Programación multimedia y dispositivos móviles

9

Room Persistence Library

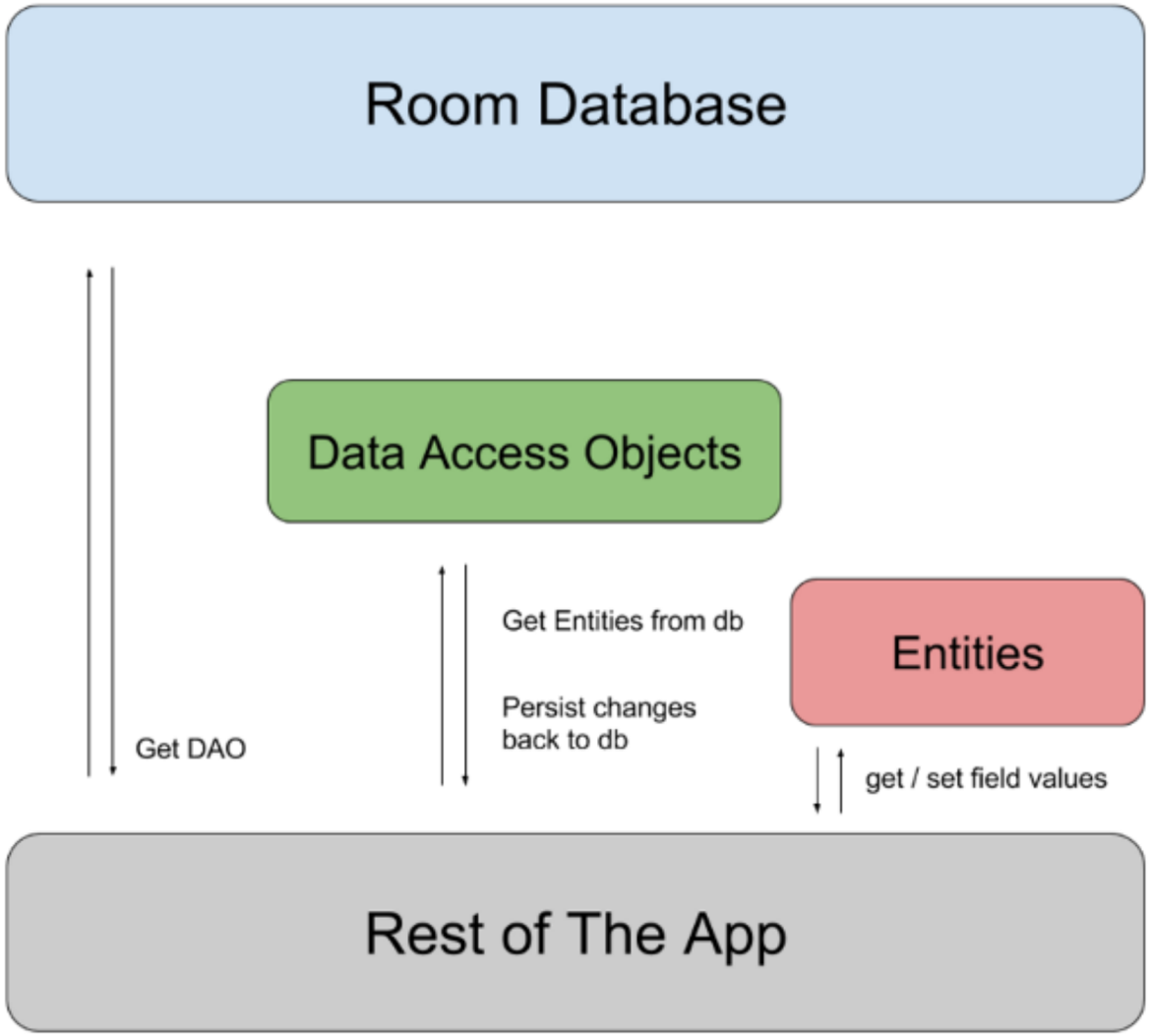
IES Nervión
Miguel A. Casado Alías

Introducción

- Liberado en noviembre de 2017
- Es uno de los componentes de la colección de bibliotecas (libraries) destinadas a ayudar a implementar la arquitectura de aplicación recomendada por Android
- Proporciona una capa de abstracción sobre SQLite
- Es una bibliteca de tipo ORM (Object Relational Mapping)
- Android recomienda usar Room en vez de trabajar directamente sobre SQLite, como se hacía anteriormente
- Debemos añadir dependencias en buid.gradle de la app:

```
implementation "android.arch.persistence.room:runtime:1.0.0"
annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
```

