


Android SQLite Database and Content Provider Tutorial

Muhammad Rizal

Related papers

[Download a PDF Pack](#) of the best related papers 

Free tutorial, donate
to support



by Lars Vogel



Android SQLite Database and ContentProvider - Tutorial

Based on Android 4.0

Lars Vogel

Version 4.7

Copyright © 2010, 2011, 2012, 2013 Lars Vogel

16.06.2013

Revision History			
Revision 0.1	22.12.2010	Lars Vogel	Created
Revision 0.2 - 4.7	31.12.2010 - 16.06.2013	Lars Vogel	bug fixes and enhancements

Using the Android SQLite Database

This tutorial describes how to use the SQLite database in Android applications. It also demonstrates how to use existing ContentProvider and how to define new ones. It also demonstrates the usage of the Loader framework which allows to load data asynchronously.

The tutorial is based on Eclipse 4.2, Java 1.6 and Android 4.2.

Table of Contents

1. SQLite and Android
 - 1.1. What is SQLite?
 - 1.2. SQLite in Android
2. Prerequisites for this tutorial
3. SQLite Architecture
 - 3.1. Packages
 - 3.2. SQLiteOpenHelper
 - 3.3. SQLiteDatabase
 - 3.4. rawQuery() Example
 - 3.5. query() Example
 - 3.6. Cursor
 - 3.7. ListViews, ListActivities and SimpleCursorAdapter
4. Tutorial: Using SQLite
 - 4.1. Introduction to the project
 - 4.2. Create Project
 - 4.3. Database and Data Model
 - 4.4. User Interface
 - 4.5. Running the apps
5. Content provider and sharing data
 - 5.1. What is a content provider?
 - 5.2. Accessing a content provider
 - 5.3. Own ContentProvider
 - 5.4. Security and ContentProvider
 - 5.5. Thread Safety
6. Tutorial: Using ContentProvider



1
t



- 6.1. Overview
- 6.2. Create contacts on your emulator
- 6.3. Using the Contact Content Provider

7. Loader

- 7.1. Purpose of the Loader class
- 7.2. Using the Loader class
- 7.3. SQLite database and CursorLoader

8. Cursors and Loaders

9. Tutorial: SQLite, own ContentProvider and Loader

- 9.1. Overview
- 9.2. Project
- 9.3. Database classes
- 9.4. Create ContentProvider
- 9.5. Resources
- 9.6. Layouts
- 9.7. Activities
- 9.8. Start your application

10. Accessing SQLite databases directly

- 10.1. Storage location of the SQLite database
- 10.2. Shell access to the database

11. More on ListViews

12. Get the Kindle edition

13. Questions and Discussion

14. Links and Literature

- 14.1. Source Code
- 14.2. Android SQLite resources
- 14.3. Android Resources
- 14.4. vogella Resources

1. SQLite and Android

1.1. What is SQLite?

SQLite is an Open Source Database which is embedded into Android. SQLite supports standard relational database features like SQL syntax, transactions and prepared statements. In addition it requires only little memory at runtime (approx. 250 KByte).

SQLite supports the data types `TEXT` (similar to String in Java), `INTEGER` (similar to long in Java) and `REAL` (similar to double in Java). All other types must be converted into one of these fields before saving them in the database. SQLite itself does not validate if the types written to the columns are actually of the defined type, e.g. you can write an integer into a string column and vice versa.

More information about SQLite can be found on the SQLite website: <http://www.sqlite.org>.

1.2. SQLite in Android

SQLite is available on every Android device. Using an SQLite database in Android does not require any database setup or administration.

You only have to define the SQL statements for creating and updating the database. Afterwards the database is automatically managed for you by the Android platform.

Access to an SQLite database involves accessing the filesystem. This can be slow. Therefore it is recommended to perform database operations asynchronously, for example inside the `AsyncTask` class.

If your application creates a database, this database is by default saved in the directory `DATA/data/APP_NAME/databases/FILENAME`.

The parts of the above directory are constructed based on the following rules. `DATA` is the path which the `Environment.getDataDirectory()` method returns. `APP_NAME` is your application name. `FILENAME` is the name you specify in your application code for the database.

2. Prerequisites for this tutorial

The following assumes that you have already basic knowledge in Android development. Please check the [Android development tutorial](#) to learn the basics.

3. SQLite Architecture

3.1. Packages

The package `android.database` contains all general classes for working with databases. `android.database.sqlite` contains the SQLite specific classes.

3.2. SQLiteOpenHelper

To create and upgrade a database in your Android application you usually subclass `SQLiteOpenHelper`. In the constructor of your subclass you call the `super()` method of `SQLiteOpenHelper`, specifying the database name and the current database version.

In this class you need to override the `onCreate()` and `onUpgrade()` methods.

`onCreate()` is called by the framework, if the database does not exist.

`onUpgrade()` is called, if the database version is increased in your application code. This method allows you to update the database schema.

Both methods receive an `SQLiteDatabase` object as parameter which represents the database.

`SQLiteOpenHelper` provides the methods `getReadableDatabase()` and `getWritableDatabase()` to get access to an `SQLiteDatabase` object; either in read or write mode.

The database tables should use the identifier `_id` for the primary key of the table. Several Android functions rely on this standard.

It is best practice to create a separate class per table. This class defines static `onCreate()` and `onUpgrade()` methods. These methods are called in the corresponding methods of `SQLiteOpenHelper`. This way your implementation of `SQLiteOpenHelper` will stay readable, even if you have several tables.

3.3. SQLiteDatabase

`SQLiteDatabase` is the base class for working with a SQLite database in Android and provides methods to open, query, update and close the database.

More specifically `SQLiteDatabase` provides the `insert()`, `update()` and `delete()` methods.

In addition it provides the `execSQL()` method, which allows to execute an SQL statement directly.

The object `ContentValues` allows to define key/values. The "key" represents the table column identifier and the "value" represents the content for the table record in this column. `ContentValues` can be used for inserts and updates of database entries.

Queries can be created via the `rawQuery()` and `query()` methods or via the `SQLiteQueryBuilder` class.

`rawQuery()` directly accepts an SQL select statement as input.

`query()` provides a structured interface for specifying the SQL query.

`SQLiteQueryBuilder` is a convenience class that helps to build SQL queries.

3.4. rawQuery() Example

The following gives an example of a `rawQuery()` call.

```
Cursor cursor = getReadableDatabase().
    rawQuery("select * from todo where _id = ?", new String[] { id });
```

3.5. query() Example

The following gives an example of a `query()` call.

```
return database.query(DATABASE_TABLE,
    new String[] { KEY_ROWID, KEY_CATEGORY, KEY_SUMMARY, KEY_DESCRIPTION },
    null, null, null, null, null);
```

The method `query()` has the following parameters.

