19 · Offline First

Flutter Practice Notes

Contents

19 · Offline First

Introduction

Offline-first apps treat the **local store as the source of truth** and the network as a background sync — so the UI is instant and works with no connection, syncing when possible. This module covers the architecture, sync engine, conflict resolution, connectivity/queued mutations (outbox), and optimistic UI.

Why this module exists

Mobile networks are unreliable. Apps that block on the network feel slow and break offline. Offline-first delivers instant, resilient UX — but introduces hard problems (sync, conflicts, queued writes) that must be designed deliberately, not bolted on.

Module map

#	File	Topic	Level
1	01_offline_first_fundamentals.md	Principles + local-source-of-truth architecture	🔴
2	02_sync_engine.md	Pull/push, delta sync, sync state	🔴
3	03_conflict_resolution.md	LWW, versioning, merge, CRDTs	🔴
4	04_connectivity_and_outbox.md	Connectivity handling, queued mutations (outbox)	🔴
5	05_optimistic_ui.md	Optimistic updates + rollback	🔵

> **Cross-references:** Caching/TTL: 15 · caching_strategies. Database (local store): Module 20. Networking: Module 16. Repository boundary: 05 · repository. Background sync: Module 33. Firestore offline: 18 · firestore. State/streams: Modules 11, 02 · streams.

Prerequisites

15 Local Storage, 16 Networking, 05 · repository, 02 · streams.

What you'll be able to do after this module

- Design a local-source-of-truth, offline-first architecture.
- Build a sync engine (pull/push, delta, sync status) behind a repository.
- Choose and implement a conflict-resolution strategy.
- Queue mutations offline (outbox) and flush on reconnect.

- Implement optimistic UI with rollback.

Capstone

Offline-first notes app: Local DB is source of truth; UI reads/writes locally (instant, optimistic); a sync engine pushes an outbox + pulls deltas on connectivity, with last-write-wins + version conflict handling — all behind a repository exposing a reactive stream.

Summary

Offline-first = local store is truth, network is background sync. Get the four hard parts right — sync, conflicts, queued writes, optimistic UI — behind a repository, and the app feels instant and never "breaks offline."

Offline-First Fundamentals (Local Source of Truth)

> Offline-first inverts the usual flow: the **local store is the single source of truth** the UI reads/writes instantly; the network syncs it in the background — so the app is fast and fully functional with or without connectivity.

Introduction

The core principle: don't read/write the network directly from the UI. Instead, the UI talks to a **local store** (DB/cache), and a **sync engine** reconciles that store with the server asynchronously. This file establishes the architecture the rest of the module builds on.

Why this concept exists

Network-first apps freeze on spinners, fail offline, and feel laggy. Users expect instant, always-working apps (like native note/email apps). Offline-first delivers that by decoupling UI responsiveness from network availability — the local store answers immediately; sync happens when it can.

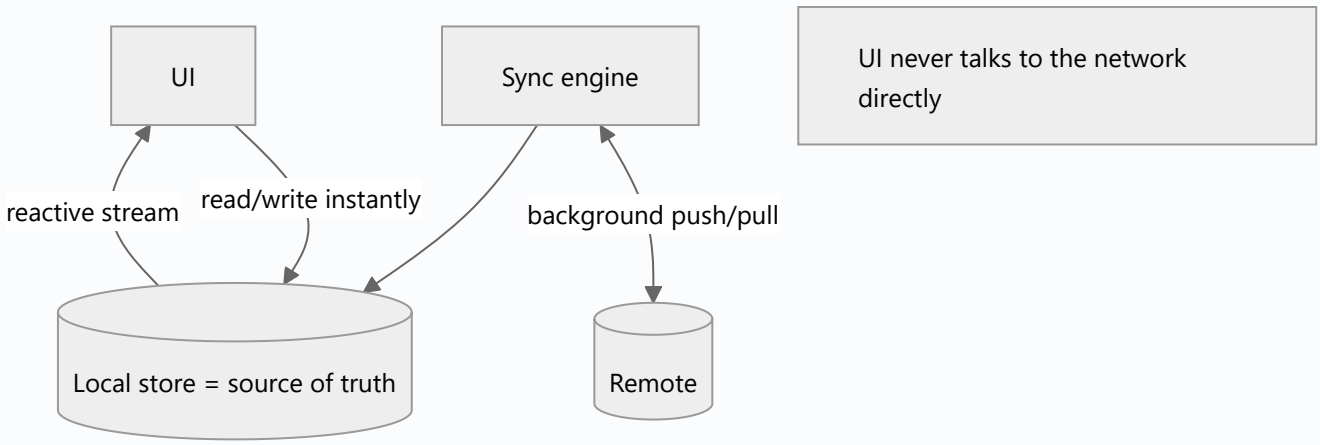
Real-world analogy

Offline-first is a **personal notebook** you always write in (local store); a secretary periodically **photocopies changes to/from the office archive** (sync). You never wait for the archive to write a note — you write locally now, and it's reconciled later. If the office is unreachable, you keep working.

Problem Statement

A notes app must open instantly, let users create/edit notes with no connection, and sync seamlessly when online — never blocking on the network. You'll design the local-source-of-truth architecture.

Internal Working



- **Local store = source of truth**: a local DB (Module 20) holds the canonical app data the UI reads (usually via a **reactive stream**) and writes to immediately.
- **UI ↔ local only**: the UI never awaits the network; it reads/writes the local store and re-renders from it.
- **Sync engine** (02_sync_engine.md): a background process pushes local changes to the server and pulls remote changes into the local store — reconciling the two.
- **Repository boundary** (05 · repository): exposes `watch()` / `save()` over the local store + coordinates sync; the UI/domain is oblivious to network state.
- **Metadata**: records carry sync metadata (dirty flag, version/updatedAt, deleted tombstone) to drive sync/conflict logic.
- **Distinguish from caching**: caching serves *reads* faster with TTL (15 · caching_strategies); offline-first makes the local store **authoritative for reads and writes**, with full bidirectional sync.

Memory Representation

The local DB holds canonical data on disk; the UI observes streams (bounded queries). Sync metadata (dirty/version) lives per record (Module 20).

Compiler Behavior

Not applicable.

Runtime Behavior

Reads/writes are local (instant); the UI updates from a local stream. Sync runs opportunistically (on connectivity/interval/app events) without blocking the UI. Offline writes are marked dirty and queued for push (04_connectivity_and_outbox.md).

Flutter Engine Behavior / Dart VM Behavior

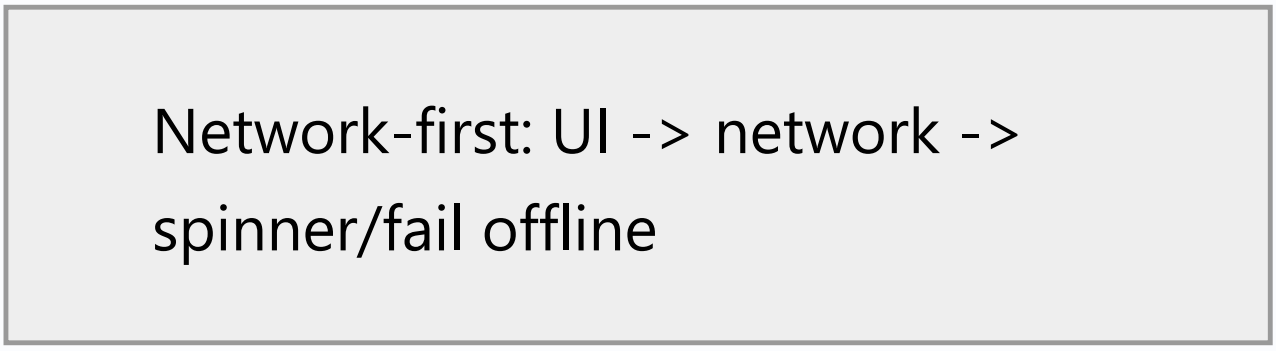
Not applicable; heavy sync/merge work should run off the UI isolate if large (02 · isolates).

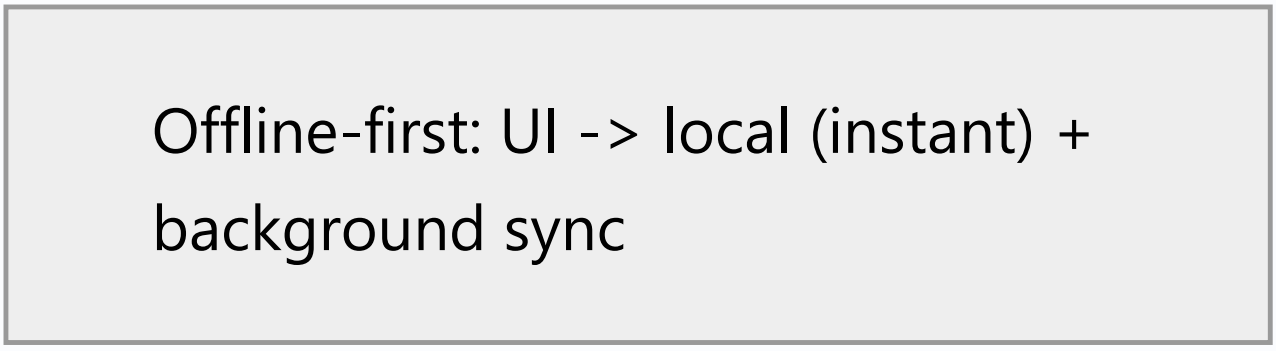
Examples

```
// Offline-first repository: UI reads/writes LOCAL; sync happens in background
abstract interface class LocalStore {
  Stream<List<Note>> watchNotes();          // reactive local reads
  Future<void> upsert(Note note);           // local write (mark dirty)
  Future<List<Note>> dirtyNotes();          // pending to push
}
abstract interface class RemoteApi {
  Future<List<Note>> pullSince(DateTime? cursor);
  Future<void> push(List<Note> notes);
}

class Note {
  final String id, text;
  final DateTime updatedAt;
  final bool dirty;      // local change not yet synced
  final int version;     // for conflict detection
  const Note(this.id, this.text, this.updatedAt, {this.dirty = false, this.version = 0});
}

class NotesRepository {
  final LocalStore local;
  final RemoteApi remote;
  NotesRepository(this.local, this.remote);

  // UI observes LOCAL store (instant, offline-capable):
  Stream<List<Note>> watch() => local.watchNotes();

  // Writes go LOCAL first (instant), marked dirty for later push:
```

```
  Future<void> saveNote(Note note) =>
      local.upsert(Note(note.id, note.text, DateTime.now(), dirty: true, version: note.version));

  // Sync engine (called on connectivity/interval) — see sync_engine.md
  Future<void> sync() async {
    await remote.push(await local.dirtyNotes());   // push local changes
    for (final n in await remote.pullSince(null)) { // pull remote changes
      await local.upsert(n);                          // reconcile into local
    }
  }
}
```

Diagrams





Common Mistakes

Mistake	Why	Fix
UI awaiting the network for reads/writes	Laggy, breaks offline	UI reads/writes local; sync in background
No sync metadata (dirty/version)	Can't track/resolve changes	Add dirty flag + version/updatedAt + tombstones
Treating cache-with-TTL as offline-first	No write sync/authority	Make local authoritative + bidirectional sync
Deleting records hard (no tombstone)	Deletions don't propagate	Soft-delete (tombstone) then sync
Sync logic in the UI	Coupling, untestable	Encapsulate in repository/sync engine

Best Practices

- Make the **local store the source of truth**; the UI reads (streams) and writes it **immediately**.
- Run **sync in the background** (on connectivity/interval/app resume), never blocking UI.
- Track **sync metadata** per record: dirty flag, version/ `updatedAt`, tombstones for deletes.
- Encapsulate everything behind a **repository** exposing reactive reads + local writes (05 · repository).
- Offload heavy merges to an isolate; design for eventual consistency.
- Reuse a proven local DB with sync support where possible (Firestore offline, PowerSync, Isar/Drift + custom sync).

