03-5. Room

Android Architecture Component

Room

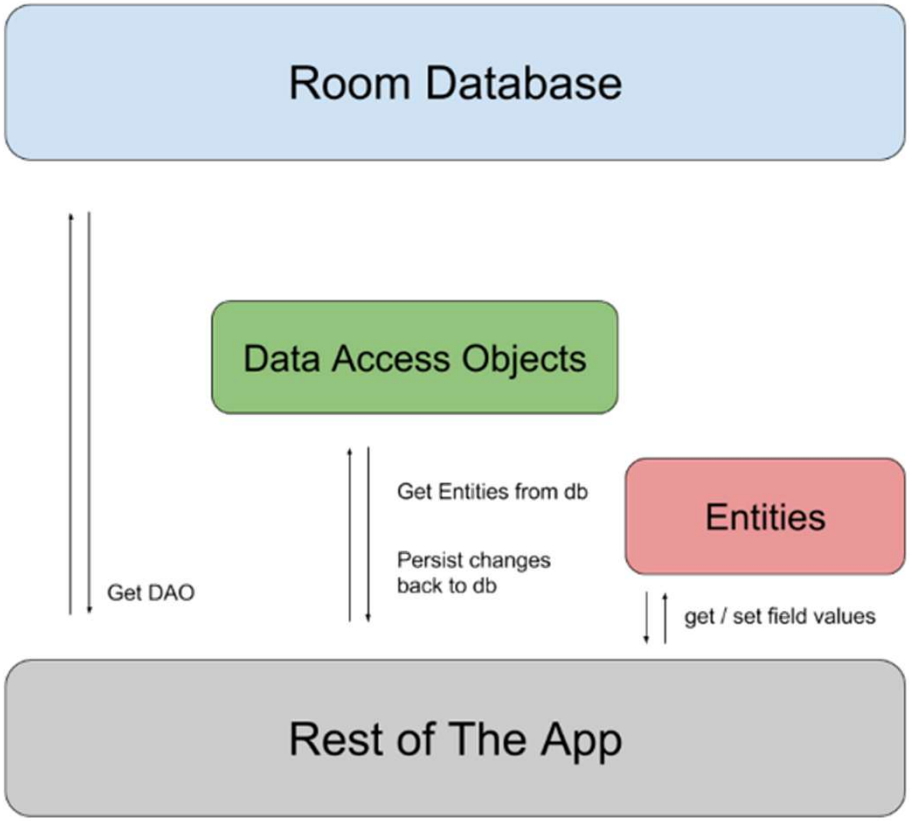
- SQLite 추상화 라이브러리
- 안드로이드 앱의 데이터베이스 이용을 Room 으로 권고

```
//annotation 처리
id ‘com.google.devtolls.ksp”


……

implementation 'androidx.room:room-ktx:2.3.0'
kapt("androidx.room:room-compiler:2.3.0")
```

Room


Room Database
Data Access Objects
Get DAO
Get Entities from db
Persist changes back to db
Entities
get / set field values
Rest of The App

Entity

- 데이터 구조를 표현하기 위한 클래스
- @Entiry 어노테이션으로 정의
- 클래스 내에 @PrimaryKey, @ColumnInfo 등의 어노테이션으로 변수 선언

```
@Entity
class User(
    @field:PrimaryKey
    var uid: Int,

    @field:ColumnInfo(name = "first_name")
    var firstName: String?,

    @field:ColumnInfo(name = "last_name")
    var lastName: String?
)
```

DAO

- DBMS를 위해 호출되는 함수를 선언하는 인터페이스
- @DAO 어노테이션으로 선언
- @Query, @Insert, @Delete 등의 어노테이션으로 함수 선언

```
@Dao
interface UserDAO {
    @Query("SELECT * FROM User")
    fun getAll(): List<User>

    @Insert
    fun insertUser(uses: User)
}
```

Database

- 데이터베이스 이용을 위한 accessor(함수)를 제공
- @Database
- 추상클래스로 작성어노테이션으로 선언
- Entity 를 어노테이션 매개변수로 지정
- 추상 함수를 가져야 하며 이 함수가 데이터베이스를 이용하기 위해 호출되는 함수
- 추상 함수는 매개변수가 없어야 하며 return 은 DAO 클래스여야 함

```
@Database(entities = arrayOf(User::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDAO
}
```

using Room

- Activity에서 DAO 클래스의 이용은 background 스레드로 실행
- RoomDatabase의 객체는 가급적 singleton 으로 획득할 것을 권장

