UA04. COMUNICANDO...

4.5.1. Ejemplo

Vamos a desglosar la explicación del código anterior por documentos, desde Gradle, hasta el MainActivity que ejecuta la aplicación:

Este archivo debe modificarse añadiendo varias líneas en la zona **plugins** y en **dependencies**:

```
id 'kotlin-kapt' // En plugins

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

Esto cargará en nuestra aplicación el plugin kapt <https://runebook.dev/es/docs/kotlin/docs/reference/kapt> y las dependencias para

utilizar los métodos de Room <https://developer.android.com/jetpack/androidx/releases/room> .

Además, incorporamos los métodos del **RecyclerView** que usaremos para mostrar la información.

Este archivo contiene las **entidades** (tablas) en forma de **data class**.

Cada entidad en Room debe tener la anotación **@Entity** delante de la definición de la clase. Además, se le puede añadir parámetro como:

- **tableName:** nombre de la tabla
- **primaryKeys:** para combinar dos o más campos como campo clave.
- **ignoreColumns**: para ignorar algún campo que no deseemos cargar.

Debe existir una clase por cada entidad.

```
@Entity(tableName = "task_entity")
data class TaskEntity (
    @PrimaryKey(autoGenerate = true)
    var id:Int = 0, // Id de la tarea
    var name:String = "", // Nombre de la tarea
    var isDone:Boolean = false // Booleano que indica si la tarea est
)
```

La anotación **@Entity** la utilizamos para añadirle un nombre a nuestra entidad como tabla de la base de datos.

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.

La interfaz **TaskDao** será la primera capa sobre la base de datos y encargada de la comunicación con esta mediante sentencias SQL. **Hay que crear una por cada entidad.**

Bajo la notación **@**, se declara la función que se ejecutará en cada caso. Estas funciones o métodos, deben declararse como **suspend** para poder ejecutarlos en hilos diferentes al principal ya que, las lecturas/escrituras a bases de datos, consumen muchos recursos.

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.
- **@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**. Si usamos o**nConflict**, podremos decidir la acción en caso de conflicto: **Ignore**, **Abort** o **Replace**
- **@Delete**: Como su propio nombre indica **borra** de la tabla un objeto que le pasemos.

```
@Dao
interface TaskDao {
    @Query("SELECT * FROM task_entity")
    suspend fun getAllTasks(): MutableList< TaskEntity>  // Función c

    @Insert
```

```
    suspend fun addTask(taskEntity : TaskEntity):Long    // Función c

    @Query("SELECT * FROM task_entity where id like :id")
    suspend fun getTaskById(id: Long): TaskEntity        // Función c

    @Update
    suspend fun updateTask(task: TaskEntity):Int         // Función c

    @Delete
    suspend fun deleteTask(task: TaskEntity):Int         // Función c
}
```

Estas funciones se declararán en el MainActivity.

Esta clase nos permite la creación de la Base de Datos.

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

Como se puede observar, a la notificación @Database se le pasa como parámetro un array de Entidades (tipo clase) y se especifica la versión 1 de la base de datos (en el futuro, si se amplia la base de datos, se puede crear otra versión y el programa se adaptará a la base de datos adecuada)

El objeto base de datos **TasksDatabase** hereda la estructura Room y contiene una única función, **taskDao()** que devuelve la interface **TaskDao** encargada del manejo de la base de datos.

Esta clase va a extender de **Application()** y 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, TasksDatab
    }
}
```

La instancia **database** necesita tres parámetros:

1. el **contexto** (**this**),
2. la clase de nuestra base de datos (**TasksDatabase**) declarada de forma global y accesible de forma estática,
3. el nombre que le pondremos a la base de datos, 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.

```
< ?xml version="1.0" encoding="utf-8"?>
< RelativeLayout xmlns:android="http://schemas.android.com/apk/res/ar
    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>
```

```
< ?xml version="1.0" encoding="utf-8"?>
< LinearLayout xmlns:android="http://schemas.android.com/apk/res/andr
    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>
```

Adapter <https://keepcoding.io/blog/metodo-onbindviewholder-en-android/> es una herramienta que proporciona el sistema operativo Android, con la cual tenemos la posibilidad de transformar una cosa en otra diferente. De manera que podríamos decir que hace referencia a una especificación que realizamos en el programa, la cual permite que la información almacenada por medio de código se adapte a lo que el usuario verá en la pantalla.

