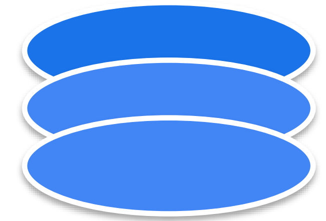
Structured Data & SQL



Why tables beat files, and how to talk to a database

Your app needs to remember things

App data lives in memory (variables, lists, objects)

When the process dies, everything is gone

Users expect their data to survive restarts

We need persistent storage — but what kind?

Your app without a database

Key-value (SharedPreferences)
Small settings:
theme, login token

Files
Unstructured blobs:
images, documents

Database
Structured records:
search, filter, relate

Why not just a file?

Saving one thing is easy: writeText(data)

But real apps manage many records — contacts, messages, songs

Find one record among thousands? → Scan entire file

Delete one record? → Rewrite entire file

Relate records to each other? → Ad-hoc conventions, fragile

Your data as a flat file

A database gives you indexed lookup, structured relationships, and atomic operations — without reinventing the wheel

Structured data: tables

Data organized into rows (records) and columns (attributes)
The schema declares every column's name and type up front

students

id	name	email	year
1	Bart	bart@fox.com	4
2	Lisa	lisa@fox.com	2
3	Milhouse	milhouse@fox.com	4
4	Ralph	ralph@fox.com	2

Rows (horizontal)
One row = one student record
Variable count — grows as data is added

Columns (vertical)
One column = one attribute (name, email, ...)
Fixed count — set by the schema

The schema is a type system — try inserting text where a number is expected and the database says no

The rule: lists go in their own table

One value per row? Fine as a column. A variable-length list? Needs its own table.

One column per phone number is fine — everyone has at most one or two

But a playlist has 5, 50, or 500 songs — you can't add columns for that

Same pattern in Notebook: a note can have many photos → images table

The wrong way to do playlists. How many song columns do we need?

id	name	song1	song2	song3	...
1	Road Trip	Peaches En Regalia	Blue in Green	Goodbye Porkpie Hat	???
2	Study	La Fille Aux Cheveux			

The fix: a separate table for the list

playlists

id	name
1	Road Trip
2	Study

playlist_songs

id	playlist_id	title	pos
1	1	Peaches En Regalia	1
2	1	Blue in Green	2
3	1	Goodbye Porkpie Hat	3
4	2	La Fille Aux Cheveux	1

Primary Key — a unique value (or combination) that identifies each row

Each song is its own row — add 5 or 500, no schema change needed

playlist_id stores the playlist's id — linking each song to its parent

The position column preserves the ordering within each playlist

Foreign key — linking rows across tables

Foreign Key (FK) — a column whose values are primary keys from another table

playlists

id (PK)	name
1	Road Trip
2	Study

playlist_songs

id	playlist_id (FK)	title	pos
1	1	Peaches En Regalia	1
2	1	Blue in Green	2
3	1	Goodbye Porkpie Hat	3
4	2	La Fille Aux Cheveux	1

playlist_id = 1 means "this song belongs to Road Trip"

The database enforces the link — rejects inserts with a playlist_id that doesn't match any playlist

Keys: matching data across tables

Primary key = unique ID for each row

Foreign key = "look up that row over there"

students

id (PK)	name
1	Bart
2	Lisa
3	Milhouse

enrollments

student_id (FK)	course_id	grade
1	101	B-
2	101	A+
3	102	B+

Primary key — unique per row, like a student ID number. The database guarantees no duplicates.

Foreign key — points to a primary key in another table. The database ensures the target row exists.

