មជ្ឈមណ្ឌលកូរ៉េ សហ្វវែរ អេច អ ឌី

Korea Software HRD Center

Android Development

Room Persistence

អ្នកប្រឹក្សាយោបល់:បណ្ឌិត គីម ថេខ្យុង

www.kshrd.com.kh

Content

1. Overview

- Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.
- In order to user Room in you app, you need to add dependency to your app’s build.gradle file

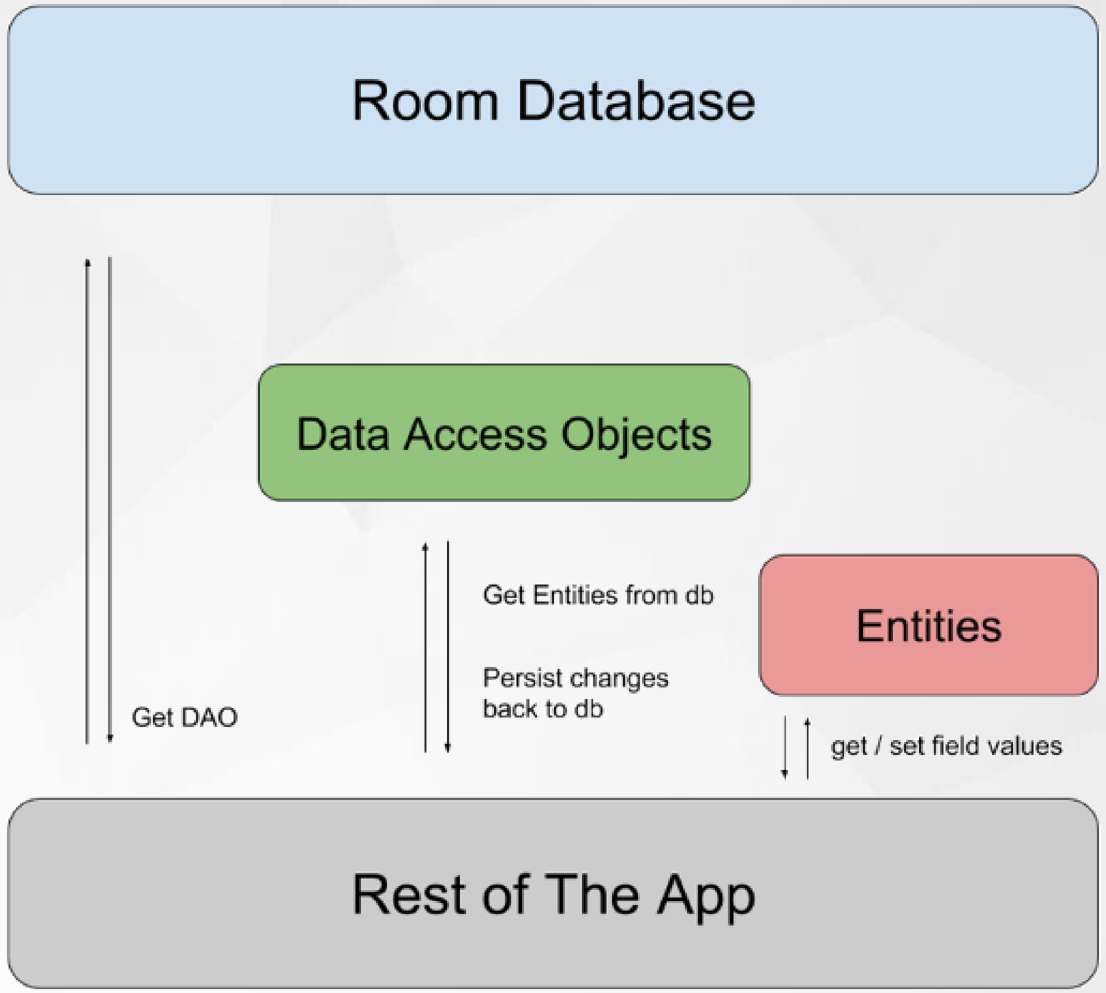
```
implementation "android.arch.persistence.room:runtime:1.0.0"
annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
```

- There three major components in Room:

1. **Database:**
2. **Entity:** Represents a table within the database
3. **DAO:** Contains the method used for accessing the database

1. Overview(con…)

- These components, along with their relationships with the rest of the app



2. Room components

1. **Database**

- Contains the database holder and serves as the main access point for the underlying connection to your app’s persisted, relational data.
- Database class should satisfied the follow condition:
  - Be an abstract class that extends RoomDatabase
  - Be annotate with @Database
  - Include the list of entities associated with the database within the annotion
  - Contain an abstract method that has 0 argument and return the class that is annotated with @Dao
- At runtime, you can get an instance of Database by calling **`Room.databaseBuilder()`** or **`Room.inMemoryDatabaseBuilder()`**

