IES DR LLUÍS SIMARRO
ENSENYAMENT PÚBLIC
XATIVA

Programación multimedia y dispositivos móviles

UD09. Persistencia de datos II: Base de datos local con Room.

Desarrollo de Aplicaciones Multiplataforma

Anna Sanchis Perales

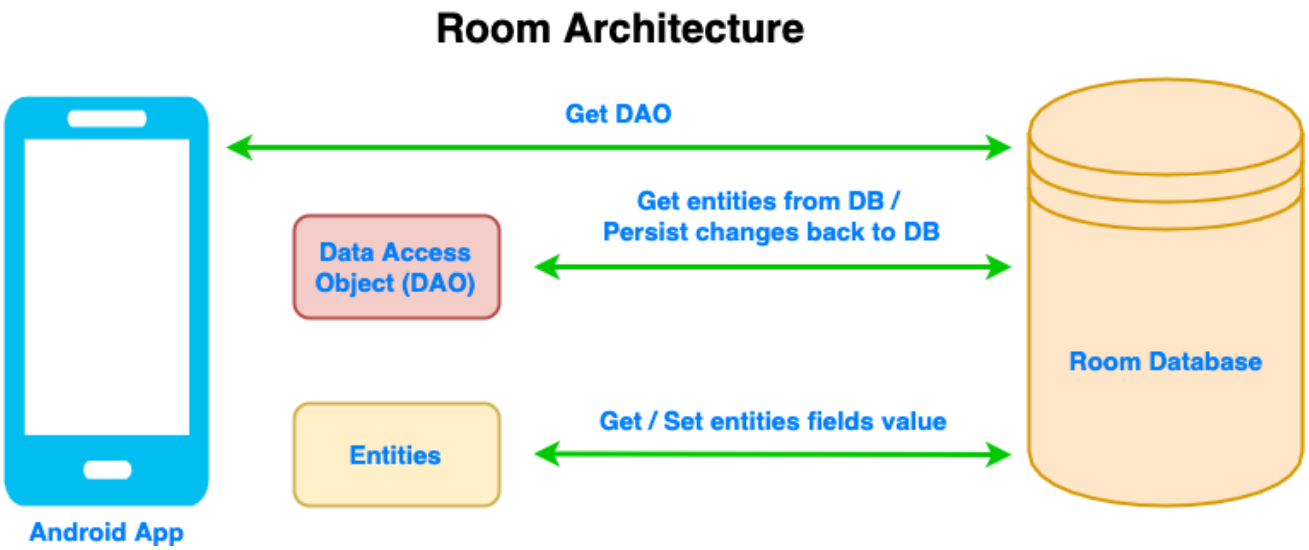
ÍNDICE

1. ¿QUÉ ES ROOM?

Room es una biblioteca de persistencia que simplifica y abstrae el acceso a la base de datos SQLite. Fue introducida por Google como parte de la arquitectura de componentes de Android.

Room proporciona una capa de abstracción sobre SQLite que facilita la creación, la consulta y la gestión de bases de datos en aplicaciones Android. Algunas de las características clave de Room incluyen:

- Entidades: Representan las tablas de la base de datos. Cada entidad en Room se mapea a una tabla en la base de datos SQLite.
- DAO (Data Access Object): Define métodos que acceden a la base de datos. Los DAOs permiten realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en las entidades.
- Base de datos: Define la base de datos y actúa como punto de acceso principal para la conexión a la base de datos. La clase de base de datos anotada con @Database generalmente extiende RoomDatabase.
- Anotaciones: Room utiliza anotaciones de Kotlin para generar código relacionado con la base de datos en tiempo de compilación, lo que facilita el desarrollo y evita errores comunes.



https://developer.android.com/training/data-storage/room?hl=es-419

https://dam.org.es/ejemplos-room/

En esta sección, se presenta un ejemplo de implementación de una base de datos con Room de una sola entidad de datos y un DAO único. Para ello vamos a crear un proyecto de nombre **Tema9App1**. La entidad de datos será la de User. Según vayamos avanzando en la explicación utilizaremos esta aplicación para aprender todo lo que se vaya viendo.