Diagrama de la arquitectura de ROOM



Entidades

- Representan una tabla en la BD
- Deben ser clases con la anotación @Entity
- No se permiten referencias entre objetos de tipo entidad
- Las propiedades de los objetos de tipo entidad deben ser accesibles desde el exterior, bien mediante getters y setters (recomendado), o bien cambiando su visibilidad (public por ejemplo)
- Pueden tener un constructor vacío (o no tener constructor) si el correspondiente DAO puede acceder a los atributos
- Si el correspondiente DAO no puede acceder a los atributos, entonces debe haber uno o varios constructores con parámetros (con todos los atributos del objeto y/o con varios)

Entidades: Clave primaria

- Hay que definir una clave primaria para cada entidad, incluso aunque sólo tenga una propiedad
- Si queremos que la clave primaria sea autogenerada: @PrimaryKey(autoGenerate=true)
- Si queremos definir una clave primaria compuesta, usaremos la propiedad primaryKeys de @Entity

```
@Entity
class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

- @Ignore hace que no se genere columna para ese atributo

```
@Entity(primaryKeys = {"firstName", "lastName"})
class User {
```

Entidades: Nombres de tablas y columnas

- Por defecto Room usa el nombre de la clase como nombre para la tabla (SQLite no distingue entre mayúsculas y minúsculas a este respecto)
- Para asignar otro nombre a la tabla, usaremos la propiedad tableName de @Entity
- Si no queremos usar el nombre del atributo como nombre para la columna, podemos utilizar @ColumnInfo

```
@Entity(tableName = "users")
class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

Entidades: Índices

- Podemos indexar algunas columnas con la propiedad “indices” de @Entity en conjunción con @Index

```
@Entity(indices = {@Index("name"),
        @Index(value = {"last_name", "address"})})
class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String address;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

Entidadades: Unicidad

- Para hacer que el valor de una columna o de un conjunto de columnas sea único en una tabla, usaremos la propiedad “unique” de @Index:

```
@Entity(indices = {@Index(value = {"first_name", "last_name"},
        unique = true)})
class User {
    @PrimaryKey
    public int id;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

Entidades: Claves ajenas

- Se definen con la propiedad foreignKeys de @Entity, en conjunción con @ForeignKey
- Las propiedades onDelete y onUpdate de @ForeignKey pueden tomar los valores CASCADE, NO_ACTION, RESTRICT, SET_DEFAULT o SET_NULL

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,
                                  parentColumns = "id",
                                  childColumns = "user_id"))
class Book {
    @PrimaryKey
    public int bookId;

    public String title;

    @ColumnInfo(name = "user_id")
    public int userId;
}
```

Entidades: Objetos anidados

- La anotación @Embedded permite anidar un objeto en una entidad
- Así pues, los atributos del objeto anidado también serán columnas de la tabla que representa la entidad que lo contiene

```
class Address {
    public String street;
    public String state;
    public String city;

    @ColumnInfo(name = "post_code")
    public int postCode;
}

@Entity
class User {
    @PrimaryKey
    public int id;

    public String firstName;

    @Embedded
    public Address address;
}
```

DAOs: Data Access Objects

- Contienen los métodos para acceder a la BD: consultas, inserciones de tuplas, borrado de tuplas, etc...
- Un DAO puede ser una clase abstracta o una interfaz
- Si es una case abstracta puede tener un constructor que reciba como parámetro un objeto RoomDatabase
- Room creará la implementación de cada DAO en tiempo de compilación
- Room no permite por defecto acceder a la BD desde el hilo principal de la aplicación. Esto se puede deshabilitar mediante el método allowMainThreadQueries() (SÓLO PARA TESTING, PUES HACE QUE LA UI PIERDA FLUIDEZ)
- Las consultas que devuelven LiveData o Flowable se hacen en segundo plano de por sí

DAOs: Inserción

- Anotamos el método con @Insert. Podemos indicar qué hacer en caso de conflicto con “onConflict”
- Puede devolver un long que reresenta el Id de la fila insertada. O un array de long o un List<Long> si se han insertado varios elementos

```
@Dao
public interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public void insertUsers(User... users);

    @Insert
    public void insertBothUsers(User user1, User user2);

    @Insert
    public void insertUsersAndFriends(User user, List<User> friends);
}
```

DAOs: Actualización

- Usaremos @Update. Le pasaremos una o varias entidades al método, y Room actualizará las tuplas en BD que coincidan con las claves primarias de los objetos pasados por parámetro

```
@Dao
public interface MyDao {
    @Update
    public void updateUsers(User... users);
}
```

DAOs: Borrado

- Usaremos @Delete. Le pasaremos una o varias entidades al método, y Room borrará las tuplas en BD que coincidan con las claves primarias de los objetos pasados por parámetro

