Saving Data in SQL Databases

Saving data to a database is ideal for repeating or structured data, such as contact information. This class assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` (/reference/android/database/sqlite/package-summary.html) package.

THIS LESSON TEACHES YOU TO

1. Define a Schema and Contract
2. Create a Database Using a SQL Helper
3. Put Information into a Database
4. Read Information from a Database
5. Delete Information from a Database
6. Update a Database

YOU SHOULD ALSO READ

- Using Databases

Define a Schema and Contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table that enumerates its columns.

> **Note:** By implementing the `BaseColumns` (/reference/android/provider/BaseColumns.html) interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as cursor adaptors will expect it to have. It's not required, but this can help your database work harmoniously with the Android framework.

For example, this snippet defines the table name and column names for a single table:

```
public static abstract class FeedEntry implements BaseColumns {
    public static final String TABLE_NAME = "entry";
    public static final String COLUMN_NAME_ENTRY_ID = "entryid";
    public static final String COLUMN_NAME_TITLE = "title";
    public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    ...
}
```

To prevent someone from accidentally instantiating the contract class, give it an empty constructor.

```
// Prevents the FeedReaderContract class from being instantiated.
private FeedReaderContract() {}
```

Create a Database Using a SQL Helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
```

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedReaderContract.FeedEntry.TABLE_NAME + " (" +
    FeedReaderContract.FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    " )";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + TABLE_NAME_ENTRIES;
```

Just like files that you save on the device's internal storage (/guide/topics/data/data-storage.html#filesInternal), Android stores your database in private disk space that's associated application. Your data is secure, because by default this area is not accessible to other applications.

A useful set of APIs is available in the SQLiteOpenHelper (/reference/android/database/sqlite/SQLiteOpenHelper.html) class. When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call getWritableDatabase() (/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase()) or getReadableDatabase() (/reference/android/database/sqlite/SQLiteOpenHelper.html#getReadableDatabase()).

> **Note:** Because they can be long-running, be sure that you call getWritableDatabase() (/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase()) or getReadableDatabase() (/reference/android/database/sqlite/SQLiteOpenHelper.html#getReadableDatabase()) in a background thread, such as with AsyncTask (/reference/android/os/AsyncTask.html) or IntentService (/reference/android/app/IntentService.html).

To use SQLiteOpenHelper (/reference/android/database/sqlite/SQLiteOpenHelper.html), create a subclass that overrides the onCreate() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onCreate(android.database.sqlite.SQLiteDatabase)), onUpgrade() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onUpgrade(android.database.sqlite.SQLiteDatabase, int, int)) and onOpen() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onOpen(android.database.sqlite.SQLiteDatabase)) callback methods. You may also want to implement onDowngrade() (/reference/android/database/sqlite/SQLiteOpenHelper.html#onDowngrade(android.database.sqlite.SQLiteDatabase, int, int)), but it's not required.

For example, here's an implementation of SQLiteOpenHelper (/reference/android/database/sqlite/SQLiteOpenHelper.html) that uses some of the commands shown above:

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
```

```
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

To access your database, instantiate your subclass of `SQLiteOpenHelper` (/reference/android/database/sqlite/SQLiteOpenHelper.html):

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

Put Information into a Database

Insert data into the database by passing a `ContentValues` (/reference/android/content/ContentValues.html) object to the `insert()` (/reference/android/database/sqlite/SQLiteDatabase.html#insert(java.lang.String, java.lang.String, android.content.ContentValues)) method:

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
         FeedReaderContract.FeedEntry.TABLE_NAME,
         FeedReaderContract.FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```

The first argument for `insert()` (/reference/android/database/sqlite/SQLiteDatabase.html#insert(java.lang.String, java.lang.String, android.content.ContentValues)) is simply the table name. The second argument provides the name of a column in which the framework can insert NULL in the event that the `ContentValues` (/reference/android/content/ContentValues.html) is empty (if you instead set this to `"null"`, then the framework will not insert a row when there are no values).

Read Information from a Database

To read from a database, use the `query()` (/reference/android/database/sqlite/SQLiteDatabase.html#query(boolean, java.lang.String, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String, java.lang.String, java.lang.String, java.lang.String)) method, passing it your selection criteria and desired columns. The method combines elements of `insert()` (/reference/android/database/sqlite/SQLiteDatabase.html#insert(java.lang.String, java.lang.String, android.content.ContentValues)) and `update()` (/reference/android/database/sqlite/SQLiteDatabase.html#update(java.lang.String,

android.content.ContentValues, java.lang.String, java.lang.String[])), except the column list defines the data you want to fetch, rather than the data to insert. The results of the query are returned to you in a Cursor (/reference/android/database/Cursor.html) object.

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedReaderContract.FeedEntry._ID,
    FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE,
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedReaderContract.FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedReaderContract.FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
    );
```

To look at a row in the cursor, use one of the Cursor (/reference/android/database/Cursor.html) move methods, which you must always call before you begin reading values. Generally, you should start by calling moveToFirst() (/reference/android/database/Cursor.html#moveToFirst()), which places the "read position" on the first entry in the results. For each row, you can read a column's value by calling one of the Cursor (/reference/android/database/Cursor.html) get methods, such as getString() (/reference/android/database/Cursor.html#getString(int)) or getLong() (/reference/android/database/Cursor.html#getLong(int)). For each of the get methods, you must pass the index position of the column you desire, which you can get by calling getColumnIndex() (/reference/android/database/Cursor.html#getColumnIndex(java.lang.String)) or getColumnIndexOrThrow() (/reference/android/database/Cursor.html#getColumnIndexOrThrow(java.lang.String)). For example:

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedReaderContract.FeedEntry._ID)
);
```

Delete Information from a Database

To delete rows from a table, you need to provide selection criteria that identify the rows. The database API provides a mechanism for creating selection criteria that protects against SQL injection. The mechanism divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

```
// Define 'where' part of query.
```

```
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

Update a Database

When you need to modify a subset of your database values, use the `update()` (/reference/android/database/sqlite/SQLiteDatabase.html#update(java.lang.String, android.content.ContentValues, java.lang.String, java.lang.String[])) method.

Updating the table combines the content values syntax of `insert()` (/reference/android/database/sqlite/SQLiteDatabase.html#insert(java.lang.String, java.lang.String, android.content.ContentValues)) with the `where` syntax of `delete()` (/reference/android/database/sqlite/SQLiteDatabase.html#delete(java.lang.String, java.lang.String, java.lang.String[])).

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedReaderContract.FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedReaderContract.FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```