Little Robots

Room:
the blessed
object mapper

Hugo Visser
@botteaap
hugo@littlerobots.nl

Android persistence

SharedPreferences: small amounts of key-value pairs

SQLite: Large(r) amounts of structured data (Objects)

Persistence frameworks

Backed by SQLite: SQL Delight, GreenDAO, DBFlow, Cupboard :), ...

NoSQL: Realm, Objectbox, ...

Knowing SQL

- SQL Delight: everything is SQL → little or no abstractions, source of truth
- Room → Queries and migrations are SQL
- Cupboard → SQL as fallback

Key points

Part of Architecture Components → opinionated app architecture

Does not support entity relations by design (e.g. no lazy loading)

Database operations use SQL, no abstractions.

Current stable version: 1.0.0

Created by Google

Getting started (Java)

```
android {
   defaultConfig {
       javaCompileOptions {
           annotationProcessorOptions {
               arguments = ["room.schemaLocation": "$projectDir/schemas".toString()]
           }
       }
   }
}

dependencies {
   implementation "android.arch.persistence.room:runtime:1.0.0"
   annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
   // 26.1.+ implements LifecyleOwner for LiveData and friends
   implementation 'com.android.support:appcompat-v7:27.0.2'
   // For LiveData & Lifecycle support
   implementation "android.arch.lifecycle:livedata:1.1.0"
   annotationProcessor "android.arch.lifecycle:compiler:1.0.0"
}
```

Getting started (Kotlin)

```
apply plugin: 'kotlin-kapt'

kapt {
    arguments {
        arg("room.schemaLocation", "${projectDir}/schemas")
    }
}

dependencies {
    implementation "android.arch.persistence.room:runtime:1.0.0"
    kapt "android.arch.persistence.room:compiler:1.0.0"

    // 26.1.+ implements LifecyleOwner for LiveData and friends
    implementation 'com.android.support:appcompat-v7:27.0.2'
    // For LiveData & Lifecycle support
    implementation "android.arch.lifecycle:livedata:1.1.0"
    kapt "android.arch.lifecycle:compiler:1.1.0"
}
```

Room annotation processor

Code generation

Compile time (query) validation

Least amount of runtime overhead

Room main ingredients

Entities → your objects

DAO’s → how you persist and retrieve entities

RoomDatabase → where everything is persisted

Entities

```
@Entity
public class User {
   @PrimaryKey(autoGenerate = true)
   public Long id;
   public String firstName;
   public String lastName;
}
```

Entities

```
@Entity
public class User {
   @PrimaryKey(autoGenerate = true)
   public String id;
   public String firstName;
   public String lastName;
}
```

```
error: If a primary key is annotated with
autoGenerate, its type must be int, Integer, long
or Long.
```

Entities

```
@Entity
public class User {
   @PrimaryKey
   public String id;
   public String firstName;
   public String lastName;
}
```

```
error: You must annotate primary keys with
@NonNull. SQLite considers this a bug and Room
does not allow it.
```

Entities

```
@Entity
public class User {
   @PrimaryKey
   @NonNull
   private final String id;
   public String firstName;
   public String lastName;

   public User(@NonNull String id) {
       this.id = id;
   }

   @Ignore
   public User() {
       this.id = UUID.randomUUID().toString();
   }

   @NonNull
   public String getId() {
       return id;
   }
}
```

Entities

```java
@Entity
public class User {
   @PrimaryKey
   @NonNull
   private final String id;
   @NonNull
   private final String firstName;
   private final String lastName;

   public User(@NonNull String id, @NonNull String firstName, String lastName) {
       this.id = id;
       this.firstName = firstName;
       this.lastName = lastName;
   }

   @Ignore
   public User(String firstName, String lastName) {
       this(UUID.randomUUID().toString(), firstName, lastName);
   }

   // getters -->
}
```

```java
@NonNull
public String getId() {
    return id;
}

@NonNull
public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}
```

Entities

```
@Entity
data class User(@PrimaryKey val id: String = UUID.randomUUID().toString(),
                val firstName: String,
                val lastName: String?)


val user = User(firstName = "Hugo", lastName = "Visser")
```