Performance

Local reads/writes are instant (no network latency); sync amortizes network work in the background. Offload large syncs off-isolate; bound queries/streams (Module 21).

Advantages / Disadvantages

- **+** Instant UX, works offline, resilient to flaky networks, decoupled from connectivity, fewer spinners.
- **−** Complexity (sync/conflicts/queueing), eventual consistency (not instant global truth), metadata/tombstone bookkeeping, harder testing.

Interview Questions

1. 🟢 **What defines offline-first?** — The local store is the source of truth the UI reads/writes instantly; the network syncs in the background.
2. 🟢 **How does it differ from caching?** — Caching speeds up reads with TTL; offline-first makes the local store authoritative for reads **and** writes with bidirectional sync.

3. 🟡 **Why does the UI never talk to the network directly?** — To stay instant and functional offline; it talks to the local store, and a sync engine reconciles with the server.
4. 🟡 **What sync metadata do records need?** — A dirty flag (unsynced local change), a version/ `updatedAt` (conflict detection), and tombstones (soft-delete propagation).
5. 🟡 **What consistency model does offline-first imply?** — Eventual consistency — local is immediately correct; global convergence happens after sync.
6. 🔴 **What are the hard problems introduced?** — Sync (pull/push/delta), conflict resolution, queued offline mutations (outbox), and optimistic UI/rollback.
7. 🔴 **Where does sync logic belong?** — In a sync engine behind the repository, off the UI; the UI/domain stays connectivity-agnostic.

Senior Engineer Tips

- Decide offline-first **up front** — retrofitting it into a network-first app is painful (touches data model, repos, UI).
- Model records with sync metadata from day one (dirty/version/tombstone); it's the backbone of sync/conflict logic.
- Lean on existing solutions (Firestore offline, PowerSync, WatermelonDB-style, or Isar/Drift + a documented sync protocol) before hand-rolling.

Architect Perspective

Offline-first is a top-level architectural decision that reshapes the data layer: local-source-of-truth + background sync behind repositories, with eventual consistency. It delivers premium UX and resilience but demands rigorous handling of sync, conflicts, and queued writes — the subject of the rest of this module (02_sync_engine.md, 03_conflict_resolution.md, 04_connectivity_and_outbox.md).

Summary

- Offline-first: local store is source of truth (instant reads/writes), network syncs in background.
- Track sync metadata (dirty/version/tombstone); encapsulate in a repository; embrace eventual consistency.
- Distinct from caching; introduces sync/conflict/outbox/optimistic-UI complexity to design deliberately.

Revision Notes

- Local store = source of truth; UI reads (stream)/writes local instantly; sync in background.
- Metadata: dirty flag + version/ `updatedAt` + tombstones.

- Not caching (authoritative + bidirectional sync); eventual consistency.
- Repository-encapsulated; offload heavy sync off-isolate; decide up front.

Practice Questions

1. Why doesn't the UI await the network in offline-first?
2. What metadata must records carry and why?
3. How is offline-first different from cache-with-TTL?

Coding Questions

1. Design `LocalStore` / `RemoteApi` / `NotesRepository` where UI reads/writes local + a `sync()` method.
2. Add dirty/version/tombstone metadata to the model.
3. Sketch how a write flows: UI → local (dirty) → later push.

Mini Project

Offline-first skeleton (Flutter): Build a `NotesRepository` where `watch()` streams from a fake local store and `saveNote()` writes locally (dirty), plus a `sync()` that pushes dirty notes and pulls remote ones into local. Prove the UI works with the "network" unavailable. Acceptance: UI reads/writes local only; sync in background; metadata tracked; runs. (Sync details continue in 02_sync_engine.md.)

The Sync Engine (Pull/Push, Delta Sync, Sync State)

> A sync engine reconciles the local store with the server: **push** local (dirty) changes up, **pull** remote changes down — ideally as **deltas** (only what changed since a cursor) — while tracking sync state and running opportunistically on connectivity/interval/app events.

Introduction

The sync engine is the background reconciler of offline-first. This file covers the sync loop (push then pull), **delta sync** (change cursors/timestamps), sync triggers, ordering, and exposing sync **status** — the plumbing that keeps local and remote convergent.

Why this concept exists

Two independently-mutated stores (local + remote) must converge. Naive "fetch everything every time" is slow/expensive; and offline writes must eventually reach the server. A sync engine formalizes *what* to send/fetch, *when*, and *in what order* — efficiently and reliably.

