

# Lecture #4

## Local Persistence &

# Architecture Components

Fall 2023

# Local Persistence Options

- Internal storage
  - Internal cache files
- External storage
- Shared preferences
- **Databases**



# Options

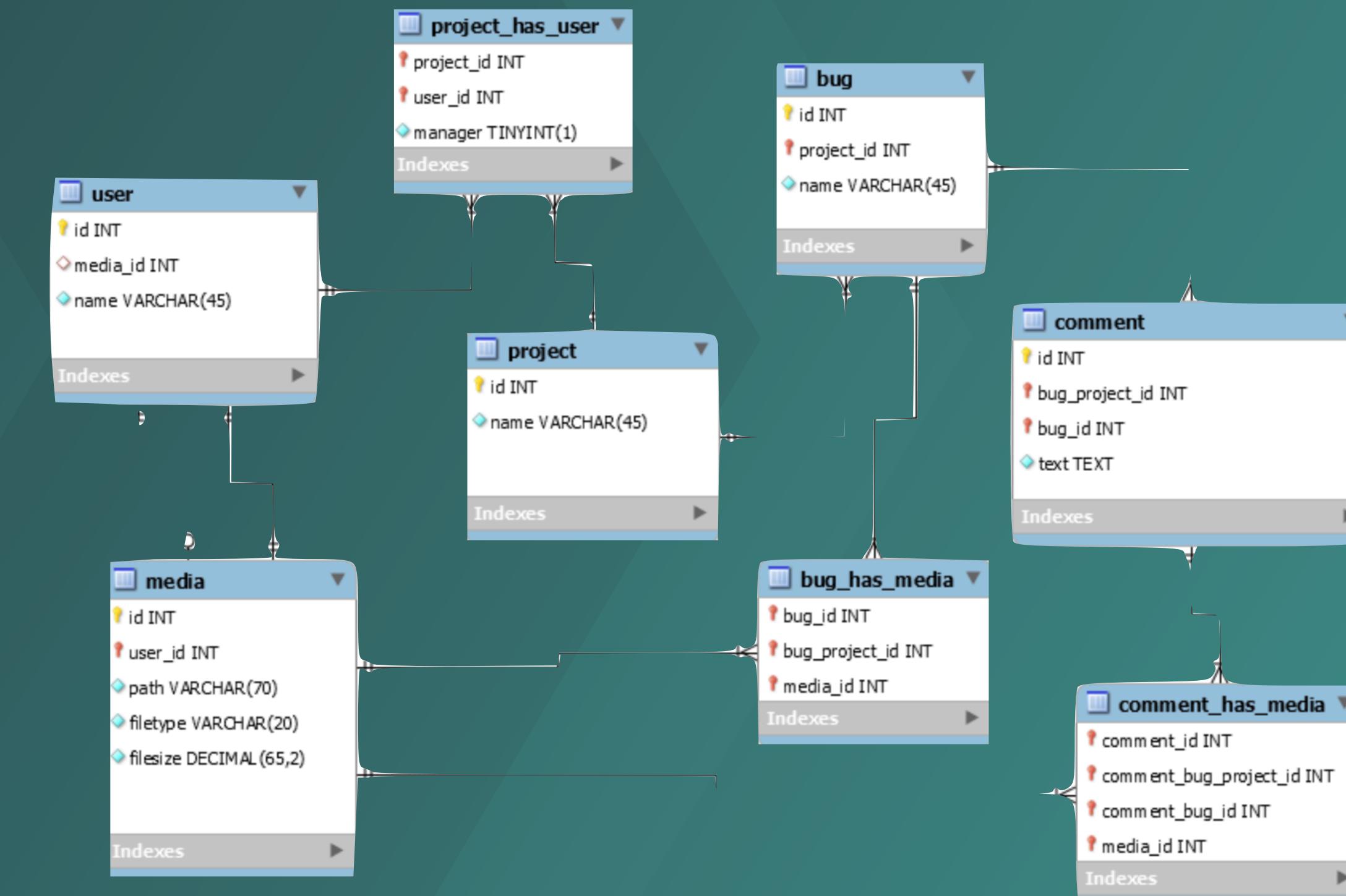
- SQLite
- Realm
- Room
- SQLDelight



# SQLite

Define a schema and a contract

```
object FeedReaderContract {  
    // Table contents are grouped  
    // together in an anonymous object.  
    object FeedEntry : BaseColumns {  
        const val TABLE_NAME = "entry"  
        const val COLUMN_NAME_TITLE = "title"  
        const val COLUMN_NAME_SUBTITLE = "subtitle"  
    }  
}
```



# SQLite Helper

Create a database using an SQL helper

```
private const val SQL_CREATE_ENTRIES = """  
CREATE TABLE ${FeedEntry.TABLE_NAME} (  
    ${BaseColumns._ID} INTEGER PRIMARY KEY,  
    ${FeedEntry.COLUMN_NAME_TITLE} TEXT,  
    ${FeedEntry.COLUMN_NAME_SUBTITLE} TEXT)  
"""
```

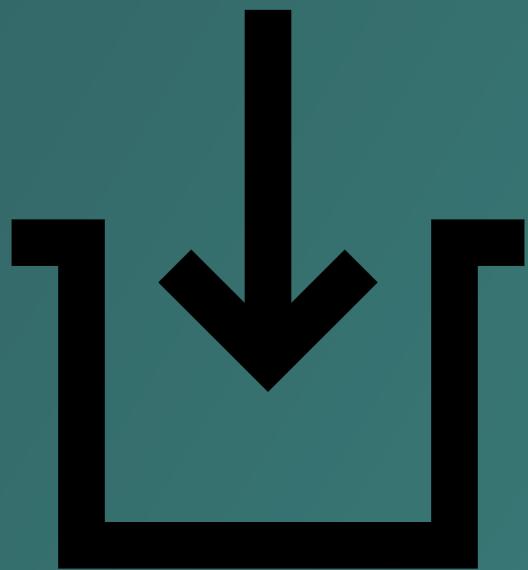
```
private const val SQL_DELETE_ENTRIES =  
    "DROP TABLE IF EXISTS ${FeedEntry.TABLE_NAME}"
```



# SQLite

```
class FeedReaderDbHelper(context: Context) :  
    SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {  
    override fun onCreate(db: SQLiteDatabase) {  
        db.execSQL(SQL_CREATE_ENTRIES)  
    }  
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
        // This database is only a cache for online data, so its upgrade policy is  
        // to simply to discard the data and start over  
        db.execSQL(SQL_DELETE_ENTRIES)  
        onCreate(db)  
    }  
    override fun onDowngrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {  
        onUpgrade(db, oldVersion, newVersion)  
    }  
    companion object {  
        // If you change the database schema, you must increment the database version.  
        const val DATABASE_VERSION = 1  
        const val DATABASE_NAME = "FeedReader.db"  
    }  
}
```