Table 1. Parameters of the query() method

Parameter	Comment
String dbName	The table name to compile the query against.
String[] columnNames	A list of which table columns to return. Passing "null" will return all columns.
String whereClause	Where-clause, i.e. filter for the selection of data, null will select all data.
String[] selectionArgs	You may include ?s in the "whereClause". These placeholders will get replaced by the values from the selectionArgs array.
String[] groupBy	A filter declaring how to group rows, null will cause the rows to not be grouped.
String[] having	Filter for the groups, null means no filter.
String[] orderBy	Table columns which will be used to order the data, null means no ordering.

If a condition is not required you can pass `null`, e.g. for the group by clause.

The "whereClause" is specified without the word "where", for example a "where" statement might look like: "_id=19 and summary=?".

If you specify placeholder values in the where clause via `?`, you pass them as the `selectionArgs` parameter to the query.

3.6. Cursor

A query returns a `Cursor` object. A `Cursor` represents the result of a query and basically points to one row of the query result. This way Android can buffer the query results efficiently; as it does not have to load all data into memory.

To get the number of elements of the resulting query use the `getCount()` method.

To move between individual data rows, you can use the `moveToFirst()` and `moveToNext()` methods. The `isAfterLast()` method allows to check if the end of the query result has been reached.

`Cursor` provides typed `get*()` methods, e.g. `getLong(columnIndex)`, `getString(columnIndex)` to access the column data for the current position of the result. The "columnIndex" is the number of the column you are accessing.

`Cursor` also provides the `getColumnIndexOrThrow(String)` method which allows to get the column index for a column name of the table.

A `Cursor` needs to be closed with the `close()` method call.

3.7. ListViews, ListActivities and SimpleCursorAdapter

`ListView`s are `Views` which allow to display a list of elements.

`ListActivities` are specialized activities which make the usage of `ListView`s easier.

To work with databases and `ListView`s you can use the `SimpleCursorAdapter`. The `SimpleCursorAdapter` allows to set a layout for each row of the `ListView`s.

You also define an array which contains the column names and another array which contains the IDs of `Views` which should be filled with the data.

The `SimpleCursorAdapter` class will map the columns to the `Views` based on the `Cursor` passed to it.

To obtain the `Cursor` you should use the `Loader` class.

[Database Synchronization](#)

www.pervasync.com

Supports My SQL, OracleDB, MS SQL Server and SQLite.



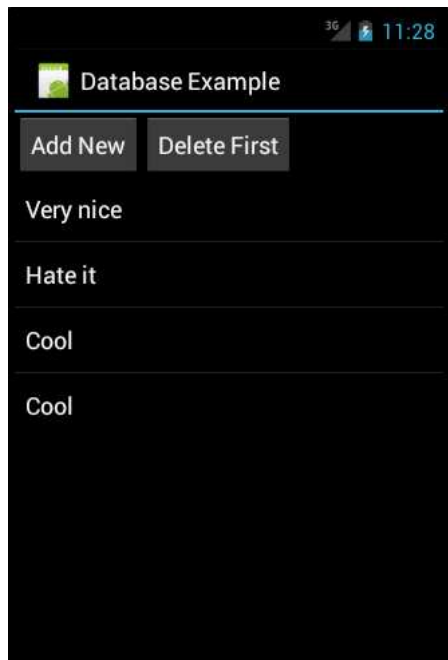
AdChoices

4. Tutorial: Using SQLite

4.1. Introduction to the project

The following demonstrates how to work with an SQLite database. We will use a data access object (DAO) to manage the data for us. The DAO is responsible for handling the database connection and for accessing and modifying the data. It will also convert the database objects into real Java Objects, so that our user interface code does not have to deal with the persistence layer.

The resulting application will look like the following.



Using a DAO is not always the right approach. A DAO creates Java model objects; using a database directly or via a `ContentProvider` is typically more resource efficient as you can avoid the creation of model objects.

I still demonstrate the usage of the DAO in this example to have a relatively simple example to begin with. Use the latest version of Android 4.0. This is currently API Level 15. Otherwise I would have to introduce the `Loader` class, which should be used as of Android 3.0 for managing a database `Cursor`. And this class introduces additional complexity.

4.2. Create Project

Create the new Android project with the name `de.vogella.android.sqlite.first` and an activity called `TestDatabaseActivity`.

4.3. Database and Data Model

Create the `MySQLiteHelper` class. This class is responsible for creating the database. The `onUpgrade()` method will simply delete all existing data and re-create the table. It also defines several constants for the table name and the table columns.

```
package de.vogella.android.sqlite.first;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class MySQLiteHelper extends SQLiteOpenHelper {

    public static final String TABLE_COMMENTS = "comments";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_COMMENT = "comment";

    private static final String DATABASE_NAME = "comments.db";
    private static final int DATABASE_VERSION = 1;

    // Database creation sql statement
    private static final String DATABASE_CREATE = "create table "
        + TABLE_COMMENTS + "(" + COLUMN_ID
        + " integer primary key autoincrement, " + COLUMN_COMMENT
        + " text not null)";

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

    @Override
    public void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(MySQLiteHelper.class.getName(),
            "Upgrading database from version " + oldVersion + " to "
            + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_COMMENTS);
        onCreate(db);
    }
}
```

Create the `Comment` class. This class is our model and contains the data we will save in the database and show in the user interface.

```
package de.vogella.android.sqlite.first;

public class Comment {
    private long id;
    private String comment;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getComment() {
        return comment;
    }

    public void setComment(String comment) {
        this.comment = comment;
    }

    // will be used by the ArrayAdapter in the ListView
    @Override
    public String toString() {
        return comment;
    }
}
```

```
}  
}
```

