Inhaltsverzeichnis

1 SQLite - allgemeines

Wann immer ihr Daten strukturiert auf dem Android™ Gerät ablegen wollt, ist SQLite die erste Wahl. Egal, ob es gespeicherte Leveldaten für ein Spiel ist, ob es Userdaten sind oder irgendwelche andere Daten, die nach dem Schließen der App immer noch zur Verfügung stehen müssen, SQLite nimmt euch diese Arbeit ab.

Ich gehe davon aus, dass ihr die Unterlagen zum Gelb- und Orangegurt durchgearbeitet habt. Wenn nicht, solltet ihr das nachholen – vor allem das Thema „Threading" wird für die Anwendung von SQL notwendig sein. Weiterhin solltet ihr Grundkenntnisse im Umgang mit Datenbanken und vor allem mit SQL haben. Sollte dies nicht der Fall sein, kann ich euch unter www.codeconcert.de im Bereich „für Schüler" das Unterrichtsmaterial der 12. Klassen empfehlen, dort habe ich einiges zu diesem Thema zusammengetragen. Diese Unterlagen nutzen zwar nicht SQLite, sondern MySQL, wobei das zum einen beim Thema SQL allgemein keine Rolle spielt und zum anderen es durchaus sinnvoll ist, sich mit MySQL auch mal zu beschäftigen. Ansonsten – wie immer – solltet ihr versuchen eigene Ideen zu verwirklichen – dabei lernt ihr am meisten.
Solltet ihr die ersten Inhalte verpasst haben, so könnt ihr dies jederzeit nachholen. Sämtliche Informationen zum Gelbgurt und Orangegurt findet ihr im Netz unter www.codeconcert.de.

Bevor wir anfangen, noch ein paar Worte zu SQLite. Hierbei handelt es sich um eine Programmbibliothek für eine offene Datenbank, welche als „public domain" in den Umlauf gebracht wird. SQLite unterstützt Transaktionen, Trigger, Subselects (Unterabfragen) und ähnliche Features, welche im SQL-92 Standard definiert wurden. Aus diesem Grund wird SQLite sehr breit genutzt – in Browsern, vielen Handy Betriebssystemen und Anwenderprogrammen wie bspw. Skype.

Mit einer der größten Vorteile von SQLite ist die Tatsache, dass man sich kaum um Konfigurationen kümmern muss. Dies ist ein Hauptgrund, warum man es so einfach in eigenen Softwarepaketen einsetzen kann. Der User dieser Software merkt im Regelfall nicht, dass er gerade mit einer Datenbank arbeitet. Dies bedeutet aber auch, dass der Programmierer sich vorab über ein paar Details Gedanken machen muss. Dies werden wir in diesem Kurs vor allem beim Thema „Upgrade" behandeln.

Wir werden uns die Nutzung im Rahmen von Android ansehen. Hierbei setzen wir eine einfache Datenbank auf, speichern Daten und lesen sie wieder aus. Weiterhin werfen wir einen Blick auf die Upgradefunktion. Am Schluss werden wir die ganze Kommunikation in einen eigenen Thread verfrachten, da es in Android sinnvoll ist die Kommunikation in Threads durchzuführen, welche unabhängig vom UI Thread sind.

2 Die Bibliotheken

Die relevanten Klassen finden wir im Package „android.database".

Gehen wir kurz die einzelnen Klassen und dort ausgewählte Methoden durch. In einem späteren Kapitel werden wir dann die Elemente zu einem sinnvollen Programm zusammenstecken.

2.1 SQLiteOpenHelper

Beginnen wir mit der Klasse SQLiteOpenHelper. Diese ist im Wesentlichen dazu gedacht, Datenbanken zu erzeugen und zu managen. Um sie zu nutzen, sollte eine eigene Klasse von ihr abgeleitet werden:

```
public class MySQLiteHelper extends SQLiteOpenHelper {
```

Damit haben wir die Möglichkeit, unsere Funktionalität einzubetten. Der zweite Schritt ist, den Konstruktor zu bestücken:

```
public MySQLiteHelper(Context context) {
  super(context, "com.mypackage.myapp.MyDatabase", null, 1);
}
```

Dieser ruft eigentlich nur den Konstruktor der Superklasse (also SQLiteOpenHelper) auf. Hier werden vier Parameter benötigt:
Der erste Parameter ist „**context**": Dies ist der Applikationskontext, der der SQLiteOpenHelper eine Rerferenz auf die Laufzeitumgebung gibt. Im Regelfall ist das eine Referenz auf die Activity.

Für die saubere Integration in das Android Framework wird noch ein weiterer Konstruktor benötigt, welchen wir nicht selbst aufrufen, sondern der innerhalb von Android verwendet wird:

```
  public MySQLiteHelper(Context context, String name,
     CursorFactory factory, int version) {
    super(context, name, factory, version);
  }
```

Der nächste Parameter ist der **Datenbankname**, der möglichst eindeutig sein sollte, weshalb meistens der Packagename mit eingebaut wird (dies hat aber nichts mit dem tatsächlichen Java Package zu tun – es handelt sich hier nur um eine Möglichkeit, den Namen eindeutig zu wählen). Üblicherweise verpackt man den Namen in eine Konstante. Der dritte Parameter ist für eine **Cursorfactory** – dies benötigen wir zu diesem Zeitpunkt nicht und setzen ihn einfach auf „null". Der letzte Parameter dagegen ist wieder wichtig für uns, er gibt die **Versionsnummer** der Datenbank an. Die erste Version ist die 1. Alle Änderungen nach dem Publizieren der App müssen mit einer neuen Versionsnummer versehen sein. Es ist absolut wichtig, dass ihr genau dokumentiert, welche Versionsnummer welche Struktur aufweist, damit ihr in der Lage seid, von jeder Version auf die aktuelle Version einen Upgrade durchzuführen. Davon später aber mehr.

Nun kümmern wir uns um die Erstellung der Datenbank. Dies erfolgt beim ersten Aufruf der App und zwar genau beim ersten Öffnen der Datenbank – also nicht bei der Erzeugung unserer Helper Klasse, sondern tatsächlich beim ersten Zugriffsversuch. Android prüft also, ob die Datenbank bereits existiert. Wenn nicht, wird die Methode „onCreate()" aufgerufen. Dort müssen alle Create Table Statements liegen. Wir werden weiter unten ein Beispiel für dieses Statement ansehen.

```
public void onCreate(SQLiteDatabase db) {
  // Create Table Aktivitäten
}
```

Sollte allerdings die Datenbank bereits vorhanden sein, wird geprüft, ob die gefundene Version im Android System der der aktuellen Software entspricht. Ist dies nicht der Fall, wird „onUpgrade()" aufgerufen. Hier steht der Code, welcher die existierenden Daten sichert und die Datenbank umstrukturiert.

