SQLite. Кошкин дом. Часть первая.

Изучим SQLite с самых основ. Для начала мы научимся работать с SQLite без привязки к Android. Это позволит вам набить руку, лучше узнать её возможности. После этого будет легче интегрировать базу данных в Android.

Если вы знакомы с SQL, например, MySQL, то многое будет понятным. Я буду ориентироваться на новичков.

База данных нужна для долгого хранения большого количества данных, которые не пропадут после закрытия приложения. Кроме того, база данных на основе SQL позволяет производить различные манипуляции по выборе - найти самого толстого котика, найти самого молодого кота, найти только котов и только кошек и т.д.

Чтобы было легче понять принцип работы с базой данных, представьте себе следующую аналогию. Допустим на диске вы создали новую папку - это аналог базы данных. В новой папке вы можете создать несколько текстовых файлов с информацией (рецепты, дни рождения, список гостей) - это таблицы базы данных.

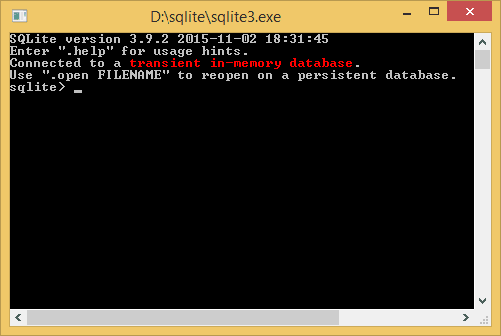
Таблица - это важная часть базы данных. Информация в таблицах упорядочена, чтобы её можно быстро найти. Таблица состоит из вертикальных столбцов (column) и горизонтальных рядов (row).

Мы построили новый пятизвёздочный отель с уникальным названием "Кошкин дом". Нам требуется учёт всех постояльцев гостиницы. Без базы данных не обойтись. Создадим базу данных **hotel** с таблицей **guests**, в которой будут столбцы **name** (имя), **city**(город), **gender** (пол), **age** (возраст).

sqlite3.exe

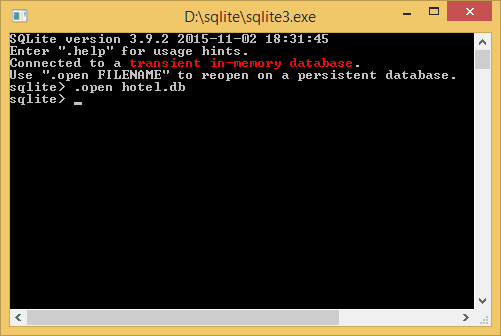
Давайте изучать SQLite. Начнём с командной строки под Windows. Хорошая новость - ничего не надо скачивать. Когда вы устанавливали инструменты разработки Android, то в папке **SDK\platform-tools** уже есть исполняемый файл **sqlite3.exe**. Для удобства я создал новую папку **sqlite** для опытов и скопировал туда этот файл.

Если сейчас запустить исполняемый файл, то увидим следующее окно.



В окне выводится подсказка, что для работы с базой данных нужно набрать команду **.open FILENAME**. Вводим:

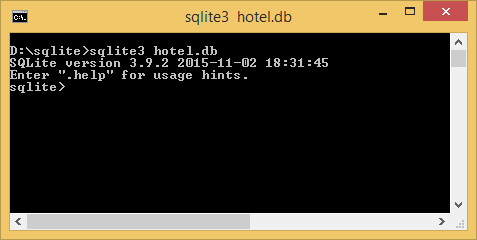
.open hotel.db



В папке рядом с **sqlite3.exe** появится новый файл **hotel.db**.

Далее в окне наберите команду .help для просмотра всех команд. Затем введите команду .quit, чтобы закрыть программу.

Теперь запустите отдельно командную строку и введите команду sqlite3 hotel.db.



Вы открыли базу данных **hotel** и программа выводит приглашение **sqlite>** для ввода специальных команд.

Прежде чем заполнять базу новыми данными, нужно ознакомиться с используемыми типами данных в SQLite:

* NULL - пустое значение
* INTEGER - целое число
* REAL - дробное число
* TEXT - строка
* BLOB - для изображений и бинарных файлов

Как видите, в SQLite не используются булевы типы. Поэтому для таких случаев используется тип **INTEGER** со значениями 0 для **false** и 1 для **true**.

Также для оптимизации используйте **INTEGER** вместо **TEXT** там, где это возможно. Например, для номерного фонда вы можете использовать значения 0, 1, 2, 3 вместо описания "одноместный", "двухместный", "люкс" и т.д.

Вернёмся к приглашению **sqlite>** и введём команду .tables. Так как у нас нет таблиц, то ничего не выводится. Запомним эту команду.

Создание таблицы

В SQL используется язык, очень похожий на разговорный английский. Так, для создания новой таблицы используется следующий синтаксис:

CREATE TABLE table\_name (column\_name\_1 data\_type\_1,

column\_name\_2 data\_type\_2,

...);

В переводе звучит так: СОЗДАТЬ ТАБЛИЦУ такую-то (первый\_столбец тип столбца, второй\_столбец тип столбца, ...).

Команды SQL принято писать заглавными буквами, хотя это не обязательно. Не путайте команды SQL с командами **sqlite3.exe**, которые начинаются с точки. Например, команду **.help** нужно вводить только в нижнем регистре.

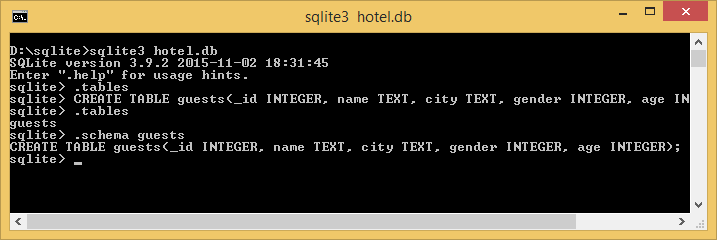
В нашем случае для создания таблицы, которая будет содержать информацию о гостях, потребуется ввести команду.

CREATE TABLE guests(\_id INTEGER, name TEXT, city TEXT, gender INTEGER, age INTEGER);

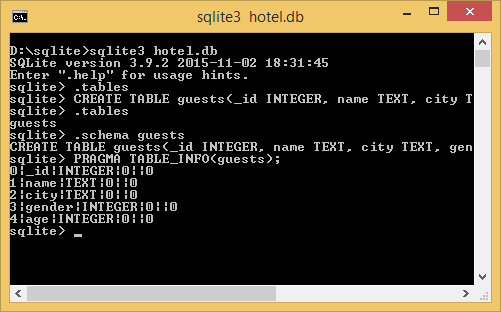
После ввода этой команды снова выполните команду **.tables**. Вы увидите созданную таблицу.



Теперь введите команду .schema, чтобы увидеть схему таблицы. Вы увидите собственную команду, которую вводили для создания таблицы.



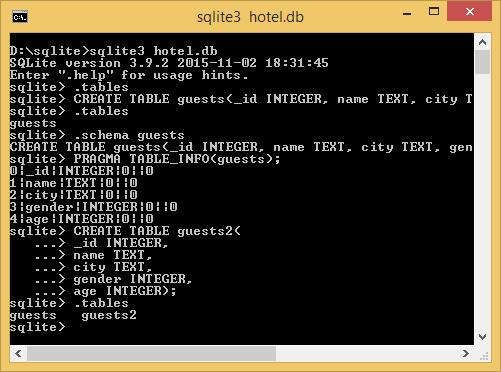
Познакомимся ещё с одной командой PRAGMA TABLE\_INFO(guests). Видим некоторое подобие таблицы с нулевыми значениями.



Если вы где-то сделали ошибку и хотите заново создать таблицу, то для удаления таблицы используется команда **DROP TABLE table\_name;**. При желании можно поставить проверочное условие **IF EXISTS**. Убедитесь, что таблицы больше нет командой .tables.

DROP TABLE IF EXISTS guests;

Небольшой совет - при вводе команд SQL принято записывать в столбик, отделяя параметры друг от друга. Когда вы находитесь в режиме ввода команд, то нажатие клавиши **Enter** приводит к переводу строки с приглашением вида **...>**. Продолжайте вводить команды в этом режиме. Когда вы закончите выражение точкой с запятой, то программа поймёт, что следующее нажатие **Enter**должно запустить команду. Для демонстрации я создал вторую таблицу **guests2**.



Небольшая подсказка: наши команды сохраняются в истории. Поэтому во время ввода вы можете нажать на клавишу "Стрелка Вверх" на клавиатуре, чтобы быстро получить доступ к длинной строке. Также можно нажать клавишу F7 - в этом случае появится диалоговое окно со списком предыдущих команд.

Экспорт данных

Можно экспортировать объекты базы данных в SQL-формате при помощи команды **.dump**. Без аргументов будет экспортирована вся база. Чтобы экспорт шел в файл, а не на экран (по умолчанию), то используйте команду **.output [filename]**. А чтобы восстановить вывод данных на экран, используйте команду **.output stdout**:

sqlite> .output guests.sql

sqlite> .dump

sqlite> .output stdout

Когда вы выполните эти команды, то у вас на диске появится файл **guests.sql** (если такой файл уже был, то он будет перезаписан).

Через командную строку:

sqlite3 hotel.db .dump

Дамп сохраняем в файле:

sqlite3 hotel.db .dump > guests.sql

Для создания новой базы данных hotel.db из нашего дампа guests.sql можно так:

sqlite3 hotel.db < guests.sql

Другой способ создания базы данных заключается в использовании опции *init*:

sqlite3 -init test.sql test2.db

В этом случае будет создана база, и мы войдем в оболочку программы. Можно добавить команду **.exit**, чтобы выйти из оболочки:

sqlite3 -init test.sql test2.db .exit

Импорт данных

Импортировать данные можно двумя способами. Если данные содержатся в SQL-формате, то можно воспользоваться командой **.read**. Если файл содержит данные в формате CSV, то используется команда **.import [file][table]**. Данные обычно разделяются вертикальной чертой |. Но можно использовать и другие разделители. Разделитель можно задать командой **.separator**, а увидеть используемый разделитель можно командой **.show**.

sqlite> .show

echo: off

eqp: off

explain: off

headers: on

mode: column

nullvalue: ""

output: stdout

colseparator: "|"

rowseparator: "\n"

stats: off

width:

В строчке **separator** вы можете видеть используемый символ.

Мы использовали для экспорта команду **.dump**, поэтому, чтобы импортировать данные из созданного файла **guests.sql**, нам подойдет команда **.read**.

.read guests.sql

SELECT \* FROM guests;

Экспорт и импорт части данных

Не всегда нужно сохранять всю базу, иногда нужно сохранить только её часть. Предположим, мы решили сохранить только те ряды таблицы, в которых имена котов заканчиваются на **ik**:

sqlite> .output ik.csv

sqlite> .separator ,

sqlite> SELECT \* FROM guests WHERE VALUE name '%ik';

sqlite> .output stdout

Теперь, если мы захотим импортировать сохраненные данные в такую же таблицу с схожей структурой, то делаем следующее:

sqlite> CREATE TABLE guests2(\_id integer PRIMARY KEY, name TEXT);

sqlite> .import ik.csv guests2

Настройка

Мы можем поменять вид приглашения, который по умолчанию имеет вид **sqlite>** через команду **.prompt**. Давайте поменяем на более знакомое и правильное:

sqlite> .prompt 'cat>'

cat>

Теперь вместо глупого приглашения используется любимое нами слово.

CRUD

Потренировавшись с созданием таблиц, перейдём к их наполнению. Существуют четыре базовых операции при работе с записями таблицы.

* **C**reate - создать новую запись
* **R**ead - прочитать запись
* **U**pdate - обновить запись
* **D**elete - удалить запись

По первым буквам операций создано сокращение CRUD для быстрого запоминания, что нужно реализовать программисту в своей программе.

Начнём с вставки новой записи в таблицу. Синтаксис команды.

INSERT INTO table\_name (column\_name\_1, column\_name\_2, ...)

VALUES (value\_1, value\_2, ...);

Переводим: ВСТАВИТЬ В такую-то таблицу (первый\_столбец, второй\_столбец, ...) Значения (первое\_значение, второе\_значение, ...)

Вставляем первую запись в таблицу.

INSERT INTO guests (\_id, name, city, gender, age) VALUES (1, "Васька", "Питер", 1, 6);

Итак, в гостинице поселился первый гость. При вводе команды важно соблюдать очерёдность столбцов и их значений. Например, можно было указать имена столбцов в обратном порядке, но тогда и значения следовало бы ввести также в обратном порядке. Но обычно стараются перечислять столбцы в том же порядке, как они создавались.

Существует укороченная запись без перечисления столбцов. В этом случае нужно указывать все значения и в том порядке, в котором создавались соответствующие столбцы.

INSERT INTO guests VALUES (1, "Васька", "Питер", 1, 6);

Проверить наличие записи можно с помощью команды:

SELECT \* from guests;

Звёздочка (\*) выводит все записи из указанной таблицы. Перед данной командой можно задать режим вывода записей через команду **.mode режим**. Доступны варианты: ascii, column, csv, html, insert, line, tabs, tcl.

Также удобно включить показ имён столбцов через команду **.header on**.

.header on .mode column SELECT \* FROM guests;



Вы можете теперь вводить новые записи, не забывая увеличивать значение идентификатора **\_id**. Но нет никакой гарантии, что однажды вы не ошибётесь и не введёте одинаковый идентификатор. Для таблицы базы данных это очень плохая ситуация, так как теряется принцип уникальности для записи.

Вторая возможная проблема - пропуск столбца. Например, возможна такая запись.

INSERT INTO guests (\_id, city, gender, age) VALUES (1, "Питер", 1, 6);

Мы пропустили столбец **name** и после вставки записи там будет пустое значение. Получается, что гость живёт в номере отеля без имени. Непорядок.

Для решения подобных проблем в SQL есть специальные ключевые слова: **PRIMARY KEY**, **AUTOINCREMENT**, **NOT NULL**, **DEFAULT value**.

**PRIMARY KEY** (первичный ключ) обеспечивает уникальность в таблице. В таблице может быть только один первичный ключ.

Первичный ключ - столбец таблицы, имеющий уникальное значение для каждой записи. Назначается при создании таблицы. Ключ не может содержать **NULL**, потому что теряется уникальность, ведь в других записях тоже может оказаться **NULL**. Значения первичного ключа должны оставаться неизменными.

