Tonia Tkachuk

320 Followers About Follow

Unsplash — https://unsplash.com/photos/2JIvboGLeho

ANDROID DATABASE

Android app example using Room database and coroutines

Tonia Tkachuk Apr 7, 2018 · 7 min read ★

With **Android Architecture Components**, along with handling lifecycle events, realtime data updates in UI (ViewModel with LiveData) and pagination of loaded data (Paging), comes **Room** — small, yet powerful SQLite ORM. In this post I'm gonna demonstrate its core capabilities on an example Android application, written in Kotlin.

Remember those times implementing *SQLiteOpenHelper* and checking SQL queries in run-time? Good news is that you don't have to do it anymore! Room performs compile-time checks on your SQL queries and you don't have to write any SQLite code which is not in a direct relation with your data queries. Great, let's use it!

First of all, **Room** is a part of Architecture Components, which means it works really well with ViewModel, LiveData and Paging (but does not depend on those modules!). Also, RxJava, coroutines and Kotlin are perfectly fine too, as you'll see next.

In order to use Room in my project, I have below dependencies in my app's ***build.gradle*** file:

```
dependencies {
    def lifecycle_version = "2.2.0"
    def room_version = "2.2.5"

    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.3.0-alpha01'
    implementation 'com.google.android.material:material:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'

    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-extensions:$lifecycle_version"

    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
}
```

build.gradle hosted with ♡ by GitHub view raw

support - ***room-ktx*** dependencies. Check for the latest Room version here. The *lifecycle* ones are for LiveData and ViewModel. And the others are mostly for UI that we'll build.

In order for all of them to work, please add the required plugins at the top of the file, like this:

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

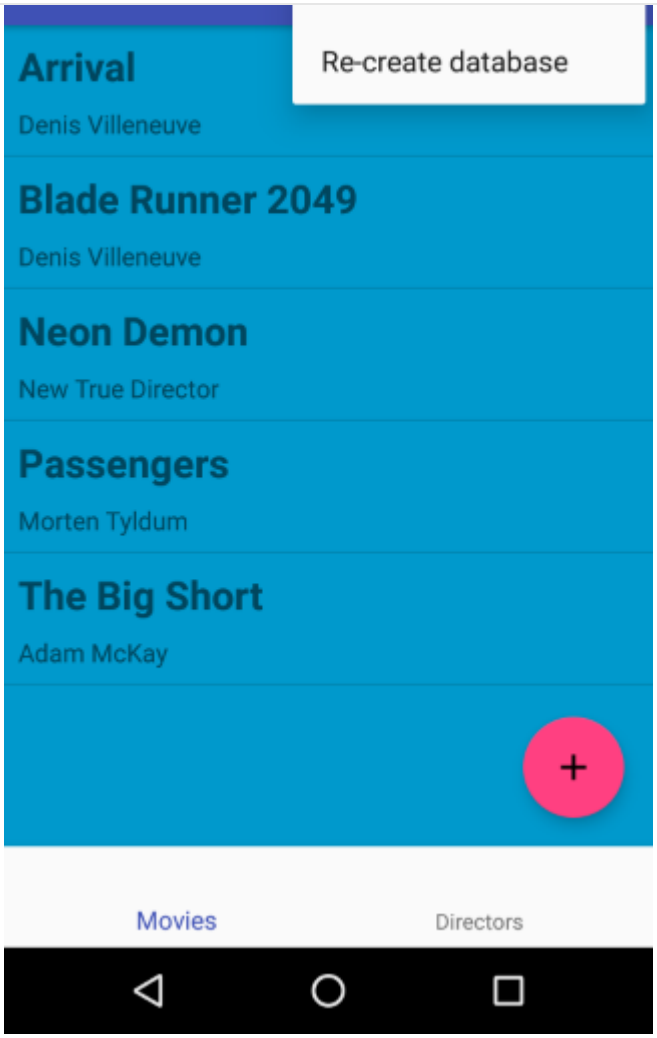
build.gradle hosted with ♡ by GitHub view raw

Additionally, provide a location of DB's schema in *defaultConfig* scope. This way you can always check how the generated scheme looks like, maybe decide to modify your DB tables based on the outcome. In my case, I chose *src* > *main* > *assets* > *schema* path, but this is just my preference, you can choose any other.

```
javaCompileOptions {
            annotationProcessorOptions {
                arguments = [
                        "room.schemaLocation": "$projectDir/src/main/assets/schemas".toString(),
                        "room.incremental": "true"
                ]
            }
        }
```

build.gradle hosted with ♡ by GitHub view raw

In my example application (jump to the source code if you wish) I have one Activity with *BottomNavigationView*, holding two tabs — Movies and Directors. There is a *RecyclerView* to show the list of each category, *FloatingActionButton* — to add a new movie/director, and two options in *ActionBar's* overflow menu — *Delete list data* and *Re-create database*. Also, when clicking on a list item, you can update its data.



main activity with the list of movies from DB

I use two tables — *Director* and *Movie*. Same director can be the author of many movies, so it's *one-to-many* relationship. I assume the full name of the director to be unique and mandatory. Movie's title doesn't have to be unique, but needs to be provided as well.

