1) Phase 0 Sprint Plan (3–5 sprints → 4 sprints)

Sprint	Name	Sprint goal	Deliverables (demoable)	Demo scenario (end of sprint)
1	**Foundation: PWA + Auth**	App installs, loads offline shell, and login/session is stable	PWA shell, service worker cache, login/logout, protected routes, basic layout with status bar	Install app → open offline → see login screen → go online → login → land on Home
2	**Branch Isolation: location_id + RLS proof**	RLS is **proven**: Branch A cannot read/write Branch B	DB tables for Phase 0 (locations/items/categories/units/sales*), RLS policies, location selection, role guard basics, RLS test scripts/checks	Create 2 locations + 2 users → login as A → cannot view/insert B data (UI + direct API)
3	**Offline POS: local sale + queue**	Cash sale works fully offline and is durable	POS screen, local catalog cache (IndexedDB), create invoice+lines locally, outbox queue record, status badges (Pending/Failed/Synced)	Go offline → search item from cache → add to cart → post sale → shows Pending Sync after refresh
4	**Sync: push sale + recover from failures**	Auto sync pushes queued sales safely (retry-safe, no duplicates), shows clear sync state	Sync engine (pull masters + push FIFO), retry/backoff (basic), server RPC for sale posting (atomic), failure UI, simple sales list	Post 2 sales offline → reconnect → auto sync FIFO → see Synced → simulate failure → shows Failed → Retry works

*Sales tables = sales_invoices, sales_invoice_lines (and minimal payments if you keep it). The "atomic post" should go through an RPC for safety.

---

2) Sprint-by-sprint detail

Sprint 1 — Foundation: PWA + Auth

Goal: Prove the **app shell**, **service worker caching**, and **Supabase Auth** basics.

User stories

- **S1-US1:** As a user, I can install Flux as a PWA and open it.
- **S1-US2:** As a user, I can log in, stay logged in on refresh, and log out.
- **S1-US3:** As a user, I cannot access business screens unless logged in.

Task breakdown

Frontend

- Next.js App Router structure: /login, /app/* protected routes
- App shell layout: header + left menu placeholder
- Online/Offline indicator (simple: browser online event)
- "Pending count" placeholder badge (0 for now)

Backend (Supabase)

- Create Supabase project + env setup
- Enable Auth (email/password)
- Create minimal user_profiles table (or equivalent) to store role + location assignment (even if Phase 0 uses 1 user)

Offline/Sync

- Register service worker and cache shell routes/assets
- Add IndexedDB setup (empty stores created)

QA

- Test installability + offline load to login screen
- Test protected routing redirects
- Test logout clears session

Dependencies

- Supabase project created + keys in env
- Basic hosting/dev HTTPS setup (PWA works better with HTTPS)

Acceptance criteria (clear checks)

- App can be installed and opens in standalone mode (supported browser).
- With network disabled, app still loads to **login screen** (no blank page).
- User can login/logout; refresh keeps session; protected routes require auth.

Sprint 2 — Branch Isolation: location_id + RLS proof

Goal: Prove **multi-branch isolation** is real (not just UI filtering).

User stories

- **S2-US1:** As a user, after login I select an active location and the app stores it locally.
- **S2-US2:** As a developer, I enforce RLS so Branch A cannot read/write Branch B rows.
- **S2-US3:** As a manager/admin, I can CRUD minimal master data for my own branch (online for now).