Во многих случаях для этой цели создают новый столбец, который будет содержать уникальный номер. В Android столбец называют **\_id**.

Чтобы база данных сама заботилась об уникальности первичного ключа, можно добавить к нему ключевое слово **AUTOINCREMENT**, которое будет автоматически увеличивать значение на единицу при вставке новой записи.

**AUTOINCREMENT** (автоувеличение) автоматически вычисляет следующее значение ряда таблицы при добавлении. Удобно использовать у идентификаторов.

Поэтому скрипт создания таблицы должен иметь следующий вид.

CREATE TABLE guests(\_id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT,

city TEXT,

gender INTEGER,

age INTEGER);

Удалите таблицу **guests** с помощью команды DROP TABLE guest; и заново создайте таблицу с этим же именем.

Теперь для вставки новой записи вам не нужно указывать значение для первого столбца **\_id**, база данных сама сгенерирует нужное значение. Вставим новые записи.

INSERT INTO guests (name, city, gender, age) VALUES ("Васька", "Питер", 1, 6); INSERT INTO guests (name, city, gender, age) VALUES ("Мурзик", "Мурманск", 1, 4);

Если бы вы попытались использовать старый вариант с идентификатором, то получили бы сообщение об ошибке. Установленное нами правило теперь не позволяет вставлять записи с собственными идентификаторами. Мы передали эти полномочия базе данных.

**NOT NULL** не разрешает оставлять пустым определённый столбец. Если при вставке мы пропустим этот столбец, то снова увидим сообщение об ошибке.

**DEFAULT value** - если при вставке новой строки мы не зададим значения для столбца, то применится значение по умолчанию. Данный параметр можно комбинировать с предыдущим.

В этом случае мы можем пропустить столбец, но вместо ошибки база данных подставит значение по умолчанию.

Увы, но вам снова придётся удалить таблицу и воссоздать её по новому правилу.

CREATE TABLE guests(\_id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

city TEXT NOT NULL,

gender INTEGER NOT NULL DEFAULT 3,

age INTEGER NOT NULL DEFAULT 0);

Заново заселяем Ваську и Мурзика, команды не меняются. Допустим, к нам заехал необычный гость, у которого неудобно было спросить возраст и пол. Вводим только имя и город, остальное добавится автоматически по умолчанию.

INSERT INTO guests (name, city) VALUES ("Ктулху", "Москва");

Можно выводить не все записи из таблицы, а только нужные. Например, запись с идентификатором 1.

SELECT \* FROM guests WHERE \_id=1;

Если не нужны все столбцы, то перечисляем нужные через запятую вместо звёздочки.

SELECT name, city FROM guests WHERE \_id < 3;

Или такой вариант.

SELECT name, city FROM guests WHERE name !="Ктулху";

Чтобы не искать все записи, можно ограничить поиск условием **WHERE**, после которого идёт имя столбца и условие равенства. Показать всех котов, чей возраст меньше 15.

SELECT \* FROM guests WHERE age < 15;

Вместо звёздочки можно указать столбцы, которые вам нужны. Например, нам нужны только имена котов с этим же условием.

SELECT name FROM guests WHERE age < 15;

Столбцы указываются через запятую. Нам нужны имена и адреса котов с этим же условием.

SELECT name, city FROM guests WHERE age < 15;

Условие **WHERE** можно объединять с помощью ключевого слова **AND**. Список котов младше 15 лет и проживающих в Москве.

SELECT name, city FROM guests WHERE age < 15 AND city="Мурманск";

Также доступны слова **OR** (ИЛИ):

SELECT name, city FROM guests WHERE age < 15 OR city="Мурманск";

Слов **AND** и **OR** может быть несколько в одном запросе.

Проверку на **NULL** можно сделать с помощью ключевого слова **IS NULL** (если столбец таблицы создавался без **NOT NULL**)

SELECT \* FROM guests WHERE age IS NULL;

Ключевое слово **LIKE** позволяет сократить множество операторов **OR**. Например, мы хотим узнать имена котов, которые заканчиваются на "ик":

SELECT name FROM guests WHERE name LIKE '%ик';

Символ **%** в строке указывает на любое слово с нужным окончанием (представляет любое количество неизвестных символов).

Также можно использовать спецсимвол **\_** для одного символа. Ищем Ваську.

SELECT name FROM guests WHERE name LIKE '\_аська';

Стоит заметить, что в командной строке примеры могут не работать, так как там не используется кодировка UTF-8. Проверяйте на английских словах. В других программах или в Android такой проблемы не будет, там всегда используется правильная кодировка.

С помощью ключевого слова **BETWEEN** можно быстро и удобно задать диапазон.

SELECT name FROM guests WHERE age BETWEEN 10 and 20;

Сравните с более длинной записью

SELECT name FROM guests WHERE age >= 10 and age <=20;

С помощью условия **IN** за которыми в скобках идут нужные значения, можно задать нужные параметры.

SELECT name FROM guests WHERE age IN (10, 29);

Это короче, чем

SELECT name FROM guests WHERE age = 10 OR age = 29;

Ключевое слово **NOT IN** выполняет обратную задачу и позволяет получить записи, которые не входят в данное условие.

SELECT name FROM guests WHERE age NOT IN (10);

Ключевое слово **NOT** можно использовать не только с **IN**, но и с **BETWEEN**, **LIKE**.

Узнать число записей можно через функцию **COUNT**. Если запись содержит **NULL**, то она не учитывается.

SELECT COUNT(name) FROM guests;

Для показа минимального или максимального значения используются функции **MIN** или **MAX**:

SELECT name, MAX(age) FROM guests;

Если нам нужно вывести только определённое количество записей, то используйте ключевое слово **LIMIT** с указанием значения.

SELECT \* FROM guests LIMIT 3;

Существует расширенная версия, когда можно указать два значения через запятую. В первой указывается номер записи (отсчёт от 0), а вторая - число записей. Например, показать вторую запись из таблицы.

SELECT \* FROM guests LIMIT 1,1;

Мы рассмотрели половину операций с записями - CREATE и READ. Теперь нужно научиться изменять данные. Конечно, самый простой способ - удалить запись, а затем добавить новую с исправленными данными. Но это очень неуклюжий способ.

Для этого существует команда **UPDATE имя\_таблицы SET column = value WHERE условие**. После **UPDATE** указываете таблицу, после **SET** - в каком столбце нужно внести изменения и указывается новое значение, а затем указывается условие.

Можно обновлять группу столбцов, указывая их через запятую.

UPDATE имя\_таблицы SET первый\_столбец="новое значение", второй\_столбец="новое значение";

Команда **UPDATE** заменяет собой пару команд **INSERT/DELETE**. Обновить данные в нужном столбце:

UPDATE имя\_таблицы SET имя\_столбца = новое\_значение WHERE имя\_столбца = старое\_значение

Также можно производить математические действия: прибавлять, отнимать, умножать, делить. Увеличим возраст кота на день рождения.

UPDATE guests SET age=age+1 WHERE name="Мурзик";

Изменим прописку у Ктулху.

UPDATE guests SET city = "Нью-Йорк" WHERE name = "Ктулху";

Если не указать условие **WHERE**, то город изменится у всех гостей сразу. Удобно, если приехала большая делегация из одного города.

