Flux POS & Inventory System

API Specification - Phase 0 (Walking Skeleton) + Phase 1 (Trimmed MVP)

Version: 1.0
Date: January 31, 2026
Backend: Supabase (PostgreSQL + PostgREST + RPC)

This document defines the backend API surface for Flux for Phase 0 and Phase 1. It covers standard CRUD APIs (PostgREST), transactional RPCs (posting sales, GRNs, transfers, adjustments), read-optimized views, authentication, security, and the offline-first sync contract.

Alignment to your project files
• Flux - Project Charter New: offline-first PWA + branch isolation + essential retail flows
• MVP Definition for Flux (New): trimmed MVP boundaries for Phase 0/1
• Flux Baseline SRS + Requirements Catalogue: actors, rules, offline queue, LWW conflict policy
• Roadmap: phased delivery order for a solo build
• Flux latest scope: screen fields that may require API payload extensions
• Database architecture design (Core ERD): tables, keys, and constraints used by the API

Contents

1. Overview and goals

Flux is an offline-first POS and inventory system. For Phase 0 and Phase 1, the backend must support: secure branch isolation (RLS), reliable offline sales + sync, and inventory control using a ledger (stock_moves) plus a snapshot (stock_balances).

Key modules supported by this API (Phase 0 + 1)

• Authentication and role-based access (Super Admin / Store Manager / Cashier).

• Masters: locations, categories, units, items, suppliers, customers.

• POS: sales invoices, invoice lines, payments (cash/card/credit), draft -> posted.

• Inventory: stock_moves (ledger), stock_balances (snapshot), stock_lots (batch + expiry).

• Purchasing: GRN (header + lines) with lot support.

• Transfers: create + receive between locations.

• Adjustments: manager-only stock corrections.

2. Architecture and environments

Flux uses Supabase with three API surfaces:

• **PostgREST Data API** for tables and views.

• **RPC endpoints** (PostgreSQL functions) for multi-step transactions (posting documents).

• **Auth endpoints** handled through Supabase Auth SDK.

Base paths

```
REST (tables/views):  /rest/v1/<resource>
RPC (functions):      /rest/v1/rpc/<function_name>
Auth:                /auth/v1/*
```

Environments: staging and production Supabase projects. Clients select the correct base URL through environment variables in Next.js.

3. API standards

3.1 Authentication

All requests require a valid Supabase access token (except login). Clients should use Supabase SDK for token refresh.

3.2 Required headers

Header	Meaning / Rule
Authorization: Bearer <token>	User session token from Supabase Auth.
apikey: <anon_key>	Supabase project API key (use anon key on client).
Content-Type: application/json	JSON payloads for inserts and RPC calls.
Prefer: return=representation	Return inserted/updated row(s) for immediate UI updates.

3.3 Filtering, sorting, pagination (PostgREST)

```
Filter by location:   ?location_id=eq.<uuid>
Select columns:       ?select=id,name,barcode
Sort (descending):    ?order=created_at.desc
Limit/offset:         ?limit=50&offset=0
Single row:           ?id=eq.<uuid>
```

3.4 Error model

Clients must keep failed items in the offline queue (with retry metadata) and show friendly messages.

HTTP	Meaning	Client action
400	Bad request (missing/invalid fields)	Do not auto-retry; show validation error.
401	Unauthenticated	Re-login / refresh token.
403	Forbidden (RLS / role)	Show 'not allowed'; do not retry.
404	Not found	No retry unless user expects it.
409	Conflict (unique constraint)	Use upsert/idempotency; if still conflict, re-number.
429	Rate limit	Retry with backoff.
500+	Server error / outage	Retry later; keep in queue.

4. Data APIs (CRUD)

For simple CRUD on master tables, Flux can use PostgREST directly. For posting documents (sale/GRN/transfer/adjustment), use RPCs to keep inventory consistent.

4.1 Locations

Table: locations. Name should be unique.

Method	Path	Purpose / Notes
GET	`/rest/v1/locations?select=*`	List locations the user can access (RLS-scoped).
POST	`/rest/v1/locations`	Create a location (Super Admin only).
PATCH	`/rest/v1/locations?id=eq.<id>`	Update location fields (Super Admin).