Create the `CommentsDataSource` class. This class is our DAO. It maintains the database connection and supports adding new comments and fetching all comments.

```
package de.vogella.android.sqlite.first;  
  
import java.util.ArrayList;  
import java.util.List;  
  
import android.content.ContentValues;  
import android.content.Context;  
import android.database.Cursor;  
import android.database.SQLException;  
import android.database.sqlite.SQLiteDatabase;  
  
public class CommentsDataSource {  
  
    // Database fields  
    private SQLiteDatabase database;  
    private MySQLiteHelper dbHelper;  
    private String[] allColumns = { MySQLiteHelper.COLUMN_ID,  
        MySQLiteHelper.COLUMN_COMMENT };  
  
    public CommentsDataSource(Context context) {  
        dbHelper = new MySQLiteHelper(context);  
    }  
  
    public void open() throws SQLException {  
        database = dbHelper.getWritableDatabase();  
    }  
  
    public void close() {  
        dbHelper.close();  
    }  
  
    public Comment createComment(String comment) {  
        ContentValues values = new ContentValues();  
        values.put(MySQLiteHelper.COLUMN_COMMENT, comment);  
        long insertId = database.insert(MySQLiteHelper.TABLE_COMMENTS, null,  
            values);  
        Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,  
            allColumns, MySQLiteHelper.COLUMN_ID + " = " + insertId, null,  
            null, null, null);  
        cursor.moveToFirst();  
        Comment newComment = cursorToComment(cursor);  
        cursor.close();  
        return newComment;  
    }  
  
    public void deleteComment(Comment comment) {  
        long id = comment.getId();  
        System.out.println("Comment deleted with id: " + id);  
        database.delete(MySQLiteHelper.TABLE_COMMENTS, MySQLiteHelper.COLUMN_ID  
            + " = " + id, null);  
    }  
  
    public List<Comment> getAllComments() {  
        List<Comment> comments = new ArrayList<Comment>();  
  
        Cursor cursor = database.query(MySQLiteHelper.TABLE_COMMENTS,  
            allColumns, null, null, null, null, null);  
  
        cursor.moveToFirst();  
        while (!cursor.isAfterLast()) {  
            Comment comment = cursorToComment(cursor);  
            comments.add(comment);  
            cursor.moveToNext();  
        }  
        // Make sure to close the cursor  
        cursor.close();  
        return comments;  
    }  
  
    private Comment cursorToComment(Cursor cursor) {  
        Comment comment = new Comment();  
        comment.setId(cursor.getLong(0));  
        comment.setComment(cursor.getString(1));  
        return comment;  
    }  
}
```

4.4. User Interface

Change your `main.xml` layout file in the `res/layout` folder to the following. This layout has two buttons

for adding and deleting comments and a `ListView` which will be used to display the existing comments. The comment text will be generated later in the activity by a small random generator.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:id="@+id/group"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <Button
            android:id="@+id/add"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Add New"
            android:onClick="onClick"/>

        <Button
            android:id="@+id/delete"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Delete First"
            android:onClick="onClick"/>

    </LinearLayout>

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

Change your `TestDatabaseActivity` class, to the following. We use here a `ListActivity` for displaying the data.

```
package de.vogella.android.sqlite.first;

import java.util.List;
import java.util.Random;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;

public class TestDatabaseActivity extends ListActivity {
    private CommentsDataSource datasource;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        datasource = new CommentsDataSource(this);
        datasource.open();

        List<Comment> values = datasource.getAllComments();

        // Use the SimpleCursorAdapter to show the
        // elements in a ListView
        ArrayAdapter<Comment> adapter = new ArrayAdapter<Comment>(this,
            android.R.layout.simple_list_item_1, values);
        setListAdapter(adapter);
    }

    // Will be called via the onClick attribute
    // of the buttons in main.xml
    public void onClick(View view) {
        @SuppressWarnings("unchecked")
        ArrayAdapter<Comment> adapter = (ArrayAdapter<Comment>) getListAdapter();
        Comment comment = null;
        switch (view.getId()) {
            case R.id.add:
                String[] comments = new String[] { "Cool", "Very nice", "Hate it" };
                int nextInt = new Random().nextInt(3);
                // Save the new comment to the database
                comment = datasource.createComment(comments[nextInt]);
                adapter.add(comment);
                break;
            case R.id.delete:
                if (getListAdapter().getCount() > 0) {
```

```

        comment = (Comment) getListAdapter().getItem(0);
        datasource.deleteComment(comment);
        adapter.remove(comment);
    }
    break;
}
adapter.notifyDataSetChanged();
}

@Override
protected void onResume() {
    datasource.open();
    super.onResume();
}

@Override
protected void onPause() {
    datasource.close();
    super.onPause();
}
}

```

4.5. Running the apps

Install your application and use the Add and Delete button. Restart your application to validate that the data is still there.

5. Content provider and sharing data

5.1. What is a content provider?

A SQLite database is private to the application which creates it. If you want to share data with other applications you can use a content provider.

A content provider allows applications to access data. In most cases this data is stored in an SQLite database.

While a content provider can be used within an application to access data, its is typically used to share data with other application. As application data is by default private, a content provider is a convenient to share you data with other application based on a structured interface.

A content provider must be declared in the *AndroidManifest.xml* file.

5.2. Accessing a content provider

The access to a content provider is done via an URI. The basis for the URI is defined in the declaration of the *ContentProvider* in the *AndroidManifest.xml* file via the *android:authorities* attribute.

Tip

As it is required to know the URIs of an *content providers* to access it, it is good practice to provide public constants for the URIs to document them to other developers.

Many Android datasources, e.g. the contacts, are accessible via *content providers*.

5.3. Own ContentProvider

To create your own *ContentProvider* you have to define a class which extends *android.content.ContentProvider*. You also declare your *ContentProvider* in the *AndroidManifest.xml* file. This entry must specify the *android:authorities* attribute which allows to identify the *ContentProvider*. This authority is the basis for the URI to access data and must be unique.

```
<provider
```

```
        android:authorities="de.vogella.android.todos.contentprovider"
        android:name=".contentprovider.MyTodoContentProvider" >
    </provider>
```

Your `ContentProvider` must implement several methods, e.g. `query()`, `insert()`, `update()`, `delete()`, `getType()` and `onCreate()`. In case you do not support certain methods its good practice to throw an `UnsupportedOperationException()`.

The `query()` method must return a `Cursor` object.

5.4. Security and ContentProvider

Until Android version 4.2 a content provider is by default available to other Android applications. As of Android 4.2 a content provider must be explicitly exported.

To set the visibility of your content provider use the `android:exported=false|true` parameter in the declaration of your content provider in the `AndroidManifest.xml` file.

Tip

It is good practice to always set the `android:exported` parameter to ensure correct behavior across Android versions.

5.5. Thread Safety

If you work directly with databases and have multiple writers from different threads you may run into concurrency issues.

The `ContentProvider` can be accessed from several programs at the same time, therefore you must implement the access thread-safe. The easiest way is to use the keyword `synchronized` in front of all methods of the `ContentProvider`, so that only one thread can access these methods at the same time.

If you do not require that Android synchronizes data access to the `ContentProvider`, set the `android:multiprocess=true` attribute in your `<provider>` definition in the `AndroidManifest.xml` file. This permits an instance of the provider to be created in each client process, eliminating the need to perform interprocess communication (IPC).

