База данных SQLite

Если приложение должно сохранять данные между своими запусками, можно использовать базу данных SQLite – компактную встраиваемую СУБД, входящую в состав Android.

SQLite не является клиент-серверным приложением и не требует наличия сервера или сетевых соединений. Это уменьшает накладные расходы и упрощает работу с базой. Вся база данных хранится в обычном файле в каталоге приложения.

SQLite поддерживает синтаксис SQL и многие особенности SQL: таблицы, транзакции, представления, триггеры, вложенные запросы и т. д.

При этом SQLite является кроссплатформенной: есть сборки под все крупные операционные системы (Windows, Linux, Mac, Android, iOS). Огромное количество приложений используют SQLite. При этом исходный код SQLite не просто открыт, а выпущен в общественное достояние (public domain), то есть может использоваться кем угодно в любых целях без каких-либо атрибуций.

Доступ к базе

Для работы с SQLite используется специальный базовый класс `SQLiteOpenHelper`, который содержит основные методы для начала работы с базой. Приложение должно создать свой класс на его основе, переопределив два метода:

- `onCreate`, который вызывается в тех случаях, когда базы данных ещё нет. Приложение должно создать требуемые таблицы и, при необходимости, наполнить их начальными данными.
- `onUpgrade`, вызывается если версия открываемой базы данных старая. Приложение может внести требуемые изменения в структуру базы данных: добавить нужные столбцы, создать новые таблицы, дополнить словари и т. д. В качестве параметров метод принимает номера текущей и новой версий базы данных.

При инициализации класса нужно передать ему название базы и текущую версию, в примере ниже эти даные для удобства объявлены как константы:

```
class DbHelper(context: Context) :
      SQLiteOpenHelper(context, DB_NAME, null, DB_VERSION) {

    companion object {
        const val DB_NAME = "books.db"
        const val DB_VERSION = 1
    }

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL("CREATE TABLE IF NOT EXISTS devices(" +
```

```
                "id INTEGER PRIMARY KEY, " +
                "model TEXT, " +
                "price INT, " +
                "status INT)")
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // Апгрейд базы, если требуется
    }
}
```

В основной программе нужно создать экземпляр этого класса, а затем получить ссылку на объект базы данных – обычно для этого используется свойства `writeableDatabase`:

```
val helper = DbHelper(this)
val db = helper.writableDatabase
```

После завершения работы с базой следует вызвать метод `close()`, чтобы закрыть соединение.

SQL-инъекция

При выполнении операций с базой часто требуется встраивать в запрос данные, полученные от пользователя. Если пользователь хочет взломать приложение, он может особым образом сформировать данные, чтобы запрос выполнился не так, как было запланировано.

Например, приложение делает выборку заказа по номеру таким запросом:

```
"SELECT * FROM orders WHERE number=$number"
```

Тогда злонамеренный пользователь может вместо номера заказа написать что-то вроде:

```
1; DROP TABLE orders;
```

Если приложение не валидирует данные пользователя, то в результате этот текст будет встроен в запрос, и получится следующее:

```
"SELECT * FROM orders WHERE number=1; DROP TABLE orders;"
```

СУБД разобьёт эту строку на две, поскольку точка с запятой считается разделителем команд, затем выполнит первую команду – произведёт выборку данных `SELECT`, а потом и вторую – `DROP TABLE`, уничтожив, таким образом, всю таблицу с заказами.

Этот пример показывает насколько важно всегда производить проверку данных, получаемых от пользователя, чтобы проверять их валидность, ведь SQL-инъекции могут не только повреждать данные, но и приводить к их утечкам.

Чтобы избежать SQL-инъекции, нужно передавать значения, полученные от пользователя, отдельно от запроса. В самом запросе при этом вместо значений ставятся знаки вопроса:

```
"id=? AND age<?"
```

А значения в том же порядке передаются в массиве:

```
arrayOf(id, "30")
```

Операции с данными

Выборка данных

Выборка данных из базы производится методом `query`, который получает в качестве параметров имя таблицы и информацию об условиях выборки. Метод `query` может принимать следующие параметры:

- `table` – имя таблицы, это единственный обязательный параметр;
- `columns` – список столбцов;
- `selection` – условия выборки (`WHERE` в классическом SQL);
- `selectionArgs` – аргументы для условий выборки;
- `groupBy` – группировка (`GROUP BY`);
- `having` – использование условий для агрегатных функций (`HAVING`);
- `orderBy` – сортировка (`ORDER BY`);
- `limit` – максимальное количество строк в результате.

Если какие-либо параметры не требуются, то их можно опустить, заменив ключевым словом `null`.

В качестве результата метод возвращает объект `Cursor`, с помощью которого можно перемещаться по выбранным из таблицы строкам.

```
val c = db.query("users",              // Название таблицы
    arrayOf("id", "name", "email"),  // Список выбираемых столбцов
    "name LIKE ?",                   // Условие
    arrayOf("Иван"),                 // Подстановки в условие
    null, null, null, null)

c.moveToFirst()                      // Перемещение в начало списка
do {
    val id = c.getInt(0)             // Получение значения столбца по его номеру
    // ...
} while (c.moveToNext())             // Переход к следующей строке, если она есть
c.close()
```

Добавление данных

Вставка новых значений производится методом `insert`, сами значения должны быть заранее подготовлены:

```
val cv = ContentValues()
cv.put("name", "Иван Иванов")
cv.put("email", "ivan@ivanov.ru")
val rowID = db.insert("users", null, cv)
```

Метод возвращает идентификатор добавленной строки, если добавление произошло успешно, в противном случае возвращается -1.