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
  // Upgradeaktivitäten
}
```

Folgende Vorgehensweise für die Upgradeaktivitäten könnten – je nach Situation – sinnvoll sein (eventuell in einer Transaktion eingebettet):

- Sichern der Daten (bspw. durch Alter table und einem Rename)
- Extrahieren der Daten aus der DB und Struktur an das neue DB-Layout anpassen.
- Erzeugen der neuen Tabelle bzw. Tabellen (bspw. durch Aufruf von "onCreate(db)")
- Beladen der Daten mit der neuen Struktur.
- Löschen der gesicherten Tabellen

Wichtig ist, dass wir auf dem System jede vorher existierende Version finden könnten. Wenn wir bspw. bei der Datenbank gerade die 4. Version erzeugen, müssen wir damit rechnen, dass folgende Situationen vorherrschen:

- Datenbank noch nicht vorhanden
- Upgrade von Version 1 auf Version 4
- Upgrade von Version 2 auf Version 4
- Upgrade von Version 3 auf Version 4

Es ist also wichtig, dass wir immer wissen, wie die einzelnen Versionen der Datenbank aussehen, damit wir beim Upgrade die richtigen Schritte für die Datenkonvertierung vornehmen.

Nun noch eine kurze Beschreibung der für uns wichtigsten Methoden, welche wir bei der Nutzung unserer abgeleiteten Klasse benötigen werden:

Methode:	**Funktion:**
getWritableDatabase()	Diese Methode erzeugt eine Datenbank (sofern sie noch nicht existiert) und öffnet diese. Sie wird sowohl für Schreib-, als auch für Lesezugriffe verwendbar sein. Sollten Probleme (wie bspw. „Disk full“) entstehen, wird dies eine Exception zur Folge haben. Die Methode gibt ein SQLiteDatabase Objekt zurück.
getReadableDatabase()	Diese Methode erzeugt eine Datenbank (sofern sie noch nicht existiert) und öffnet diese. Sie wird (entgegen dem, was der Name erwarten lässt) sowohl für Schreib-, als auch für Lesezugriffe verwendbar sein. Lediglich bei „Disk full“ wird sie ausschließlich für Lesezugriffe geöffnet. Die Methode gibt ein SQLiteDatabase Objekt zurück.
close()	Schließt das SQLiteDatabase Objekt. Diese Methode sollte auf jeden Fall aufgerufen werden, wenn der Datentransfer von/zu der Datenbank erledigt ist.

Anmerkung: Die Methoden getWritableDatabase und getReadableDatabase sollten immer über einen eigenen Thread ausgeführt werden, da bei einem eventuellen Datenbankupgrade die Verarbeitungszeit für den UI Thread zu lange sein könnte.

2.2 SQLiteDatabase

Dies ist das Objekt, welches den eigentlichen Zugriff zur Datenbank ermöglicht. Sie beinhaltet eine Vielzahl von Methoden, von denen wir uns an dieser Stelle auch nur die wichtigsten heraussuchen:

Methode:	**Funktion:**
query(…)	Die Query ist eine einfache Datenbankabfrage, bei der diverse Parameter für die Datenbankabfrage vorgesehen sind. Hier gibt es diverse Überladungen, so dass wir weiter unten noch einfache Beispiele uns ansehen werden. Der Rückgabewert einer Query ist vom Typ „Cursor“ (siehe Unten)
rawQuery(sql, selectionArgs)	Im Gegensatz zur “query” beinhaltet die rawQuery ganz normales SQL, wobei sie als Prepared Statement ausgelegt ist, so dass im SQL die einzelnen SQL Parameter mit einem “?” platziert werden und die Argumente als String Array in einem zweiten Parameter angegeben werden.
insert(table, nullColumnHack, values)	Mit dieser Methode wird ein INSERT Statement ausgeführt. Der Parameter „table“ wird der Tabellenname angegeben, „nullColumnHack“ wird benötigt, um „null“ Zeilen einzufügen (wobei wir dies derzeit nicht brauchen und den Parameter stets „null“ setzen) und der dritte Paramter ist ein Objekt vom Typ „ContentValues“, welches die Werte als Key/Value Pairs ablegt. Der Key ist hierbei der Spaltenname und Value entsprechend der (typisierte) Wert. Der Rückgabewert ist die ColumnID der eingefügten Zeile, was gleich einem Feld wäre, welches mit „id primary key autoincrement“ deklariert wurde.
execSQL(sqlStatement)	Diese Methode führt einfach nur das SQL Statement aus, welches als Parameter übergeben wird. Sie wird für SQL Statements wie bspw. DROP TABLE oder CREATE TABLE genutzt. Die Methode ist explizit nicht als Ersatz für SELECT, INSERT, UPDATE oder DELETE Statements gedacht, hierfür gibt es eigene Methoden (siehe API, bzw. „rawQuery“, „query“ oder „insert“).

2.3 Cursor

Bei Abfragen auf die Datenbank wird ein Cursor zurückgegeben. Dieser läuft durch sämtliche Datensätze und ermöglicht es uns, die Daten in eigene Objekte abzuspeichern. Ein Cursor liefert also die Daten sequenziell an das Programm ab, nicht als Ganzes. Man kann sich den Cursor somit als eine Art Zeiger vorstellen, der über die Ergebnismenge geschoben wird.

Folgende Methoden sind für uns die wichtigsten:

Methode:	**Funktion:**
moveToFirst()	Verschiebt den Cursor auf die erste Position. Die Methode gibt ein „false“ zurück, wenn keine Datensätze vorhanden sind.
moveToNext()	Verschiebt den Cursor auf die nächste Position. Die Methode gibt ein „true” zurück, wenn eine nächste Position existiert, ansonsten ein “false”.
getCount()	Gibt die Anzahl der ermittelten Datensätze der Abfrage zurück.
getPosition()	Gibt die aktuelle Position des Cursors als int Wert zurück.
close()	Schließt den Cursor. Dies sollte immer nach dem Auslesen des letzten Datensatzes erfolgen.
getBlob(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen byte Array.
getDouble(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen double Wert.
getFloat(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen float Wert.
getInt(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen int Wert.
getLong(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen long Wert.
getShort(columnIndex)	Liest die Spalte an der Position „columnIndex“ aus und konvertiert sie in einen short Wert.
getString(columnIndex)	Liest die Spalte an der Position „columnIndex“ als String aus.

2.4 ContentValues

Die Hilfsklasse “ContentValues” ist für das Einfügen von neuen Datensätzen hilfreich. Ein “insert” Statement benötigt neben dem Tabellennamen auch die einzufügenden Werte, welche in einem ContenValues Objekt verpackt werden. Hierzu wird das Objekt erstellt und mit Hilfe der folgenden Methode bewirtschaftet:

Methode:	**Funktion:**
put(key, value)	Platziert einen Wert „value“ mit dem Key „key“. Als Key ist der Spaltenname der Tabelle zu setzen und als Value der typisierte Wert, der in die entsprechende Spalte geschrieben werden soll. Folgende Typen sind vorgesehen: Byte, Integer, Float, Short, byte[], String, Double, Long, Boolean – also alles Objekte.

3 Best Practice

Es wird empfohlen, eine sogenannte „Contract Klasse“ zu erstellen, in der die Tabellennamen und Spaltennamen eingetragen werden. Insofern hat man die relevanten Informationen an einer Stelle konzentriert. Diese Klasse wird als „abstract“ deklariert und beinhaltet statische abstrakte innere Klassen pro Tabelle. Weiterhin wird ein leerer Konstruktor empfohlen, damit keine versehentlichen Instantiierungen vorgenommen werden.

