **database** necesitará tres parámetros, el **contexto** (**this**), la clase de nuestra base de datos (**TasksDatabase**) y el nombre que le pondremos, en este caso, **"tasks-db"**.

Para que esta clase se lance al abrir la app debemos ir al **AndroidManifest.xml** y añadir **android:name=".MisNotasApp"** dentro de la etiqueta

activity_main

Aunque tengamos varias clases por detrás, nuestra app solo tendrá un **layout**, una sola vista. La idea es crear algo sencillo y usable, por lo que me contaremos con una barra superior donde añadir las tareas y luego un **RecyclerView** donde se muestren todas.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/r
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/background_light"
    tools:context="com.cursokotlin.misnotas.UI.MainActivity">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvTask"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@+id/rlAddTask"/>
    <RelativeLayout
        android:id="@+id/rlAddTask"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:elevation="10dp"
        android:layout_margin="10dp"
        android:background="@android:color/white">
        <EditText
            android:id="@+id/etTask"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:hint="añade una tarea"
            android:layout_alignParentLeft="true"
            android:layout_toLeftOf="@+id/btnAddTask"
            />
        <Button
            android:id="@+id/btnAddTask"
            android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:layout_alignParentRight="true"
            android:text="Añadir"/>
    </RelativeLayout>
</RelativeLayout>
```

MainActivity

Con la vista preparada, es el momento de empezar a generar la lógica. Empezamos creando las variables necesarias.

```
lateinit var recyclerView: RecyclerView
lateinit var tasks: MutableList<TaskEntity>
```

Ahora nos vamos al **OnCreate** de nuestra actividad, lo primero que haremos será instanciar tasks como una **arrayList** y acto seguido llamaremos a la función **getTasks()** que vamos a crear. Esta función será la encargada de acceder a nuestro **DAO** para hacer la primera consulta, recuperar todas las entidades que el usuario tenga guardadas. Así que por ahora dejamos el **OnCreate** así.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    tasks = ArrayList()
    getTasks()
}
```

Con lo realizado hasta ahora, ya tendríamos suficiente para crear y manejar la base de datos. De entre las posibles complicaciones que nos podemos encontrar en este primer contacto con Room, solo hay que tener una cosa importante en cuenta: **las operaciones de acceso a la base de datos no pueden realizarse en el hilo principal de la app**. Si hacemos esto, la app se cerrará abruptamente.

Tendremos que detenernos para conocer un poco el uso de los **hilos**. Los hilos son los que permiten la multitarea de un dispositivo, por ejemplo en un hilo puedes ir guardando información asíncronamente (por detrás) mientras el usuario sigue haciendo cosas.

En Android, el **hilo principal** es el encargado de la parte visual de la aplicación, por lo que **no nos deja acceder a la base de datos** desde ahí, por ello crearemos un **hilo secundario** que de modo **asíncrono** hará la petición a la base de datos y recuperará la información que necesitemos.

Todas estas operaciones de base de datos deberían hacerse fuera del hilo principal. Esto se puede realizar manualmente, o usar alguna de las integraciones que ya se nos dan implementadas.

Lo único que necesitamos es añadir la palabra **suspend** delante de las funciones del DAO, ¡y listo!

```
@Query("SELECT * from task_entity")
suspend fun getAll(): List
```

En el **MainActivity**, creamos el método **getTasks()** como corrutina de la siguiente manera:

```
fun getTasks()= runBlocking{
        launch {
            tasks = MisNotasApp.database.taskDao().getAllTasks()
            setUpRecyclerView(tasks)
        }
    }
```

Antes de continuar, vamos a crear un nuevo layout llamado item_task.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/andrc
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_margin="10dp">
    <CheckBox
        android:id="@+id/cbIsDone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dp"
        android:layout_marginEnd="10dp" />
    <TextView
        android:id="@+id/tvTask"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
```

```
        android:textSize="18sp"
        tools:text="Test"/>
</LinearLayout>
```

Nos quedará una celda muy sencilla. La idea es que cuando pulsemos el **Checkbox,** se actualice en la base de datos el objeto y, si hacemos click en cualquier otra parte de la vista, se borre de la base de datos.

Crearemos una nueva clase, **TaskAdapter**, a la que le pasaremos 3 parámetros: la lista de tareas que tenemos almacenadas en nuestra base de datos y funciones. Estas funciones nos permitirán recuperar el evento del click en cada una de las celdas, ya sea la vista completa o un componente concreto.

```
class TasksAdapter(
val tasks: List,
val checkTask: (TaskEntity) -> Unit,
val deleteTask: (TaskEntity) -> Unit) : RecyclerView.Adapter() {
```

En el **onBindViewHolder** añadimos el **setOnClickListener** a cada uno de los componentes que nos interese, en este caso se ha puesto al **checkbox** y, para controlar el click en cualquier otra parte de la vista, podemos acceder a **itemView** que nos devuelve la celda completa.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val item = tasks[position]
    holder.bind(item, checkTask, deleteTask)
}
```

Y ya completamos **TaskAdapter** con los dos métodos que nos faltan **onCreateViewHolder** y **getItemCount**.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): Vi
    val layoutInflater = LayoutInflater.from(parent.context)
    return ViewHolder(layoutInflater.inflate(R.layout.item_task, pare
}override fun getItemCount(): Int {return tasks.size}
```

Para finalizar, añadimos la clase ViewHolder que solo tendrá de novedad en la función bind donde, a través de .isChecked, podemos iniciar la vista con el checkbox marcado o no, así que comprobaremos si está a true nuestra entidad y si es así pues lo marcamos.

Una vez configurada la celda, le añadimos .setOnClickListener a nuestro checkBox y al itemView que es el componente completo pasando el propio objeto.