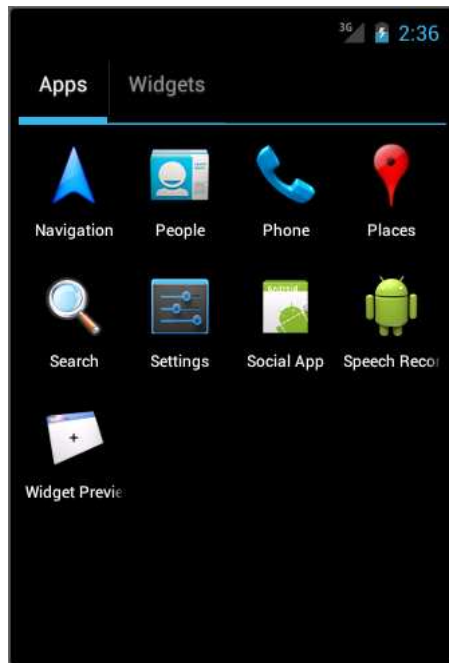
6. Tutorial: Using ContentProvider

6.1. Overview

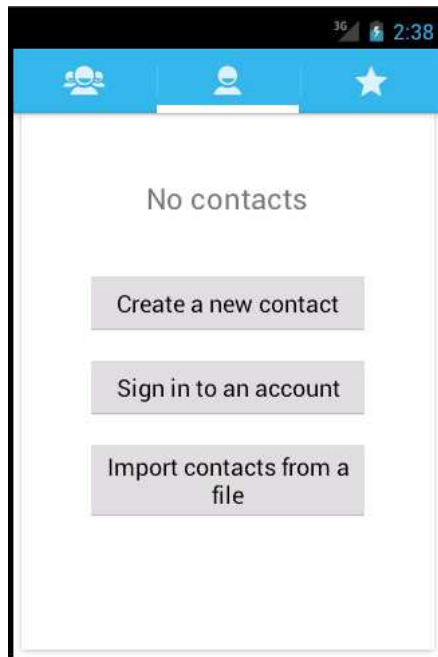
The following example will use an existing `ContentProvider` from the People application.

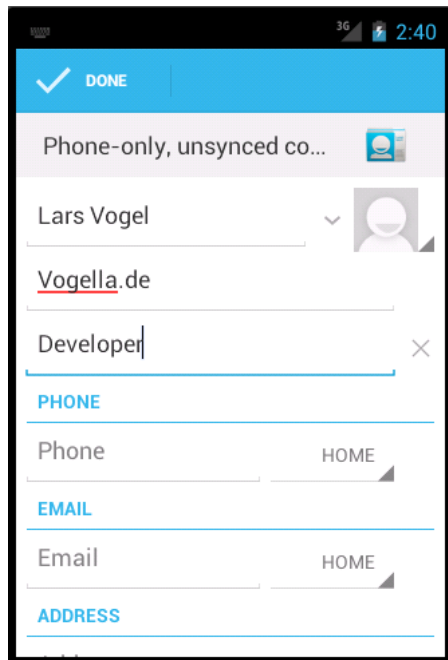
6.2. Create contacts on your emulator

For this example we need a few maintained contacts. Select the home menu and then the People entry to create contacts.



The app will ask you if you want to login. Either login or select "Not now". Press ""Create a new contact". You can create local contacts.





Finish adding your first contact. Afterwards the app allows you to add more contacts via the + button. As a result you should have a few new contacts in your application.

6.3. Using the Contact Content Provider

Create a new Android project called `de.vogella.android.contentprovider` with the activity called `ContactsActivity`.

Change the corresponding layout file in the `res/layout` folder. Rename the ID of the existing `TextView` to `contactview`. Delete the default text.

The resulting layout file should look like the following.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/contactview"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

Access to the contact `ContentProvider` requires a certain permission, as not all applications should have access to the contact information. Open the `AndroidManifest.xml` file, and select the Permissions tab. On that tab click the Add button, and select the Uses Permission. From the drop-down list select the `android.permission.READ_CONTACTS` entry.

Change the coding of the activity.

```
package de.vogella.android.contentprovider;

import android.app.Activity;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.widget.TextView;

public class ContactsActivity extends Activity {

    /** Called when the activity is first created. */

    @Override
```

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_contacts);
    TextView contactView = (TextView) findViewById(R.id.contactview);

    Cursor cursor = getContacts();

    while (cursor.moveToNext()) {

        String displayName = cursor.getString(cursor
            .getColumnIndex(ContactsContract.Data.DISPLAY_NAME));
        contactView.append("Name: ");
        contactView.append(displayName);
        contactView.append("\n");
    }

    private Cursor getContacts() {
        // Run query
        Uri uri = ContactsContract.Contacts.CONTENT_URI;
        String[] projection = new String[] { ContactsContract.Contacts._ID,
            ContactsContract.Contacts.DISPLAY_NAME };
        String selection = ContactsContract.Contacts.IN_VISIBLE_GROUP + " = '"
            + ("1") + "'";
        String[] selectionArgs = null;
        String sortOrder = ContactsContract.Contacts.DISPLAY_NAME
            + " COLLATE LOCALIZED ASC";

