# 18 · Firebase

Flutter Practice Notes

## Contents

# 18 · Firebase

## Introduction

Firebase is Google's Backend-as-a-Service (BaaS): auth, databases (Firestore/Realtime DB), storage, serverless functions, messaging, analytics, and crash reporting — accessed from Flutter via **FlutterFire** plugins. This module covers setup and each major service, with security and architecture guidance.

## Why this module exists

Firebase lets small teams ship full-stack apps fast without running servers. But its convenience hides sharp edges: security rules, cost/read-count modeling, offline behavior, and vendor lock-in. Using it *well* (behind repositories, with proper rules) is the skill.

## Module map

| # | File | Topic | Level |
|---|---|---|---|
| 1 | 01_firebase_setup_and_core.md | FlutterFire setup, init, project config | 🟢 |
| 2 | 02_firebase_auth.md | `FirebaseAuth`, providers, state | 🔵 |
| 3 | 03_firestore.md | NoSQL modeling, queries, real-time, offline, rules | 🔴 |
| 4 | 04_realtime_db_and_storage.md | Realtime Database + Cloud Storage | 🔵 |
| 5 | 05_cloud_functions.md | Serverless functions, triggers, callable | 🔵 |
| 6 | 06_observability_and_messaging.md | Crashlytics, Analytics, FCM, Remote Config, App Check | 🔵 |

> **Cross-references:** Auth patterns: Module 17. Repository boundary: 05 · repository. Offline/sync: Module 19. Notifications: Module 32. Monitoring: Module 52. Security: Module 37. Streams: 02 · streams.

## Prerequisites

16 Networking, 17 Authentication, 05 · repository, 02 · streams.

## What you'll be able to do after this module

- Set up FlutterFire and initialize Firebase correctly (dev/prod).
- Implement auth with `FirebaseAuth` + providers and reactive auth state.
- Model, query, and stream Firestore data with security rules and offline support.

- Use Realtime Database, Storage, Cloud Functions, FCM, Crashlytics/Analytics, Remote Config, App Check.
- Wrap Firebase behind repositories to limit lock-in and stay testable.

## Capstone

**Firebase-backed app slice:** Email/Google auth → Firestore-backed data (real-time + offline) with security rules → file upload to Storage → a Cloud Function trigger → Crashlytics + Analytics + FCM — all behind repositories.

## Summary

Firebase is a fast, integrated BaaS. Use its services through repository boundaries, write strict security rules, model for cost (read counts), and be deliberate about lock-in. Powerful for MVPs and many production apps when used with discipline.

# Firebase Setup & Core (FlutterFire)

> FlutterFire is the official set of Firebase plugins for Flutter; you register your app in a Firebase project, generate config via the **FlutterFire CLI**, add `firebase_core`, and call `Firebase.initializeApp()` before `runApp` — then layer service plugins on top.

## Introduction

Every Firebase feature depends on core setup: a Firebase project, per-platform app registration, generated `firebase_options.dart`, and initialization. This file covers the setup flow, multi-environment (dev/prod) config, and initialization ordering.

## Why this concept exists

Firebase services need project credentials and platform-specific config (API keys, bundle ids). The FlutterFire CLI automates generating these into `firebase_options.dart`, avoiding manual, error-prone plist/json wrangling — and initialization must happen before any service is used.

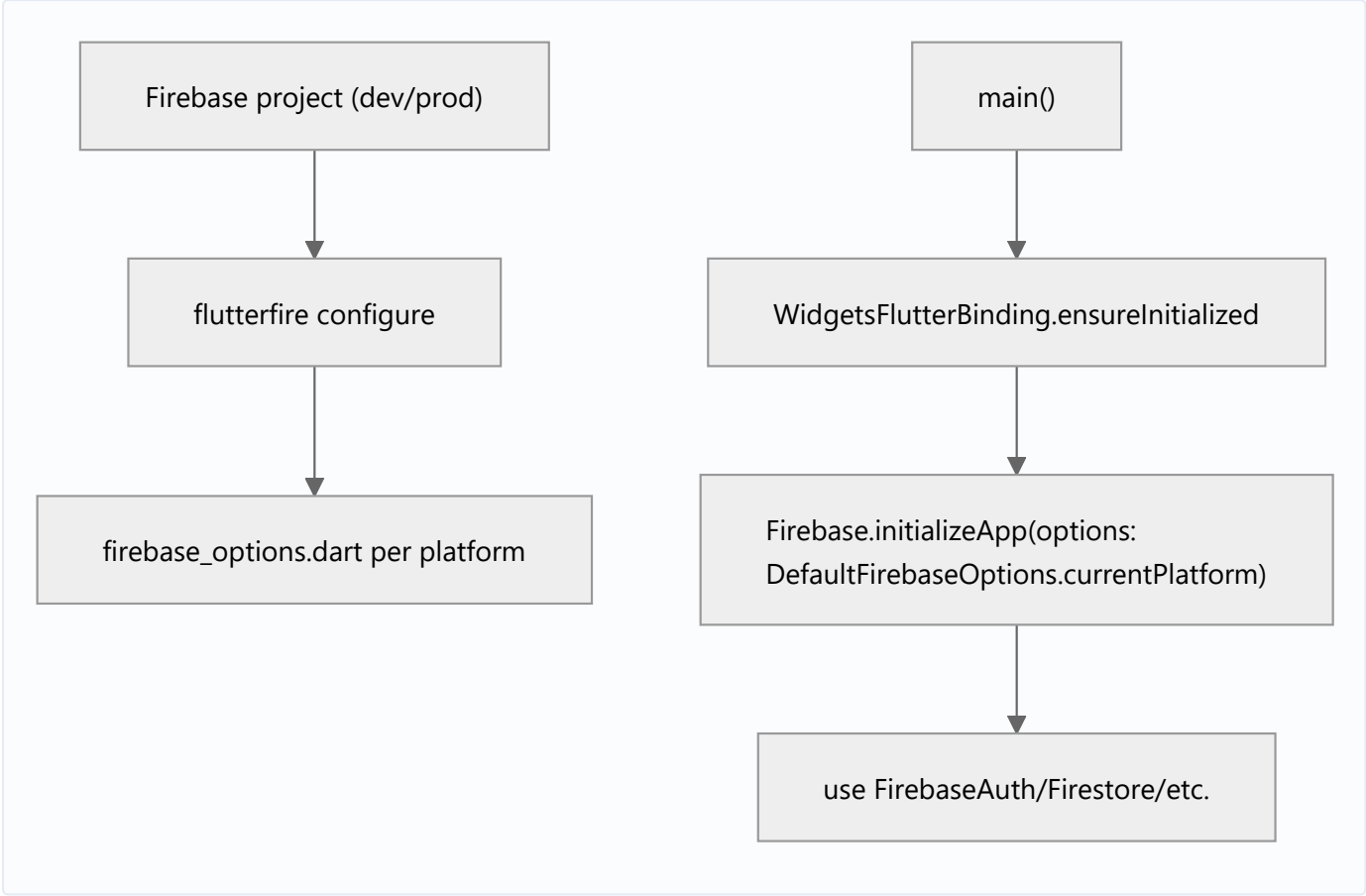
## Real-world analogy

Setup is **connecting your appliance to utilities**: you register with the utility company (Firebase project), get a meter installed per unit (per-platform app registration), receive your account config (`firebase_options.dart`), and flip the main breaker (`initializeApp`) before anything runs.

## Problem Statement

Initialize Firebase in a Flutter app for Android + iOS with separate dev and prod projects, ensuring services are ready before the app runs. You'll use the FlutterFire CLI + `Firebase.initializeApp` with flavors.

## Internal Working




- **Project + apps**: create a Firebase project; register Android (package name + SHA for some features), iOS (bundle id), web, etc.
- **FlutterFire CLI**: `flutterfire configure` generates `firebase_options.dart` (`DefaultFirebaseOptions.currentPlatform`) — no manual `google-services.json` / `GoogleService-Info.plist` copying for config (though platform files may still be needed for some services/build steps).
- **Core plugin**: `firebase_core`; add per service (`firebase_auth`, `cloud_firestore`, ...).
- **Init**: in `main`, `WidgetsFlutterBinding.ensureInitialized()` then `await Firebase.initializeApp(options: ...)` **before** `runApp` (06 · app_entry_point).
- **Multi-environment (dev/prod)**: separate Firebase projects; use **build flavors** and per-flavor options (multiple `firebase_options` or `--project`/entrypoints) so dev doesn't write to prod data.

## Memory Representation

Firebase instances are singletons (`FirebaseAuth.instance`, `FirebaseFirestore.instance`); wrap them behind repositories/DI (14 DI).

## Compiler Behavior

`firebase_options.dart` is generated code (regenerate on config changes). Web vs mobile differ — the CLI handles per-platform options.

## Runtime Behavior

Using any Firebase service before `initializeApp` throws. Init is async (network/plugin setup); pre-`runApp` init adds a little startup time — keep it minimal, defer non-critical service init (10 · embedder_and_startup).

## Flutter Engine Behavior

Plugins cross the embedder to native Firebase SDKs; requires platform config (Gradle/CocoaPods) the CLI/`flutter` tooling sets up (10 · embedder_and_startup).

## Dart VM Behavior

Not applicable.