Director's class looks like below:

```
1  @Entity(
2      tableName = TABLE_NAME,
3      indices = [Index(value = [FULL_NAME], unique = true)])
4  data class Director(
5      @PrimaryKey(autoGenerate = true)
6      @ColumnInfo(name = "did") var id: Long = 0,
7      @ColumnInfo(name = FULL_NAME) var fullName: String,
8      @Ignore var age: Int = 0) {
```

```
12      companion object {
13          const val TABLE_NAME = "director"
14          const val FULL_NAME = "full_name"
15      }
16  }
```

Director.kt hosted with ♡ by GitHub view raw

Data structure classes such as *Director* should be annotated with *@Entity* to become a table in database. You can provide optional parameters such as *tableName* and *indices*. Adding an index on a column which will be used in search queries, speeds the search. I have checked this with *full_name* value and the search was indeed faster. Read more on indices here.

Note the *unique = true* for *full_name* index. Room does not have *@Unique* annotation, but you can enforce uniqueness on an index, so it's one way to do it.

Each table needs to have at least one primary key. In our case it's the *id*. *@ColumnInfo(name = "did")* renames *id* to *did* for a column's name. I don't want to bother with providing the ids myself, hence *autoGenerate = true* for Room to do that.

I also have an ignored field *age* with *@Ignore* annotation. This will exclude it from being added to database's schema.

Also, empty constructor with default values is there because of this error otherwise:

```
/movies-
room/app/build/tmp/kapt3/stubs/debug/com/lomza/moviesroom/db/Director
.java:10: error: Entities and POJOs must have a usable public
constructor. You can have an empty constructor or a constructor whose
parameters match the fields (by name and type).

Tried the following constructors but they failed to match:
Director(long,java.lang.String,int) -> [param:id -> matched field:id,
param:fullName -> matched field:fullName, param:age -> matched
field:unmatched]
```

by providing *parentColumns* and *childColumns* values (there is an alternative to foreignKeys and it's a @Relation annotation). You can provide two more params: *onUpdate* and *onDelete*. By default their values are 1 or *ForeignKey.NO_ACTION*. By writing *onDelete = ForeignKey.CASCADE* I basically tell Room to delete a movie if its director gets removed from the database. Other possible values are: *RESTRICT*, *SET_NULL* or *SET_DEFAULT*.

```
@Entity(
    tableName = TABLE_NAME,
    foreignKeys = [ForeignKey(
        entity = Director::class,
        parentColumns = ["did"],
        childColumns = [DIRECTOR_ID],
        onDelete = ForeignKey.CASCADE
    )],
    indices = [Index(TITLE), Index(DIRECTOR_ID)]
)
data class Movie(
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "mid") var id: Long = 0,
    @ColumnInfo(name = TITLE) var title: String,
    @ColumnInfo(name = DIRECTOR_ID) var directorId: Long) {

    companion object {
        const val TABLE_NAME = "movie"
        const val TITLE = "title"
        const val DIRECTOR_ID = "directorId"
    }
}
```

Movie.kt hosted with ♡ by GitHub view raw

That's it for the entity classes. Now let's define methods that we're gonna use to manipulate our data. Create a *DirectorDao* interface with *@Dao* annotation (it also can be an abstract class instead of interface). Room provides four different annotations: *@Insert, @Update, @Delete and @Query*. For insert and update you can provide *OnConflictStrategy* value, customising the behaviour in case of arisen conflict. By default the transaction is aborted, but in our case, we ignore it. As a parameter of *insert()*,

return *int*, getting an info of how many rows were affected. All of those functions will be suspend ones, as we'll use them inside of coroutine scope, in order to not block the UI thread.

In case of *DirectorDao*, I return long for *insert(Director director)*, as I will need an id of a newly created row for a movie object.

There are a few other queries that we need: *findDirectorById()*, *findDirectorByName()* and *getAllDirectors()*. *full_name* column is used for searching and ordering. It is a good candidate for an index.

