ROOM [KOTLIN]

Android nos ofrece una alternativa a trabajar con SQLite directamente: la librería **Room**. Nos ofrece una capa de abstracción que nos evita las complejidades de SQLite. Para usar **Room** m será preciso incluir primero en el **build.gradle** tanto el plugin kotlin-kapt como las dependencias de la librería.

El plugin:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'kotlin-kapt'
}
```

Y las dependencias:

```
dependencies {
    def room_version : String = "2.5.2"
    implementation "androidx.room:room-runtime:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
}
```

Room tiene una arquitectura de tres capas cuyas clases se marcan con anotaciones:

- **Entity**: Representa una tabla dentro de la base de datos
- **DAO (Data Access Object)**: Interfaz con los métodos de acceso.
- **RoomDatabase**: Clase que define el conjunto de Entidades y DAO que se van a usar.

Entidades

Room utiliza le patrón ORM para mapear las tablas de la base de datos en Kotlin. Gracias a esto, solo tendremos que definir una clase como **@Entiy** y sus propiedades. Room se encargará de los detalles de la tabla asociada.

```
@Entity
data class User (@PrimaryKey val id: Long, val name: String, val surname: String)
```

Por defecto, el nombre de la tabla será el nombre de la clase en minúsculas, lo mismo que el nombre de las columnas en el caso de las propiedades. No obstante, existen otras etiquetas que podemos usar para añadir características adicionales a las tablas:

- **@Entity**: permite mediante la propiedad tableName darle un nombre diferente al de por defecto a la tabla.
- **@PrimaryKey**: indica el atributo que será clave primaria. Podemos indicar también mediante autoGenerate = true que la clave primaria se incremente automáticamente.
- **@ColumnInfo**: define atributos adicionales de una columna de la tabla, como asignarle un nombre diferente, un índice o un valor por defecto.
- **@Ignore**: previene que el atributo se almacene en la base de datos.
- **@Index**: genera un índice sobre la columna
- **@ForeignKey**: Define una clave externa sobre una clave primaria de otra Entidad.

A modo de ejemplo:

```
@Entity (tableName = "ciudades", indices = [Index("name")])
data class Ciudades (
    @PrimaryKey (autoGenerate = true) val id: Long,
    val name: String,
    @ColumnInfo (name = "area") val superficie : Float,
    @Ignore val picture : Bitmap?
    )
```

DAO

Para cada una de las entidades desarrollaremos una interfaz tipo DAO con los métodos que permitirán realizar las consultas y modificaciones de datos. Los métodos se deberán marcar con las correspondientes anotaciones **@Query**, **@Insert**, **@Update** y **@Delete**. No existe limitación al número de veces que puedas emplear cada una de las etiquetas en una clase; no obstante, un mismo método no puede tener diferentes etiquetas.

A modo de ejemplo:

```
@Dao
interface CiudadesDao {

    @Query ("SELECT * FROM ciudades ORDER BY name")
    fun getAll () : List <Ciudades>
}
```

En los proyectos web convencionales se suelen emplear frameworks y otras tecnologías para gestionar bases de datos muy grandes con muchos datos. En estos entornos se generan DAO de forma estandarizada mediante patrones de diseño que suelen terminar por disponer de multitud de métodos simples que pueden no usarse jamás. Estas decisiones de diseño suelen tomarse por dos motivos: homogeneidad en el producto final y posibilidad de ampliar el programa en el futuro para atender a nuevas necesidades. En las **apps** en cambio no tiene sentido añadir métodos de base de datos 'extra' si no vas a utilizarlos. Esta lógica se puede aplicar a cualquier otro aspecto de la programación con Android.

Database

Una vez definidas las Entidades y los DAO, sólo falta definir la base de datos que los contendrá. Para ello se crea una clase abstracta que herede de **RoomDatabase** y la marcaremos con la anotación **@Database**.

```
@Database (entities = [Ciudades::class], version = 1)
abstract class MyRoomDatabase  : RoomDatabase (){

    abstract fun ciudadesDao () : CiudadesDao

}
```

En la anotación **@Database** se incluirán todas las entidades que se van a emplear en nuestra base de datos, así como su versión actual. A demás, se incluyen los métodos para recuperar cada uno de los DAO definidos. Estos métodos serán funciones abstractas (sin argumentos) con las que se obtendrá acceso a las clases anotadas con **@Dao**.

A partir de aquí, ya podemos obtener una instancia de Database llamando a el método **Room.databaseBuilder()**, con lo que ya podremos trabajar con la base de datos.

```
val db = Room.databaseBuilder( context: this,
    MyRoomDatabase::class.java,
     name: "myDatabase").build()

val listado : List <Ciudades> = db.ciudadesDao().getAll();
```

Iniciar esta clase es costos, y por lo general, solo necesitamos una instancia para trabajar con la base de datos. Por tanto, es mejor usar un patrón singleton para tener una única instancia de esa clase, y reutilizarla en vez de instanciarla cada vez que se necesite. Por ejemplo, podríamos convertir **MyRoomDatabase** para que quede así:

```
@Database (entities = [Ciudades::class], version = 1)
abstract class MyRoomDatabase  : RoomDatabase () {

    companion object {
        // Volatile = la variable puede ser usada pro diferentes hilos
        @Volatile private var instance : MyRoomDatabase? = null
        private val LOCK = Any()

        // Codigo a ejecutar al construirse la clase. Funciona como un semaforo, usando
        // synchronized (LOCK) para que no puedan crearse varias instancias a la vez.
        operator fun invoke (context : Context) = instance ?: synchronized (LOCK){
            instance ?: buildDatabase (context).also { instance = it}
        }

        private fun buildDatabase (context : Context) = Room.databaseBuilder (context,
            MyRoomDatabase::class.java, name: "myDatabase").build()
    }

    abstract fun ciudadesDao(): CiudadesDao
}
```

El acceso a base de datos con Room se hace siempre en un **hilo separado**, y si no lo hacemos así se generará una excepción. Por tanto, para crear una instancia de la base de datos se hace:

```
val db = MyRoomDatabase ( context: this)

GlobalScope.launch(Dispatchers.IO){
    val listado : List <Ciudades> = db.ciudadesDao().getAll()
    GlobalScope.launch(Dispatchers.Main){
        // Cargamos el listado en un adapter o lo que queramos hacer

    }
}
```

Práctica 41

[Kotlin] Realiza una app que sea capaz de almacenar sus canciones favoritas. Se guardará el título de la canción, el autor, y la URL de YouTube donde se puede escuchar. La app debe permitir listar todas las canciones, añadir una canción, borrar una canción y modificar sus datos. Las canciones deberán aparecer en formato lista.

Práctica 42

[Kotlin] Lo mismo que antes, pero al pulsar una canción de la lista se hará un *intent* y se reproducirá la canción en el navegador.

Práctica 43

[Java] Lo mismo que antes, pero al pulsar una canción de la lista se hará un *intent* y se reproducirá la canción en otro activity que tendrá un componente **Media Player**. Este ejercicio propuesto es para que investigues un rato si ya has terminado otras tareas. Esto lo veremos más adelante. Que te diviertas.