Последняя операция - удаление записи из таблицы. Команда **DELETE FROM имя\_таблицы**;

Без условия **WHERE** мы удалим все записи. Вряд ли вам это нужно. Давайте выпишем из гостиницы Ктулху, вежливо объяснив ему, что отель только для котов. Выпроводив незванного гостя, удаляем запись из таблицы.

DELETE FROM guests WHERE \_id = 3;

Вам не надо перечислять все столбцы, достаточно указать в условии нужный столбец. Условие **WHERE** работает аналогично как в команде **SELECT** и позволяет использовать ключевые слова **LIKE**, **BETWEEN** и т.д.

Кстати, вы можете посмотреть, какой идентификатор был вставлен в таблицу последним через команду:

SELECT LAST\_INSERT\_ROWID();

Добавить новый столбец в таблицу можно с помощью необязательного ключевого слова **ALTER**, за которым идёт название столбца в таблице.

ALTER TABLE guests ADD COLUMN weight INTEGER;

Чтобы указать, после какого столбца нужно добавить новый столбец, используйте ключевое слово **AFTER**.

Другие ключевые слова: **FIRST**, **BEFORE**, **LAST**, **SECOND**, **THIRD**.

Кроме **ADD**, также можно изменить имя и тип данных столбцов (**CHANGE**), изменить тип данных или позиции столбцов (), удалить столбец из таблицы (). Не все эти команды поддерживаются в SQLite, хотя часто используются в обычных SQL.

Переименовать саму таблицу (**RENAME TO**).

ALTER TABLE guests RENAME TO cats;

Преобразовать текст из указанного столбца в верхний регистр.

UPDATE guests SET name = UPPER(name);

Завершить работу с **sqlite3** можно через команду:

.exit

# SQLite и Android. Кошкин дом. Часть вторая

мы изучили возможности SQLite. Теперь нужно научиться подключать базу данных в приложении на Android.

SQLite зарекомендовала себя в качестве чрезвычайно надёжной системы баз данных, которая используется во многих бытовых электронных устройствах и программах, включая некоторые MP3-проигрыватели, iPhone, iPod Touch, Mozilla Firefox и др.

С помощью SQLite вы можете создавать для своего приложения независимые реляционные базы данных. Android хранит базы данных в каталоге **/data/data/<имя\_вашего\_пакета>/databases** на эмуляторе, на устройстве путь может отличаться. По умолчанию все базы данных закрытые, доступ к ним могут получить только те приложения, которые их создали.

Каждая база данных состоит из двух файлов. Имя первого файла базы данных соответствует имени базы данных. Это основной файл баз данных SQLite, в нём хранятся все данные. Вы будете создавать его программно. Второй файл — файл журнала. Его имя состоит из имени базы данных и суффикса "-journal". В файле журнала хранится информация обо всех изменениях, внесенных в базу данных. Если в работе с данными возникнет проблема, Android использует данные журнала для отмены (или отката) последних изменений. Вы с ним не будете взаимодействовать, но если вы будете просматривать внутренности своего устройства, то будете знать, зачем этот файл там присутствует.

## Интерфейс

Для начала создадим интерфейс программы. Для первой активности **MainActivity** выберем шаблон **Basic Activity**. Сразу же создадим вторую активность **EditorActivity** из шаблона **Empty Activity**.

В первой активности есть кнопка **Floating Action Button**, через которую будем попадать на вторую активность.

FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);

fab.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

Intent intent = new Intent(MainActivity.this, EditorActivity.class);

startActivity(intent);

}

});

Вторая активность предназначена для добавления новых гостей, которые поселяются в наш отель "Кошкин дом". Настроим его.

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical"

android:padding="@dimen/activity\_horizontal\_margin"

tools:context=".EditorActivity">

<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="horizontal">

<TextView

style="@style/CategoryStyle"

android:text="Общая информация" />

<LinearLayout

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_weight="2"

android:orientation="vertical"

android:paddingLeft="4dp">

<EditText

android:id="@+id/edit\_guest\_name"

style="@style/EditorFieldStyle"

android:hint="Имя"

android:inputType="textCapWords" />

<EditText

android:id="@+id/edit\_guest\_city"

style="@style/EditorFieldStyle"

android:hint="Город"

android:inputType="textCapWords" />

</LinearLayout>

</LinearLayout>

<LinearLayout

android:id="@+id/container\_gender"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="horizontal">

<TextView

style="@style/CategoryStyle"

android:text="Пол" />

<LinearLayout

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_weight="2"

android:orientation="vertical">

<Spinner

android:id="@+id/spinner\_gender"

android:layout\_width="wrap\_content"

android:layout\_height="48dp"

android:paddingRight="16dp"

android:spinnerMode="dropdown" />

</LinearLayout>

</LinearLayout>

<LinearLayout

android:id="@+id/container\_age"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:orientation="horizontal">

<TextView

style="@style/CategoryStyle"

android:text="Возраст" />

<RelativeLayout

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:layout\_weight="2"

android:paddingLeft="4dp">

<EditText

android:id="@+id/edit\_guest\_age"

style="@style/EditorFieldStyle"

android:hint="Возраст"

android:inputType="number" />

</RelativeLayout>

</LinearLayout>

</LinearLayout>

Экран состоит из нескольких текстовых полей и одного выпадающего списка для выбора пола гостя.

Инициализируем текстовые поля и выпадающий список.

private EditText mNameEditText;

private EditText mCityEditText;

private EditText mAgeEditText;

private Spinner mGenderSpinner;

/\*\*

\* Пол для гостя. Возможные варианты:

\* 0 для кошки, 1 для кота, 2 - не определен.

\*/

private int mGender = 2;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_editor);

mNameEditText = (EditText) findViewById(R.id.edit\_guest\_name);

mCityEditText = (EditText) findViewById(R.id.edit\_guest\_city);

mAgeEditText = (EditText) findViewById(R.id.edit\_guest\_age);

mGenderSpinner = (Spinner) findViewById(R.id.spinner\_gender);

setupSpinner();

}

/\*\*

\* Настраиваем spinner для выбора пола у гостя.

\*/

private void setupSpinner() {

ArrayAdapter genderSpinnerAdapter = ArrayAdapter.createFromResource(this,

R.array.array\_gender\_options, android.R.layout.simple\_spinner\_item);

genderSpinnerAdapter.setDropDownViewResource(android.R.layout.simple\_dropdown\_item\_1line);

mGenderSpinner.setAdapter(genderSpinnerAdapter);

mGenderSpinner.setSelection(2);

mGenderSpinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {

@Override

public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {

String selection = (String) parent.getItemAtPosition(position);

if (!TextUtils.isEmpty(selection)) {

if (selection.equals(getString(R.string.gender\_female))) {

mGender = 0; // Кошка

} else if (selection.equals(getString(R.string.gender\_male))) {

mGender = 1; // Кот

} else {

mGender = 2; // Не определен

}

}

}

@Override

public void onNothingSelected(AdapterView<?> parent) {

mGender = 2; // Unknown

}

});

}

Добавим несколько строковых ресурсов.

<resources>

<string name="app\_name">CatHouse</string>

<string name="action\_settings">Settings</string>

<string name="action\_save">Сохранить</string>

<string name="action\_delete">Удалить</string>

<string name="gender\_unknown">Неизвестно</string>

<string name="gender\_male">Кот</string>

<string name="gender\_female">Кошка</string>

<string-array name="array\_gender\_options">

<item>@string/gender\_female</item>

<item>@string/gender\_male</item>

<item>@string/gender\_unknown</item>

</string-array>

</resources>

## Подписываем контракт

Теперь можно заняться интеграцией базы данных в приложение.