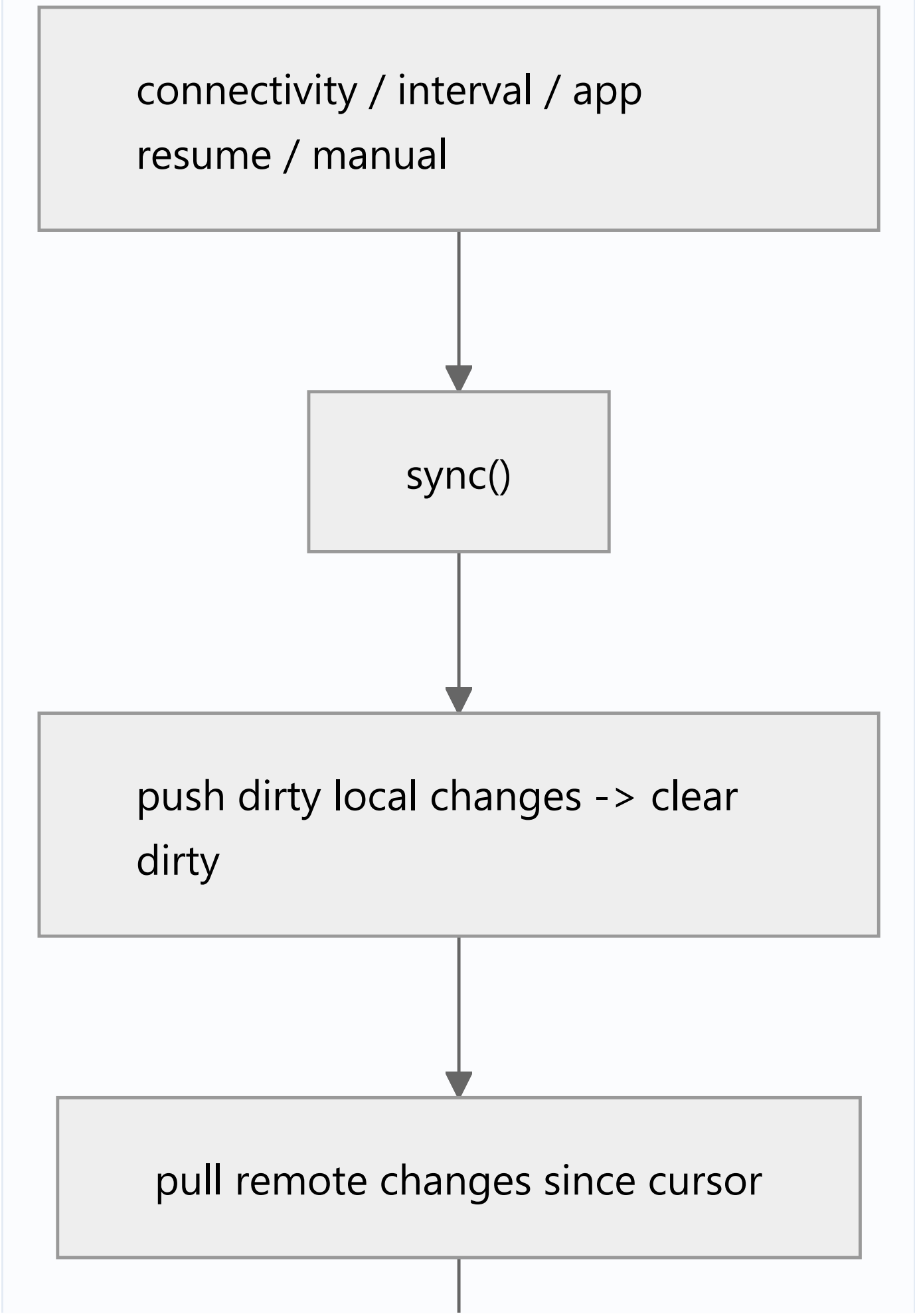
Real-world analogy

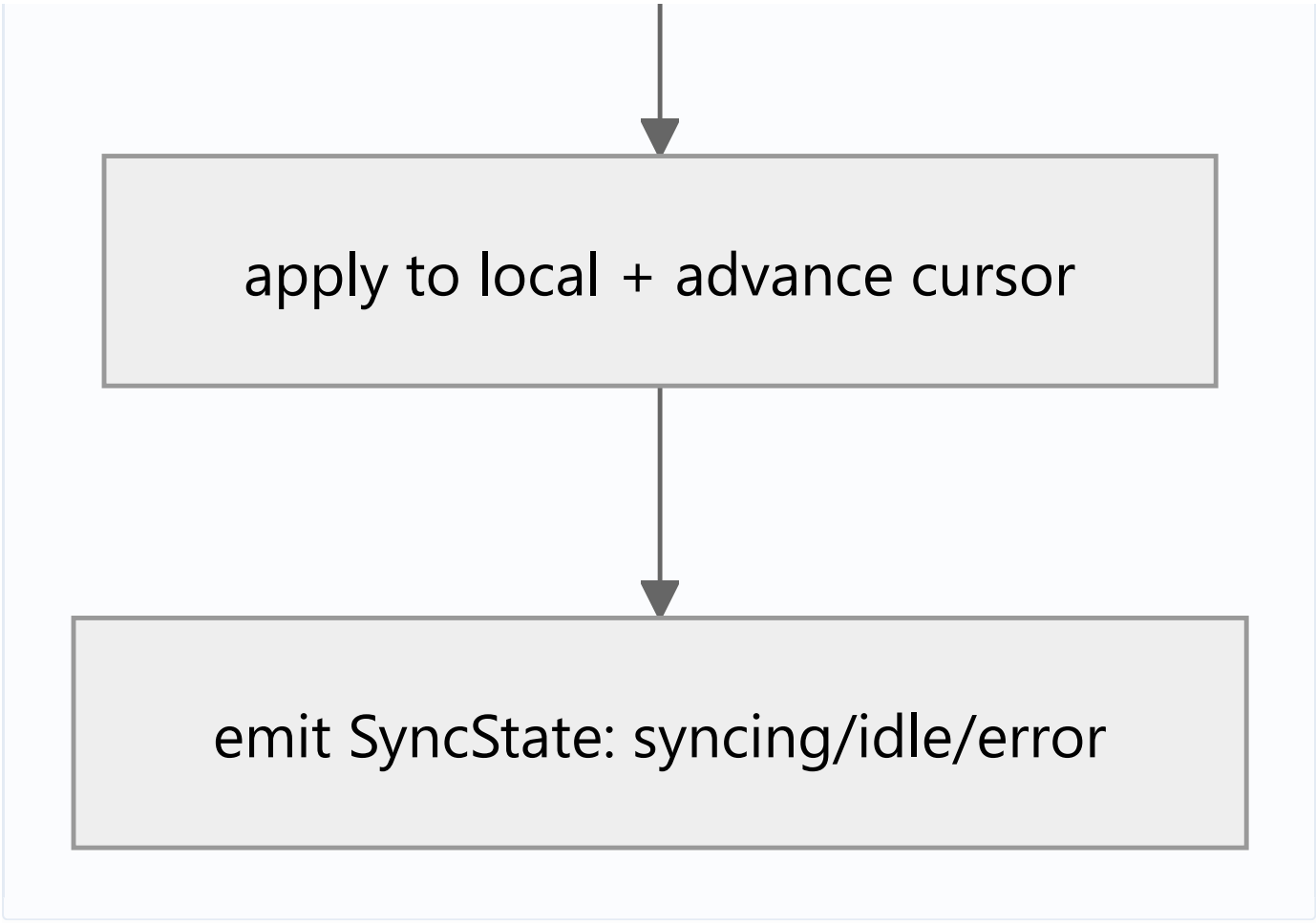
A **shipping-and-receiving dock**: outbound (push) sends your packages (local changes) to the warehouse; inbound (pull) receives new stock (remote changes) since your last delivery slip (cursor). You don't recount the whole warehouse each time — you process only what changed (delta).

Problem Statement

Sync a notes app efficiently: push dirty notes, pull only notes changed since the last sync, handle deletes, run on reconnect/interval, and show a "syncing/synced/error" status. You'll build the sync loop with a delta cursor.

Internal Working


connectivity / interval / app resume / manual
sync()
push dirty local changes -> clear dirty
pull remote changes since cursor



- **Push (upload)**: send local **dirty** records (creates/updates/deletes-as-tombstones) to the server; on success, clear their dirty flag and record the server's version. This drains the **outbox** (04_connectivity_and_outbox.md).
- **Pull (download)**: fetch remote changes **since a cursor** (a timestamp or opaque token) — **delta sync** — apply them to the local store, then **advance the cursor**. Full sync only on first run/reset.
- **Order**: typically **push before pull** (so the server has your latest, and pulled data reflects it), then merge with conflict handling (03_conflict_resolution.md).
- **Triggers**: on connectivity regained, periodic interval, app resume, after a local write (debounced), or manual pull-to-refresh; background sync via WorkManager (Module 33).
- **Sync state**: expose a `SyncState` (`idle` / `syncing` / `error` + last-synced time) as a stream for UI indicators.
- **Idempotency & retries**: syncs can fail/retry — make push idempotent (server upsert by id) and retry with backoff; avoid overlapping syncs (a lock/flag).
- **Deletes**: use **tombstones** (soft-delete flag) so deletions propagate; purge tombstones after all peers have synced.

Memory Representation

The cursor + sync metadata persist locally (DB); sync batches load changed records only (delta) → bounded memory. Large syncs run off-isolate (02 · isolates).

Compiler Behavior

Not applicable.

Runtime Behavior

Sync runs async in the background; overlapping runs are prevented; failures retry with backoff; the cursor only advances on a fully-applied pull (so a crash mid-sync re-pulls safely).

Flutter Engine Behavior / Dart VM Behavior

Not applicable beyond async; heavy merges → isolate.

