05 - Room

Adam Terlo

Материалы

github.com/adamxrvn/hse-lyceum-android-course

Почему Room?

Room — это часть Android Architecture Components, предоставляющая абстракцию над SQLite для более удобной и безопасной работы с базой данных в приложении. Это помогает избежать шаблонного кода и облегчает работу с базой данных на устройствах Android.

Установка

Чтобы использовать Room в приложении, добавьте следующие зависимости в файл `build.gradle` приложения:

```
val room_version = "2.6.1"
implementation("androidx.room:room-runtime:$room_version")
kapt "androidx.room:room-compiler:$room_version"
implementation("androidx.room:room-ktx:$room_version")
```

Ключевые Компоненты Room

1. **Entity:**
Entity представляет собой таблицу в базе данных. Каждый экземпляр Entity — это строка в таблице.

2. **DAO (Data Access Object):**
DAO определяет методы для доступа к данным в базе данных, такие как вставка, удаление и запросы. DAO обеспечивает абстракцию от непосредственных операций с базой данных. Пример: Методы для получения всех клиента банка, добавленого клиента или удаления счёта.

3. **Database:**
Database содержит базу данных и служит контейнером для хранения Entities и DAO. Она подключает приложение к базе данных и управляет версиями схемы.

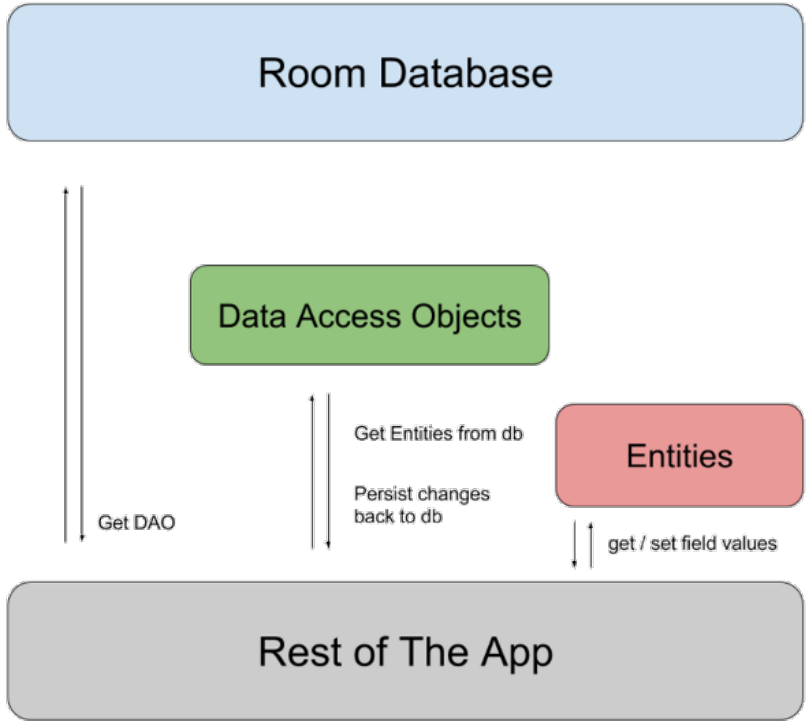


Рис 1. Схема архитектуры Room

Создание Entity

```
@Entity(tableName = "clients")
data class Client(
    @PrimaryKey(autoGenerate = true)
    val clientId: Int? = null,
    val name: String, val surname: String,
    val middleName: Int,
    val dateOfBirth: String,
    val email: String,
    val address: String,
    val status: String,
    val gender: String,
)
```

Создание Entity

```
@Entity(tableName = "accounts")
data class Account(
    @PrimaryKey(autoGenerate = true)
    val accountId: Int? = null,
    val accountNumber: String,
    val accountType: String,
    val balance: Float,
    val client: Client
)
```

ForeignKey

```
@Entity(
    tableName = "accounts",
    foreignKeys = arrayOf(
        ForeignKey(
            entity = Client::class,
            parentColumns = arrayOf("clientId"),
            childColumns = arrayOf("client"),
            onDelete = ForeignKey.CASCADE
        )
    )
)
data class Account(
    ...
)
```

Создание DAO

Создаем kotlin interface в проекте приложения.

```
@Dao
interface AccountDao {
    ...
}
```

Dao

SELECT (Выборка данных)

Используется для чтения данных из одной или нескольких таблиц.

