
## Лекция #40. Работа с SQLite базами данных

У разработчиков **Android** есть несколько вариантов управления и хранения данных:

1. Файлы: для сохранения необработанных данных на устройстве.

2. **SharedPreferences**: для сохранения небольших наборов данных в виде пар ключ-значение.

3. **SQLite**: идеально подходит для сохранения структурированных данных в частной базе данных.

4. **Ресурсы/Ассеты**: используются для хранения статических файлов, обычно связанных с элементами пользовательского интерфейса, такими как строки, макеты или изображения.

**SQLite** — это реляционная база данных, в которой данные сохраняются в таблицах, которые могут ссылаться друг на друга. Эта база данных встроена в **Android SDK** и использует **SQL** для управления данными. Несмотря на то, что он мощный, его не всегда легко использовать, поскольку требуется знание SQL-запросов и тщательное управление, чтобы избежать определенных ошибок, таких как атаки с использованием SQL-инъекций.

Чтобы упростить задачу, на **Google I/O 2017** была представлена библиотека **Room**. **Room** — это уровень абстракции над **SQLite**, который упрощает доступ к базе данных, сохраняя при этом все функции **SQLite**. Такое упрощение избавляет разработчиков от написания большей части сложного кода, необходимого для преобразования SQL-запросов в объекты данных и обратно.

Используя **Room**, разработчикам достаточно определить схемы данных, описать, как с ними взаимодействовать, используя специальные аннотации:

- @Database - аннотация для объявления базы данных.
- @Entity - аннотация для объявления сущности базы данных.
- @Dao - аннотация для объявления интерфейса, который будет заниматься манипулированием данными базы данных.
- @PrimaryKey - аннотация для объявления первичного ключа сущности.
- @ColumnInfo - аннотация для настроек конкретного столбца сущности.
- @Query - аннотация, которая позволяет выполнить SQL-запрос в методах DAO-интерфейса.
- @Insert - аннотация, которая позволяет выполнить вставку в таблицу базы данных.
- @Update - аннотация, которая позволяет выполнить обновление некоторых строк в таблице базы данных.
- @Delete - аннотация, которая позволяет выполнить удаление некоторых строк в таблице базы данных.
- @Transaction - аннотация, которая помечает метод в **DAO**-интерфейсе как транзакция.

Это упрощает сопоставление объектов со строками базы данных. **Room** предлагает более читаемый и надежный способ доступа к базам данных, что приводит к более чистому и понятному коду.

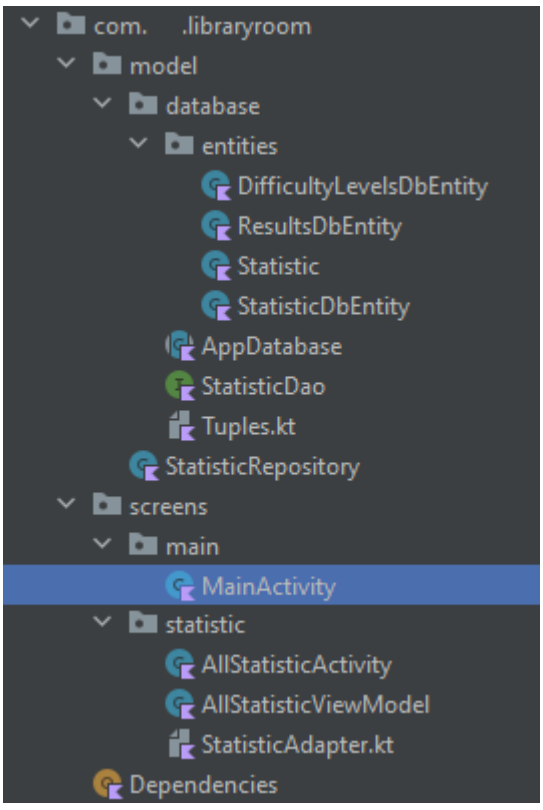
Также следует отметить, что существуют специальные **Tuple**-классы, которые никак не помечаются, но являются важной частью при разработке. Данные классы используются при взаимодействии с базой данных (например, когда нам необходимо получить какую-то часть данных из таблицы, а не все данные сразу). Более подробно **Tuple**-классы будут рассмотрены в практическом примере.