# SQLite - Insert



```
// Gets the data repository in write mode
val db = dbHelper.writableDatabase
// Create a new map of values, where column names are the keys
val values = ContentValues().apply {
    put(FeedEntry.COLUMN_NAME_TITLE, title)
    put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle)
}
// Insert the new row, returning the primary key value of the new row
val newRowId = db?.insert(FeedEntry.TABLE_NAME, null, values)
```

# SQLite - Query

```
val dbHelper = FeedReaderDbHelper(context)
val db = dbHelper.readableDatabase
// Define a projection that specifies which columns from the database
// you will actually use after this query.
val projection = arrayOf(
    BaseColumns._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE)
// Filter results WHERE "title" = 'My Title'
val selection = "${FeedEntry.COLUMN_NAME_TITLE} = ?"
val selectionArgs = arrayOf("My Title")
// How you want the results sorted in the resulting Cursor
val sortOrder = "${FeedEntry.COLUMN_NAME_SUBTITLE} DESC"
val cursor = db.query(
    FeedEntry.TABLE_NAME // The table to query
```



# SQLite - Query

```
val dbHelper = FeedReaderDbHelper(context)  
  
val db = dbHelper.readableDatabase  
val projection = arrayOf(...)  
val selection = "${FeedEntry.COLUMN_NAME_TITLE} = ?"  
val selectionArgs = arrayOf("My Title")  
val sortOrder = "${FeedEntry.COLUMN_NAME_SUBTITLE} DESC"  
val cursor = db.query(...)  
val itemIds = mutableListOf<Long>()  
with(cursor) {  
    while (moveToNext()) {  
        val itemId = getLong(getColumnIndexOrThrow(BaseColumns._ID))  
        itemIds.add(itemId)  
    }  
}
```



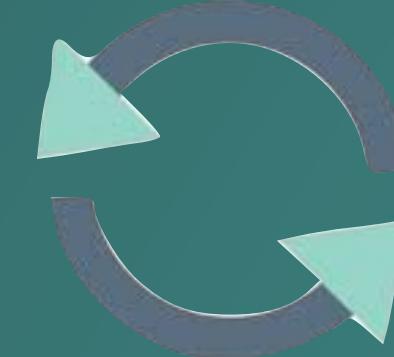
# SQLite - Delete



```
val dbHelper = FeedReaderDbHelper(context)

val db = dbHelper.writableDatabase
// Define 'where' part of query.
val selection = "${FeedEntry.COLUMN_NAME_TITLE} LIKE ?"
// Specify arguments in placeholder order.
val selectionArgs = arrayOf("MyTitle")
// Issue SQL statement.
val deletedRows = db.delete(FeedEntry.TABLE_NAME, selection, selectionArgs)
```

# SQLite - Update



```
val dbHelper = FeedReaderDbHelper(context)
val db = dbHelper.writableDatabase
// New value for one column
val title = "MyNewTitle"
val values = ContentValues().apply {
    put(FeedEntry.COLUMN_NAME_TITLE, title)
}
// Which row to update, based on the title
val selection = "${FeedEntry.COLUMN_NAME_TITLE} LIKE ?"
val selectionArgs = arrayOf("MyOldTitle")
val count = db.update(
    FeedEntry.TABLE_NAME,
    values,
```

# SQLite - Management

```
val dbHelper = FeedReaderDbHelper(context)
```

```
val db = dbHelper.writableDatabase
```

```
val db = dbHelper.readableDatabase
```

```
override fun onDestroy() {  
    dbHelper.close()  
    super.onDestroy()  
}
```