Entities

```
@Entity(tableName = "k_user", primaryKeys = arrayOf("firstName", "lastName"))
data class User(val id: String = UUID.randomUUID().toString(),
                val firstName: String,
                val lastName: String)
```

Entities

```
@Entity(indices = arrayOf(Index("firstName", "lastName", unique = true)))
data class User(@PrimaryKey val id: String = UUID.randomUUID().toString(),
                @ColumnInfo(index = true) val firstName: String,
                @ColumnInfo(name = "last_name",
                            typeAffinity = ColumnInfo.TEXT,
                            collate = ColumnInfo.NOCASE) val lastName: String?)
```

Entities

Annotated Java or Kotlin classes

Must specify a non-null primary key

Database representation is controlled by annotations

Entities vs POJOs

Distinction is made in the Room documentation

`@Entity` → Object that has a table in the database

POJO → *any* object that a result can be mapped to

Data access objects (DAOs)

Define methods of working with entities

`interface` or `abstract` class marked as `@Dao`

One or many DAOs per database

DAO @Query

```
@Dao
interface UserDao {
    @Query("select * from User limit 1")
    fun firstUser(): User

    @Query("select * from User")
    fun allUsers(): List<User>

    @Query("select firstName from User")
    fun firstNames(): List<String>

    @Query("select * from User where firstName = :fn")
    fun findUsersByFirstName(fn: String): List<User>

    @Query("delete from User where lastName = :ln")
    fun deleteUsersWithLastName(ln: String): Int

    @Query("select firstName as first, lastName as last from User where lastName = :ln")
    fun findPersonByLastName(ln: String): List<Person>
}
```

Map result on any POJO

```
data class Person(val first: String, val last:String)
```

Room matches column names

Must be able to convert column to type

@Insert

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
fun addUser(user: User)

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

@Insert
fun addUserWithBooks(user: User, books: List<Book>)
```

@Delete

```
@Delete
fun deleteUser(user: User)

@Delete
fun deleteUsers(users: List<User>)
```

@Update

```
@Update
fun updateUser(user: User)

@Update
fun updateUsers(users: List<User>)
```

User with books

```
@Entity
data class User @Ignore constructor(@PrimaryKey val id: String,
                 val firstName: String,
                 val lastName: String?,
                 @field:Ignore val books: List<Book>) {

   constructor(id: String = UUID.randomUUID().toString(),
               firstName: String,
               lastName: String?) : this(id, firstName, lastName, listOf())
}
```

Needs `@field:Ignore` on `books`

Primary constructor is `@Ignored` for Room

Abstract DAO class 

```
@Dao
abstract class UserBookDao {
   @Insert
   protected abstract fun insertSingleUser(user: User)
   @Insert
   protected abstract fun insertBooks(books: List<Book>)

   @Transaction
   open fun insertUser(user: User) {
       insertBooks(user.books)
       insertSingleUser(user)
   }
}
```

The database

Annotated with `@Database`

`abstract` class that extends `RoomDatabase`

`Room.databaseBuilder()` → create singleton (use DI, or other means)

```
@Database(entities = arrayOf(User::class, Book::class), version = 1)
abstract class MyDatabase : RoomDatabase() {
    abstract fun getUserDao(): UserDao

    companion object {
        private var instance: MyDatabase? = null

        fun getInstance(context: Context): MyDatabase {
            return instance ?: Room.databaseBuilder(context, MyDatabase::class.java, "my.db").
                                                                    build().also { instance = it }
        }
    }
}
```

Let’s do this!

```
val dao = MyDatabase.getInstance(context).getUserDao()
val users = dao.allUsers()
```

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

It was **never** okay to do database ops on the main thread!

(but we all did it)

`@Insert`, `@Delete` and `@Update`

Nothing provided by Room to run off the main thread

Run in an executor, AsyncTask, RxJava, Kotlin co-routines

For testing only: `allowMainThreadQueries()` on the database builder

Abstract Dao strikes again

```
abstract class BaseDao<in T> {
    @Insert
    protected abstract fun insertSync(vararg obj:T): LongArray

    suspend fun insert(vararg obj: T): LongArray {
        return insertSync(*obj)
    }
}

fun updateMyData() {
    val dao = MyDatabase.getInstance(this).getMyDao()
    async {
        dao.insert(User(firstName = "John", lastName = "Doe"))
    }
}
```

Observable queries

Using LiveData → `android.arch.lifecycle:livedata`

RxJava 2 → `android.arch.persistence.room:rxjava2`

Delivers result async with updates