4.2 Categories

Core ERD fields: id, name, location_id, audit. Latest scope fields (category_no, short_code, level, comment) are optional extensions.

Method	Path	Purpose / Notes
GET	`/rest/v1/categories?location_id=eq.<location_id>&select=*`	List categories for a branch.
POST	`/rest/v1/categories`	Create category. Client should send UUID id for offline.
PATCH	`/rest/v1/categories?id=eq.<id>`	Update category fields.

4.3 Units

Core ERD fields: id, name, symbol, location_id, audit.

Method	Path	Purpose / Notes
GET	`/rest/v1/units?location_id=eq.<location_id>&select=*`	List units for a branch.
POST	`/rest/v1/units`	Create unit.
PATCH	`/rest/v1/units?id=eq.<id>`	Update unit.

4.4 Items

Core ERD fields: name, barcode, price, cost, category_id, unit_id, location_id, audit. Phase 1 needs is_batch_tracked (extension).

Method	Path	Purpose / Notes
GET	`/rest/v1/items?location_id=eq.<location_id>&select=*`	List items (used for catalog caching for offline POS).

GET	`/rest/v1/items?barcode=eq.<barcode>&location_id=eq.<location_id>&select=*`	Lookup item by barcode for scan-to-bill.
POST	`/rest/v1/items`	Create item. Barcode should be unique per location when not null.
PATCH	`/rest/v1/items?id=eq.<id>`	Update item master fields.

4.5 Suppliers

Core ERD includes contact_info as a single text field; Latest scope expects structured address/contact fields (extension).

Method	Path	Purpose / Notes
GET	`/rest/v1/suppliers?location_id=eq.<location_id>&select=*`	List suppliers for a branch.
POST	`/rest/v1/suppliers`	Create supplier.
PATCH	`/rest/v1/suppliers?id=eq.<id>`	Update supplier.

4.6 Customers

Credit sales should require customer_id at the invoice level (enforced in app + RPC).

Method	Path	Purpose / Notes
GET	`/rest/v1/customers?location_id=eq.<location_id>&select=*`	List customers for a branch.
POST	`/rest/v1/customers`	Create customer (Phase 1 supports credit_limit and credit_days).
PATCH	`/rest/v1/customers?id=eq.<id>`	Update customer.

4.7 Transaction tables (recommended read-only)

Do not POST directly to document/ledger tables in production. Use RPCs to keep stock consistent.

Method	Path	Purpose / Notes
GET	`/rest/v1/sales_invoices?location_id=eq.<location_id>&order=created_at.desc&select=*`	List invoices.
GET	`/rest/v1/sales_invoice_lines?invoice_id=eq.<invoice_id>&select=*`	Invoice lines.
GET	`/rest/v1/payments?invoice_id=eq.<invoice_id>&select=*`	Payments for an invoice.

GET	`/rest/v1/grns?location_id=eq.<location_id>&order=created_at.desc&select=*`	List GRNs.
GET	`/rest/v1/grn_lines?grn_id=eq.<grn_id>&select=*`	GRN lines.
GET	`/rest/v1/stock_transfers?location_id=eq.<location_id>&order=created_at.desc&select=*`	Transfers created by this branch (source-owned).
GET	`/rest/v1/stock_transfer_lines?transfer_id=eq.<transfer_id>&select=*`	Transfer lines.
GET	`/rest/v1/stock_adjustments?location_id=eq.<location_id>&order=created_at.desc&select=*`	Adjustments list.
GET	`/rest/v1/stock_adjustment_lines?adjustment_id=eq.<id>&select=*`	Adjustment lines.

4.8 Inventory raw reads

Prefer views (Section 6) for screen-friendly payloads.

Method	Path	Purpose / Notes
GET	`/rest/v1/stock_balances?location_id=eq.<location_id>&select=*`	Raw snapshot rows (may be lot-aware).
GET	`/rest/v1/stock_moves?location_id=eq.<location_id>&order=created_at.desc&select=*`	Ledger rows for audits.
GET	`/rest/v1/stock_lots?location_id=eq.<location_id>&select=*`	Lots/batches for Phase 1.