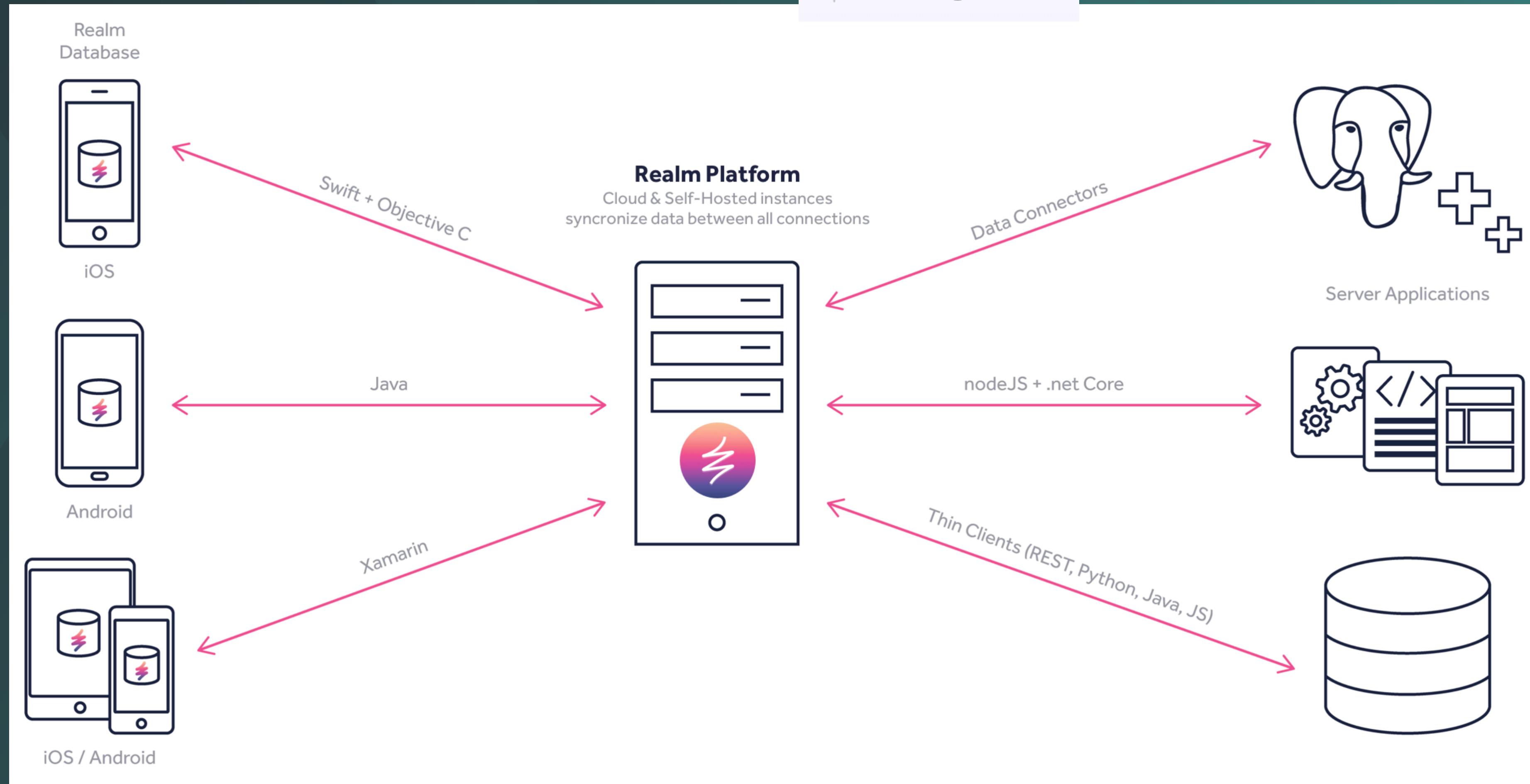
# SQLite - Caution

DEMO

- There is **no compile-time verification of raw SQL queries**. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to **use lots of boilerplate code** to convert between SQL queries and data objects.



# Realm



<https://realm.io>

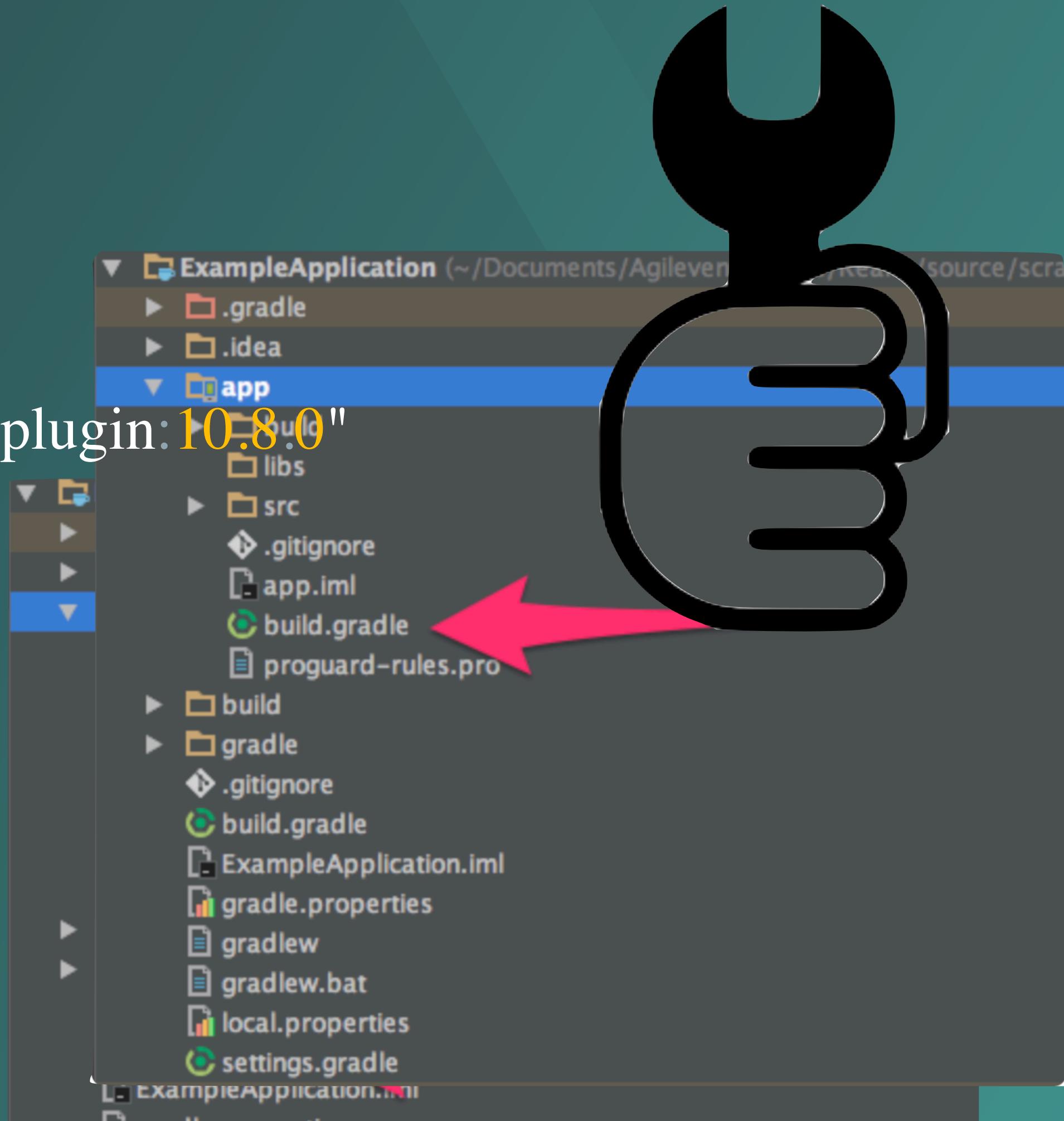
# Realm - Installation

In project level build.gradle:

```
buildscript {  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath "io.realm:realm-gradle-plugin:10.8.0"  
    }  
}
```

In module level build.gradle:

```
apply plugin: 'realm-android'
```



# Realm - Domain

```
open class Dog : RealmObject() {  
    var name: String? = null  
    @LinkingObjects("dog")  
    val owners: RealmResults<Person>? = null  
}
```

```
open class Person(  
    @PrimaryKey var id: Long = 0,  
    var name: String = "",  
    var age: Int = 0,  
    // Other objects in a one-to-one  
    // relation must also subclass RealmObject  
    var dog: Dog? = null  
) : RealmObject()
```

# Realm - Usage

```
// Use them like regular java objects  
Dog dog = new Dog();  
dog.setName("Rex");  
dog.setAge(1);
```

**Initialization**

```
// Initialize Realm (just once per application)  
Realm.init(context);
```

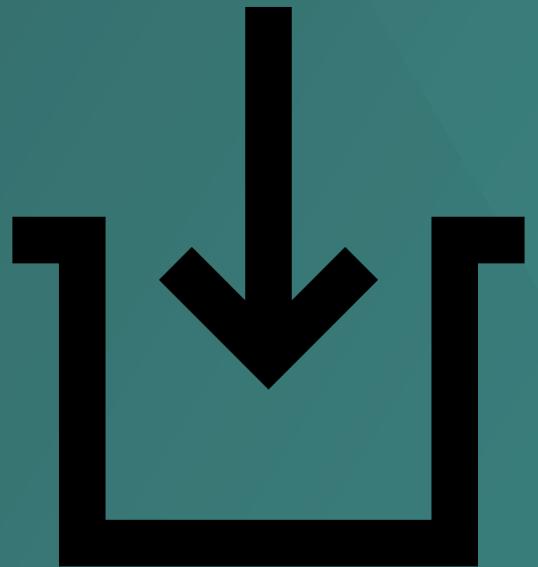
```
// Get a Realm instance for this thread  
Realm realm = Realm.getDefaultInstance();
```

```
// Query Realm for all dogs younger than 2 years old  
final RealmResults<Dog> puppies = realm.where(Dog.class).  
    lessThan("age", 2).findAll();  
puppies.size();
```