```
@Query("select * from User")
fun allUsers(): LiveData<List<User>>
@Query("select * from User")
fun allUsersRx(): Flowable<List<User>>
```

Observable queries

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val dao = MyDatabase.getInstance(this).getUserDao()
    dao.allUsers().observe(this, Observer {
        it?.let {
            showUsers(it)
        }
    })
}

private fun showUsers(users: List<User>) {
}
```

TypeConverter

Convert a field (object) to column and vice versa

Can be scoped from field, entity, ..., Database

No default converters for things like `Date` or `enums` :(

TypeConverter

```
object MyTypeConverters {
    @TypeConverter
    @JvmStatic
    fun dateToLong(date: Date?) : Long? {
        return date?.time
    }

    @TypeConverter
    @JvmStatic
    fun longToDate(value: Long?) : Date? {
        return if (value == null) {
            null
        } else {
            Date(value)
        }
    }
}
```

TypeConverter

```
@Entity
data class User(@PrimaryKey val id: String,
                val firstName: String,
                val lastName: String,
                @field:TypeConverters(MyTypeConverters::class) val birthDate: Date)

@Entity
@TypeConverters(MyTypeConverters::class)
data class User(@PrimaryKey val id: String,
                val firstName: String,
                val lastName: String,
                val birthDate: Date)

@TypeConverters(MyTypeConverters::class)
@Database(...)
abstract class MyDatabase : RoomDatabase() {
}
```

TypeConverter

```
@Query("select * from User where birthDate > :date")
fun bornAfterDate(date: Date): List<User>
```

Complex objects

Flatten objects onto a table using `@Embedded`

```
data class Address(val street: String,
                   val houseNumber: String,
                   val city: String)

@Entity
class User(@PrimaryKey(autoGenerate = true) val id: Long,
           val name: String,
           val address: Address)
```

@Embedded

User table will now have a `street`, `houseNumber` and `city` column too

```
data class Address(val street: String,
                   val houseNumber: String,
                   val city: String)

@Entity
class User(@PrimaryKey(autoGenerate = true) val id: Long,
           val name: String,
           @Embedded val address: Address)
```

Migrations

When adding, changing, removing entities: schema updates

Database version number

Migrate from version x to version y

Missing migration will crash your app, but save user data

Creating & using migrations

```
class UserBirthDayMigration : Migration(1, 2) {
   override fun migrate(database: SupportSQLiteDatabase) {
       // execute the statements required
       database.execSQL("ALTER TABLE User ADD COLUMN `birthDate` INTEGER")
   }
}
```

```
Room.databaseBuilder(context, MyDatabase::class.java, "mydatabase.db").
       addMigrations(UserBirthDayMigration(), MyOtherMigration()).
       build()
```