2. Room components(con…)

- **Example**

```
@Database(version = 1, entities = {User.class, Book.class})
abstract class AppDatabase extends RoomDatabase {
     // BookDao is a class annotated with @Dao.
     abstract public BookDao bookDao();
     // UserDao is a class annotated with @Dao.
     abstract public UserDao userDao();
     // UserBookDao is a class annotated with @Dao.
     abstract public UserBookDao userBookDao();
 }
```

2. Room components(con…)

2. **Entity:**

- represent the table within the data
- Define a set of related fields as entities
- For each entity, table is created within the associated Database object to hold the data
- By default, Room creates a column for each field that's defined in the entity.
- You must be reference the entity class through the **`entities`** array in the Database class

2. Room components(con…)

2. Entity:

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

    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

2. Room components(con…)

2. **Entity:**
   - List of annotations in entity class
     - @Entity: annotated the class that you want to create entity
     - @PrimaryKey: annotate the field of entity class to define at least 1 field as a primarykey
     - @Ignore: annotate the field of entity class that you don’t want to persist
     - @ColumnInfo: annotate the field of entity class that you want a column to have a different name
     - @Index: annotate indices and uniqueness.
       - to add indices to an entity, you have to include the **indices** property with the @Entity annotation, listing the name of columns that you want to include index
       - Also use @Index to enforce the uniqueness property by setting the **unique** property of an @Index annotation to true
     - @Foreignkey: define relationship between object. You have to include foreignkeys property with @Entity annotation
     - @Embedded: annotate the **reference field (public Address address)** that you want to create nested objects

2. Room components(con…)

@Entity example

```
@Entity(tableName = "users")
class User {
    ...
}
```

2. Room components(con…)

@PrimaryKey example

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

    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

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

    @Ignore
    Bitmap picture;
}
```

2. Room components(con…)

@ColumnInfo example

```
@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;
}
```

2. Room components(con…)

@Index example

```
@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;
}
```

```
@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;
}
```

2. Room components(con…)

@ForeignKey example

```
@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;
}
```

2. Room components(con…)

@Embedded example

```
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;
}
```

2. Room components(con…)

3. Dao(Data access object)

- This layer is use to access your app’ data
- the set of Dao object form the main component of room
- Each DAO includes method that offer abstract access to your app’s database
- A Dao can be either an interface or an abstract class.
- Room create each Dao implementation at compile time.
- By accessing a database using a DAO class instead if query builders or direct queries
- It allow you to easily mock database access as you test you app

2. Room components(con…)

Note: Room doesn't support database access on the main thread unless you have called **`allowMainThreadQueries()`** on the builder because it might lock the UI for a long period of time.

- **Dao** class contain all methods that allow you to query, insert, delete, and update data
- **Define methods for convenience**
  - Insert: Create a DAO method and annotate it with `@Insert`

```
@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);
}
```

2. Room components(con…)

- **Define methods for convenience**
  - Update: the update convenience method modifies a set of entities. You just create a method and annotate with `@Update`

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

  - Delete: Annotate the method with @Delete

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

2. Room components(con…)

- **Query in formation**
  - Simple queries

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

  - Passing parameter into query

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age > :minAge")
    public User[] loadAllUsersOlderThan(int minAge);
}
```

2. Room components(con…)

- **Query in formation**
  - Returning subsete of columns

```
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();
}
```

  - Querying multiple tables

```
@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> findBooksBorrowedByName(String userName);
}
```

3. Database Migration

- When you change or add new entities you must increase database version in @Database annotation
- There are two steps to migrate your database
  - You must increase database version in @Database annotation
  - You must provide Migration to update you database objects. In this way it will keep all data in your database
  - Or call **fallbackToDestructiveMigration()** method in the builder in case Room will re-create all of the tables

- Use call **fallbackToDestructiveMigration()** method:

```
INSTANCE= Room.databaseBuilder(context,AppRoomDatabase.class,DATABASE_NAME)
        .allowMainThreadQueries()
        .fallbackToDestructiveMigration()
        //.addMigrations(MIGRATION_1_2)
        .build();
```

3. Database Migration (continue)

- **Implement Migration to update you database objects**
  - Create a Migration object

```
public static final Migration MIGRATION_1_2= new Migration(CURRENT_DATABASE_VERSION,
                                                                 NEXT_DATABASE_VERSION) {
    @Override
    public void migrate(@NonNull SupportSQLiteDatabase database) {
        //Log.e("db-current-version-> ",database.getVersion()+"");
        database.execSQL("alter table Category add sub_category TEXT;");
    }
};
```

3. Database Migration (continue)

- Call addMigration(mMigration)

```
INSTANCE= Room.databaseBuilder(context,AppRoomDatabase.class,DATABASE_NAME)
        .allowMainThreadQueries()
        .addMigrations(MIGRATION_1_2)
        .build();
```