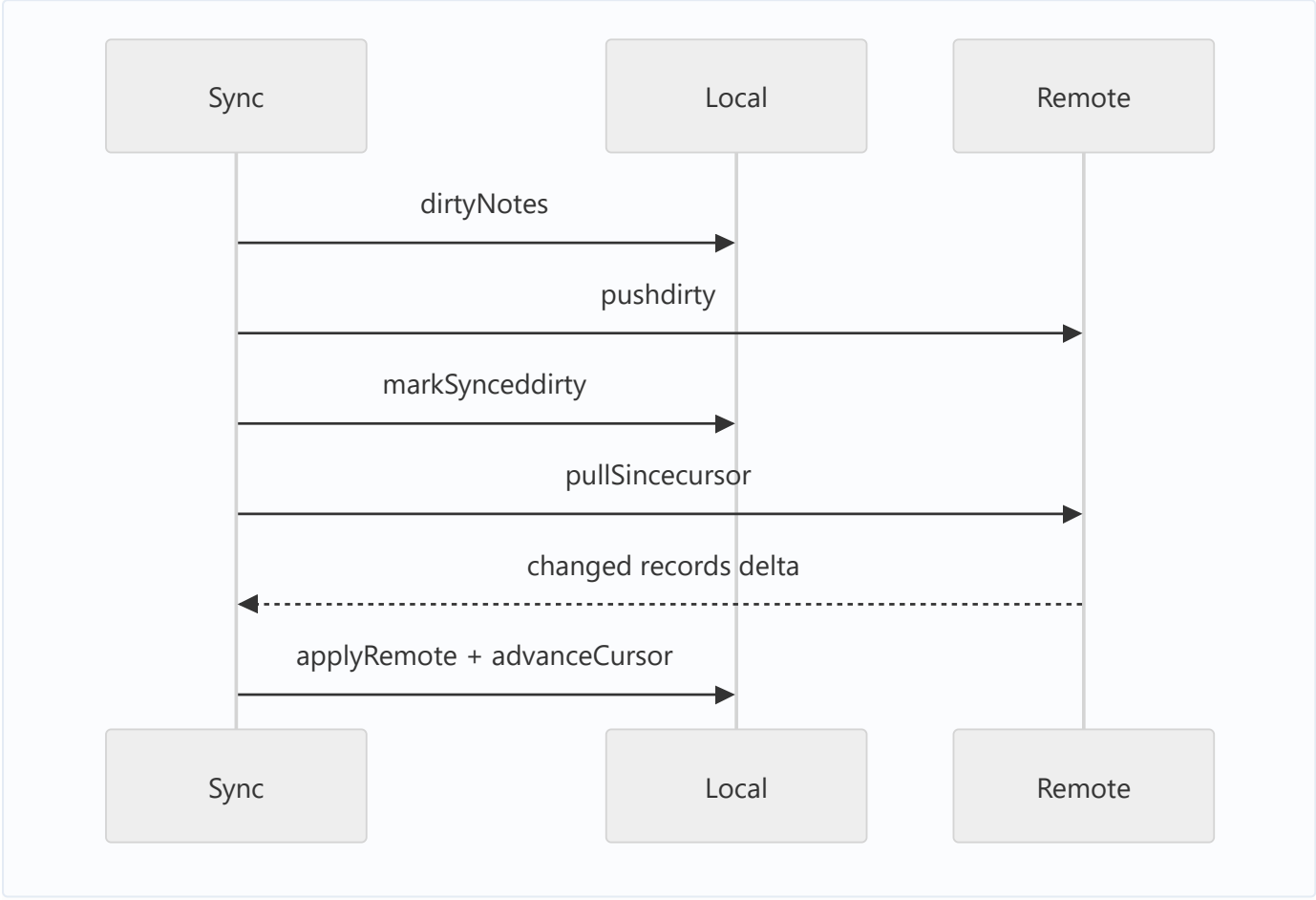
Examples

```
enum SyncStatus { idle, syncing, error }
class SyncState {
  final SyncStatus status;
  final DateTime? lastSynced;
  const SyncState(this.status, [this.lastSynced]);
}

class SyncEngine {
  final LocalStore local;   // watchNotes/upsert/dirtyNotes/cursor
  final RemoteApi remote;   // push / pullSince
  bool _running = false;
  final _state = // broadcast StreamController<SyncState> in real code
      _StateBus();

  SyncEngine(this.local, this.remote);
  Stream<SyncState> get state => _state.stream;

  Future<void> sync() async {
    if (_running) return;              // avoid overlapping syncs
    _running = true;
    _state.emit(const SyncState(SyncStatus.syncing));
```

```
    try {
      // 1) PUSH dirty local changes (idempotent upsert by id)
      final dirty = await local.dirtyNotes();
      if (dirty.isNotEmpty) {
        await remote.push(dirty);
        await local.markSynced(dirty);  // clear dirty + store server version
      }
      // 2) PULL remote deltas since cursor
      final cursor = await local.syncCursor();
      final changes = await remote.pullSince(cursor);
      for (final note in changes) {
        await local.applyRemote(note);  // reconcile (conflict handling here)
      }
      await local.advanceCursor(_maxUpdatedAt(changes) ?? cursor);
      _state.emit(SyncState(SyncStatus.idle, DateTime.now()));
    } catch (e) {
      _state.emit(const SyncState(SyncStatus.error)); // retry later w/ backoff
    } finally {
      _running = false;
    }
  }

  DateTime? _maxUpdatedAt(List<Note> notes) =>
      notes.isEmpty ? null : notes.map((n) => n.updatedAt).reduce((a, b) => a.isAfter(b) ? a : b);
}

// stand-ins
abstract interface class LocalStore {
  Future<List<Note>> dirtyNotes();
  Future<void> markSynced(List<Note> notes);
  Future<DateTime?> syncCursor();
  Future<void> advanceCursor(DateTime? cursor);
  Future<void> applyRemote(Note note);
}
abstract interface class RemoteApi {
  Future<void> push(List<Note> notes);
  Future<List<Note>> pullSince(DateTime? cursor);
}
```

```
class Note { final String id; final DateTime updatedAt; const Note(this.id, this.updatedAt); }
class _StateBus { Stream<SyncState> get stream async* {} void emit(SyncState s) {} }
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Full sync every time	Slow, costly	Delta sync with a cursor
Pull before push	Pulled data misses local changes	Push then pull (usually)
Advancing cursor before applying	Lost changes on crash	Advance only after successful apply
Overlapping syncs	Races/duplication	Guard with a running flag/lock
Hard deletes	Deletions don't propagate	Tombstones + later purge
Non-idempotent push	Duplicates on retry	Upsert by id; idempotent server
No sync status	Opaque UX	Expose `SyncState` stream

Best Practices

- Do **delta sync** with a persisted **cursor** (timestamp/token); full sync only on first run/reset.
- **Push before pull**; make push **idempotent** (upsert by id) and retry with **backoff**.
- **Advance the cursor only after** changes are fully applied (crash-safe).
- **Prevent overlapping syncs**; trigger on connectivity/interval/resume/manual (debounce after writes).
- Use **tombstones** for deletes; purge when safe.
- Expose **sync state** for UI; run heavy merges off-isolate; do background sync via WorkManager (Module 33).

Performance

Delta sync minimizes bandwidth/battery vs full sync; batching + backoff avoid storms; off-isolate merges keep the UI smooth (Module 21).

Advantages / Disadvantages

- **+** Efficient convergence, resilient (idempotent/retry), background/opportunistic, observable status.
- **−** Cursor/tombstone bookkeeping, ordering/idempotency subtleties, conflict handling required, testing complexity.

Interview Questions

1. 🟢 **What does a sync engine do?** — Reconciles local and remote by pushing local (dirty) changes and pulling remote changes into the local store.
2. 🟢 **What is delta sync?** — Fetching only records changed since a stored cursor (timestamp/token), instead of everything.
3. 🟡 **Why push before pull?** — So the server has your latest changes before you pull, and the pulled state (and conflict resolution) accounts for them.
4. 🟡 **When do you advance the sync cursor?** — Only after remote changes are fully applied locally, so a crash mid-sync safely re-pulls.
5. 🟡 **How do you propagate deletes?** — Tombstones (soft-delete flags) synced like updates, purged after convergence.
6. 🔴 **How do you make sync safe under retries/crashes?** — Idempotent push (upsert by id), overlap guard, cursor-after-apply, and backoff retries.
7. 🔴 **What triggers a sync?** — Connectivity regained, intervals, app resume, post-write (debounced), manual refresh, and background jobs (WorkManager).

Senior Engineer Tips

- Persist the cursor and advance it **transactionally after apply**; this single rule prevents most "lost update on crash" bugs.
- Make push idempotent server-side (upsert by client id / dedupe key) so retries are safe.
- Expose sync status + last-synced time; users trust apps that show sync state.

Architect Perspective

The sync engine is the heart of offline-first: a background, delta-based, idempotent reconciler behind the repository, feeding conflict resolution and draining the outbox. Its correctness (ordering, cursor discipline, idempotency) determines data integrity; it composes with connectivity handling, background execution, and conflict strategy into a resilient data layer (04_connectivity_and_outbox.md, 03_conflict_resolution.md, Module 33).

Summary

- Sync = push dirty local changes, pull remote deltas since a cursor, apply, advance cursor.
- Push before pull; idempotent push + backoff; advance cursor after apply; tombstones for deletes.
- Guard overlaps, trigger opportunistically, expose sync state, offload heavy merges.

Revision Notes

- Push (dirty, idempotent upsert) → pull (delta since cursor) → apply → advance cursor (after apply).
- Triggers: connectivity/interval/resume/post-write/manual/background.
- Tombstones for deletes; overlap guard; retry+backoff; `SyncState` stream.
- Cursor discipline = crash safety; heavy merges off-isolate.

Practice Questions

1. Why is delta sync better than full sync?
2. Why advance the cursor only after applying changes?
3. How do you make push safe under retries?

Coding Questions

1. Implement `SyncEngine.sync()` (push dirty → pull delta → apply → advance cursor) with an overlap guard.

2. Add idempotent push (upsert by id) + retry with backoff.
3. Expose a `SyncState` stream and drive a UI status indicator.

Mini Project

Delta sync engine (Flutter): Build a `SyncEngine` over fake local/remote stores implementing push-then-pull delta sync with a persisted cursor (advanced after apply), tombstone deletes, overlap guard, retry/backoff, and a `SyncState` stream. Trigger it on a fake connectivity event. Acceptance: delta (not full) sync; idempotent push; crash-safe cursor; deletes propagate; status observable; runs. (Conflicts handled in 03_conflict_resolution.md.)

Conflict Resolution (LWW, Versioning, Merge, CRDTs)

> When the same record is edited both locally and remotely before syncing, you have a **conflict** — resolve it with a deliberate strategy: last-write-wins (simple, lossy), version/field merge (smarter), or CRDTs (automatic convergence for collaborative data).

Introduction

Concurrent edits are inevitable in offline-first (two devices, or device + server). This file covers detecting conflicts (versions/vector clocks) and the strategies — **last-write-wins (LWW)**, **field-level merge**, **manual resolution**, and **CRDTs** — with their tradeoffs.

Why this concept exists

Sync must decide *whose change wins* when both sides changed. Ignoring conflicts silently loses data. A chosen, documented strategy makes convergence predictable and matches the data's needs (a note vs a collaborative doc vs a counter differ).

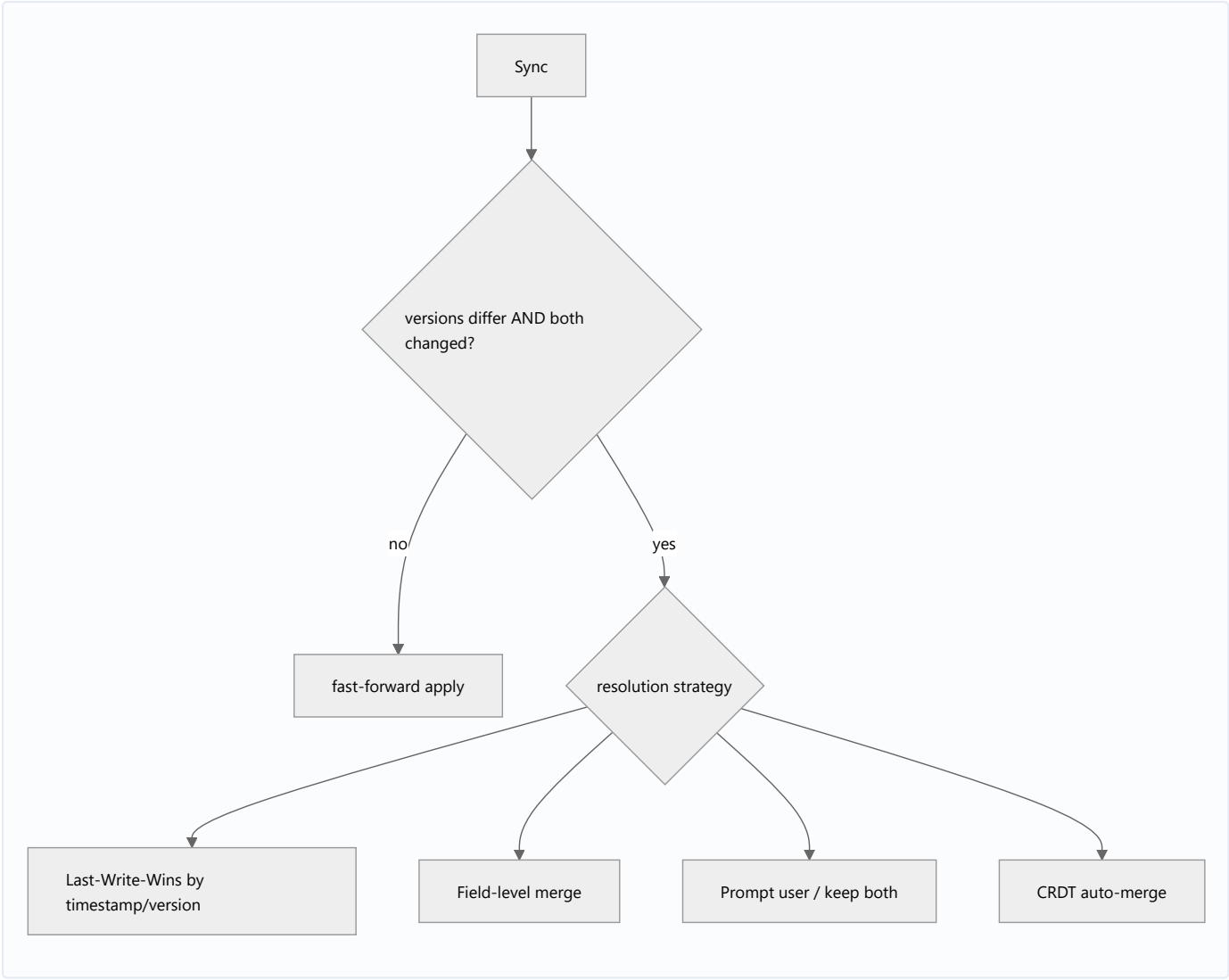
Real-world analogy

Two editors mark up the **same document offline**, then both submit. Conflict resolution is the **editor-in-chief's policy**: "latest submission wins" (LWW), "merge non-overlapping edits" (field merge), "ask a human to reconcile overlaps" (manual), or a **shared live doc that auto-merges keystrokes** (CRDT).

Problem Statement

Device A and the server both edit note #7 while A was offline. On sync, decide the winner without silently losing intent. You'll detect the conflict (version) and apply a strategy per the data type.

Internal Working



- **Detection**: compare **versions** (monotonic counter) or **vector clocks**; a conflict = the remote's base version differs from what the local change was based on (both diverged). Timestamps alone are unreliable (clock skew) — prefer versions/logical clocks.
- **Strategies**:
  - **Last-Write-Wins (LWW)**: newest timestamp/version wins; simple, but **loses** the other side's change. Fine for low-contention/whole-record data.
  - **Field-level merge**: merge non-conflicting fields; only truly-overlapping fields need a tiebreak. Preserves more intent.
  - **Manual / keep-both**: surface the conflict to the user (or store both versions) — for high-value data where silent loss is unacceptable.
  - **CRDTs (Conflict-free Replicated Data Types)**: data structures (counters, sets, sequences/text) that **merge automatically and commutatively** to a consistent state regardless of order — ideal for collaborative editing/counters; more complex.
- **Server authority**: often the server is the tiebreaker/validator; some designs resolve on-device, others server-side.

- **Choice by data**: whole-record note → LWW/field-merge; counter → CRDT; collaborative text → CRDT/OT; financial → manual/server-authoritative.

Memory Representation

Records carry version/clock metadata; CRDTs carry extra state (e.g., per-replica counters). Manual/keep-both may store multiple versions temporarily (02_sync_engine.md).

Compiler Behavior

Not applicable.

Runtime Behavior

On pull/apply, the engine detects conflicts and applies the strategy; LWW/merge produce a single record; CRDTs merge deterministically; manual defers to the user.

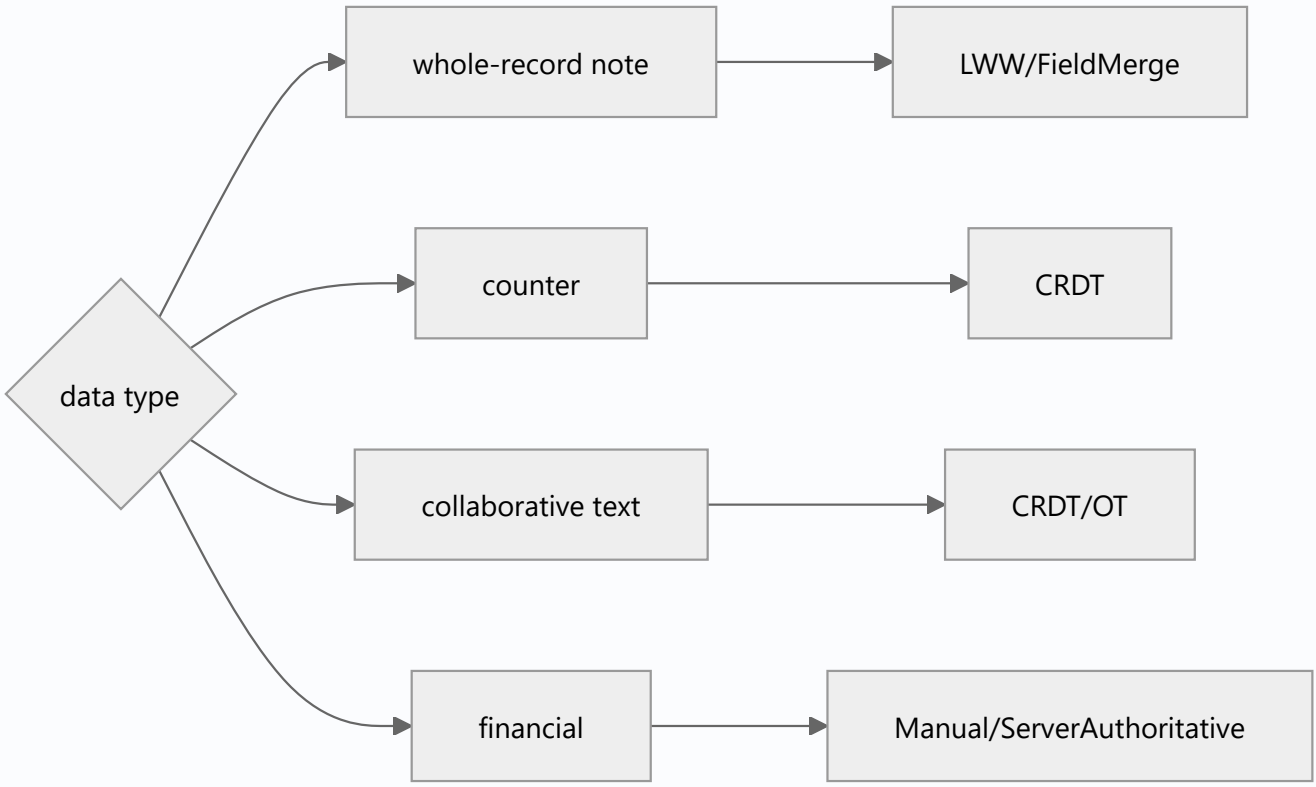
Flutter Engine Behavior / Dart VM Behavior

Not applicable; heavy merges off-isolate.

Examples