        return managedQuery(uri, projection, selection, selectionArgs,
            sortOrder);
    }
}

```

If you run this application the data is read from the `ContentProvider` of the People application and displayed in a `TextView`. Typically you would display such data in a `ListView`.

7. Loader

7.1. Purpose of the Loader class

The `Loader` class allow you to load data asynchronously. They can monitor the source of the data and deliver new results when the content changes. They also persist data between configuration changes.

If the result is retrieved by the `Loader` after the object has been disconnected from its parent (activity or fragment), it can cache the data.

Loaders have been introduced in Android 3.0 and are part of the compatibility layer for Android versions as of 1.6. They are available in the activity and the Fragment class.

7.2. Using the Loader class

You can use the abstract `AsyncTaskLoader` class as the basis for your own `Loader` implementations.

The creation of a `Loader` is done via the `getLoaderManager().initLoader(0, null, this)` method call.

The third parameter of `initLoader()` is the class which is called once the initialization has been started (callback class). Typically the activity is used as the callback class. The first parameter is a unique ID which can be used by the callback class to identify that `Loader` later. The second parameter is a bundle which can be given to the callback class for more information.

The `Loader` is not directly created by the `getLoaderManager().initLoader()` method call, but must be created by the callback class in the `onCreateLoader()` method.

Once the `Loader` has finished reading data asynchronously, the `onLoadFinished()` method of the callback class is called. Here you can update your user interface.

It is good practice that an activity which uses a `Loader` implements the `LoaderManager.LoaderCallbacks` interface.

7.3. SQLite database and CursorLoader

Android provides a `Loader` default implementation to handle SQLite database connections, the `CursorLoader` class.

For a `ContentProvider` based on an SQLite database you would typically use the `CursorLoader` class. This Loader performs the database query in a background thread so that the application is not blocked.

The `CursorLoader` class is the replacement for Activity-managed cursors which are deprecated now.

If the `Cursor` becomes invalid, the `onLoaderReset()` method is called on the callback class.

8. Cursors and Loaders

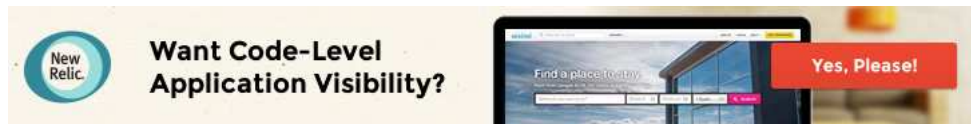
One of the challenges with accessing databases is that this access is slow. The other challenge is that the application needs to consider the life-cycle of the components correctly, e.g. opening and closing the cursor if a configuration change happens.

To manage the life-cycle you could use the `managedQuery()` method in activities prior to Android 3.0.

As of Android 3.0 this method is deprecated and you should use the `Loader` framework to access the `ContentProvider`.

The `SimpleCursorAdapter` class, which can be used with `ListViews`, has the `swapCursor()` method. Your `Loader` can use this method to update the `Cursor` in its `onLoadFinished()` method.

The `CursorLoader` class reconnect the `Cursor` after a configuration change.



9. Tutorial: SQLite, own ContentProvider and Loader

9.1. Overview

The following demo is also available in the Android Market. To allow more users to play with the app, it has been downported to Android 2.3. If you have a barcode scanner installed on your Android phone, you can scan the following QR Code to go to the example app in the Android market. Please note that the app looks and behaves differently due to the different Android versions, e.g. you have an `OptionsMenu` instead of the `ActionBar` and the theme is different.



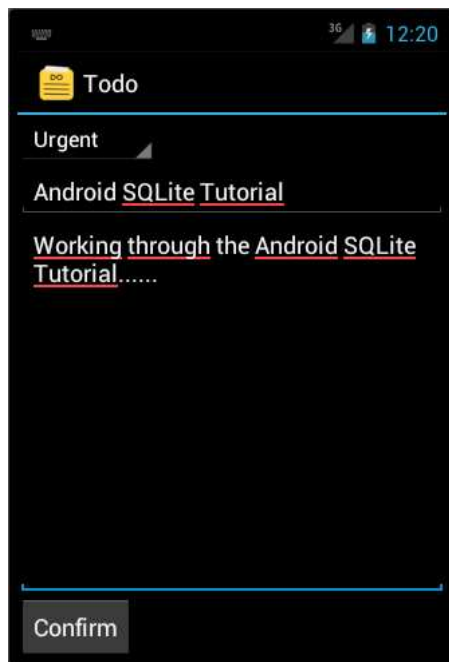
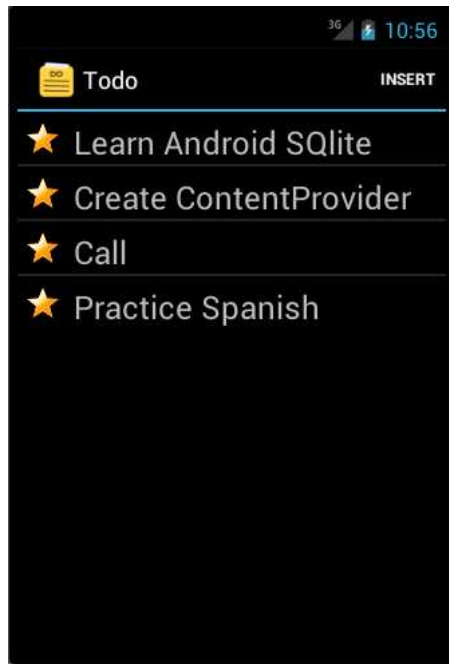
We will create a "To-do" application which allows the user to enter tasks for himself. These items will be stored in the SQLite database and accessed via a `ContentProvider`.

The tasks are called "todo items" or "todos" in this tutorial.

The application consists out of two activities, one for seeing a list of all todo items and one for creating and changing a specific todo item. Both activities will communicate via `Intents`.

To asynchronously load and manage the `Cursor` the main activity will use a `Loader`.

The resulting application will look similar to the following.



9.2. Project

Create the project `de.vogella.android.todos` with the activity called `TodosOverviewActivity`. Create another activity called `TodoDetailActivity`.

9.3. Database classes

Create the package `de.vogella.android.todos.database`. This package will store the classes for the database handling.

As said earlier I consider having one separate class per table as best practice. Even though we have only one table in this example we will follow this practice. This way we are prepared in case our database schema grows.

Create the following class. This class also contains constants for the table name and the columns.

```
package de.vogella.android.todos.database;

import android.database.sqlite.SQLiteDatabase;
import android.util.Log;

public class TodoTable {

    // Database table
    public static final String TABLE_TODO = "todo";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_CATEGORY = "category";
    public static final String COLUMN_SUMMARY = "summary";
    public static final String COLUMN_DESCRIPTION = "description";

    // Database creation SQL statement
    private static final String DATABASE_CREATE = "create table "
        + TABLE_TODO
        + "("
        + COLUMN_ID + " integer primary key autoincrement, "
        + COLUMN_CATEGORY + " text not null, "
        + COLUMN_SUMMARY + " text not null, "
        + COLUMN_DESCRIPTION
        + " text not null"
        + ");";

    public static void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE);
    }

    public static void onUpgrade(SQLiteDatabase database, int oldversion,
        int newVersion) {
        Log.w(TodoTable.class.getName(), "Upgrading database from version "
            + oldversion + " to " + newVersion
            + ", which will destroy all old data");
        database.execSQL("DROP TABLE IF EXISTS " + TABLE_TODO);
        onCreate(database);
    }
}
```

Create the following `TodoDatabaseHelper` class. This class extends `SQLiteOpenHelper` and calls the static methods of the `TodoTable` helper class.

```
package de.vogella.android.todos.database;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class TodoDatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "todotable.db";
    private static final int DATABASE_VERSION = 1;

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

    // Method is called during creation of the database
    @Override
    public void onCreate(SQLiteDatabase database) {
        TodoTable.onCreate(database);
    }

    // Method is called during an upgrade of the database,
    // e.g. if you increase the database version
    @Override
    public void onUpgrade(SQLiteDatabase database, int oldversion,
        int newVersion) {
        TodoTable.onUpgrade(database, oldversion, newVersion);
    }
}
```

We will use a `ContentProvider` for accessing the database; we will not write a data access object (DAO) as we did in the previous SQLite example.

9.4. Create ContentProvider

Create the package `de.vogella.android.todos.contentprovider`.

Create the following `MyTodoContentProvider` class which extends `ContentProvider`.

