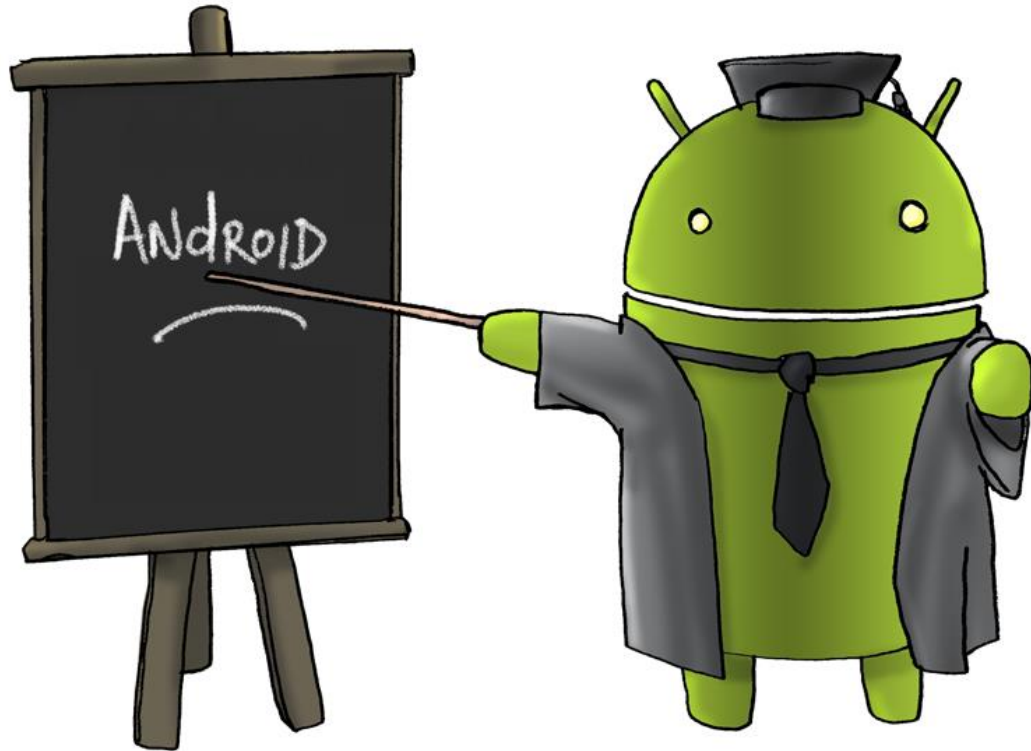# Android Development-Database

By,
Prof. Hiten M Sadani

SQL Databases

Using SQL databases in Android.

Android (as well as iPhone OS) uses an embedded standalone program called **sqlite3** which can be used to:

create a database,
define SQL tables,
indices,
queries, views,
triggers

Insert rows, delete rows,
change rows, run queries
and
administer a SQLite database
file.

SQLLite

- Embedded RDBMS
- ACID Compliant
- Size – about 257 Kbytes
- Not a client/server architecture
  - Accessed via function calls from the application
- Writing (insert, update, delete) locks the database, queries can be done in parallel

SQLLite

- Datastore – single, cross platform file (kinda like an MS Access DB)
  - Definitions
  - Tables
  - Indicies
  - Data

SQLite Data Types

- This is quite different than the normal SQL data types so please read:

- **NULL**. The value is a NULL value.

- **INTEGER**. The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.

- **REAL**. The value is a floating point value, stored as an 8-byte IEEE floating point number.

SQLite Data Types

- **TEXT**. The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).

- **BLOB**. The value is a blob of data, stored exactly as it was input. http://www.sqlite.org/datatype3.html

Storage classes

- **NULL** – null value
- **INTEGER** - signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value
- **REAL** - a floating point value, 8-byte IEEE floating point number.
- **TEXT** - text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

SQLite Data Types

- Boolean Datatype

- SQLite does not have a separate Boolean storage class. Instead, Boolean values are stored as integers 0 (false) and 1 (true).

android.database.sqlite

- Contains the SQLite database management classes that an application would use to manage its own private database.