Relating data across tables

A foreign key can model a many-to-one relationship.
Combine two to model many-to-many:

students

id	name
1	Bart
2	Lisa
3	Milhouse

courses

id	name	teacher
101	CS 371M	Krabappel
102	Math 301	Hoover

enrollments

student_id PK,FK	course_id PK,FK	grade
1	101	B-
1	102	C
2	101	A+
3	102	B+

Foreign keys model many-to-one relationships
enrollments.student_id → students.id
enrollments.course_id → courses.id

Composite key: (student_id, course_id)
Neither is unique alone, but
the pair uniquely identifies each row

SQL: Structured Query Language

A declarative language: describe what data you want,
not how to find it

Standard syntax across databases (SQLite, PostgreSQL, MySQL, ...)

Generally case-insensitive (SELECT = select)

Four core operations:

C
CREATE
INSERT INTO

R
READ
SELECT

U
UPDATE
UPDATE...SET

D
DELETE
DELETE FROM

CREATE TABLE — the schema is a contract

The schema declares what your data looks like.
The database rejects anything that doesn't match — strong typing for free.

```
CREATE TABLE students (
    id     INTEGER PRIMARY KEY
           AUTOINCREMENT,
    name   TEXT NOT NULL,
    email  TEXT,
    year   INTEGER
)
```

INTEGER, TEXT, REAL
Type — DB rejects wrong types

PRIMARY KEY
Unique per row, never null

AUTOINCREMENT
DB assigns next id for you

NOT NULL
Column must have a value

```
INSERT INTO students (name, year) VALUES (42, 'four');  →  Type mismatch!
The schema catches bugs at the database boundary.
```

SELECT — reading data

General form

```
SELECT   <columns>   FROM   <table>
 WHERE   <condition>
 ORDER BY   <column>   [ASC | DESC]
 LIMIT   <n>
```

Example

```
SELECT name, email FROM students
WHERE year = 4
ORDER BY name
```

→ Result:

name	email
Bart	bart@fox.com
Milhouse	milhouse@fox.com

More examples

```
SELECT * FROM students                            -- all rows, all columns
SELECT * FROM students ORDER BY id DESC LIMIT 3 -- most recent 3
```

INSERT — creating data

General form

```
INSERT INTO <table> (<columns>)
VALUES (<values>)
```

Example

```
-- id is auto-assigned (AUTOINCREMENT)
INSERT INTO students (name, email, year)
VALUES ('Nelson', 'nelson@fox.com', 4)
```

id	name	email	year
1	Bart	bart@fox.com	4
2	Lisa	lisa@fox.com	2
3	Milhouse	milhouse@fox.com	4
4	Ralph	ralph@fox.com	2
5	Nelson	nelson@fox.com	4

UPDATE — modifying data

General form

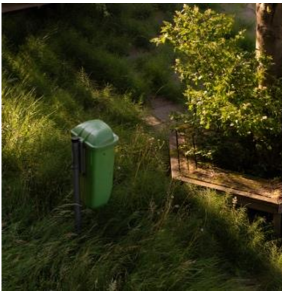
```
UPDATE <table> SET
    <column1> = <value1>,
    <column2> = <value2>
WHERE <condition>
```

Example

```
UPDATE students SET email = 'bart_simpson@fox.com'
WHERE id = 1
```

id	name	email	year
1	Bart	bart_simpson@fox.com	4
2	Lisa	lisa@fox.com	2

DELETE — removing data



General form

```
DELETE FROM <table>
WHERE <condition>
```

Example

```
DELETE FROM students WHERE id = 4
```

Without WHERE, DELETE removes every row and UPDATE changes every row!
Always double-check your WHERE clause.