Notice the return type of *getAllDirectors()*. I wrap the list of results in *LiveData*, as I would like the list to update automatically when the underlined data changes.

```
@Dao
interface DirectorDao {

    @Query("SELECT * FROM director WHERE did = :id LIMIT 1")
    suspend fun findDirectorById(id: Long): Director?

    @Query("SELECT * FROM director WHERE full_name = :fullName LIMIT 1")
    suspend fun findDirectorByName(fullName: String?): Director?

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(director: Director): Long

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(vararg directors: Director)

    @Update(onConflict = OnConflictStrategy.IGNORE)
    suspend fun update(director: Director)

    @Query("DELETE FROM director")
    suspend fun deleteAll()

    @get:Query("SELECT * FROM director ORDER BY full_name ASC")
    val allDirectors: LiveData<List<Director>>
}
```

DirectorDao.kt hosted with ♡ by GitHub view raw

classes created, lets create a database class itself. It should be *abstract* and extend *RoomDatabase*. With *@Database* annotation we provide the array of entities and a version. Usually, there's no need for more than one instance of a DB, thus make it a singleton.

For DB initialisation, use *Room.databaseBuilder()*. In case you don't want to to use coroutines or RxJava just yet, simply add *allowMainThreadQueries() to* the builder. If not provided and you run DB operations on the Main thread, each DB transaction will throw the below exception:

```
java.lang.IllegalStateException: Cannot access database on the main
thread since it may potentially lock the UI for a long period of time
```

I'm also adding a callback hooked to DB's creation (there's also another one for opening — *onOpen()*), so I can pre-populate our database with some data.

What you definitely need to do in *RoomDatabase* class, is to provide abstract methods for getting DAOs. In our case it's *movieDao()* and *directorDao()*. The whole class looks like this now:

```
1  @Database(entities = [Movie::class, Director::class], version = 1)
2  abstract class MoviesDatabase : RoomDatabase() {
3
4      abstract fun movieDao(): MovieDao
5      abstract fun directorDao(): DirectorDao
6
7      companion object {
8          private var INSTANCE: MoviesDatabase? = null
9          private const val DB_NAME = "movies.db"
10
11         fun getDatabase(context: Context): MoviesDatabase {
12             if (INSTANCE == null) {
13                 synchronized(MoviesDatabase::class.java) {
14                     if (INSTANCE == null) {
15                         INSTANCE = Room.databaseBuilder(context.applicationContext, MoviesDataba
16                             // allowMainThreadQueries() // Uncomment if you don't want to use Rx
```

```
                                super.onCreate(db)
                                Log.d("MoviesDatabase", "populating with data...")
                                GlobalScope.launch(Dispatchers.IO) { rePopulateDb(INSTANCE)
                            }
                        }).build()
                }
            }
        }

        return INSTANCE!!
    }
}
```

MoviesDatabase.kt hosted with ♡ by GitHub view raw

rePopulateDb() is a public suspend function, inside of *DbUtils.kt* file, taking *MoviesDatabase* as an input, deleting all its data and inserting an example one:

```
suspend fun rePopulateDb(database: MoviesDatabase?) {
    database?.let { db ->
        withContext(Dispatchers.IO) {
            val movieDao: MovieDao = db.movieDao()
            val directorDao: DirectorDao = db.directorDao()

            movieDao.deleteAll()
            directorDao.deleteAll()

            val directorOne = Director(fullName = "Adam McKay")
            val directorTwo = Director(fullName = "Denis Villeneuve", age = 35)
            val directorThree = Director(fullName = "Morten Tyldum", age = 26)
            val movieOne = Movie(title = "The Big Short", directorId = directorDao.insert(direct
            val dIdTwo = directorDao.insert(directorTwo)
            val movieTwo = Movie(title = "Arrival", directorId = dIdTwo)
            val movieThree = Movie(title = "Blade Runner 2049", directorId = dIdTwo)
            val movieFour = Movie(title = "Passengers", directorId = directorDao.insert(director
            movieDao.insert(movieOne, movieTwo, movieThree, movieFour)
        }
    }
}
```

Next, don't forget about the *ViewModel* classes. *Director* and *Movie* have their own ones. Really they're just wrappers around DAOs methods, but they will take care of updating the UI on database changes.

```
class DirectorsViewModel(application: Application) : AndroidViewModel(application) {

    private val directorDao: DirectorDao = MoviesDatabase.getDatabase(application).directorDao()
    val directorList: LiveData<List<Director>>

    init {
        directorList = directorDao.allDirectors
    }

    suspend fun insert(vararg directors: Director) {
        directorDao.insert(*directors)
    }

    suspend fun update(director: Director) {
        directorDao.update(director)
    }

    suspend fun deleteAll() {
        directorDao.deleteAll()
    }
}
```