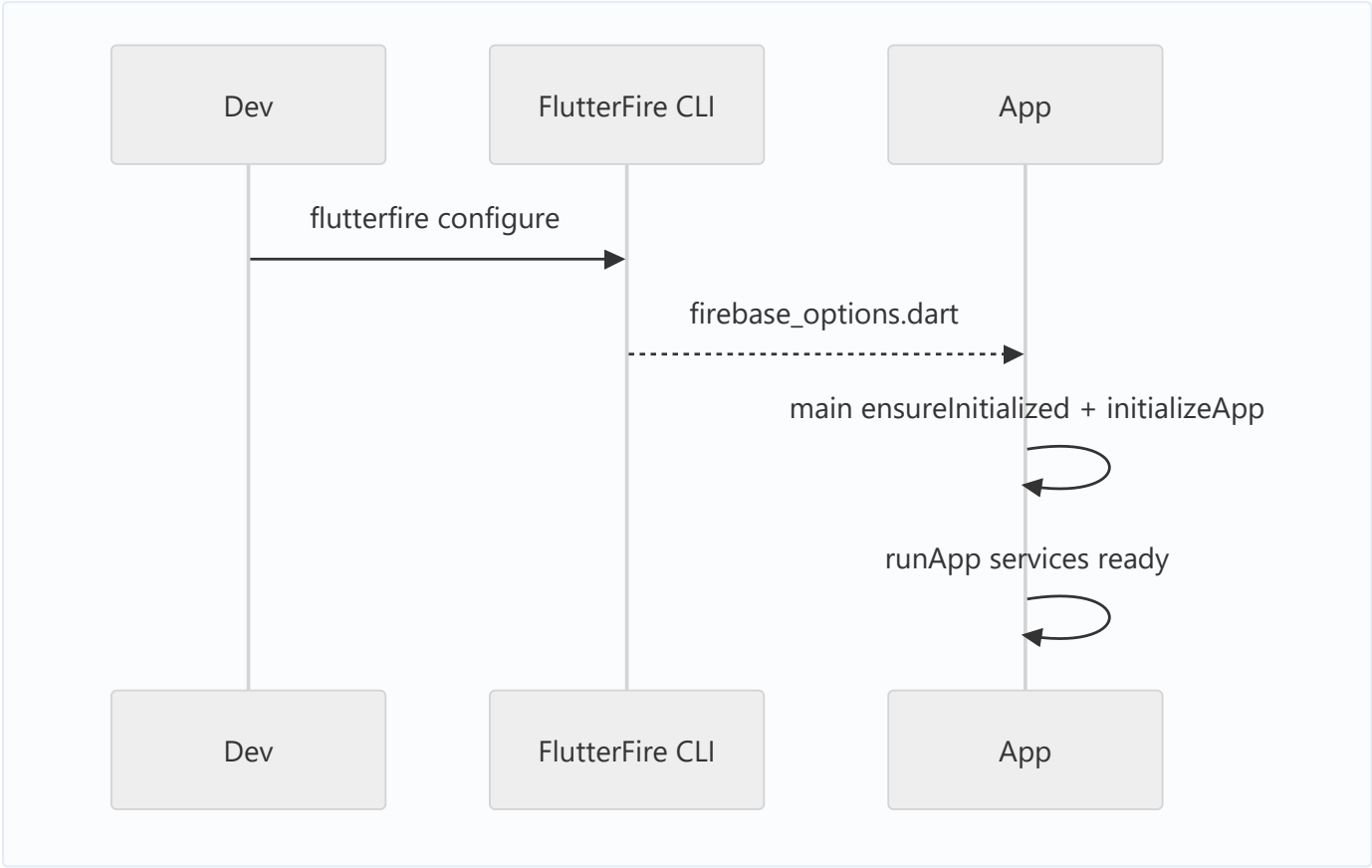
## Examples

```yaml
# pubspec.yaml
dependencies:
  firebase_core: ^3.0.0
  firebase_auth: ^5.0.0
  cloud_firestore: ^5.0.0
```

```
import 'package:firebase_core/firebase_core.dart';
import 'package:flutter/material.dart';
import 'firebase_options.dart'; // generated by `flutterfire configure`

Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();      // required before async init
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform, // per-platform config
  );
  runApp(const MyApp());                           // now services are safe to use
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) =>
      const MaterialApp(home: Scaffold(body: Center(child: Text('Firebase ready'))));
}
```

```
Setup steps:
  dart pub global activate flutterfire_cli
  flutterfire configure --project=my-dev-project   # generates firebase_options.dart
  # For flavors: separate projects + per-flavor options / entrypoints (main_dev.dart, main_prod.dart)
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Using services before `initializeApp` | Throws | `await Firebase.initializeApp()` first |
| Missing `ensureInitialized()` | Binding not ready | Call it before async init |
| One project for dev + prod | Dev pollutes prod data | Separate projects + flavors |
| Committing sensitive config carelessly | Some keys OK public, but rules matter | Rely on security rules + App Check (37) |
| Heavy Firebase init blocking startup | Slow cold start | Init core only; defer non-critical services |

## Best Practices

- Use the **FlutterFire CLI** to generate/refresh `firebase_options.dart`.
- `ensureInitialized()` → `await Firebase.initializeApp()` **before** `runApp`; keep it lean.
- Use **separate Firebase projects for dev/prod** with build flavors/entrypoints.
- Wrap Firebase singletons behind **repositories/DI** to limit lock-in and enable testing.
- Rely on **security rules + App Check** (public client keys are expected; rules are the real defense — 37, 06_observability_and_messaging.md).

## Performance

Init is a one-time startup cost; keep pre-`runApp` work minimal and defer non-critical service init post-first-frame (10 · embedder_and_startup, Module 21).

## Advantages / Disadvantages

- + Fast full-stack setup, official tooling, cross-platform config automation, integrated services.
- − Vendor lock-in, per-platform/flavor config complexity, startup init cost, security depends on rules (not client secrecy).

## Interview Questions

1. 🟢 **What is FlutterFire?** — The official set of Firebase plugins for Flutter (`firebase_core` + per-service packages).
2. 🟢 **How do you initialize Firebase?** — `WidgetsFlutterBinding.ensureInitialized()` then `await Firebase.initializeApp(options: DefaultFirebaseOptions.currentPlatform)` before `runApp`.
3. 🟡 **What does the FlutterFire CLI generate?** — `firebase_options.dart` with per-platform configuration (`DefaultFirebaseOptions.currentPlatform`).
4. 🟡 **How do you separate dev and prod?** — Separate Firebase projects + build flavors/entrypoints with per-flavor options, so dev never writes to prod data.
5. 🟡 **What happens if you use a service before init?** — It throws; initialization must complete first.
6. 🔴 **Are Firebase client API keys secret?** — No — they're generally public identifiers; real security comes from **security rules** and **App Check**, not key secrecy.
7. 🔴 **How do you limit Firebase lock-in?** — Wrap services behind repositories/abstractions so the domain doesn't depend on Firebase types directly.

## Senior Engineer Tips

- Keep Firebase types out of your domain — repositories map Firebase docs/results to entities so you could swap backends and stay testable (05 · repository).
- Set up dev/prod projects and flavors from day one; accidental prod writes from dev are painful.
- Defer non-critical Firebase init (analytics, remote config) until after the first frame for faster cold start.

## Architect Perspective

Firebase setup is the foundation and the lock-in decision point. Isolating Firebase behind repositories/DI, using separate environments, and treating rules/App Check as the security layer keeps the app testable, secure, and portable — essential before layering on Firestore, functions, and messaging (03_firestore.md, Module 37).

## Summary

- FlutterFire + `firebase_core`; generate config via the CLI; `ensureInitialized()` → `initializeApp()` before `runApp`.
- Separate dev/prod projects + flavors; wrap services behind repositories/DI.
- Client keys aren't secrets — security is rules + App Check; keep init lean.

## Revision Notes

- FlutterFire: `firebase_core` + service plugins; `flutterfire configure` → `firebase_options.dart`.
- `ensureInitialized()` → `await Firebase.initializeApp(options: currentPlatform)` before `runApp`.
- Dev/prod = separate projects + flavors/entrypoints.
- Repositories over Firebase types (lock-in/testing); security = rules + App Check (keys are public).

## Practice Questions

1. What must happen before using any Firebase service?
2. How do you keep dev writes out of prod?
3. Why aren't Firebase client keys a security concern by themselves?

## Coding Questions

1. Initialize Firebase in `main` with generated options.
2. Set up dev/prod entrypoints with per-flavor options.
3. Wrap `FirebaseAuth.instance` behind an injected repository.

## Mini Project

**Firebase bootstrap (Flutter):** Configure a Firebase project via the CLI, initialize in `main` (dev/prod entrypoints), and expose a `FirebaseService`/repository wrapping a core instance behind DI. Defer a non-critical service init post-first-frame. Acceptance: init before `runApp`; dev/prod separated; Firebase types wrapped; app runs.

# Firebase Authentication

> `FirebaseAuth` provides managed authentication — email/password, phone, and federated providers (Google/Apple/etc.) — with a reactive `authStateChanges()` stream and server-issued ID tokens, so you get a full auth backend without running one.

## Introduction

FirebaseAuth (package: `firebase_auth`) handles sign-up/in/out, provider linking, and session persistence, exposing the current user and an auth-state stream. This file covers the providers, the reactive auth state (driving guards/UI), ID tokens, and wrapping it behind an `AuthRepository`.

