Cloud-Native Data with Firebase

What changes when your database lives on a server
you never manage

What is Firebase?

A suite of Google-hosted backend services.
No server code, no SQL, no migrations.

Firestore
NoSQL document database for structured app data

Storage
File hosting for images, video, user uploads

Auth
Email, Google, GitHub login with one API call

+ more
Analytics, Crashlytics, Cloud Functions, ML Kit

You write the mobile app. Firebase handles the backend.

Documents, not rows

Firestore organizes data as collections of documents, not tables of rows

SQL table: notes

id	note	timestamp
1	Buy groceries	2025-03-20 14:30
2	Call dentist	2025-03-21 09:15

Firestore document: allNotes/Xk9..mP

```
{
  "name": "Alice",
  "ownerUid": "uid_abc123",
  "text": "Buy groceries",
  "pictureUUIDs": ["img_01", "img_02"],
  "timeStamp": March 20, 2025 at 2:30 PM
}
```

No schema means no migrations — but also no type safety from the database. Your Kotlin data class is the schema.

A Note in Kotlin

SQLite needs a schema defined in SQL strings.
Firestore just needs a data class.

Notebook (SQLite)

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

FireNote (Firestore)

```
data class Note(
    var name: String = "",
    var ownerUid: String = "",
    var text: String = "",
    var pictureUUIDs: List<String> = listOf(),
    @ServerTimestamp
    val timeStamp: Timestamp? = null,
    @DocumentId
    var firestoreID: String = ""
)
```

@ServerTimestamp — server writes the time

@DocumentId — auto-generated unique key

The Firestore API

Everything starts with a collection reference.
Four operations cover all of CRUD.

Setup

```
val db = FirebaseFirestore.getInstance()
val collection = db.collection("allNotes")
```

Operation	SQL equivalent	Firestore call
Create	INSERT INTO notes ...	collection.add(note)
Read	SELECT * FROM notes ...	collection.orderBy("timeStamp").get()
Update	UPDATE notes SET ... WHERE id = ?	collection.document(id).set(note)
Delete	DELETE FROM notes WHERE id = ?	collection.document(id).delete()

Every call returns a Task — nothing blocks. You get the result in a callback, maybe seconds later.

Everything is async

SQLite returns immediately. Firestore returns a Task — your code has to be structured around callbacks.

SQLite (synchronous)

```
// Returns a row ID immediately
val id = db.insert("notes",
    null, contentValues)

// Continue with id right here
insertImages(id, images)
callback(db.allNotes)
```

Firestore (asynchronous)

```
// Returns a Task, not a result
db.collection("allNotes")
    .add(note)
    .addOnSuccessListener {
        Log.d(TAG, "Note created")
        dbFetchNotes(callback)
    }
    .addOnFailureListener { e ->
        Log.w(TAG, "Error", e)
    }
```

addOnSuccessListener / addOnFailureListener is the fundamental pattern for all Firebase operations.

Fetching data: one-shot queries

The Firestore equivalent of SELECT with ORDER BY and LIMIT

ViewModelDBHelper.kt — dbFetchNotes()

```
db.collection("allNotes")
    .orderBy("timeStamp", Query.Direction.DESCENDING)
    .limit(100)
    .get()
    .addOnSuccessListener { result ->
        Log.d(TAG, "fetch ${result.documents.size}")
        callback(result.documents.mapNotNull {
            it.toObject(Note::class.java)
        })
    }
    .addOnFailureListener {
        Log.d(TAG, "fetch FAILED", it)
    }
```

.orderBy() + .limit()
like SQL ORDER BY / LIMIT

.get() — one-shot fetch
returns Task<QuerySnapshot>

.toObject(Note::class.java)
deserializes via reflection —
needs default values + no-arg ctor

Mutate, then refetch

After every mutation, refetch from the server.
The database is always the source of truth.

ViewModelDBHelper.kt — createNote()

```
db.collection("allNotes")
    .add(note)
    .addOnSuccessListener {
        Log.d(TAG, "Note created id: ${note.firestoreID}")
        dbFetchNotes(callback)   // refetch → callback → LiveData
    }
```

Same pattern as the SQLite deck: mutate → refetch → callback → LiveData.
The round-trip just goes to the cloud now.

Server timestamps & document IDs

Two things the server knows better than the client

@ServerTimestamp

Server writes the time, not the client

No clock-skew problems across devices

Consistent ordering for all users

Field must be Timestamp? = null — Firestore fills it in on write

@DocumentId

Firestore auto-generates a unique ID for each document

No AUTOINCREMENT, no collisions across devices

The field is not stored inside the document — it is the document name

Set firestoreID = "" and Firestore populates it on read

Both annotations let the server provide values the client cannot reliably compute on its own.

What about real-time updates?

FireNote uses one-shot .get() — simple and sufficient.
But Firestore can also push changes to you.

One-shot (FireNote)

```
// Pull: you ask, server answers
db.collection("allNotes")
    .get()
    .addOnSuccessListener { result ->
        callback(result.toNotes())
    }

// Simple, predictable
// You control when data refreshes
```

Real-time listener (Hex)

```
// Push: server tells you when data changes
db.collection("games")
    .addSnapshotListener { snapshot, e ->
        if (e != null) return@add...
        callback(snapshot!!.toObjects(...))
    }

// Fires immediately with current data,
// then again on every server-side change
```

One-shot is enough for most apps.
Listeners shine when multiple users need to see each other's changes instantly (chat, games).

The Repository absorbs the change

The ViewModel doesn't know or care that data now goes to the cloud

MainViewModel.kt — same callbacks, same LiveData

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

{ _notesList.value = it } is the same callback whether data comes from SQLite or Firestore.

Key takeaways

Documents, not rows

No schema, no migrations.
Your Kotlin data class
is the schema

Everything is async

Callbacks, not return values.
addOnSuccessListener
is the core pattern

Server = source of truth

Timestamps, IDs, ordering.
Mutate → refetch →
callback → LiveData