При работе с базой данных принято создавать новый пакет **data** внутри основного пакета. Щёлкаем правой кнопкой мыши по имени пакета, выбираем **New | Package** и вводим новое имя.

В последних рекомендациях Гугла рекомендуется создавать класс-контракт. Будем придерживаться этого правила. Мы как бы подписываем контракт на работу с базой данных и предоставляем все нужные данные.

Внутри созданного пакета создаём новый класс **HotelContract**. Класс-контракт является контейнером для базы данных и может содержать несколько внутренних классов, которые представляют отдельные таблицы (не забывайте, что база данных может содержать несколько таблиц). Внутри класса создаём внутренний класс. В нашем случае будет один класс для таблицы **guests**.

Нам следует задать схему таблицы и константы для столбцов для удобства. Класс будет выглядеть так.

package ru.alexanderklimov.cathouse.data;

import android.provider.BaseColumns;

public final class HotelContract {

private HotelContract() {

};

public static final class GuestEntry implements BaseColumns {

public final static String TABLE\_NAME = "guests";

public final static String \_ID = BaseColumns.\_ID;

public final static String COLUMN\_NAME = "name";

public final static String COLUMN\_CITY = "city";

public final static String COLUMN\_GENDER = "gender";

public final static String COLUMN\_AGE = "age";

public static final int GENDER\_FEMALE = 0;

public static final int GENDER\_MALE = 1;

public static final int GENDER\_UNKNOWN = 2;

}

}

В классе используется реализация интерфейса **BaseColumn**:

public static final class GuestEntry implements **implements BaseColumns** {

Что это нам даёт? В большинстве случаев работа с базой данных происходит через специальные объекты **Cursor**, которые требуют наличия в таблице колонки с именем **\_id**. Вы можете создать столбец вручную в коде, а можно положиться на **BaseColumn**, который создаст столбец с нужным именем автоматически. Дело ваше. Если вы не будете работать с курсорами, то можете использовать и стандартное наименование **id** или вообще не использовать данный столбец, но не советую так поступать, чтобы не вырабатывать вредных привычек.

После создания класса мы можем изменить код в **EditorActivity** в том месте, где происходит выбор пола гостя через выпадающий список.

if (selection.equals(getString(R.string.gender\_female))) {

mGender = HotelContract.GuestEntry.GENDER\_FEMALE; // Кошка

} else if (selection.equals(getString(R.string.gender\_male))) {

mGender = HotelContract.GuestEntry.GENDER\_MALE; // Кот

} else {

mGender = HotelContract.GuestEntry.GENDER\_UNKNOWN; // Не определен

}

## SQLiteOpenHelper

Следующий шаг - создание класса в пакете **data**, который наследуется от специального класса **SQLiteOpenHelper** и непосредственно работает с базой данных. В классе создаются константы для удобной работы. Также реализуются методы **onCreate()** и **onUpgrade()**.

Созданный класс будет работать с базой данных - добавлять, выбирать, удалять записи и прочие операции.

Напомню, как выглядит схема нашей таблицы.

CREATE TABLE guests(\_id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL,

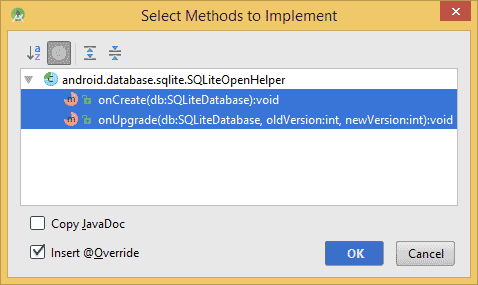
city TEXT NOT NULL,

gender INTEGER NOT NULL DEFAULT 3,

age INTEGER NOT NULL DEFAULT 0);

Щёлкаем правой кнопкой мыши на имени пакета в левой части студии и выбираем в меню **New** | **Java Class** и в диалоговом окне выбираем имя для нового класса, например, **HotelDbHelper**. Слово **Helper** обычно используют, чтобы показать, что класс является обёрткой (вспомогательным классом) какого-то абстрактного класса. Впрочем, вы можете придумать более замысловатое название, например, **ILoveNewYork** или **CatsForever**. Спустя год, когда вы вернётесь к своему примеру, это будет так увлекательно вспоминать, для чего был создан класс с таким красивым именем.

У нас появится заготовка. Наследуемся от **SQLiteOpenHelper**. Студия предложит создать два обязательных метода **onCreate()** и **onUpgrade()**, о которых поговорим позже.



После добавления методов студия по-прежнему ругается. Теперь ему подавай конструкторы. Получится такой код.:

Класс **HotelDbHelper**.

package ru.alexanderklimov.cathouse.data;

import android.content.Context;

import android.database.sqlite.SQLiteDatabase;

import android.database.sqlite.SQLiteOpenHelper;

import ru.alexanderklimov.cathouse.data.HotelContract.GuestEntry;

public class HotelDbHelper extends SQLiteOpenHelper {

public static final String LOG\_TAG = HotelDbHelper.class.getSimpleName();

/\*\*

\* Имя файла базы данных

\*/

private static final String DATABASE\_NAME = "hotel.db";

/\*\*

\* Версия базы данных. При изменении схемы увеличить на единицу

\*/

private static final int DATABASE\_VERSION = 1;

/\*\*

\* Конструктор {@link HotelDbHelper}.

\*

\* @param context Контекст приложения

\*/

public HotelDbHelper(Context context) {

super(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

/\*\*

\* Вызывается при создании базы данных

\*/

@Override

public void onCreate(SQLiteDatabase db) {

// Строка для создания таблицы

String SQL\_CREATE\_GUESTS\_TABLE = "CREATE TABLE " + GuestEntry.TABLE\_NAME + " ("

+ HotelContract.GuestEntry.\_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "

+ GuestEntry.COLUMN\_NAME + " TEXT NOT NULL, "

+ GuestEntry.COLUMN\_CITY + " TEXT NOT NULL, "

+ GuestEntry.COLUMN\_GENDER + " INTEGER NOT NULL DEFAULT 3, "

+ GuestEntry.COLUMN\_AGE + " INTEGER NOT NULL DEFAULT 0);";

// Запускаем создание таблицы

db.execSQL(SQL\_CREATE\_GUESTS\_TABLE);

}

/\*\*

\* Вызывается при обновлении схемы базы данных

\*/

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

}

}

Третий параметр **null** в суперклассе используется для работы с курсорами. Сейчас их не используем, поэтому оставим в покое.

Как вы уже догадались, константа **DATABASE\_NAME** отвечает за имя файла, в котором будет храниться база данных приложения. Можно придумать любое имя и обойтись без расширения. Но мне так привычнее.

Вторая константа **DATABASE\_VERSION** требует дополнительных объяснений. Она отвечает за номер версии базы. Принцип её работы схож с номером версий самого приложения. Когда мы видим, что вышла новая версия Chrome 33, то понимаем, что пора обновляться. Аналогично поступает и само приложение, когда замечает, что номер версии базы стал другим. Как только программа заметила обновление номера базы, она запускает метод **onUpgrade()**, который у нас сформировался автоматически. В этом методе необходимо разместить код, который должен сработать при обновлении базы.

