Flux

Offline-First & Sync Strategy

Baseline Release: Phase 0 (Walking Skeleton) + Phase 1 (Trimmed MVP)

This document explains, in clear and testable rules, how Flux behaves when the device is offline and how it synchronises data to Supabase when connectivity returns. It is written to match the Project Charter, Trimmed MVP, Baseline SRS, Requirements Catalogue, and the Core ERD decisions.

Field	Value
Document ID	F-OFSS-001
Version	1.0 (baseline)
Status	Draft for implementation
Applies to	Phase 0 + Phase 1
Tech stack	Next.js (App Router), TypeScript, Tailwind CSS, Supabase, IndexedDB (PWA)
Core decisions	UUID primary keys; location_id + RLS isolation; client-side sync queue; LWW conflicts

Primary reference files (project sources)

- Flux - Project Charter New.pdf
- MVP Definition for Flux (New).pdf
- Flux Baseline SRS.pdf
- Flux Baseline SRS Requirements Calatague.pdf
- Database architecture design (Core ERD).pdf
- Roadmap for Flux Development.pdf
- Flux latest scope.pdf
- Documentations Required in Each Release.docx

1. Purpose and scope

Flux is an offline-first POS and inventory system. “Offline-first” means the app still lets users do core work even with no internet, and it does not lose data if the tab is closed or the device restarts. When internet comes back, Flux synchronises changes to Supabase automatically.

This document defines:

- What must work offline in Phase 0 and Phase 1 (and what is allowed to be online-only).
- How data is stored on the device (IndexedDB) and how the sync queue is structured.
- Sync rules (pull and push), ordering, retries, and failure handling.
- Conflict handling rules, including the initial Last-Write-Wins (LWW) policy.
- ID strategy (UUID), versioning approach, and audit trail expectations.

This strategy is written to align with Flux’s baseline constraints: Supabase Row Level Security (RLS) for branch isolation using location_id, UUID primary keys for offline generation, and the Core ERD decision that sync flags live only on the client.

2. Key principles and definitions

To keep implementation simple and reliable, Flux uses the following principles:

- System of record: Supabase/PostgreSQL is the long-term system of record. Local data is a cache and an outbox.
- Outbox pattern: Every offline change is written to IndexedDB first, then synced later (no “in-memory only” work).
- Stable identity: Every business record has a UUID generated on the client, so retries don’t create duplicates.
- Branch isolation: Every cached row and every sync request is scoped by location_id to match RLS rules.
- Inventory correctness: Stock is driven by an append-only ledger (stock_moves) and a snapshot (stock_balances).
- Simple conflict rule (baseline): Last-Write-Wins (LWW) using updated_at for master edits.

Terminology

Term	Meaning in Flux
Location / Branch	A shop or warehouse. Data is isolated by location_id and enforced by RLS.
Cached masters	Items, categories, units, customers, suppliers, lots, and settings needed for offline work
Queue / Outbox	IndexedDB list of unsynced transactions (sale, GRN, transfer, adjustment) with metada
Push	Sending queued local changes to Supabase when online.
Pull	Fetching server changes to refresh the local cache (deltas since last sync).
Conflict	Two different edits to the same logical record (e.g., item name changed on two devices

Idempotent sync	Repeating a sync request does not create duplicate server records.

3. What must work offline

Offline capability is defined by phase. Phase 0 proves the architecture end-to-end. Phase 1 makes it usable in real shops.

3.1 Phase 0 (Walking Skeleton) - offline minimum

The following actions must work offline in Phase 0:

- POS sale (hot path): create a sale, add item lines from cached catalog, post the sale, and store it locally.
- Receipt printing: printing after posting is supported (browser print). If printing needs internet, show a clear message.
- Durability: if the tab closes/restarts, the pending sale remains in IndexedDB and can sync later.
- Offline UI signals: user sees Online/Offline state and a Pending Sync count.

Phase 0 does not require offline-first administration (e.g., creating new users) or large reports. The user must log in while online at least once. After that, the app can continue working offline until a re-login is needed.

3.2 Phase 1 (Trimmed MVP) - offline operational scope

In Phase 1, Flux extends offline support to cover day-to-day operations:

- POS sales: same as Phase 0, but now includes cash/card/credit payments and optional customer selection (customer is mandatory for credit).
- GRN (Goods Received Note): create and post GRNs offline (including batch/expiry capture).
- Stock transfers: create transfer-out offline; destination can receive when online or offline (then sync later).
- Stock adjustments: simple increase/decrease adjustments offline with a reason note.
- Masters (recommended offline): create/edit items, customers, suppliers, categories, units, locations; queue changes for sync.
- Basic reporting from local data: show recent sales and a basic stock balance view from the last synced snapshot plus pending moves.

