Flux

Branch & Multi-Location Rules

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

This document defines how Flux behaves in a multi-branch setup: what data is shared vs branch-specific, how Row Level Security (RLS) is applied using location_id, and the end-to-end workflow rules for stock transfers between locations. It is written to be directly usable for implementation and testing.

Field	Value
Document ID	F-BMLR-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 mechanism	location_id + Supabase RLS for branch isolation; UUID keys for offline-safe inserts

1. Purpose and scope

Flux is designed for single shops and multi-branch retail. In Phase 0 and Phase 1, the goal is to keep multi-branch logic simple, safe, and compatible with offline work. This means we follow a strong rule: every business row is owned by exactly one location (using location_id), and access is enforced by Supabase RLS.

This document defines:

- The branch model (what a location is, and how users operate in a location).
- What data is shared across branches vs branch-specific in Phase 0+1.
- How Super Admin, Store Manager, and Cashier access data across locations.
- Stock transfer workflow rules (transfer-out and receive) and the status lifecycle.
- Edge cases: partial receipt, mismatch, cancellation, and offline posting.

2. Branch model (locations and user context)

A Location is a shop or warehouse. Every transaction happens in exactly one location context. Users are assigned to one or more locations based on their role.

2.1 Location identity

- Each location has a UUID id (generated client-side if needed) and a human-friendly name.
- location_id is attached to almost all master and transactional records to enforce isolation.
- Stock is always tracked per location (stock_balances and stock_moves are location-scoped).

2.2 Active location (how the UI should behave)

- A user works in one active location at a time (selected at login or via a switcher).
- All create/edit actions automatically stamp location_id = active location.
- Search screens and lists default to showing only the active location’s records.
- If a user has access to multiple locations (Super Admin), they can switch and the UI reloads data for that location.

2.3 Roles and cross-branch behaviour (Phase 0+1)

Role	Location access	What they can do (Phase 0+1)
Super Admin	All locations (cross-branch)	Setup locations and masters; view/manage data across locations; s
Store Manager	Assigned location only	Manage items, customers, suppliers, GRN, transfers, adjustments;
Cashier	Assigned location only	POS sales, receipt print, view limited sales history; no admin maste

3. Shared vs branch-specific data (Phase 0 + Phase 1)

Because the Core ERD scopes most tables with location_id, the baseline rule is: masters and transactions are branch-owned. This reduces conflicts and keeps RLS policies simple.

3.1 Data scope rules (baseline)

- Branch-owned: if a table has location_id, rows belong to one location. Other locations cannot read/write those rows unless explicitly allowed by RLS (e.g., transfer destination view).
- System-wide: tables without location_id are global (example: the locations table itself; auth.users).
- No silent sharing: Phase 0+1 does not automatically share item/customer rows across branches.

3.2 Practical classification table

Entity	Phase 0+1 scope	Notes / implications
Locations	Shared list	A location is its own record; used as a reference for other tables.
Users (Supabase Auth)	Shared	Auth is global; business data access is filtered by RLS.
Categories	Branch-specific	Each branch can maintain its own category list (simpler baseline).
Units	Branch-specific	Units are also location-scoped in the Core ERD.
Items	Branch-specific	Items include price and barcode uniqueness per location; no global item
Pricing	Branch-specific	Because price lives on the item row (location-scoped), pricing is per bran
Customers	Branch-specific	Customer list is per branch; credit limits/days apply within that branch.
Suppliers	Branch-specific	Supplier list is per branch (can be standardised later).
Stock balances & movements	Branch-specific	On-hand is always per location; never shared.
Sales invoices	Branch-specific	Sales belong to the selling branch only.
GRNs	Branch-specific	Purchasing receipts belong to the receiving branch only.
Transfers	Cross-branch document	Owned by the source (location_id = from_location_id). Destination gets li

Why this is a good baseline: Branch-specific masters remove many cross-branch conflicts (two branches editing the same item row). It also matches the offline requirement: each device caches only its branch data in IndexedDB.

