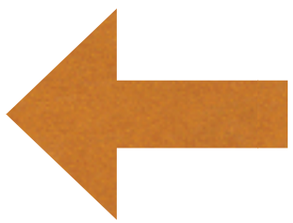
EMBEDDED SYSTEMS PROGRAMMING 2017-18

SQLite

DATA STORAGE: ANDROID

- Shared Preferences
- Filesystem: internal storage
- Filesystem: external storage
- SQLite  (Also available in iOS and WP)
- Network (e.g., Google Drive)

SQLITE

- **Software library that implements a lightweitght SQL database engine**
- No dependencies from external libraries
- One source file (“amalgamation”), one binary file
- Code is mature, extensively checked and portable
- License: completely open

SQLITE: LICENSE

The author disclaims copyright
to this source code.

In place of a legal notice,
here is a blessing:

May you do good and not evil.

May you find forgiveness
for yourself and forgive others.

May you share freely,
never taking more than you give.

SQLITE: FEATURES

- SQLite implements nearly all the features mandated by the SQL-92 standard

- Foreign key support is present since version 3.6.19

- For more info on unimplemented features, look up
  - https://www.sqlite.org/omitted.html
  - https://www.sqlite.org/foreignkeys.html#fk_unsupported

IMPORTANT

- Regardless of the chosen platform, regardless of the fact that you are embracing SQLite or not, what you really need to work with an SQL database is

  - an understanding of the **fundamental concepts behind relational databases,**

  - a good knowledge of the **SQL language**

SQL EPITOME (1/6)

- An SQL database is a relational database made by one or more **tables**.

- A table is made up of **columns** and **rows**.
  Each row represents a record.
  Each column represents data associated with records

- Constraints may be specified concerning data in a table or relations between tables

SQL EPITOME (2/6)

- Defining an (empty) table `addressbook` with three columns: unique identifier, name, phone number

```
create table addressbook
(
    _id   integer primary key,
    name  text,
    phone text
);
```

SQL EPITOME (3/6)

- Inserting a row (i.e., a record) into the table

```
insert into addressbook
values
(
    736,
    'John Doe',
    '555-1212'
);
```

SQL EPITOME (4/6)

- Updating a row (i.e., a record) inside the table

```
update table addressbook
set phone='555-1424'
where _id=736;
```

SQL EPITOME (5/6)

- Deleting a row (i.e., a record) from the table

```
delete from addressbook
where _id=736;
```

- Deleting multiple rows

```
delete from addressbook
where name like "%doe%";
```

SQL EPITOME (6/6)

- Querying, i.e. selecting a subset of rows and colums satisfying a given property

```
select name, phone
from mytable
where
      _id > 100
      and
      name like "%doe%"
order by name;
```

- The query may involve multiple tables (inner join, outer join...)

SQLITE: CORE APIS (1/4)

- **`int sqlite3_open(char *filename, sqlite3 **ppDb)`**
  Opens a connection to the SQLite database identified by `filename`.
  Returns a database connection entity `ppDb`.
  Like all SQLite3 APIs, returns an integer error code

- **`int sqlite3_close(sqlite3 *pDB)`**
  closes a database connection previously opened by a call to `sqlite3_open()`

SQLITE: CORE APIS (2/4)

- **`int sqlite3_prepare_v2(sqlite3 *pDB, char *sqlStatement, int nByte, sqlite3_stmt **ppStmt, char **pzTail)`**
  Converts the SQL statement `sqlStatement` into a prepared statement object.
  Returns a pointer `ppStmt` to the prepared object

- **`int sqlite3_finalize(sqlite3_stmt *pStmt)`**
  Destroys a prepared statement.
  Every prepared statement must be destroyed with this routine in order to avoid memory leaks

- **`int sqlite3_step(sqlite3_stmt *pStmt)`**
  Evaluates a prepared statement up to the point where the first row of the result is available

SQLITE: CORE APIS (3/4)

