



Database **setup**

A practical guide to choosing, modeling, and securing your iOS app's database with AI — from first schema to production.

WHO THIS GUIDE IS FOR

Beginners — if you've never set up a database before, this explains what your options actually are and why one is the sane default for almost everyone starting out.

Technical readers — if you already know SQL from NoSQL, this covers the specific trade-offs, AI-prompting patterns, and security pitfalls that matter once you're actually shipping.

SECTION	COVERS
01 · Foundations	Why the database decision is expensive to reverse, and how to think about it
02 · Choosing	Firestore vs. Realtime Database vs. Supabase
03 · Data modeling	Prompting AI for a schema, a worked example, and its NoSQL blind spots
04 · Security	A real firestore.rules file, AI's common mistakes, testing in the emulator
05–07 · Shipping	Wiring it into React Native, composite indexes, and the full workflow



01 · FOUNDATIONS & MINDSET

Pick boring, pick **proven**

Your database is one of the few decisions in this stack that's genuinely expensive to reverse. Every collection, query, and security rule you write gets built on top of whatever you pick now.

Relational (SQL)

Data lives in fixed-column tables, with keys linking related rows. Postgres and MySQL are classic examples. Rigid by design — you define a schema up front, and every row conforms to it. That's a feature when data is highly structured and interconnected.

Document-based (NoSQL)

Data lives in flexible, JSON-like documents grouped into collections, with no schema enforced by the database itself. Firestore is the example used throughout this guide. That flexibility is a feature when your data model is still evolving — true for almost every early-stage app.

WHY NOT THE NEWEST OPTION

This guide's defaults — Firestore and Firebase Realtime Database — aren't the newest or most-hyped tools on the market. They're mature, extensively documented, battle-tested by millions of apps, and priced predictably. That's exactly what you want underneath your data; save your appetite for new tools for parts of the stack where experimenting doesn't put user data at risk. It's the same reasoning behind the cautious stance on Supabase covered on the next page.

- a Migrating live data is a different order of problem.** Swapping a UI library later is annoying but doable — moving real user data with real downtime risk is not.
- b Pricing model matters as much as the schema.** Some platforms bill in ways that are easy to reason about small and punishing at growth scale — get this right before you need to.



02 · CHOOSING A DATABASE

Firestore is the default

For a typical app — social feed, marketplace, productivity tool, content app — Firestore should be your starting assumption unless you have a specific reason to reach for something else.

Firestore

Pay-per-use billing that tracks actual usage, not guessed capacity. Fully managed — nothing to provision or patch. Flexible document model suited to an evolving data shape. Deep integration with Firebase Auth, Storage, and Cloud Functions.

Firebase Realtime DB

The better tool specifically when latency is the whole point: live chat, live location tracking, collaborative multiplayer-style sync. A common pattern — Firestore for everything, Realtime DB bolted on just for the one or two features that need sub-second sync.

THE SUPABASE CAVEAT

Be cautious with Supabase — its pricing can escalate quickly and unpredictably as usage grows. The legitimate exception: a hard requirement for native Postgres/SQL — complex multi-table joins, strict schema enforcement, or a fundamentally relational data model (heavy financial ledgers, deeply interconnected records). If that's your situation, it's worth the added cost vigilance. For everyone else, it's an unnecessary risk.

	FIRESTORE	REALTIME DB	SUPABASE
Cost model	Pay-per-use	Pay-per-use, less granular	Tiered — can escalate fast
Data model	Document (NoSQL)	JSON tree	Relational (Postgres)
Queries	Indexed, no joins	Simple key-based only	Full SQL, joins
Best for	Most early-stage apps	Chat, live location, sync	Genuine relational needs



03 · DATA MODELING WITH AI

Give it a feature **list**

The most reliable way to get a usable first-draft schema is to hand Claude Code a concrete feature list rather than asking it to "design a database." Specificity gets a specific answer.

COPY-PASTE · STRONG SCHEMA PROMPT

Here's my app's feature list: users can create posts with images, follow other users, comment on posts, and like posts. Design a Firestore schema for this. Specify collections, subcollections, and document fields. Explain your reasoning for any denormalization decisions, and flag anywhere a document could grow unbounded.

Pitfalls AI-generated schemas tend to walk into

- 1 Careless denormalization.** Some duplication is normal, but AI will sometimes duplicate fields that change frequently — ask what happens when the source value changes.
- 2 Subcollections used incorrectly.** Data nested under a parent that actually needs to be queried independently (e.g. "all comments by this user") is hard to reach that way.
- 3 Unbounded document growth.** Firestore caps documents at 1MB. Watch any field described as a "list of X" and ask whether X has a natural ceiling.