```
public final class MyContractClass {
  public MyContractClass() {}

  public static abstract class MyTable1 implements BaseColumns {
    public static final String TABLE_NAME = "MyTab1";
    public static final String COL_MYCOL1 = "MyCol1";
    public static final String COL_MYCOL2 = "MyCol2";
    // …
  }
```

```
  public static abstract class MyTable2 implements BaseColumns {
    public static final String TABLE_NAME = "MyTab2";
    public static final String COL_MYCOL1 = "MyCol1";
    public static final String COL_MYCOL2 = "MyCol2";
    // …
  }
}
```

Die inneren Klassen implementieren die BaseColumns. Dies wird empfohlen, damit die Tabellen mit dem Android Framework sauber zusammenarbeiten (bspw. für ContentProvider). Hierdurch werden die beiden Spalten _COUNT und _ID einheitlich benannt. Für uns ist vor allem _ID interessant, weil hierdurch eine Spalte existiert, welche eine eindeutige ID für jeden Dateneintrag entsteht.

Anmerkung: Die beiden Spalten werden nicht automatisch erstellt, sondern _ID muss im CREATE TABLE Statement selbst eingefügt werden und _COUNT wird von der Query belegt. Es geht hier also nur um eine einheitliche Namensgebung der Spalten.

4 Konkretes Beispiel

Sehen wir uns das Ganze in einem Konkreten Beispiel an. Wir wollen eine Tabelle namens „Users“ erschaffen, welche folgende Struktur aufweist:

_ID	**NameValue**
1	Peter.Mueller
2	Maria.Schmidt
3	Petra.Schneider

Wir haben also zwei Spalten, von denen eine die Standard ID aus BaseColumns ist und ein Feld, welches eine Namensinformation beinhaltet.

4.1 Die Contract Klasse

Beginnen wir mit unserer Contract Klasse. Hier werden die Namen unserer Tabellenstruktur hinterlegt. Da wir nur eine Tabelle mit zwei Spalten haben ist der Contract relativ kurz.

```
public final class ContractExample {
  public ContractExample() {}

  public static abstract class TabUsers implements BaseColumns {
     public static final String TABLE_NAME = "Users";
     public static final String COL_NAMEVAL = "NameValue";
  }
}
```

4.2 Die Helper Klasse

Nun können wir die Helper Klasse erstellen, welche sich um die Datenbankstrukturen kümmern. Idealerweise verpacken wir die Create Statements in Konstanten, damit wir die Struktur sauber an einer Stelle vorfinden.

```
public class DbHelperExample extends SQLiteOpenHelper {
  public static final int DATABASE_VERSION = 1;
  public static final String DATABASE_NAME = "com.example.mydb";

  public static final String SQL_CREATE_TABUSERS =
    "CREATE TABLE " + ContractExample.TabUsers.TABLE_NAME + "( " +
    ContractExample.TabUsers._ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
    + ContractExample.TabUsers.COL_NAMEVAL + " TEXT)";

  public DbHelperExample (Context context) {
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
  }
```

```
  public DbHelperExample(Context context, String name,
     CursorFactory factory, int version) {
    super(context, name, factory, version);
  }

  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(SQL_CREATE_TABUSERS);
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    // Version 1 => no upgrade necessary
  }
}
```

4.3 Die SQL Statements

4.3.1 Daten rein

Nun kommen die eigentlichen Statements. Hier können wir entweder die Helper Klasse um die SQL Abfragen erweitern, oder wir erstellen uns eine eigene Klasse (bspw. „MySqlStatements“), in der wir die Abfragen platzieren. Jede Abfrage packen wir in eine eigene Methode, so dass wir den Überblick behalten. Beginnen wir nun damit ein Statement zu schreiben, welches Datenwerte in die Tabelle schreibt:

```
public static long insertNewUser(SQLiteDatabase db, String userName) {
  ContentValues values = new ContentValues();
  values.put(ContractExample.TabUsers.COL_NAMEVAL, userName);
  long newRowId = db.insert(
     ContractExample.TabUsers.TABLE_NAME, null, values);
  return newRowId;
}
```

Diese Methodik ist die einfachste, um Daten zu speichern. Wer es umfangreicher haben möchte, kann auch mit Hilfe der rawQuery Methode durchgeführt werden. Diese ist zwar eigentlich für Abfragen gedacht, funktioniert aber auch für INSERT. Dies sollte aber wirklich nur dann durchgeführt werden, wenn die insert Methode an ihre Grenzen stößt.

4.3.2 Daten raus

Um an die Daten wieder ranzukommen wenden wir erstmal die einfache rawQuery an:

```
public static String getSingleUserName(SQLiteDatabase db, int userId) {
  String userName = null;
  final String SQL_GET_SINGLE_USER_NAME =
       "SELECT " + ContractExample.TabUsers.COL_NAMEVAL +
       " FROM " + ContractExample.TabUsers.TABLE_NAME +
       " WHERE " + ContractExample.TabUsers._ID + " = ?";
  String[] selectionArgs = new String[] {String.valueOf(userId)};
  Cursor cursor = db.rawQuery(SQL_GET_SINGLE_USER_NAME,
    selectionArgs);
  if (cursor != null) {
    if (cursor.moveToFirst()) {
      do {
        userName = cursor.getString(0);
      } while (cursor.moveToNext());
    }
    cursor.close();
  }
```

```
  return userName;
}
```

Ich habe die Methode so aufgebaut, dass sie auch einfach für Abfragen von mehreren Ergebnissätzen umgebaut werden kann. Gehen wir die Methode kurz durch. Zuerst erzeugen wir uns eine Rückgabevariable, welche mit „null“ initialisiert wird, damit wir bei leeren Ergebnismengen die „null“ finden und entsprechend interpretieren können.