```
@Dao
public interface MyDao {
    @Delete
    public void deleteUsers(User... users);
}
```

DAOs: Consultas

- Se usa la anotación @Query. Cada método anotado así se verifica en tiempo de compilación. Si la consulta es errónea dará un error en tiempo de compilación
- Room verifica que los nombres de las columnas devueltas por la consulta coinciden con los nombres de los atributos del objeto devuelto por el método
  - Si solo coinciden algunos nombres => Warning
  - Si no coincide ningún nombre => Error

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user")
    public User[] loadAllUsers();
}
```

DAOs: Consultas con parámetros

- Podemos pasar parámetros a los métodos para usarlos en las consultas

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
    public User[] loadAllUsersBetweenAges(int minAge, int maxAge);

    @Query("SELECT * FROM user WHERE first_name LIKE :search "
           + "OR last_name LIKE :search")
    public List<User> findUserWithName(String search);
}
```

DAOs: Colecciones como parámetros

```
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    public List<NameTuple> loadUsersFromRegions(List<String> regions);
}
```

DAOs: Devolver un subconjunto de columnas

- Podemos hacerlo creando una clase que tenga como atributos los nombres de las columnas que queremos que devuelva la consulta. Los atributos tienen que ser accesibles por el DAO (public, getters & setters, etc...)

```
public class NameTuple {
    @ColumnInfo(name="first_name")
    public String firstName;

    @ColumnInfo(name="last_name")
    public String lastName;
}
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user")
    public List<NameTuple> loadFullName();
}
```

DAOs: Consultas que devuelven observables

- Podemos hacer que los métodos devuelvan objetos observables, y así hacer uso del patrón observador
- Esto es ideal cuando queremos actualizar la UI debido a que hay cambios en la BD
- Room actualizará el dato observable cuando se produzca algún cambio, así que podremos observar ese cambio en el dato desde una Actividad o Fragment y actualizar la UI
- Room soporta los siguientes tipos de observables:
  - LiveData: específico de Android
  - Flowable: propio de RxJava2
  - Publisher: propio de RxJava2
- Para usar RxJava2 hay que incluir dependencia en gradle: android.arch.persistence.room:rxjava2

DAOs: LiveData

```
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    public LiveData<List<User>> loadUsersFromRegionsSync(List<String> regions);
}
```

DAOs: RxJava2

```
@Dao
public interface MyDao {
    @Query("SELECT * from user where id = :id LIMIT 1")
    public Flowable<User> loadUserById(int id);
}
```

DAOs: Consulta de varias tablas

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM book "
            + "INNER JOIN loan ON loan.book_id = book.id "
            + "INNER JOIN user ON user.id = loan.user_id "
            + "WHERE user.name LIKE :userName")
   public List<Book> findBooksBorrowedByNameSync(String userName);
}
```

Room Database

- Es el punto de entrada a la BD
- Debe ser anotada con @Database e incluir la lista de todas las entidades asociadas y del número de versión
- Debe ser una clase abstracta que extienda de RoomDatabase
- Debe tener uno o más métodos abstractos sin parámetros que devuelvan cada uno de los DAOs
- Normalmente solo se necesita una instancia de la BD Room para toda la aplicación, así que es conveniente usar el patrón *singleton*

Ejemplo de Room Database

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {

    public abstract UserDao userDao();

    private static AppDatabase INSTANCE;

    static AppDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (AppDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                            AppDatabase.class, name: "user_database.db")
                            .build();
                }
            }
        }
        return INSTANCE;
    }

}
```

- Para usarla, obtenemos la BD, luego un DAO a partir de ella, y ya podemos llamar a los métodos:

  AppDatabase.getDatabase(context).userDao().getUser(id)

Entidades: Relaciones

- La anotación @Relation nos permite obtener todas las entidades relacionadas con otra que queramos obtener de la BD. Es útil en las relaciones 1:N
- El atributo marcado con @Relation debe ser List o Set, y debe tener un setter o ser accesible para el DAO que lo use
- No se puede usar @Relation en atributos de una entidad
- Propiedades de @Relation:
  - parentColumn: atributo identificador en objeto padre
  - entityColumn: atributo de objeto relacionado (hijo) que se corresponde con parentColumn
  - entity: objeto relacionado (subordinado al objeto padre)
  - projection: obtener solo algunos atributos del objeto relacionado
- VER EJEMPLO (stackoverflow.com)

@Transaction

- Marca un método de un DAO como **transacción de BD**[1]
- Cuando se usa en un método no abstracto de un DAO, toda su implementación será una transacción. La transacción se considerará correcta salvo que se lance una excepción en el cuerpo del método
- Es conveniente usarlo en métodos @Query cuando:
  - El resultado de la consulta es extremadamente grande
  - El resultado implica un objeto con atributos @Relation
- No es necesario usarlo en métodos marcados como @Insert, @Update o @Delete, porque ya se ejecutan en una transacción de por sí

1. Conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica. Más información.

Ejemplo de @Transaction en método no abstracto