## Why this concept exists

Building secure auth (password hashing, token issuance, provider integration, session persistence) is hard and risky. Firebase Auth provides it managed, integrating with the rest of Firebase (Firestore rules use `request.auth`), so teams focus on the app, not auth infrastructure.

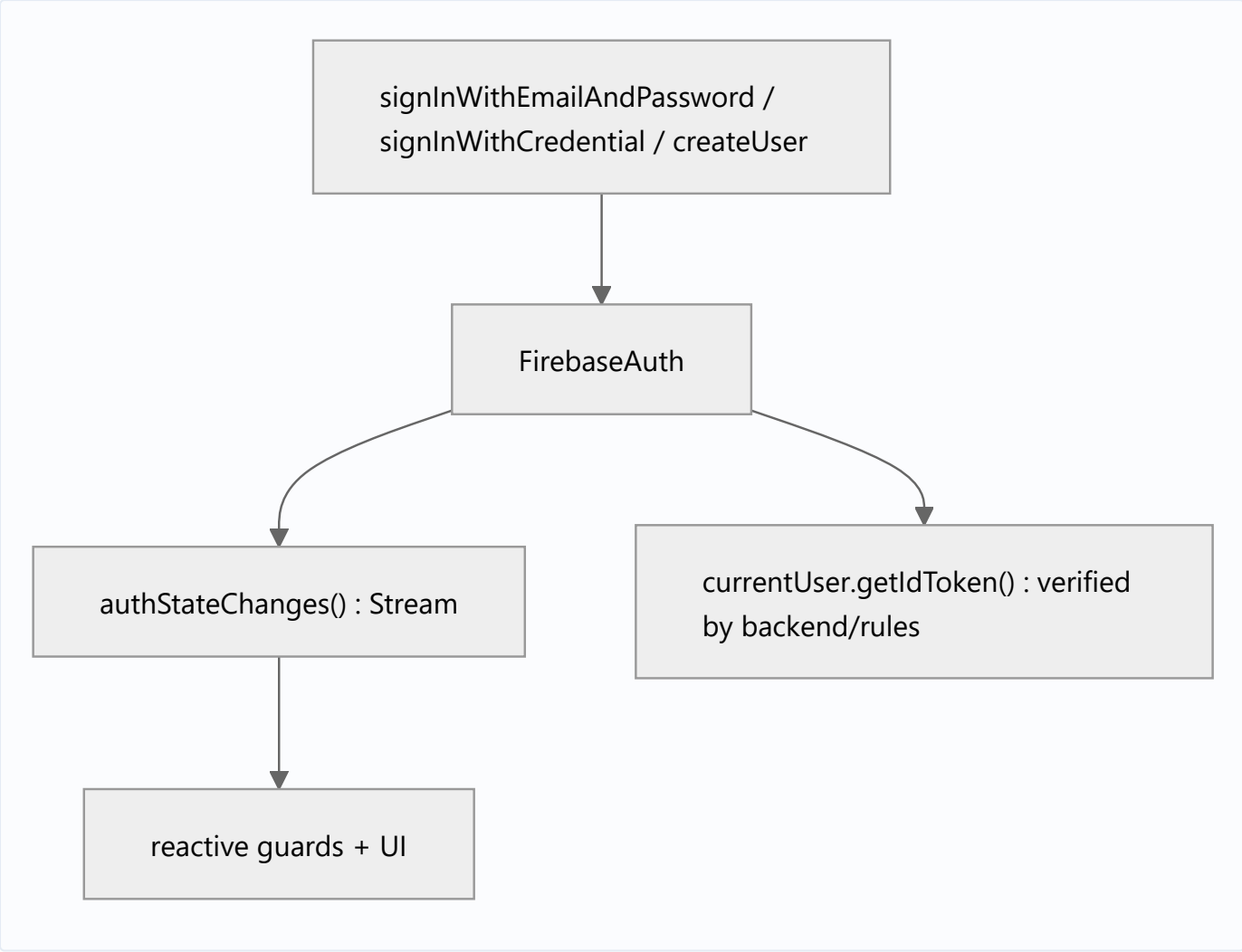
## Real-world analogy

FirebaseAuth is an **outsourced security desk**: it checks IDs (email/password/social), issues badges (ID tokens), remembers who's inside (session persistence), and broadcasts entry/exit (auth-state stream) — you don't run the desk, you consume its decisions.

## Problem Statement

Implement email/password + Google sign-in, expose reactive auth state to drive route guards, and access the ID token for backend/Firestore. You'll use `FirebaseAuth` behind an `AuthRepository`.

## Internal Working




- **Methods**: `createUserWithEmailAndPassword`, `signInWithEmailAndPassword`, `signInWithCredential` (federated — combine with `google_sign_in` / `sign_in_with_apple` from 17 · oauth), `signInAnonymously`, phone auth, `sendPasswordResetEmail`, `signOut`.
- **Reactive state**: `authStateChanges()` (`Stream<User?>`) emits on sign-in/out/token refresh — feed it to your auth state / router `refreshListenable` (17 · session_management, 13 · guards).
- **ID token**: `user.getIdToken()` returns a verifiable JWT; send it to your backend (verify server-side) or rely on it in Firestore rules (`request.auth.uid`).
- **Persistence**: Firebase persists the session (auto-restores on launch) — no manual token storage needed for Firebase-only apps.
- **Errors**: `FirebaseAuthException` with codes (`wrong-password`, `email-already-in-use`, `user-not-found`) — map to domain failures.

## Memory Representation

`FirebaseAuth.instance` is a singleton; the current user + session are managed by the SDK (persisted natively). Wrap behind a repository (05 · repository).

## Compiler Behavior

Not applicable.

## Runtime Behavior

Auth methods are async and may throw `FirebaseAuthException`; `authStateChanges()` emits reactively. ID tokens auto-refresh; `getIdToken(true)` forces refresh.

## Flutter Engine Behavior

Crosses the embedder to native Firebase Auth SDKs; federated flows may open provider UIs.

## Dart VM Behavior

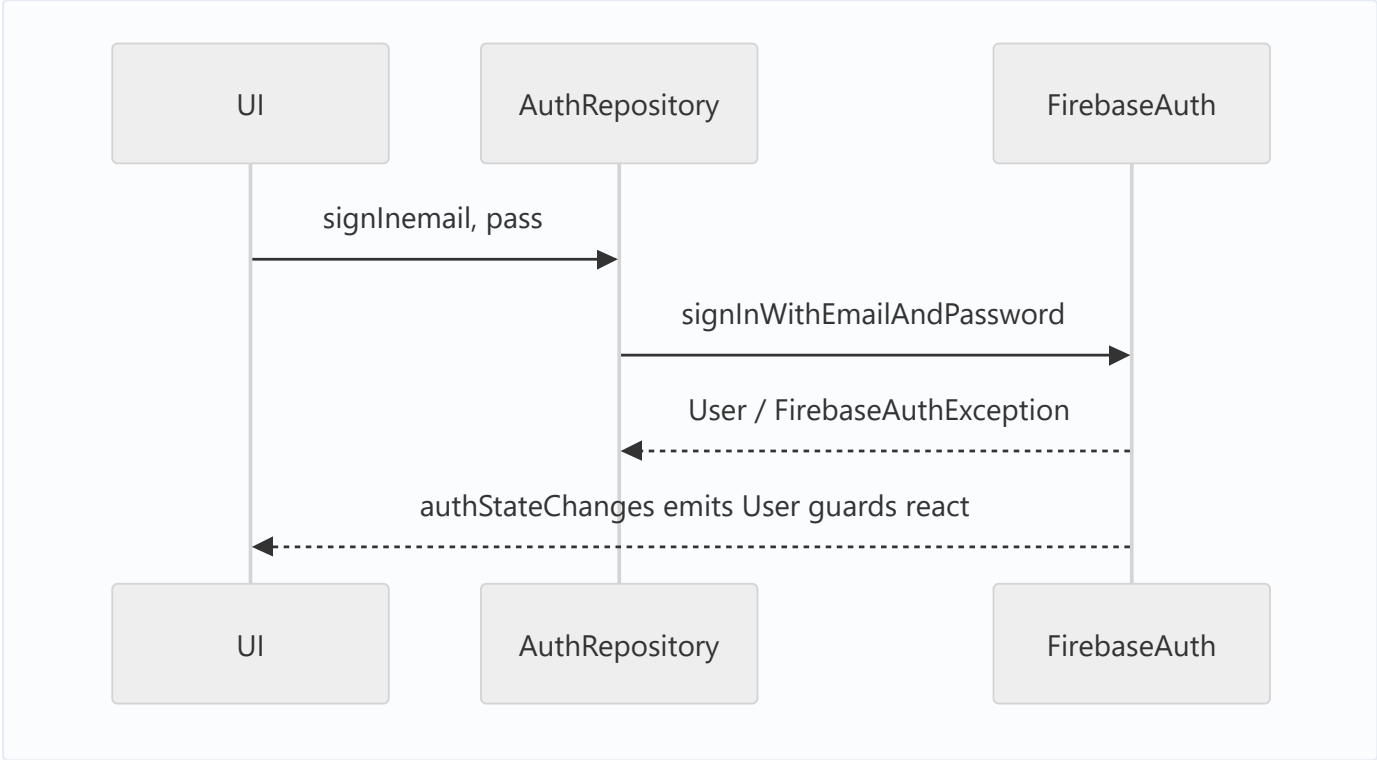
Not applicable.

## Examples

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:google_sign_in/google_sign_in.dart';

class FirebaseAuthRepository {
  final FirebaseAuth _auth;
  FirebaseAuthRepository([FirebaseAuth? auth]) : _auth = auth ?? FirebaseAuth.instance;

  // Reactive auth state -> drives guards/UI
  Stream<User?> get authState => _auth.authStateChanges();
  User? get currentUser => _auth.currentUser;

  Future<void> signUp(String email, String password) async {
    try {
      await _auth.createUserWithEmailAndPassword(email: email, password: password);
    } on FirebaseAuthException catch (e) {
      throw _mapError(e); // map to domain failure
    }
  }

  Future<void> signIn(String email, String password) async {
    try {
      await _auth.signInWithEmailAndPassword(email: email, password: password);
```

```
    } on FirebaseAuthException catch (e) {
      throw _mapError(e);
    }
  }

  Future<void> signInWithGoogle() async {
    final account = await GoogleSignIn().signIn();
    if (account == null) return; // cancelled
    final gAuth = await account.authentication;
    final cred = GoogleAuthProvider.credential(
      idToken: gAuth.idToken, accessToken: gAuth.accessToken);
    await _auth.signInWithCredential(cred); // federated sign-in
  }

  Future<String?> idToken() => _auth.currentUser?.getIdToken(); // for backend/rules
  Future<void> signOut() => _auth.signOut();

  Object _mapError(FirebaseAuthException e) => switch (e.code) {
        'wrong-password' || 'user-not-found' => 'Invalid credentials',
        'email-already-in-use' => 'Email already registered',
        _ => 'Auth error: ${e.code}',
      };
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Manually storing Firebase tokens in secure storage | Firebase persists sessions itself | Rely on `authStateChanges()` /persistence |
| Ignoring `FirebaseAuthException` codes | Poor error UX | Map codes to domain failures |
| Trusting client `currentUser` for authorization | Spoofable | Enforce via Firestore rules / backend token verification |
| Not driving guards from `authStateChanges()` | Stale routing | Feed the stream to `refreshListenable` |
| Leaking `User` /Firebase types to UI | Lock-in/coupling | Map to a domain user in the repository |

## Best Practices

- Drive **reactive auth state** from `authStateChanges()`; feed guards/UI (13, 17 · session_management).
- Rely on Firebase's **built-in session persistence** (no manual token storage for Firebase-only apps).
- Map `FirebaseAuthException` **codes** to friendly domain failures.
- **Verify ID tokens server-side** (or via Firestore rules `request.auth`) — never trust client `currentUser` for authorization.

- Wrap in an `AuthRepository` mapping `User` →domain user (limit lock-in, testability).
- Combine federated sign-in with `google_sign_in` / `sign_in_with_apple` (17 · oauth).

## Performance

Managed and efficient; `authStateChanges()` is a cheap stream. Token refresh is automatic. Negligible client cost.

## Advantages / Disadvantages

- + Managed secure auth, many providers, reactive state, built-in persistence, integrates with Firestore rules, fast to build.
- − Vendor lock-in, less control than custom auth, ties into the Firebase ecosystem, client user isn't an authorization source.

## Interview Questions

1. 🟢 **What does `FirebaseAuth` provide?** — Managed authentication (email/phone/federated), session persistence, current user, and a reactive auth-state stream.
2. 🟢 **How do you observe auth changes?** — `authStateChanges()` returns a `Stream<User?>` that emits on sign-in/out/token refresh.
3. 🟡 **How do you do Google sign-in with Firebase?** — Get a Google credential (via `google_sign_in`) and call `signInWithCredential(GoogleAuthProvider.credential(...))`.