## Практическая часть

В качестве не сложного примера, создадим приложение, которое будет "имитировать" создание и отображение статистических данных какой-то игры (например, судоку). Приложение будет состоять из двух экранов: первый - заполнение и отправка данных в базу; второй - список со всеми данными из базы. Статистические данные будут состоять из следующих компонентов: результат игры (победа / поражение), уровень сложности (легкая, сложная и т.д.), количество ошибок, количество набранных очков.

Важное уточнение. В данной лекции не будут приведены листинги кода с верстокой **xml**-файлов и всех классов приложения.

При работе над заданием у вас должна получиться примерно следующая структура проекта:




В первую очередь необходимо указать все зависимости, которые будут использованы приложением. Для этого в файл сборки **build.gradle** нашего приложения:

```
dependencies {

    ...

    implementation 'androidx.room:room-runtime:2.5.0' // Библиотека "Room"
    kapt "androidx.room:room-compiler:2.5.0" // Кодогенератор
    implementation 'androidx.room:room-ktx:2.5.0' // Дополнительно для Kotlin Coroutines, Kotlin Flows
}
```

```
plugins {
    ...
    id 'kotlin-kapt' // !!!
}
```

```
android {
    ...

    defaultConfig {
        ...

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

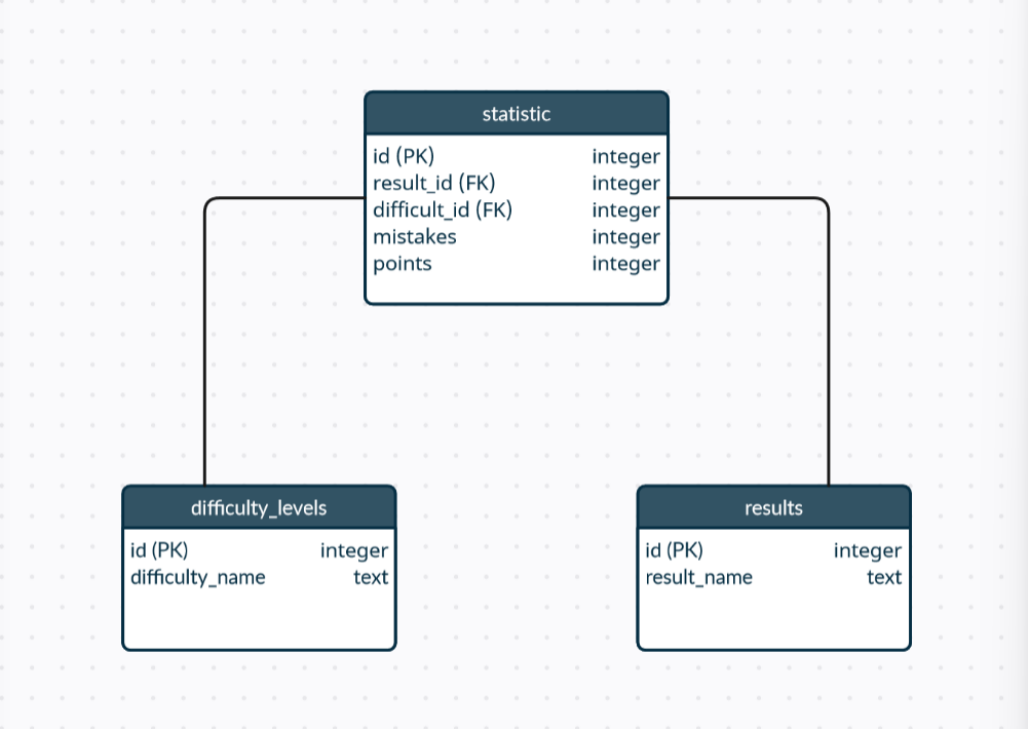
Данные блоки кода подключают нужные библиотеки (первый блок), добавляют нужный плагин (второй блок) и указывают нужный путь к каталогу, который будет хранить схему нашей базы данных (третий блок).

Перед тем как перейти к описанию сущностей, необходимо представить как база данных будет выглядеть, из каких таблиц она будет состоять. В данном практическом примере база данных будет состоять из трех таблиц:

- таблица "difficulty_levels", в которой будут хранится все доступные уровни сложности;
- таблица "results", в которой будут хранится все доступные результаты игры;

- таблица "statistic", в которой будут хранится все статистические данные.

Схематично база данных будет выглядеть следующим образом:




После того, как была разобрана схема базы данных, необходимо перейти к созданию сущностей. Создадим data-class DifficultyLevelsDbEntity, который будет описывать таблицу "difficulty_levels":

