**SQLite на Android**

[Запускаем SQLite на эмуляторе](http://developer.alexanderklimov.ru/android/sqlite/android-sqlite.php#emulator)  
[Класс SQLiteOpenHelper: Создание базы данных](http://developer.alexanderklimov.ru/android/sqlite/android-sqlite.php#SQLiteOpenHelper)  
[Класс SQLiteDatabase: Управление базой данных](http://developer.alexanderklimov.ru/android/sqlite/android-sqlite.php#SQLiteDatabase)  
[Вставляем картинки в базу](http://developer.alexanderklimov.ru/android/sqlite/android-sqlite.php#imagedb)

SQLite доступен на любом Android-устройстве, его не нужно устанавливать отдельно.

SQLite поддерживает типы TEXT (аналог String в Java), INTEGER (аналог long в Java) и REAL (аналог double в Java). Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite сама по себе не проверяет типы данных, поэтому вы можете записать целое число в колонку, предназначенную для строк, и наоборот.

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| NULL | пустое значение |
| INTEGER | целочисленное значение |
| REAL | значение с плавающей точкой |
| TEXT | строки или символы в кодировке UTF-8, UTF-16BE или UTF-16LE |
|  |  |
| NUMERIC | здесь можно хранить булевы значения, а также время и дату |
| BLOB | бинарные данные |

Вы не найдёте здесь тип, работающий с датами. Можно использовать строковые значения, например, как *2013-03-28* (28 марта 2013 года). Для даты со временем рекомендуется использовать формат *2013-03-27T07:58*. В таких случаях можно использовать некоторые функции SQLite для добавления дней, установки начала месяца и т.д. Учтите, что SQLite не поддерживает часовые пояса.

Также не поддерживается тип **boolean**. Используйте числа 0 для **false** и 1 для **true**.

Не используйте тип **BLOB** для хранения данных (картинки) в Android. Лучше хранить в базе путь к изображениям, а сами изображения хранить в файловой системе.

Обратите внимание, что здесь не используется популярный в MySQL тип **varchar(n)**, ограничивающий длину строки.

Всё, что вам нужно для начала работы с базой данных - задать необходимые настройки для создания или обновления базы данных.

Так как сама база данных SQLite представляет собой файл, то по сути при работе с базой данных, вы взаимодействуете с файлом. Поэтому операции чтения и записи могут быть довольно медленными. Настоятельно рекомендуется использовать асинхронные операции, например, при помощи класса AsyncTask.

Когда ваше приложение создаёт базу данных, она сохраняется в каталоге **DATA/data/имя\_пакета/databases/имя\_базы.db**.

Метод **Environment.getDataDirectory()** возвращает путь к каталогу **DATA**.

Основными пакетами для работы с базой данных являются **android.database** и **android.database.sqlite**.

База данных SQLite доступна только приложению, которое создаёт её. Если вы хотите дать доступ к данным другим приложениям, вы можете использовать контент-провайдеры (ContentProvider).

**Запускаем SQLite на эмуляторе**

С помощью утилиты [ADB](http://developer.alexanderklimov.ru/android/theory/adb.php) можно запустить SQLite на эмуляторе и работать с базами данных напрямую.

Напомню, что можно активировать команду **sqlitе3** для одной из перечисленных баз данных, введя следующую команду:

#sqlite3 /data/data/com.android.providers.contacts/databases/contacts.db

Для завершения работы с **sqlite3** напишите:

sqlite> .exit

Обратите внимание: приглашение для adb - это #, а приглашение для sqlitе3 - это **sqlite>**.

Описание доступных команд **sqlite3** есть на сайте: http://www.sqlite.org/sqlite.html. Перечислим некоторые важные команды.