DirectorsViewModel.kt hosted with ♡ by GitHub view raw

The fragment to display Directors's list, looks like the following:

```
class DirectorsListFragment : Fragment() {

    private lateinit var directorsListAdapter: DirectorsListAdapter
    private lateinit var directorsViewModel: DirectorsViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        initData()
```

```
12          val view = inflater.inflate(R.layout.fragment_directors, container, false)
13          initView(view)
14
15          return view
16      }
17
18      private fun initData() {
19          directorsViewModel = ViewModelProvider(this).get(DirectorsViewModel::class.java)
20          directorsViewModel.directorList.observe(this,
21              Observer { directors: List<Director> ->
22                  directorsListAdapter.setDirectorList(directors)
23              }
24          )
25      }
26
27      private fun initView(view: View) {
28          val recyclerView: RecyclerView = view.findViewById(R.id.recyclerview_directors)
29          directorsListAdapter = DirectorsListAdapter(requireContext())
30          recyclerView.adapter = directorsListAdapter
31          recyclerView.addItemDecoration(DividerItemDecoration(context, DividerItemDecoration.VERT
32          recyclerView.layoutManager = LinearLayoutManager(context)
33      }
34
35      fun removeData() {
36          GlobalScope.launch(Dispatchers.IO) { directorsViewModel.deleteAll() }
37      }
38
39      companion object {
40          fun newInstance(): DirectorsListFragment = DirectorsListFragment()
41      }
42  }
```

DirectorsListFragment.kt hosted with ♡ by GitHub view raw

What's interesting in this code, is the *DirectorsViewModel*, that I get from *ViewModelProvider* and observe directors list from it. Whenever *onChanged()* gets called, new directors list is set on the list adapter and we see changes in the UI.

on dialog's *Save* press, using IO dispatcher, like this:

GlobalScope.launch(Dispatchers.IO) { saveDirector(directorEditText.text.toString()) }

And the *saveDirector()* function itself:

```
1   private suspend fun saveDirector(fullName: String) {
2           if (TextUtils.isEmpty(fullName)) {
3               return
4           }
5           val directorDao = MoviesDatabase.getDatabase(requireContext()).directorDao()
6           if (directorFullNameExtra != null) {
7               // clicked on item row -> update
8               val directorToUpdate = directorDao.findDirectorByName(directorFullNameExtra)
9               if (directorToUpdate != null) {
10                  if (directorToUpdate.fullName != fullName) {
11                      directorToUpdate.fullName = fullName
12                      directorDao.update(directorToUpdate)
13                  }
14              }
15          } else {
16              directorDao.insert(Director(fullName = fullName))
17          }
18      }
```

DirectorSaveDialogFragment.kt hosted with ♡ by GitHub view raw

I don't need to check if the name is unique when inserting, as Room is doing it for me. It will ignore the transaction if the same name already exists in DB (thanks to *OnConflictStrategy.IGNORE*). I could do the same with update actually ;)

The method for saving movie is a bit more complicated, because I need an id of either newly inserted or updated or just the old director from DB. Once I have it, I provide it to new movie, or update an existing one.

```
1   private suspend fun saveMovie(movieTitle: String, movieDirectorFullName: String) {
2           if (TextUtils.isEmpty(movieTitle) || TextUtils.isEmpty(movieDirectorFullName)) {
3               return
4           }
```

```
        if (movieDirectorFullNameExtra != null) {
            // clicked on item row -> update
            val directorToUpdate = directorDao.findDirectorByName(movieDirectorFullNameExtra)
            if (directorToUpdate != null) {
                directorId = directorToUpdate.id
                if (directorToUpdate.fullName != movieDirectorFullName) {
                    directorToUpdate.fullName = movieDirectorFullName
                    directorDao.update(directorToUpdate)
                }
            }
        } else {
            // we need director id for movie object; in case director is already in DB,
            // insert() would return -1, so we manually check if it exists and get
            // the id of already saved director
            val newDirector = directorDao.findDirectorByName(movieDirectorFullName)
            directorId = newDirector?.id ?: directorDao.insert(Director(fullName = movieDirector
        }

        if (movieTitleExtra != null) {
            // clicked on item row -> update
            val movieToUpdate = movieDao.findMovieByTitle(movieTitleExtra)
            if (movieToUpdate != null) {
                if (movieToUpdate.title != movieTitle) {
                    movieToUpdate.title = movieTitle
                    if (directorId != -1L) {
                        movieToUpdate.directorId = directorId
                    }
                    movieDao.update(movieToUpdate)
                }
            }
        } else {
            // we can have many movies with same title but different director
            movieDao.insert(Movie(title = movieTitle, directorId = directorId))
        }
    }
```