```

package de.vogella.android.todos.contentprovider;

import java.util.Arrays;
import java.util.HashSet;

import android.content.ContentProvider;
import android.content.ContentResolver;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import de.vogella.android.todos.database.TODODatabaseHelper;
import de.vogella.android.todos.database.TODOTable;

public class MyTodoContentProvider extends ContentProvider {

    // database
    private TODODatabaseHelper database;

    // Used for the UriMacher
    private static final int TODOS = 10;
    private static final int TODO_ID = 20;

    private static final String AUTHORITY = "de.vogella.android.todos.contentprovider";

    private static final String BASE_PATH = "todos";
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY
        + "/" + BASE_PATH);

    public static final String CONTENT_TYPE = ContentResolver.CURSOR_DIR_BASE_TYPE
        + "/todos";
    public static final String CONTENT_ITEM_TYPE = ContentResolver.CURSOR_ITEM_BASE_TYP
E
        + "/todo";

    private static final UriMatcher sURIMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    static {
        sURIMatcher.addURI(AUTHORITY, BASE_PATH, TODOS);
        sURIMatcher.addURI(AUTHORITY, BASE_PATH + "/" + "#", TODO_ID);
    }

    @Override
    public boolean onCreate() {
        database = new TODODatabaseHelper(getContext());
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {

        // Using SQLiteQueryBuilder instead of query() method
        SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

        // Check if the caller has requested a column which does not exists
        checkColumns(projection);

        // Set the table
        queryBuilder.setTables(TODOTable.TABLE_TODO);

        int uriType = sURIMatcher.match(uri);
        switch (uriType) {
            case TODOS:
                break;
            case TODO_ID:
                // Adding the ID to the original query
                queryBuilder.appendWhere(TODOTable.COLUMN_ID + "="
                    + uri.getLastPathSegment());
                break;
            default:
                throw new IllegalArgumentException("Unknown URI: " + uri);
        }

        SQLiteDatabase db = database.getWritableDatabase();
        Cursor cursor = queryBuilder.query(db, projection, selection,
            selectionArgs, null, null, sortOrder);
        // Make sure that potential listeners are getting notified
        cursor.setNotificationUri(getContext().getContentResolver(), uri);

        return cursor;
    }

    @Override
    public String getType(Uri uri) {
        return null;
    }
}

```

```

@Override
public Uri insert(Uri uri, ContentValues values) {
    int uriType = SURIMatcher.match(uri);
    SQLiteDatabase sqlDB = database.getWritableDatabase();
    int rowsDeleted = 0;
    long id = 0;
    switch (uriType) {
        case TODOS:
            id = sqlDB.insert(TodoTable.TABLE_TODO, null, values);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return Uri.parse(BASE_PATH + "/" + id);
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    int uriType = SURIMatcher.match(uri);
    SQLiteDatabase sqlDB = database.getWritableDatabase();
    int rowsDeleted = 0;
    switch (uriType) {
        case TODOS:
            rowsDeleted = sqlDB.delete(TodoTable.TABLE_TODO, selection,
                selectionArgs);
            break;
        case TODO_ID:
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
                rowsDeleted = sqlDB.delete(TodoTable.TABLE_TODO,
                    TodoTable.COLUMN_ID + "=" + id,
                    null);
            } else {
                rowsDeleted = sqlDB.delete(TodoTable.TABLE_TODO,
                    TodoTable.COLUMN_ID + "=" + id
                    + " and " + selection,
                    selectionArgs);
            }
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return rowsDeleted;
}

@Override
public int update(Uri uri, ContentValues values, String selection,
    String[] selectionArgs) {
    int uriType = SURIMatcher.match(uri);
    SQLiteDatabase sqlDB = database.getWritableDatabase();
    int rowsUpdated = 0;
    switch (uriType) {
        case TODOS:
            rowsUpdated = sqlDB.update(TodoTable.TABLE_TODO,
                values,
                selection,
                selectionArgs);
            break;
        case TODO_ID:
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
                rowsUpdated = sqlDB.update(TodoTable.TABLE_TODO,
                    values,
                    TodoTable.COLUMN_ID + "=" + id,
                    null);
            } else {
                rowsUpdated = sqlDB.update(TodoTable.TABLE_TODO,
                    values,
                    TodoTable.COLUMN_ID + "=" + id
                    + " and "
                    + selection,
                    selectionArgs);
            }
            break;
        default:
            throw new IllegalArgumentException("Unknown URI: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return rowsUpdated;
}

private void checkColumns(String[] projection) {
    String[] available = { TodoTable.COLUMN_CATEGORY,
        TodoTable.COLUMN_SUMMARY, TodoTable.COLUMN_DESCRIPTION,
        TodoTable.COLUMN_ID };
    if (projection != null) {
        HashSet<String> requestedColumns = new HashSet<String>(Arrays.asList(projection))
;
    }
}

```

```

        HashSet<String> availableColumns = new HashSet<String>(Arrays.asList(available));
        // check if all columns which are requested are available
        if (!availableColumns.containsAll(requestedColumns)) {
            throw new IllegalArgumentException("Unknown columns in projection");
        }
    }
}
}
}

```

MyTodoContentProvider implements `update()`, `insert()`, `delete()` and `query()` methods. These methods map more or less directly to the `SQLiteDatabase` interface.

It also has the `checkColumns()` method to validate that a query only requests valid columns.

Register your `ContentProvider` in your `AndroidManifest.xml` file.

```

<application
    <!-- Place the following after the Activity
         Definition
    -->
    <provider
        android:name=".contentprovider.MyTodoContentProvider"
        android:authorities="de.vogella.android.todos.contentprovider" >
    </provider>
</application>

```

9.5. Resources

Our application requires several resources. First define a menu `listmenu.xml` in the folder `res/menu`. If you use the Android resource wizard to create the "listmenu.xml" file, the folder will be created for you; if you create the file manually you also need to create the folder manually.

This XML file will be used to define the option menu in our application. The `android:showAsAction="always"` attribute will ensure that this menu entry is displayed in the `ActionBar` of our application.

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >

    <item
        android:id="@+id/insert"
        android:showAsAction="always"
        android:title="Insert">
    </item>

</menu>

```

The user will be able to select the priority for the todo items. For the priorities we create a string array. Create the following file `priority.xml` in the `res/values` folder.

```

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

    <string-array name="priorities">
        <item>Urgent</item>
        <item>Reminder</item>
    </string-array>

</resources>

```

Define also additional strings for the application. Edit `strings.xml` under `res/values`.