Foreign key with ON DELETE CASCADE: deleting a parent row automatically deletes all child rows that reference it

SQLite: a database in your pocket

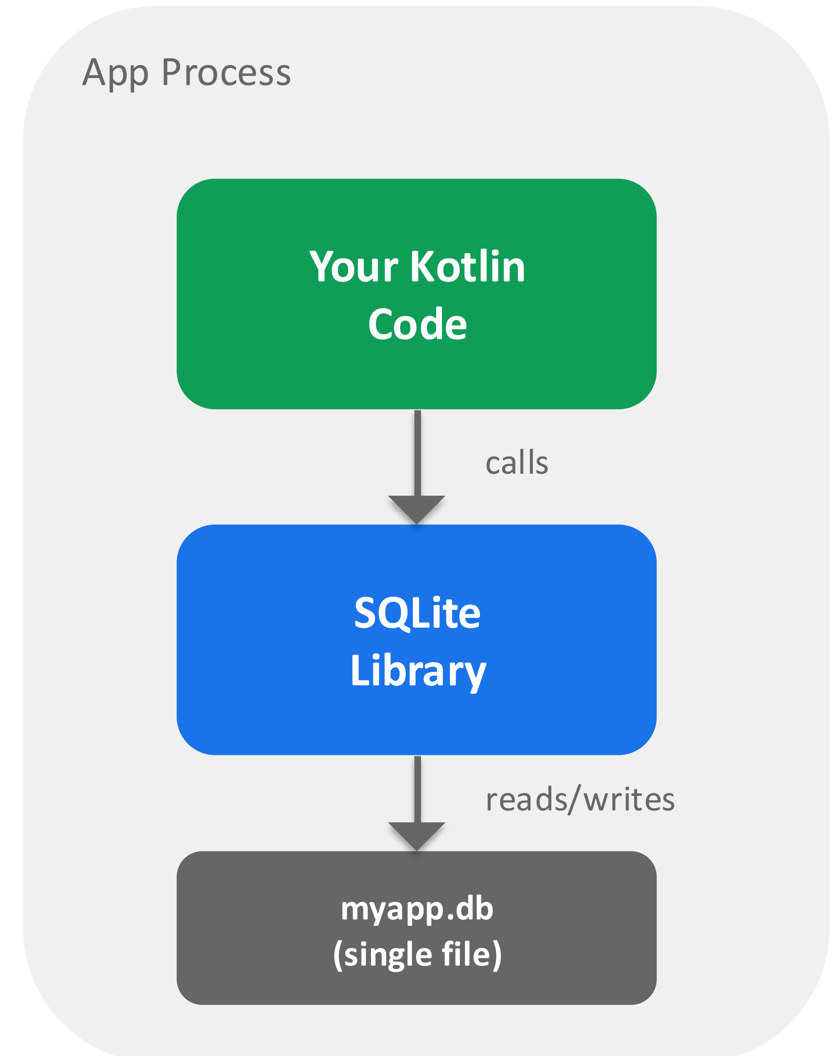
Not a server — a C library linked directly into your app

Entire database is a single file on disk

Built into every Android device (and iOS, browsers, ...)

Most widely deployed database engine in the world

Zero configuration, no setup, no admin

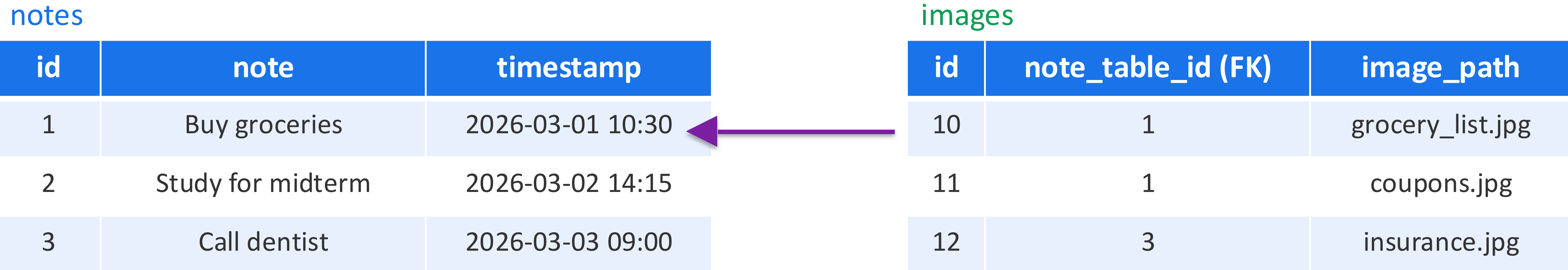


Applying it: the Notebook app

Two tables, four operations, one architecture

Notebook's data model: notes + images

A note has text and a timestamp. Each note can have many photos.
That's a one-to-many relationship — two tables, linked by a foreign key.