**Usage**

# Realm - Insert

```
open class Person(  
    @PrimaryKey var id: Long = 0,  
    var name: String = "",  
    var age: Int = 0,  
    // Other objects in a one-to-one  
    // relation must also subclass RealmObject  
    var dog: Dog? = null  
) : RealmObject()
```



```
realm.executeTransaction { realm ->  
    // Add a person  
    val person = realm.createObject<Person>(0)  
    person.name = "Young Person"  
    person.age = 14  
}
```

# Realm - Query

```
open class Person(  
    @PrimaryKey var id: Long = 0,  
    var name: String = "",  
    var age: Int = 0,  
    // Other objects in a one-to-one  
    // relation must also subclass RealmObject  
    var dog: Dog? = null  
) : RealmObject()
```

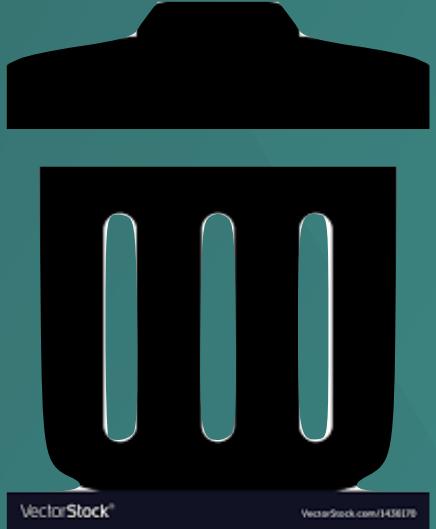


```
val age = 22  
val persons = realm.where<Person>().  
    equalTo("age", age).findAll()!!
```

# Realm - Delete

```
open class Person(  
    @PrimaryKey var id: Long = 0,  
    var name: String = "",  
    var age: Int = 0,  
    // Other objects in a one-to-one  
    // relation must also subclass RealmObject  
    var dog: Dog? = null  
) : RealmObject()
```

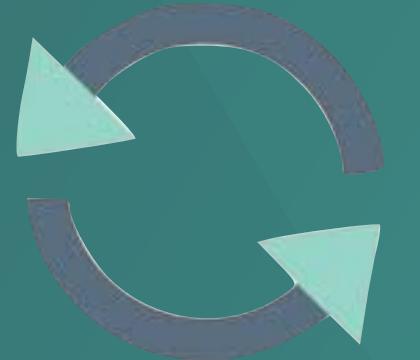
```
val age = 22  
val persons = realm.where<Person>().  
    equalTo("age", age).findAll()!!  
persons.deleteAllFromRealm()
```



# Realm - Update

DEMO

```
open class Person(  
    @PrimaryKey var id: Long = 0,  
    var name: String = "",  
    var age: Int = 0,  
    // Other objects in a one-to-one  
    // relation must also subclass RealmObject  
    var dog: Dog? = null  
) : RealmObject()  
    // Find the first person (no query conditions)  
    // and read a field  
    val person = realm.where<Person>().findFirst()!!  
    // Update person in a transaction  
    realm.executeTransaction { _ ->  
        person.name = "Updated Person"  
        person.age = 99  
    }
```



# Android Jetpack



**Accelerate Development**



**Eliminate boilerplate code**



**Build high quality, robust apps**



<https://developer.android.com/jetpack/>

# Android Jetpack Components



## Foundation

AppCompat  
Android KTX  
Multidex  
Test



## Architecture

Data Binding  
Lifecycles  
LiveData  
Navigation  
Paging  
Room  
ViewModel  
WorkManager



## Behavior

Download Manager  
Media & playback  
Notifications  
Permissions  
Sharing  
Slices



## UI

Animations & Transitions  
Auto  
Emoji  
Fragment  
Layout  
Palette  
TV  
Wear OS

# Adding Components

In project level build.gradle:

```
allprojects {  
    repositories {  
        google()  
        jcenter()  
    }  
}
```



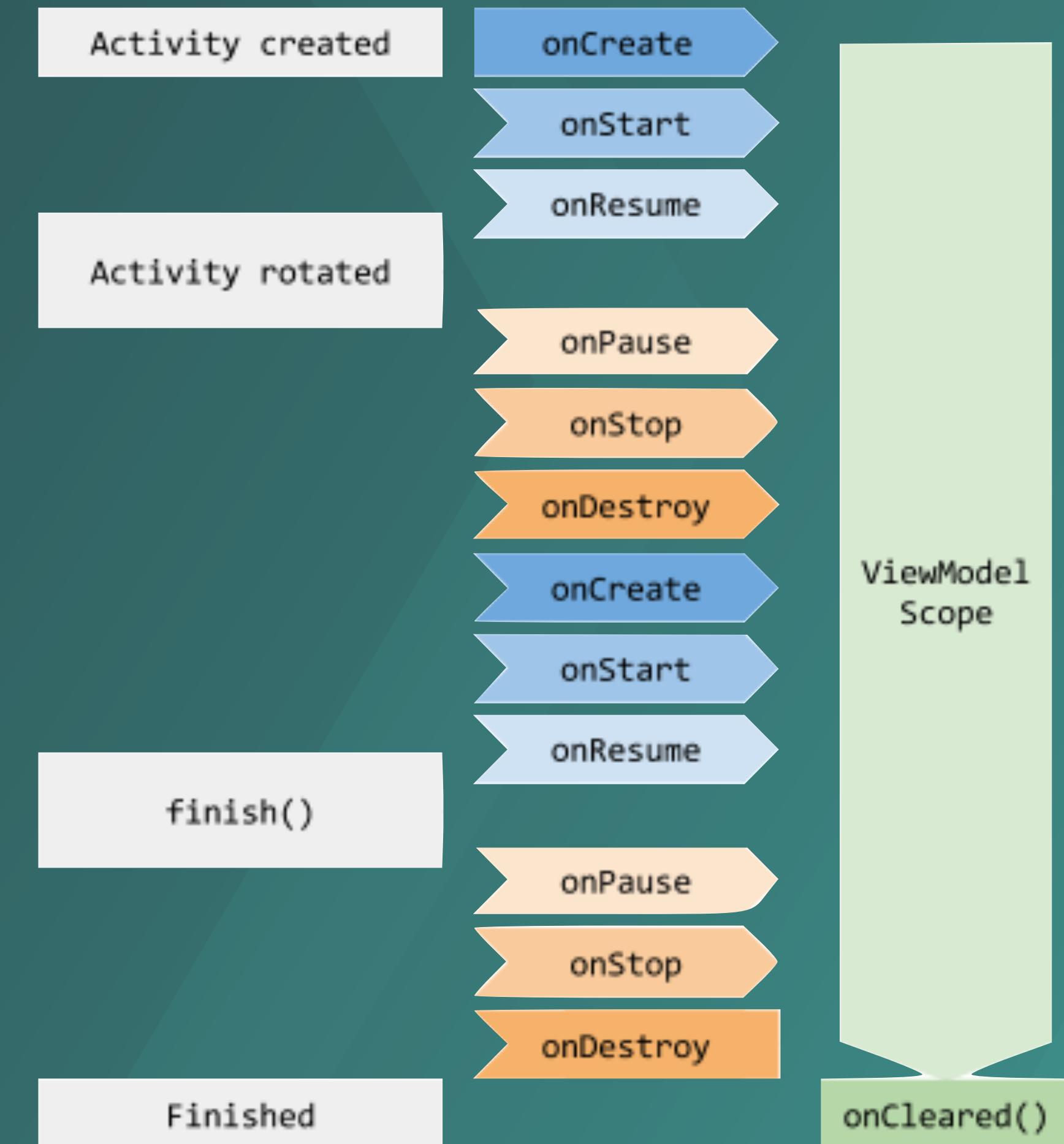
In module level build.gradle:

```
dependencies {  
    def lifecycle_version = "2.3.1"  
    // ViewModel and LiveData  
    implementation "androidx.lifecycle:lifecycle-extensions-ktx:$lifecycle_version"  
    // alternatively - just ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"  
    // alternatively - just LiveData  
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"  
}
```