- **`int sqlite3_column_count(sqlite3_stmt *pStmt)`**
  Gives the number of columns in the result set returned by the prepared statement

- **`int sqlite3_column_type(sqlite3_stmt *pStmt, int iCol)`**
  Returns the datatype code for the initial data type of the result column `iCol`.
  The returned value is one of `SQLITE_INTEGER`, `SQLITE_FLOAT`, `SQLITE_TEXT`, `SQLITE_BLOB`, or `SQLITE_NULL`

- **`int sqlite3_column_int(sqlite3_stmt *pStmt, int iCol)`, `double sqlite3_column_double(sqlite3_stmt*, int iCol)`, `...`**
  Family of functions that return information about a single column

SQLITE: CORE APIS (4/4)

- **int sqlite3_exec(sqlite3 *pDB,**
  const char *sqlString,
  int (*callback)(void*,int,char,char**),**
  void *, char **errmsg)

  Convenience wrapper for `sqlite3_prepare_v2()`, `sqlite3_step()`, and `sqlite3_finalize()`.

  Runs the SQL statements contained in `sqlString`.

  If the callback function of the 3rd argument to `sqlite3_exec()` is not `NULL`, then it is invoked for each result row coming out of the evaluated SQL statements

SQLITE: ERROR CODES

SQLITE_OK	Successful result
SQLITE_ERROR	SQL error or missing database
SQLITE_INTERNAL	Internal logic error in SQLite
SQLITE_PERM	Access permission denied
SQLITE_ABORT	Callback routine requested an abort
SQLITE_BUSY	The database file is locked
SQLITE_LOCKED	A table in the database is locked
SQLITE_NOMEM	A `malloc()` failed
SQLITE_READONLY	Attempt to write a readonly database
SQLITE_INTERRUPT	Operation terminated by `sqlite3_interrupt()`
SQLITE_IOERR	Some kind of disk I/O error occurred
SQLITE_CORRUPT	The database disk image is malformed
SQLITE_NOTFOUND	Unknown opcode in `sqlite3_file_control()`
SQLITE_FULL	Insertion failed because database is full
SQLITE_CANTOPEN	Unable to open the database file
SQLITE_PROTOCOL	Database lock protocol error
SQLITE_EMPTY	Database is empty
SQLITE_SCHEMA	The database schema changed
SQLITE_TOOBIG	String or BLOB exceeds size limit
SQLITE_CONSTRAINT	Abort due to constraint violation
SQLITE_MISMATCH	Data type mismatch
SQLITE_MISUSE	Library used incorrectly
SQLITE_NOLFS	Uses OS features not supported on host
SQLITE_AUTH	Authorization denied
SQLITE_FORMAT	Auxiliary database format error
SQLITE_RANGE	2nd parameter to `sqlite3_bind` out of range
SQLITE_NOTADB	File opened that is not a database file
SQLITE_ROW	`sqlite3_step()` has another row ready
SQLITE_DONE	`sqlite3_step()` has finished executing

CORE SQLITE: EXAMPLES (1/3)

- Creating a table

```
char *err;

const char *sqlString =
    "CREATE TABLE IF NOT EXISTS addressbook ("
    "_id INTEGER PRIMARY KEY AUTOINCREMENT, "
    "name TEXT NON NULL, "
    "phone TEXT);";

if (sqlite3_exec(db, sqlString, NULL, NULL, &err) != SQLITE_OK)
{
    sqlite3_close(db);
    LogError(0, @"Error while creating table.");
}
```

BIND VARIABLES

- SQLite can accept a string where parameters are identified by templates (like a question mark "?") and replace the templates with the real names of the parameters

```
INSERT INTO addressbook VALUES (?, ?, ?);
```

```
INSERT INTO addressbook VALUES (736, 'John Doe', '555-1212');
```

- Use the **`sqlite_bind_XXX()`** family of functions

CORE SQLITE: EXAMPLES (2/3)