notes

id	note	timestamp
1	Buy groceries	2026-03-01 10:30
2	Study for midterm	2026-03-02 14:15
3	Call dentist	2026-03-03 09:00

images

id	note_table_id (FK)	image_path
10	1	grocery_list.jpg
11	1	coupons.jpg
12	3	insurance.jpg

Foreign key: images.note_table_id → notes.id
ON DELETE CASCADE: delete a note → its images vanish automatically

Schema in Kotlin — Note.kt

Each data class declares its table schema as a companion constant.
The CREATE TABLE string defines the columns, types, and constraints.

Note.kt

```
data class Note(val id: Long,
    val text: String,
    val timestamp: String,
    val imageList: List<Image>)
companion object {
  const val CREATE_TABLE =
    "CREATE TABLE notes ("
    + "id INTEGER PRIMARY KEY "
    + "AUTOINCREMENT, note TEXT,"
    + "timestamp DATETIME "
    + "DEFAULT CURRENT_TIMESTAMP)"
}
```

The table this creates:

notes

id	note	timestamp
1	Buy groceries	2026-03-01 10:30
2	Study for midterm	2026-03-02 14:15
3	Call dentist	2026-03-03 09:00

AUTOINCREMENT → DB assigns next id for you

DEFAULT CURRENT_TIMESTAMP → auto-filled on insert

One class manages your entire database

DatabaseHelper extends SQLiteOpenHelper — Android calls its methods at the right time so you never manage the DB file yourself:

onCreate

Called once when the DB file is first created

```
db.execSQL(Note.CREATE_TABLE)
db.execSQL(Image.CREATE_TABLE)
```

onUpgrade

Called when you bump the version number

```
db.execSQL("DROP TABLE ...")
onCreate(db)  // recreate
```

onOpen

Called every time the DB is opened

```
db.execSQL(
  "PRAGMA foreign_keys=ON")
```

Gotcha: SQLite disables foreign keys by default!

Without PRAGMA foreign_keys=ON in onOpen, your CASCADE rules silently do nothing.

SQL injection attacks

Building SQL by gluing in user input
lets attackers inject their own commands into your query:

Dangerous: building SQL with string concatenation

```
val search = editText.text.toString()
val sql = "SELECT * FROM notes WHERE note = '$search'"
db.rawQuery(sql, null)
```

Normal: search = "groceries"

```
SELECT * FROM notes
  WHERE note = 'groceries'
```

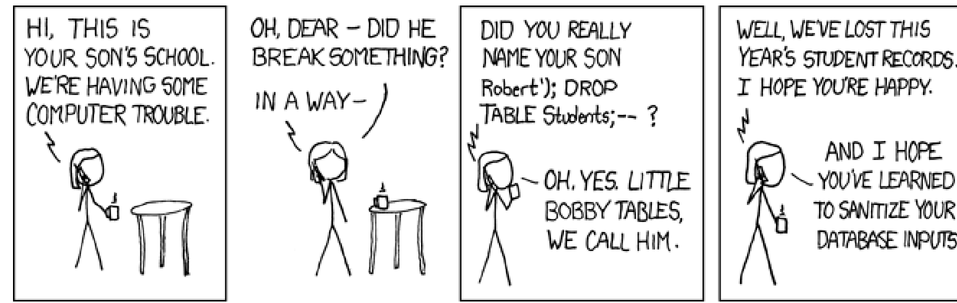
Attack: search = "'; DROP TABLE notes;--"

```
SELECT * FROM notes
  WHERE note = '';
DROP TABLE notes;--'
```