```
@Dao
public abstract class ProductDao {
    @Insert
     public abstract void insert(Product product);
    @Delete
     public abstract void delete(Product product);
    @Transaction
     public void insertAndDeleteInTransaction(Product newProduct, Product oldProduct) {
         // Anything inside this method runs in a single transaction.
         insert(newProduct);
         delete(oldProduct);
     }
}
```

Ejemplo de @Transaction en método abstracto

```
class ProductWithReviews extends Product {
    @Relation(parentColumn = "id", entityColumn = "productId", entity = Review.class)
    public List<Review> reviews;
}
@Dao
public interface ProductDao {
    @Transaction @Query("SELECT * from products")
    public List<ProductWithReviews> loadAll();
}
```

Conversores de tipos

- A veces nuestras entidades tienen tipos de datos propios o tipos de datos que Room no sabe cómo guardar en BD
- Si hacemos conversores de tipos, Room podrá trabajar con esos tipos de datos

```
public class Converters {
    @TypeConverter
    public static Date fromTimestamp(Long value) {
        return value == null ? null : new Date(value);
    }

    @TypeConverter
    public static Long dateToTimestamp(Date date) {
        return date == null ? null : date.getTime();
    }
}
```

```
@Database(entities = {User.class}, version = 1)
@TypeConverters({Converters.class})
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

Callback: onOpen y onCreate

- RoomDatabase.Callback es una clase abstracta que nos permite implementar dos métodos:
  - onOpen: es llamado cada vez que se abre la BD
  - onCreate: es llamado cuando se crea la BD por primera vez. Se llama después de que las tablas han sido creadas
- Podemos usar onCreate para precargar datos en la BD, o para crear tablas auxiliares distintas de las que se crean para cada clase marcada como @Entity, por ejemplo.
- Para añadir nuestra clase de Callback usamos addCallback

```
INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
        AppDatabase.class, name: "user_database.db")
        .addCallback(dbCallback)
        .build();
```

Ejemplo de Callback

```
RoomDatabase.Callback dbCallback = new RoomDatabase.Callback(){
    public void onCreate (SupportSQLiteDatabase db){
        super.onCreate(db);
        //Create tables with execSQL
        String SQL_CREATE_TABLE = "CREATE TABLE  log" +
                "(" +
                "_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                "info TEXT, " +
                "date  TEXT)";
        db.execSQL(SQL_CREATE_TABLE);

        //Insert data with SupportSQLiteDatabase.insert()
        ContentValues cv = new ContentValues();
        cv.put("id", "1");
        cv.put("name", "Michael Jackson");
        db.insert( table: "artists", OnConflictStrategy.IGNORE, cv);

        cv.put("id", "2");
        cv.put("name", "Taylor Swift");
        db.insert( table: "artists", OnConflictStrategy.IGNORE, cv);
        //Insert using DAO methods in another thread
        Executors.newSingleThreadScheduledExecutor().execute(() -> {
                getDatabase(context).userDao().insertOrReplaceUsers(/* Users */);
        });
    }
    public void onOpen (SupportSQLiteDatabase db){
        super.onOpen(db);
        ContentValues contentValues = new ContentValues();
        contentValues.put("info", "db open");
        contentValues.put("date", LocalDateTime.now().toString());

        db.insert( table: "log", OnConflictStrategy.IGNORE, contentValues);
    }
};
```

Migraciones

- Si cambiamos el esquema (añadir tablas, eliminar tablas, modificar tablas…) pero no incrementamos el número de versión de la BD, la aplicación fallará
- Si aumentamos el número de versión de la BD, pero no proporcionamos alguna migración, la aplicación o bien fallará o bien las tablas se eliminarán junto a todos sus datos (llamando a *fallbackToDestructiveMigration()* en el builder) para ser creadas nuevamente
- Una clase *Migration* especificará las acciones a llevar a cabo para actualizar de una versión de la BD a otra
- Dichas clases *Migration* deben ser añadidas llamando al método *addMigrations()* del builder
- Se pueden añadir varias rutas de migración para una misma versión, y el sistema elegirá la ruta óptima (p.ej: de 1 a 2, de 2 a 3, y de 1 a 3… el sistema optará por ejecutar la de 1 a 3)

Ejemplo de migraciones

```
Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name")
        .addMigrations(MIGRATION_1_2, MIGRATION_2_3).build();

static final Migration MIGRATION_1_2 = new Migration(1, 2) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, "
                + "`name` TEXT, PRIMARY KEY(`id`))");
    }
};

static final Migration MIGRATION_2_3 = new Migration(2, 3) {
    @Override
    public void migrate(SupportSQLiteDatabase database) {
        database.execSQL("ALTER TABLE Book "
                + " ADD COLUMN pub_year INTEGER");
    }
};
```

Testing

- Probando la BD
- Probando las migraciones