```
  String userName = null;
```

Danach erzeugen wir uns unser Statement. Dies kann auch als statische finale Konstante in der Klasse abgelegt werden.

```
  final String SQL_GET_SINGLE_USER_NAME =
       "SELECT " + ContractExample.TabUsers.COL_NAMEVAL +
       " FROM " + ContractExample.TabUsers.TABLE_NAME +
       " WHERE " + ContractExample.TabUsers._ID + " = ?";
```

Das Fragezeichen in der Abfrage „?“ steht für den Parameter, welcher über die „selectionArgs“ übergeben wird.

```
  String[] selectionArgs = new String[] {String.valueOf(userId)};
```

Nun wird die Abfrage ausgeführt und das Ergebnis in das Cursorobjekt übergeben:

```
  Cursor cursor = db.rawQuery(SQL_GET_SINGLE_USER_NAME,
    selectionArgs);
```

Sollten wir einen validen Cursor bekommen haben (also != null) und der nicht leer sein (also cursor.moveTo-First() false ergibt), so können wir mit der Verarbeitung anfangen:

```
  if ((cursor != null) {
    if (cursor.moveToFirst())) {
```

Nun lesen wir alle Positionen des Cursors aus. Da wir das Select auf einen Primary Key einschränken, werden wir lediglich einen Datensatz zu erwarten haben. Dieser wird in die Variable „userName“ geschrieben und an den Aufrufer zurückgeliefert.

```
      do {
        userName = cursor.getString(0);
      } while (cursor.moveToNext());
    }
    cursor.close();
  }
  return userName;
```

Eine Alternative bieten die „query“ Methoden. Diese werden nicht mit einem SQL Statement parametriert, sondern sie werden Konfiguriert. Hier ein Beispiel anhand des folgenden query Befehls:

```
query (String table, String[] columns, String selection,
  String[] selectionArgs, String groupBy, String having,
  String orderBy, String limit)
```

Folgende Bedeutung haben die einzelnen Parameter:

Parameter:	**Funktion:**
table	Name der Tabelle
columns	Array mit allen Spaltennamen, welche benötigt werden.
selection	Filter entsprechend der WHERE Klausel eines SELECT Befehls. Werte können mit einem „?“ ersetzt werden. Wichtig ist auch, dass das Schlüsselwort „WHERE“ hier nicht auftaucht, da es intern eingesetzt wird.
selectionArgs	Für jedes „?“ in dem selection String wird hier ein Wert erwartet.
groupBy	GROUP BY Parameter wie sie in einem SQL Statement vorkommen (ohne die Schlüsselwörter GROUP und BY)
having	HAVING Parameter für das GROUP BY wie sie in einem SQL Statement vorkommen (ohne das Schlüsselwort HAVING)
orderBy	ORDER BY Parameter wie sie in einem SQL Statement vorkommen (ohne die Schlüsselwörter ORDER und BY)
limit	Limitierung der Datensätze, wie in SQL (also mit Ober- und Untergrenze)

Ein „null“ Wert für die einzelnen Parameter wirken sich aus, als würde er nicht im SQL Statement existieren.

Folgend das gleiche Statement wie es oben mit „rawQuery“ erstellt wurde, entsprechend mit „query“:

```
Cursor cursor = db.query(ContractExample.TabUsers.TABLE_NAME,
  new String[] {ContractExample.TabUsers.COL_NAMEVAL},
  ContractExample.TabUsers._ID +" = ?",
  new String[] {String.valueOf(userId)},
  null, null, null, null);
```

Neben der oben angesprochenen query Methode bietet uns die Klasse SQLiteDatabase noch diverse weiter. Ich verweise an dieser Stelle einfach mal an die API Dokumentation im Netz. Für den SQL „Experten“ wird die rawQuery wohl die besten Dienste leisten, wobei dies jeder für sich selbst entscheiden muss.

4.3.3 Daten ändern

Updates werden ebenfalls über eine eigene SQLiteDatabase Methode erledigt. Hierfür sehen wir uns mal folgendes Beispiel an:

```
public static boolean updateUser(SQLiteDatabase db, String newUserName,
int userId) {
  ContentValues values = new ContentValues();
  values.put(ContractExample.TabUsers.COL_NAMEVAL, newUserName);
  String whereClause = ContractExample.TabUsers._ID + " = ?";
  String[] whereArgs = new String[] {String.valueOf(userId)};

  int noOfRows = db.update(ContractExample.TabUsers.TABLE_NAME,
    values, whereClause, whereArgs);
  return noOfRows > 0;
}
```

Da ein Update eine Kombination eines INSERTS und eines SELECTS ist, finden wir Elemente dieser beiden Aktionen auch im UPDATE wieder.

Der obere Teil entspricht der Wertfestlegung eines INSERTS:

```
  ContentValues values = new ContentValues();
  values.put(ContractExample.TabUsers.COL_NAMEVAL, newUserName);
```

Danach folgt die Festlegung der WHERE Klausel:

```
  String whereClause = ContractExample.TabUsers._ID + " = ?";
  String[] whereArgs = new String[] {String.valueOf(userId)};
```

Das Update liefert uns dann die Anzahl der veränderten Zeilen an, welche wir in diesem Beispiel als boolean Rückgabewert der Methode umfunktionieren, so dass „true“ zurückgegeben wird, wenn mindestens eine Zeile verändert wurde, ansonsten „false“.

4.3.4 Daten löschen

Um Datensätze wieder aus der Tabelle zu entfernen, bietet SQLiteDatabase das delete Statment:

```
public static boolean deleteUser(SQLiteDatabase db, int userId) {
  String whereClause = ContractExample.TabUsers._ID + " = ?";
  String[] whereArgs = new String[] {String.valueOf(userId)};

  int noOfRows = db.delete(ContractExample.TabUsers.TABLE_NAME,
    whereClause, whereArgs);
  return noOfRows > 0;
}
```

Wie zu erwarten war, benötigt der Löschbefehl ebenfalls eine WHERE Klausel um die zu löschenden Datensätze zu identifizieren. Alles andere sollte für uns nun selbsterklärend sein.

4.4 Die Nutzung in userer App

Nun haben wir alle Voraussetzungen geschaffen, SQLite zu nutzen. Innerhalb unserer App können wir nun die Helper Klasse Instanziieren. Ein guter Ort hierfür ist die onCreate Methode der Activity:

```
private DbHelperExample myDbHelper = null;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  this.myDbHelper = new DbHelperExample(this);
  // evtl. Löschen von Tabellen während Entwicklungsphase, wenn nötig
  // weiterer Code für onCreate…
}
```

Anmerkung: Es hat sich als sinnvoll erwiesen während der Entwicklungsphase der Tabellenstruktur sich ein Statement vorzuhalten, welches die Tabelle komplett löscht. Dies wird gleich nach der Instanziierung der Helperklasse ausgeführt, sofern die Tabellenstruktur angepasst werden muss. Folgender Code würde eine Tabelle löschen: db.execSQL("DROP TABLE IF EXISTS " + ContractExample.TabUsers.TABLE_NAME*);*

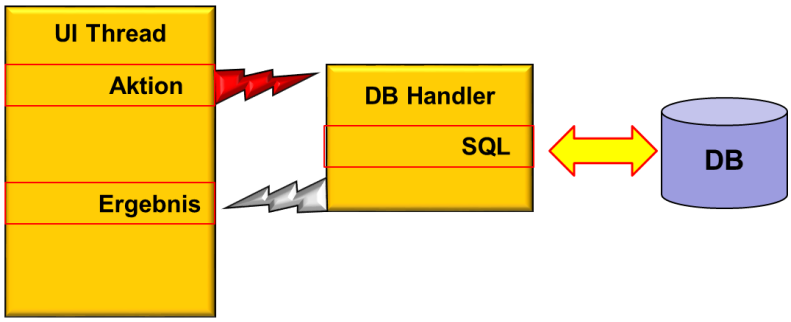
Im verarbeitenden Code soll nun ein Statement ausgeführt werden (bspw. getriggert durch einen Button-Klick). Die Nutzung unserer Datenbank würde somit wie folgt aussehen (sofern wir die insert Methode in unserer Helperklasse eingebaut haben):

```
SQLiteDatabase db = this.myDbHelper.getWritableDatabase();
DbHelperExample.insertNewUser(db, "Irgendeinname");
this.myDbHelper.close();
```

Der oben stehende Code sollte tunlichst in einem eigenen Thread laufen. Zwar ist das insertNewUser Statement im Regelfall schnell genug, um auf dem UI Thread zu laufen, was zwar programmiertechnisch „unsauber“ ist aber funktioniert, das eigentliche Problem ist die Methode „getWritableDatabase()“. Bei einem Upgrade würde dies die „onUpgrade()“ Methode durchlaufen und diese würde je nach Datenmenge durchaus zu einem ANR Fehler führen.

5 Einbetten in einen Thread

Wie mehrfach erwähnt, müssen wir für einen sauberen Betrieb unserer App die Datenbankkommunikation in einen eigenen Thread verpacken.



Wir werden die Klasse, welche den Thread ausführt „DbHandlerExample“ nennen. Hierbei sind folgende Themen zu berücksichtigen:

- Der DbHandlerExample benötigt eine Referenz auf die Activity, um den Rücktransport der Ergebnisdaten zu ermöglichen.
- Weiterhin benötigt der DbHandlerExample eine Referenz auf die SQL Helperklasse. Hierüber werden die SQL Befehle und vor allem das Öffnen der Datenbank realisiert.
- Die Klasse braucht eine Möglichkeit, die Eingangsdaten für die SQL Befehle (also welcher Befehl und welche Parameter) zu übermitteln.
- Ebenso benötigen wir einen Kanal, um die Ergebnisdaten an den UI Thread zurückzuliefern.

Wie das Ganze aussehen kann, werden wir uns jetzt anhand eines Beispiels ansehen. Beginnen wir gleich mit unserem Handler. Vorab der Code als Ganzes:

```
public class DbHandlerExample implements Runnable {
  private DbHelperExample myDbHelper = null;
  private MyActivity myActiviy = null;
  private String[] sqlInputData = null;
  private String[] sqlOutputData = null;
  private boolean isBusy = false;
  private int currentSqlCommand = -1;
  public static final int SQL_CMD_NEW_USER = 0;

  public DbHandlerExample(DbHelperExample myDbHelper,
    MyActivity myActivity) {
    this.myDbHelper = myDbHelper;
    this.myActiviy = myActivity;
  }

  public synchronized boolean setNewSqlAction(int newSqlCommand,
    String[] sqlData) {
    if (this.isBusy) {
      return false;
    }
    this.isBusy = true;
    this.currentSqlCommand = newSqlCommand;
    this.sqlInputData = sqlData;
    return true;
  }

  @Override
  public void run() {
    sqlOutputData = null;
    SQLiteDatabase db = this.myDbHelper.getWritableDatabase();

    switch (this.currentSqlCommand) {
    case SQL_CMD_NEW_USER:
      long newUserId = DbHelperExample.insertNewUser(db,
        this.sqlInputData[0]);
```

```
        this.sqlOutputData = new String[] {String.valueOf(newUserId)};
        break;
      // weitere Befehle
      }

      this.myDbHelper.close();
      this.myActiviy.runOnUiThread(new Runnable() {
        public void run() {
          myActiviy.handleSqlResponse(currentSqlCommand, sqlOutputData);
        }
      });
      this.isBusy = false;
    }

    public boolean isBusy() {
      return isBusy;
    }
}
```

Gehen wir den Code mal durch. Zuerst wird die Klassendeklaration durchgeführt. Da die Klasse innerhalb eines eigenen Therads ausgeführt werden muss, wird das „Runnable“ Interface implementiert werden. Die dadurch zu realisierende „run“ Methode wird weiter unten besprochen.

```
public class DbHandlerExample implements Runnable {
```

Danach werden die Instanzvariablen deklariert. Zuerst die beiden Referenzen auf die Activity und der Helperklasse. Diese werden später über den Konstruktor bewirtschaftet:

```
  private DbHelperExample myDbHelper = null;
  private MyActivity myActiviy = null;
```

Danach folgen die Variablen für die Kommunikation zwischen der Activity und dem dbHandler. Die Input/Output Data Variablen werden die eigentlichen Daten beinhalten. Bspw. wird in dem sqlInputData die ID eines gesuchten Users eingetragen und in sqlOutputData das Queryergebnis (also für unser Beispiel der Name des Users) geschrieben, damit die Information an die Activity gesendet werden kann. Weiterhin wurde noch eine Variable für eine Blockierung von 2 parallelen Anfragen implementiert und schließlich noch eine Integervariable, in der das angefragte SQL Kommando (als Zahl verschlüsselt) hinterlegt wird.