```
class Note {
  final String id, title, body;
  final int version;        // monotonic; incremented on each change
  final DateTime updatedAt;
  const Note(this.id, this.title, this.body, this.version, this.updatedAt);
}

// Last-Write-Wins (by version, fallback updatedAt)
Note resolveLWW(Note local, Note remote) =>
    remote.version > local.version ? remote
    : local.version > remote.version ? local
    : (remote.updatedAt.isAfter(local.updatedAt) ? remote : local);

// Field-level merge: keep each side's change to different fields; tiebreak overlaps
Note resolveFieldMerge(Note base, Note local, Note remote) {
  String pick(String b, String l, String r) {
    final localChanged = l != b, remoteChanged = r != b;
    if (localChanged && !remoteChanged) return l;   // only local changed
    if (remoteChanged && !localChanged) return r;   // only remote changed
    if (!localChanged && !remoteChanged) return b;
    // both changed the same field -> tiebreak (LWW)
    return remote.version >= local.version ? r : l;
  }
  return Note(
    base.id,
    pick(base.title, local.title, remote.title),
    pick(base.body, local.body, remote.body),
    (local.version > remote.version ? local.version : remote.version) + 1,
    DateTime.now(),
  );
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Timestamp-only LWW	Clock skew → wrong winner/data loss	Use versions/logical clocks (+timestamp tiebreak)
Silent LWW on high-value data	Loses user intent	Field-merge / manual / keep-both
No base version (can't detect conflict)	Can't tell divergence	Track the version each change was based on
Rolling own CRDT casually	Very hard to get right	Use a library / restrict to simple CRDTs
One strategy for all data	Mismatch	Choose per data type/contention

Best Practices

- **Detect** conflicts with versions/logical clocks (not raw timestamps); keep the **base version** a change was made against.
- **Choose the strategy per data**: LWW for low-contention whole records; field-merge to preserve intent; manual/keep-both for high-value data; **CRDTs** for counters/sets/collaborative text.
- Prefer **server-authoritative** validation for critical/financial data.

- Make resolution **deterministic and documented**; test conflicting-edit scenarios.
- Run heavy merges off-isolate; surface unresolved conflicts to the user when needed.

Performance

LWW/field-merge are O(fields); CRDTs carry extra state/merge cost but avoid round-trips. Off-isolate large merges; conflict handling is infrequent relative to reads (Module 21).

Advantages / Disadvantages

- **+ LWW:** trivial. **+ Field-merge:** preserves more intent. **+ CRDT:** automatic, order-independent convergence (great for collaboration).
- **− LWW:** data loss. **− Field-merge:** overlaps still need tiebreak. **− CRDT:** complexity/state overhead. **− Manual:** UX burden.

Interview Questions

1. 🟢 **What causes a sync conflict?** — The same record changed both locally and remotely (divergent versions) before syncing.
2. 🟢 **What is last-write-wins and its downside?** — Newest change wins by timestamp/version; it silently discards the losing side's change (data loss).
3. 🟡 **Why prefer versions/logical clocks over timestamps?** — Device clocks skew; versions/vector clocks reliably capture causal order and divergence.
4. 🟡 **What is field-level merge?** — Merge non-overlapping field changes automatically; only fields both sides changed need a tiebreak — preserving more intent than whole-record LWW.
5. 🟡 **What are CRDTs and when do you use them?** — Data types that merge automatically and commutatively to a consistent state (counters, sets, sequences); ideal for collaborative editing/counters.
6. 🔴 **How do you resolve conflicts for financial data?** — Prefer server-authoritative validation / manual resolution — never silent LWW that could lose money.
7. 🔴 **How do you choose a strategy?** — By data type and contention: whole-record→LWW/field-merge, counters/collab→CRDT, critical→manual/server.

Senior Engineer Tips

- Track the **base version** each local edit was made against — without it you can't distinguish a fast-forward from a real conflict.
- Don't hand-roll complex CRDTs; use libraries or restrict to well-understood ones (G-Counter, LWW-register, OR-Set).

- Default to field-merge for editable records (better UX than whole-record LWW); reserve manual for high-stakes overlaps.

Architect Perspective

Conflict resolution is where offline-first correctness is won or lost. Choosing per-data strategies (with version-based detection, server authority for critical data, and CRDTs for collaboration) and documenting them prevents silent data loss and surprises. It's a core part of the sync design (02_sync_engine.md) and the hardest to retrofit — decide early (Module 48).

Summary

- Conflicts arise from concurrent local+remote edits; detect via versions/logical clocks (not timestamps).
- Strategies: LWW (simple/lossy), field-merge (preserves intent), manual/keep-both (high-value), CRDTs (auto-converge for collaboration).
- Choose per data type; prefer server authority for critical data; make resolution deterministic and tested.

Revision Notes

- Detect: versions/vector clocks + base version (timestamps unreliable).
- LWW (lossy) / field-merge (intent-preserving) / manual/keep-both / CRDT (auto-converge: counters/sets/text).
- Choose per data + contention; server-authoritative for critical/financial.
- Don't hand-roll complex CRDTs; heavy merges off-isolate; test conflicts.

Practice Questions

1. Why are timestamps a poor basis for LWW?
2. When is a CRDT the right choice?
3. Which strategy for a collaborative counter vs a bank balance?

Coding Questions

1. Implement version-based LWW and a field-level merge for a record.
2. Detect a conflict using a base version + local/remote versions.
3. Implement a simple G-Counter CRDT (per-replica counts, merge = max).

Mini Project

Conflict resolver (Dart): Implement conflict detection (base+local+remote versions) and three resolvers — LWW, field-merge, and a G-Counter CRDT — with tests covering fast-forward, overlapping-field, and concurrent-counter cases. Wire the chosen resolver into the sync engine's apply step. Acceptance: correct detection; per-strategy resolution; CRDT converges regardless of merge order; tests pass.

Connectivity & the Outbox (Queued Mutations)

> Offline writes must not be lost: record each mutation in a durable **outbox** (a persisted queue) and **flush** it to the server when connectivity returns — with idempotency, ordering, and retry — so the app accepts writes anytime and reliably delivers them later.