android.database.sqlite - Classes

- SQLiteCloseable - An object created from a SQLiteDatabase that can be closed.
- SQLiteCursor - A Cursor implementation that exposes results from a query on a SQLiteDatabase.
- SQLiteDatabase - Exposes methods to manage a SQLite database.
- SQLiteOpenHelper - A helper class to manage database creation and version management.
- SQLiteProgram - A base class for compiled SQLite programs.
- SQLiteQuery - A SQLite program that represents a query that reads the resulting rows into a CursorWindow.
- SQLiteQueryBuilder - a convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
- SQLiteStatement - A pre-compiled statement against a SQLiteDatabase that can be reused.

android.database.sqlite.SQLiteDatabase

- Contains the methods for: creating, opening, closing, inserting, updating, deleting and quering an SQLite database
- These methods are similar to JDBC but more method oriented than what we see with JDBC (remember there is not a RDBMS server running)

Database - Creation

- **openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)**
- This method only opens the existing database with the appropriate flag mode. The common flags mode could be
- OPEN_READWRITE
- OPEN_READONLY

SQL Databases

How to create a SQLite database?

public static *SQLiteDatabase*.**openDatabase** (
        String path, *SQLiteDatabase.CursorFactory* factory, *int* flags)

Open the database according to the flags OPEN_READWRITE OPEN_READONLY CREATE_IF_NECESSARY . Sets the locale of the database to the the system's current locale.

Parameters

path   to database file to open and/or create

factory   an optional factory class that is called to instantiate a cursor when query is called, or null for default

flags   to control database access mode

Returns   the newly opened database

Throws   *SQLiteException* if the database cannot be opened

CursorFactory class

- The reason of passing null is you want the standard SQLiteCursor behaviour.

- If you want to implement a specialized Cursor you can get it by by extending the Cursor class(this is for doing additional operations on the query results).

- And in these cases, you can use the CursorFactory class to return an instance of your Cursor implementation.

Database - Creation

- **openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)**
- It not only opens but create the database if it do not exists. This method is equivalent to openDatabase method.

Database - Creation

- **openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)**
- It takes the File object as a path rather then a string. It is equivalent to file.getPath()

openOrCreateDatabase( )

- This method will open an existing database or create one in the application data area

import android.database.sqlite.SQLiteDatabase;

SQLiteDatabase myDatabase;

myDatabase = openOrCreateDatabase ("my_sqlite_database.db" ,
                SQLiteDatabase.CREATE_IF_NECESSARY , null);

SQLite Database Properties

- Important database configuration options include: version, locale, and thread-safe locking.

import java.util.Locale;

myDatabase.setVersion(1);
myDatabase.setLockingEnabled(true);
myDatabase.SetLocale(Locale.getDefault());

Location of Database

- In order to see that where is your database is created.

- Open your android studio, connect your mobile.

- Go **tools/android/android device monitor**. Now browse the file explorer tab.

- Now browse this folder **/data/data/<your.package.name>/databases/<database-name>.**

Database - Helper class

- **Android SQLite SQLiteOpenHelper**

- Android has features available to handle changing database schemas, which mostly depend on using
the SQLiteOpenHelper class.

- **SQLiteOpenHelper** is designed to get rid of two very common problems.

Database - Helper class

- When the application runs the first time – At this point, we do not yet have a database. So we will have to create the tables, indexes, starter data, and so on.

- When the application is upgraded to a newer schema – Our database will still be on the old schema from the older edition of the app. We will have option to alter the database schema to match the needs of the rest of the app.

Database - Helper class

- For managing all the operations related to the database , an helper class has been given and is called **SQLiteOpenHelper**.

- It automatically manages the creation and update of the database. Its syntax is given below

```
class DBHelper(context: Context,DATABASE_NAME:String) : SQLiteOpenHelper(context, DATABASE_NAME, null, 1) {
    override fun onCreate(db: SQLiteDatabase) {}
    override fun onUpgrade(database: SQLiteDatabase, oldVersion: Int, newVersion: Int) {}
}
```

Database - Helper class

- SQLiteOpenHelper wraps up these logic to create and upgrade a database as per our specifications.

- For that we'll need to create a custom subclass of SQLiteOpenHelper implementing at least the following three methods.

Database - Helper class

- **Constructor**
- This takes the Context (e.g., an Activity), the name of the database, an optional cursor factory (we'll discuss this later), and an integer representing the version of the database schema you are using (typically starting from 1 and increment later).

Database - Helper class

- **onCreate(SQLiteDatabase db)**
- It's called when there is no database and the app needs one.
- It passes us a SQLiteDatabase object, pointing to a newly-created database, that we can populate with tables and initial data.