4. Branch rules for pricing, customers, and stock

These rules remove ambiguity when multiple branches exist.

4.1 Pricing rules (Phase 0+1)

- Price is stored on the Items table (location-scoped), so each branch can have its own selling price.
- If the business wants the same price everywhere, the Super Admin must keep items aligned manually in Phase 0+1.
- Phase 0+1 does not include a “global price list” or “price sync across branches” automation.
- If promotions/advanced pricing exist in the full scope, they are deferred beyond Phase 1.

4.2 Customer rules (Phase 1)

- Customers are branch-scoped (location_id). A customer created in Branch A is not automatically visible in Branch B.
- Credit sales require a customer selection. Credit limit and credit days are enforced using the branch’s customer record.
- Cross-branch customer balances (single customer account across all branches) is deferred beyond Phase 1.

4.3 Stock rules

- Stock is tracked per location using stock_moves (ledger) and stock_balances (snapshot).
- A sale reduces stock only in the selling location.
- A GRN increases stock only in the receiving location.
- A transfer moves stock by decreasing the source location first, and increasing the destination only when received.

5. Transfer workflow rules (inter-branch / inter-location)

Transfers move stock between two locations. In Phase 0+1, transfers are a simple two-step workflow: Transfer-Out (source) → Receive (destination). This keeps stock correct even when one side is offline.

5.1 Transfer document ownership and visibility

- Owner: The source location owns the transfer header and lines. The transfer row has location_id = from_location_id (Core ERD rule).
- Destination visibility: The destination location must be allowed (via RLS) to read transfers where to_location_id = its location_id, for the purpose of receiving.
- No unrelated access: Destination should not see source-only masters like customers or unrelated sales.

5.2 Status lifecycle (recommended baseline)

Status	Meaning	Who can set it
Draft	Not posted. Can be edited freely.	Source (Manager/Super Admin)
InTransit	Transfer-out posted. Stock reduced at source.	Source (posting action)
Received	Receive posted. Stock increased at destination.	Destination (Manager/Super Admin)
Cancelled (draft only)	Draft transfer removed/voided before posting.	Source (Manager/Super Admin)

5.3 Step-by-step rules

A) Create transfer-out (source)

- Required: from_location_id, to_location_id, at least one line item with quantity > 0.
- For batch-tracked items: lot/batch must be selected (and must exist in source stock).
- Recommended validation: prevent transfer quantities that exceed available stock (policy can be WARN or BLOCK).
- Save as draft, then post when ready.

B) Post transfer-out (source)

- On posting, create stock_moves that reduce stock at the source location for each line.
- Set status = InTransit.
- After posting, header/lines become read-only in Phase 0+1 (avoid rewriting history).
- If posting happens offline, it is saved to IndexedDB outbox and synced later.

C) Receive transfer (destination)

- Destination searches transfers where to_location_id = destination location and status = InTransit.
- Destination confirms received quantities (baseline: must match transfer quantities).
- On receiving, create stock_moves that increase stock at the destination location.
- Set status = Received.

- If receiving happens offline, the receipt is queued and synced later (must be idempotent).

5.4 Mismatch, partial receipt, and damages (Phase 0+1 policy)

To keep Phase 1 simple and safe, the baseline policy is no partial receiving and no quantity edits on receive. If real life differs (missing/damaged items), handle it using a stock adjustment at the destination (or a return transfer later).

- If destination receives fewer items than sent: receive full quantity only if physically received; otherwise do not receive and escalate.
- If destination needs to accept mismatch: create a stock adjustment with a reason (Phase 1 supports adjustments).
- Partial transfers, transit optimisation, and transfer corrections are deferred beyond Phase 1.

6. RLS rule notes (how access should be enforced)

RLS policies must support both strict branch isolation and transfer receiving. The following describes expected behaviour (exact SQL policies can be documented in the API/RLS document set).