Cuando creamos un **RecyclerViewAdapter**, la clase padre nos pide que debemos configurar tres métodos diferentes, los cuales son:

- **getItemCount**: este método le dice al **RecyclerView** la cantidad de vistas que tenemos que renderizar. En este caso, el método **getItemCount** le indicará el tamaño de los elementos que están almacenados en la lista.
- **onCreateViewHolder**: este es otro de los métodos que nos pide la clase padre. En pocas palabras, es el que va a funcionar como el **contenedor** en el que estaría la vista.
- **onBindViewHolder**: este es el último de los métodos, el cual nos devuelve un **View Holder** con el fin de renderizarle la posición.

Crearemos una nueva clase, **TaskAdapter**, a la que le pasaremos 3 parámetros:

1. la lista de tareas que tenemos almacenadas en nuestra base de datos,
2. función que se ejecuta al pulsar el checkbox (**check**)
3. función que se ejecuta al pulsar la tarea (**delete**)

Estas funciones nos permitirán recuperar el evento del click en cada una de las celdas. Devuelve la vista de elementos en la lista.

```
class TasksAdapter(
    val tasks: List< TaskEntity>,                    // Objeto Lista c
    val checkTask: (TaskEntity) -> Unit,             // chequeo de tar
    val deleteTask: (TaskEntity) -> Unit             // borrado de tar
) : RecyclerView.Adapter< TasksAdapter.ViewHolder>() {    // Devuelve

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

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.item_task,
    }

    override fun getItemCount(): Int {
        return tasks.size     // Devuelve el número de tareas de la l
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val tvTask = view.findViewById< TextView>(R.id.tvTask)
        val cbIsDone = view.findViewById< CheckBox>(R.id.cbIsDone)

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

```
class MainActivity : AppCompatActivity() {

    lateinit var recyclerView: RecyclerView
    lateinit var adapter: TasksAdapter
    lateinit var tasks: MutableList< TaskEntity>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        tasks = ArrayList()          // Se prepara la lista
        getTasks()                   // Se carga la lista de tareas a
        findViewById< Button>(R.id.btnAddTask).setOnClickListener {
            addTask(TaskEntity(name = findViewById< EditText>(R.id.et
    }

    fun clearFocus(){
        findViewById< EditText>(R.id.etTask).setText("") // Borra el
    }

    fun Context.hideKeyboard() {    // Oculta el teclado de texto
        val inputMethodManager = getSystemService(Activity.INPUT_METH
        inputMethodManager.hideSoftInputFromWindow(currentFocus?.winc
    }

    fun getTasks()= runBlocking {       // Corrutina que saca de la b
        launch {                        // Inicio del hilo
            tasks = MisNotasApp.database.taskDao().getAllTasks()    /
            setUpRecyclerView(tasks)        // se pasa la lista a la
        }
```

```
    }

    fun addTask(task:TaskEntity)= runBlocking{  // Corrutina que añac
        launch {
            val id = MisNotasApp.database.taskDao().addTask(task)   /
            val recoveryTask = MisNotasApp.database.taskDao().getTask
            tasks.add(recoveryTask) // Añade al final de la lista, el
            adapter.notifyItemInserted(tasks.size)  // El adaptador r
            clearFocus()        // Se elimina el texto del et ...
            hideKeyboard()      // y se oculta el teclado
        }
    }

    fun updateTask(task: TaskEntity) = runBlocking{
        launch {
            task.isDone = !task.isDone  // Marca o desmarca el checkb
            MisNotasApp.database.taskDao().updateTask(task) // Actual
        }
    }

    fun deleteTask(task: TaskEntity)= runBlocking{
        launch {
            val position = tasks.indexOf(task)  // Busca la posición
            MisNotasApp.database.taskDao().deleteTask(task) // ... y
            tasks.remove(task)      // Finalmente, la elimina de la l
            adapter.notifyItemRemoved(position) // El adaptador notif
        }
    }

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