Some features may be kept online-only in Phase 1 if they increase risk (e.g., complex cross-branch reports). If a feature is online-only, Flux must show a clear “needs internet” message and never lose the user’s work.

4. Data that must be available offline

To sell and manage stock while offline, Flux must store a small but sufficient set of data on the device. All local data is namespaced by location_id.

4.1 Cached master data (read cache)

- Items: id (UUID), name, barcode, unit, selling price, and flags (batch-tracked or not).
- Categories and Units used by items.
- Customers (Phase 1): basic details + credit_limit + credit_days for credit checks.
- Suppliers (Phase 1): basic details for GRNs.
- Lots/Batches (Phase 1): stock_lots with batch_number and expiry_date for batch-tracked items.
- Branch settings needed for POS rules (e.g., negative stock policy WARN/BLOCK).

4.2 Local operational data (write stores)

- Draft documents: sales drafts, GRN drafts, transfers drafts (if your UI supports saving drafts).
- Outbox / Sync queue: all posted offline transactions waiting to be synced.
- Local sync metadata: last_sync_at per entity group (items, customers, suppliers, lots, documents).
- Optional local “computed” views: recent sales list; on-hand snapshot last pulled from server.

Important: Offline “synced/failed” flags live only in IndexedDB. Once a row exists in Supabase, it is treated as synced. This matches the Core ERD decision and avoids polluting the database with client state.

5. IndexedDB schema and queue item format

Flux uses IndexedDB because it can store large structured data safely and works well for offline PWAs. To keep sync predictable, Flux uses a small number of stores:

- catalog: cached masters by entity type (items, categories, units, customers, suppliers, lots).
- drafts: user drafts that are not yet posted (optional but recommended).
- outbox: the sync queue (posted offline work).
- sync_meta: last_sync_at per entity type and per location_id.
- errors: optional log of permanent sync failures for support.

5.1 Required fields for each outbox (queue) record

Field	Type	Why it exists
queue_id	UUID	Unique ID for the queue row (not the business record).
entity_type	string	Example: sales_invoice, grn, stock_transfer, stock_adjustment, master_updat
entity_id	UUID	The business record ID (matches the server primary key).
location_id	UUID	Required for RLS and for local namespacing.
payload	JSON	Full document payload (header + lines + child records).
client_created_at	timestamp	Client time when created; useful for ordering and auditing.
client_updated_at	timestamp	Client time when last edited; used for conflict policy inputs.
sync_status	enum	pending | syncing | synced | failed
retry_count	int	How many times we tried to sync this item.
last_attempt_at	timestamp	Last time we tried to sync.
next_retry_at	timestamp	When to try again (backoff).
last_error	string	Short error summary for the user / support.

The payload should be self-contained: for example, a sales invoice payload includes the invoice header, lines, and payments. This avoids partial sync and keeps each sync request atomic.

6. Server-side alignment (Supabase and Core ERD)

The sync strategy must match Flux's database rules and ERD so that the server can safely accept retries and maintain stock correctness.

6.1 Stable IDs and dedupe controls

- All primary keys are UUIDs, generated client-side (works offline).
- Document identity is protected by unique constraints, e.g., invoice_number unique per location, GRN number unique per location.
- Barcode is unique per location (prevents duplicate scans mapping to multiple items).
- Stock balances have uniqueness constraints to prevent duplicate balance rows under offline retries.

6.2 Inventory update model (ledger + snapshot)

Flux uses an inventory ledger (stock_moves) as the source of truth and a snapshot table (stock_balances) for fast current on-hand. Posting a document inserts stock_moves and updates stock_balances. This is important for sync because it makes stock updates deterministic even when documents arrive later from offline devices.

6.3 RLS and location_id requirements

- Every master and transactional row is scoped by location_id (except truly system-wide configs).
- Supabase RLS policies must block cross-branch reads/writes. The client must always include location_id in requests.
- For transfers, the transfer header carries its own location_id equal to from_location_id to keep RLS simple.