4. 🟡 **Do you need to store Firebase tokens manually?** — No; Firebase persists the session and auto-restores it; use `authStateChanges()` / `currentUser`.
5. 🟡 **How are auth errors surfaced?** — As `FirebaseAuthException` with codes (`wrong-password`, etc.) — map to domain failures.
6. 🔴 **How is authorization enforced with Firebase Auth?** — Server-side: Firestore/Storage **security rules** using `request.auth.uid`, or backend verification of the ID token — not client `currentUser`.
7. 🔴 **How do you limit lock-in?** — Wrap `FirebaseAuth` behind an `AuthRepository` mapping `User` →domain user, so the app doesn't depend on Firebase types.

## Senior Engineer Tips

- Treat `authStateChanges()` as the single source of truth for session; drive router guards + UI from it.
- Never authorize on client `currentUser`; put authorization in Firestore rules / backend token checks.

- Keep Firebase `User` out of your domain — map it in the repository so swapping auth backends is feasible.

## Architect Perspective

Firebase Auth gives a production auth backend with reactive state and ecosystem integration (rules), letting you skip auth infrastructure. Wrapped behind a repository with server-enforced authorization, it fits the same session/guard architecture as custom auth (Module 17) while trading control for speed and lock-in.

## Summary

- `FirebaseAuth`: managed auth (email/phone/federated), reactive `authStateChanges()`, built-in persistence, ID tokens.
- Drive guards/UI from the auth stream; map errors; verify authorization server-side (rules/backend).
- Wrap in an `AuthRepository` (domain user) to limit lock-in and stay testable.

## Revision Notes

- Methods: email/phone/`signInWithCredential` (federated)/anonymous; `authStateChanges()` `Stream<User?>`.
- Firebase persists session (no manual storage); `getIdToken()` for backend/rules.
- Authorization via rules (`request.auth`) / backend — not client `currentUser`.
- Map `FirebaseAuthException` codes; wrap in `AuthRepository` (User→domain).

## Practice Questions

1. Why not manually store Firebase tokens?
2. How do you enforce authorization with Firebase Auth?
3. How do guards react to Firebase sign-in/out?

## Coding Questions

1. Build a `FirebaseAuthRepository` (email sign-in/up, Google, signOut, `authState`).
2. Drive `go_router` guards from `authStateChanges()`.
3. Map `FirebaseAuthException` codes to friendly messages; test with a fake.

## Mini Project

**Firebase auth (Flutter):** Implement email/password + Google sign-in behind a `FirebaseAuthRepository` mapping `User` →domain user, expose reactive auth state to `go_router` guards, map errors, and access the ID token. Acceptance: reactive guards from `authStateChanges()`; errors mapped; Firebase types wrapped; server-enforced authorization noted; runs.

# Cloud Firestore

> Firestore is a serverless NoSQL **document/collection** database with real-time listeners, offline persistence, and **security rules** enforced server-side; you model for your queries (denormalize), stream data reactively, and pay per document read — so cost-aware modeling matters.

## Introduction

Firestore stores data as **documents** (JSON-like maps) in **collections**, supports **real-time** `snapshots()` streams, **offline** caching, and rich (but constrained) queries. This file covers data modeling, queries, real-time/offline, security rules, and cost — the make-or-break topics.

## Why this concept exists

Firestore provides a scalable, real-time, offline-capable database with no server to run and rules-based security. But NoSQL modeling differs from SQL: you **model around queries** (denormalize, avoid joins), and billing is **per read/write/delete**, so structure drives both correctness and cost.

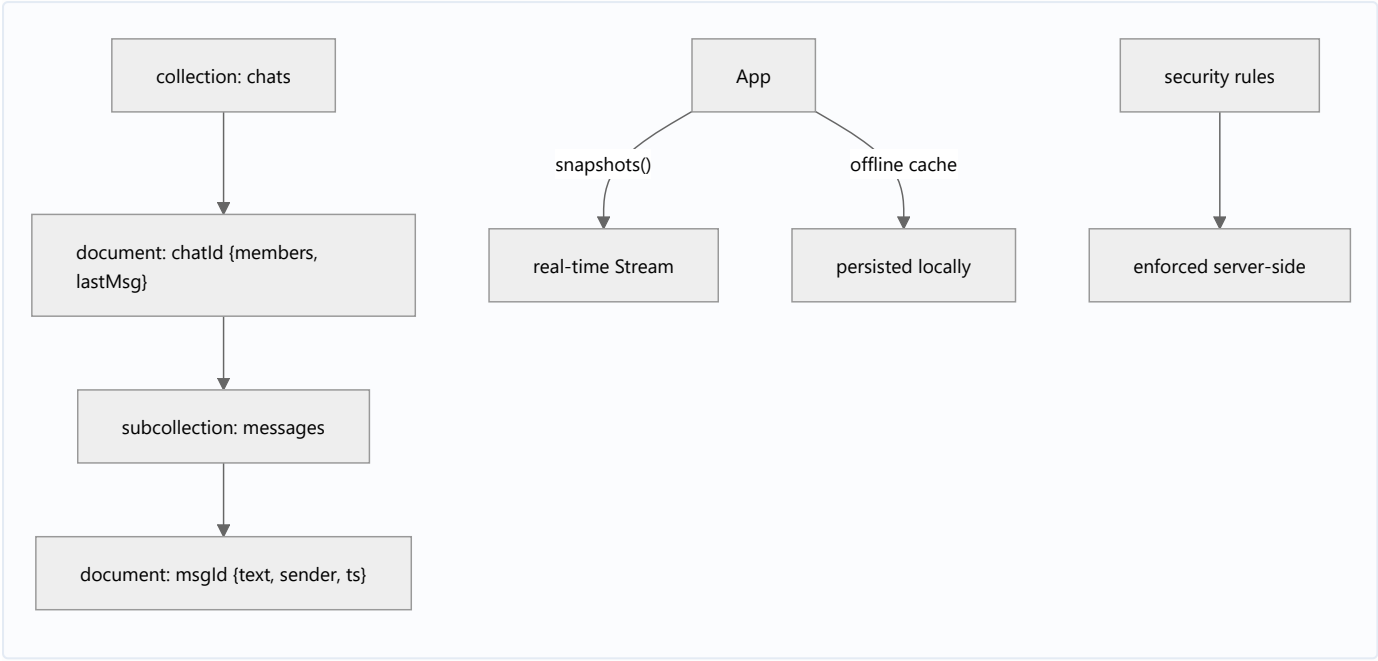
## Real-world analogy

Firestore is a **filing system of folders (collections) holding labeled files (documents)**, some files pointing to sub-folders (subcollections). There's no cross-file "join query" like a relational DB — so you **arrange files to match how you'll look them up**, and you're charged each time you pull a file (read).

## Problem Statement

Model a chat app (users, chats, messages) for efficient queries, stream a chat's messages in real time, work offline, and secure it so users only read their own chats. You'll model collections, stream `snapshots()`, and write rules.

## Internal Working




- **Model**: collections → documents (maps) → subcollections. **Denormalize** and model **around read queries** (no joins; duplicate data to avoid extra reads).
- **Reads**: `get()` (once) or `snapshots()` (`Stream` of live updates — 02 · streams); queries via `.where()/.orderBy()/.limit()` (composite queries need **indexes**; Firestore prompts to create them).
- **Writes**: `set` / `update` / `add` / `delete`; **batches** and **transactions** for atomic multi-doc writes.
- **Real-time**: listeners push changes instantly; ideal for chat/collaborative UIs.
- **Offline**: enabled by default on mobile — cached reads/writes work offline and sync on reconnect (see Module 19).
- **Security rules**: server-side rules (`match` / `allow`) using `request.auth`/resource data — the real access control (client can't be trusted). E.g., only members read a chat.
- **Cost**: billed per **document read/write/delete** (+ storage/bandwidth) — large lists/over-fetching get expensive; paginate + model tightly.

## Memory Representation

Snapshots hold document data in memory; offline cache persists locally. Large query results cost memory + read charges — paginate (`limit`, `startAfter`) (15 · caching_strategies).

## Compiler Behavior

Documents are untyped maps by default; use converters (`withConverter`) or DTO mapping for type safety (02 · json).

## Runtime Behavior

`snapshots()` emits on every change (incl. local writes optimistically, then server-confirmed); missing indexes throw with a link to create them; rules violations return permission-denied.

## Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond async/streams.

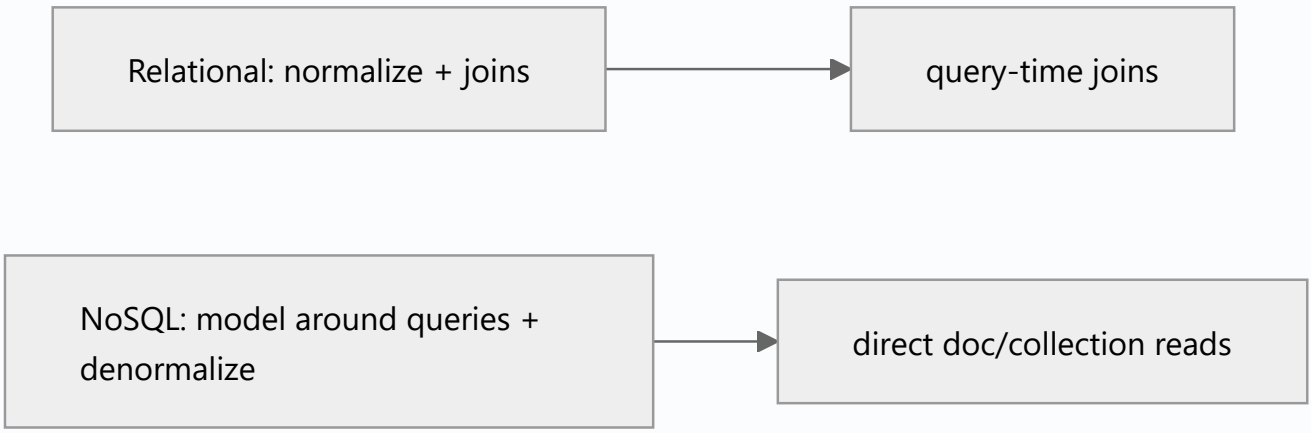
## Examples