Метод **onCreate()** вопросов не вызывает - здесь создаётся сама база данных с необходимыми данными для работы.

Метод вызывается, если в устройстве нет базы данных и наш класс должен создать его. Как мы помним, у метода есть параметр **db**, который относится к классу **SQLiteDatabase**. У класса есть специальный метод **execSQL()**, которому нужно передать запрос (SQL-скрипт) для создания таблицы. Для создания таблицы в SQL используется команда **CREATE TABLE ...**. Для удобства вынесем команду в отдельную строку. Аналогично поступим с командой **DROP TABLE**. Так как строка очень длинная и состоит из множества строковых переменных, которые нужно соединить в одну цепочку, то поступают следующим образом. Создаём ещё одну строковую константу для формирования скрипта и передадим её в метод.

@Override

public void onCreate(SQLiteDatabase db) {

// Строка для создания таблицы

String SQL\_CREATE\_GUESTS\_TABLE = "CREATE TABLE " + GuestEntry.TABLE\_NAME + " ("

+ HotelContract.GuestEntry.\_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "

+ GuestEntry.COLUMN\_NAME + " TEXT NOT NULL, "

+ GuestEntry.COLUMN\_CITY + " TEXT NOT NULL, "

+ GuestEntry.COLUMN\_GENDER + " INTEGER NOT NULL DEFAULT 3, "

+ GuestEntry.COLUMN\_AGE + " INTEGER NOT NULL DEFAULT 0);";

// Запускаем создание таблицы

db.execSQL(SQL\_CREATE\_GUESTS\_TABLE);

}

Основная сложность - не пропустить пробелы в запросе. Очень часто пропущенный пробел становится источником проблем и ваше приложение не может создать таблицу. Можете сначала написать сам скрипт создания таблицы, а уже потом заменять отдельные слова константами. Идентификатор **\_id** всегда должен использовать **INTEGER PRIMARY KEY AUTOINCREMENT**, остальные колонки на ваше усмотрение.

Теперь нужно объяснить, зачем нужен этот метод **onUpgrade()**. Представьте ситуацию, что вы первоначально создали в базе таблицу, в которую заносятся имена котов и их электронные адреса и телефоны (продвинутые кошаки). Вроде всё замечательно. Если нужно поздравить усатых-полосатых с Международным днём кошек, который отмечается 1 марта, то проблем нет никаких. У вас есть список имён, по которому вы можете пройтись и лично написать каждому письмо. Пользователи, скачавшие ваше приложение, с удовольствием заполняют базу данных и дружно пишут письма мелким почерком. И вдруг до вас дошло, что совершили непростительную ошибку. Вы забыли добавить в базу данных даты рождения котов. А значит их никто не поздравит и не погладит (((.



Вы исправляете досадное упущение и выкладываете новую версию программы в открытый доступ. Новые пользователи, которые установят программу первый раз, радуются жизни - у них есть все необходимые данные для работы. Но что делать тем, кто уже работает со старой программой? Обновившись, они увидят дополнительное текстовое поле для ввода даты рождения, но в старой базе нет колонки для хранения новых данных. И ваша программа завершится с ошибкой. Полностью удалять и устанавливать новую версию программы тоже не выход - тогда пропадут старые данные, что тоже не желательно. Для таких случаев вы пишете код в методе **onUpgrade()**, чтобы при обновлении поменялась структура базы данных у старых пользователей. Мы позже попробуем смоделировать эту ситуацию.

Итак, метод **onUpgrade()** вызывается при несовпадении версий. Часто в этом методе просто удаляют существующую таблицу и заменяют её на новую. Это самое простое и практичное решение. Впрочем, на первых порах, вам вряд ли придётся заниматься подобными делами, поэтому метод можно оставить даже пустым.

@Override

public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {

// Запишем в журнал

Log.w("SQLite", "Обновляемся с версии " + oldVersion + " на версию " + newVersion);

// Удаляем старую таблицу и создаём новую

db.execSQL("DROP TABLE IF IT EXISTS " + DATABASE\_TABLE);

// Создаём новую таблицу

onCreate(db);

}

Когда ваше приложение будет готово, то в папке **data/data/имя\_пакета/databases** появится файл **hotel.db** (позже я вам покажу). Этот файл и будет вашей базой данных, в которой будет находиться созданная вами таблица. На данный момент в студии нет готового плагина для просмотра таблиц (в Eclipse есть), но вроде уже видел плагин от сторонних разработчиков. А пока вам придётся скачивать из устройства файл базы данных и просматривать его на компьютере специальными программами, работающими с SQLite на локальном компьютере.

## Работаем с записями базы данных

Чтобы проверить работоспособность базы данных, в главной активности поместим вспомогательный метод **displayDatabaseInfo()**для отображения информации.

package ru.alexanderklimov.cathouse;

import android.content.Intent;

import android.database.Cursor;

import android.database.sqlite.SQLiteDatabase;

import android.os.Bundle;

import android.support.design.widget.FloatingActionButton;

import android.support.v7.app.AppCompatActivity;

import android.support.v7.widget.Toolbar;

import android.view.Menu;

import android.view.MenuItem;

import android.view.View;

import android.widget.TextView;

import ru.alexanderklimov.cathouse.data.HotelContract.GuestEntry;

import ru.alexanderklimov.cathouse.data.HotelDbHelper;

public class MainActivity extends AppCompatActivity {

private HotelDbHelper mDbHelper;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);

setSupportActionBar(toolbar);

FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);

fab.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

Intent intent = new Intent(MainActivity.this, EditorActivity.class);

startActivity(intent);

}

});

mDbHelper = new HotelDbHelper(this);

}

@Override

protected void onStart() {

super.onStart();

displayDatabaseInfo();

}

@Override

public boolean onCreateOptionsMenu(Menu menu) {

// Inflate the menu; this adds items to the action bar if it is present.

getMenuInflater().inflate(R.menu.menu\_main, menu);

return true;

}

@Override

public boolean onOptionsItemSelected(MenuItem item) {

switch (item.getItemId()) {

case R.id.action\_insert\_new\_data:

// Пока ничего не делаем

return true;

case R.id.action\_delete\_all\_entries:

// Пока ничего не делаем

return true;

}

return super.onOptionsItemSelected(item);

}

private void displayDatabaseInfo() {

// Создадим и откроем для чтения базу данных

SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Зададим условие для выборки - список столбцов

String[] projection = {

GuestEntry.\_ID,

GuestEntry.COLUMN\_NAME,

GuestEntry.COLUMN\_CITY,

GuestEntry.COLUMN\_GENDER,

GuestEntry.COLUMN\_AGE };

// Делаем запрос

Cursor cursor = db.query(

GuestEntry.TABLE\_NAME, // таблица

projection, // столбцы

null, // столбцы для условия WHERE

null, // значения для условия WHERE

null, // Don't group the rows

null, // Don't filter by row groups

null); // порядок сортировки

TextView displayTextView = (TextView) findViewById(R.id.text\_view\_info);

try {

displayTextView.setText("Таблица содержит " + cursor.getCount() + " гостей.\n\n");

displayTextView.append(GuestEntry.\_ID + " - " +

GuestEntry.COLUMN\_NAME + " - " +

GuestEntry.COLUMN\_CITY + " - " +

GuestEntry.COLUMN\_GENDER + " - " +

GuestEntry.COLUMN\_AGE + "\n");