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "test6"
).build()
```

```
dao = db.userDao()

var user = User(0, firstName = "gildong", lastName = "hong")
thread {
    dao.insertUser(user)
}
```

Entity

- Entity Class
  - Entity 클래스는 @Entity 로 선언되는 클래스
  - Entity 클래스에 데이터를 저장하기 위한 table 이 자동으로 만들어 짐
  - table 이름은 클래스명이 되며, 대소문자는 무시됨
  - @PrimaryKey 어노테이션으로 PK property 를 지정할 수 있음
  - 생성자는 자유롭게 추가 가능

Entity

•Entity property

- Entity 클래스내의 property 에 해당되는 column 이 만들어 짐
- property 명에 해당되는 column 과 매핑이 되며 대소문자를 구분
- property 중 데이터베이스와 상관없이 소스적으로만 이용하고자 하는 field 는 @Ignore 어노테이션을 추가하면 됨
- 데이터베이스와 매핑되는 property 는 생성자 property 혹은 class body 에 정의 가능
- 생성자 property 에는 @Ignore 추가할 수 없음
- 자바로 변형 시 field 는 public 으로 선언되거나 private 으로 선언되면 getter/setter 함수가 추가되어 있어야 함

```
@Entity
class User2(
    @PrimaryKey(autoGenerate = true) var id: Int=0,
    var lastName: String,
    var firstName: String,
    var address: String ="seoul"
)
```

Entity

•@PrimaryKey

- Entity 클래스에는 최소 하나 이상의 primary key field 가 선언되어야 함
- autoGenerate=true 속성으로 값이 자동 증가되게 만들 수 있음
- autoGenerate=true 가 선언되었다고 하더라도 실제 데이터가 지정이 되면 그 값으로 DB INSERT 가 됨
- 만약 id 값이 0이 설정되면 값이 자동 증가
- 자동으로 insert 된 id 값은 max id 값에 1이 더해진 값

```
@PrimaryKey(autoGenerate = true) var id: Int=0,
```

- 여러 컬럼을 묶어서 primary key 로 선언할 수 있음
- primaryKeys 에 나열된 property 는 non-nullable 로 선언되어 있어야 함

```
@Entity(primaryKeys = arrayOf("lastName", "firstName"))
```

Entity

•@Entity
- 테이블 명은 기본으로 Entity 클래스명을 따르지만 원한다면 바꿀 수 있음
- @Entity 어노테이션에 tableName 속성으로 table 명을 지정

•@ColumnInfo
- @ColumnInfo(name="first_name") 를 이용하여 field 명과 다른 column 명 지정 가능

•index
- indices 속성으로 @Entity 에 index 정보 설정

```
@Entity(
    tableName = "tb_user",
    indices = arrayOf(
        Index(value = ["lastName"]),
        Index(value = ["firstName", "lastName"], unique = true)
    )
)
```

foreignKey

•onDelete 속성 혹은 onUpdate을 이용해 parent table 의 데이터가 수정,삭제 되었을 때 어떻게 되어야 하는지를 명시

- CASCADE : 모든 child row를 삭제, 수정
- NO_ACTION : child 쪽에서는 아무 일도 발생하지 않음
- RESTRICT : 만약 child에 parent key 로 매핑 된 데이터가 있다면 parent 부분의 삭제 및 수정을 금지
- SET_DEFAULT : child key column 을 default 값으로 셋팅
- SET_NULL : child key column 을 null 로 셋팅

```
@Entity(
    foreignKeys = arrayOf(
        ForeignKey(
            entity = User::class,
            parentColumns = arrayOf("id"),
            childColumns = arrayOf("user_id"),
            onDelete = ForeignKey.CASCADE,
        )
    )
)
```

DAO

- 데이터를 저장하거나 획득하기 위한 작업 클래스
- DAO 클래스는 interface 혹은 추상 클래스로 작성하며 추상함수에 정의
- 어노테이션 정보를 보고 DB 작업을 하는 클래스를 generate
- DAO 의 함수 호출은 background thread 에서 실행
- Main Thread 에서 DAO 를 실행하고자 한다면 Database Builder에서 allowMainThreadQueries() 를 호출
- DAO 의 반환 값은 LiveData 혹은 RxJava 의 Flowable, Coroutine 의 Flow 가 될 수 있음

@Insert

- insert 함수의 매개변수는 insert되는 entity 객체 하나 일수도 있고, 배열이거나 List 일 수도 있음
- @insert 함수는 insert 되는 row 의 pk 값을 받을 수도 있음
- 하나만 insert 시킨다면 insert 된 row 의 pk 가 long 타입으로 리턴
- 배열이나 List 인 경우에는 long[] 혹은 List<Long> 타입으로 리턴

```
@Insert
fun insertUser(uses: User)

@Insert
fun insertUsers(vararg users: User): List<Long>

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

@Insert
fun insertUsersAndFriends(users: List<User>)
```

@Update, @Delete

- @Update
  - 매개변수로 대입된 객체의 데이터를 모두 업데이트
  - 리턴 값은 업데이트된 row count 값을 Int 로 받을 수 있음

```
@Update
fun updateUsers(vararg users: User)
```

- @Delete
  - 대입된 객체와 PK 를 기준으로 row 를 삭제
  - 리턴 값은 삭제된 row count 값을 Int 로 받을 수 있음