Task breakdown

Backend (Supabase DB)

- Create Phase 0 tables (minimum set):
  - locations (may be global list; but still controlled by RLS rules)
  - categories, units, items (all include location_id, UUID PK, audit fields)
  - sales_invoices, sales_invoice_lines (include location_id, UUID PK, audit fields)
- Add **uniqueness constraints** that prevent retry duplicates:
  - items: unique (location_id, barcode) where barcode not null
  - sales_invoices: unique (location_id, invoice_number) (or a safe unique doc key)
- Implement RLS policies:
  - Manager/Cashier: can read/write only rows where location_id = assigned_location_id
  - Super Admin (later): cross-branch (Phase 0 can skip, but keep policy-ready)

Frontend

- Location select screen after login (only show allowed locations)
- Store active_location_id in IndexedDB (or local storage if you must, but IndexedDB preferred for consistency)
- Update all queries to always filter by location_id = active_location_id

Offline/Sync

- Persist active location + profile in IndexedDB
- Add a simple “Sync Now” button placeholder (no real sync yet)

QA

- RLS verification with 2 locations + 2 users:
  - User A cannot list User B’s items/categories
  - User A cannot insert a row using location_id=B (server rejects)

Dependencies

- Sprint 1 auth working
- user_profiles has assigned location_id for each user (or JWT claim strategy)

Acceptance criteria (clear checks)

- After login, user must select (or auto-load) an active location; UI only shows that location’s data.
- **Hard proof:** Using the client/API, Branch A cannot read/write Branch B rows (403/empty due to RLS).
- Items barcode uniqueness is enforced per location (server constraint).

Sprint 3 — Offline POS: local sale + queue

Goal: Create a sale fully offline (invoice + lines saved locally) and show clear statuses.

User stories

- **S3-US1:** As a cashier, I can search items offline from cached catalog.
- **S3-US2:** As a cashier, I can create a sale offline (cart → totals → post) and it is saved locally.
- **S3-US3:** As a user, I see Online/Offline and Pending Sync count + per-transaction status.

Task breakdown

Frontend

- POS screen (minimal but real):

    - item search (barcode/text) from local cache
    - cart lines, qty edit, totals
    - “Post Sale (Cash)” creates a **local posted invoice**
- Sales list screen (local-first):
    - shows invoices from local store with status badge

Offline/Sync (IndexedDB)

- Create IndexedDB stores (minimum):
    - catalog_items (cached master items)
    - local_sales (invoice header + lines)
    - outbox (sync queue)
    - sync_meta (last pull timestamps)
- Outbox item format (must include):
    - entity_type, entity_id, location_id, payload, retry_count, last_error, status, timestamps
- Status rules:
    - When posted offline → Pending
    - On restart → still pending (durable)

Backend

- Provide a simple “pull items” query (for online caching)
- No posting to server yet (that is Sprint 4)

QA

- Offline test:
    - Disable network → post sale → refresh tab → sale still exists and shows Pending
- Validation:
    - qty > 0, money with 2 decimals, required fields for posting

Dependencies

- Sprint 2: Items exist and can be pulled/cached
- POS needs at least one item in catalog cache (initial sync once online)

Acceptance criteria (clear checks)

- With network off, user can create a cash sale and it is stored in IndexedDB.
- Pending sale survives refresh/reopen and remains in outbox queue.
- UI always shows Online/Offline + Pending count; sales list shows Pending badge.

Sprint 4 — Sync: push sale + recover from failures

Goal: Sync works: push queued sales to Supabase safely, FIFO, retry-safe, clear status.

User stories

- **S4-US1:** As a user, when online returns, pending sales sync automatically in FIFO order.
- **S4-US2:** As a user, sync failures do not lose data; failed items show error and can be retried.
- **S4-US3:** As a developer, retries do not create duplicates (idempotent).

Task breakdown

Backend (critical)

- Implement post_sale RPC (atomic transaction):
    - Upsert invoice + lines (and minimal payment row if used)
    - Enforce RLS + location_id
    - Update inventory model if you include it in Phase 0 proof (recommended minimal):
        - Insert stock_moves and update stock_balances (or at least show sale persisted and later you add stock in Phase 1—**but your Phase 0 exit says stock should update**, so do the minimum now)
- Make RPC replay-safe:

- Same UUID → treated as already posted (no double insert)

Sync engine (client)

- Implement sync loop:
  1. **Pull** masters first (items) using updated_at > last_sync_at
  2. **Push** outbox FIFO: next pending item → call RPC
  3. On success: mark Synced + update local sale status
  4. On failure:
     - 400/403 = permanent → mark Failed + show error (no auto retry)
     - 500/network = retry later → increment retry_count and schedule next_retry_at (simple backoff)