```
import 'package:cloud_firestore/cloud_firestore.dart';


class Message {
  final String id, text, sender;
  final DateTime ts;
  Message(this.id, this.text, this.sender, this.ts);
}


class ChatRepository {
  final FirebaseFirestore _db;
  ChatRepository([FirebaseFirestore? db]) : _db = db ?? FirebaseFirestore.instance;


  // Real-time stream of a chat's messages (ordered, paginated)
  Stream<List<Message>> watchMessages(String chatId) {
    return _db
        .collection('chats').doc(chatId).collection('messages')
        .orderBy('ts', descending: true)
        .limit(50) // paginate to bound reads/cost
        .snapshots()
        .map((snap) => snap.docs
            .map((d) => Message(
                  d.id, d['text'] as String, d['sender'] as String,
                  (d['ts'] as Timestamp).toDate(),
                ))
            .toList());
  }


  Future<void> sendMessage(String chatId, String text, String sender) {
    return _db.collection('chats').doc(chatId).collection('messages').add({
```

```
      'text': text, 'sender': sender, 'ts': FieldValue.serverTimestamp(),
    });
  }
}
```

```
// firestore.rules (server-enforced authorization):
match /chats/{chatId} {
  allow read, write: if request.auth != null
      && request.auth.uid in resource.data.members;   // only members
  match /messages/{msgId} {
    allow read: if request.auth != null
        && request.auth.uid in get(/databases/$(database)/documents/chats/$(chatId)).data.members;
    allow create: if request.auth.uid == request.resource.data.sender;
  }
}
```

## Diagrams



## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Modeling like SQL (expecting joins) | Firestore has no joins | Denormalize; model around queries |
| Unbounded queries | Huge read cost + memory | `limit` + pagination ( `startAfter` ) |
| No/weak security rules | Data exposed (client untrusted) | Strict rules with `request.auth` /membership |
| Leaking `DocumentSnapshot` /Firestore types to UI | Lock-in/coupling | Map to entities in the repository |
| Ignoring index requirements | Query throws | Create composite indexes (Firestore links them) |
| Treating reads as free | Surprise bills | Model/paginate for cost; cache |

## Best Practices

- **Model around your queries**: denormalize, use subcollections, avoid needing joins; duplicate data deliberately.
- Use `snapshots()` for real-time UI; **paginate** ( `limit` / `startAfter` ) to bound reads/memory/cost.
- Write **strict security rules** ( `request.auth` , membership/ownership) — authorization lives here, not the client.
- Use **transactions/batches** for atomic multi-doc updates; `FieldValue.serverTimestamp()` for consistent times.
- **Map documents→entities** in a repository ( `withConverter` or manual); don't leak Firestore types.
- Leverage **offline persistence**; think about cost per read.

## Performance

Real-time listeners are efficient (deltas); cost/perf hinge on **query scope** — paginate and index. Denormalization trades write cost/duplication for cheap reads. Offline cache serves instantly (Module 21, Module 19).

## Advantages / Disadvantages

- **+** Serverless, real-time, offline, scalable, rules-based security, fast to build.
- **−** NoSQL modeling constraints (no joins, denormalization), per-read billing, index management, vendor lock-in, weak ad-hoc querying vs SQL.

## Interview Questions

1. 🟢 **How does Firestore structure data?** — Collections → documents (maps) → subcollections; no tables/joins.
2. 🟢 **How do you get real-time updates?** — `snapshots()` returns a `Stream` that emits on every change.
3. 🟡 **Why "model around queries"?** — Firestore has no joins and bills per read; you denormalize/duplicate so each screen's data is a direct, cheap query.
4. 🟡 **How is authorization enforced?** — Server-side **security rules** using `request.auth`/resource data; the client can't be trusted.
5. 🟡 **How do you control cost and memory?** — Paginate (`limit` / `startAfter`), avoid unbounded queries, denormalize to reduce reads, and cache.
6. 🔴 **How do you do atomic multi-document updates?** — Transactions (read-then-write with retries) or batched writes.
7. 🔴 **What are Firestore's querying limitations vs SQL?** — No joins, limited compound queries (need composite indexes), no aggregations historically (some added), and range constraints — model accordingly.

## Senior Engineer Tips

- Design the data model **from the screens/queries backward**; the biggest Firestore mistakes are SQL-style normalization and unbounded reads.
- Treat security rules as production code: test them (emulator) and enforce ownership/membership.
- Wrap Firestore in repositories mapping docs→entities; it caps lock-in and makes UI code type-safe and testable.

## Architect Perspective

Firestore trades relational flexibility for serverless scale, real-time, and offline — powerful when you **model around queries** and **secure with rules**. Cost-aware, denormalized modeling and repository boundaries are the architectural keys; it pairs with offline-first (19) and rules/App Check security (37).

## Summary

- Firestore: NoSQL documents/collections, real-time `snapshots()`, offline, server-side rules.
- Model around queries (denormalize, no joins), paginate for cost, secure with rules, map docs→entities.

- Great for real-time/offline apps when modeled and secured deliberately; mind per-read billing and lock-in.

## Revision Notes

- Collections→documents→subcollections; no joins → denormalize/model around queries.
- `get()` vs `snapshots()` (real-time Stream); `.where/orderBy/limit` (composite indexes); transactions/batches.
- Offline by default; security rules (`request.auth`) = authorization; per-read billing → paginate.
- Map docs→entities in repository (`withConverter`); don't leak Firestore types.

## Practice Questions

1. Why can't you model Firestore like a relational DB?
2. Where is authorization enforced and how?
3. How do you bound read cost/memory?

## Coding Questions

1. Model a chat (chats + messages subcollection) and stream messages with `snapshots()` + pagination.
2. Write security rules restricting reads to chat members.
3. Map documents to entities via a repository (`withConverter` or manual).

## Mini Project

**Real-time chat data layer (Flutter):** Model users/chats/messages, build a `ChatRepository` streaming paginated messages via `snapshots()` (mapped to entities), sending messages with `serverTimestamp`, plus security rules limiting access to members. Acceptance: query-shaped model; real-time + paginated; rules enforce membership; Firestore types wrapped; runs (emulator ok).

# Realtime Database & Cloud Storage

> **Realtime Database** is Firebase's original JSON-tree database — great for simple, high-frequency real-time sync (presence, live counters); **Cloud Storage** stores large files (images, videos, uploads) with resumable transfers and security rules.

## Introduction

Two more Firebase services: **Realtime Database (RTDB)** — a single JSON tree with low-latency sync — and **Cloud Storage** — object storage for blobs. This file covers when to use RTDB vs Firestore, and how to upload/download files with Storage, plus their security rules.

## Why this concept exists

Not everything fits Firestore. RTDB excels at simple, very-frequent real-time updates (presence, live scores) with lower latency and a different pricing model. Cloud Storage exists because databases shouldn't hold large binaries — files go to object storage with URLs referenced from the DB.

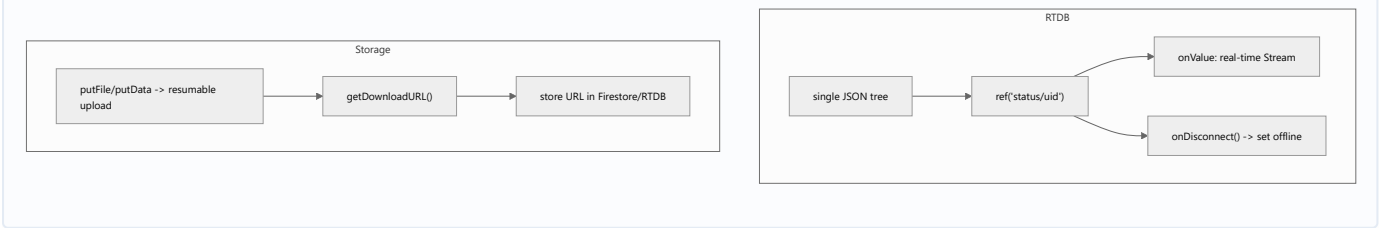
## Real-world analogy

RTDB is a **shared live whiteboard** (one big tree everyone sees update instantly); Firestore is an **organized filing cabinet** (structured docs). Cloud Storage is the **warehouse** for big crates (files) — the filing cabinet just holds the shipping label (download URL).

## Problem Statement

Track user **presence** (online/offline, high-frequency) and let users **upload a profile photo**. You'll use RTDB for presence and Cloud Storage for the image, storing its URL in the DB.

## Internal Working




- **Realtime Database** (`firebase_database`): one JSON tree; `ref(path)` → `set/update/push`; `onValue` / `onChildAdded` streams; **`onDisconnect()`** for presence (auto-write on disconnect). Lower

latency, simpler queries than Firestore, priced by bandwidth/storage (not per-read). Best for **presence, live counters, simple high-frequency sync**.

- **RTDB vs Firestore**: RTDB = one tree, limited querying, cheaper for high-frequency small updates; Firestore = structured collections, richer queries/scaling, per-read billing. Prefer **Firestore** for most apps; RTDB for presence/simple real-time.
- **Cloud Storage** ( `firebase_storage` ): `ref(path).putFile/putData()` returns an `UploadTask` (progress, pause/resume); `getDownloadURL()` yields a shareable URL to store in the DB; `getData` / `writeToFile` to download. Handles large files, resumable transfers.
- **Security rules**: both have server-side rules (Storage rules gate by `request.auth` /path; RTDB rules by path/auth) — the real access control.

## Memory Representation

RTDB snapshots hold tree data in memory; Storage streams bytes (don't load huge files fully — stream). Store only **URLs** (not blobs) in the DB (15 · file_storage).

## Compiler Behavior

Not applicable.

## Runtime Behavior

RTDB `onValue` emits on any change under the path; `onDisconnect` executes server-side when the client drops. Storage uploads/downloads report progress via task streams; failures need retry/error handling.

## Flutter Engine Behavior / Dart VM Behavior

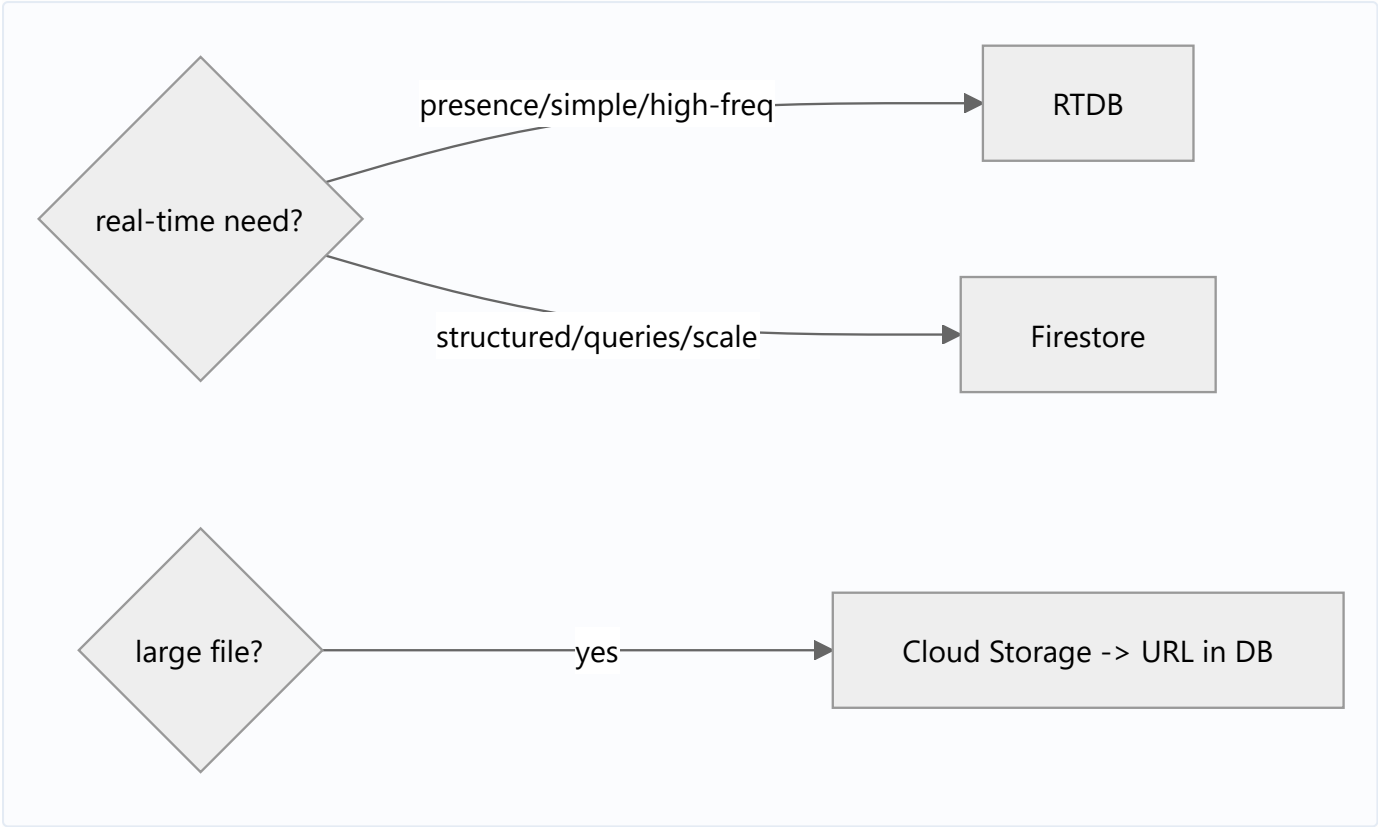
Not applicable beyond async/streams.

## Examples

```
import 'dart:io';
import 'package:firebase_database/firebase_database.dart';
import 'package:firebase_storage/firebase_storage.dart';

// --- Realtime Database: presence ---
class PresenceService {
  final _db = FirebaseDatabase.instance;
  void goOnline(String uid) {
    final ref = _db.ref('status/$uid');
    ref.set({'state': 'online', 'lastSeen': ServerValue.timestamp});
    ref.onDisconnect().set({'state': 'offline', 'lastSeen': ServerValue.timestamp});
  }
  Stream<bool> watchOnline(String uid) =>
      _db.ref('status/$uid/state').onValue.map((e) => e.snapshot.value == 'online');
}