Database - Helper class

- **onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)**

- It's called when the schema version we need does not match the schema version of the database, It passes us a **SQLiteDatabase** object and the old and new version numbers.

- Hence we can figure out the best way to convert the database from the old schema to the new one.

Opening and Closing Android SQLite Database Connection

```kotlin
@Throws(SQLException::class)
fun open(): DBManager {
    dbHelper = DatabaseHelper(context)
    database = dbHelper.getWritableDatabase()
    return this
}


fun close() {
    dbHelper.close()
}
```

Database Table Creation

- we can create table or insert data into table using execSQL method defined in SQLiteDatabase class. Its syntax is given below

```
mydatabase.execSQL("CREATE TABLE IF NOT EXISTS Author(Username VARCHAR,Password VARCHAR);")
execSQL(String sql, Object[] bindArgs)
```

Creating Tables

- Create a static string containing the SQLite CREATE statement, use the execSQL( ) method to execute it.

```
val createAuthor:String = "CREATE TABLE   authors (
id  INTEGER PRIMARY KEY AUTOINCREMENT,
fname  TEXT,
lname  TEXT)"

myDatabase.execSQL(createAuthor)
```

Database - Insertion

- we can create table or insert data into table using execSQL method defined in SQLiteDatabase class. Its syntax is given below

- mydatabase.execSQL("INSERT INTO Author VALUES('admin','admin');");

- **execSQL(String sql, Object[] bindArgs)**

- This method not only insert data , but also used to update or modify already existing data in database using bind arguments

Inserting new Record into Android SQLite database table

- **Content Values** creates an empty set of values using the given initial size.

```kotlin
fun insert(name: String?, desc: String?) {
    val contentValue = ContentValues()
    contentValue.put(DatabaseHelper.SUBJECT, name)
    contentValue.put(DatabaseHelper.DESC, desc)
    database.insert(DatabaseHelper.TABLE_NAME, null, contentValue)
}
```

insert( )

- long insert(String table, String nullColumnHack, ContentValues values)

```
import  android.content.ContentValues;

val  values:ContentValues = ContentValues( );
values.put("firstname" , "J.K.");
values.put("lastname" , "Rowling");
val newAuthorID:Long =  myDatabase.insert("tbl_authors" , "" , values);
```

Android insert datetime value in SQLite database

- **Inserting current timestamp by default**
- If you don't want to insert the timestamp manually each time you create a row, you can do it by keeping the default value while creating the table. Use the **DEFAULT** keyword and one of the following data type.

Android insert datetime value in SQLite database

- **CURRENT_TIME** – Inserts only time
  CURRENT_DATE – Inserts only date
  CURRENT_TIMESTAMP – Inserts both time and date

- CREATE TABLE users(id INTEGER PRIMARY KEY, username TEXT, created_at DATETIME DEFAULT CURRENT_TIMESTAMP);

Android insert datetime value in SQLite database

- **Using datetime() while inserting the row**

- You can also insert datetime manually using **datetime()** function while inserting the row.

- db.execSQL("INSERT INTO users(username, created_at) VALUES('ravitamada', 'datetime()'");

Android insert datetime value in SQLite database

- **Using java Date functions**
- You can also use java **Date()** and **SimpleDateFormat()** methods. Create a function that returns timestamp and use the value while setting the content value for date column.
- Following function **getDateTime()** returns datetime.

Android insert datetime value in SQLite database

- Following function **getDateTime()** returns datetime.

```kotlin
private fun getDateTime(): String? {
    val dateFormat = SimpleDateFormat(
        "yyyy-MM-dd HH:mm:ss", Locale.getDefault()
    )
    val date = Date()
    return dateFormat.format(date)
}
```

Android insert datetime value in SQLite database

- and use the value returned by **getDateTime()** to set content value.

```
ContentValues values = new ContentValues();
values.put('username', 'ravitamada');
values.put('created_at', getDateTime());
// insert the row
long id = db.insert('users', null, values);
```

Updating Record in Android SQLite database table

```kotlin
fun update(_id: Long, name: String?, desc: String?): Int {
    val contentValues = ContentValues()
    contentValues.put(DatabaseHelper.SUBJECT, name)
    contentValues.put(DatabaseHelper.DESC, desc)
    return database.update(
        DatabaseHelper.TABLE_NAME,
        contentValues,
        DatabaseHelper._ID.toString() + " = " + _id,
        null
    )
}
```

update( )