// Узнаем индекс каждого столбца

int idColumnIndex = cursor.getColumnIndex(GuestEntry.\_ID);

int nameColumnIndex = cursor.getColumnIndex(GuestEntry.COLUMN\_NAME);

int cityColumnIndex = cursor.getColumnIndex(GuestEntry.COLUMN\_CITY);

int genderColumnIndex = cursor.getColumnIndex(GuestEntry.COLUMN\_GENDER);

int ageColumnIndex = cursor.getColumnIndex(GuestEntry.COLUMN\_AGE);

// Проходим через все ряды

while (cursor.moveToNext()) {

// Используем индекс для получения строки или числа

int currentID = cursor.getInt(idColumnIndex);

String currentName = cursor.getString(nameColumnIndex);

String currentCity = cursor.getString(cityColumnIndex);

int currentGender = cursor.getInt(genderColumnIndex);

int currentAge = cursor.getInt(ageColumnIndex);

// Выводим значения каждого столбца

displayTextView.append(("\n" + currentID + " - " +

currentName + " - " +

currentCity + " - " +

currentGender + " - " +

currentAge));

}

} finally {

// Всегда закрываем курсор после чтения

cursor.close();

}

}

}

Массив **projection** - это список столбцов, которые нас интересуют. В SQL-запросе мы их указываем в операторе **SELECT**:

SELECT **name, city** FROM guests;

В методе **query()** третий и четвёртый параметр определяют условие **WHERE**. Возьмём случай с выражением:

SELECT \* FROM guests WHERE \_id = 1;

В коде такое выражение выглядело бы так.

String selection = GuestEntry.\_ID + "=?";

String[] selectionArgs = {"1"};

Как видим, в знак вопроса подставляется нужное значение.

Посмотрите ещё [несколько примеров](http://developer.alexanderklimov.ru/android/sqlite/cathouse2.php#updatedata).

Последний аргумент отвечает за сортировку по возрастанию или убыванию. Например, по возрасту.

SELECT name FROM guests WHERE \_id > 1 BY age DESC;

// Зададим условие для выборки - список столбцов

String[] projection = {

GuestEntry.COLUMN\_NAME };

String selection = GuestEntry.\_ID + ">?";

String[] selectionArgs = {"1"};

Cursor cursor = db.query(

GuestEntry.TABLE\_NAME, // таблица

projection, // столбцы

selection, // столбцы для условия WHERE

selectionArgs, // значения для условия WHERE

null, // Don't group the rows

null, // Don't filter by row groups

GuestEntry.COLUMN\_AGE + " DESC"); // порядок сортировки

## Чтение данных

Считывать данные также можно двумя способами. В любом случае результат возвращается в виде объекта [Cursor](http://developer.alexanderklimov.ru/android/sqlite/cursor.php). Не путайте его с курсором мыши, который бегает у вас на экране.



### Первый способ. Метод query()

Извлечение данных происходит через метод **query()**. Данные хранятся в наборе строк, которые можно представить в виде таблицы. Из этой таблицы вы уже можете извлечь конкретное значение.

У метода **query()** множество параметров. В первом параметре укажите имя таблицы, во втором - массив имён колонок, далее идут дополнительные условия. Пока везде оставим **null**. В нашем примере мы добавили одну запись и извлечь её просто.

Cursor cursor = db.query(

GuestEntry.TABLE\_NAME, // таблица

projection, // столбцы

null, // столбцы для условия WHERE

null, // значения для условия WHERE

null, // Don't group the rows

null, // Don't filter by row groups

null); // порядок сортировки

// Проходим через все ряды

while (cursor.moveToNext()) {

// Используем индекс для получения строки или числа

int currentID = cursor.getInt(idColumnIndex);

String currentName = cursor.getString(nameColumnIndex);

String currentCity = cursor.getString(cityColumnIndex);

int currentGender = cursor.getInt(genderColumnIndex);

int currentAge = cursor.getInt(ageColumnIndex);

// Выводим значения каждого столбца

displayTextView.append(("\n" + currentID + " - " +

currentName + " - " +

currentCity + " - " +

currentGender + " - " +

currentAge));

}

### Второй способ. Метод rawQuery()

Второй способ использует сырой (raw) SQL-запрос. Сначала формируется строка запроса и отдаётся методу **rawQuery()**.

// Абстрактный пример

// Метод 2: Сырой SQL-запрос

String query = "SELECT " + DatabaseHelper.COLUMN\_ID + ", "

+ DatabaseHelper.CAT\_NAME\_COLUMN + " FROM " + DatabaseHelper.TABLE\_NAME;

Cursor cursor2 = mDatabase.rawQuery(query, null);

while (cursor2.moveToNext()) {

int id = cursor2.getInt(cursor2

.getColumnIndex(DatabaseHelper.COLUMN\_ID));

String name = cursor2.getString(cursor2

.getColumnIndex(DatabaseHelper.CAT\_NAME\_COLUMN));

Log.i("LOG\_TAG", "ROW " + id + " HAS NAME " + name);

}

cursor2.close();

Запустите проект. При запуске создаётся база данных. Убедиться в этом можно, если запустить **Android Device Monitor**. Выберите вкладку **File Explorer** и найдите своё приложение (на эмуляторе). Вы увидите, что появилась папка **data/data/имя\_пакета/databases** с файлом **hotel.db**. Метод **getReadableDatabase** создаёт или открывает базу данных.

Сейчас мы увидим, что пока у нас 0 гостей.

Небольшое предупреждение. При работе с базой данных мы обращаемся к файлу. Если база данных очень большая, то запросы не будут мгновенными. Операции с файлами являются медленными, поэтому следует использовать многопоточность. Для наших примеров это не страшно, поэтому мы пока не будем усложнять код.

## Вставка данных для проверки

Рассмотрим, как вставлять новые данные. Добавим в меню главной активности пункт "Вставить данные". Для вставки данных применяется метод **ContentValues.put()**. В методе указываются ключ и значение. В качестве ключа выступает имя столбца таблицы, а его значением будет нужная информация о госте. Так как идентификатор будет вставляться автоматически, то его не используем. После того, как вы заполните все столбцы таблицы, вызывайте метод **insert()**, который и разместит данные в базе.

Напишем вспомогательный метод.

private void insertGuest() {

// Gets the database in write mode

SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Создаем объект ContentValues, где имена столбцов ключи,

// а информация о госте является значениями ключей

ContentValues values = new ContentValues();

values.put(GuestEntry.COLUMN\_NAME, "Мурзик");

values.put(GuestEntry.COLUMN\_CITY, "Мурманск");

values.put(GuestEntry.COLUMN\_GENDER, GuestEntry.GENDER\_MALE);

values.put(GuestEntry.COLUMN\_AGE, 7);

long newRowId = db.insert(GuestEntry.TABLE\_NAME, null, values);

}

Вызовем метод в обработчике нажатия пункта меню.