Пример: получение списка клиентов с балансом выше определенного.

```
@Dao
interface AccountDao {

    @Query("SELECT * FROM accounts WHERE BALANCE > :balance")
    suspend fun getByBalance(balance: Float): Flow<List<Account>>
}
```

Dao

INSERT (Вставка данных)

Позволяет вставлять новые строки в таблицу.

Пример: регистрация нового клиента в системе.

```
@Dao
interface ClientDao {

    @Insert
    suspend fun insertClient(client: Client)

}
```

Dao

UPDATE (Обновление данных)

Модифицирует существующие строки в таблице.

```
@Dao
interface AccountDao {

    @Update
    suspend fun updateAccount(account: Account)

}
```

Dao

UPSERT

Если строка существует, то upsert обновит её. Иначе, добавит новую.

```
@Dao
interface AccountDao {

    @Upsert
    suspend fun upsertAccount(account: Account)

}
```

Создание DAO

DELETE (Удаление данных)

Удаляет строки из таблицы.

Пример: удаление информации о клиенте банка.

```
@Dao
interface AccountDao {
    @Delete
    suspend fun deleteAccount(account: Account)
}
```

Оператор ORDER BY

Оператор `ORDER BY` используется в SQL для сортировки результатов запроса по одному или нескольким столбцам, в порядке возрастания или убывания.

Пример: сортировка по одному столбцу

Сортировка списка клиентов по фамилии в алфавитном порядке:

```
@Dao
interface ClientDao {
    @Query("SELECT * FROM clients ORDER BY name")
    suspend fun getAllClientsSortedByName(): Flow<List<Client>>
}
```

Оператор ORDER BY

Пример: сортировка по нескольким столбцам

Сортировка списка клиентов сначала по фамилии, затем по имени:

```
@Dao
interface ClientDao {
    @Query("SELECT * FROM clients ORDER BY name ASC, surname ASC")
    suspend fun getAllClientsSortedByNameSurname(): Flow<List<Client>>
}
```

`ASC` означает сортировку по возрастанию, `DESC` — по убыванию. Если направление не указано, по умолчанию используется `ASC`.

Оператор LIMIT

Оператор `LIMIT` применяется в SQL для ограничения количества строк, возвращаемых запросом. Это особенно полезно при работе с большими объемами данных или когда нужно получить только первые несколько записей результата.

Пример: ограничение количества строк

Получение первых пяти клиентов из таблицы:

```
@Dao
interface ClientDao {
    @Query("SELECT * FROM clients ORDER BY name LIMIT :limit")
    suspend fun getByLimit(limit: Int): Flow<List<Client>>
}
```

Оператор LIMIT

Пример: ограничение с указанием смещения

Пропуск первых пяти записей и выбор следующих пяти:

```
@Dao
interface ClientDao {
    @Query("SELECT * FROM clients ORDER BY name LIMIT :limit OFFSET
:offset")
    suspend fun getByLimit(limit: Int, offset:Int): Flow<List<Client>>
}
```

`LIMIT` часто используется для пагинации результатов на веб-страницах или в приложениях, где необходимо показывать данные порциями.

Эти команды помогут контролировать объем данных, загружаемых из базы данных, что может существенно ускорить время загрузки и уменьшить нагрузку на сервер.

Агрегатные функции

AVG (Среднее значение)

Возвращает среднее значение числового столбца.

```
@Query("SELECT AVG(balance) FROM accounts")
suspend fun getAverageBalance(): Float
```

SUM (Сумма)

Возвращает сумму значений столбца.

```
@Query("SELECT SUM(balance) FROM accounts WHERE
accountType = :accountType")
suspend fun getSum(accountType: String): Float
```

Агрегатные функции

COUNT (Количество)

Возвращает количество строк, удовлетворяющих критерию.

```
@Query("SELECT COUNT(*) FROM accounts")
suspend fun getAccountsCount(): Int
```

MIN (Минимальное значение)

Возвращает наименьшее значение в столбце.

```
@Query("SELECT MIN(balance) FROM accounts")
    suspend fun getMinBalance(): Float
```

Агрегатные функции

MAX (Максимальное значение)

Возвращает наибольшее значение в столбце.

```
@Query("SELECT MAX(balance) FROM accounts")
suspend fun getMaxBalance(): Float
```

Оператор JOIN

`JOIN` используется для объединения строк из двух или более таблиц, основываясь на общем столбце между ними.