6.1 Baseline expectations

- Users can read/write rows where location_id is in their permitted locations (usually one location for Manager/Cashier).
- Super Admin can read/write across locations (as defined by the requirements catalogue).
- Transfers: destination can read transfer header/lines where to_location_id is their location_id, even though location_id equals the source.
- Receive action: destination can create a receive note that references the transfer and stamps destination location_id on the receive document.

Because transfers are cross-branch by nature, they are the main place where RLS needs an exception. All other masters/transactions remain strictly scoped.

7. Offline considerations for multi-location flows

Offline-first design affects multi-location behaviour in two ways: what is cached locally, and how transfers sync safely.

7.1 What each device is allowed to cache

- A device caches only its active location’s masters and transactions (items, customers, sales, GRNs, stock).
- Transfers are cached if the device is either the source (from_location_id) or the destination (to_location_id).
- A device must never cache unrelated branches’ customers, sales, or financial data.

7.2 Sync safety rules (high-level)

- Transfer-out and receive are synced as separate queued documents. Both must be idempotent using UUIDs.
- Source posting reduces source stock immediately (even offline). Destination stock increases only after receive posts.
- If receive is synced before transfer-out (rare but possible with multiple offline devices), server must reject receive until transfer exists (dependency check) OR accept and reconcile later via a controlled process. Baseline recommendation: reject with a clear dependency error and retry after pull.

8. Acceptance checklist and example scenario

Use this for testing multi-branch rules in Phase 0+1.

8.1 Checklist

- Manager of Branch A cannot view Branch B customers, sales, or stock.
- Manager of Branch A can create transfer-out to Branch B and post it.
- Manager of Branch B can see inbound transfers (to_location_id = Branch B) and post receive.
- Stock decreases in Branch A at transfer-out posting; stock increases in Branch B only at receiving.
- Duplicate sync does not double-count transfer-out or receive (idempotency).
- Super Admin can switch between branches and see each branch’s data as required.

8.2 Example: transfer stock from Branch A to Branch B

- Branch A is low on space, Branch B needs 10 units of Item X.
- Branch A Manager creates Transfer-Out: from Branch A → to Branch B, adds Item X qty 10, posts it.
- Branch A on-hand for Item X decreases by 10 immediately.
- Branch B Manager opens Inbound Transfers, selects this transfer, confirms receipt, posts Receive.
- Branch B on-hand for Item X increases by 10, transfer status becomes Received.

9. Guardrails: not included in Phase 0 + Phase 1

The following are common requests in multi-branch systems, but they are deferred to later phases to keep the baseline stable:

- Global (shared) item master with automatic propagation to all branches.
- Single customer account across all branches (shared receivables).
- Advanced pricing engine, promotions, loyalty, and automatic price syncing.
- Partial transfers, transit tracking, and complex transfer corrections (e.g., split deliveries).
- Cross-branch consolidated accounting (GL, P&L;, Balance Sheet).

These can be added later without breaking Phase 0+1 if we keep the location_id rule consistent and introduce a tenant-wide layer carefully.

10. References (project sources)

This document is aligned with the following Flux project source files provided for this release:

- Documentations Required in Each Release.docx (Section 10: Branch/Multi-Location Rules)
- Flux - Project Charter New.pdf (multi-branch value, location_id + RLS requirement, transfers)
- MVP Definition for Flux (New).pdf (trimmed scope: minimum multi-location transfers; offline constraints)
- Flux Baseline SRS.pdf (branch isolation, roles, transfers, offline expectations)
- Flux Baseline SRS Requirements Calatague.pdf (role access rules; RLS constraints; multi-location permissions)
- Database architecture design (Core ERD).pdf (location_id scoping; transfer header location_id = from_location_id; status lifecycle)
- Roadmap for Flux Development.pdf (phase plan: transfers and receiving; offline queue service expanded in Phase 1)
- Flux latest scope.pdf (full module list used to confirm what is deferred beyond Phase 1)

End of document.