57. An Android SQLite Database Tutorial

The chapter entitled *"An Overview of Android SQLite Databases"* covered the basic principles of integrating relational database storage into Android applications using the SQLite database management system. The previous chapter took a minor detour into the territory of designing TableLayouts within the Android Studio Layout Editor tool, in the course of which, the user interface for an example database application was created. In this chapter, work on the *Database* application project will be continued with the ultimate objective of completing the database example.

57.1 About the Database Example

As is probably evident from the user interface layout designed in the preceding chapter, the example project is a simple data entry and retrieval application designed to allow the user to add, query and delete database entries. The idea behind this application is to allow the tracking of product inventory.

The name of the database will be *productID.db* which, in turn, will contain a single table named *products*. Each record in the database table will contain a unique product ID, a product description and the quantity of that product item currently in stock, corresponding to column names of "productid", "productname" and "productquantity", respectively. The productid column will act as the primary key and will be automatically assigned and incremented by the database management system.

The database schema for the *products* table is outlined in Table 57-2:

Column	Data Type
productid	Integer / Primary Key / Auto Increment
productname	Text
productquantity	Integer

Table 57-2

57.2 Creating the Data Model

Once completed, the application will consist of an activity and a database handler class. The database handler will be a subclass of SQLiteOpenHelper and will provide an abstract layer between the underlying SQLite database and the activity class, with the activity calling on the database handler to interact with the database (adding, removing and querying database entries). In order to implement this interaction in a structured way, a third class will need to be implemented to hold the database entry data as it is passed between the activity and the handler. This is actually a very simple class capable of holding product ID, product name and product quantity values, together with getter and setter methods for accessing these values. Instances of this class can then be created within the activity and database handler and passed back and forth as needed. Essentially, this class can be thought of as representing the database model.

Within Android Studio, navigate within the Project tool window to *app* -> *java* and right-click on the package name. From the popup menu, choose the *New* -> *Java Class* option and, in the *Create New Class* dialog, name the class *Product* before clicking on the *OK* button.

Once created the *Product.java* source file will automatically load into the Android Studio editor. Once loaded, modify the code to add the appropriate data members and methods:

```
package com.ebookfrenzy.database;

public class Product {

    private int _id;
    private String _productname;
    private int _quantity;

    public Product() {

    }

    public Product(int id, String productname, int quantity) {
        this._id = id;
        this._productname = productname;
        this._quantity = quantity;
    }

    public Product(String productname, int quantity) {
        this._productname = productname;
        this._quantity = quantity;
    }

    public void setID(int id) {
        this._id = id;
    }

    public int getID() {
        return this._id;
    }

    public void setProductName(String productname) {
        this._productname = productname;
    }

    public String getProductName() {
        return this._productname;
    }

    public void setQuantity(int quantity) {
        this._quantity = quantity;
    }

    public int getQuantity() {
        return this._quantity;
    }
}
```

The completed class contains private data members for the internal storage of data columns from database entries and a set of methods to get and set those values.

57.3 Implementing the Data Handler

The data handler will be implemented by subclassing from the Android SQLiteOpenHelper class and, as outlined in “An Overview of Android SQLite Databases”, adding the constructor, *onCreate()* and *onUpgrade()* methods. Since the handler will be required to add, query and delete data on behalf of the activity component, corresponding methods will also need to be added to the class.

Begin by adding a second new class to the project to act as the handler, this time named *MyDBHandler*. Once the new class has been created, modify it so that it reads as follows:

```
package com.ebookfrenzy.database;

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

public class MyDBHandler extends SQLiteOpenHelper {

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

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

    }
}
```

Having now pre-populated the source file with template *onCreate()* and *onUpgrade()* methods, the next task is to add a constructor method. Modify the code to declare constants for the database name, table name, table columns and database version and to add the constructor method as follows:

```
package com.ebookfrenzy.database;

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

public class MyDBHandler extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "productDB.db";
    public static final String TABLE_PRODUCTS = "products";

    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PRODUCTNAME = "productname";
    public static final String COLUMN_QUANTITY = "quantity";

    public MyDBHandler(Context context, String name,
      SQLiteDatabase.CursorFactory factory, int version) {
        super(context, DATABASE_NAME, factory, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {

    }

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

    }
}
```

Next, the *onCreate()* method needs to be implemented so that the *products* table is created when the database is first initialized. This involves constructing a SQL CREATE statement containing instructions to create a new table with the appropriate columns and then passing that through to the *execSQL()* method of the SQLiteDatabase object passed as an argument to *onCreate()*:

```
@Override
public void onCreate(SQLiteDatabase db) {
       String CREATE_PRODUCTS_TABLE = "CREATE TABLE " +
             TABLE_PRODUCTS + "("
             + COLUMN_ID + " INTEGER PRIMARY KEY," +
                     COLUMN_PRODUCTNAME
             + " TEXT," + COLUMN_QUANTITY + " INTEGER" + ")";
      db.execSQL(CREATE_PRODUCTS_TABLE);
}
```

The *onUpgrade()* method is called when the handler is invoked with a greater database version number from the one previously used. The exact steps to be performed in this instance will be application specific, so for the purposes of this example, we will simply remove the old database and create a new one:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                            int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_PRODUCTS);
        onCreate(db);
}
```

All that now remains to be implemented in the *MyDBHandler.java* handler class are the methods to add, query and remove database table entries.

57.3.1The Add Handler Method

The method to insert database records will be named *addProduct()* and will take as an argument an instance of our Product data model class. A ContentValues object will be created in the body of the method and primed with key-value pairs for the data columns extracted from the Product object. Next, a reference to the database will be obtained via a call to *getWritableDatabase()* followed by a call to the *insert()* method of the returned database object. Finally, once the insertion has been performed, the database needs to be closed:

```
public void addProduct(Product product) {

        ContentValues values = new ContentValues();
        values.put(COLUMN_PRODUCTNAME, product.getProductName());
        values.put(COLUMN_QUANTITY, product.getQuantity());

        SQLiteDatabase db = this.getWritableDatabase();

        db.insert(TABLE_PRODUCTS, null, values);
        db.close();
}
```

57.3.2The Query Handler Method

The method to query the database will be named *findProduct()* and will take as an argument a String object containing the name of the product to be located. Using this string, a SQL SELECT statement will be constructed to find all matching records in the table. For the purposes of this example, only the first match will then be returned, contained within a new instance of our Product data model class:

```
public Product findProduct(String productname) {
        String query = "SELECT * FROM " + TABLE_PRODUCTS + " WHERE " +
COLUMN_PRODUCTNAME + " = \"" + productname + "\"";

        SQLiteDatabase db = this.getWritableDatabase();
        Cursor cursor = db.rawQuery(query, null);

        Product product = new Product();

        if (cursor.moveToFirst()) {
                cursor.moveToFirst();
                product.setID(Integer.parseInt(cursor.getString(0)));
                product.setProductName(cursor.getString(1));
                product.setQuantity(Integer.parseInt(cursor.getString(2)
                cursor.close();
        } else {
                product = null;
        }
        db.close();
        return product;
}
```

57.3.3The Delete Handler Method

The deletion method will be named *deleteProduct()* and will accept as an argument the entry to be deleted in the form of a Product object. The method will use a SQL SELECT statement to search for the entry based on the product name and, if located, delete it from the table. The success or otherwise of the deletion will be reflected in a Boolean return value:

```
public boolean deleteProduct(String productname) {

        boolean result = false;

        String query = "SELECT * FROM " + TABLE_PRODUCTS + " WHERE " +
COLUMN_PRODUCTNAME + " = \"" + productname + "\"";

        SQLiteDatabase db = this.getWritableDatabase();

        Cursor cursor = db.rawQuery(query, null);

        Product product = new Product();

        if (cursor.moveToFirst()) {
                product.setID(Integer.parseInt(cursor.getString(0)));
                db.delete(TABLE_PRODUCTS, COLUMN_ID + " = ?",
         new String[] { String.valueOf(product.getID()) });
                cursor.close();
                result = true;
        }
        db.close();
        return result;
}
```

57.4 Implementing the Activity Event Methods

The final task prior to testing the application is to wire up *onClick* event handlers on the three buttons in the user interface and to implement corresponding methods for those events. Locate and load the *activity_database.xml* file into the Layout Editor tool, switch to Text mode and locate and modify the three button elements to add *onClick* properties:

```
<Button
    android:text="@string/add"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button3"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="newProduct" />

<Button
    android:text="@string/find"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="lookupProduct" />

<Button
    android:text="@string/delete"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button"
    android:layout_weight="1"
    style="@style/Widget.AppCompat.Button.Borderless"
    android:onClick="removeProduct" />
```

Load the *DatabaseActivity.java* source file into the editor and implement the code to identify the views in the user interface and to implement the three “onClick” target methods:

```
package com.ebookfrenzy.database;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class DatabaseActivity extends AppCompatActivity {

    TextView idView;
    EditText productBox;
    EditText quantityBox;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_database);

        idView = (TextView) findViewById(R.id.productID);
        productBox = (EditText) findViewById(R.id.productName);
        quantityBox =
                (EditText) findViewById(R.id.productQuantity);
    }

    public void newProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);

        int quantity =
                Integer.parseInt(quantityBox.getText().toString());

        Product product =
                new Product(productBox.getText().toString(),
                            quantity);

        dbHandler.addProduct(product);
        productBox.setText("");
        quantityBox.setText("");
    }

    public void lookupProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null, null, 1);
        Product product = dbHandler.findProduct(
              productBox.getText().toString());

        if (product != null) {
            idView.setText(String.valueOf(product.getID()));

            quantityBox.setText(
                     String.valueOf(product.getQuantity()));
        } else {
            idView.setText("No Match Found");
        }
    }

    public void removeProduct (View view) {
        MyDBHandler dbHandler = new MyDBHandler(this, null,
                null, 1);

        boolean result = dbHandler.deleteProduct(
                productBox.getText().toString());

        if (result)
        {
            idView.setText("Record Deleted");
            productBox.setText("");
            quantityBox.setText("");
        }
        else
            idView.setText("No Match Found");
    }
}
```

57.5 Testing the Application

With the coding changes completed, compile and run the application either in an AVD session or on a physical Android device. Once the application is running, enter a product name and quantity value into the user interface form and touch the *Add* button. Once the record has been added the text boxes will clear. Repeat these steps to add a second product to the database. Next, enter the name of one of the newly added products into the product name field and touch the *Find* button. The form should update with the product ID and quantity for the selected product. Touch the *Delete* button to delete the selected record. A subsequent search by product name should indicate that the record no longer exists.

57.6 Summary

The purpose of this chapter has been to work step by step through a practical application of SQLite based database storage in Android applications. As an exercise to develop your new database skill set further, consider extending the example to include the ability to update existing records in the database table.