7. Sync flow (pull and push)

Flux sync is a repeating cycle that runs whenever the device is online. The baseline order is: Pull first (refresh masters) then Push (send queued transactions).

7.1 When sync runs

- On app start (after login).
- When network changes from offline to online.
- On a timer while the app is open (e.g., every 30–60 seconds).
- Optionally via Background Sync API where supported (best-effort).

7.2 Pull rules (server → device)

Pull keeps the local cache fresh. Pull MUST be scoped by location_id and SHOULD use delta queries (only changed rows) based on updated_at.

- Maintain last_sync_at per entity type (items, customers, suppliers, lots, settings).
- Fetch rows with updated_at > last_sync_at for the current location_id.
- Upsert into the local catalog store (replace by id).
- After successful pull, update last_sync_at.

7.3 Push rules (device → server)

Push processes the outbox in FIFO order (oldest first). Each outbox record is synced as a single atomic unit (one transaction on the server) so that headers and lines do not get out of sync.

- Select the next pending outbox item where next_retry_at <= now().
- Send payload to the server using an upsert/insert approach that is safe to retry (idempotent).
- If server accepts and commits: mark as synced and remove from “pending” list.
- If server rejects with a permanent error (validation/RLS): mark as failed and show the error.
- If network/server error: increment retry_count and schedule next_retry_at using backoff.

Important ordering rule: If a transaction references other data, the sync engine must ensure the dependency exists on the server. Example: a GRN line references item_id and (optional) lot_id. Therefore, masters must be pulled first and missing masters must be handled.

8. Idempotency and duplicate prevention

Offline sync retries are normal. Flux must behave safely when the same queued item is posted more than once.

8.1 Baseline dedupe mechanisms

- UUID primary keys: the same entity_id is used every time the client retries.
- Unique indexes: server rejects duplicates for barcode, invoice_number, GRN number, and balance rows.
- Upsert strategy: prefer insert-once semantics for posted documents; if the same UUID exists, treat as already synced.

8.2 Transaction atomicity

A common failure mode is “header created but lines failed”. To avoid this, Flux SHOULD sync documents using a single server transaction (e.g., a Supabase RPC function) that inserts header + lines + payments and then updates stock moves/balances together. If any part fails, the server rolls back and the client will retry safely.

9. Conflict handling (baseline rules)

Conflicts are most common when two devices edit the same master record while offline. Flux uses a simple baseline policy: Last-Write-Wins (LWW) based on updated_at timestamps. The newest update wins.

9.1 Master data conflicts (items, customers, suppliers, etc.)

- If the same master record is edited on two devices, the edit with the newer updated_at overwrites the older one on the server.
- After sync, the client pulls the final server row and updates its local cache.
- Flux should log conflicts (at least locally) so managers can review if needed (Phase 2 can add a conflict review UI).

9.2 Transaction conflicts (sales/GRN/transfer/adjustment)

Posted transactions are treated as immutable business facts. In Phase 0+1, Flux minimises “editing posted documents”. Instead, corrections happen by creating new documents (e.g., a return/refund) rather than rewriting history.

- If the same transaction is posted twice due to retries, the server must ignore the duplicate (idempotency).
- If two different sales consume the last stock at the same time, negative stock policy applies (WARN/BLOCK). Stock ledger remains consistent.
- Transfers are two-step: source posts Transfer-Out, destination posts Receive. If receive happens twice, server must de-duplicate by UUID.
- GRNs can be edited while draft; once posted, no post-adjustments in baseline.

9.3 “Same item updated in two branches”

In the current Core ERD and baseline SRS, most masters are location-scoped (each row has location_id). That means Branch A editing its item does not change Branch B’s item row, so cross-branch conflicts do not occur. If Flux later introduces global (shared) items across branches, then the conflict rule becomes important across branches too. The recommended future design is: global item master (tenant-wide) + per-branch override tables for price/stock rules. For Phase 0+1, treat masters as branch-scoped unless explicitly configured otherwise.

10. Retry rules and failure cases

Flux must be clear and predictable when sync fails. The user should never wonder whether data is lost.