- Adding a row to a table

```
void insertIntoAddressbook(int i, char* name, char* phone)
{
    char *sql = "INSERT INTO addressbook VALUES (?, ?, ?);";
    sqlite3_stmt *stmt;

    if (sqlite3_prepare_v2(db, sql, -1, &stmt, nil) == SQLITE_OK)
    {
        sqlite3_bind_int (stmt, 1, i);
        sqlite3_bind_text(stmt, 2, name, -1, NULL);
        sqlite3_bind_text(stmt, 3, phone, -1, NULL);
    }

    if (sqlite3_step(stmt) != SQLITE_DONE)
        LogError(@"Error while adding row.");

    sqlite3_finalize(stmt);
}
```

CORE SQLITE: EXAMPLES (3/3)

- Performing a query

```
void processContactById(int contactId)
{
    sqlite3_stmt * statement;
    char query_stmt[64];

    snprintf(query_stmt, 64,
             "SELECT name, phone FROM addressbook WHERE _id=%d", contactId);

    if (sqlite3_prepare_v2(db, query_stmt, -1, &statement, NULL) == SQLITE_OK)
    {
        if (sqlite3_step(statement) == SQLITE_ROW)
        {
            // Obtain values with sqlite3_column_text(statement, 0)
            // and sqlite3_column_text(statement, 1),
            // then use them for whatever you like
        }
    }

    sqlite3_finalize(statement);
}
```

SQLITE: ANDROID

- Android supports SQLite well
- The SQLite version depends on the Android release and on the choices of the device vendor.
  Android 2.2 and 2.3 usually ship with SQLite 3.6.22.
  Android 4.0+ usually ships with SQLite ≥3.7.x

- Java Package: `android.database.sqlite`
- Tool: `sqlite3`

ANDROID.DATABASE.SQLITE

- Provides SQLite DB management classes
- Most important classes:
  - `SQLiteDatabase`
  - `SQLiteOpenHelper`
  - `SQLiteStatement`
  - `SQLiteQueryBuilder`, `SQLiteCursor`

SQLITEDATABASE (1/2)

Offers methods to perform common DB management tasks on the database associated with a class instance

- **`SQLiteDatabase openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)`**
  Opens a database according to `flags`

- **`void close()`**
  Closes a database

SQLITEDATABASE (2/2)

- **`void execSQL(String sql)`**
  Executes a single SQL statement that is neither a SELECT nor any other SQL statement that returns data

- There are also convenience methods named **`insert`**, **`delete`**, **`replace`**, **`update`**, ... to ease the execution of the corresponding SQL commands

- **`Cursor rawQuery(String sql, String[] selectionArgs)`**
  Runs the provided SQL statement returning data, and returns a `Cursor` over the result set

CURSOR

Provides random access to the result set returned by a DB query

- **`int GetCount()`**
  Returns the number of rows in the cursor

- **`boolean moveToFirst()`, `moveToLast()`, `moveToNext()`, `moveToPrevious()`, `moveToPosition(int position)`**
  Moves the cursor to the specified row

- **`int getType(int columnIndex)`** (Android 3.0+)
  Returns the data type of the given column's value

- **`getString(int columnIndex)`, `getInt(int columnIndex)`, `getFloat(int columnIndex)`, ...**
  Returns the value for the given column in the current row

SQLITEOPENHELPER (1/2)

- Helper class that wraps an `SQLiteDatabase`, providing support for DB creation and management

- Two methods:
  - **`onCreate`**,
  - **`onUpgrade`**,

which are `abstract` because their implementation is tailored to the specific database

SQLITEOPENHELPER (1/2)

- **`abstract void onCreate(SQLiteDatabase db)`**
  Called when the database is created for the first time.
  The implementation should use this method to create tables and relations between tables

- **`abstract void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)`**
  Called when the database schema needs to be upgraded (e.g., because a new version of the application has been installed).
  The implementation should use this method to drop/add tables, or do anything else it needs to upgrade to the new schema version