5. Transaction RPCs (posting documents safely)

RPCs handle multi-step business transactions and MUST execute in a single database transaction.

Replay-safe rule: Each document has a client-generated UUID. RPCs must upsert by that UUID so retries do not create duplicates.

5.1 post_sale

Creates/updates a sales invoice, lines and payments, then writes stock_moves and updates stock_balances.

Endpoint

```
POST /rest/v1/rpc/post_sale
```

Request (JSON)

```
{
  "invoice": { "id":"uuid", "invoice_number":"text", "customer_id":"uuid|null",
    "total_amount": 2450.00, "payment_status":"paid|partial|unpaid", "status":"posted",
    "location_id":"uuid", "created_by":"uuid", "created_at":"timestamptz", "updated_at":"timestamptz"
  "lines":[ { "id":"uuid", "item_id":"uuid", "lot_id":"uuid|null", "qty":2.000, "unit_price":120.00,
  "payments":[ { "id":"uuid", "method":"Cash|Card|Credit", "amount":2450.00 } ]
}
```

Response (JSON)

```
{ "ok": true, "invoice_id":"uuid", "invoice_number":"text", "posted_at":"timestamptz",
  "stock_updates":[ { "item_id":"uuid", "lot_id":"uuid|null", "new_on_hand":123.000 } ] }
```

Rules

• Upsert invoice/lines/payments by id to support retries.

• Write stock_moves (move_type='Sale') with negative quantity_change per line.

• Update stock_balances (unique by location_id + item_id + lot_id).

• For Credit payment, customer_id must be present and credit rules are enforced (app + RPC checks as needed).

• Reject if stock would go negative (recommended) and return OUT_OF_STOCK.

Example

```
POST /rest/v1/rpc/post_sale
{ "invoice": { "id":"7a2f...e21b", "invoice_number":"INV-20260131-001", "customer_id":null,
  "total_amount":2450.00, "payment_status":"paid", "status":"posted", "location_id":"b11c...9d01",
  "created_by":"0c77...f3aa", "created_at":"2026-01-31T10:15:00Z", "updated_at":"2026-01-31T10:15:00Z
  "lines":[ { "id":"a1...", "item_id":"i1...", "lot_id":null, "qty":2.000, "unit_price":120.00, "tota
  "payments":[ { "id":"p1...", "method":"Cash", "amount":2450.00 } ] }
```

5.2 post_grn

Creates/updates a GRN and its lines. Creates/attaches lots (batch/expiry) then writes stock_moves and updates balances.

Endpoint

```
POST /rest/v1/rpc/post_grn
```

Request (JSON)

```
{
  "grn": { "id":"uuid", "grn_number":"text", "supplier_id":"uuid",
    "total_amount":155000.00, "status":"posted", "location_id":"uuid",
    "created_by":"uuid", "created_at":"timestamptz", "updated_at":"timestamptz" },
  "lines":[ { "id":"uuid", "item_id":"uuid", "qty":50.000, "unit_cost":95.00, "total_cost":4750.00,
    "lot": { "id":"uuid", "batch_number":"text", "expiry_date":"date" } } ]
}
```

Response (JSON)

```
{ "ok": true, "grn_id":"uuid", "grn_number":"text", "posted_at":"timestamptz",
  "lots":[ { "lot_id":"uuid", "item_id":"uuid", "batch_number":"text", "expiry_date":"date" } ] }
```

Rules

• Replay-safe upsert by grn id and line ids.

• For batch-tracked items, lot info must be present; upsert stock_lots by lot id.

• Write stock_moves (move_type='GRN') with positive quantity_change per line.

• Update stock_balances (lot-aware).

