Working with Android Databases

It's good practice to create a helper class to simplify your database interactions.

Consider creating a database adapter, which adds an abstraction layer that encapsulates database interactions. It should provide intuitive, strongly typed methods for adding, removing, and updating items. A database adapter should also handle queries and wrap creating, opening, and closing the database.

It's often also used as a convenient location from which to publish static database constants, including table names, column names, and column indexes.

The following snippet shows the skeleton code for a standard database adapter class. It includes an extension of the SQLiteOpenHelper class, used to simplify opening, creating, and upgrading the database.

```
import android.content.Context;
import android.database.*;
import android.database.sqlite.*;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.util.Log;
public class MyDBAdapter {
private static final String DATABASE_NAME = “myDatabase.db”;
private static final String DATABASE_TABLE = “mainTable”;
private static final int DATABASE_VERSION = 1;
// The index (key) column name for use in where clauses.
public static final String KEY_ID=”_id”;
// The name and column index of each column in your database.
public static final String KEY_NAME=”name”;
public static final int NAME_COLUMN = 1;
// TODO: Create public field for each column in your table.
// SQL Statement to create a new database.
private static final String DATABASE_CREATE = “create table “ +
DATABASE_TABLE + “ (“ + KEY_ID +
“ integer primary key autoincrement, “ +
KEY_NAME + “ text not null);”;
// Variable to hold the database instance
private SQLiteDatabase db;
// Context of the application using the database.

private final Context context;
// Database open/upgrade helper
private myDbHelper dbHelper;
public MyDBAdapter(Context _context) {
context = _context;
dbHelper = new myDbHelper(context, DATABASE_NAME, null,
DATABASE_VERSION);
}
public MyDBAdapter open() throws SQLException {
db = dbHelper.getWritableDatabase();
return this;
}
public void close() {
db.close();
}
public long insertEntry(MyObject _myObject) {
ContentValues contentValues = new ContentValues();
// TODO fill in ContentValues to represent the new row
return db.insert(DATABASE_TABLE, null, contentValues);
}
public boolean removeEntry(long _rowIndex) {
return db.delete(DATABASE_TABLE, KEY_ID +
“=” + _rowIndex, null) > 0;
}
public Cursor getAllEntries () {
return db.query(DATABASE_TABLE, new String[] {KEY_ID, KEY_NAME},
null, null, null, null, null);
```

```
}
public MyObject getEntry(long _rowIndex) {
MyObject objectInstance = new MyObject();
// TODO Return a cursor to a row from the database and
// use the values to populate an instance of MyObject
return objectInstance;
}
public int updateEntry(long _rowIndex, MyObject _myObject) {
String where = KEY_ID + "=" + _rowIndex;
ContentValues contentValues = new ContentValues();
// TODO fill in the ContentValue based on the new object
return db.update(DATABASE_TABLE, contentValues, where, null);
}
private static class myDbHelper extends SQLiteOpenHelper {
public myDbHelper(Context context, String name,
CursorFactory factory, int version) {
super(context, name, factory, version);
}
// Called when no database exists in
// disk and the helper class needs
// to create a new one.
@Override
public void onCreate(SQLiteDatabase _db) {
_db.execSQL(DATABASE_CREATE);
}
// Called when there is a database version mismatch meaning that
// the version of the database on disk needs to be upgraded to
// the current version.
@Override
public void onUpgrade(SQLiteDatabase _db, int _oldVersion, int _newVersion) {
// Log the version upgrade.
Log.w("TaskDBAdapter", "Upgrading from version " +
_oldVersion + " to " +
_newVersion +
", which will destroy all old data");
// Upgrade the existing database to conform to the new version.
// Multiple previous versions can be handled by comparing
// _oldVersion and _newVersion values.
// The simplest case is to drop the old table and create a
// new one.
_db.execSQL("DROP TABLE IF EXISTS " + DATABASE_TABLE);
// Create a new one.
onCreate(_db);
}
}
}
```

Using the SQLiteOpenHelper

SQLiteOpenHelper is an abstract class that wraps up the best practice pattern for creating, opening, and upgrading databases. By implementing and using an SQLiteOpenHelper, you hide the logic used to decide if a database needs to be created or upgraded before it's opened.