CRITIQUE BEFORE YOU BUILD

Once you have a draft schema, paste it back and ask: "*What's wrong with this schema? Where will this break down as data grows, and where are the query patterns inefficient?*" AI is reliably better at critique than generation — run this pass before writing a single line of code against the schema.



03 · DATA MODELING WITH AI

What that actually **looks** like

"Collection", "subcollection" and "denormalization" are three words that only mean something once you've seen the shape they describe. Here is the schema from the prompt on page 4, drawn out.

THE SHAPE · COLLECTIONS AND SUBCOLLECTIONS

```
users/           (collection)
└─ {userId}      (document) ← one per user

posts/           (collection)
└─ {postId}      (document) ← one per post
    └─ comments/ (SUBcollection – a collection
        └─ {commentId} living inside one document)
```

ONE DOCUMENT · POSTS/{POSTID}

```
{
  "authorId": "u_8f2c",           // who owns this post
  "authorName": "Nelson",         // ← DENORMALIZED (copied)
  "authorPhoto": "https://.../u8f2c.jpg", // ← DENORMALIZED
  "body": "First post",
  "imageUrl": "https://.../p1.jpg",
  "likeCount": 42,                // a counter, not a list
  "createdAt": "2026-08-17T09:12:00Z"
}
```

- Why the name is copied.** Firestore has no joins. Without `authorName` on the post, rendering a 20-post feed means 20 extra reads to `users/` just to draw the bylines — slower and 20× the cost. Copying it is the fix, and it's called denormalization.
- What copying costs you.** When someone changes their display name, every post they've ever written now shows the old one. You need a plan for that — a Cloud Function that fans the update out, or a deliberate decision that stale bylines are acceptable. **Ask your agent this question about every duplicated field it proposes.**
- Why comments are a subcollection, not an array.** An array field lives inside the post document, which is capped at 1MB — a popular post would eventually hit it and start failing writes. A subcollection has no such ceiling, and lets you page comments in 20 at a time instead of loading all of them with the post.
- Why `likeCount` is a number.** Same reason: storing an array of every user who liked something is an unbounded field. Store the count on the post, and the individual likes as their own documents if you need to know who.



04 · SECURITY & RULES

Rules are your real **boundary**

Firestore security rules decide who can read or write which documents. They live independently of your app code — anyone can call the Firestore API directly, so rules are the actual security boundary, not a formality on top of it.

THE COMMON BEGINNER MISTAKE

Developing with rules wide open (`allow read, write: if true;`) to avoid friction, then shipping without ever tightening them. It's an easy trap because an open-rules app works perfectly in testing — the bug is invisible until someone else finds it. Treat real security rules as a required step before launch, not an optional hardening pass.

Common AI-generated security mistakes

- a Overly permissive rules.** Allowing any authenticated user to write to any user's document, rather than scoping the write to the document owner.
- b Missed auth checks on subcollections.** Main collections get locked down correctly while an edge-case path is left open.
- c Trusting client-supplied data in rules.** A rule that checks a field the client itself set — rather than `request.auth` — can often be spoofed.

DON'T SHIP THIS · FIRESTORE.RULES

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if true; // anyone, anywhere, any document
    }
  }
}
```

That's the wide-open version. The next page is the same schema written properly, with each of the three mistakes above visibly fixed.



04 · SECURITY & RULES

The rules file, *specified*

The schema from page 5 — users, their posts, and comments on those posts — with every read and write scoped to the person who owns the document. Ask for it in these terms and the three mistakes on the previous page can't appear.

COPY-PASTE · PROMPT · OWNER-SCOPED RULES

Write firestore.rules for this schema:

```
users/{userId}
posts/{postId}           (fields include authorId)
posts/{postId}/comments/{commentId}  (fields include authorId)
```

Requirements, all three of which I want visible in the file:

1. A user may only read and write their own users/{userId} document. Extract a helper (isOwner) rather than repeating the condition — every write must be scoped to the document's OWNER, not to any signed-in user.
2. Write a separate match block for the comments subcollection. Rules do not cascade, so a rule on posts grants nothing beneath it — I want that block written out explicitly even though it looks redundant.
3. On create, compare the incoming authorId against request.auth.uid — the VERIFIED token — never against a field in the request payload. On update and delete, check the existing document's authorId the same way.

Then explain, line by line, who can do what. I want to be able to read the file back and describe it myself.

Those three requirements are the previous page's three failure modes, written as instructions. **a.** Owner-scoping fixes "overly permissive". **b.** The explicit `comments` block fixes "missed subcollections". **c.** Comparing against `request.auth.uid` fixes "trusting client data" — the payload's `authorId` is whatever the caller typed; the token is verified. Read the generated file against all three before you deploy it.



04 · SECURITY & RULES

Prove them **locally**

The emulator runs Firestore and Auth on your own machine, so you can fire deliberately forbidden requests at your rules without touching production data.

COPY-PASTE · TERMINAL · EMULATOR AND DEPLOY

```
# Install the Firebase CLI and emulator suite
npm install -g firebase-tools
firebase login
firebase init emulators      # select Firestore + Auth
firebase emulators:start

# Run your rules tests against the emulator
npm test

# Deploy the rules only once the tests pass
firebase deploy --only firestore:rules
```

COPY-PASTE · PROMPT · RULES TESTS

Write emulator tests for my firestore.rules using
@firebase/rules-unit-testing, covering BOTH directions:

- Requests that should SUCCEED (a user reads and writes their own document)
- Requests that should FAIL (a user reads another user's document, an unauthenticated write, a write to a comments subcollection under someone else's post, and a create where authorId doesn't match the auth uid)

I want the failing cases asserted explicitly, not assumed.

WHY THE FAILING CASES ARE THE POINT

A test suite that only checks allowed requests passes just as happily against `allow read, write: if true;` — it proves nothing. The forbidden cases are the only ones that demonstrate your rules do anything at all. Make `npm test` the thing you run before every `firebase deploy`: a rule loosened for one debugging session has a way of surviving into production.



05 · CONNECTING TO YOUR STACK

Wire it into **React Native**

Following the recommended stack, connecting Firestore to a React Native app is a short, ordered sequence — get sign-in working before you build out data access, not after.

STEP 1**Install the SDK**

Use `@react-native-firebase/firestore` — it integrates more natively with Expo/RN than the web SDK.

**STEP 2****Set up auth**

Firebase Authentication first — your security rules will almost always key off `request.auth.uid`.

**STEP 3****Initialize once**

Config (API keys, project ID) lives in one dedicated file the rest of your app imports.

**STEP 4****Scaffold hooks**

Prompt Claude Code for typed hooks like `useUserPosts()` instead of raw calls scattered through components.

COPY-PASTE · TERMINAL · INSTALL

```
npx expo install @react-native-firebase/app \
  @react-native-firebase/auth @react-native-firebase/firestore
npx expo install expo-build-properties
```

```
# Native modules don't exist in Expo Go — from here you run:
npx expo run:ios --device
```

In the Firebase console, add an iOS app whose bundle ID matches `app.json` exactly, download `GoogleService-Info.plist`, and place it at your project root. Then reference it from `app.json` under `ios.googleServicesFile`, and add the React Native Firebase plugins to the `plugins` array.



05 · CONNECTING TO YOUR STACK

Initialize it **once**

One file that sets up Firebase and exports the handles, and one convention for anything environment-specific. Everything else in your app imports from here.

COPY-PASTE · PROMPT · INITIALIZE ONCE

Create `src/lib/firebase.ts` as the single initialization point for Firebase in this Expo app.

- Import from `@react-native-firebase/firestore` and `@react-native-firebase/auth`
- Export the firestore and auth handles, plus a typed reference per top-level collection (users, posts), so the rest of the app imports from this file and never calls `firestore().collection(...)` inline
- No config object: the native SDK reads `GoogleService-Info.plist` automatically. If you find yourself pasting API keys here, stop and tell me why

Then tell me which existing files should be changed to import from here instead of initializing their own handles.

ENVIRONMENT VARIABLES & CONFIG

The plist covers Firebase itself. Anything else environment-specific (a project ID for scripts, third-party keys) goes in `.env` and is read via `app.config.js`. Firebase config values aren't secrets in the traditional sense, but different environments need different configs and hardcoding one makes switching error-prone. Never commit `.env` — commit a `.env.example` template instead.