```
class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val tvTask = view.findViewById(R.id.tvTask)
    val cbIsDone = view.findViewById(R.id.cbIsDone)

    fun bind(task: TaskEntity, checkTask: (TaskEntity) -> Unit, delet
        tvTask.text = task.name
        cbIsDone.isChecked = task.isDone
        cbIsDone.setOnClickListener{checkTask(task)}
        itemView.setOnClickListener { deleteTask(task) }
    }
}
```

Con nuestro adapter completo es el paso de configurarlo desde MainActivity con nuestra función setUpRecyclerView() a la cual le pasaremos la lista de tareas que hemos recuperado.

```
fun setUpRecyclerView(tasks: List) {
    adapter = TasksAdapter(tasks, { updateTask(it) }, {deleteTask(it)
    recyclerView = findViewById(R.id.rvTask)
    recyclerView.setHasFixedSize(true)
    recyclerView.layoutManager = LinearLayoutManager(this)
    recyclerView.adapter = adapter
}
```

Debemos fijarnos que al instanciar el adapter le pasamos tres parámetros:

1. lista de tareas,
2. updateTask(it)
3. deleteTask(it).

Estos no son parámetros sino métodos que tendremos en el MainActivity, que serán llamados automáticamente cuando se ejecute el evento del click que configuramos en el adapter.

```
fun updateTask(task: TaskEntity) =runBlocking {
      launch {
         task.isDone = !task.isDone
         MisNotasApp.database.taskDao().updateTask(task)
       }
   }fun deleteTask(task: TaskEntity)=runBlocking{     launch{
```

En el método deleteTask() debemos hacer algo más, para empezar, buscaremos en nuestra lista tasks la posición del item que vamos a borrar para tener una referencia. Para ello usaremos la función de las listas indexOf(item) que nos devolverá dicha posición y la almacenamos en una variable, luego borraremos el objeto de la base de

datos y de nuestra lista y acabaremos en el hilo principal avisando al adapter que hemos removido un objeto, pasándole la posición.

Últimos retoques

Vamos a configurar el botón de añadir tareas, que lo que hará será crear un objeto **Task**, almacenarlo en base de datos y luego añadirlo a la lista que tiene el adapter. Lo primero que haremos será ir a nuestro DAO y añadir una nueva función de insertar.

```
@Insert
fun addTask(taskEntity : TaskEntity):Long
```

Simplemente recibirá un objeto **TaskEntity** y lo añadirá a la base de datos. Fijaros que devuelve un **Long**, eso es porque nos dará automáticamente la **ID** del item añadido.

Nos vamos a nuestro **onCreate** del **MainAcitivity** y añadimos lo siguiente.

```
btnAddTask.setOnClickListener {
addTask(TaskEntity(name = etTask.text.toString()))}
```

Simplemente le hemos asignado al evento del click un método llamado **addTask()** al cual le pasamos un objeto nuevo con el texto de la celda. Dicha función añadirá a la base de datos la tarea, luego recuperemos dicha tarea y la añadiremos a la lista del **adapter**.

```
fun addTask(task:TaskEntity)= runBlocking{
      launch {
          val id = MisNotasApp.database.taskDao().addTask(task)
          val recoveryTask = MisNotasApp.database.taskDao().getTask
              tasks.add(recoveryTask)
              adapter.notifyItemInserted(tasks.size)
```

```
                clearFocus()
                hideKeyboard()
            }
        }
```

Así que lo que estamos haciendo es añadir la tarea y luego recuperamos dicho objeto a través de **getTaskById** pasándole la **ID** que nos devuelve **addTask**. Obviamente debemos añadir a nuestro DAO la función de recuperar el item.

```
@Query("SELECT * FROM task_entity where id like :arg0")
fun getTaskById(id: Long): TaskEntity
```

Cuando hemos acabado de realizar esto, en el hilo principal añadimos el objeto recuperado a la lista, le decimos al adapter que hemos añadido un objeto nuevo a través de **adapter.notifyItemInserted** (hay que pasarle pa posición, como es el último objeto añadido, podemos saber cual es recuperando el tamaño de la lista) y luego los métodos **clearFocus()** y **hideKeyboard()** simplemente nos quitarán el texto del **editText** y bajarán el teclado.

```
fun clearFocus(){
   etTask.setText("")
}
fun Context.hideKeyboard() {
    val inputMethodManager = getSystemService(Activity.INPUT_METHOD_S
    inputMethodManager.hideSoftInputFromWindow(currentFocus.windowTok
}
```