The code snippet above shows how to extend the SQLiteOpenHelper class by overriding the constructor, onCreate, and onUpgrade methods to handle the creation of a new database and upgrading to a new version, respectively.

In the previous example, onUpgrade *simply drops the existing table and replaces it with the new defi -nition. In practice, a better solution is to migrate existing data into the new table.*

To use an implementation of the helper class, create a new instance, passing in the context, database name, current version, and a CursorFactory (if you're using one).

Call getReadableDatabase or getWriteableDatabase to open and return a readable/writable instance of the database.

A call to getWriteableDatabase can fail because of disk space or permission issues, so it's good practice to provide fallback to the getReadableDatabase method as shown below:

```
dbHelper = new myDbHelper(context, DATABASE_NAME, null, DATABASE_VERSION);
SQLiteDatabase db;
try {
db = dbHelper.getWritableDatabase();
}
catch (SQLiteException ex){
db = dbHelper.getReadableDatabase();
}
```

Behind the scenes, if the database doesn't exist, the helper executes its onCreate handler. If the database version has changed, the onUpgrade handler will fi re. In both cases, the get<read/write>ableDatabase call will return the existing, newly created, or upgraded database as appropriate.

Opening and Creating Databases without the SQLiteHelper

You can create and open databases without using the SQLiteHelper class with the openOrCreateDatabase method on the application Context.

Setting up a database is a two-step process. First, call openOrCreateDatabase to create the new database. Then, call execSQL on the resulting database instance to run the SQL commands that will create

your tables and their relationships. The general process is shown in the snippet below:

```
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "mainTable";
private static final String DATABASE_CREATE =
"create table " + DATABASE_TABLE +
" ( _id integer primary key autoincrement," +
"column_one text not null);";
SQLiteDatabase myDatabase;
private void createDatabase() {
myDatabase = openOrCreateDatabase(DATABASE_NAME,
Context.MODE_PRIVATE, null);
myDatabase.execSQL(DATABASE_CREATE);
}
```

Android Database Design Considerations

There are several considerations specifi c to Android that you should consider when designing your database:

❑ Files (such as bitmaps or audio fi les) are not usually stored within database tables. Instead, use a string to store a path to the fi le, preferably a fully qualifi ed Content Provider URI.

❑ While not strictly a requirement, it's strongly recommended that all tables include an autoincrement key fi eld, to function as a unique index value for each row. It's worth noting that if you plan to share your table using a Content Provider, this unique ID fi eld is mandatory.

Querying Your Database

All database queries are returned as a Cursor to a result set. This lets Android manage resources more effi ciently by retrieving and releasing row and column values on demand.

To execute a query on a database, use the query method on the database object, passing in:

❑ An optional Boolean that specifi es if the result set should contain only unique values

❑ The name of the table to query

❑ A projection, as an array of Strings, that lists the columns to include in the result set

❑ A "where" clause that defi nes the rows to be returned. You can include ? wildcards that will be

replaced by the values stored in the selection argument parameter.

❑ An array of selection argument strings that will replace the ?'s in the "where" clause

❑ A "group by" clause that defi nes how the resulting rows will be grouped

❑ A "having" fi lter that defi nes which row groups to include if you specifi ed a "group by" clause

- A String that describes the order of the returned rows
- An optional String that defi nes a limit to the returned rows

The following skeleton code shows snippets for returning some, and all, of the rows in a particular table:

```
// Return all rows for columns one and three, no duplicates
String[] result_columns = new String[] {KEY_ID, KEY_COL1, KEY_COL3};
Cursor allRows = myDatabase.query(true, DATABASE_TABLE, result_columns, null, null, null, null, null, null);
// Return all columns for rows where column 3 equals a set value
// and the rows are ordered by column 5.
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;
Cursor myResult = myDatabase.query(DATABASE_TABLE, null, where, null, null, null, order);
```

In practice, it's often useful to abstract these query commands within an adapter class to simplify data access.

Extracting Results from a Cursor

To extract actual values from a result Cursor, fi rst use the moveTo<location> methods described previously to position the Cursor at the correct row of the result set.