Изменение данных

За изменение данных отвечает метод `update`, в качестве параметров он принимает имя таблицы, список значений, условие и аргументы условия:

```
val cv = ContentValues()
cv.put("name", "Петр")
cv.put("email", "petr@tpu.ru")
val updCount = db.update("users", cv, "id=?", arrayOf("5"))
```

Метод возвращает количество строк, которые были изменены.

Удаление данных

Удаление производится методом `delete`, в качестве параметров принимает имя таблицы, условия и аргументы условия:

```
db.delete("users", "id=?", arrayOf("5"))
```

Данные просто удаляются, без каких-либо дополнительных запросов или подтверждений – всё это может быть сделано заранее в интерфейсе приложения.

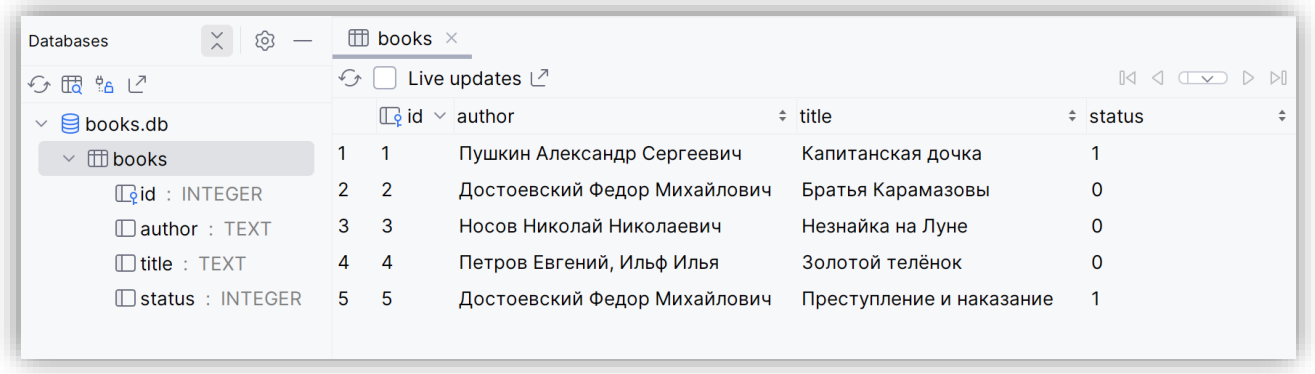
Прямое выполнение команд

При необходимости можно выполнить и обычный SQL-запрос:

```
db.execSQL("DELETE FROM devices WHERE id=5")
db.execSQL("DELETE FROM devices WHERE id=?", arrayOf("5"))
```

Просмотр баз данных

Для более удобной отладки работы с базами данных в Android Studio входит специальный инструмент Database Inspector, который можно вызвать с помощью меню View → Tool Windows → App Inspection. В появившейся панели есть несколько вкладок, и на владке Database Inspector можно посмотреть базы данных приложения, таблицы и сами данные, а также выполнить над ними какие-либо запросы:



Задание

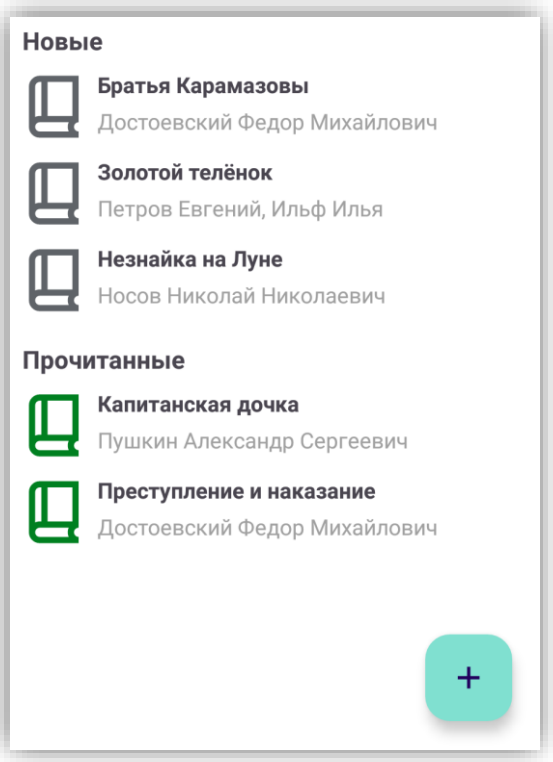
Разработайте приложение для ведения списка книг для чтения.

Для каждой книги должны быть определены автор и название, при желании можно добавить и другие атрибуты.

Книга может находиться в одном из двух статусов:

- Новая
- Прочитана

Приложение должно выводить список книг по секциям: сначала новые, затем прочитанные:



Приложение должно позволять добавлять и изменять книги, а также удалять их из списка. Для редактирования информации о книге можно использовать диалог или другую активность. При изменении статуса книги (новая/прочитана) она автоматически перемещается в соответствующую секцию списка.

Информация о книгах должна храниться в базе данных SQLite. Для доступа к базе можно использовать как методы, описанные в данной теме, так и библиотеку Room, по выбору.