EXAMPLE (1/2)

```
public class MyOpenHelper extends SQLiteOpenHelper
{
    private static final String DATABASE_NAME = "mydb.db";
    private static final int DATABASE_VERSION = 2;
    public static final String TABLE = "addressbook";
    public static final String NAME = "name";
    public static final String PHONE = "phone";

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

    @Override
    public void onCreate(SQLiteDatabase db)
    {
        String sql = "create table " + TABLE + "( " + BaseColumns._ID
        + " integer primary key autoincrement, " + NAME + " text not null, "
        + PHONE + " text);";
        db.execSQL(sql);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        String sql = null;

        if (oldVersion == 1) sql = "alter table " + TABLE + " add " + PHONE + " text;";

        if (sql != null) db.execSQL(sql);
    }
}
```

EXAMPLE (2/2)

- Somewhere in an activity
  an instance of `MyOpenHelper` is allocated and used

```
...

    MyOpenHelper ab;

...

    ab = new MyOpenHelper(this);

...

    // Add a new record
    SQLiteDatabase db = ab.getWritableDatabase();
    ContentValues values = new ContentValues();
    values.put(ab.NAME, "John Doe");
    values.put(ab.PHONE, "555-1212");
    db.insert(ab.TABLE, null, values);

...
```

SQLITESTATEMENT

- Encapsulates a pre-compiled statement that is intended for reuse

- The statement must be compiled with the `SQLiteDatabase` method `compileStatement(String)`

- The statement works only with the database it has been compiled for

SQLITEQUERYBUILDER, SQLITECURSOR

- **SQLiteQueryBuilder** class
  Helps build SQL queries for `SQLiteDatabase` objects. The key method of this class is

```
String buildQuery(String[] projectionIn, String selection,
                  String groupBy, String having, String sortOrder,
                  String limit)
```

- **SQLiteCursor** class
  Encapsulate results from a query. The SQL statement for the query and the name of the `SQLiteDatabase` are passed as parameters to the constructor

SQLITE3

- Command-line program.
  Can be invoked from an adb remote shell
- Gives you the ability to execute SQLite statements on a database and includes some useful extra commands
- Note: database files for package `<x>` are stored under `/data/data/<x>/databases/`
- Not installed on several devices

ROOM

- Database object mapping library: separates data access from data storage (with SQLite)

- Java package: `android.arch.persistence.room`
- Component must be added to app's `build.gradle`

```
…
dependencies {
    …
    // Room
    implementation "android.arch.persistence.room:runtime:1.0.0"
    annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
    …
}
…
```

ROOM: COMPONENTS

- Components are annotated Java classes

- **Database** class: defines a database.
  It should be an abstract class that extends RoomDatabase.

- **Entity** class: defines a database row.
  For each Entity, a database table is created to hold the items.

- **Dao** class: defines a Data Access Object, i.e., a provider of the methods to access the database

EXAMPLE (1/3)

- Sample Entity class: `Person.java`

```
@Entity
public class Person {
    @PrimaryKey
    private int id;

    @ColumnInfo(name = "name")
    private String name;

    @ColumnInfo(name = "phone")
    private String phone;

    // Getters and setters are omitted for brevity,
    // but they are required for Room to work
}
```

EXAMPLE (2/3)

- **Sample Dao class:** `PersonDao.java`

```
@Dao
public interface PersonDao {
    @Query("SELECT * FROM person")
    List<Person> getAll();

    @Query("SELECT * FROM person WHERE id IN (:userIds)")
    List<Person> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM person WHERE name LIKE :partofname LIMIT 1")
    Person findByName(String partOfName);

    @Insert
    void insertAll(Person... persons);

    @Delete
    void delete(Person person);
}
```

EXAMPLE (3/3)

- Sample Database class: `AppDatabase.java`

```
@Database(entities = {Person.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract PersonDao personDao();
}
```

- Creating an instance of `AppDatabase`:

```
AppDatabase db = Room
.databaseBuilder(getApplicationContext(), AppDatabase.class,
"database-name")
.build();
```

LAST MODIFIED: APRIL 6, 2018