Example

```
POST /rest/v1/rpc/post_grn
{ "grn": { "id":"g1...", "grn_number":"GRN-20260131-001", "supplier_id":"s1...",
  "total_amount":155000.00, "status":"posted", "location_id":"b11c...9d01",
  "created_by":"0c77...f3aa", "created_at":"2026-01-31T09:00:00Z", "updated_at":"2026-01-31T09:00:00Z
  "lines":[ { "id":"gl1...", "item_id":"i1...", "qty":50.000, "unit_cost":95.00, "total_cost":4750.00
    "lot": { "id":"l1...", "batch_number":"BATCH-2026-01", "expiry_date":"2027-06-30" } } ] }
```

5.3 create_transfer

Creates a stock transfer (from -> to). Recommended: on create, write TransferOut moves and reduce source stock; mark InTransit.

Endpoint

```
POST /rest/v1/rpc/create_transfer
```

Request (JSON)

```
{
  "transfer": { "id":"uuid", "transfer_number":"text",
    "from_location_id":"uuid", "to_location_id":"uuid",
    "status":"Pending|InTransit", "location_id":"uuid (same as from_location_id)",
    "created_by":"uuid", "created_at":"timestamptz", "updated_at":"timestamptz" },
  "lines":[ { "id":"uuid", "item_id":"uuid", "lot_id":"uuid|null", "qty":10.000 } ]
}
```

Response (JSON)

```
{ "ok": true, "transfer_id":"uuid", "transfer_number":"text", "status":"InTransit" }
```

Rules

• location_id must equal from_location_id for clean RLS.

• Write stock_moves (move_type='TransferOut') at source with negative quantities.

• Update source stock_balances (lot-aware).

• Do not write TransferIn until receive_transfer.

Example

```
POST /rest/v1/rpc/create_transfer
{ "transfer": { "id":"t1...", "transfer_number":"TR-20260131-001", "from_location_id":"b11c...9d01",
  "to_location_id":"c22d...1a02", "status":"InTransit", "location_id":"b11c...9d01",
  "created_by":"0c77...f3aa", "created_at":"2026-01-31T12:00:00Z", "updated_at":"2026-01-31T12:00:00Z
  "lines":[ { "id":"tl1...", "item_id":"i1...", "lot_id":null, "qty":10.000 } ] }
```

5.4 receive_transfer

Receives a transfer at destination. Writes TransferIn moves at destination, updates balances, sets status Received.

Endpoint

```
POST /rest/v1/rpc/receive_transfer
```

Request (JSON)

```
{ "transfer_id":"uuid", "received_by":"uuid", "received_at":"timestamptz" }
```

Response (JSON)

```
{ "ok": true, "transfer_id":"uuid", "status":"Received" }
```

Rules

• Only destination branch users with permission should receive (Manager/Super Admin).

• Write stock_moves (move_type='TransferIn') at destination with positive quantities.

• Update destination stock_balances (lot-aware).

• Replay-safe: if already received, return ok and do not duplicate moves.

Example

```
POST /rest/v1/rpc/receive_transfer
{ "transfer_id":"t1...", "received_by":"0c77...f3aa", "received_at":"2026-01-31T15:30:00Z" }
```

5.5 post_stock_adjustment

Posts a stock adjustment (manager-only) and updates stock_moves + stock_balances.

Endpoint

```
POST /rest/v1/rpc/post_stock_adjustment
```

Request (JSON)

```
{
  "adjustment": { "id":"uuid", "reason":"text", "status":"posted", "location_id":"uuid",
    "created_by":"uuid", "created_at":"timestamptz", "updated_at":"timestamptz" },
  "lines":[ { "id":"uuid", "item_id":"uuid", "lot_id":"uuid|null", "qty_change": -3.000 } ]
}
```

Response (JSON)

```
{ "ok": true, "adjustment_id":"uuid", "posted_at":"timestamptz" }
```

Rules

• Cashiers must not be allowed to post adjustments (Phase 0/1 rules).

• Write stock_moves (move_type='Adjustment') with quantity_change = qty_change.

• Update stock_balances (lot-aware).

• Replay-safe upsert by adjustment id and line ids.