- int update(String table, ContentValues values, String whereClause, String[ ] whereArgs)

```kotlin
fun updateBookTitle(bookId: Int, newTitle: String?) {
    val values = ContentValues()
    values.put("title", newTitle)
    myDatabase.update(
        "tbl_books", values,
        "id=?", arrayOf(bookId.toString())
    )
}
```

Android SQLite – Deleting a Record

```kotlin
fun delete(_id: Long) {
    database.delete(DatabaseHelper.TABLE_NAME, DatabaseHelper._ID.toString() + "=" + _id, null)
}
```

delete( )

- int delete(String table, String whereClause, String[] whereArgs)

```
fun deleteBook(bookId: Int) {
    myDatabase.delete("tbl_books", "id=?", arrayOf(bookId.toString()))
}
```

Android SQLite Cursor

- A Cursor represents the entire result set of the query. Once the query is fetched a call to **cursor.moveToFirst()** is made. Calling moveToFirst() does two things:

- It allows us to test whether the query returned an empty set (by testing the return value)

- It moves the cursor to the first result (when the set is not empty)

- The following code is used to fetch all records:

Database - Fetching

- We can retrieve anything from database using an object of the Cursor class.

- We will call a method of this class called rawQuery and it will return a resultset with the cursor pointing to the table. We can move the cursor forward and retrieve the data.

Database - Fetching

- Cursor resultSet = mydatbase.rawQuery("Select * from Author",null);
- resultSet.moveToFirst();
- String username = resultSet.getString(0);
- String password = resultSet.getString(1);

Database - Fetching

- There are other functions available in the Cursor class that allows us to effectively retrieve the data. That includes

- **getColumnCount()**

- This method return the total number of columns of the table.

- **getColumnIndex(String columnName)**

- This method returns the index number of a column by specifying the name of the column

Database - Fetching

- There are other functions available in the Cursor class that allows us to effectively retrieve the data. That includes

- **getColumnName(int columnIndex)**

- This method returns the name of the column by specifying the index of the column

- **getColumnNames()**

- This method returns the array of all the column names of the table.

Database - Fetching

- There are other functions available in the Cursor class that allows us to effectively retrieve the data. That includes

- **getCount()**

- This method returns the total number of rows in the cursor

- **getPosition()**

- This method returns the current position of the cursor in the table

- **isClosed()**

- This method returns true if the cursor is closed and return false otherwise

- Another way to use a Cursor is to wrap it in a CursorAdapter. Just as ArrayAdapter adapts arrays, CursorAdapter adapts Cursor objects, making their data available to an AdapterView like a ListView.

```kotlin
fun fetch(): Cursor? {
    val columns =
        arrayOf<String>(DatabaseHelper._ID, DatabaseHelper.SUBJECT, DatabaseHelper.DESC)
    val cursor: Cursor =
        database.query(DatabaseHelper.TABLE_NAME, columns, null, null, null, null, null)
    if (cursor != null) {
        cursor.moveToFirst()
    }
    return cursor
}
```

Queries

- Method of SQLiteDatabase class and performs queries on the DB and returns the results in a Cursor object
- Cursor c = mdb.query(p1,p2,p3,p4,p5,p6,p7)
  - p1 ; Table name (String)
  - p2 ; Columns to return (String array)
  - p3 ; WHERE clause (use null for all, ?s for selection args)
  - p4 ; selection arg values for ?s of WHERE clause
  - p5 ; GROUP BY (null for none) (String)
  - p6 ; HAVING (null unless GROUP BY requires one) (String)
  - p7 ; ORDER BY (null for default ordering)(String)
  - p8 ; LIMIT (null for no limit) (String)

Simple Queries

- *SQL* - "SELECT * FROM ABC;"
  SQLite - Cursor c = mdb.query(abc,null,null,null,null,null,null);

- *SQL* - "SELECT * FROM ABC WHERE C1=5"
  SQLite - Cursor c = mdb.query(
         abc,null,"c1=?" , new String[ ] {"5"},null,null,null);