Типы JOIN

INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, etc.

Типы JOIN

INNER JOIN

Возвращает строки, когда есть совпадение в обеих таблицах.

```
@Query("SELECT * FROM clients INNER JOIN accounts
ON clients.clientId = accounts.client")
suspend fun getClientsWithAccounts():
Flow<List<Client>>
```

LEFT JOIN

Возвращает все строки из левой таблицы и совпадающие строки из правой таблицы. Если совпадений нет, результат будет содержать `NULL` на месте столбцов правой таблицы.

RIGHT JOIN

Аналогичен `LEFT JOIN`, но возвращает все строки из правой таблицы.

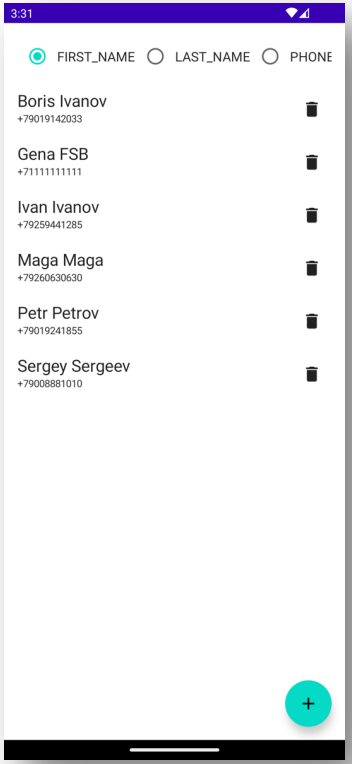
FULL OUTER JOIN

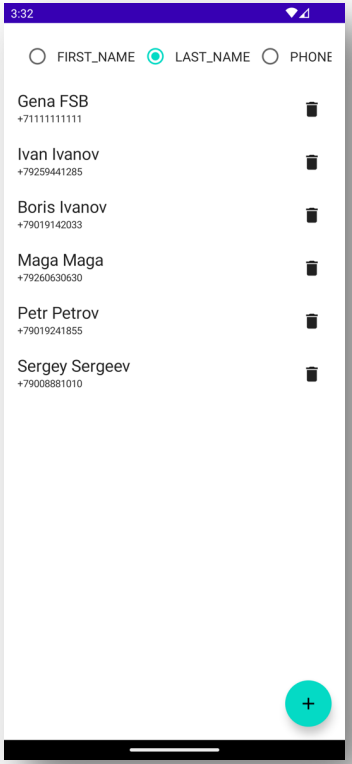
Возвращает строки, когда есть совпадение в одной из таблиц.

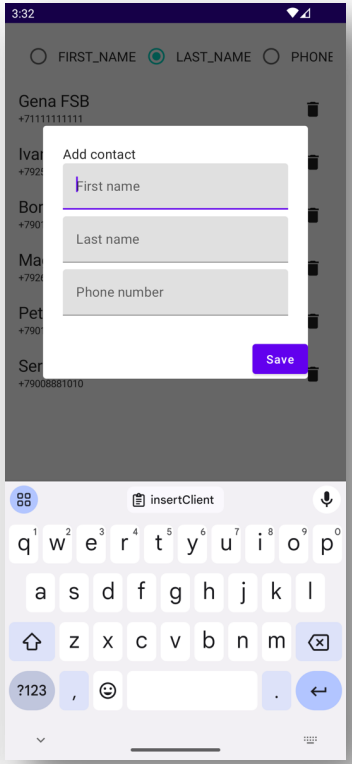
Создание Database

```
@Database(entities = [Client::class], version = 1)
abstract class ClientDatabase : RoomDatabase() {

    abstract val dao: ClientDao


}
```

Задание на сегодня







Оригинал [Philipplackner]