```
@Entity(tableName = "difficulty_levels")
data class DifficultyLevelsDbEntity(
    @PrimaryKey val id: Long,
    @ColumnInfo(name = "difficulty_name") val difficultyName: String
)
```

В данном случае мы пометили класс аннотацией @Entity, в которой переопределили свойство tableName - данное свойство задаёт имя таблицы. В случае, если бы свойство не было бы определенно, то таблица назвалась аналогично названию класса, т.е. DifficultyLevelsDbEntity.

Также поля класса были помечены аннотациями @PrimaryKey и @ColumnInfo. Первая аннотация помечает поле класса, как первичный ключ, а вторая задаёт название столбца отличное от названии переменной. У

аннотации @ColumnInfo есть и другие свойства, например значение по умолчанию, более подробно про свойства данной аннотации можно узнать здесь.

Аналогично создадим класс ResultsDbEntity:

```
@Entity(tableName = "results")
data class ResultsDbEntity(
    @PrimaryKey val id: Long,
    @ColumnInfo(name = "result_name") val resultName: String
)
```

Теперь перейдем к созданию более сложной сущности - StatisticDbEntity:

```
@Entity(
    tableName = "statistic",
    indices = [Index("id")],
    foreignKeys = [
        ForeignKey(
            entity = ResultsDbEntity::class,
            parentColumns = ["id"],
            childColumns = ["result_id"]
        ),
        ForeignKey(
            entity = DifficultyLevelsDbEntity::class,
            parentColumns = ["id"],
            childColumns = ["difficult_id"]
        )
    ]
)
data class StatisticDbEntity(
    @PrimaryKey(autoGenerate = true) val id: Long,
    @ColumnInfo(name = "result_id") val resultId: Long,
    @ColumnInfo(name = "difficult_id") val difficultId: Long,
    val mistakes: Long,
    val points: Long
)
```

Аннотации в свойствах data-class'a очень похожи на те, что были созданы ранее. Единственное отличие - в @PrimaryKey было определено свойство

autoGenerate = true. Данное свойство "объясняет" библиотеке, что при вставке нового объекта в таблицу, необходимо сгенерировать индекс самостоятельно. Например, в таблице находится пять элементов, при вставке нового элемента поле id автоматически станет равно шести.

Также в аннотации @Entity было определенно больше свойств. Свойство indices "объясняет" библиотеки по каком полю производить индексацию, в данном случае - id.

Свойство foreignKeys объявляет составные ключи. В данном случае составных ключа два - result_id и difficult_id. В объекте ForeignKey указываются сущность-родитель (entity), столбец-родитель (parentColumns) и столбец-ребенок (childColumns).

После того как были созданы все сущности базы данных, создадим интерфейс, помеченный аннотацией @Dao. Данный интерфейс будет взаимодействовать с базой данных с помощью специальных методов. Пока оставим данный интерфейс пустым, к его реализации следует вернуться чуть позже:

```
@Dao
interface StatisticDao {

}
```

Когда сущности были созданы, dao-интерфейс объявлен необходимо создать абстрактный класс AppDatabase, который будет описывать базу данных:

```
@Database(
    version = 1,
    entities = [
        DifficultyLevelsDbEntity::class,
        ResultsDbEntity::class,
        StatisticDbEntity::class
    ]
)
abstract class AppDatabase : RoomDatabase() {

    abstract fun getStatisticDao(): StatisticDao
```

```
}
```

Данный класс помечен аннотацией @Database, в которой необходимо обязательно описать два свойства: entities и version. Первое свойство принимает все сущности, которые были описаны выше, а второе свойство задаёт версию базы данных.

Версия базы данных используется для контроля базы данных, ее данных и т.д. Зачастую это используется при миграции базы данных с одной версии на другую, но это уже совсем другая история...

В самом классе создан абстрактный метод, который возвращает dao-интерфейс.

Так же создадим tuple-класс StatisticInfoTuple, который будет использоваться при "вытягивании" статистических данных из таблицы. Данный класс очень похож на StatisticDbEntity, но поля, которые хранят результат и уровень сложности, уже имеют тип String, так как там будут хранится значения результата и уровня сложности.