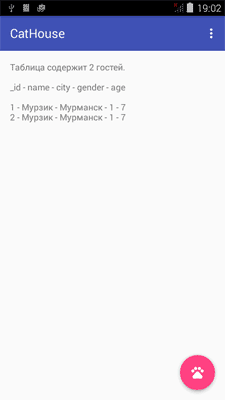
case R.id.action\_insert\_new\_data:

insertGuest();

displayDatabaseInfo();

return true;

Сразу после вставки вызываем метод **displayDatabaseInfo()**, чтобы увидеть результат. Можно нажимать несколько раз. Так как данные жёстко заданы в коде, то увидим одинаковые данные, кроме увеличивающего значения идентификатора.



## Вставка данных. Общая информация

Теперь разберём подробнее, как делать вставки.

### Первый способ. ContentValues

Для вставки сначала подготавливаются данные с помощью класса **ContentValues**. Вы указываете имя колонки таблицы и значение для неё, т.е. работает по принципу "ключ-значение". Когда подготовите все данные во все столбцы, то вызывайте метод **insert()**, который сразу раскидает данные по столбцам.

Способ очень удобен, требует мало кода и легко читаем. Вы создаёте экземпляр класса, а затем с помощью метода **put()**записываете в нужную колонку нужные данные. После чего вызывается метод **insert()**, который помещает подготовленные данные в таблицу.

У метода **insert()** три аргумента. В первом указывается имя таблицы, в которую будут добавляться записи. В третьем указывается объект **ContentValues**, созданный ранее. Второй аргумент используется для указания колонки. SQL не позволяет вставлять пустую запись, и если будет использоваться пустой **ContentValue**, то укажите во втором аргументе **null** во избежание ошибки.

### Второй способ. SQL-запрос

Существует также другой способ вставки через метод **execSQL()**, когда подготавливается нужная строка и запускается скрипт. Этот способ возможно понравится PHP-кодерам, которые привыкли к такому синтаксису.

В этом варианте используется традиционный SQL-запрос **INSERT INTO...**. Основное неудобство при этом способе - не запутаться в кавычках. Если что-то не вставляется, то смотрите логи сообщений.

// Абстрактный пример

db = new DatabaseHelper(this);

SQLiteDatabase sqdb = db.getWritableDatabase();

String insertQuery = "INSERT INTO " +

DatabaseHelper.DATABASE\_TABLE +

" (" + DatabaseHelper.CAT\_NAME\_COLUMN + ") VALUES ('Васька')";

sqdb.execSQL(insertQuery);

Научившись вставлять данные, можно заняться второй активностью, которая и предназначена для этих целей.

## Наполняем базу данных

Создадим вспомогательный метод для вставки записи в базу данных. Для этого считываем данные, которые вводятся в текстовые поля, а далее по предыдущему учебному примеру.

private void insertGuest() {

// Считываем данные из текстовых полей

String name = mNameEditText.getText().toString().trim();

String city = mCityEditText.getText().toString().trim();

String ageString = mAgeEditText.getText().toString().trim();

int age = Integer.parseInt(ageString);

HotelDbHelper mDbHelper = new HotelDbHelper(this);

SQLiteDatabase db = mDbHelper.getWritableDatabase();

ContentValues values = new ContentValues();

values.put(GuestEntry.COLUMN\_NAME, name);

values.put(GuestEntry.COLUMN\_CITY, city);

values.put(GuestEntry.COLUMN\_GENDER, mGender);

values.put(GuestEntry.COLUMN\_AGE, age);

// Вставляем новый ряд в базу данных и запоминаем его идентификатор

long newRowId = db.insert(GuestEntry.TABLE\_NAME, null, values);

// Выводим сообщение в успешном случае или при ошибке

if (newRowId == -1) {

// Если ID -1, значит произошла ошибка

Toast.makeText(this, "Ошибка при заведении гостя", Toast.LENGTH\_SHORT).show();

} else {

Toast.makeText(this, "Гость заведён под номером: " + newRowId, Toast.LENGTH\_SHORT).show();

}

}

Метод вызывается в меню для значка с галочкой, которая выводится на панели действия активности.

case R.id.action\_save:

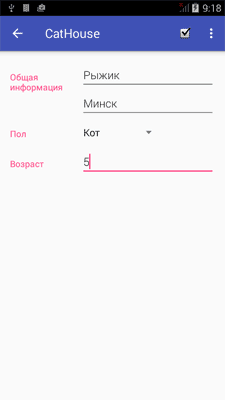
insertGuest();

// Закрываем активность

finish();

return true;

Запускаем проект и проверяем работу кода.



## Изменение данных

Обновление не реализовано в программе, проделайте это самостоятельно.

Если запись уже существует, но вам нужно изменить какое-то значение, то вместо **insert()** используйте метод **update()**. В остальном принцип тот же. Предположим, что повторном осмотре котёнка выяснилось, что это кот, а не кошка. Если вы уже назвали котёнка Муркой, то логично назвать его теперь Мурзиком. Вызываем метод **put()**, а затем обновляем запись в базе данных.

ContentValues values = new ContentValues();

values.put(GuestEntry.COLUMN\_NAME, "Мурзик");

db.update(GuestEntry.TABLE\_NAME,

values,

GuestEntry.COLUMN\_NAME + "= ?", new String[]{"Мурка"});

Первый параметр метода **update()** содержит имя таблицы. Второй параметр указывает, какие значения должны использоваться для обновления. Третий параметр задает условия отбора обновляемых записей (WHERE). В приведенном примере **"NAME = ?"**означает, что столбец **NAME** должен быть равен некоторому значению. Символ **?** обозначает значение столбца, которое определяется содержимым последнего параметра. Если в двух последних параметрах метода передаётся значение **null**, будут обновлены ВСЕ записи в таблице.

Возможны и сложные условия.

db.update(GuestEntry.TABLE\_NAME,

values,

"NAME = ? OR EMAIL = ?",

new String[] {"Васька", "vaska@cat.com"});

Если столбец не является строкой, то его нужно преобразовать в строку, чтобы использовать в качестве условий.

db.update(GuestEntry.TABLE\_NAME.TABLE\_NAME,

values,

"\_id = ?",

new String[] {Integer.toString(1)});

Будьте осторожны с обновлениями. Если в последних двух параметрах передать значение **null**, то будут обновлены все записи в таблице, так как в запросе нет условий.

db.update(mDatabaseHelper.TABLE\_NAME,

values,

null, null);

## Удаление данных

Также не реализовано. Проделайте самостоятельно.

Метод **delete()** класса **SQLiteDatabase** работает по тому же принципу, как и метод **update()**. Он имеет следующую форму:

public int delete (String table,

String whereClause,

String[] whereArgs)

## Внедрение опасного кода

При работе с базой данной надо следить за безопасностью данных. Опытный пользователь может удалить базу. На своём устройстве он делать этого может и не будет, но на устройстве жертвы вполне.

Простой вариант атаки. Допустим у нас есть поле для ввода идентификатора, чтобы узнать информацию о госте. Нормальный пользователь введёт число "3" для поиска третьего гостя. В коде это будет следующим образом.

String selection = "GuestEntry.\_ID + " == " + userInput + ";";

Тогда идентификатор будет **\_ID == 2;**.

Хакер может ввести следующую строку в текстовое поле:

1; DROP TABLE guests;

В коде это превратится в следующее:

\_ID == 1; DROP TABLE guests;

Таким образом вредитель внедрил нежелательный код, который удалит таблицу.

Также вы можете скачать [исходный код](http://developer.alexanderklimov.ru/android/apk/cathouse_sqlite.zip).