Frontend

- Sync Queue screen:
  - list outbox items with status + last error + "Retry now"
- Sales list:
  - show Pending/Synced/Failed, filter by status

QA

- Failure simulation:
  - network drop mid-sync
  - RPC returns 409 conflict (duplicate invoice_number) → verify retry-safe handling
  - RLS reject (user tries wrong location_id) → verify permanent fail

Dependencies

- Sprint 3 outbox format exists
- Sprint 2 RLS policies already enforced
- RPC + DB constraints ready

Acceptance criteria (clear checks)

- Create 2 sales offline → reconnect → sales sync automatically **in order** and appear on server.
- If sync fails (network/server), item stays in outbox and becomes Failed/Retrying with message; manual Retry works.
- Retrying the same queued sale does **not** create duplicates in Supabase.
- UI always shows Online/Offline + pending count; each sale shows status.

---

3) Testing Plan for Phase 0 (very practical)

A) How to verify RLS isolation (must be "hard proof")

Setup

- Create **Location A** and **Location B**
- Create **User A** assigned to Location A, and **User B** assigned to Location B

Tests (do both UI + API level)

1. **Read test**
   - Login as User A → open Items list → must show only Location A items
   - Try a direct fetch of /items?location_id=eq.<LocationB> → should return empty or forbidden (depending on policy)
2. **Write test**
   - Login as User A → try creating an item but force payload location_id=LocationB
   - Expected: **403 Forbidden** (RLS) and item not created
3. **Cross-check**
   - Login as User B → confirm User A's test item does not appear

Pass condition

- No cross-branch read/write is possible even if someone tampers the request payload.

B) How to test offline mode + sync reliability

Offline POS

- Open app online once → confirm items cached
- Disable network (DevTools → Network → Offline)
- Create sale → post → confirm:
  - sale saved in IndexedDB
  - outbox has 1 queue item
  - UI shows Pending count + “Pending” badge
- Refresh tab / close & reopen → still pending

Sync

- Re-enable network
- Confirm sync triggers automatically:
  - outbox item goes Pending → Syncing → Synced
  - sale appears in server sales list (and optionally stock updates)

C) Failure cases you must test (Phase 0)

1. **Network drop mid-sync**
   - Start sync, then switch offline during the RPC call
   - Expected: queue item remains Pending/Failed, not lost; retry later succeeds
2. **Duplicate retry**
   - Force same outbox item to send twice (simulate double-click / app restart)
   - Expected: server has **one** invoice for that UUID; client ends Synced
3. **Partial sync prevention**
   - Ensure you never create “header saved but lines missing”
   - Pass by using a **single RPC transaction** for posting sale
4. **RLS rejection**
   - Corrupt an outbox payload with wrong location_id
   - Expected: server rejects; client marks item Failed with clear “not allowed” error

---

4) Definition of Done (DoD) for completing Phase 0

Phase 0 is complete only when all points below are true on at least **one real device + one browser**:

PWA + Shell

- App installs as PWA and launches in standalone mode.
- With no internet, app opens to login screen and core shell routes (no blank page).

Auth

- Login/logout/session refresh works.
- All business screens are blocked when logged out.

RLS Branch Isolation (non-negotiable)

- Two locations exist; User A cannot read/write User B’s data through UI or API, even with request tampering.

Offline POS (non-negotiable)

- Cash sale can be created and posted fully offline using cached items.
- Sale is stored durably in IndexedDB; closing/reopening does not lose it.

Sync (non-negotiable)

- When online returns, queued sales sync automatically FIFO.
- Failures do not lose data; failed items show error and can be retried.
- Retries do not create duplicates (UUID + constraints + idempotent RPC).

Basic POS end-to-end

- Items → cart → totals → invoice saved locally → invoice synced to server → sales list shows it as Synced.