```
data class StatisticInfoTuple(
    val id: Long,
    @ColumnInfo(name = "result_name") val result: String,
    @ColumnInfo(name = "difficulty_name") val difficult: String,
    val mistakes: Long,
    val points: Long
)
```

После всех проделанных действий необходимо перейти к реализации dao-интерфейса. В данном интерфейсе создадим три метода, которые будут вставлять новые статистические данные, удалять данные по уникальному значению (id) и получать список всех данных из таблицы statistic:

```
@Dao
interface StatisticDao {

    @Insert(entity = StatisticDbEntity::class)
    fun insertNewStatisticData(statistic: StatisticDbEntity)
```

```
    @Query("SELECT statistic.id, result_name, difficulty_name, mistakes, points FROM statistic\n" +
            "INNER JOIN results ON statistic.result_id = results.id\n" +
            "INNER JOIN difficulty_levels ON statistic.difficult_id = difficulty_levels.id;")
    fun getAllStatisticData(): List<StatisticInfoTuple>

    @Query("DELETE FROM statistic WHERE id = :statisticId")
    fun deleteStatisticDataById(statisticId: Long)
}
```

Метод insertNewStatisticData принимает объект класса StatisticDbEntitty - объект, который необходимо вставить. Также данный метод помечен аннотацией @Insert, в которой определено свойство entity, благодаря которому происходит вставка в нужную таблицу.

Методы getAllStatisticData и deleteStatisticDataById помечены аннотацией @Query, которая принимает строку с SQL-запросом. Именно благодаря данному запросу выполняется получение всех элементов или удаление какого-то конкретного элемента.

И последнее, что необходимо сделать с базой данной - создать ее. Для этого выполним следующее:

```
object Dependencies {

    private lateinit var applicationContext: Context

    fun init(context: Context) {
        applicationContext = context
    }

    private val appDatabase: AppDatabase by lazy {
        Room.databaseBuilder(applicationContext, AppDatabase::class.java, "database.db")
            .createFromAsset("room_article.db")
            .build()
    }

    val statisticRepository: StatisticRepository by lazy { StatisticRepository(appDatabase.getStatisticDao()) }
}
```

В данном примере создается база данных appDatabase с помощью специального билдера: Room.databaseBuilder, который принимает контекст, класс, содержащий описание нашей базы данных, и название самой базы.

Так же у билдера вызван метод createFromAssets, данный метод заполняет базу данных приготовленными значениями. Т.е., если необходимо, чтобы при инициализации база данных хранила в себе какие-либо значения (например, уровни сложности и доступные результаты), нужно создать отдельно базу данных с помощью сторонних программ, таких как DB Browser for SQLite, заполнить ее и сохранить ее в папку assets приложения.

После того, как все манипуляции с базой данных были реализованы, необходимо создать data-класс Statistic, который будет использоваться во всем приложении. В данном классе создан метод toStatisticDbEntity, конвертирующий данный класс в сущность:

```
data class Statistic(
    val resultId: Long,
    val difficultId: Long,
    val mistakes: Long,
    val points: Long
) {

    fun toStatisticDbEntity(): StatisticDbEntity = StatisticDbEntity(
        id = 0,
        resultId = resultId,
        difficultId = difficultId,
        mistakes = mistakes,
        points = points
    )
}
```

Теперь необходимо создать репозиторий, который будет обращаться к dao-интерфейсу и манипулировать данными базы данных:

```
class StatisticRepository(private val statisticDao: StatisticDao) {

    suspend fun insertNewStatisticData(statisticDbEntity: StatisticDbEntity) {
        withContext(Dispatchers.IO) {
```

```
            statisticDao.insertNewStatisticData(statisticDbEntity)
        }
    }

    suspend fun getAllStatisticData(): List<StatisticInfoTuple> {
        return withContext(Dispatchers.IO) {
            return@withContext statisticDao.getAllStatisticData()
        }
    }

    suspend fun removeStatisticDataById(id: Long) {
        withContext(Dispatchers.IO) {
            statisticDao.deleteStatisticDataById(id)
        }
    }
}
```

В данном репозитории три метода, которые вставляют новые данные, получают всю статистику и удаляют какой-то элемент по индетификатору. Все эти методы являются suspend-функциями, т.к. будут вызываться из корутин. Так же следует изменить Dispatcher (withContext), т.к. обращаться к базе данных из основного потока нельзя.

Все эти методы вызываются в **корутинах**, запущеных в **MainActivity**, например вставка нового значения:

```
fun insertNewStatisticDataInDatabase(mistakes: Long, points: Long) {
    viewModelScope.launch {
        val newStatistic = Statistic(currentResult, currentDifficultyLevel, mistakes, points)
        statisticRepository.insertNewStatisticData(newStatistic.toStatisticDbEntity())
    }
}
```