Чтобы просмотреть список таблиц:

sqlite> .tables

Быстрый доступ к главной таблице:

SELECT name FROM sqlite\_master

WHERE type IN ('table', 'view') AND name NOT LIKE 'sqlite %'

UNION ALL

SELECT name FROM sqlite\_temp\_master

WHERE type IN ('tаblе', 'view')

ORDER ВY 1

Таблица **sqlite\_master** - это главная таблица (master table), в которой отслеживаются таблицы и виды, содержащиеся в базе данных. Следующая команда распечатывает инструкцию **create** для таблицы **people**, находящейся в базе данных **contacts.db**:

.schema people

Это один из способов, позволяющих узнать названия всех столбцов, которые содержатся в таблице базы данных. Можно скопировать базу данных на локальный компьютер и изучать её в более комфортных условиях. Чтобы переместить файл **contacts.db**, можно дать следующую команду:

adb pull /data/data/com.android.providers.contacts/databases/contacts.db d:/somelocaldir/contacts.db

**Классы для работы с SQLite**

Работа с базой данных сводится к следующим задачам:

* Создание и открытие базы данных
* Создание таблицы
* Создание интерфейса для вставки данных (insert)
* Создание интерфейса для выполнения запросов (выборка данных)
* Закрытие базы данных

**Класс ContentValues**

Класс ContentValues используется для добавления новых строк в таблицу. Каждый объект этого класса представляет собой одну строку таблицы и выглядит как ассоциативный массив с именами столбцов и значениями, которые им соответствуют.

**Курсоры**

В Android запросы к базе данных возвращают объекты класса [Cursor](http://developer.alexanderklimov.ru/android/theory/cursor.php). Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе.

**Класс SQLiteOpenHelper: Создание базы данных**

Библиотека Android содержит абстрактный класс **SQLiteOpenHelper**, с помощью которого можно создавать, открывать и обновлять базы данных. Это основной класс, с которым вам придётся работать в своих проектах. При реализации этого вспомогательного класса от вас скрывается логика, на основе которой принимается решение о создании или обновлении базы данных перед ее открытием. Класс **SQLiteOpenHelper** содержит два обязательных абстрактных метода:

* **onCreate()** — вызывается при первом создании базы данных
* **onUpgrade()** — вызывается при модификации базы данных

Также используются другие методы класса:

* onDowngrade(SQLiteDatabase, int, int)
* onOpen(SQLiteDatabase)
* getReadableDatabase()
* getWritableDatabase()

В приложении необходимо создать собственный класс, наследуемый от **SQLiteOpenHelper**. В этом классе необходимо реализовать указанные обязательные методы, описав в них логику создания и модификации вашей базы.

В этом же классе принято объявлять открытые строковые константы для названия таблиц и полей создаваемой базы данных, которые клиенты могут использовать для определения столбцов при выполнении запросов к базе данных. Например, так можно объявить константы для таблицы **CONTACT**:

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

public static final String NAME = "FIRST\_NAME";

public static final String PHONE = "PHONE";

Лучше, если вы будете давать сразу понятные имена, указывающие на работу с таблицей. Если имя переменной **TABLE\_NAME** ещё можно оставить, то для других лучше использовать более говорящие имена, например, **KEY\_NAME** и **KEY\_PHONE** или **COLUMN\_NAME**, **COLUMN\_PHONE**.

Ваш класс, расширяющий **SQLiteOpenHelper**, также неявно наследует интерфейс **BaseColumns**, в котором определена строковая константа **\_ID**, представляющая имя поля для идентификаторов записей. В создаваемых таблицах базы данных поле **\_ID** должно иметь тип **INTEGER PRIMARY KEY AUTOINCREMENT**. Описатель **AUTOINCREMENT** является необязательным. Часто в других примерах идентификатор создаётся вручную, смотрите как вам удобнее. Только всегда называйте его именно **\_id**. Такое название используется в Android для работы с курсорами и поэтому придерживайтесь данного правила.

В методе **onCreate()** необходимо реализовать логику создания таблиц и при необходимости заполнить их начальными данными при помощи SQL-команды, например:

@Override

public void onCreate(SQLiteDatabase db)

{

db.execSQL("CREATE TABLE + TABLE\_NAME

(\_id INTEGER PRIMARY KEY AUTOINCREMENT,

COLUMN\_NAME TEXT,

COLUMN\_PHONE TEXT);");

}

Здесь показана условная команда, в реальном коде вам нужно учитывать пробелы между константами, чтобы получилась правильная строка SQL-команды.

Также надо указать номер версии базы данных, начиная со значения 1. По этому номеру класс-обёртка будет понимать, что структуру базы данных следует обновить.

private static final int DATABASE\_VERSION = 1;

В методе **onUpgrade()** можно, например, реализовать запрос в базу данных на уничтожение таблицы (DROP TABLE), после чего вновь вызвать метод**onCreate()** для создания версии таблицы с обновленной структурой, например, так:

@Override

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

{

db.execSQL("DROP TABLE IF EXISTS + TABLE\_NAME);

onCreate(db);

}

В параметрах метода используются номера версий базы данных - старая и новая. О номере версии говорилось выше.

Обновляется структура базы данных следующим образом. Сначала меняем порядковый номер.

private static final int DATABASE\_VERSION = 2;

При первой установке приложения базы данных ещё не существует. Тут проверять пока нечего. При установке новой версии приложения система проверит номер версии базы данных. Если он больше, чем было, то вызовется метод **onUpgrade()**. Если наоборот, то вызовется метод **onDowngrade()** (обычно так происходит редко).

Как использовать? Просто ставим условие на проверку.

@Override

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

if (oldVersion == 1) {

// Код для работы с базой данных с версией 1

}

if (oldVersion < 3) {

// Код для работы с базой данных с версией 1 или 2

}

}

Допустим, в таблицу нужно добавить новую колонку в дополнение к существующим колонкам NAME, DESCRIPTION и IMAGE\_RESOURCE\_ID. Для изменения таблицы используется выражение SQL с ключевым словом **ALTER**

ALTER TABLE DATABASE\_TABLE ADD COLUMN AGE INTEGER

Для замены имени таблицы используется выражение (меняем имя таблицы DOG на CAT):

ALTER TABLE DOG RENAME TO CAT

Для удаления таблицы:

DROP TABLE DOG

Соответственно, нужно вызвать метод **SQLiteDatabase.execSQL()**, передав ему нужную команду SQL.

Для удобства создадим вспомогательный метод **updateDatabase()**.

private void updateDatabase(SQLiteDatabase db, int oldVersion, int newVersion) {

if (oldVersion < 1) {

db.execSQL("CREATE TABLE DATABASE\_TABLE (\_id INTEGER PRIMARY KEY AUTOINCREMENT, "

+ "NAME TEXT, "

+ "DESCRIPTION TEXT, "

+ "IMAGE\_RESOURCE\_ID INTEGER);");

insertCat(db, "Barsik", "Very clever cat", R.drawable.barsik);

insertCat(db, "Murzik", "Beautiful cat", R.drawable.murzik);

insertCat(db, "Vaska", "Always hungry", R.drawable.vaska);

}

if (oldVersion < 2) {

//Код для добавления новой колонки

myDatabase.execSQL("ALTER TABLE CONTACT ADD COLUMN AGE INTEGER;");

}

}

Созданный метод теперь можно использовать в коде следующим образом.

// Поменяли номер версии базы данных

private static final int DATABASE\_VERSION = 2;

@Override

public void onCreate(SQLiteDatabase db){

updateDatabase(db, 0, DB\_VERSION);

}

@Override

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

updateDatabase(db, oldVersion, newVersion);

}

Чтобы использовать реализацию вспомогательного класса, создайте новый экземпляр, передайте его конструктору контекст, имя базы данных, текущую версию и объект класса **CursorFactory** (если вы его используете). Вызовите метод **getReadableDatabase()** или **getWritableDatabase()**, чтобы открыть и вернуть экземпляр базы данных, с которой мы имеем дело (он будет доступен как для чтения, так и для записи). Вызов метода **getWritableDatabase()** может завершиться неудачно из-за проблем с полномочиями или нехваткой места на диске, поэтому лучше предусмотреть откат к методу **getReadableDatabase()**.