Example

```
POST /rest/v1/rpc/post_stock_adjustment
{ "adjustment": { "id":"a1...", "reason":"Damaged items", "status":"posted", "location_id":"b11c...9d
  "created_by":"0c77...f3aa", "created_at":"2026-01-31T16:00:00Z", "updated_at":"2026-01-31T16:00:00Z
  "lines":[ { "id":"al1...", "item_id":"i1...", "lot_id":null, "qty_change":-3.000 } ] }
```

6. Views (read APIs for screens)

Views are recommended for screens so the client receives ready-to-use payloads (less client joining).

6.1 v_items_with_stock (recommended)

Recommended columns: item_id, name, barcode, price, on_hand_qty, low_stock_flag.

Method	Path	Purpose / Notes
GET	`/rest/v1/v_items_with_stock?location_id=eq.<location_id>&select=*`	Items + on_hand_qty (sum across balances).

6.2 v_lots_expiring_soon (Phase 1)

Recommended columns: lot_id, item_id, batch_number, expiry_date, days_to_expiry, on_hand_qty.

Method	Path	Purpose / Notes
GET	`/rest/v1/v_lots_expiring_soon?location_id=eq.<location_id>&select=*`	Lots nearing expiry for alerts and reports.

6.3 v_customer_outstanding (optional, Phase 1)

Optional view for credit tracking; can be added without changing table design.

Method	Path	Purpose / Notes
GET	`/rest/v1/v_customer_outstanding?location_id=eq.<location_id>&select=*`	Outstanding credit per customer (derived from invoices and payments).

7. Offline-first sync contract

Flux must work without internet for POS sales and basic data entry. The client stores masters and pending documents in IndexedDB and syncs when online.

Baseline rules used here: FIFO sync order, queue records include entity_type and payload, sync flags are client-only, and conflict policy starts as Last-Write-Wins using updated_at.

7.1 Client queue record (IndexedDB)

```
{
  "queue_id": "uuid",
  "entity_type": "sale|grn|transfer|adjustment|master",
  "entity_id": "uuid",
  "location_id": "uuid",
  "payload": { ... },
  "created_at": "timestamptz",
  "updated_at": "timestamptz",
  "sync_status": "pending|syncing|synced|failed",
  "retry_count": 0,
  "last_error": "string|null"
}
```

7.2 Sync flow

• Process queue in FIFO order (oldest first).

• Masters: upsert by primary key UUID using PostgREST.

• Documents: call RPCs (post_sale, post_grn, create_transfer, receive_transfer, post_stock_adjustment).

• On network errors: mark failed and retry with backoff.

• On conflict: refresh server state and apply updated_at-based last-write-wins when reasonable.

7.3 Document numbering (practical recommendation)

Because offline devices can create documents at the same time, document numbers must avoid collisions. Recommended format: PREFIX-YYYYMMDD-DEVICESEQ (example: INV-20260131-001). Keep a unique constraint per location on invoice_number/grn_number/transfer_number.

8. Security model (RLS + roles)

Flux uses Supabase Auth for identity and Postgres RLS for branch isolation. App roles control sensitive actions.

8.1 RLS policy rule (location isolation)

For any table with location_id, policies allow access only when row.location_id matches a location the user is assigned to.

8.2 Role rules (Phase 0/1)

• Cashier: sales create/post, read masters and stock views; cannot post stock adjustments; cannot manage users.

• Store Manager: cashier permissions + GRN, transfers, adjustments for their branch.

• Super Admin/Owner: cross-branch access, location and user management.

8.3 Enforcement points

• RLS policies restrict CRUD by location_id.

• RPCs enforce additional role checks and status rules because they change stock.

• Block direct client writes to stock_moves; only RPCs write ledger rows.

9. Appendix

9.1 Core ERD table names (Phase 0 + 1)

categories, customers, grn_lines, grns, items, locations, payments, sales_invoice_lines, sales_invoices, stock_adjustment_lines, stock_adjustments, stock_balances, stock_lots, stock_moves, stock_transfer_lines, stock_transfers, suppliers, units

9.2 Suggested standard error body

```
{
  "ok": false,
  "error_code": "VALIDATION_ERROR | RLS_FORBIDDEN | UNIQUE_CONFLICT | OUT_OF_STOCK | ALREADY_RECEIVED
  "message": "Human-friendly message",
  "details": { "field": "reason" }
}
```