Introduction

Reads offline are easy (local store); **writes** offline are the hard part — you can't reach the server, but you must not drop the user's action. The **outbox pattern** persists pending mutations and replays them on reconnect. This file covers connectivity detection and the outbox (queue) mechanics.

Why this concept exists

If an offline write only lived in memory, it'd be lost on app kill. And when connectivity returns, queued writes must reach the server exactly once, in a sensible order, surviving retries. The outbox makes offline mutations **durable and eventually-delivered**.

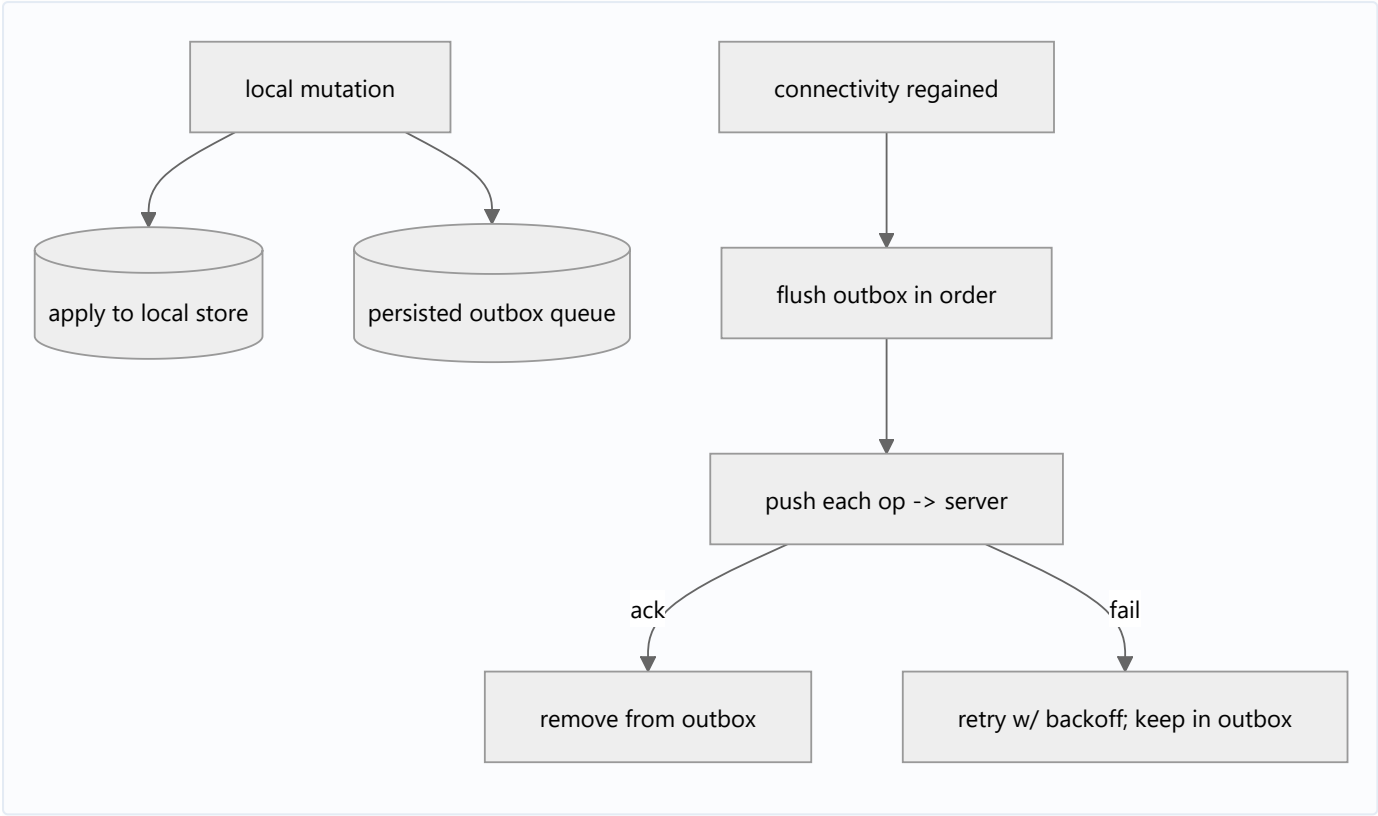
Real-world analogy

An **outbox tray**: when the mail can't go out (offline), letters (mutations) sit in the tray persistently. When the courier arrives (connectivity), they're mailed in order; if delivery fails, they stay in the tray for the next attempt — nothing is lost.

Problem Statement

A user creates and edits notes with no connection, then closes and reopens the app; when connectivity returns, all their changes must reach the server exactly once, in order. You'll persist mutations in an outbox and flush on connectivity.

Internal Working



- **Outbox**: a **persisted** queue (DB table) of pending operations (create/update/delete), each with an id, payload, timestamp, and attempt count. Written **transactionally with** the local change so they never diverge.
- **Connectivity detection**: `connectivity_plus` reports online/offline transitions; but "has a network" ≠ "server reachable" — verify with a real request/heartbeat before assuming success.
- **Flush**: on connectivity (or app resume/interval/background job), send queued ops **in order**; on server ack, **remove** the op; on failure, keep it and **retry with backoff**.
- **Idempotency**: each op carries a **client-generated id/dedupe key** so server upserts are safe under retries (the server may have applied it before the ack was lost).
- **Ordering & dependencies**: preserve causal order (create before update before delete of the same entity); coalesce redundant ops (many edits → last state) where safe.
- **Failure handling**: distinguish transient (retry) vs permanent (4xx validation → surface/park the op) errors; cap attempts; avoid poison-message loops.
- Integrates with the sync engine's **push** phase (02_sync_engine.md) and conflict resolution (03_conflict_resolution.md).

Memory Representation

The outbox is on disk (durable across restarts); only the flushing batch is in memory. Attempt counts/backoff state persist per op (Module 20).

Compiler Behavior

Not applicable.

Runtime Behavior

Writes enqueue instantly (UI unaffected). Flush runs opportunistically; failed ops remain queued and retry; app restart resumes flushing from the persisted outbox.

Flutter Engine Behavior

Connectivity plugins cross the embedder; background flush uses WorkManager/background isolates (Module 33, 02 · isolates).

Dart VM Behavior

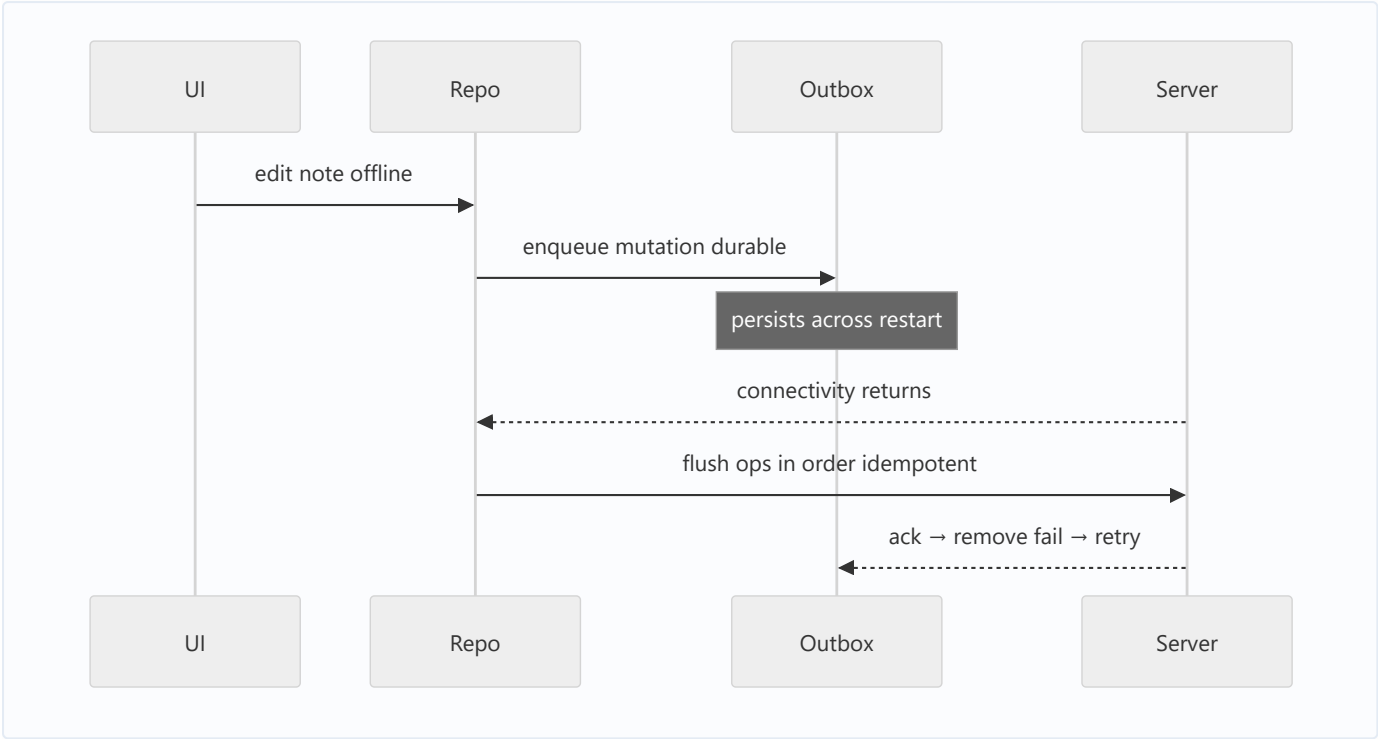
Not applicable.