10.1 Error categories

Category	Examples	How Flux handles it
Transient	No internet, DNS failure, timeout, server 5xx	Retry with backoff; keep item pending; show pending count
Authentication	Token expired, user logged out, session invalid	Pause sync; ask user to log in when online; keep outbox int
Validation	Missing required field, invalid total, missing batch expiry for batch-tracked item	Mark as failed; show error; allow user to fix and re-queue.
RLS / Permission	User tries to sync data for another location	Mark as failed; show “not allowed”; do not retry automatica
Dependency missing	Item_id not on server, lot_id invalid, foreign key missing	Add to pull-first queue; retry; if still missing, fail with clear messa

10.2 Backoff and retry schedule (recommended baseline)

- Use exponential backoff for transient failures (example: 10s, 30s, 2m, 5m, 15m, 30m).
- Cap automatic retries (example: 10 attempts). After cap, keep as Failed and require manual retry.
- Always allow “Retry now” for a single failed item (manager/admin roles).
- Do not reorder the queue. FIFO is kept to preserve business order unless an item is permanently failed.

10.3 Common failure cases to design for

- Browser closed mid-transaction: data must be durable in IndexedDB (no data loss).
- Partial server insert: avoid by syncing via atomic server transactions (RPC).
- Clock drift causing incorrect LWW: mitigate by using server time where possible or storing client timestamps separately.
- Service worker update issues: after deployment, prompt user to refresh safely; do not break offline shell.
- Large queue buildup: show pending count and optionally warn when queue exceeds threshold (e.g., >200 items).

11. ID strategy, versioning, and audit trail

Flux needs traceability so you can answer: who changed what, when, and why. In Phase 0+1 this is achieved mainly using audit fields and immutable ledgers rather than heavy audit-log UI.

11.1 ID strategy (UUID)

- Every record uses a UUID primary key generated on the client (works offline).
- UUID is the stable identity; human-friendly numbers (invoice_number, grn_number, transfer_number) are secondary identifiers.
- When a document is created offline, it must already have its UUID so sync retries are safe.

11.2 Versioning (baseline)

- Use updated_at as the baseline “version” input for LWW on masters.
- Keep a business status field on documents (draft/posted/cancelled/void) to track lifecycle.
- For inventory, stock_moves provides an immutable history of every stock change (acts as an audit ledger).
- Phase 2+ may add explicit version numbers or a full audit_log table if stronger history is required.

11.3 Audit fields

- All tables include created_at, updated_at, created_by (references auth.users).
- Client payload should carry client_created_at and client_updated_at for transparency (even if server stores its own timestamps).
- For support, store a minimal local sync log: last_error, retry_count, and timestamps per outbox item.

12. Security considerations for offline mode

Offline storage increases risk if a device is lost. Phase 0+1 security rules aim to balance usability and safety:

- Do not store sensitive secrets in IndexedDB (no long-lived tokens, no passwords).
- Cache only what is needed for the user's location_id. Never cache other branches' data.
- If the user logs out, clear local caches for that user/location to prevent the next user seeing old data on a shared device.
- Use HTTPS for all network traffic and rely on Supabase Auth + RLS to enforce access rules server-side.
- If session expires while offline: allow creating offline transactions, but require re-login before syncing them.

13. Acceptance checklist (Phase 0 + Phase 1)

Use this checklist for UAT and sprint acceptance. Each item should be demonstrable on a test device.

- Offline POS: create sale offline, close browser, reopen, sale still pending; reconnect and it syncs once.
- Pending indicator: UI always shows online/offline and pending count.
- No duplicates: force sync retry (same payload) and confirm no duplicate invoice exists on server.
- RLS: user from Location A cannot sync or view Location B records.
- Batch rules (Phase 1): cannot post GRN without batch/expiry for batch-tracked items; cannot post sale without batch selection.
- Transfers (Phase 1): transfer-out then receive updates stock moves/balances correctly; duplicate receive does not double count.
- Error handling: validation errors show as Failed with readable message; transient errors retry with backoff.

14. References

This document is derived from the following Flux project source files (provided by the project owner).

- Documentations Required in Each Release.docx (Sections 9-10)
- Flux - Project Charter New.pdf (offline-first goals, RLS, MVP definition of done)
- MVP Definition for Flux (New).pdf (offline queue + pending sync count, trimmed scope)
- Flux Baseline SRS.pdf (offline logic, sync rules, conflict policy, UI indicators)
- Flux Baseline SRS Requirements Calatague.pdf (testable offline/sync requirements)
- Database architecture design (Core ERD).pdf (schema constraints, uniqueness, no sync flags in DB)
- Roadmap for Flux Development.pdf (phase implementation steps and screens)
- Flux latest scope.pdf (full-scope screens list used to determine what is in/out of Phase 0+1)

End of document.