<https://developer.android.com/topic/libraries/architecture/adding-components>

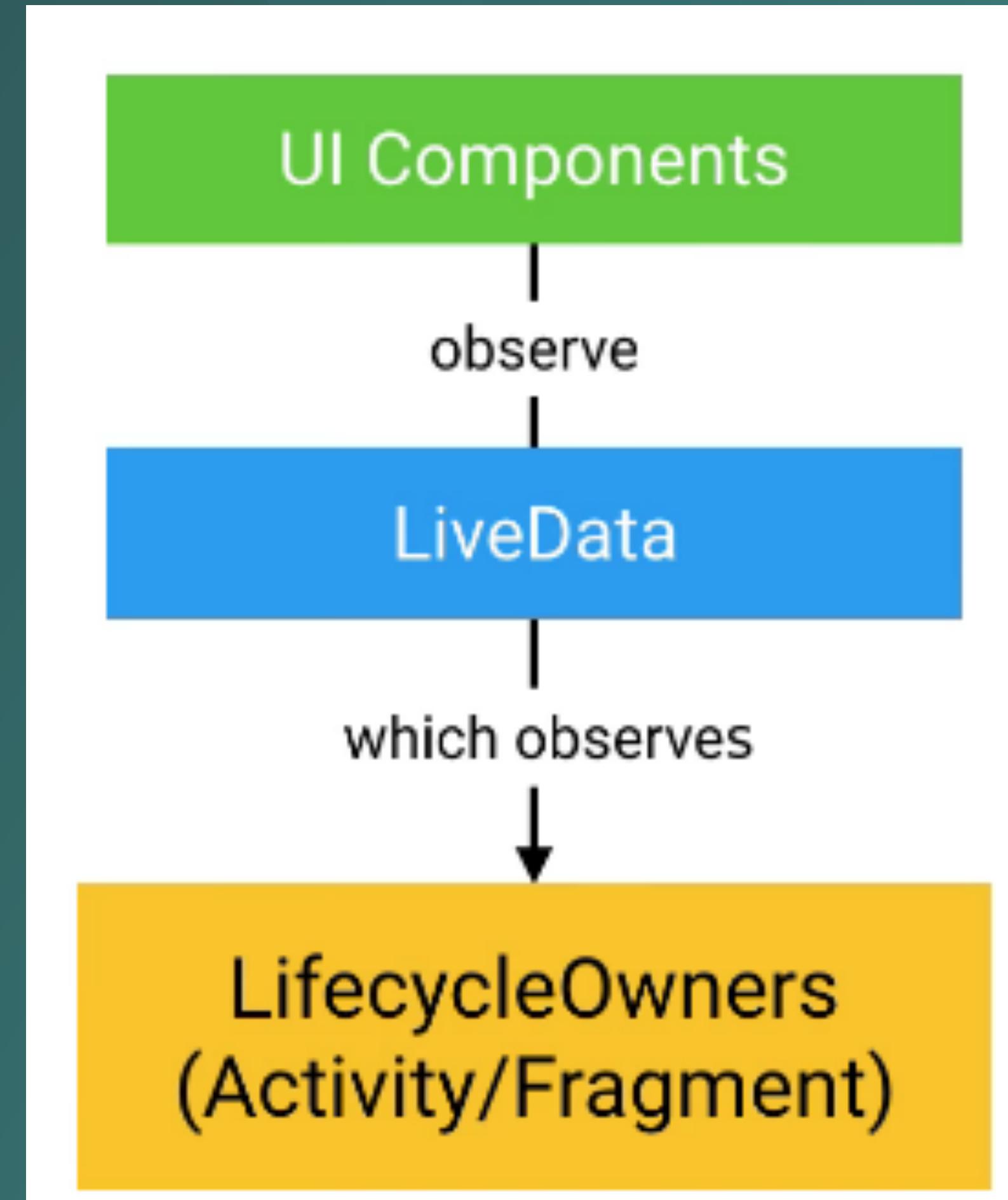
# ViewModel

```
class MyViewModel : ViewModel() {  
    private val users:  
        MutableLiveData<List<User>> by lazy {  
            loadUsers()  
        }  
    fun getUsers(): LiveData<List<User>> {  
        return users  
    }  
    private fun loadUsers() {  
        // Do an asynchronous  
        // operation to fetch users.  
    }  
}
```



# LiveData

- Ensures your UI matches your data state (Follows the observer pattern).
- No memory leaks (Observers are bound to Lifecycle).
- No crashes due to stopped activities (Inactive when the activity is in back stack).
- No more manual lifecycle handling (Observers are bound to Lifecycle).
- Always up to date data (Receives the latest data upon becoming active).
- Proper configuration changes (Immediately receives the latest available data).
- Sharing resources (Can be shared in your app).