// --- Cloud Storage: upload profile photo, store URL ---
class StorageService {
  final _storage = FirebaseStorage.instance;
  Future<String> uploadProfilePhoto(String uid, File file) async {
    final ref = _storage.ref('users/$uid/profile.jpg');
    final task = ref.putFile(file);                 // resumable upload
    task.snapshotEvents.listen((s) {
      // progress: s.bytesTransferred / s.totalBytes
    });
    await task;
    return ref.getDownloadURL();                     // store this URL in Firestore/RTDB
  }
}
```

```
// storage.rules: only the owner can write their folder
match /users/{uid}/{file=**} {
  allow read: if request.auth != null;
  allow write: if request.auth != null && request.auth.uid == uid;
}
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| Storing files/blobs in a database | Bloat, cost, limits | Cloud Storage; store the URL |
| Using RTDB for complex queries | Limited querying | Use Firestore for structured/queried data |
| No presence `onDisconnect()` | Stale "online" forever | Use `onDisconnect()` to set offline |
| Weak Storage/RTDB rules | Public data/files | Path + auth rules (owner-only writes) |
| Loading huge downloads into memory | OOM/jank | Stream to file; download at need |
| Leaking service types to UI | Coupling | Wrap in repositories |

## Best Practices

- Use **Firestore by default**; reach for **RTDB** for presence/live counters/simple high-frequency sync (use `onDisconnect()`).
- Store **files in Cloud Storage**, keep only **download URLs** in the DB; stream large transfers with progress.
- Write **strict rules** for both (owner/path/auth); rules are the security boundary.
- Wrap both behind **repositories**; handle upload/download errors + retries.
- Consider CDN/caching for served files; downscale images before upload (07 · images).

## Performance

RTDB is low-latency for small frequent updates; Storage handles large files with resumable/parallel transfers. Store URLs (not blobs) in the DB; stream big downloads; cache served images (Module 21).

## Advantages / Disadvantages

- **+ RTDB:** low latency, simple real-time, presence, cheaper high-frequency. **+ Storage:** large files, resumable, secure, URL-based.
- **− RTDB:** one tree, weak queries, easy to structure poorly. **− Storage:** must manage URLs/lifecycle/costs; both add lock-in.

## Interview Questions

1. 🟢 **RTDB vs Firestore?** — RTDB is a single JSON tree, low-latency, limited queries, priced by bandwidth — best for presence/simple real-time; Firestore is structured, richer queries/scaling, per-read billing — the default for most data.
2. 🟢 **What is Cloud Storage for?** — Storing large files/blobs (images, videos, uploads) with resumable transfers and rules; the DB holds the download URL.
3. 🟡 **How do you implement presence?** — RTDB `onDisconnect()` sets the user offline server-side when the client drops; `onValue` streams status.
4. 🟡 **Why store URLs, not files, in the database?** — Databases aren't for large binaries (cost/limits); Storage holds the file, DB references its URL.
5. 🟡 **How do you track upload progress?** — Listen to the `UploadTask.snapshotEvents` (`bytesTransferred/totalBytes`).
6. 🔴 **Where is access control for RTDB/Storage?** — Server-side security rules (path + `request.auth`), e.g., owner-only writes — not the client.
7. 🔴 **When would you pick RTDB over Firestore despite Firestore being newer?** — Very-high-frequency, low-latency, simple real-time (presence/live counters) where RTDB's model/pricing fit better.

## Senior Engineer Tips

- Default to Firestore; add RTDB only for presence/live-counter use cases where its latency/pricing win.
- Always downscale/validate images before upload and stream large downloads; store CDN-friendly URLs.
- Test Storage/RTDB rules (emulator); owner/path scoping prevents cross-user data/file access.

## Architect Perspective

RTDB and Storage complement Firestore: RTDB for ephemeral high-frequency real-time (presence), Storage for blobs referenced by URL. Behind repositories with strict rules, they compose into a cost-effective, secure backend; choosing among RTDB/Firestore/Storage per data shape is a key architectural call (03_firestore.md, Module 37).

## Summary

- RTDB: JSON-tree, low-latency real-time (presence via `onDisconnect`); Firestore is the structured default.
- Cloud Storage: large files with resumable transfers; store URLs in the DB, not blobs.
- Secure both with server-side rules; wrap in repositories; stream large transfers.

## Revision Notes

- RTDB (JSON tree, `onValue` / `onDisconnect`, presence/high-freq, bandwidth-priced) vs Firestore (structured, per-read).
- Storage: `putFile` / `putData` (resumable, progress) → `getDownloadURL()` → store URL in DB.
- Rules (path + `request.auth`) = security; owner-only writes.
- Store URLs not blobs; stream big files; wrap in repositories.

## Practice Questions

1. When choose RTDB over Firestore?
2. Why store a file's URL in the DB, not the file itself?
3. How does presence use `onDisconnect()`?

## Coding Questions

1. Implement presence with RTDB (`onDisconnect` + `onValue`).
2. Upload a photo to Storage with progress and store its URL.
3. Write Storage rules allowing only the owner to write their folder.

## Mini Project

**Presence + profile photo (Flutter):** Build a `PresenceService` (RTDB `onDisconnect`) streaming online status and a `StorageService` uploading a (downscaled) profile photo with progress, storing the URL in Firestore. Add owner-only Storage rules. Acceptance: presence updates in real time; upload with progress; URL stored (not blob); rules scoped; runs (emulator ok).

# Cloud Functions

> Cloud Functions run your backend code serverlessly in response to **triggers** (Firestore/Auth/Storage events, HTTPS) or **callable** functions invoked from the app — the place for trusted server-side logic (secrets, validation, fan-out, third-party calls) that must not live on the client.

## Introduction

Cloud Functions (deployed via the Firebase CLI, written in JS/TS/Python) execute server-side. Flutter apps call **callable functions** ( `cloud_functions` package) or trigger **event functions** (on document create, user signup, file upload). This file covers when/why to use them, callable vs triggers, and security.

## Why this concept exists

Some logic must be trusted and server-side: using secret API keys, validating/sanitizing writes, sending notifications, charging payments, aggregating data, or calling third-party services. Putting these on the client is insecure/unreliable. Functions provide serverless backend compute integrated with Firebase.

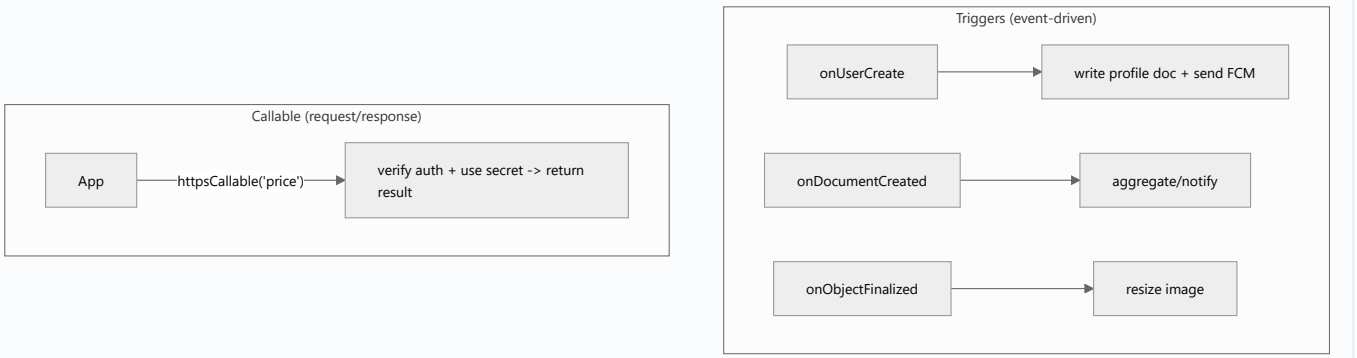
## Real-world analogy

Cloud Functions are **automated back-office staff**: some react to events ("when a new order file arrives, process it" — triggers), others handle direct requests ("compute my quote" — callable). They work behind a locked door (server) with access to the safe (secrets) the public (client) can't reach.

## Problem Statement

When a user signs up, create their profile doc and send a welcome notification (trigger); and let the app request a server-computed price using a secret pricing key (callable). You'll use an Auth/Firestore trigger and a callable function.

## Internal Working




- **Trigger functions** (background, event-driven): run on Firebase events — Auth ( `onUserCreate` ), Firestore ( `onDocumentCreated/Updated` ), Storage ( `onObjectFinalized` ), Pub/Sub/schedule. Used for fan-out, aggregation, notifications, post-processing.
- **Callable functions** ( `onCall` ): invoked from the app via `FirebaseFunctions.instance.httpsCallable('name').call(data)` ; Firebase passes the caller's **auth context** automatically (verify it). Return typed JSON.
- **HTTPS functions** ( `onRequest` ): plain HTTP endpoints (webhooks, REST) — no auto auth context.
- **Security**: **verify auth/permissions inside the function** (callable gives `context.auth` ); never trust client input; keep **secrets** in function config/Secret Manager (never in the app).
- **Deployment**: `firebase deploy --only functions` ; functions run in the cloud (Node/Python), not in Dart.
- **Cold starts/cost**: serverless functions have cold-start latency and per-invocation cost; keep them lean.

## Memory Representation

Functions run server-side (not in the app); the Flutter side only sends/receives JSON. Callable results map to domain models in a repository (05 · repository).

## Compiler Behavior

Not applicable to Flutter (functions are JS/TS/Python); the Dart side is a typed call/result.

## Runtime Behavior

Callables are async network calls (handle errors/cold starts/timeouts). Triggers run asynchronously after events; they can retry (make idempotent). HTTPS/webhooks need their own auth.

## Flutter Engine Behavior / Dart VM Behavior

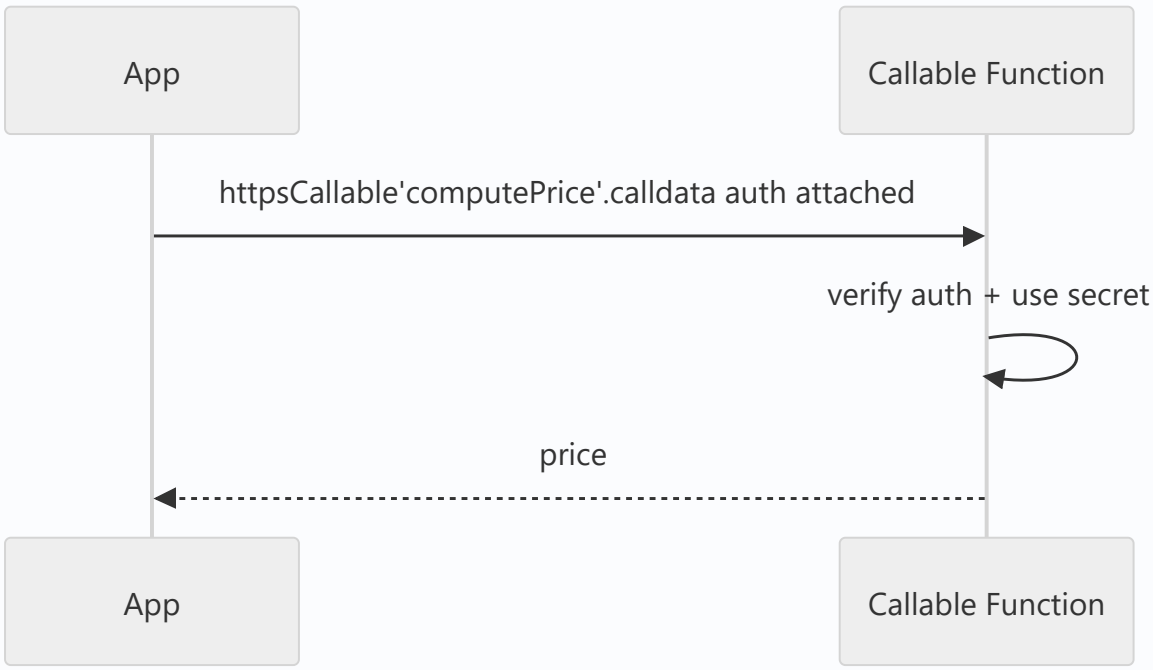
Not applicable.