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, Todo!</string>
    <string name="app_name">Todo</string>
    <string name="no_todos">Currently there are no Todo items maintained</string>
    <string name="menu_insert">Add Item</string>
    <string name="menu_delete">Delete Todo</string>
    <string name="todo_summary">Summary</string>
    <string name="todo_description">Delete Todo</string>
    <string name="todo_edit_summary">Summary</string>
    <string name="todo_edit_description">Description</string>
    <string name="todo_edit_confirm">Confirm</string>
</resources>

```

9.6. Layouts

We will define three layouts. One will be used for the display of a row in the list, the other ones will be used by our activities.

The row layout refers to an icon called reminder. Paste an icon of type "png" called "reminder.png" into your `res/drawable` folders (`drawable-hdpi`, `drawable-mdpi`, `drawable-ldpi`)

If you do not have an icon available you can copy the icon created by the Android wizard (`ic_launcher.png` in the `res/drawable*` folders) or rename the reference in the layout file. Please note that the Android Development Tools sometimes change the name of this generated icon, so your file might not be called "`ic_launcher.png`".

Alternatively you could remove the icon definition from the "`todo_row.xml`" layout definition file which you will create in the next step.

Create the "`todo_row.xml`" layout file in the folder `res/layout`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="30dp"
        android:layout_height="24dp"
        android:layout_marginLeft="4dp"
        android:layout_marginRight="8dp"
        android:layout_marginTop="8dp"
        android:src="@drawable/reminder" >
    </ImageView>

    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="6dp"
        android:lines="1"
        android:text="@+id/TextView01"
        android:textSize="24dp"
        >
    </TextView>

</LinearLayout>
```

Create the `todo_list.xml` layout file. This layout defines how the list looks like.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </ListView>

    <TextView
        android:id="@android:id/empty"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_todos" />

</LinearLayout>
```

Create the `todo_edit.xml` layout file. This layout will be used to display and edit an individual todo item in the `TodoDetailActivity` activity.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```

        android:layout_height="match_parent"
        android:orientation="vertical" >

        <Spinner
            android:id="@+id/category"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:entries="@array/priorities" >
        </Spinner>

        <LinearLayout
            android:id="@+id/LinearLayout01"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >

            <EditText
                android:id="@+id/todo_edit_summary"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:hint="@string/todo_edit_summary"
                android:imeOptions="actionNext" >
            </EditText>
        </LinearLayout>

        <EditText
            android:id="@+id/todo_edit_description"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_weight="1"
            android:gravity="top"
            android:hint="@string/todo_edit_description"
            android:imeOptions="actionNext" >
        </EditText>

        <Button
            android:id="@+id/todo_edit_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/todo_edit_confirm" >
        </Button>

    </LinearLayout>

```

9.7. Activities

Change the coding of your activities to the following. First `TodosOverviewActivity.java`.

```

package de.vogella.android.todos;

import android.app.ListActivity;
import android.app.LoaderManager;
import android.content.CursorLoader;
import android.content.Intent;
import android.content.Loader;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.ContextMenu.ContextMenuInfo;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.AdapterView.AdapterContextMenuInfo;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
import de.vogella.android.todos.contentprovider.MyTodoContentProvider;
import de.vogella.android.todos.database.TODOTable;

/*
 * TodosOverviewActivity displays the existing todo items
 * in a list
 *
 * You can create new ones via the ActionBar entry "Insert"
 * You can delete existing ones via a long press on the item
 */

public class TodosOverviewActivity extends ListActivity implements
    LoaderManager.LoaderCallbacks<Cursor> {
    private static final int ACTIVITY_CREATE = 0;
    private static final int ACTIVITY_EDIT = 1;
    private static final int DELETE_ID = Menu.FIRST + 1;
    // private Cursor cursor;
    private SimpleCursorAdapter adapter;

```

```

/** Called when the activity is first created. */

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.todo_list);
    this.getListView().setDividerHeight(2);
    fillData();
    registerForContextMenu(getListView());
}

// Create the menu based on the XML definition
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.listmenu, menu);
    return true;
}

// Reaction to the menu selection
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.insert:
            createTodo();
            return true;
    }
    return super.onOptionsItemSelected(item);
}

@Override
public boolean onContextItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case DELETE_ID:
            AdapterContextMenuInfo info = (AdapterContextMenuInfo) item
                .getMenuInfo();
            Uri uri = Uri.parse(MyTodoContentProvider.CONTENT_URI + "/"
                + info.id);
            getContentResolver().delete(uri, null, null);
            fillData();
            return true;
    }
    return super.onContextItemSelected(item);
}

private void createTodo() {
    Intent i = new Intent(this, TodoDetailActivity.class);
    startActivity(i);
}

// Opens the second activity if an entry is clicked
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    super.onListItemClick(l, v, position, id);
    Intent i = new Intent(this, TodoDetailActivity.class);
    Uri todoUri = Uri.parse(MyTodoContentProvider.CONTENT_URI + "/" + id);
    i.putExtra(MyTodoContentProvider.CONTENT_ITEM_TYPE, todoUri);

    startActivity(i);
}

private void fillData() {

    // Fields from the database (projection)
    // Must include the _id column for the adapter to work
    String[] from = new String[] { TodoTable.COLUMN_SUMMARY };
    // Fields on the UI to which we map
    int[] to = new int[] { R.id.label };

    getLoaderManager().initLoader(0, null, this);
    adapter = new SimpleCursorAdapter(this, R.layout.todo_row, null, from,
        to, 0);

    setListAdapter(adapter);
}

@Override
public void onCreateContextMenu(ContextMenu menu, View v,
    ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
    menu.add(0, DELETE_ID, 0, R.string.menu_delete);
}

// Creates a new loader after the initLoader () call
@Override
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    String[] projection = { TodoTable.COLUMN_ID, TodoTable.COLUMN_SUMMARY };
    CursorLoader cursorLoader = new CursorLoader(this,
        MyTodoContentProvider.CONTENT_URI, projection, null, null, null);
}

```

```

        return cursorLoader;
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        adapter.swapCursor(data);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        // data is not available anymore, delete reference
        adapter.swapCursor(null);
    }
}

```

And `TodoDetailActivity.java`

```

package de.vogella.android.todos;

import android.app.Activity;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.Toast;
import de.vogella.android.todos.contentprovider.MyTodoContentProvider;
import de.vogella.android.todos.database.TODOTable;

/*
 * TodoDetailActivity allows to enter a new todo item
 * or to change an existing
 */
public class TodoDetailActivity extends Activity {
    private Spinner mCategory;
    private EditText mTitleText;
    private EditText mBodyText;

    private Uri todoUri;

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.todo_edit);

        mCategory = (Spinner) findViewById(R.id.category);
        mTitleText = (EditText) findViewById(R.id.todo_edit_summary);
        mBodyText = (EditText) findViewById(R.id.todo_edit_description);
        Button confirmButton = (Button) findViewById(R.id.todo_edit_button);

        Bundle extras = getIntent().getExtras();

        // Check from the saved Instance
        todoUri = (bundle == null) ? null : (Uri) bundle
            .getParcelable(MyTodoContentProvider.CONTENT_ITEM_TYPE);

        // Or passed from the other activity
        if (extras != null) {
            todoUri = extras
                .getParcelable(MyTodoContentProvider.CONTENT_ITEM_TYPE);
        }

        fillData(todoUri);
    }

    confirmButton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            if (TextUtils.isEmpty(mTitleText.getText().toString())) {
                makeToast();
            } else {
                setResult(RESULT_OK);
                finish();
            }
        }
    });
}

private void fillData(Uri uri) {
    String[] projection = { TODOTable.COLUMN_SUMMARY,
        TODOTable.COLUMN_DESCRIPTION, TODOTable.COLUMN_CATEGORY };
    Cursor cursor = getContentResolver().query(uri, projection, null, null,
        null);
    if (cursor != null) {
        cursor.moveToFirst();
    }
}