With the Cursor at the desired row, use the type-safe get methods (passing in a column index) to return the value stored at the current row for the specifi ed column, as shown in the following snippet:

```
String columnValue = myResult.getString(columnIndex);
```

Database implementations should publish static constants that provide the column indexes using more easily recognizable variables based on the column names. They are generally exposed within a database adapter as described previously.

The following example shows how to iterate over a result cursor, extracting and summing a column of fl oats:

```
int GOLD_HOARDED_COLUMN = 2;
Cursor myGold = myDatabase.query("GoldHoards", null, null, null, null, null, null);
float totalHoard = 0f;
// Make sure there is at least one row.
if (myGold.moveToFirst()) {
// Iterate over each cursor.
do {
float hoard = myGold.getFloat(GOLD_HOARDED_COLUMN);
totalHoard += hoard;
} while(myGold.moveToNext());
}
float averageHoard = totalHoard / myGold.getCount();
```

Because SQLite database columns are loosely typed, you can cast individual values into valid types as required. For example, values stored as floats can be read back as Strings.

Adding, Updating, and Removing Rows

The SQLiteDatabase class exposes specialized insert, delete, and update methods to encapsulate the SQL statements required to perform these actions. Nonetheless, the execSQL method lets you execute any valid SQL on your database tables should you want to execute these operations manually.

Any time you modify the underlying database values, you should call refreshQuery on any Cursors that currently have a view on the table.

Inserting New Rows

To create a new row, construct a ContentValues object, and use its put methods to supply values for each column. Insert the new row by passing the Content Values object into the insert method called on the target database object — along with the table name — as shown in the snippet below:

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();
// Assign values for each row.
newValues.put(COLUMN_NAME, newValue);
```

```
[ ... Repeat for each column ... ]
// Insert the row into your table
myDatabase.insert(DATABASE_TABLE, null, newValues);
```

Updating a Row on the Database

Updating rows is also done using Content Values.

Create a new ContentValues object, using the put methods to assign new values to each column you want to update. Call update on the database object, passing in the table name, the updated Content Values object, and a where statement that returns the row(s) to update.

The update process is demonstrated in the snippet below:

```
// Define the updated row content.
ContentValues updatedValues = new ContentValues();
// Assign values for each row.
updatedValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]
String where = KEY_ID + "=" + rowId;
// Update the row with the specified index with the new values.
myDatabase.update(DATABASE_TABLE, updatedValues, where, null);
```

Deleting Rows

To delete a row, simply call delete on your database object, specifying the table name and a where clause that returns the rows you want to delete, as shown in the code below:

```
myDatabase.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```

Saving Your To-Do List

Previously in this chapter, you enhanced the To-Do List example to persist the Activity's UI state across sessions. That was only half the job; in the following example, you'll create a private database to save the to-do items:

1. Start by creating a new ToDoDBAdapter class. It will be used to manage your database interactions. Create private variables to store the SQLiteDatabase object and the Context of the calling application. Add a constructor that takes the owner application's Context, and include static class variables for the name and version of the database and a name for the to-do item table.

```
package com.paad.todolist;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;
public class ToDoDBAdapter {
private static final String DATABASE_NAME = "todoList.db";
private static final String DATABASE_TABLE = "todoItems";
private static final int DATABASE_VERSION = 1;
private SQLiteDatabase db;
private final Context context;
public ToDoDBAdapter(Context _context) {
this.context = _context;
}
}
```

2. Create public convenience variables that defi ne the column names and indexes; this will make it easier to fi nd the correct columns when extracting values from query result Cursors.

```
public static final String KEY_ID = "_id";
public static final String KEY_TASK = "task";
public static final int TASK_COLUMN = 1;
public static final String KEY_CREATION_DATE = "creation_date";
public static final int CREATION_DATE_COLUMN = 2;
```