Examples

```
import 'dart:async';

class Mutation {
  final String id;            // client-generated (idempotency key)
  final String type;          // 'create' | 'update' | 'delete'
  final Map<String, dynamic> payload;
  final int attempts;
  const Mutation(this.id, this.type, this.payload, {this.attempts = 0});
}

abstract interface class Outbox {          // persisted (DB-backed)
  Future<void> enqueue(Mutation m);
  Future<List<Mutation>> pending();        // ordered
  Future<void> remove(String id);
  Future<void> bumpAttempts(String id);
}
abstract interface class RemoteApi { Future<void> apply(Mutation m); }
abstract interface class Connectivity { Stream<bool> get onOnline; Future<bool> get isOnline; }

class OutboxSync {
  final Outbox outbox;
```

```
  final RemoteApi remote;
  final Connectivity connectivity;
  bool _flushing = false;
  OutboxSync(this.outbox, this.remote, this.connectivity) {
    connectivity.onOnline.listen((online) { if (online) flush(); }); // flush on reconnect
  }

  // Called by the repository on every offline-capable write:
  Future<void> record(Mutation m) => outbox.enqueue(m); // durable, transactional with local write

  Future<void> flush() async {
    if (_flushing || !await connectivity.isOnline) return;
    _flushing = true;
    try {
      for (final m in await outbox.pending()) {         // in order
        try {
          await remote.apply(m);                         // idempotent (upsert by m.id)
          await outbox.remove(m.id);                      // ack -> remove
        } on Exception {
          await outbox.bumpAttempts(m.id);                // keep + retry later (backoff/cap)
          break;                                          // stop on first failure (preserve order)
        }
      }
    } finally {
      _flushing = false;
    }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
In-memory-only queue	Lost on app kill	Persist the outbox (DB)
Assuming "online" = server reachable	Captive portals/dead servers	Verify with a real request
Non-idempotent ops	Duplicates on lost-ack retries	Client id / dedupe key + server upsert
Ignoring order/dependencies	create-after-update errors	Preserve causal order; stop-on-failure
Infinite retry on 4xx (poison)	Stuck queue	Distinguish permanent errors; cap/park
Not writing outbox transactionally with local	Divergence (local changed, no op)	Enqueue in the same transaction

Best Practices

- **Persist** the outbox and enqueue **transactionally** with the local write (never lose an op).
- Use **client-generated ids/dedupe keys** + **idempotent** server upserts so retries are safe.
- **Flush on connectivity/resume/interval/background**; verify server reachability, not just "has network."
- Preserve **causal order**; **stop on failure** (or dependency-aware ordering); **retry with backoff**, cap attempts, and **park poison messages**.
- Coalesce redundant ops where safe; integrate with the sync engine's push + conflict handling.
- Do background flush via **WorkManager** (Module 33).

Performance

Enqueue is instant (UI unaffected); flush batches network work opportunistically. Bounded/persisted queue avoids memory growth; backoff prevents storms (Module 21).

Advantages / Disadvantages

- **+** Never-lose-writes offline, reliable eventual delivery, instant UX, resilient to restarts/flaky networks.
- **−** Durable-queue + idempotency + ordering complexity, poison-message handling, retry/backoff tuning.

Interview Questions

1. 🟢 **What is the outbox pattern?** — A persisted queue of pending mutations that's flushed to the server when connectivity returns, so offline writes aren't lost.
2. 🟢 **Why must the outbox be persisted?** — In-memory queues are lost on app kill; durability guarantees eventual delivery across restarts.
3. 🟡 **Why idempotency, and how?** — A lost ack can cause a retry after the server already applied the op; client-generated ids + server upsert make replays safe.
4. 🟡 **Why isn't "device online" enough to flush?** — Captive portals/dead servers mean network ≠ reachable; verify with a real request before treating flush as successful.
5. 🟡 **How do you handle ordering?** — Preserve causal order (create→update→delete of an entity); typically stop on first failure to avoid out-of-order application.
6. 🔴 **How do you handle a permanently-failing op (poison message)?** — Detect permanent (e.g., 4xx validation) errors, cap attempts, and park/surface it instead of infinite retry blocking the queue.
7. 🔴 **Why enqueue transactionally with the local write?** — So the local state and the pending op never diverge (no "changed locally but no op to sync" or vice versa).

Senior Engineer Tips

- Store the outbox in the same DB and **enqueue in the same transaction** as the local mutation — atomicity here prevents the nastiest sync bugs.
- Always attach a **client id** to mutations; it's your idempotency key and correlation id for conflict handling.
- Separate transient vs permanent failures early; a poison message that infinitely retries can block all delivery.

Architect Perspective

The outbox is the durability guarantee of offline-first writes — the write-side counterpart to the local-source-of-truth read model. Combined with the sync engine (push), conflict resolution, and connectivity/background execution, it delivers "accept writes anytime, deliver reliably later." Its idempotency/ordering/durability discipline is core to data integrity at scale (02_sync_engine.md, 03_conflict_resolution.md, Module 33).

Summary

- Persist offline mutations in a durable **outbox**, enqueued transactionally with the local write; flush on connectivity/resume/background.
- Ensure **idempotency** (client ids + server upsert), preserve **order**, retry with **backoff**, and park poison messages.
- Verify real reachability; integrate with sync/conflict handling; background flush via WorkManager.

Revision Notes

- Outbox = persisted queue of mutations (id/type/payload/attempts); enqueue transactional with local write.
- Flush on connectivity/resume/interval/background; verify reachability (not just "online").
- Idempotent (client id + upsert); preserve causal order; stop-on-fail; backoff + cap + park poison.
- Integrates with sync push + conflict resolution; background via WorkManager.

Practice Questions

1. Why persist the outbox instead of keeping it in memory?
2. How do you make replayed mutations safe?
3. How do you prevent a poison message from blocking the queue?

Coding Questions

1. Implement an `OutboxSync` that enqueues mutations and flushes in order on connectivity.
2. Add idempotency (client id + upsert) and retry-with-backoff + attempt cap.
3. Enqueue the outbox op transactionally with the local write (pseudocode/DB).

Mini Project

Durable outbox (Flutter): Build an `Outbox` (DB-backed) + `OutboxSync` that records offline note mutations, persists across restart, flushes in causal order on a fake connectivity event with idempotent upserts, retry/backoff, and poison-message parking. Prove writes survive app kill and deliver once. Acceptance: durable + transactional enqueue; idempotent/ordered flush; survives restart; poison handled; runs.

Optimistic UI (Updates & Rollback)

> Optimistic UI applies a change to the local state **immediately** — assuming success — so the UI feels instant; if the server/sync later rejects it, you **roll back** (or reconcile) to the confirmed state.

Introduction

In offline-first, writes hit the local store and the UI updates instantly (optimistic), while sync confirms with the server in the background. This file covers doing that safely: apply-immediately, track pending state, and **roll back** on failure — the UX payoff of offline-first.

Why this concept exists

Waiting for a server round-trip before showing a "like," sent message, or edited field feels slow. Optimistic UI removes that latency by trusting the local write and reconciling later — but it must handle the rare rejection gracefully (rollback) so the UI never lies permanently.

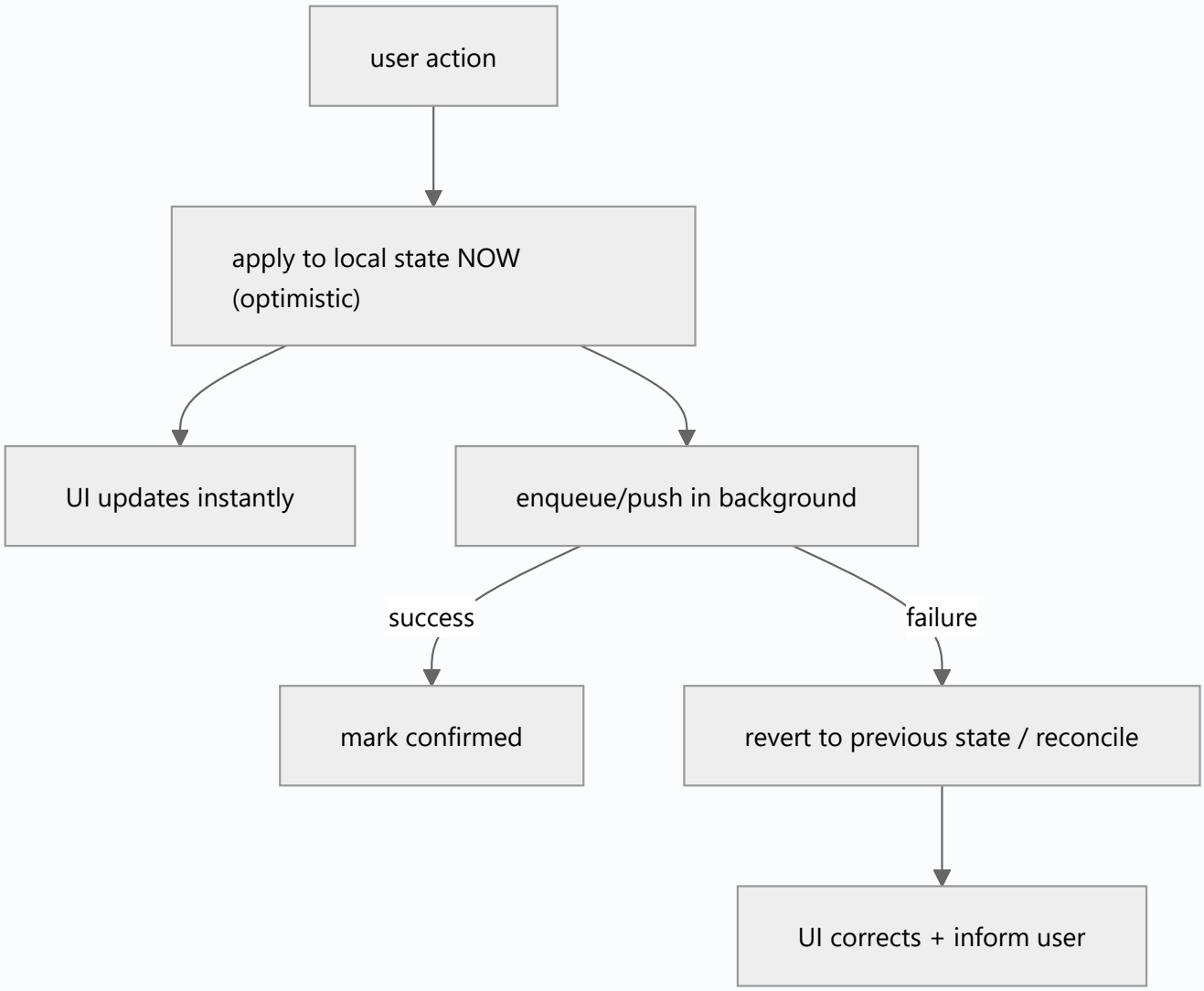
Real-world analogy

Optimistic UI is **speaking before the interpreter confirms**: you say your line immediately (instant UX), and if the interpreter reports it was misheard/rejected, you **correct it** (rollback). Fast by default, corrected when necessary.

Problem Statement

A "like" button should turn filled instantly (offline too), a message should appear as "sent" immediately, and if the server later rejects the like, the UI should revert. You'll apply optimistically, mark pending, and roll back on failure.

Internal Working



- **Apply immediately**: mutate the local store/state at once; the UI (reading local) reflects it instantly — works offline (queued via outbox — 04_connectivity_and_outbox.md).
- **Track pending/confirmed**: mark the item as `pending` (optionally show a subtle indicator); on server ack → `confirmed`; on reject → rollback.
- **Rollback**: keep the **previous value** (or a reverse operation) to restore if the server rejects (validation/conflict). Then optionally inform the user or re-apply server truth.
- **Reconciliation**: when sync pulls server state, it may **replace** the optimistic value with the authoritative one (which usually matches). Conflicts route to conflict resolution (03_conflict_resolution.md).
- **Idempotency**: pair with client ids so retries don't double-apply (04_connectivity_and_outbox.md).
- **Scope**: great for low-risk, reversible actions (likes, edits, reorder); be cautious for irreversible/financial actions (prefer confirmation there).

Memory Representation

Keep the prior value (or inverse op) for rollback until confirmation; a pending set/flag tracks in-flight optimistic changes (Module 11).

Compiler Behavior

Not applicable.

Runtime Behavior

The UI updates on the local write (frame-fast); background sync confirms/rejects later; on reject the local state is reverted and the UI re-renders. Rare and brief; most optimistic updates confirm.

Flutter Engine Behavior / Dart VM Behavior

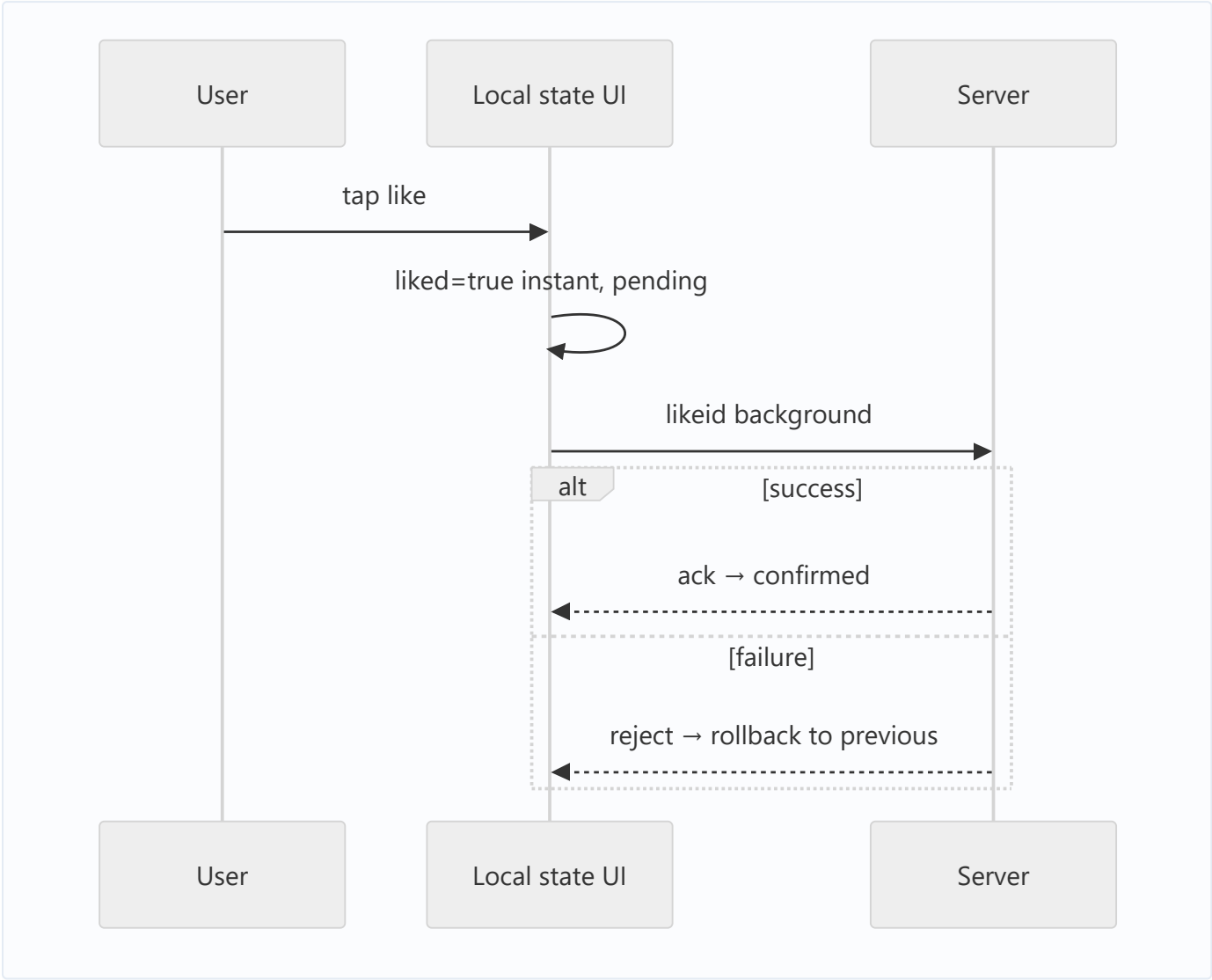
Not applicable.

Examples