The user's text became executable SQL —
the database ran DROP TABLE and destroyed all data!
(-- is a SQL comment that hides the trailing quote)

The fix: separate data from commands

Never concatenate user input into SQL. Use APIs that keep data separate from commands:



xkcd.com/327

Safe: ContentValues

```
val cv = ContentValues()
cv.put("name",
    editText.text.toString())
db.insert("students", null, cv)
```

Safe: parameterized query (?)

```
db.rawQuery(
  "SELECT * FROM notes "
  + "WHERE note = ?",
  arrayOf(search))
```

Both patterns separate SQL structure from user data — the database knows what is code and what is a value

Creating a note — the insert flow

db.insert() returns the new row's auto-generated id.
Use that id as the foreign key when inserting child image rows:

DatabaseHelper.insertNote()

```
// Step 1: Insert the note — get back its id
val values = ContentValues()
values.put("note", text)
val noteId = db.insert("notes", null, values)

// Step 2: Insert each image, linking via FK
for (image in images) {
    val imgValues = ContentValues()
    imgValues.put("note_table_id", noteId)
    imgValues.put("image_path", image.imagePath)
    db.insert("images", null, imgValues)
}
```

notes

id	note	timestamp
7	Buy groceries	(auto)

noteId = 7
(foreign key)

images

id	note_table_id (FK)	image_path
10	7	grocery_list.jpg
11	7	coupons.jpg

db.insert() returns the new id → use it as the FK for child rows.
This two-step pattern (parent first, then children) is fundamental to relational inserts.

Querying data — Cursors

db.query() returns a Cursor — iterate row by row, extract columns:

DatabaseHelper.allNotes

```
val allNotes: List<Note> get() {
    val cursor = readableDatabase.query(
        Note.TABLE_NAME, null,
        null, null, null, null,
        Note.COLUMN_TIMESTAMP + " DESC")
    return cursor.use {
        val notes = mutableListOf<Note>()
        if (it.moveToFirst()) { do {
            val id = it.getLong(
                it.getColumnIndexOrThrow("id"))
            notes.add(Note(id, ...))
        } while (it.moveToNext()) }
        notes
    }
}
```

id	note	timestamp
3	Call dentist	03-03 09:00
2	Study midterm	03-02 14:15
1	Buy groceries	03-01 10:30

cursor → row 2

moveToFirst() — position at row 1

moveToNext() — advance; false at end

Filtered queries

db.query() uses selection and selectionArgs to filter rows with parameterized ? placeholders:

SQLiteDatabase.query() parameters

```
db.query(
    table,          // "businesses"
    columns,        // null = all
    selection,      // "city = ?"
    selectionArgs,  // arrayOf("Austin")
    groupBy,        // null
    having,         // null
    orderBy,        // "price ASC"
    limit           // "10"
)
```

Example: find cheap Austin restaurants

```
val where = "city = ? AND price < ?"
val args  = arrayOf("Austin", "3")

val cursor = db.query(
    "businesses", null,
    where, args,
    null, null,
    "price ASC", "20")
```

Resulting SQL: SELECT * FROM businesses WHERE city = 'Austin' AND price < 3 ORDER BY price ASC LIMIT 20

Each ? is replaced by the corresponding element of selectionArgs — same parameterized query pattern that prevents SQL injection

Notebook architecture: layered design

Fragment
(HomeFragment, NoteEdit, ImageRoll)

Observes LiveData, displays UI, handles taps

↓

ViewModel
(MainViewModel)

Exposes LiveData to UI, DB is source of truth

↓

Repository
(ViewModelDBHelper)

Abstracts DB calls, single point of access

↓

Database
(DatabaseHelper → SQLite)