3. Create a new taskDBOpenHelper class within the ToDoDBAdapter that extends SQLiteOpenHelper. It will be used to simplify version management of your database. Within it, overwrite the onCreate and onUpgrade methods to handle the database creation and upgrade logic.

```
private static class toDoDBOpenHelper extends SQLiteOpenHelper {
public toDoDBOpenHelper(Context context, String name,
CursorFactory factory, int version) {
super(context, name, factory, version);
}
// SQL Statement to create a new database.
private static final String DATABASE_CREATE = “create table “ +
DATABASE_TABLE + “ (“ + KEY_ID +
“ integer primary key autoincrement, “ +
KEY_TASK + “ text not null, “ + KEY_CREATION_DATE + “ long);”;
@Override
public void onCreate(SQLiteDatabase _db) {
_db.execSQL(DATABASE_CREATE);
}
@Override
public void onUpgrade(SQLiteDatabase _db, int _oldVersion,
int _newVersion) {
Log.w(“TaskDBAdapter”, “Upgrading from version “ +
_oldVersion + “ to “ +
_newVersion +
“, which will destroy all old data”);
// Drop the old table.
_db.execSQL(“DROP TABLE IF EXISTS “ + DATABASE_TABLE);
// Create a new one.
onCreate(_db);
}
}
```

4. Within the ToDoDBAdapter class, add a private instance variable to store an instance of the toDoDBOpenHelper class you just created; assign it within the constructor.

```
private toDoDBOpenHelper dbHelper;
public ToDoDBAdapter(Context _context) {
this.context = _context;
dbHelper = new toDoDBOpenHelper(context, DATABASE_NAME,
null, DATABASE_VERSION);
}
```

5. Still in the adapter class, create open and close methods that encapsulate the open and close logic for your database. Start with a close method that simply calls close on the database object.

```
public void close() {
db.close();
}
```

6. The open method should use the toDoDBOpenHelper class. Call getWritableDatabase to let the helper handle database creation and version checking. Wrap the call to try to provide a readable database if a writable instance can’t be opened.

```
public void open() throws SQLiteException {
try {
db = dbHelper.getWritableDatabase();
} catch (SQLiteException ex) {
db = dbHelper.getReadableDatabase();
}
}
```

7. Add strongly typed methods for adding, removing, and updating items.

```
// Insert a new task
public long insertTask(ToDoItem _task) {
// Create a new row of values to insert.
ContentValues newTaskValues = new ContentValues();
// Assign values for each row.
newTaskValues.put(KEY_TASK, _task.getTask());
newTaskValues.put(KEY_CREATION_DATE, _task.getCreated().getTime());
// Insert the row.
return db.insert(DATABASE_TABLE, null, newTaskValues);
```

```
}
// Remove a task based on its index
public boolean removeTask(long _rowIndex) {
return db.delete(DATABASE_TABLE, KEY_ID + "=" + _rowIndex, null) > 0;
}
// Update a task
public boolean updateTask(long _rowIndex, String _task) {
ContentValues newValue = new ContentValues();
newValue.put(KEY_TASK, _task);
return db.update(DATABASE_TABLE, newValue,
KEY_ID + "=" + _rowIndex, null) > 0;
}
```

8. Now add helper methods to handle queries. Write three methods — one to return all the items, another to return a particular row as a Cursor, and fi nally, one that returns a strongly typed ToDoItem.

```
public Cursor getAllToDoItemsCursor() {
return db.query(DATABASE_TABLE,
new String[] { KEY_ID, KEY_TASK, KEY_CREATION_DATE},
null, null, null, null, null);
}
public Cursor setCursorToToDoItem(long _rowIndex) throws SQLException {
Cursor result = db.query(true, DATABASE_TABLE,
new String[] {KEY_ID, KEY_TASK},
KEY_ID + "=" + _rowIndex, null, null, null,
null, null);
if ((result.getCount() == 0) || !result.moveToFirst()) {
throw new SQLException("No to do items found for row: " +
_rowIndex);
}
return result;
}
public ToDoItem getToDoItem(long _rowIndex) throws SQLException {
Cursor cursor = db.query(true, DATABASE_TABLE,
new String[] {KEY_ID, KEY_TASK},
KEY_ID + "=" + _rowIndex,
null, null, null, null, null);
if ((cursor.getCount() == 0) || !cursor.moveToFirst()) {
throw new SQLException("No to do item found for row: " +
_rowIndex);
}
String task = cursor.getString(TASK_COLUMN);
long created = cursor.getLong(CREATION_DATE_COLUMN);
ToDoItem result = new ToDoItem(task, new Date(created));
return result;
}
```

9. That completes the database helper class. Return the ToDoList Activity, and update it to persist the to-do list array.