```
class Post {
  final String id;
  final bool liked;
  final int likes;
  final bool pending;   // optimistic in-flight
  const Post(this.id, this.liked, this.likes, {this.pending = false});
  Post copyWith({bool? liked, int? likes, bool? pending}) =>
      Post(id, liked ?? this.liked, likes ?? this.likes, pending: pending ?? this.pending);
}

abstract interface class PostStore {           // local source of truth
  Post get(String id);
  void set(Post p);                             // updates local + notifies UI
}
abstract interface class LikeApi { Future<void> like(String id, bool liked); }

class LikeController {
  final PostStore store;
  final LikeApi api;
  LikeController(this.store, this.api);
```

```
  Future<void> toggleLike(String id) async {
    final prev = store.get(id);                 // keep previous for rollback
    // 1) OPTIMISTIC: apply immediately (UI instant, works offline)
    final optimistic = prev.copyWith(
      liked: !prev.liked,
      likes: prev.likes + (prev.liked ? -1 : 1),
      pending: true,
    );
    store.set(optimistic);
    try {
      await api.like(id, optimistic.liked);     // background confirm (or via outbox)
      store.set(optimistic.copyWith(pending: false)); // confirmed
    } catch (_) {
      store.set(prev);                          // 2) ROLLBACK to previous state
      // optionally: show a snackbar "Couldn't like, try again"
    }
  }
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No rollback on failure	UI shows a lie permanently	Keep previous value; revert on reject
Optimistic for irreversible/financial actions	Bad if it fails	Require confirmation for high-risk actions
Not tracking pending state	Can't reconcile/indicate in-flight	Mark `pending` until confirmed
Double-apply on retry	Duplicated effect	Idempotency (client id)
Ignoring server-truth reconciliation	Local drifts from server	Let sync replace with authoritative value
Blocking UI to await server	Defeats the point	Apply local first, confirm in background

Best Practices

- **Apply locally first** (instant UX), enqueue/push in the background; keep the **previous value** for rollback.

- Track **pending** → **confirmed/rolled-back**; optionally show subtle pending indicators.
- **Roll back** on rejection and **inform the user**; reconcile with server truth on sync.
- Use **idempotency** (client ids) so retries don't double-apply.
- Reserve optimistic UI for **low-risk, reversible** actions; require confirmation for irreversible/financial ones.

Performance

Removes perceived latency (frame-fast updates) — a major UX win. Rollbacks are rare/brief. No extra cost beyond keeping a prior value (Module 21).

Advantages / Disadvantages

- **+** Instant, responsive UX; works offline; hides network latency; feels native.
- **−** Rollback/reconciliation complexity; brief incorrect state on failure; unsuitable for irreversible actions; needs idempotency.

Interview Questions

1. 🟢 **What is optimistic UI?** — Applying a change locally immediately (assuming success) so the UI is instant, reconciling/rolling back if the server later rejects it.
2. 🟢 **What must you keep for rollback?** — The previous state (or an inverse operation) to restore if the server rejects the change.
3. 🟡 **How does it relate to offline-first?** — Writes hit the local source of truth instantly (optimistic); the outbox/sync confirms with the server in the background.
4. 🟡 **Why track a pending state?** — To indicate in-flight changes and to know what to reconcile/roll back on ack/reject.
5. 🟡 **When should you NOT use optimistic UI?** — For irreversible/high-risk/financial actions where a failed assumption is costly — require confirmation instead.
6. 🔴 **Why is idempotency needed?** — Retries (lost acks) could double-apply the change; client-id-based idempotency prevents duplicates.
7. 🔴 **How do you reconcile with server truth?** — On sync pull, replace optimistic values with authoritative ones; route genuine conflicts to conflict resolution.

Senior Engineer Tips

- Model state so rollback is trivial: keep `previous` or use immutable `copyWith` and restore it — don't try to "undo" mutations in place.
- Pair optimistic UI with the outbox: the optimistic local change *is* what gets synced; rollback = revert local + drop/mark the op.

- Show restraint: optimistic for likes/edits/reorder; explicit confirmation for payments/deletes-of-record.

Architect Perspective

Optimistic UI is the UX dividend of offline-first: it makes local-source-of-truth feel instant while sync/outbox/conflict handling ensure eventual correctness. Designed with pending/rollback/idempotency and scoped to reversible actions, it delivers native-feeling responsiveness — completing the offline-first stack (01_offline_first_fundamentals.md, 04_connectivity_and_outbox.md, 03_conflict_resolution.md).

Summary

- Optimistic UI applies changes locally immediately and rolls back on server rejection.
- Track pending→confirmed/rolled-back; keep the previous value; use idempotency; reconcile with server truth.
- Instant UX for reversible actions; require confirmation for irreversible ones.

Revision Notes

- Apply local now (instant) → background confirm → rollback on reject (keep previous value).
- Track `pending`; idempotency (client id) for retries; reconcile with server truth on sync.
- Reversible/low-risk only; confirm irreversible/financial actions.
- Pairs with outbox (the optimistic change is what syncs).

Practice Questions

1. What do you keep to enable rollback?
2. Why is optimistic UI risky for financial actions?
3. How does it combine with the outbox?

Coding Questions

1. Implement optimistic like-toggle with rollback on failure.
2. Add a pending indicator + idempotent retry.
3. Reconcile an optimistic value when sync pulls the authoritative one.

Mini Project — Module capstone

Offline-first notes (Flutter): Combine all five files: local store as source of truth (instant reads/writes), **optimistic** edits with rollback, a durable **outbox** for offline mutations, a **sync engine** (delta push/pull), and **conflict resolution** (LWW/field-merge) — all behind a `NotesRepository` exposing a reactive stream and a `SyncState`. Prove: works fully offline, survives restart, reconciles + resolves conflicts on reconnect, and rolls back rejected optimistic writes. Acceptance: instant/optimistic UX; never-lost writes; delta sync; conflicts resolved; rollback works; runs.