DbHelper dbHelper = new DbHelper(context, DATABASE\_NAME, null, DATABASE\_VERSION);

SQLiteDatabase db;

try {

db = dbHelper.getWritableDatabase();

}

catch (SQLiteException ex){

db = dbHelper.getReadableDatabase();

}

Если база данных не существует, вспомогательный класс вызывает свой обработчик **onCreate()**; если версия базы данных изменилась, вызовется обработчик**onUpgrade()**. В любом случае вызов методов **getWritableDatabase/getReadableDatabase**, в зависимости от ситуации, вернет существующую, только что созданную или обновленную базу данных.

Оба метода имеют разные названия, но возвращают один и тот же объект. Только метод для чтения **getReadableDatabase()** сначала проверит доступность на чтение/запись. В случае ошибки метод проверит доступность на чтение и только потом уже вызовет исключение. Второй метод сразу проверяет доступ к чтению/записи и вызывает исключение при отказе в доступе.

Если вы планируете использовать несколько таблиц, то рекомендуется создавать для каждой таблицы отдельный класс.

**SQLiteDatabase**

Для управления базой данных SQLite существует класс **SQLiteDatabase**. В классе **SQLiteDatabase** определены методы **query()**, **insert()**, **delete()** и **update()** для чтения, добавления, удаления, изменения данных. Кроме того, метод **execSQL()** позволяет выполнять любой допустимый код на языке SQL применимо к таблицам базы данных, если вы хотите провести эти (или любые другие) операции вручную.

Каждый раз, когда вы редактируете очередное значение из базы данных, нужно вызывать метод **refreshQuery()** для всех курсоров, которые имеют отношение к редактируемой таблице.

Для составления запроса используются два метода: **rawQuery()** и **query()**, а также класс **SQLiteQueryBuilder**.

**Метод query()**

Для чтения данных используют вызов метода **query()**:

Cursor query (String table,

String[] columns,

String selection,

String[] selectionArgs,

String groupBy,

String having,

String sortOrder)

В метод **query()** передают семь параметров. Если какой-то параметр для запроса вас не интересует, то оставляете *null*:

* **table** — имя таблицы, к которой передается запрос;
* **String[] columnNames** — список имен возвращаемых полей (массив). При передаче *null* возвращаются все столбцы;
* **String whereClause** — параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение null возвращает все строки. Например: *\_id = 19 and summary = ?*
* **String[] selectionArgs** — значения аргументов фильтра. Вы можете включить ? в "whereClause"". Подставляется в запрос из заданного массива;
* **String[] groupBy** - фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается null;
* **String[] having** — фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;
* **String[] orderBy** — параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

Объект **Cursor**, возвращаемый методом **query()**, обеспечивает доступ к набору записей результирующей выборки. Для обработки возвращаемых данных объект **Cursor** имеет набор методов для чтения каждого типа данных — **getString()**, **getInt()** и **getFloat()**.

Чтобы получить все записи из нужных столбцов без условий, достаточно указать имя таблицы в первом параметре и строчный массив во втором. В остальных параметрах оставляем **null**:

Cursor cursor = db.query("CAT",

new String[] {"NAME", "DESCRIPTION"},

null, null, null, null, null);

Для запроса с условием используются третий и четвёртый параметр. Например, нам нужны записи, у которого имя кота будет Барсик:

Cursor cursor = db.query("CAT",

new String[] {"NAME", "DESCRIPTION"},

"NAME = ?",

new String[] {"Barsik"},

null, null, null);

Параметр **"NAME = ?"** означает, что столбец NAME должен содержать значение, указанное следующем параметре.

Множественные условия. Об этом говорилось выше.

Cursor cursor = db.query("CAT",

new String[] {"NAME", "DESCRIPTION"},

"NAME = ? OR DESCRIPTION = ?",

new String[] {"Murzik", "Nice"},

null, null, null);

Пример при использовании числовых значений, например, идентификатор первичного ключа (об этом тоже говорилось)

Cursor cursor = db.query("CAT",

new String[] {"NAME", "DESCRIPTION"},

"\_id = ?",

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

null, null, null);

Для сортировки используется последний параметр. Нужно указать нужный столбец, по которому будет проводиться сортировка. По алфавиту с буквы A - ASC, наоборот - DESC

Cursor cursor = db.query("CAT",

new String[] {"\_id", "NAME", "AGE"},

null, null, null, null,

**"NAME ASC"**);

Возможна сортировка по нескольким столбцам.

Cursor cursor = db.query("CAT",

new String[] {"\_id", "NAME", "DESCRIPTION"},

null, null, null, null,

"DESCRIPTION DESC, NAME");

Вы можете использовать функции SQL для составления запросов: AVG(), COUNT(), SUM(), MAX(), MIN() и др.

// Возвращает число котов в столбце count

// Аналог SELECT COUNT(\_id) AS COUNTER FROM CAT

Cursor cursor = db.query("CAT",

new String[] {"COUNT(\_id) AS counter"},

null, null, null, null, null);

Ещё один абстрактный пример подсчёта средней суммы:

Cursor cursor = db.query("MILK",

new String[] {"AVG(PRICE) AS price"},

null, null, null, null, null);

**rawQuery()**

Также существует метод **rawQuery()**, принимающий сырой SQL-запрос.

Метод **rawQuery()** - это сырой запрос, как есть, т.е. пишется строка запроса, как это обычно делается в SQL. Пример выглядит следующим образом:

Cursor cursor = getReadableDatabase().

rawQuery("select \* from todo where \_id = ?", new String[] { id });

Класс **SQLiteQueryBuilder** - удобный способ для построения запросов. Выбор за вами!

**Метод insert()**

Для вставки новой записи в базу данных SQLite используется метод **insert()**:

long insert (String table, String nullColumnHack, ContentValues values);

В метод **insert()** необходимо передать три параметра:

* **table** — имя таблицы, в которую будет вставлена запись;
* **nullColumnHack** — в базе данных SQLite не разрешается вставлять полностью пустую строку, и если строка, полученная от клиента контент-провайдера, будет пустой, то только этому столбцу явно будет назначено значение null;
* **values** — карта отображений (класс **Map** и его наследники), передаваемая клиентом контент-провайдера, которая содержит пары ключ-значение. Ключи в карте должны быть названиями столбцов таблицы, значения — вставляемыми данными.

Метод **insert()** возвращает идентификатор \_id вставленной строки или -1 в случае ошибки.

Для создания новой строки понадобится объект **ContentValues**, точнее, его метод **put()**, чтобы обеспечить данными каждый столбец. Объект **ContentValues**представляет собой пару **name/value** данных. Вставьте новую строку, передавая в метод **insert()**, вызванный в контексте нужной нам базы данных, имя таблицы и объект **ContentValues**

// Создайте новую строку со значениями для вставки.

ContentValues newValues = new ContentValues();

// Задайте значения для каждой строки.

newValues.put(COLUMN\_NAME, newValue);

[ ... Повторите для каждого столбца ... ]

// Вставьте строку в вашу базу данных.

myDatabase.insert(DATABASE\_TABLE, null, newValues);

**Метод update()**

Для обновления и удаления записей в базе данных используют соответственно методы **update()** и **delete()**:

int update (String table, ContentValues values, String whereClause, String[] whereArgs)

int delete (String table, String whereClause, String[] whereArgs)

В этих методах два последних параметра формируют SQL-выражение **WHERE** аналогично рассмотренному методу **query()** для чтения данных. Эти методы возвращают число модифицированных или удаленных строк.