- SQL – "SELECT title,id FROM BOOKS ORDER BY title ASC"
  SQLite – String colsToReturn [ ] {"title","id"};
         String sortOrder = "title ASC";
         Cursor c = mdb.query("books",colsToReturn,
         null,null,null,null,sortOrder);

android.database

- http://developer.android.com/reference/android/database/package-summary.html
- Contains classes and interfaces to explore data returned through a content provider.
- The main thing you are going to use here is the Cursor interface to get the data from the resultset that is returned by a query

  http://developer.android.com/reference/android/database/Cursor.html

```kotlin
import android.content.ContentValues
import android.content.Context
import android.database.Cursor
import android.database.DatabaseUtils
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper
class DBHelper(context: Context?) :
    SQLiteOpenHelper(context, DATABASE_NAME, null, 1) {
    private val hp: HashMap<*, *>? = null
    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(
            "create table contacts " +
                    "(id integer primary key, name text,phone text,email text, street text,place text)"
        )
    }
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        db.execSQL("DROP TABLE IF EXISTS contacts")
        onCreate(db)
    }
```

```kotlin
fun insertContact(
    name: String?,
    phone: String?,
    email: String?,
    street: String?,
    place: String?
): Boolean {
    val db = this.writableDatabase
    val contentValues = ContentValues()
    contentValues.put("name", name)
    contentValues.put("phone", phone)
    contentValues.put("email", email)
    contentValues.put("street", street)
    contentValues.put("place", place)
    db.insert("contacts", null, contentValues)
    return true
}

fun getData(id: Int): Cursor {
    val db = this.readableDatabase
    return db.rawQuery("select * from contacts where id=$id", null)
}
```

```kotlin
fun numberOfRows(): Int {
    val db = this.readableDatabase
    return DatabaseUtils.queryNumEntries(db, CONTACTS_TABLE_NAME).toInt()
}
fun updateContact(id: Int?, name: String?, phone: String?, email: String?, street: String?, place: String?): Boolean {
    val db = this.writableDatabase
    val contentValues = ContentValues()
    contentValues.put("name", name)
    contentValues.put("phone", phone)
    contentValues.put("email", email)
    contentValues.put("street", street)
    contentValues.put("place", place)
    db.update("contacts", contentValues, "id = ? ", arrayOf(Integer.toString(id!!)))
    return true
}

fun deleteContact(id: Int?): Int {
    val db = this.writableDatabase
    return db.delete("contacts","id = ? ", arrayOf(Integer.toString(id!!)))
}
```

```kotlin
val allCotacts: ArrayList<String>
    get() {
        val array_list = ArrayList<String>()
        val db = this.readableDatabase
        val res = db.rawQuery("select * from contacts", null)
        res.moveToFirst()
        while (res.isAfterLast == false) {
            array_list.add(res.getString(res.getColumnIndex(CONTACTS_COLUMN_NAME)))
            res.moveToNext()
        }
        return array_list
    }

companion object {
    const val DATABASE_NAME = "MyDBName.db"
    const val CONTACTS_TABLE_NAME = "contacts"
    const val CONTACTS_COLUMN_ID = "id"
    const val CONTACTS_COLUMN_NAME = "name"
    const val CONTACTS_COLUMN_EMAIL = "email"
    const val CONTACTS_COLUMN_STREET = "street"
    const val CONTACTS_COLUMN_CITY = "place"
    const val CONTACTS_COLUMN_PHONE = "phone"
```

57

```kotlin
}
```

Androidx Room

```kotlin
import androidx.room.Dao
import androidx.room.Insert
import androidx.room.Query


@Dao
interface BookDao {

    @Insert
    fun saveBook(bookEntity: BookEntity)

    @Query( value: "Select * from BookEntity")
    fun readAll() : List<BookEntity>

    @Query( value: "Delete from BookEntity where bookId= :id")
    fun deleteBook(id:Int)

    @Query( value: "Update  BookEntity set bookName='Updated' where bookId= :id")
    fun updateBook(id:Int)
}
```

Androidx Room

```java
@Entity(tableName = "word_table")
public class Word {

    @PrimaryKey
    @NonNull
    @ColumnInfo(name = "word")
    private String mWord;

    public Word(String word) {this.mWord = word;}

    public String getWord(){return this.mWord;}
}
```

```java
@Dao
public interface WordDao {

    // allowing the insert of the same word multiple times by passing a
    // conflict resolution strategy
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insert(Word word);

    @Query("DELETE FROM word_table")
    void deleteAll();

    @Query("SELECT * from word_table ORDER BY word ASC")
    List<Word> getAlphabetizedWords();

}
```

SQL Databases

Questions