## Examples

```
// Flutter side: call a callable function
import 'package:cloud_functions/cloud_functions.dart';

class PricingRepository {
  final FirebaseFunctions _functions;
  PricingRepository([FirebaseFunctions? f]) : _functions = f ?? FirebaseFunctions.instance;

  Future<double> quote(String sku, int qty) async {
    final callable = _functions.httpsCallable('computePrice');
    final result = await callable.call({'sku': sku, 'qty': qty}); // auth passed automatically
    return (result.data['price'] as num).toDouble(); // map result -> domain
  }
}
```

```
// Server side (functions/index.js) — trusted logic, secrets stay here:
const { onCall } = require('firebase-functions/v2/https');
const { onUserCreated } = require('firebase-functions/v2/auth'); // illustrative

// Callable: verify auth + use a secret pricing key
exports.computePrice = onCall((request) => {
  if (!request.auth) throw new Error('unauthenticated'); // verify caller
  const { sku, qty } = request.data;
  const price = lookupPrice(sku) * qty; // uses server-only secret/config
  return { price };
});

// Trigger: create profile + welcome on signup
exports.onSignup = onUserCreated(async (event) => {
  const uid = event.data.uid;
  await createProfileDoc(uid);   // fan-out
  await sendWelcomeNotification(uid); // FCM
});
```

## Diagrams




## Common Mistakes

| Mistake | Why | Fix |
| --- | --- | --- |
| Putting secret keys/logic on the client | Exposed/insecure | Move to Cloud Functions (secrets in config) |
| Not verifying auth in callables | Anyone can invoke | Check `request.auth` /permissions |
| Non-idempotent triggers | Retries duplicate effects | Make triggers idempotent |
| Ignoring cold starts/timeouts | Slow/failed calls | Keep functions lean; handle timeouts client-side |
| Trusting client input | Injection/abuse | Validate/sanitize server-side |
| Leaking function result types to UI | Coupling | Map to entities in a repository |

## Best Practices

- Use functions for **trusted server-side logic**: secrets, validation, fan-out, notifications, payments, third-party calls.
- **Callables** for app-invoked request/response (auto auth context — verify it); **triggers** for event reactions (make **idempotent**).
- Keep **secrets server-side** (function config/Secret Manager); never in the app.
- Handle **cold starts/timeouts/errors** on the client; keep functions lean.
- Wrap callable results behind **repositories** (map to entities).

## Performance

Serverless scales automatically but has **cold-start latency** and per-invocation cost; keep functions small and warm critical paths if needed. Offload heavy/trusted work here to keep the client light (Module 21).

## Advantages / Disadvantages

- **+** Trusted server compute without managing servers, event-driven integration, secret-safe, scalable, integrates with FCM/Firestore.
- **−** Cold starts + cost, another language/runtime (JS/TS/Python), deployment/ops, vendor lock-in, idempotency care for triggers.

## Interview Questions

1. 🟢 **What are Cloud Functions for?** — Running trusted server-side logic serverlessly in response to events (triggers) or app calls (callable/HTTPS).
2. 🟢 **Callable vs trigger functions?** — Callable = app-invoked request/response with auto auth context; trigger = runs automatically on Firebase events (Auth/Firestore/Storage).
3. 🟡 **Why not put secret keys/logic on the client?** — Clients are inspectable; secrets/trusted logic belong server-side in functions (config/Secret Manager).
4. 🟡 **How do callables handle auth?** — Firebase passes the caller's auth context ( `request.auth` ); you must verify it inside the function.
5. 🟡 **Why must triggers be idempotent?** — They can retry on failure; non-idempotent side effects (double-charging, duplicate notifications) would repeat.
6. 🔴 **What's the cold-start tradeoff?** — First invocations after idle incur latency; keep functions lean, handle timeouts, and warm critical paths if needed.
7. 🔴 **How do functions integrate with the rest of Firebase?** — Triggers react to Auth/Firestore/Storage events and can write Firestore/send FCM; callables serve the app — forming server-side glue.

## Senior Engineer Tips

- Treat callables like any API: verify auth, validate input, return typed results, and map them behind a repository.
- Make every trigger idempotent (check-then-act, dedupe keys) — retries are guaranteed eventually.
- Keep secrets and third-party calls exclusively in functions; the client should never hold API secrets.

## Architect Perspective

Cloud Functions provide the trusted server tier of a Firebase app: secure logic, secrets, validation, fan-out, and integrations — without running servers. Combined with rules (data access) they enforce security end-to-end; the tradeoffs (cold starts, lock-in, ops) are the cost of serverless. They're where payments, notifications, and cross-service orchestration belong (Modules 31, 32, 37).

## Summary

- Cloud Functions run trusted server-side logic via triggers (events) or callables (app requests).
- Keep secrets/validation/fan-out server-side; verify auth in callables; make triggers idempotent.
- Handle cold starts/errors; wrap results behind repositories; the client never holds secrets.

## Revision Notes

- Triggers (Auth/Firestore/Storage events, idempotent) vs callable ( `httpsCallable` , auto auth context) vs HTTPS (webhooks).
- Secrets/trusted logic server-side (config/Secret Manager); verify `request.auth` ; validate input.
- Cold starts + cost → lean functions; map results to entities.
- Server glue: react to events, write Firestore, send FCM.

## Practice Questions

1. When use a callable vs a trigger?
2. Why keep secret keys in functions, not the app?
3. Why must triggers be idempotent?

## Coding Questions

1. Call a callable `computePrice` from a `PricingRepository` , mapping the result.
2. Sketch an `onUserCreate` trigger that creates a profile + sends FCM.
3. Add auth verification + input validation to a callable (server pseudocode).

## Mini Project

**Server-side pricing + signup trigger (Flutter + functions):** Implement a callable `computePrice` (verifies auth, uses a server-only key) consumed by a `PricingRepository` , plus an `onUserCreate` trigger creating a profile doc and sending a welcome FCM. Acceptance: secret stays server-side;

callable auth verified; trigger idempotent; result mapped to domain; client handles errors/cold starts.

# Observability & Messaging (Crashlytics, Analytics, FCM, Remote Config, App Check)

> Firebase rounds out with operational services: **Crashlytics** (crash reporting), **Analytics** (events/funnels), **Cloud Messaging/FCM** (push notifications), **Remote Config** (server-tuned flags), and **App Check** (attest requests come from your genuine app) — the observability, growth, and abuse-prevention layer.

## Introduction

Beyond data/auth, Firebase offers services for *running* an app: knowing when it crashes (Crashlytics), how it's used (Analytics), pushing messages (FCM), toggling behavior remotely (Remote Config), and blocking abuse (App Check). This file surveys each and how they fit.

## Why this concept exists

Shipping isn't enough — you must observe crashes, understand usage, re-engage users, tune behavior without redeploying, and stop bots/abuse. These managed services provide that operational layer that would otherwise require significant custom infrastructure.

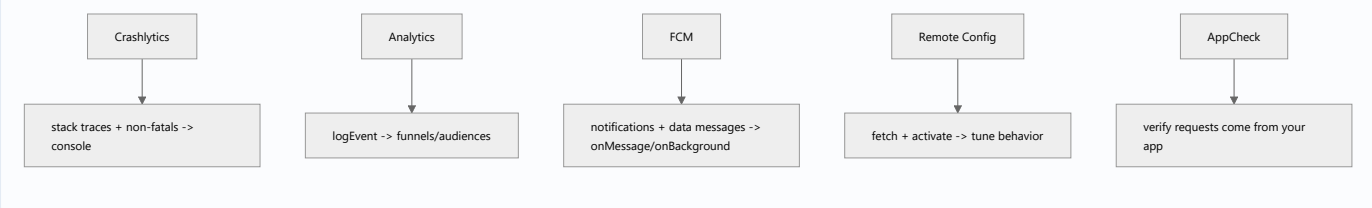
## Real-world analogy

These are a store's **operations department**: security cameras catching accidents (Crashlytics), foot-traffic analytics (Analytics), the PA/loyalty texts (FCM), adjustable signage/promos (Remote Config), and the ID-check at the door verifying you're a real customer, not a bot (App Check).

## Problem Statement

You need crash reports with stack traces, key usage events, push notifications (with deep links), a feature flag you can flip server-side, and protection so only your genuine app hits your backend. You'll wire each service.

## Internal Working



- **Crashlytics** ( `firebase_crashlytics` ): capture fatal crashes + **non-fatal** errors with stack traces; wire `FlutterError.onError` and `PlatformDispatcher.instance.onError` / `runZonedGuarded` to record uncaught errors (Module 38, Module 52). Add custom keys/logs for context.
- **Analytics** ( `firebase_analytics` ): `logEvent(name, parameters)` for funnels, user properties, audiences; screen tracking. Respect privacy/consent.
- **Cloud Messaging (FCM)** ( `firebase_messaging` ): push **notification** and **data** messages; foreground ( `onMessage` ), background ( `onBackgroundMessage` ), and tap handling (deep-link routing — 13 · deep_linking, Module 32); device tokens + topics; requires permission (iOS) and platform setup.
- **Remote Config** ( `firebase_remote_config` ): fetch server-defined key-values (feature flags, A/B params); `fetchAndActivate()` with sensible defaults + fetch intervals — tune behavior without an app update.
- **App Check** ( `firebase_app_check` ): attests requests originate from your untampered app (via Play Integrity / DeviceCheck / reCAPTCHA), protecting backends/Firestore/Functions from abuse — a key part of "keys aren't secrets, rules + App Check are the defense" (01_firebase_setup_and_core.md, Module 37).

## Memory Representation

These are lightweight client SDKs sending events/reports/tokens; Remote Config caches fetched values locally. FCM handlers run on the isolate (background handler is a top-level function).

## Compiler Behavior

FCM's background handler must be a top-level/static function (runs in a background isolate — 02 · isolates).

## Runtime Behavior

Crashlytics batches reports (visible after next launch/upload); Analytics events are batched; FCM delivers foreground/background/terminated differently; Remote Config serves cached values until fetched+activated; App Check attaches attestation tokens to requests.