```



```

String category = cursor.getString(cursor
    .getColumnIndexOrThrow(TodoTable.COLUMN_CATEGORY));

for (int i = 0; i < mCategory.getCount(); i++) {

    String s = (String) mCategory.getItemAtPosition(i);
    if (s.equalsIgnoreCase(category)) {
        mCategory.setSelection(i);
    }
}

mTitleText.setText(cursor.getString(cursor
    .getColumnIndexOrThrow(TodoTable.COLUMN_SUMMARY)));
mBodyText.setText(cursor.getString(cursor
    .getColumnIndexOrThrow(TodoTable.COLUMN_DESCRIPTION)));

// Always close the cursor
cursor.close();
}
}

protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    saveState();
    outState.putParcelable(MyTodoContentProvider.CONTENT_ITEM_TYPE, todoUri);
}

@Override
protected void onPause() {
    super.onPause();
    saveState();
}

private void saveState() {
    String category = (String) mCategory.getSelecteditem();
    String summary = mTitleText.getText().toString();
    String description = mBodyText.getText().toString();

    // Only save if either summary or description
    // is available

    if (description.length() == 0 && summary.length() == 0) {
        return;
    }

    ContentValues values = new ContentValues();
    values.put(TodoTable.COLUMN_CATEGORY, category);
    values.put(TodoTable.COLUMN_SUMMARY, summary);
    values.put(TodoTable.COLUMN_DESCRIPTION, description);

    if (todoUri == null) {
        // New todo
        todoUri = getContentResolver().insert(MyTodoContentProvider.CONTENT_URI, values);
    } else {
        // Update todo
        getContentResolver().update(todoUri, values, null, null);
    }
}

private void makeToast() {
    Toast.makeText(TodoDetailActivity.this, "Please maintain a summary",
        Toast.LENGTH_LONG).show();
}
}
}

```

The resulting `AndroidManifest.xml` looks like the following.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="de.vogella.android.todos"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minsdkversion="15" />

    <application
        android:icon="@drawable/icon"
        android:label="@string/app_name" >
        <activity
            android:name=".TodosOverviewActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=".TodoDetailActivity"

```

```

        android:windowSoftInputMode="stateVisible|adjustResize" >
    </activity>

    <provider
        android:name=".contentprovider.MyTodoContentProvider"
        android:authorities="de.vogella.android.todos.contentprovider" >
    </provider>
</application>

</manifest>

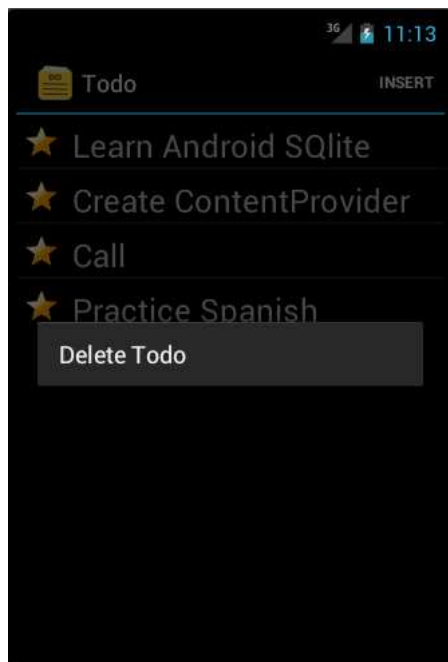
```

Please note that `android:windowSoftInputMode="stateVisible|adjustResize"` is defined for the `TodoDetailActivity`. This makes the keyboard harmonize better with the widgets, but it is not required for this tutorial.

9.8. Start your application

Start your application. You should be able to enter a new todo item via the "Insert" button in the ActionBar.

An existing todo item can be deleted on the list via a long press.



To change an existing todo item, touch the corresponding row. This starts the second activity.

10. Accessing SQLite databases directly



SQLite stores the whole database in a file. If you have access to this file, you can work directly with the data base. Accessing the SQLite database file only works in the emulator or on a rooted device.

A standard Android device will not grant read-access to the database file.

10.2. Shell access to the database

It is possible to access an SQLite database on the emulator or a rooted device via the command line. For this use the following command to connect to the device.

```
adb shell
```

The command `adb` is located in your Android SDK installation folder in the "platform-tools" subfolder.

Afterwards you use the "cd" command to switch the database directory and use the "sqlite3" command to connect to a database. For example in my case:

```
# Switch to the data directory
cd /data/data
# Our application
cd de.vogella.android.todos
# Switch to the database dir
cd databases
# Check the content
ls
# Assuming that there is a todotable file
# connect to this table
sqlite3 todotable.db
```

The most important commands are:

Table 2. SQLite commands

Command	Description
.help	List all commands and options.
.exit	Exit the sqlite3 command.
.schema	Show the CREATE statements which were used to create the tables of the current database.



[BACK TO TOP](#)

You find the complete documentation of SQLite at <http://www.sqlite.org/sqlite.html>.

11. More on ListViews

Please see [Android ListView Tutorial](#) for an introduction into ListViews and ListActivities.

12. Get the Kindle edition

This tutorial is available for your Kindle.



13. Questions and Discussion

Before posting questions, please see the [vogella FAQ](#). If you have questions or find an error in this article please use the [www.vogella.com Google Group](#). I have created a short list [how to create good questions](#) which might also help you.

14. Links and Literature

14.1. Source Code

[Source Code of Examples](#)

14.2. Android SQLite resources

SQLite website

SQL Tutorial

SQLiteManager Eclipse Plug-in

14.3. Android Resources

Android Tutorial

ListView Tutorial

Intents

Android Background Processing Tutorial

Android Location API and Google Maps

Android and Networking

14.4. vogella Resources

vogella Training Android and Eclipse Training from the vogella team

Android Tutorial Introduction to Android Programming

GWT Tutorial Program in Java and compile to JavaScript and HTML

Eclipse RCP Tutorial Create native applications in Java

JUnit Tutorial Test your application

Git Tutorial Put everything you have under distributed version control system