```
@Delete
fun deleteUsers(vararg users: User)
```

@Query

- @Query 은 모든 어노테이션의 기본
- select 만을 목적으로 하지 않고 read/write 가 가능함
- sql 문은 직접 작성이 가능함
- @Query 에 정의된 sql 문은 컴파일 단계에서 검증이 이루어 짐
- parameter 설정은 콜론(:) 기호를 이용함

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

@Query("UPDATE User SET firstName = :value WHERE id = :id")
fun updateName(value: String?, id: Int): Int
```

@Query

- Entity 클래스로 결과값을 받지 않고 select 된 일부의 데이터만 매핑되는 POJO 클래스 사용 가능

```
class NameVO {
    @ColumnInfo(name = "firstName")
    var fName: String? = null
    var lastName: String? = null
}
```

```
@Query("SELECT firstName, lastName FROM User")
fun loadFullName(): List<NameVO>
```

@Query

- DAO function 을 suspend 함수로 선언 가능

```
@Query("SELECT * FROM user WHERE id = :id")
suspend fun getUserSuspendFunction(id: Int): User
```

```
launch {
    dao.getUserSuspendFunction(1).apply {
        Log.d("kkang", "1, ${firstName}, ${lastName}")
    }
}
```

- @Query 의 결과로 LiveData 이용 가능

```
@Query("SELECT * FROM user WHERE id = :id")
fun getUserLiveData(id: Int): LiveData<User>
```

```
dao.getUserLiveData(1).observe(this){
    Log.d("kkang", "2, ${it.firstName}, ${it.lastName}")
}
```

@Query

- @Query 의 결과로 Flow 이용 가능

```
@Query("SELECT * FROM user")
fun getUserFlow(): Flow<User>
```

```
launch {
    dao.getUserFlow().collect {
        Log.d("kkang", "3, ${it.firstName}, ${it.lastName}" )
    }
}
```

- @Query 의 결과로 Cursor 이용 가능

```
@Query("SELECT * FROM user WHERE id = :id")
fun getUserCursor(id: Int): Cursor
```

```
val cursor = dao.getUserCursor(1)
if(cursor.moveToFirst()){
    Log.d("kkang", "4, ${cursor.getString(1)}, ${cursor.getString(2)}" )
}
```

Migration

- 데이터베이스에 스키마를 변경
- 스키마 변경은 Database 클래스의 버전 정보를 이용

```
val migration = object: Migration(1,2){
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE USER ADD COLUMN email TEXT DEFAULT null ")
    }
}

@Database(entities = arrayOf(User::class), version = 2)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDAO
}
```

```
val db = Room.databaseBuilder(
    applicationContext,
    AppDatabase::class.java, "test6"
)
    .addMigrations(migration)
    .build()
```

Converter

- 기초 타입 이외의 타입 매핑은 지원하지 않음

```
@Entity
class User(
    @PrimaryKey(autoGenerate = true)
    var id: Int=0,
    var lastName: String,
    var datas: List<String>
)
//Cannot figure out how to save this field into database. You can consider adding a type converter for it.
//private java.util.List<java.lang.String> datas;
```

Converter

- @TypeConverter 로 어떻게 데이터가 매핑 되어야 하는지를 명시해서 사용
- 함수명은 자유롭게 정의, 매개변수와 리턴 타입을 보고 어떤 Converter 를 사용할지 자동 결정

```
class Converters {
    @TypeConverter
    fun fromListToString(value: List<String>?): String {
        return Gson().toJson(value)
    }

    @TypeConverter
    fun fromStringToList(value: String): List<String> {
        return Gson().fromJson(value, Array<String>::class.java).toList()
    }
}
```

```
@Database(entities = arrayOf(User::class), version = 2)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDAO
}
```

Nested Object

- 하나의 Entity 클래스에서 다른 Entity 클래스를 참조하는 경우
- 두 Entity 클래스의 데이터를 하나의 데이블에 매핑하고 싶은 경우, @Embedded 어노테이션을 이용

```
data class Address(
    var street: String,
    var state: String,
    var city: String)

@Entity
data class User(
    @PrimaryKey(autoGenerate = true)
    var id: Int=0,
    var lastName: String,
    var firstName: String,
    @Embedded var address: Address,
)
```

Nested Object

•Nested Object 를 Entity 로 등록하려면 Converter 이용

```
@Entity
data class Address(
    @PrimaryKey(autoGenerate = true)
    var id: Int=0,
    ……….
)
@Entity
data class User(
    @PrimaryKey(autoGenerate = true)
    var id: Int=0,
    ……..
    var address: Address,
)
```

```
@TypeConverter
fun fromStringToAddress(value: String?): Address {
    val addressType: Type = object : TypeToken<Address>() {}.type
    return Gson().fromJson(value, addressType)
}
@TypeConverter
fun fromAddressToString(address: Address): String {
    val gson = Gson()
    return gson.toJson(address)
```

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare