SQLite database in Android without content provider

by Benny Bottema - Friday, July 29, 2011

http://www.bennybottema.com/2011/07/29/sqlite-database-in-android-without-content-provider/

NOTE: This article demonstrates a way to set up a database for private in-app use only. If you need to provide content to the outside world, this article is not for you.

You want use a database inside your application to store static or dynamic content, right? Then perhaps you went to the Android docs on creating databases? The first thing you'll notice is that it mentions content providers and the NotePadProvider example of this principle. Once you start digging into that, you may very well get lost at first, consider how convoluted this system works, with uri's, matchers, paths and all. It gets messy quickly with lots of constants denoting your exposed content URI's on top of your database properties and mapping in between.

Toss all that out please, you don't need it.

Ignoring all the content provider stuff, it really becomes quite simple. All you need now are two classes that initialize, fill and query your database. You don't even need to change your AndroidManifest file. Let's start with creating the database.

- download demo source

Part 1: Database version management with SQLiteOpenHelper

Android has a useful class, SQLiteOpenHelper, that automates version management of your database and if needed creates it for you. By extending it, you can provide some specific details, such as a queries to create and fill your tables.

Here's an example:

```
public class AppleSQLiteOpenHelper extends SQLiteOpenHelper {

 private static final String VERSION = 1;

 public AppleSQLiteOpenHelper(Context context) {
  super(context, "ApplesDatabase", null, VERSION);
 }

 @Override
 public void onCreate(SQLiteDatabase db) {
  db.execSQL(
   "create table apples ( " +
    "_id integer primary key autoincrement, " +
    "name text not null, " +
    "category text null " +
   ");");
 }

 @Override
 public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersi
on) {
 }
}
```

First thing you'll notice is the use of a **version** value and the ***onUpgrade*** method. Android internally keeps track of which version of the database it has last deployed. The first time, when there is no database, the database is created using *onCreate*. From then on, each time you provide a higher version in the constructor, Android will call *onUpgrade*. This is a very neat mechanic to be able to release new versions of your application to the public and progressively update old database versions. You can not do a downgrade, however.

Here's an example where we rename the CATEGORY field:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersio
n) {
 if (oldVersion == 1) {
  db.execSQL("alter table apples rename column category to family");
 }
}
```

In the previous example, we're using the field _ID as id field, this is deliberate. The Cursor object, which will return our data needs a field names '_ID', either by column name or alias in a query.

Android's SQLite classes are generally database entity central instead of query central, meaning you fetch and put data by providing table names and columns names and values seperately and don't get to use your own queries (but it is possible). Because of this, it will be useful to keep a couple of public static constants denoting your names:

```
public class AppleSQLiteOpenHelper extends SQLiteOpenHelper {
 private static final int VERSION = 2;

 public static final String TABLE_APPLES = "apples";

 public static final String FIELD_ID = "_id";
 public static final String FIELD_NAME = "name";
 public static final String FIELD_FAMILY = "family";

 public AppleSQLiteOpenHelper(Context context) {
  super(context, "ApplesDatabase", null, VERSION);
 }

 @Override
 public void onCreate(SQLiteDatabase db) {
  db.execSQL(
   "create table " + TABLE_APPLES + " ( " +
    FIELD_ID + " integer primary key autoincrement, " +
    FIELD_NAME + " text not null, " +
    FIELD_FAMILY + " text null " +
   ");");
 }

 (..)
}
```

Now that we've got some handles to use when querying, let's do some querying.

Part 2: Using the database

To create our queries, Android has another helpful class, SQLiteQueryBuilder. This class helps us build queries using the table and field handles from the *HelloWorldSQLiteOpenHelper*.

Here's an example:

```
public class AppleProvider { // or AppleDao, AppleFetcher, AppleShmapp
le

 private SQLiteOpenHelper sqliteOpenHelper;

 public AppleProvider(Context context) {
  sqliteOpenHelper = new AppleSQLiteOpenHelper(context);
 }
 public Cursor getAllApples() {
  SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
  sqlBuilder.setTables(AppleSQLiteOpenHelper.TABLE_APPLES);
  SQLiteDatabase db = sqliteOpenHelper.getReadableDatabase();
  Cursor cursor = sqlBuilder.query(db, null, null, null, null, null, n
ull);
  return cursor;
 }
}
```

This returns a Cursor, which you can directly use in your view using a SimpleCursorAdapter. Personally, I don't like having a database cursor in my view, however; I like my data in a more usable format such as a List. This way the database won't be in use while rendering my view and I can perform some extra processing on the data returned:

```
public List getAllAppleNames() {
 SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
 sqlBuilder.setTables(AppleSQLiteOpenHelper.TABLE_APPLES);
 SQLiteDatabase db = AppleSQLiteOpenHelper.getReadableDatabase();
 Cursor cursor = sqlBuilder.query(db, null, null, null, null, null, nu
ll);
 List appleNames = CursorUtil.extractStringsFromCursor(cursor, AppleSQ
LiteOpenHelper.FIELD_NAME);
 return appleNames;
}

// inside CursorUtil class
public static List extractStringsFromCursor(Cursor cursor, String colu
mnName) {
 ArrayList cursorStrings = new ArrayList();
 int columnIndex = cursor.getColumnIndex(columnName);
 cursor.moveToFirst();
 while(!cursor.isAfterLast()) {
  cursorStrings.add(cursor.getString(columnIndex));
    cursor.moveToNext();
 }
 return cursorStrings;
}
```

I made my own utility method that extracts what I need. You could make it more generic ofcourse, but right now this is all we need. You can still directly use this list in your view, this time using a ListAdapter or some other form.

And there it is. All you need to use a database is a *SQLiteOpenHelper* to create and upgrade your database while hiding some name handles, and a another class to perform the queries. Technically, you can merge them into one, but then the methods of the *SQLiteOpenHelper* become exposed as well, which is messy.

- download demo source