Raw SQL via SQLiteOpenHelper

The Repository pattern

The ViewModel calls the same methods no matter where data lives.
Swap the implementation — the ViewModel never changes:

ViewModel always calls:

```
// MainViewModel.kt — same callback no matter what
dbHelp.fetchNotes { _notesList.value = it }
dbHelp.createNote(text, imgs) { _notesList.value = it }
dbHelp.removeNote(note) { _notesList.value = it }
```

Today: SQLite

```
class ViewModelDBHelper(ctx: Context) {
  fun fetchNotes(cb: (List<Note>)->Unit)
    { cb(db.allNotes) }          // sync
  fun createNote(t, imgs, cb: ...)
    { db.insertNote(t, imgs); cb(db.allNotes) }
}
```

Tomorrow: Firebase

```
class ViewModelDBHelper() {
  fun fetchNotes(cb: (List<Note>)->Unit) {
    db.collection("notes").get()
      .addOnSuccessListener { cb(it.toNotes()) } }
  fun createNote(note, cb: ...) {
    db.add(note)
      .addOnSuccessListener { fetchNotes(cb) } }
}
```

The payoff — identical ViewModels

Because ViewModelDBHelper hides the backend, the ViewModel CRUD code is the same for SQLite and Firebase:

Notebook (SQLite)

```
fun createNote(text: String, imageList: List<Image>) {
    dbHelp.createNote(text, imageList) { freshList ->
        _notesList.value = freshList
    }
}
fun removeNoteAt(position: Int) {
    val note = getNote(position)
    dbHelp.removeNote(note) { freshList ->
        _notesList.value = freshList
    }
}
```

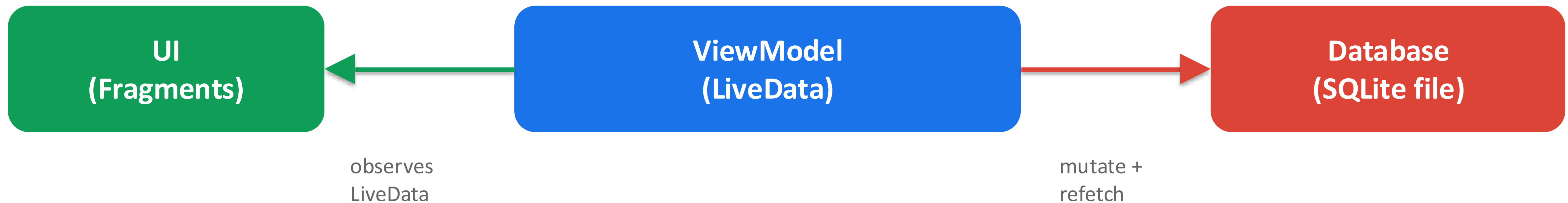
FireNote (Firebase)

```
fun createNote(text: String, pictureUUIDs: List<String>) {
    val note = Note(name = ..., text = text, ...)
    dbHelp.createNote(note) { freshList ->
        _notesList.value = freshList
    }
}
fun removeNoteAt(position: Int) {
    val note = getNote(position)
    note.pictureUUIDs.forEach { storage.deleteImage(it) }
    dbHelp.removeNote(note) { freshList ->
        _notesList.value = freshList
    }
}
```

The callback body { freshList -> _notesList.value = freshList } is character-for-character identical.
Only domain-specific setup differs (Firebase constructs a Note with auth user info, deletes cloud images).