# LiveData

```
class StockLiveData(symbol: String) : LiveData<BigDecimal>() {  
    private val mStockManager = StockManager(symbol)  
    override fun onActivityCreated(savedInstanceState: Bundle?) {  
        super.onActivityCreated(savedInstanceState)  
        val myPriceListener: LiveData<BigDecimal> = StockLiveData(symbol)  
        myPriceListener.observe(this, Observer<BigDecimal> {  
            price ->  
            mStockManager.requestPriceUpdates(myPriceListener)  
            // Update the UI.  
        })  
        override fun onInactive() {  
            mStockManager.removeUpdates(myPriceListener)  
        }  
    }  
}
```

# Share Data Between Fragments

```
class SharedViewModel : ViewModel() {  
    val selected = MutableLiveData<Item>()  
  
    fun select(item: Item) {  
        selected.value = item  
    }  
}
```

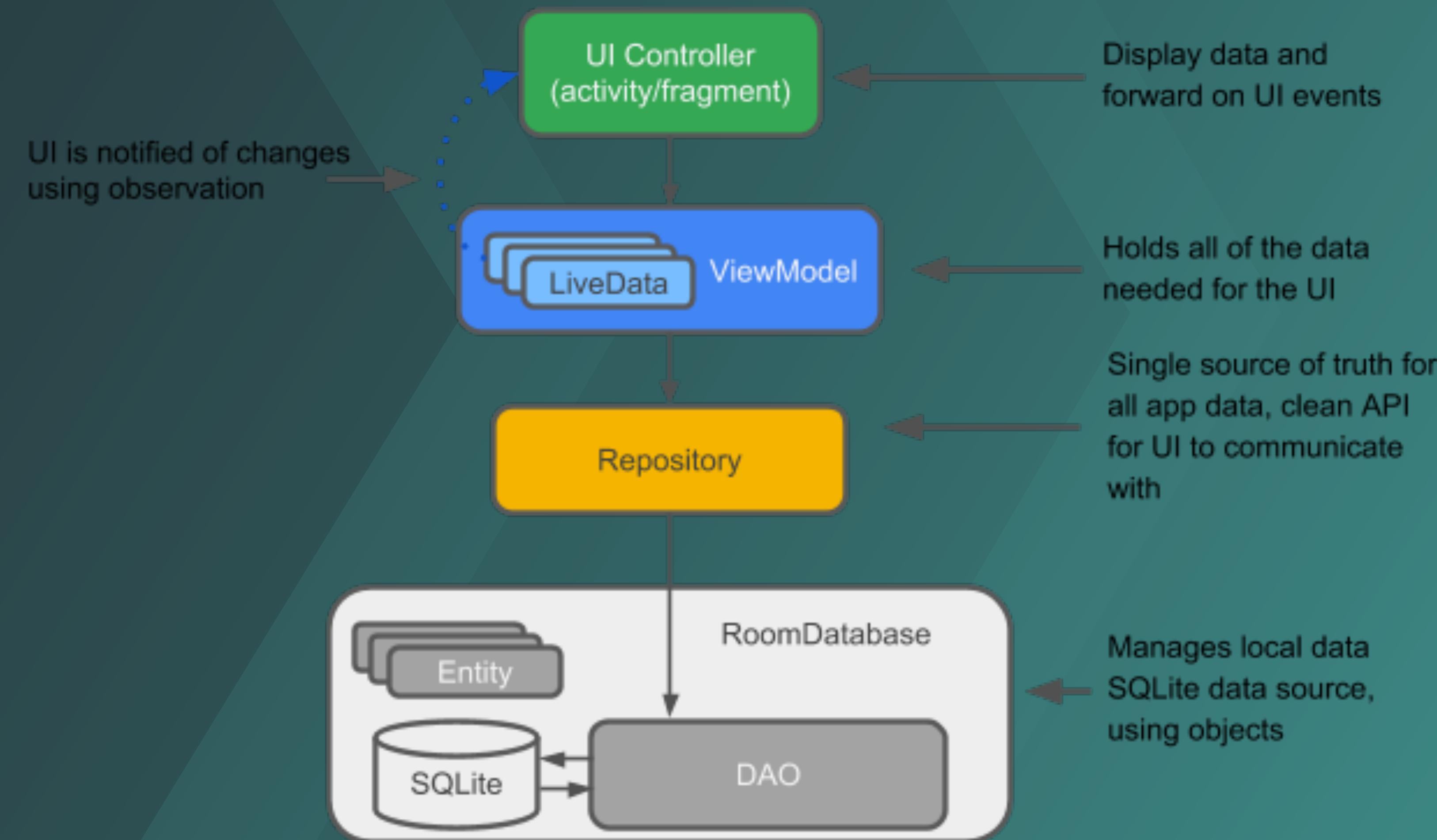
# Share Data Between Fragments

```
class MasterFragment : Fragment() {  
    private lateinit var itemSelector: Selector  
    private lateinit var model: SharedViewModel  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        model = activity?.run {  
            ViewModelProviders.of(this)[SharedViewModel::class.java]  
        } ?: throw Exception("Invalid Activity")  
        itemSelector.setOnClickListener { item ->  
            // Update the UI  
        }  
    }  
}
```

# Share Data Between Fragments

```
class DetailFragment : Fragment() {  
    private lateinit var model: SharedViewModel  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        model = activity?.run {  
            ViewModelProviders.of(this)[SharedViewModel::class.java]  
        } ?: throw Exception("Invalid Activity")  
        model.selected.observe(this, Observer<Item> { item ->  
            // Update the UI  
        })  
    }  
}
```