Migrations

Tip: use exported schema definitions `1.json`, `2.json` etc

Testing migrations

https://goo.gl/3F1kvQ

`fallbackToDestructiveMigration()` discards data

Large result sets w/ Paging

Cursors (+ CursorAdapter) have their problems

https://goo.gl/CfVs7C

Room integrates with Paging architecture component

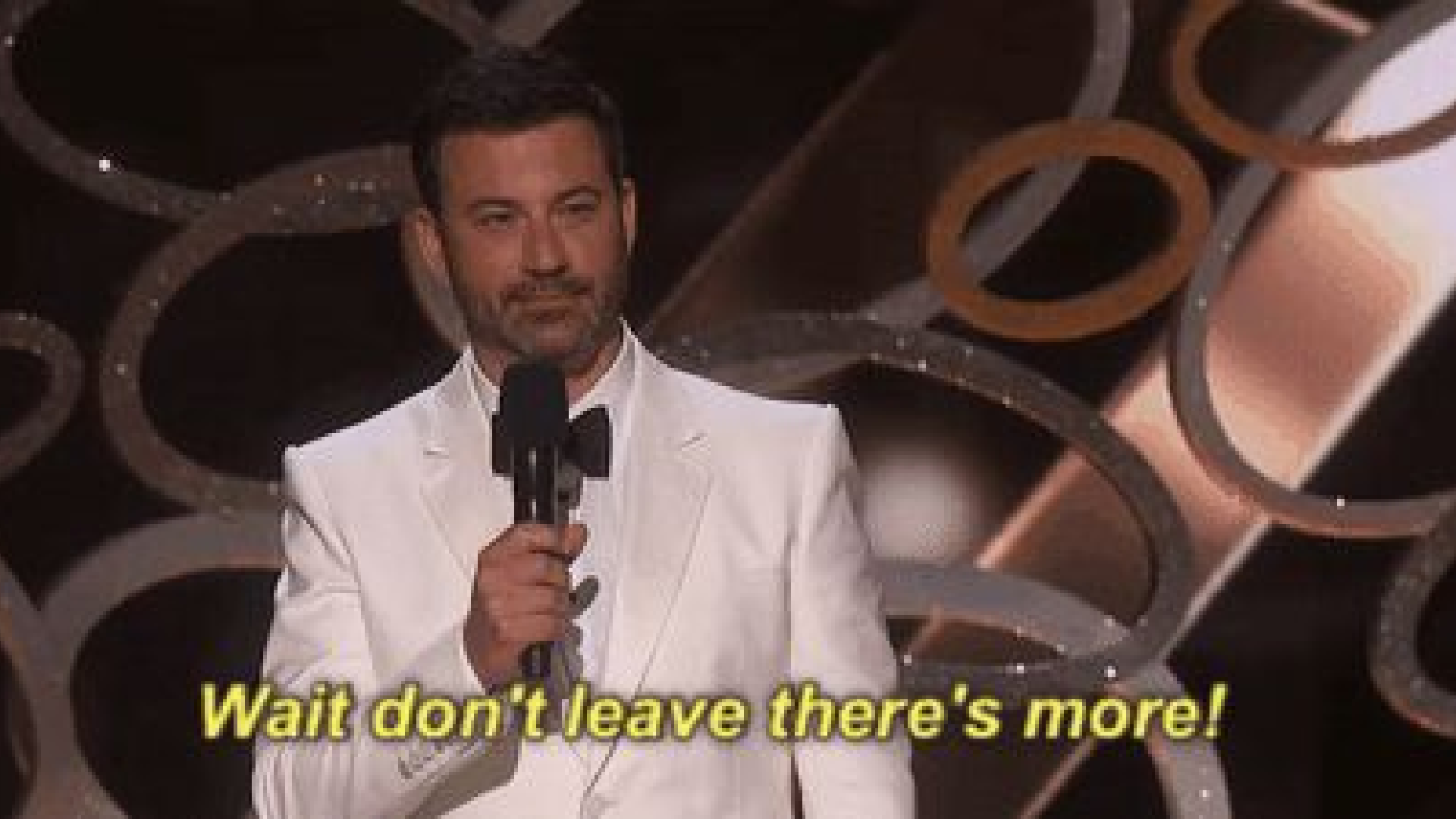
https://developer.android.com/topic/libraries/architecture/paging.html

Data sources for `PagedLists`

```
// this might be a huge list
@Query("select * from User")
fun allUsers():List<User>
```

```
// Generates a PositionalDataSource (not observable)
@Query("select * from User")
fun allUsers(): DataSource.Factory<Int, User>
```

```
// Generates a LivePagedListBuilder (observable) that uses LIMIT and OFFSET
@Query("select * from User")
fun allUsers(): LivePagedListBuilder<Int, User>
```


Wait don't leave there's more!

More stuff

Foreign key support (cascading deletes, integrity checks)

`SupportSQLiteDatabase` allows for alternative SQLite implementations

`@Relation` annotation to eager fetch relations in POJOs only

Testing support

Wrapping up

Full featured persistence library

Kotlin support

Integration with other architecture components + RxJava 2

Learn more

https://d.android.com/topic/libraries/architecture/room.html

https://d.android.com/reference/android/arch/persistence/room/package-summary.html

https://medium.com/@florina.muntenescu

https://goo.gl/NTKSb6

Thanks

@botteaap

hugo@littlerobots.nl

speakerdeck.com/hugovisser