```
echo ".env" >> .gitignore
echo "GoogleService-Info.plist" >> .gitignore
```

THEN SCAFFOLD THE DATA LAYER

With this file in place, ask Claude Code for typed hooks — `useUserPosts()`, `useCreatePost()` — that import `db` from here, rather than letting raw `firestore().collection(...)` calls scatter through your components. When the schema changes, and it will, you want one layer to update instead of thirty screens.



06 · COMMON PITFALLS

What breaks in **production**

None of these are exotic — they're the kind of thing that looks completely fine in a schema diagram and only shows up once you have real users and real data volume.

Modeling before query patterns

Firestore doesn't support joins — a schema that looks clean on a whiteboard can force fetching far more documents than needed to assemble one screen. Work backward from your actual screens first.

Ignoring indexes until it's slow

Firestore requires a composite index for most queries that filter on one field and sort on another, and only tells you once you run that query — often from a screen you built weeks earlier. Worked example on the next page.

No backup/export strategy

A bad migration script or an overly broad rule can destroy real user data in seconds. Set up scheduled exports before you have real users depending on that data.

No billing alerts

Pay-per-use billing is friendly at low usage and unforgiving when a bug causes runaway reads — a re-subscribing listener on every render is a common culprit. Set a billing alert before launch.



06 · COMMON PITFALLS

The index you'll forget

This is the most common "it worked yesterday" bug in Firestore, and it's worth seeing once before it happens to you at 2am.

THE QUERY THAT TRIGGERS IT

```
// "My posts, newest first" – one filter, one sort,  
// on two DIFFERENT fields. This needs a composite index.  
const snap = await db.collection('posts')  
  .where('authorId', '==', uid)  
  .orderBy('createdAt', 'desc')  
  .limit(20)  
  .get();
```

WHAT FIRESTORE THROWS INSTEAD OF RESULTS

[firestore/failed-precondition] The query requires an index.
You can create it here: https://console.firebase.google.com/project/YOUR-PROJECT/firestore/indexes?create_composite=Ck...

That link pre-fills the exact index, and clicking it is fine for a one-off — but the index then exists only in that project, not in your repo, so a teammate or a fresh environment hits the same error again. Declare it in a file instead:

COPY-PASTE · PROMPT · DECLARE THE INDEXES

Search this project for every Firestore query, and write `firestore.indexes.json` declaring the composite indexes they require.

- A query that filters on one field and sorts on another needs a composite index; single-field indexes are automatic, so don't declare those
- List each index next to the query that needs it, so I can check them against my screens
- Flag any query you find that will need an index I haven't hit yet in testing

Deploy it with `firebase deploy --only firestore:indexes,firestore:rules`. Single-field indexes are created automatically — it's the filter-plus-sort and multi-filter combinations that need declaring. Large indexes take minutes to build, so create them before launch, not while users stare at a broken screen.



07 · PRACTICAL WORKFLOW

The full **sequence**

From feature list to a database you can ship on — the six steps in order.

- 1 Define your data needs** from your feature list — what you'll store and how each piece will be queried.
- 2 Sketch a schema** by prompting Claude Code with that feature list.
- 3 Get an AI critique** of the draft — denormalization, subcollection misuse, unbounded growth.
- 4 Implement security rules**, review them line by line, and test in the emulator.
- 5 Connect the database** — SDK, auth flow, environment config, then a data-access layer of hooks.
- 6 Test with real usage patterns** — not just the happy path — to catch missing indexes and overbroad rules before you ship.

Stage-by-stage reference

STAGE	TOOL / DECISION
Default database	Firestore (pay-per-use)
Real-time features	Firebase Realtime Database
Relational/SQL need	Supabase — only if genuinely needed
Schema + review	Claude Code, then an AI critique pass
Access control	Firestore rules, tested via local emulator
Ongoing safety	Scheduled backups + Firebase billing alerts



07 · PRACTICAL WORKFLOW

Before real users arrive

Two five-minute jobs, and where this guide hands off to the next one.

DO BOTH OF THESE BEFORE LAUNCH

Set a **Firebase billing alert**, and turn on **scheduled exports**. Both take about five minutes, both are invisible until the day one of them saves you, and neither can be applied retroactively — a runaway listener bills you for reads that already happened, and a bad migration script destroys data that was never backed up.

Where to go next

GUIDE	PICKS UP WITH
The AI Coding Stack	The rest of the stack this database sits inside — the framework, the MCPs that feed your agent, and deployment
Launching on the App Store	Submitting the finished app, including the App Privacy disclosures your schema implies
Idea to App Store	The combined guide — all seven stages end to end, from idea through analytics