# Room



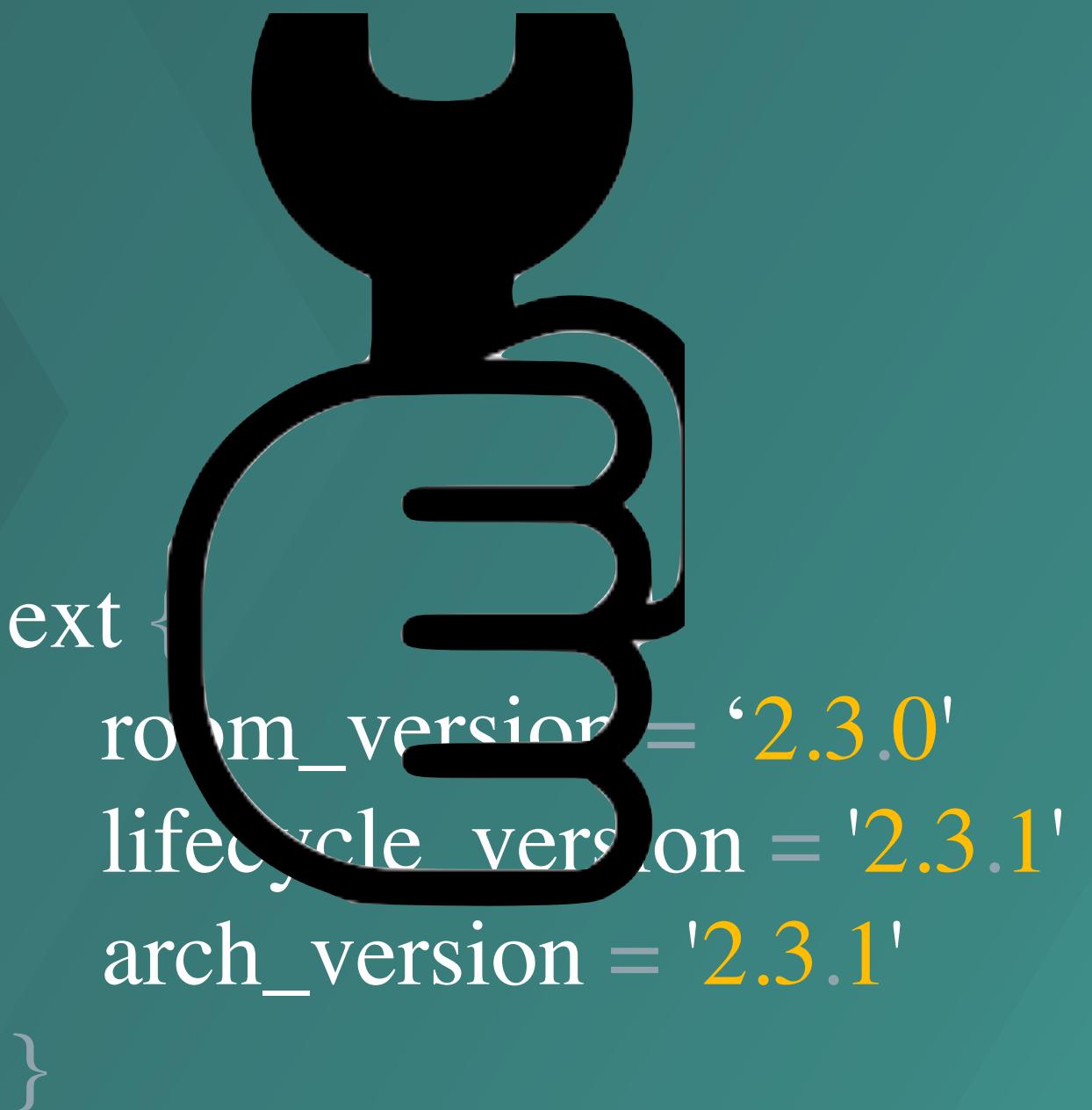
# Installation

In module level build.gradle:

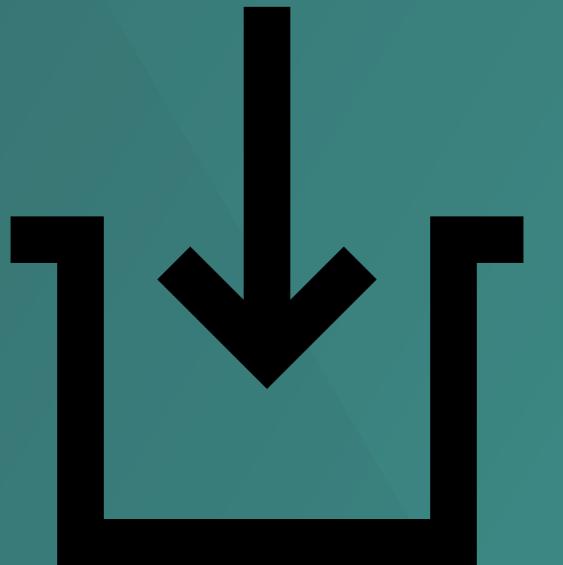
```
apply plugin: 'kotlin-kapt'
```

```
// Room components
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
// optional - Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
// optional - Test helpers
testImplementation "androidx.room:room-testing:$room_version"

// ViewModel
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
// LiveData
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
// Lifecycles only (without ViewModel or LiveData)
implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle_version"
// Annotation processor
kapt "androidx.lifecycle:lifecycle-compiler:$lifecycle_version"
// optional - Test helpers for LiveData
testImplementation "androidx.arch.core:core-testing:$arch_version"
```



# Room - Create the entity



```
@Entity(tableName = "word_table")
@Dao
class Word(@PrimaryKey
interface WordDao { @ColumnInfo(name = "word")
    @get:Query("SELECT * FROM word_table ORDER BY word ASC")
    val alphabetizedWords: LiveData<List<Word>>
}
```



```
@Insert
fun insert(word: Word)
```



```
@Query("DELETE FROM word_table")
fun deleteAll()
}
```