Обновление строк также происходит с помощью класса **ContentValues**. Создайте новый объект **ContentValues**, используя метод **put()** для вставки значений в каждый столбец, который вы хотите обновить. Вызовите метод **update()** в контексте базы данных, передайте ему имя таблицы, обновленный объект**ContentValues** и оператор **WHERE**, указывающий на строку (строки), которую нужно обновить.

// Определите содержимое обновленной строки.

ContentValues updatedValues = new ContentValues();

// Назначьте значения для каждой строки.

newValues.put(COLUMN\_NAME, newValue);

[ ... Повторите для каждого столбца ... ]

String where = KEY\_ID + "=" + rowId;

// Обновите строку с указанным индексом, используя новые значения.

myDatabase.update(DATABASE\_TABLE, newValues, where, null);

Ещё пример с использованием выражения **WHERE**:

ContentValues barsikValues = new ContentValues();

// Помещаем значение "Clever" в колонку DESCRIPTION

barsikValuesValues.put("DESCRIPTION", "Clever");

// Обновляем колонку DESCRIPTION с новым значением "Clever"

// в таблице WHERE NAME = "Barsik"

myDatabase.update(DATABASE\_TABLE,

barsikValues,

"NAME = ?",

new String[] {"Barsik"});

В первом параметре метода мы указываем имя таблицы, которую хотим обновить.

Во втором параметре мы указываем значение, которое хотим обновить. Например, в колонке "DESCRIPTION" нужно обновить значение на "Clever".

В третьем параметре указывается условие для обновления. Выражение **"NAME = ?"** означает, что колонка **NAME** должна быть равна указанному значению. Знак вопроса является подстановочным символом и из четвёртого параметра берётся нужное значение.

Обратите внимание, что последний параметр является массивом. Вы можете использовать множественные условия для запроса. Например, нам нужно обновить запись в таблице, используя сразу две колонки для составления условия.

myDatabase.update(DATABASE\_TABLE,

murzikValues,

"NAME = ? OR DESCRIPTION = ?",

new String[] {"Murzik", "Nice"});

Данный код соответствует выражению **Where NAME = "Murzik" or DESCRIPTION = "Nice"**.

Для условий всегда используются строки, поэтому может понадобиться конвертация в некоторых случаях. Например, нужно использовать идентификатор таблицы со значением 1.

myDatabase.update(DATABASE\_TABLE,

newValues,

"\_id = ?",

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

Если в последних двух параметрах использовать **null**, то будут обновлены все записи в таблице.

myDatabase.update(DATABASE\_TABLE,

barsikValues,

null,

null);

**Метод delete()**

int delete (String table, String whereClause, String[] whereArgs)

Чтобы удалить строку, просто вызовите метод **delete()** в контексте базы данных, указав имя таблицы и оператор **WHERE**. В результате вы получите строки, которые хотите удалить:

myDatabase.delete(DATABASE\_TABLE, KEY\_ID + "=" + rowId, null);

Метод практически совпадает с методом **update()**.

Удалим запись с условием **WHERE NAME = "Murzik"**

myDatabase.delete(DATABASE\_TABLE,

"NAME = ?",

new String[] {"Murzik"});

Кроме вышеперечисленных методов, в этом классе также определены различные методы для выполнения других общих задач управления базой данных.

**Метод openOrCreateDatabase: Открытие и создание баз данных без использования SQLiteOpenHelper**

Вы можете открывать и создавать базы данных без помощи класса **SQLiteOpenHelper**, используя метод **openOrCreateDatabase()**, принадлежащий объекту**Context** вашего приложения. Получите доступ к базе данных в два шага. Сначала вызовите метод **openOrCreateDatabase()**, чтобы создать новую базу данных. Затем из полученного экземпляра базы данных вызовите **execSQL()**, чтобы выполнять команды на языке SQL, с помощью которых будут созданы таблицы и установлены отношения между ними.