## Flutter Engine Behavior

All cross the embedder to native SDKs; FCM background handling and notification display involve platform notification systems (Module 32).

## Dart VM Behavior

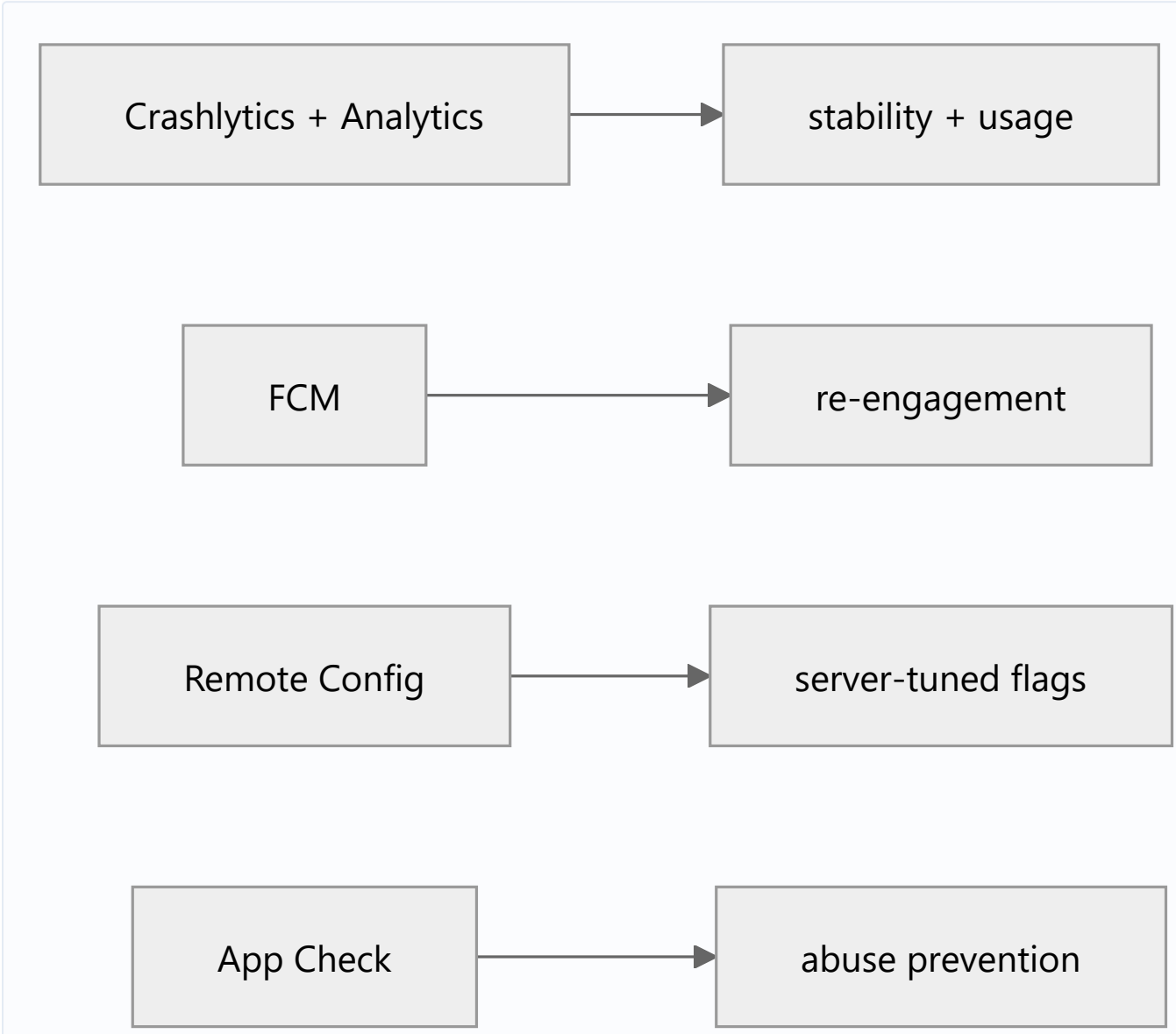
Background FCM handler runs in a separate isolate; plugin access there may need setup.

## Examples

```
import 'package:firebase_crashlytics/firebase_crashlytics.dart';
import 'package:firebase_analytics/firebase_analytics.dart';
import 'package:firebase_remote_config/firebase_remote_config.dart';
import 'package:firebase_messaging/firebase_messaging.dart';
import 'package:flutter/foundation.dart';

// Crashlytics: capture uncaught errors globally (in main, after init)
void wireCrashlytics() {
  FlutterError.onError = FirebaseCrashlytics.instance.recordFlutterFatalError;
  PlatformDispatcher.instance.onError = (error, stack) {
    FirebaseCrashlytics.instance.recordError(error, stack, fatal: true);
    return true;
  };
}

// Analytics
Future<void> logPurchase(double amount) =>
    FirebaseAnalytics.instance.logEvent(name: 'purchase', parameters: {'amount': amount});

// Remote Config feature flag
Future<bool> newCheckoutEnabled() async {
  final rc = FirebaseRemoteConfig.instance;
  await rc.setDefaults({'new_checkout': false});
  await rc.fetchAndActivate();
  return rc.getBool('new_checkout');
}

// FCM: background handler must be top-level
@pragma('vm:entry-point')
Future<void> _bgHandler(RemoteMessage message) async {
  // handle background message (no UI)
}
Future<void> wireMessaging() async {
  await FirebaseMessaging.instance.requestPermission();      // iOS permission
  FirebaseMessaging.onBackgroundMessage(_bgHandler);
  FirebaseMessaging.onMessage.listen((m) {/* foreground */});
  FirebaseMessaging.onMessageOpenedApp.listen((m) {
    // route to a screen (deep link) based on m.data
```

```
  });
}
```

## Diagrams


Crashlytics + Analytics
stability + usage
FCM
re-engagement
Remote Config
server-tuned flags
App Check
abuse prevention

## Common Mistakes

| Mistake | Why | Fix |
|---|---|---|
| No global error capture | Crashes invisible | Wire `FlutterError.onError` / `onError` /zone to Crashlytics |
| FCM background handler not top-level | Won't run | Use a top-level `@pragma('vm:entry-point')` function |
| Analytics without consent | Privacy/legal violation | Gate on consent; anonymize |
| Remote Config without defaults/intervals | Wrong behavior/quota issues | Set defaults + sensible fetch intervals |
| Relying on rules alone (no App Check) | Backend abuse from non-app clients | Add App Check |
| Logging PII in analytics/crash keys | Privacy leak | Redact; no secrets/PII (37) |

## Best Practices

- Wire **Crashlytics** to global error handlers (Module 38) + custom keys for context; record non-fatals too.
- Log **meaningful Analytics events** (funnels) with **consent** and no PII.
- Handle **FCM** in all states (foreground/background/terminated); route taps via deep links; register a **top-level background handler**.
- Use **Remote Config** for flags/A-B with **defaults** and reasonable fetch intervals; treat as tuning, not critical logic.
- Enable **App Check** to protect backends/Firestore/Functions (with rules) — client keys aren't the security layer.

## Performance

Lightweight SDKs; batch/async sending. FCM background handling and Remote Config fetch add minor cost; defer non-critical init post-first-frame (01_firebase_setup_and_core.md, Module 21).

## Advantages / Disadvantages

- **+** Managed observability/growth/security layer, integrated, fast to adopt, cross-platform.
- **−** Privacy/consent obligations, platform setup (FCM/App Check), vendor lock-in, background-handler/isolate quirks.

## Interview Questions

1. 🟢 **What does each service do?** — Crashlytics (crash/error reporting), Analytics (usage events), FCM (push), Remote Config (server-tuned flags), App Check (request attestation/abuse prevention).
2. 🟢 **How do you capture uncaught errors in Crashlytics?** — Wire `FlutterError.onError` and `PlatformDispatcher.onError` / `runZonedGuarded` to `recordError` / `recordFlutterFatalError`.
3. 🟡 **How is FCM handled in different app states?** — `onMessage` (foreground), a top-level `onBackgroundMessage` handler (background/terminated), and `onMessageOpenedApp` (tap → deep link).
4. 🟡 **What is Remote Config for?** — Fetching server-defined values (feature flags, A/B params) to tune behavior without an app update; use defaults + fetch intervals.
5. 🟡 **What problem does App Check solve?** — It attests requests come from your genuine, untampered app, protecting backends/Firestore/Functions from abuse — complementing rules.
6. 🔴 **Why aren't Firebase client keys the security boundary?** — They're public identifiers; security comes from **security rules + App Check** (server-enforced), not key secrecy.
7. 🔴 **What privacy considerations apply to Analytics/Crashlytics?** — Obtain consent, avoid PII/secrets in events/keys, and comply with platform/legal requirements.

## Senior Engineer Tips

- Treat Crashlytics + Analytics as part of your monitoring strategy (Module 52); add custom keys/breadcrumbs so crashes are diagnosable.
- Design FCM payloads to carry a route for deep-link routing on tap; test all app states on device.
- Layer **rules + App Check** for real security; never treat client keys as secret (Module 37).

## Architect Perspective

These services form the operational and abuse-prevention layer: observability (Crashlytics/Analytics — Module 52), engagement (FCM — Module 32), controllability (Remote Config), and integrity (App Check — Module 37). Wired with consent, deep-link routing, and rules+App Check, they make a Firebase app observable, tunable, and defensible in production.

## Summary

- Crashlytics (crashes/non-fatals via global handlers), Analytics (consented events), FCM (multi-state push + deep links), Remote Config (flags with defaults), App Check (attestation).
- Client keys aren't secrets — security is **rules + App Check**; respect privacy/consent; defer non-critical init.

- These form the observability/growth/security layer of a Firebase app.

## Revision Notes

- Crashlytics: wire `FlutterError.onError` / `PlatformDispatcher.onError`; record non-fatals + custom keys.
- Analytics: `logEvent` (consent, no PII); FCM: `onMessage`/top-level `onBackgroundMessage` / `onMessageOpenedApp` (deep link), permission + platform setup.
- Remote Config: defaults + `fetchAndActivate` + intervals (tuning, not critical logic).
- App Check: attest genuine app; security = rules + App Check (keys public).

## Practice Questions

1. How do you capture uncaught Flutter errors in Crashlytics?
2. How is FCM handled foreground vs background vs tap?
3. Why do rules + App Check (not client keys) provide security?

## Coding Questions

1. Wire global error handlers to Crashlytics and record a non-fatal with custom keys.
2. Handle FCM in all states and route a tapped notification via deep link.
3. Read a feature flag from Remote Config with defaults.

## Mini Project — Module capstone

**Ops-ready Firebase slice (Flutter):** Add Crashlytics (global handlers + non-fatals + custom keys), key Analytics events (with a consent gate), FCM (all states + deep-link routing on tap), a Remote Config feature flag (with defaults), and enable App Check — all initialized after core, non-critical ones deferred post-first-frame. Acceptance: crashes captured; consented analytics; FCM routes on tap; flag toggles behavior; App Check enabled; no PII/secrets logged. (Combines with auth/Firestore/functions for the full Module 18 capstone.)