MovieSaveDialogFragment.kt hosted with ♡ by GitHub view raw

Now, download the app, click *Run*, add some directors and movies. Check if the movie is removed when you delete its director. Check if update works, add your custom fields and

By the way, this is how the DB's current schema looks like in */movies-room/app/src/main/assets/schemas/com.lomza.moviesroom.db.MoviesDatabase/1.json* (*1* because of *version = 1* in *MoviesDatabase* setting.

```
{
  "formatVersion": 1,
  "database": {
    "version": 1,
    "identityHash": "627455825d506754f171b32a983cfc02",
    "entities": [
      {
        "tableName": "movie",
        "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`mid` INTEGER PRIMARY KEY AUT
        "fields": [
          {
            "fieldPath": "id",
            "columnName": "mid",
            "affinity": "INTEGER",
            "notNull": true
          },
          {
            "fieldPath": "title",
            "columnName": "title",
            "affinity": "TEXT",
            "notNull": true
          },
          {
            "fieldPath": "directorId",
            "columnName": "directorId",
            "affinity": "INTEGER",
            "notNull": true
          }
        ],
        "primaryKey": {
          "columnNames": [
            "mid"
          ],
          "autoGenerate": true
        },
        "indices": [
          {
```

```
41                "title"
42              ],
43              "createSql": "CREATE INDEX IF NOT EXISTS `index_movie_title` ON `${TABLE_NAME}` (`t
44            },
45            {
46              "name": "index_movie_directorId",
47              "unique": false,
48              "columnNames": [
49                "directorId"
50              ],
51              "createSql": "CREATE INDEX IF NOT EXISTS `index_movie_directorId` ON `${TABLE_NAME}
52            }
53          ],
54          "foreignKeys": [
55            {
56              "table": "director",
57              "onDelete": "CASCADE",
58              "onUpdate": "NO ACTION",
59              "columns": [
60                "directorId"
61              ],
62              "referencedColumns": [
63                "did"
64              ]
65            }
66          ]
67        },
68        {
69          "tableName": "director",
70          "createSql": "CREATE TABLE IF NOT EXISTS `${TABLE_NAME}` (`did` INTEGER PRIMARY KEY AUT
71          "fields": [
72            {
73              "fieldPath": "id",
74              "columnName": "did",
75              "affinity": "INTEGER",
76              "notNull": true
77            },
78            {
79              "fieldPath": "fullName",
80              "columnName": "full_name",
81              "affinity": "TEXT",
82              "notNull": true
```

```
85          "primaryKey": {
86            "columnNames": [
87              "did"
88            ],
89            "autoGenerate": true
90          },
91          "indices": [
92            {
93              "name": "index_director_full_name",
94              "unique": true,
95              "columnNames": [
96                "full_name"
97              ],
98              "createSql": "CREATE UNIQUE INDEX IF NOT EXISTS `index_director_full_name` ON `${TAI
99            }
100         ],
101         "foreignKeys": []
102       }
103     ],
104     "views": [],
105     "setupQueries": [
106       "CREATE TABLE IF NOT EXISTS room_master_table (id INTEGER PRIMARY KEY,identity_hash TEXT)
107       "INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42, '627455825d506754
108     ]
109   }
110 }
```

1.json hosted with ♡ by GitHub view raw

Just to sum up, here are some advantages and disadvantages of using **Room** database in Android:

Pros:

- smooth integration with **LiveData**
- it's an **official Google's SQLite ORM**

- super easy database creation with Room Builder
- works smoothly with Rx or Coroutines

Cons:

- no default values for columns
- no *unique* annotation
- no cascading with @*Relation* annotation; can be applied to *List* or *Set* only
- not so many features, (yet :)

Further reading:

Room Training Guide

7 Pro-tips for Room

Android Architecture Components: Room — Relationships

Understanding Migrations with Room

Squeezing Performance from SQLite: Indexes? Indexes!

Lots of examples with Room

lomza/movies-room

Example usage of Room database. Contribute to lomza/movies-room development by creating an account on GitHub.

github.com

Android Room Sqlite Development Programming