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

private static final String DATABASE\_TABLE = "mainTable";

private static final String DATABASE\_CREATE =

"create table " + DATABASE\_TABLE + " ( \_id integer primary key autoincrement," +

"column\_one text not null);";

SQLiteDatabase myDatabase;

private void createDatabase() {

myDatabase = openOrCreateDatabase(DATABASE\_NAME, Context.MODE\_PRIVATE,

null);

myDatabase.execSQL(DATABASE\_CREATE);

}

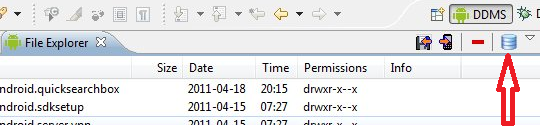
**Контент-провайдеры**

Контент-провайдеры поддерживают стандартный синтаксис запросов для чтения, изменения, вставки и удаления данных. Если необходимо предоставить доступ к своим данных для других приложений, существует два варианта:

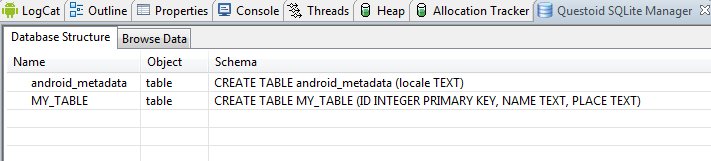
* создать собственный контент-провайдер, как подкласс класса ContentProvider;
* добавить данные к существующему провайдеру — если уже есть провайдер, который управляет теми же данными, и вы имеете разрешение для работы с этими данными.

**Плагин SQLiteManager для Eclipse**

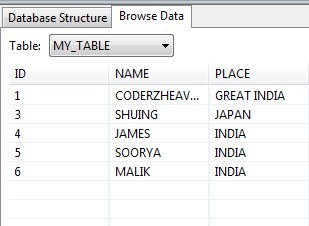
Существует плагин **SQLiteManager** для Eclipse, позволяющий видеть данные в базе. Скачайте библиотеку [com.questoid.sqlitemanager\_1.0.0.jar](http://www.coderzheaven.com/android1/com.questoid.sqlitemanager_1.0.0.jar) и скопируйте её в папку Eclipse/dropins. Перезагрузите Eclipse и откройте перспективу DDMS. Плагин будет находиться во вкладке File Explorer:



Щёлкнув по значку, мы увидим таблицы:



Вкладка Browse Data позволяет смотреть данные:



**Вставляем картинки в базу**

Вставлять картинки в базу данных не самая лучшая идея. Оставлю только для ознакомления. Так как SQLite не работает с изображениями напрямую, то нужно сконвертировать картинку в байтовый массив. А при извлечении нужно проделать обратную задачу - из байтового массива воссоздать изображение. Создадим вспомогательный класс.

package ru.alexanderklimov.test;

import java.io.ByteArrayOutputStream;

import android.graphics.Bitmap;

import android.graphics.BitmapFactory;

import android.graphics.Bitmap.CompressFormat;

public class Utilities {

// convert from bitmap to byte array

public static byte[] getBytes(Bitmap bitmap) {

ByteArrayOutputStream stream = new ByteArrayOutputStream();

bitmap.compress(CompressFormat.PNG, 0, stream);

return stream.toByteArray();

}

// convert from byte array to bitmap

public static Bitmap getImage(byte[] image) {

return BitmapFactory.decodeByteArray(image, 0, image.length);

}

}

Создадим таблицу следующим образом. Переменная **DATABASE\_TABLE** - строковая константа.

CREATE TABLE DATABASE\_TABLE (

\_id INTEGER PRIMARY KEY AUTOINCREMENT,

image\_data BLOB

);

Метод вставки в вашем классе-адаптере:

public long insert(byte[] image) {

return mDb.insert(DATABASE\_TABLE, null, createContentValues(image));

}

private ContentValues createContentValues(byte[] image) {

ContentValues cv = new ContentValues();

cv.put(COLUMN\_DATA, image);

return cv;

}