```
  private String[] sqlInputData = null;
  private String[] sqlOutputData = null;
  private boolean isBusy = false;
  private int currentSqlCommand = -1;
  public static final int SQL_CMD_NEW_USER = 0;
```

Der Konstruktor ist weitestgehend selbsterklärend – es werden lediglich die Referenzen für die Activity und der Helperklasse übergeben:

```
  public DbHandlerExample(DbHelperExample myDbHelper,
    MyActivity myActivity) {
    this.myDbHelper = myDbHelper;
    this.myActiviy = myActivity;
  }
```

Danach folgt die Methode, welche eine neue SQL Aktion einstellt. Die relevanten Informationen der Kommando ID und der Eingangsdaten werden als Parameter übergeben:

```
  public synchronized boolean setNewSqlAction(int newSqlCommand,
    String[] sqlData) {
```

Danach wird geprüft, ob die Klasse gerade ein Statement ausführt. Dies ist eine reine Sicherheitsmaßnahme. Eigentlich muss die aufrufende Struktur bereits dafür sorgen, dass keine zwei parallelen Aktionen laufen. Sollte keine Aktion laufen, so wird das busy Flag auf „true“ gesetzt, damit die Sperre nun aktiv ist.

```
    if (this.isBusy) {
      return false;
    }
    this.isBusy = true;
```

Nun werden die Kommando- und Dateninformationen übergeben. Der Returnwert „true“ indiziert, dass alles OK ist – sprich keine Aktion aktuelle läuft.

```
    this.currentSqlCommand = newSqlCommand;
    this.sqlInputData = sqlData;
    return true;
  }
```

Jetzt fehlt nur noch die eigentliche Arbeitsmethode „run“. Hier wird zuerst das Output Datenarray auf null gesetzt, damit bei eventuellen Fehlern die Acitivty anhand des „null“ Wertes dies feststellen kann. Danach wird aus der Helperklasse eine Referenz auf die Datenbank geholt. Dies bedeutet, dass eventuelle CREATE Statements bzw. die Upgrade Methode nun laufen würden – also innerhalb des Threads.

```
  public void run() {
    sqlOutputData = null;
    SQLiteDatabase db = this.myDbHelper.getWritableDatabase();
```

Nun können wir anhand des aktuell eingestellten SQL Kommandos die Daten korrekt extrahieren und die passende Methode der Helperklasse ausführen. Damit die Sache übersichtlicher wird, verpacken wir die einzelnen IDs der Kommandos in Konstanten (hier habe ich zur Demonstration lediglich ein Kommando vorgesehen „SQL_CMD_NEW_USER“ soll entsprechend einen neuen User anlegen. Also prüfen wir den Wert mit einem switch/case Statement. Wir extrahieren die Daten aus dem String Array, rufen die passende Methode auf und verpacken das Ergebnis in das Outputarray:

```
    switch (this.currentSqlCommand) {
    case SQL_CMD_NEW_USER:
      long newUserId = DbHelperExample.insertNewUser(db,
        this.sqlInputData[0]);
      this.sqlOutputData = new String[] {String.valueOf(newUserId)};
      break;
    // weitere Befehle
    }
```

Nun ist der Befehl abgeschlossenen; wir können also die Datenbank wieder schließen:

```
    this.myDbHelper.close();
```

Wie im Orangegurt zum Thema „Threads“ beschrieben, rufen wir nun über eine Queue die relevante Activity Methode auf – ebenfalls in einem neuen Thread. Diese Methode erwartet das SQL Kommando, welches ausgeführt wurde und die Ergebnisdaten im Outputarray:

```
    this.myActiviy.runOnUiThread(new Runnable() {
      public void run() {
        myActiviy.handleSqlResponse(currentSqlCommand, sqlOutputData);
      }
    });
```

Zum Schluss wird das Busyflag wieder auf "false" gesetzt, damit der nächste Befehl verarbeitet werden kann:

```
    this.isBusy = false;
```

Fahren wir nun mit der Erweiterung der Activity fort. Dort benötigen wir eine Instanzvariable mit unserem DbHandlerExample Objekt. Diese wird in der „onCreate" Methode mit dem Objekt belegt.

```
private DbHandlerExample dbHandler = null;

protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  this.myDbHelper = new DbHelperExample(this);
  this.dbHandler = new DbHandlerExample(this.myDbHelper, this);
  // weiterer Code für onCreate…
}
```

Nun können wir die Stelle realisieren, in der ein Statment antriggern, was wir in einer eigenen Methode machen können. Hierzu wird eine neue SQL Aktion gesetzt (da wir nur das Anlegen eines neuen Users umgesetzt haben, können wir auch nur dies Funktionalität antriggern). Den Parameter des Usernamens legen wir in dem Stringarray an und übergeben das Ganze in die Handlerklasse. Wenn die Methode „true" zurückgibt, dann wurden die Werte korrekt übernommen und wir können einen neuen Thread starten, der die Klasse übernimmt und ihn starten:

```
public void startNewUserCommand() {
  if (this.dbHandler.setNewSqlAction(DbHandlerExample.SQL_CMD_NEW_USER,
    new String[] {"Paul.Miller"})) {
    new Thread(this.dbHandler).start();
  } else {
    // error message
  }
}
```

Die Fehlermeldung habe ich mir hier gespart – dass muss die App lösen; bspw. mit einem eigenen Message Fragment oder einem Toast. Jetzt fehlt nur noch die Methode, welche die Responseinformationen entgegennimmt (diese wurde ja bereits in der „run" Methode angesprochen). Hier wird abhängig von dem SQL Kommando und den Responsedaten die Weiterverarbeitung angetriggert – bspw. Erfolgsmeldung an den User etc.:

```
public void handleSqlResponse(int sqlCommand, String[] responseData) {
  switch (sqlCommand) {
  case DbHandlerExample.SQL_CMD_NEW_USER:
    // whatever has to be done
    break;
  }
}
```

Anmerkung: Wenn es programmtechnisch mehrere SQL Befehle sequenziell abzuarbeiten, dann kann man auch in der Handlerklasse eine Arraylist implementieren, welche alle Befehle und Daten aufnimmt. Diese kann in „run" in einer While-Schleife abgearbeitet werden. Somit schaffen wir uns eine Art LIFO Buffer, der die

SQL Aktionen triggert. Da die Ergebnisverarbeitung in der Activity dann aber durchaus kompliziert werden kann, sollten wir uns diese Option genau überlegen.

6 Der Upgrade

Gehen wir nun davon aus, dass wir unsere kleine App mit der Datenbankversion 1 veröffentlicht haben. Nach einer Zeit erkennen wir, dass wir unsere Datenbank anpassen müssen, so dass wir die Daten nun wie rechts dargestellt ablegen müssen.

_ID	FirstName	LastName
1	Peter	Mueller
2	Maria	Schmidt
3	Petra	Schneider

Insofern müssen wir unseren Code nun ändern. Zum einen müssen sämtliche Statements angepasst werden, um auf die neue Datenbankstruktur Rücksicht zu nehmen. Ich verzichte hier darauf, die einzelnen Methoden nochmals vorzustellen – das Einzige, was wir uns merken müssen ist, dass wir die Methoden derart gestalten, als hätte die alte Struktur mit nur zwei Spalten nicht existiert. Die große Ausnahme ist die „onUpgrade“ Methode. Hier müssen wir nun auf die einzelnen Versionsupgrades Rücksicht nehmen. Für unsere Zwecke reicht aber der Upgrade von Version 1. Da wir in Zukunft auf verschiedene Upgrades achten müssen, spendieren wir jedem Level eine eigene Methode. Der untenstehende Code müsste für ein reales Projekt noch etwas stabiler gestaltet werden (bspw. zuerst die _TMP Tabelle löschen, bevor sie über RENAME erzeugt wird), doch für uns soll das unten gezeigte Level erst mal reichen:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
  switch(oldVersion) {
  case 1:
    this.upgradeFromVersion1(db);
    break;
  }
}

private void upgradeFromVersion1(SQLiteDatabase db) {
  String SQL_RENAME_TABUSERS =
    "ALTER TABLE " + ContractExample.TabUsers.TABLE_NAME +
    " RENAME TO " + ContractExample.TabUsers.TABLE_NAME + "_TMP";
  db.execSQL(SQL_RENAME_TABUSERS);
  db.execSQL(SQL_CREATE_TABUSERS);
  String[][] dataOld = null;
  final String SQL_GET_OLD_USER_TAB_DATA =
    "SELECT " + ContractExample.TabUsers._ID +
    ", " + ContractExample.TabUsers.COL_NAMEVAL +
    " FROM " + ContractExample.TabUsers.TABLE_NAME + "_TMP";
  Cursor cursor = db.rawQuery(SQL_GET_OLD_USER_TAB_DATA, null);
  if (cursor != null) {
    if (cursor.moveToFirst()) {
      dataOld = new String[cursor.getCount()][3];
      do {
        dataOld[cursor.getPosition()][0] = cursor.getString(0);
          String[] nameValues = cursor.getString(1).split("\\.");
      dataOld[cursor.getPosition()][1] = nameValues[0];
        dataOld[cursor.getPosition()][2] = nameValues[1];
      } while (cursor.moveToNext());
    }
    cursor.close();
  }
```

```
  if (dataOld != null) {
    for (int i = 0; i < dataOld.length; i++) {
      ContentValues values = new ContentValues();
      values.put(ContractExample.TabUsers._ID, dataOld[i][0]);
      values.put(ContractExample.TabUsers.COL_FIRSTNAMEVAL,
        dataOld[i][1]);
      values.put(ContractExample.TabUsers.COL_LASTNAMEVAL,
        dataOld[i][2]);
      db.insert(ContractExample.TabUsers.TABLE_NAME, null, values);
    }
  }

  db.execSQL("DROP TABLE IF EXISTS "
    + ContractExample.TabUsers.TABLE_NAME + "_TMP");
}
```

Sehen wir uns auch diesen Code mal genauer an. Einstiegspunkt für einen Upgrade ist die „onUpgrade“ Methode. Hier bekommen wir die Versionsinformationen und eine Referenz auf unsere Datenbank. Die neue Versionsnummer ist erstmal für uns uninteressant, da wir die aktuelle Version (für unser Beispiel wäre dies 2) ohnehin wissen. Die alte Versionsnummer ist insofern wichtig, als dass wir immer Klarheit darüber brauchen, von welcher Version der Datenbankstruktur wir die Daten in die aktuelle Version konvertieren müssen.

Um dies möglichst übersichtlich zu gestalten, wurde die alte Versionsnummer via switch/case ausgewertet und für jede Version eine eigene Methode geschrieben (für das aktuelle Beispiel haben wir nur eine Methode, aber der Transfer auf mehrere Methoden sollte ja kein Problem sein):

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
  switch(oldVersion) {
  case 1:
    this.upgradeFromVersion1(db);
    break;
  }
}
```

Nun kümmern wir uns um die Methode upgradeFromVersion1. Hier müssen wir zuerst die Daten verschieben. Dies geht am einfachsten, wenn wir die aktuelle Tabelle einfach umbenennen. Eventuell kann man vorher prüfen, ob die Zieltabelle „Users_TMP“ bereits existiert und entsprechend reagieren – wobei wir mal großzügig darauf verzichten:

```
  String SQL_RENAME_TABUSERS =
    "ALTER TABLE " + ContractExample.TabUsers.TABLE_NAME +
    " RENAME TO " + ContractExample.TabUsers.TABLE_NAME + "_TMP";
  db.execSQL(SQL_RENAME_TABUSERS);
```

Danach erzeugen wir die neue Tabellenstruktur, welche entsprechend in dem (nun angepassten) String SQL_CREATE_TABUSERS steht (wir erinnern uns – die Software muss ja nun die neue Struktur im Fokus haben, weshalb in SQL_CREATE_TABUSERS nun die Spalten FirstName und LastName vorhanden sind:

```
  db.execSQL(SQL_CREATE_TABUSERS);
```

Die Daten speichern wir nun in einem Stringarray. Dies ist nur ein Vorschlag – wir könnten die Datenkonvertierung auch direkt mit SQL Mitteln lösen:

```
  String[][] dataOld = null;
```

Die Größe kann natürlich erst festgelegt werden, wenn wir die Anzahl der Rows des Cursors wissen. Nun folgt die eigentliche Abfrage der alten Daten:

```
final String SQL_GET_OLD_USER_TAB_DATA =
  "SELECT " + ContractExample.TabUsers.COL_NAMEVAL +
  ", " + ContractExample.TabUsers._ID +
  " FROM " + ContractExample.TabUsers.TABLE_NAME + "_TMP";
Cursor cursor = db.rawQuery(SQL_GET_OLD_USER_TAB_DATA, null);
```

Der Cursor hat jetzt eine Referenz auf die Daten und wir können ihn nun wie gehabt auslesen:

```
if ((cursor != null) && (cursor.moveToFirst())) {
  dataOld = new String[cursor.getCount()][3];
  do {
    dataOld[cursor.getPosition()][0] = cursor.getString(0);
    String[] nameValues = cursor.getString(1).split("\\.");
    dataOld[cursor.getPosition()][1] = nameValues[0];
    dataOld[cursor.getPosition()][2] = nameValues[1];
  } while (cursor.moveToNext());
  cursor.close();
}
```

In diesem Beispiel gehen wir also davon aus, dass die nameValues immer zwei Punktseparierte Stringsteile sind. Nun haben wir die Daten im Speicher und können sie in die neue Tabelle einfügen. Auch dies ist nichts neues – wir lösen es über ein einfaches INSERT:

```
if (dataOld != null) {
  for (int i = 0; i < dataOld.length; i++) {
    ContentValues values = new ContentValues();
    values.put(ContractExample.TabUsers._ID, dataOld[i][0]);
    values.put(ContractExample.TabUsers.COL_FIRSTNAMEVAL,
      dataOld[i][1]);
    values.put(ContractExample.TabUsers.COL_LASTNAMEVAL,
      dataOld[i][2]);
    db.insert(ContractExample.TabUsers.TABLE_NAME, null, values);
  }
}
```

Jetzt, da die Daten gesichert sind, können wir die temporäre Tabelle wieder löschen:

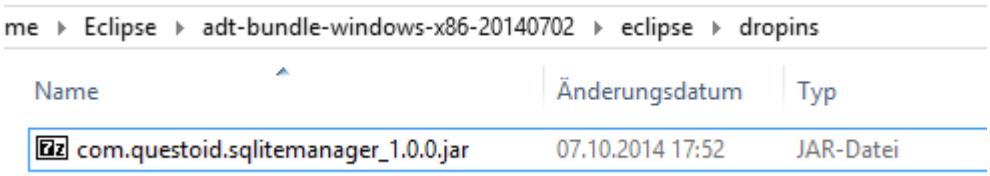
```
db.execSQL("DROP TABLE IF EXISTS "
  + ContractExample.TabUsers.TABLE_NAME + "_TMP");
```

Anmerkung: Wenn wir ein derartiges Upgrade durchführen, müssten wir eigentlich an jeder Stelle davon ausgehen, dass ein potentieller Fehler auftreten könnte, wodurch die Daten stets gesichert sein sollten.

7 SQLite Manager Plugin

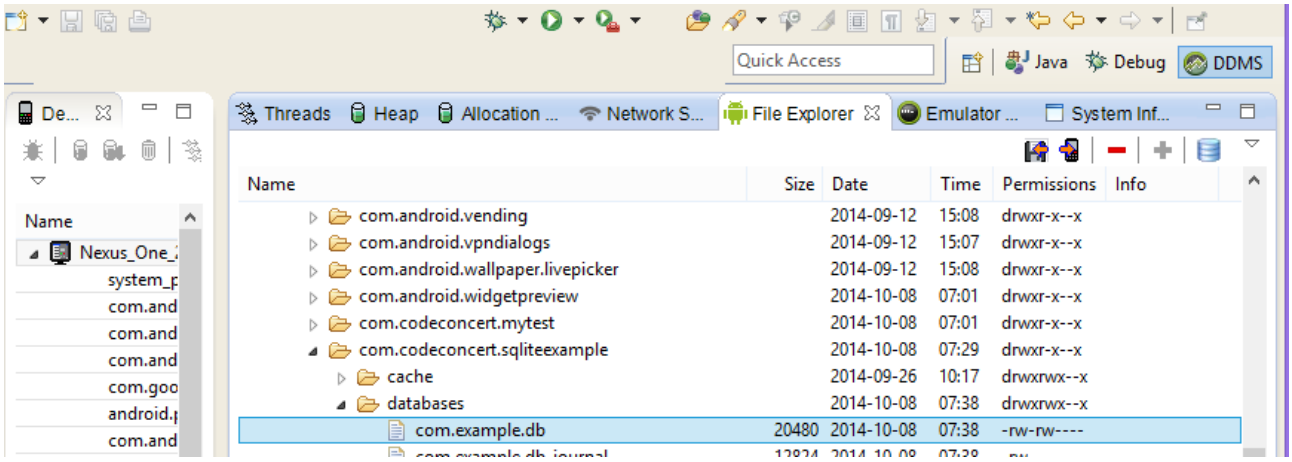
Was die Entwicklung von SQLite Datenbanken mitunter schwierig macht ist das Fehlen einer technischen Sicht auf die Daten, so wie wir es bei anderen Datenbank Management Systemen wie bspw. MySQL kennen. Abhilfe schafft hier das SQLite Manager Plugin für Eclipse – hier ein derzeit aktueller Link (wenn er nicht mehr funktioniert, einfach googlen):
http://www.coderzheaven.com/2011/04/18/sqlitemanager-plugin-for-eclipse/
Mit diesem Plugin könnt ihr die Tabellen und deren Inhalte direkt aus Eclipse heraus ansehen. Auf der Website ist eine Beschreibung, wie das Ding funktioniert. Eine Sache müsst ihr allerdings beachten – die Datenbank muss „db“ heißen, sonst könnt ihr sie mit dem Plugin nicht öffnen. Ansonsten ist das Teil eine sehr praktische Sache!

Hier nochmal ein kurzer Abriss, wie ihr vorgehen müssen, wenn ihr das *.jar File heruntergeladen habt. Ihr kopiert es im Eclipse Installationsverzeichnis in den Unterordner „dropins“:

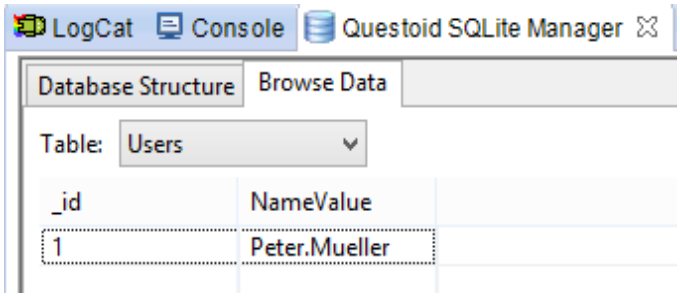


Danach startet ihr Eclipse neu und ihr startet auch einen Emulator. Clickt dann auf die Eclipse Ansichtsperspective „DDMS“ – das ist der Dalvik Debug Monitor Server, welcher uns einen recht umfangreichen Blick auf unser System erlaubt. Wir müssen nun zwei Dinge anklicken. Zum einen auf der linken Seite unseren Emulator und z
Um anderen auf der rechten Seite unser Datenbankfile. Wir finden dies unter data->data->*unser Packagename*->databases.

Wenn wir es angeklickt haben (und unter der Vorraussetzung, dass der Filename mit „.db“ endet – also muss eure Datenbank „db“ heißen!), aktiviert sich rechts oben das Datenbanksymbol. 



Mit einem Klick auf das Datenbanksymbol öffnet sich ein weiterer Reiter, welcher uns einen direkten Lesezugriff auf unsere Datenbank gewährt („Browse Data“):



Noch eine Anmerkung – wenn sich der Dialog nicht öffnet – so war es zumindest bei mir am Anfang – war auf der linken Seite der Emulator nicht ausgewählt.

8 Lizenz

9 Haftung

Ich übernehme keinerlei Haftung für die Richtigkeit der hier gemachten Angaben. Sollten Fehler in dem Dokument enthalten sein, würde ich mich über eine kurze Info unter maik.aicher@gmx.net freuen.