Start by updating the Activity's onCreate method to create an instance of the toDoDBAdapter, and open a connection to the database. Also include a call to the populateTodoList method stub.

```
ToDoDBAdapter toDoDBAdapter;
public void onCreate(Bundle icicle) {
[ ... existing onCreate logic ... ]
toDoDBAdapter = new ToDoDBAdapter(this);
// Open or create the database
toDoDBAdapter.open();
populateTodoList();
}
private void populateTodoList() { }
```

10. Create a new instance variable to store a Cursor over all the to-do items in the database. Update the populateTodoList method to use the toDoDBAdapter instance to query the database, and call startManagingCursor to let the Activity manage the Cursor. It should also make a call to updateArray, a method that will be used to repopulate the to-do list array using the Cursor.

```
Cursor toDoListCursor;
private void populateTodoList() {
// Get all the todo list items from the database.
toDoListCursor = toDoDBAdapter.getAllToDoItemsCursor();
startManagingCursor(toDoListCursor);
// Update the array.
updateArray();
}
private void updateArray() { }
```

11. Now implement the updateArray method to update the current to-do list array. Call requery on the result Cursor to ensure that it's fully up to date, then clear the array and iterate over the result set. When complete, call notifyDataSetChanged on the Array Adapter.

```
private void updateArray() {
toDoListCursor.requery();
todoItems.clear();
if (toDoListCursor.moveToFirst())
do {
String task =
toDoListCursor.getString(ToDoDBAdapter.TASK_COLUMN);
long created =
toDoListCursor.getLong(ToDoDBAdapter.CREATION_DATE_COLUMN);
ToDoItem newItem = new ToDoItem(task, new Date(created));
todoItems.add(0, newItem);
} while(toDoListCursor.moveToNext());
aa.notifyDataSetChanged();
}
```

12. To join the pieces together, modify the OnKeyListener assigned to the text entry box in the onCreate method, and update the removeItem method. Both should now use the toDoDBAdapter to add and remove items from the database rather than modifying the to-do list array directly.

12.1. Start with the OnKeyListener, insert the new item into the database, and refresh the array.

```
public void onCreate(Bundle icicle) {
super.onCreate(icicle);
setContentView(R.layout.main);
myListView = (ListView)findViewById(R.id.myListView);
myEditText = (EditText)findViewById(R.id.myEditText);
todoItems = new ArrayList<ToDoItem>();
int resID = R.layout.todolist_item;
aa = new ToDoItemAdapter(this, resID, todoItems);
myListView.setAdapter(aa);
myEditText.setOnKeyListener(new OnKeyListener() {
public boolean onKey(View v, int keyCode, KeyEvent event) {
if (event.getAction() == KeyEvent.ACTION_DOWN)
if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
ToDoItem newItem;
newItem = new ToDoItem(myEditText.getText().toString());
toDoDBAdapter.insertTask(newItem);
updateArray();
myEditText.setText("");
aa.notifyDataSetChanged();
cancelAdd();
return true;
}
return false;
}
});
registerForContextMenu(myListView);
restoreUIState();
toDoDBAdapter = new ToDoDBAdapter(this);
// Open or create the database
toDoDBAdapter.open();
populateTodoList();
}
```

12.2. Then modify the removeItem method to remove the item from the database and refresh the array list.

```
private void removeItem(int _index) {
// Items are added to the listview in reverse order,
// so invert the index.
toDoDBAdapter.removeTask(todoItems.size()-_index);
updateArray();
}
```

13. As a fi nal step, override the onDestroy method of your Activity to close your database connection.

```
@Override
public void onDestroy() {
// Close the database
toDoDBAdapter.close();
super.onDestroy();
}
```

Your to-do items will now be saved between sessions. As a further enhancement, you could change the Array Adapter to a Cursor Adapter and have the List View update dynamically, directly from changes to the underlying database.

By using a private database, your tasks are not available for other applications to view or add to them. To provide access to your tasks for other applications to leverage, you can expose them using a Content Provider.