Para usar la librería ROOM seguiremos los siguientes pasos:

1. Añadir dependencia de Room a Gradle.
2. Crear una Entity por cada modelo.
3. Crear DAOs
4. Crear subclase de RoomDatabase.

2. AÑADIR DEPENDENCIA A GRADLE

Para usar Room en tu app, agrega las siguientes dependencias al archivo build.gradle de la app:

```
dependencies {
val roomVersion = "2.6.1"

implementation("androidx.room:room-runtime:$roomVersion")
ksp("androidx.room:room-compiler:$roomVersion")
    ...
}
```

Añade también en build.gradle de la app el plugin ksp:

```
plugins {
   id("com.android.application")
   id("org.jetbrains.kotlin.android")
   id("com.google.devtools.ksp")
```

```
}
```

Y en el archivo build.gradle de Project, añade también ksp:

```
plugins {
    id("com.android.application") version "8.2.0" apply false
    id("org.jetbrains.kotlin.android") version "1.9.10" apply false
    id("com.google.devtools.ksp") version "1.9.21-1.0.15" apply false
}
```

3. ENTIDAD DE DATOS

Cuando usas la biblioteca de persistencias de Room para almacenar los datos de tu app, defines entidades para representar los **objetos** que deseas almacenar. Cada **entidad** corresponde a una **tabla** en la base de datos de Room asociada y cada **instancia** de una entidad representa una **fila** de datos en la tabla correspondiente.

Es un mapeado del comando CREATE de una tabla en SQLite.

Define cada entidad de Room como una clase con **@Entity** como anotación. Una entidad de Room incluye campos para cada columna de la tabla correspondiente en la base de datos, incluidas una o más columnas que conforman la clave primaria.

El siguiente código es un ejemplo de una entidad simple que define una tabla User con columnas para el ID, el nombre y el apellido.

Esta será la entidad que crearemos en nuestro proyecto de ejemplo:

```
@Entity
data class UserEntity(
   @PrimaryKey val id: Int,

   val firstName: String?,
   val lastName: String?
)
```

De forma predeterminada, Room usa el nombre de la clase como el nombre de la tabla de la base de datos. Si quieres que la tabla tenga un nombre diferente, configura la propiedad tableName de la anotación @Entity. De manera similar, Room usa los nombres de campos como nombres de columna en la base de datos de forma predeterminada. Si quieres que una columna tenga un nombre diferente, agrega la anotación @ColumnInfo al campo y establece la propiedad name. En el siguiente ejemplo, se muestran los nombres personalizados para tablas y columnas:

```
@Entity(tableName = "users")
data class UserEntity (
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
```

```
)
```

3.1 Cómo definir una clave primaria

Cada entidad de Room debe definir una clave primaria que identifique de manera única cada fila en la tabla de base de datos correspondiente. La manera más sencilla de hacerlo es anotar una sola columna con @PrimaryKey:

```
@PrimaryKey val id: Int
```

4. OBJETO DE ACCESO A DATOS (DAO)

Cuando usas la biblioteca de persistencias Room para almacenar los datos de tu app, interactúas con los datos almacenados mediante la definición de objetos de acceso a datos o DAO. Cada DAO incluye métodos que ofrecen acceso abstracto a la base de datos de tu app. En el tiempo de compilación, Room genera automáticamente implementaciones de los DAOs que definas.

Si usas DAOs para acceder a la base de datos de tu app en lugar de compiladores de búsquedas o búsquedas directas, puedes conservar la separación de problemas, un principio arquitectónico importante. Los DAOs también te permiten simular con mayor facilidad el acceso a la bases de datos cuando pruebas tu app.

4.1 Definir un DAO

Puedes definir cada DAO como una interfaz o una clase abstracta. Por lo general, debes usar una interfaz para casos de uso básicos. En cualquier caso, siempre debes anotar tus DAOs con @Dao. Los DAOs no tienen propiedades, pero definen uno o más métodos para interactuar con los datos de la base de datos de tu app.

El siguiente código es un ejemplo de un DAO simple que define métodos para insertar, borrar y seleccionar objetos User en una base de datos de Room:

```
@Dao
interface UserDao {
    @Insert
    fun insertAll(UserEntityList: List<UserEntity>)

    @Delete
    fun delete(user: UserEntity)

    @Query("SELECT * FROM UserEntity")
    fun getAll(): List<UserEntity>
}
```

Existen dos tipos de métodos DAO que definen interacciones de bases de datos:

- Métodos de conveniencia que te permiten insertar, actualizar y borrar filas en tu base de datos sin escribir ningún código de SQL.
- Métodos de búsqueda que te permiten escribir tu propia consulta en SQL para interactuar con la base de datos.

En las siguientes secciones, se muestra el modo para usar ambos tipos de métodos DAO y así definir las interacciones de la base de datos que necesita tu app.

4.2 Inserción

La anotación @Insert te permite definir métodos que insertan sus parámetros en la tabla adecuada en la base de datos. En el siguiente código, se muestran ejemplos de métodos @Insert válidos que insertan uno o más objetos User en la base de datos:

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insertUsers(vararg users: UserEntity)

    @Insert
    fun insertBothUsers(user1: UserEntity, user2: UserEntity)

    @Insert
    fun insertUsersAndFriends(user: UserEntity, friends:
List<UserEntity>)
}
```

Cada parámetro para un método @Insert debe ser una instancia de una clase de entidad de datos de Room con @Entity como anotación o una colección de instancias de clase de entidad de datos, cada una de las cuales dirige a una base de datos. Cuando se llama a un método @Insert, Room inserta cada instancia de entidad pasada en la tabla de base de datos correspondiente.

Si el método @Insert recibe un solo parámetro, puede mostrar un valor long, que es el nuevo rowId para el elemento insertado. Si el parámetro es un array o una colección, muestra un array o una colección de valores long en su lugar, donde cada valor debe ser el rowId de uno de los elementos insertados. A fin de obtener más información sobre los valores rowId que se muestran, consulta la documentación de referencia de la anotación @Insert y la documentación de SQLite para tablas de rowid.

4.3 Actualizar

La anotación @Update te permite definir métodos que actualizan filas específicas en una tabla de base de datos. Al igual que los métodos @Insert, los métodos @Update aceptan instancias de entidades de datos como parámetros. El siguiente código muestra un ejemplo de un método @Update que intenta actualizar uno o más objetos User en la base de datos:

```
@Dao
interface UserDao {
    @Update
    fun updateUsers(vararg users: UserEntity)
}
```

Room usa la clave primaria para hacer coincidir las instancias de entidades pasadas con las filas de la base de datos. Si no hay una fila con la misma clave primaria, Room no realiza cambios.

De manera opcional, un método @Update puede mostrar un valor int que indica la cantidad de filas que se actualizaron de forma correcta.

4.4 Borrar

La anotación @Delete te permite definir métodos que borran filas específicas de una tabla de base de datos. Al igual que los métodos @Insert, los métodos @Delete aceptan instancias de entidades de datos como parámetros. En el siguiente código, se muestra un ejemplo de un método @Delete que intenta borrar uno o más objetos User de la base de datos:

```
@Dao
interface UserDao {
    @Delete
```

```
    fun deleteUsers(vararg users: UserEntity)
}
```

Room usa la clave primaria para hacer coincidir las instancias de entidades pasadas con las filas de la base de datos. Si no hay una fila con la misma clave primaria, Room no realiza cambios.

De manera opcional, un método @Delete puede mostrar un valor int que indique la cantidad de filas que se borraron de forma correcta.

4.5 Métodos de búsqueda

La anotación @Query te permite escribir instrucciones de SQL y exponerlas como métodos DAO. Usa estos métodos de búsqueda para consultar datos desde la base de datos de tu app o cuando necesites realizar inserciones, actualizaciones y eliminaciones más complejas.

Room valida las consultas en SQL en el tiempo de compilación. Esto significa que, si hay un problema con tu búsqueda, se produce un error de compilación en lugar de un fallo en tiempo de ejecución.

Con el siguiente código, se define un método que usa una búsqueda SELECT simple para mostrar todos los objetos User de la base de datos:

```
@Query("SELECT * FROM UserEntity")
fun loadAllUsers(): Array<UserEntity>
...
```

Tienes más ejemplos de búsquedas en el siguiente enlace.

5. BASE DE DATOS

Con el siguiente código, se define una clase UserDatabase para contener la base de datos. UserDatabase define la configuración de la base de datos y sirve como el punto de acceso principal de la app a los datos persistentes. La clase de la base de datos debe cumplir con las siguientes condiciones:

- La clase debe tener una anotación @Database que incluya un array entities que enumere todas las entidades de datos asociados con la base de datos.
- Debe ser una clase abstracta que extienda RoomDatabase.
- Para cada clase DAO que se asoció con la base de datos, esta base de datos debe definir un método abstracto que tenga cero argumentos y muestre una instancia de la clase DAO.

```
@Database(entities = [UserEntity::class], version = 1)
abstract class UserDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Es el punto de entrada principal para comunicar el resto de la app con el esquema relacional de datos. Esta clase nos oculta la implementación de SQLiteOpenHelper para facilitarnos el acceso a la base de datos.

6. USO

Después de definir la entidad de datos, el DAO y el objeto de base de datos. Vamos a crear una aplicación auxiliar que nos va a ayudar en el acceso a esta base de datos.

UserApplication

Se llamará UserApplication, y heredará de Application(). En ella crearemos el patrón singletton con el objetivo de acceder a la Base de Datos desde cualquier punto de la aplicación. Además sobreescribimos el onCreate e inicializamos en él la instancia del database.

```
class UserApplication : Application() {

```

```
    companion object{
        lateinit var database: UserDatabase
    }

    override fun onCreate() {
        super.onCreate()
        database = Room.databaseBuilder(this,
            UserDatabase::class.java,
            "UserDatabase")
            .build()
    }
}
```

Ahora ya tenemos la aplicación configurada y nos faltará inicializarla. Eso lo haremos en el AndroiManifest, añadiendo el atributo name.

```
<application
    android:name=".database.UserApplication"
...
```

Tras ello ya podemos trabajar con las operaciones que hemos preparado en el DAO y lo haremos de la siguiente forma:

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Insertar usuarios, lo haremos en un hilo secundario
        Thread{
            val userEntityLists : List<UserEntity> = listOf(
                UserEntity(1, "John", "Doe"),
                UserEntity(2, "Jane", "Smith"),
                UserEntity(3, "Alice", "Johnson")
            )

            UserApplication.database.UserDao().insertAll(userEntityLists)
```

```
        }.start()

        // Obtener la lista de usuarios en otro hilo

        Thread{
            val userEntities: List<UserEntity> =
UserApplication.database.UserDao().getAll()
            // Hacer algo con la lista de usuarios (por ejemplo, imprimir
en el Log)
            for (user in userEntities) {
                println("User: ${user.firstName} ${user.lastName}")
            }
        }.start()
    }
}
```

Fíjate que las operaciones realizadas con la BD se ejecutan en un hilo secundario, diferente al hilo principal de la aplicación. Esto es así porque nos obliga Android y para evitar que nuestra app se congele por un instante mientras se procesa todo lo relacionado con el acceso a la BD. Es recomendable que se haga en un hilo secundario que se ejecute en background.