# Room - Database

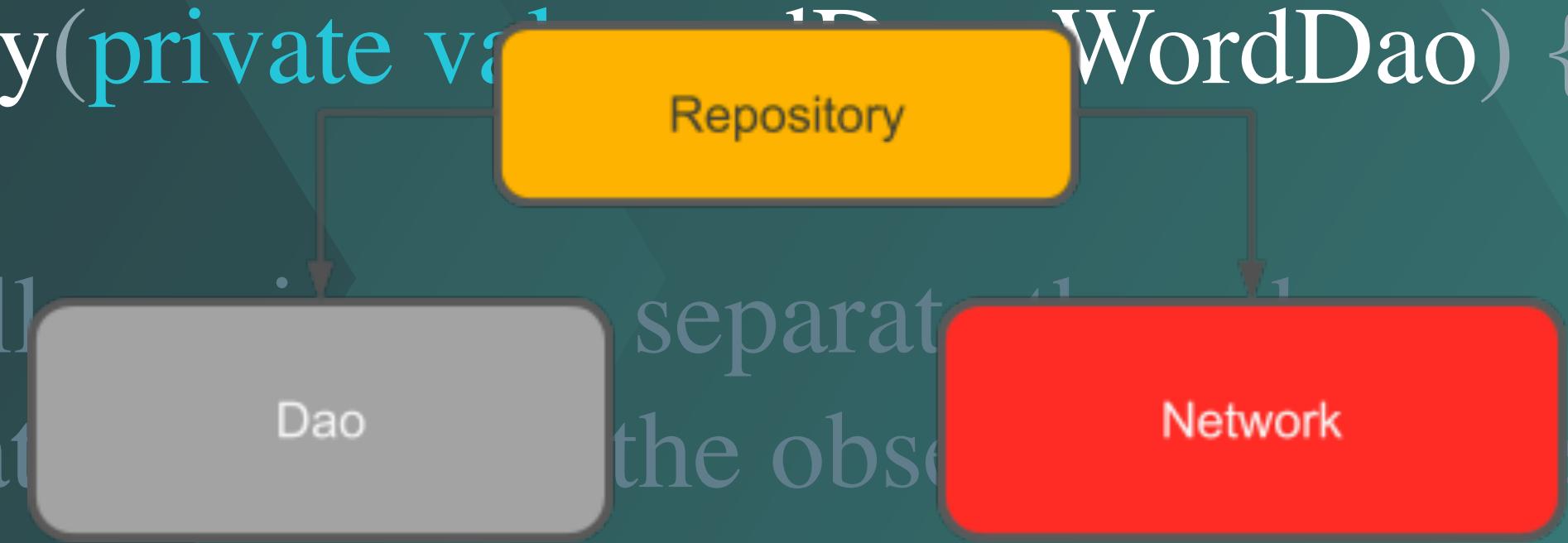
```
@Database(entities = [Word::class], version = 1)
abstract class WordRoomDatabase : RoomDatabase() {
    abstract fun wordDao(): WordDao
}
```

```
@Database(entities = [Word::class], version = 1)
abstract class WordRoomDatabase : RoomDatabase() {
    abstract fun wordDao(): WordDao
    companion object {
        private var INSTANCE: WordRoomDatabase? = null
        fun getInstance(context: Context): WordRoomDatabase? {
            if (INSTANCE == null) {
                synchronized(WordRoomDatabase::class.java) {
                    INSTANCE = Room.databaseBuilder(context.applicationContext,
                        WordRoomDatabase::class.java, "word_database")
                        .fallbackToDestructiveMigration()
                        .addCallback(sRoomDatabaseCallback)
                        .build()
                }
            }
            return INSTANCE
        }
    }
}
```

# Repository

Cache/LocalDB

```
// Declares the DAO as a private property in the constructor. Pass in the DAO  
// instead of the whole database, because we only need access to the DAO  
class WordRepository(private val wordDao: WordDao) {
```



```
// Room executes all queries on a separate thread.  
// Observed LiveData will automatically update itself  
// whenever the observed data has changed.  
val allWords: LiveData<List<Word>> = wordDao.getAllWords()
```

```
suspend fun insert(word: Word) {  
    wordDao.insert(word)  
}  
}
```

# Use in ViewModel

DEMO

```
class WordViewModel(application: Application) : AndroidViewModel(application) {  
  
    private val repository: WordRepository  
    val allWords: LiveData<List<Word>>  
  
    init {  
        val wordsDao = WordRoomDatabase.getDatabase(application).wordDao()  
        repository = WordRepository(wordsDao)  
        allWords = repository.allWords  
    }  
  
    fun insert(word: Word) = viewModelScope.launch {  
        repository.insert(word)  
    }  
}
```

# SQLDelight

Generates typesafe kotlin APIs from your SQL statements!

- Verifies schema, statements, and migrations at compile time.
- Provides IDE support like autocomplete and refactoring.



# Setup



```
buildscript {  
    repositories {  
        google()  
        mavenCentral()  
    }  
    dependencies {  
        classpath 'com.squareup.sqlDelight:gradle-plugin:1.5.0'  
    }  
}  
  
apply plugin: 'com.squareup.sqlDelight'  
  
dependencies {  
    implementation "com.squareup.sqlDelight:android-driver:1.5.0"  
}
```



# SQL Files

-- src/main/sqlodelight/com/example/sqlodelight/chess/data/Player.sq



```
CREATE TABLE chessPlayer (
    player_number INTEGER NOT NULL,
    full_name TEXT NOT NULL,
    quotes TEXT NOT NULL
);
```

```
CREATE INDEX chessPlayer_full_name ON chessPlayer(full_name);
```

```
INSERT INTO chessPlayer (player_number, full_name, quotes)
VALUES (15, 'Mikhail Tal',
    'You must take your opponent into a deep dark forest where 2+2=5,
    and the path leading out is only wide enough for one!');
```



# SQL Files

selectAll:

```
SELECT * FROM chessPlayer;
```

insert:

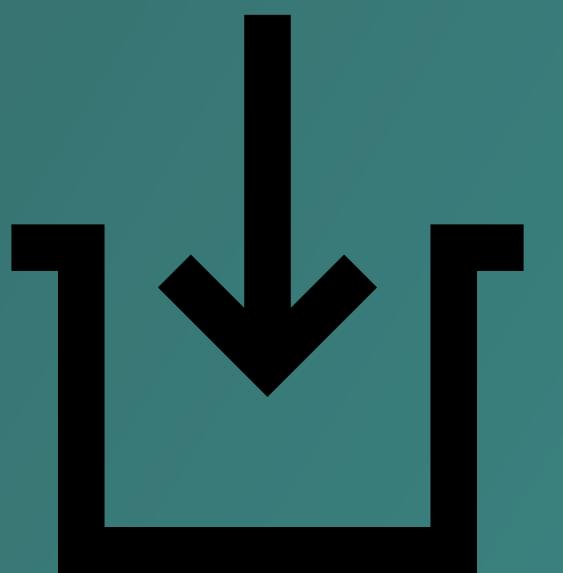
```
INSERT INTO chessPlayer(player_number, full_name)  
VALUES (?, ?);
```

update:

```
INSERT OR UPDATE INTO chessPlayer(player_number,  
full_name) VALUES (?, ?);
```

delete:

```
DELETE FROM chessPlayer WHERE player_number = ?;
```



VectorStock® VectorStock.com/F45ME7P

# Usage

DEMO

```
// In reality the database and driver above should be created a single time  
// and passed around using your favourite dependency injection/service  
// locator/singleton pattern.
```

```
val database = Database(driver)
```

```
val playerQueries: PlayerQueries = database.playerQueries
```

```
println(playerQueries.selectAll().executeAsList())  
// Prints [ChessPlayer(15, "Mikhail Tal")]
```

```
playerQueries.insert(player_number = 10, full_name = "Bobby Fischer")  
println(playerQueries.selectAll().executeAsList())  
// Prints [ChessPlayer(15, "Mikhail Tal"), HockeyPlayer(10, "Bobby Fischer")]
```



# Lecture outcomes

- Understand the old SQLite workflow
- Implement the CRUD operations
- When changing multiple entities, use transactions
- Migrate the local db from one version to another
- Use Room, ViewModel and LiveData