Database as source of truth

After every mutation, the callback delivers a fresh list from the database.
This keeps the pattern identical for SQLite and Firebase.



observes
LiveData

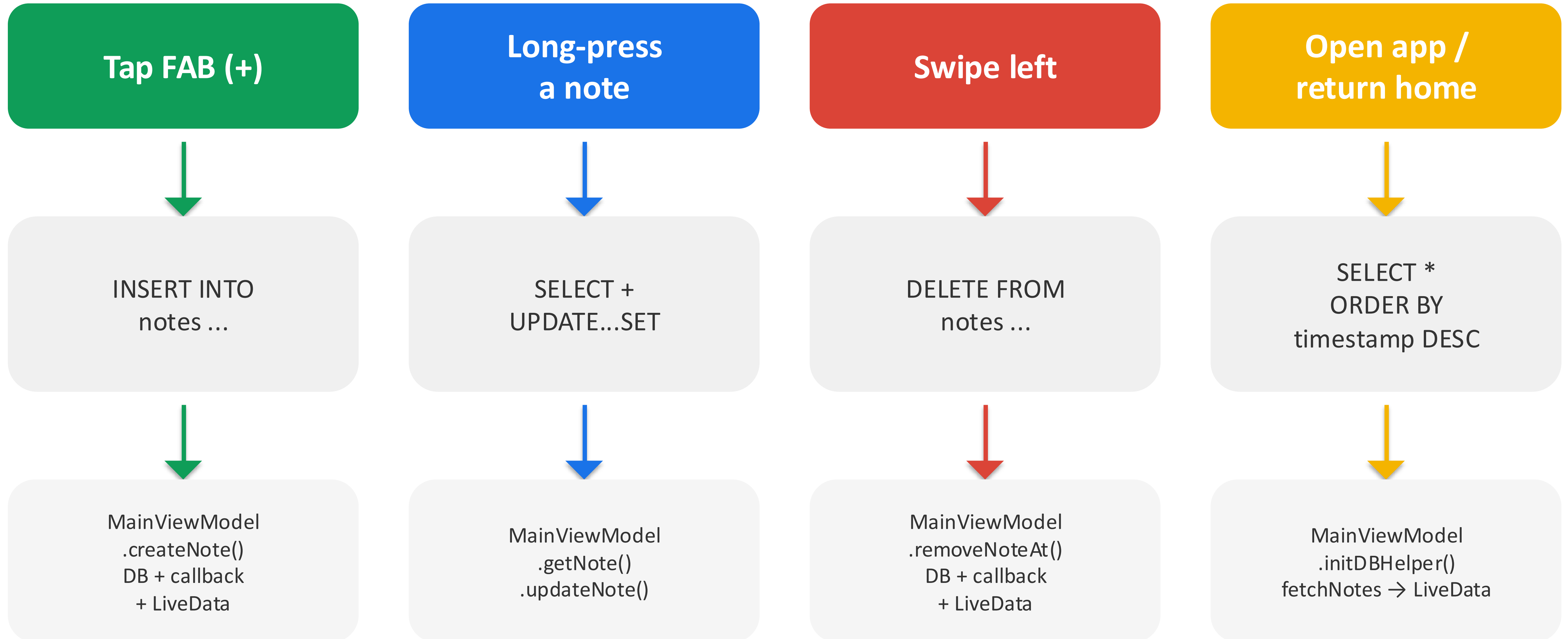
mutate +
refetch

Refetch pattern (MainViewModel.kt)

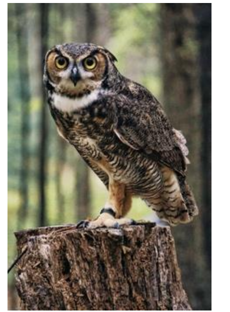
```
fun createNote(text: String, imageList: List<Image>) {
    dbHelp.createNote(text, imageList) { freshList ->
        _notesList.value = freshList  // LiveData notifies UI
    }
}
```

Refetch via callback + LiveData = simple, correct, backend-agnostic.

CRUD in the Notebook UI



Key takeaways



Tables beat files

Structured data gives you indexed lookup, typed columns, and relationships

SQL: say what, not how

CRUD covers everything. FK + CASCADE enforce integrity automatically

Layer your architecture

Repository hides SQL. Refetch via callback + LiveData keep UI in sync

Fragment **ViewModel** **Repository** **SQLite**