TECHNICAL ARCHITECTURE · HOOSIER OSTEOTRONIX

# SurgiSync Inventory Module

Studio Raine · John Raine · Architecture Reference

> This document describes how the SurgiSync Inventory Module is built. It names the specific frameworks, libraries, and patterns used in each area so the technical depth is visible on the page rather than left implied. The module extends the existing SurgiScribe production stack — no new infrastructure is introduced.

**SCOPE OF THIS ENGAGEMENT**

This proposal covers a focused, robust inventory module for loading, tracking, managing, and decrementing inventory across role-based entity accounts. Features outside the core inventory workflow — including expiration alerting, FIFO logic, confinement agreement tracking, and payment processing — are intentionally excluded from this phase. The loaner lifecycle is scoped to inventory custody only, with an explicit bridge point to the Ordering module.

**EXISTING STACK — EXTENDED, NOT REPLACED**

`Django` `Django REST Framework` `PostgreSQL on AWS RDS` `Flutter` `GetX` `Dio`
`AWS S3` `AWS SQS` `Brother Bluetooth SDK`

---

3.1 WORK AREA ONE

## Architecture & Foundation

The database schema, API surface, and audit infrastructure that every downstream feature depends on. Built first, hardened first, never reopened.

### HOW IT'S BUILT

- Schema delivered through `Django migrations`. New tables follow the column conventions already established in SurgiScribe so the codebase stays uniform.
- Endpoints are `DRF ViewSets` with `ModelSerializers`, paginated through `LimitOffsetPagination`, and version-namespaced under `/api/v1/inventory/`.
- The audit log lives in a dedicated `PostgreSQL` table with `UPDATE` and `DELETE` privileges revoked at the database role level. Even a buggy ORM call cannot violate append-only semantics.
- Every write endpoint accepts an `Idempotency-Key` HTTP header. Keys are cached in `Redis` with a 24-hour TTL; replays return the cached response with no reprocessing.

- Inventory type is a required field on every inventory record — every item must be classified as **loaned**, **consigned**, or **owned** at creation. The schema enforces this at the database constraint level.

---

3.2 WORK AREA TWO

# Role-Based Access & Entity Management

A multi-tenant permission model with four user tiers and absolute entity-level data isolation. No user can see or touch inventory that belongs to a different entity — by construction, not by convention.

### THE FOUR ROLES

**PLATFORM ADMIN**

God view. Full access across all entities. Manages the platform itself — entity creation, global config, system-level audit access.

**ENTITY ADMIN**

Full access within their own entity (parent company). Creates and manages managers and reps. Configures manufacturer policies for their entity.

**MANAGER**

Sees everything their direct reports can see. Inventory rollup across their team. Cannot see other managers' reps or other entities.

**REP**

Sees only their own inventory, cases, and loaner status. Field-first interface. No visibility into other reps' records or other entities.

### HOW IT'S BUILT

- Permissions enforced through four `DRF permission classes`: `IsPlatformAdmin`, `IsEntityAdmin`, `IsManager`, and `HasOrganization`. Each ViewSet declares the classes it requires; no endpoint ships without them.
- Entity isolation is structural, not procedural. A custom `OrganizationScopedManager` overrides `get_queryset()` to automatically inject the caller's entity filter on every read. A permission hole cannot be introduced by forgetting to add a filter — the filter is never written by hand.
- Multi-entity membership modeled through `OrganizationMembership(user, entity, role, valid_from, valid_to)`. One user can belong to multiple entities with a different role per entity, and historical role state is always queryable by date.
- Hierarchical visibility uses a `PostgreSQL recursive CTE` — a manager's dashboard aggregates all inventory records reachable through their reporting chain, not just their own.
- Test coverage is a parameterized matrix in `pytest`: every endpoint × every role × cross-entity wrong-user scenarios, written before the endpoints ship. Every permission hole gets caught before it reaches staging.

---

3.3 WORK AREA THREE

# Entity Configuration

The entity hierarchy — parent companies, facilities, manufacturers — that the inventory module references for ownership classification, access control, and loaner policy association.

**HOW IT'S BUILT**

- Three core Django models: `ParentCompany` (with `OrganizationType` enum — Distributor or OEM), `Facility` (with type, GPO/IDN affiliation, accreditation status), and `Manufacturer`.
- Each `Manufacturer` record carries a link to its loaner policy configuration — see section 3.4 for how those policies are structured and enforced.
- Admin surfaces auto-generated through `Django ModelAdmin` for all three models. Entity Admins manage their own entity's records; Platform Admins manage everything.
- Login restriction enforced at the authentication layer: only `ParentCompany` records of type Distributor or OEM can authenticate. Facilities exist as data only.

# Inventory Module — Mobile & Web

The primary deliverable. Five inventory site types, full intake workflow across three input modes, on-hand management, transfers, loaner lifecycle (custody scope), return to OEM with carrier integration, inventory decrement after use, and a full audit trail. Each capability follows the same pattern — Django model, service class, DRF ViewSet, Flutter screen, test suite.

### INVENTORY TYPE CLASSIFICATION

Every inventory item is one of three types. This is enforced at intake — a rep cannot load inventory without selecting the type. The type drives the downstream workflow, policy checks, and reporting.

**LOANED**
Manufacturer-owned. Borrowed for a specific period. Governed by the manufacturer's loaner policy. Triggers loaner lifecycle on load.

**CONSIGNED**
Manufacturer-owned. Held by the distributor indefinitely. No time-based policy enforcement. Used and decremented as surgeries occur.

**OWNED**
Distributor-owned outright. No manufacturer custody rules apply. Full control by the entity that loaded it.

### MANUFACTURER-SPECIFIC LOANER POLICY ENGINE

Every manufacturer has its own loaner policy — today stored informally in emails, spreadsheets, or memory. SurgiSync standardizes and enforces them automatically.

- `ManufacturerLoanerPolicy` model linked to each `Manufacturer` record, storing: days allowed before late, daily late fee amount, return condition requirements, and any entity-specific overrides.
- When a rep selects a manufacturer during a loaned inventory load, the system auto-associates the policy. The rep sees only the fields relevant to that manufacturer's rules — no manual lookup required.
- Policy enforcement runs server-side through a `LoanerPolicyService`. The policy is enforced in the backend and reflected in the UI where relevant — reps are not shown a browseable policy tab. If a dispute arises, the rep handles it with the manufacturer directly.
- Policy rules are managed by Entity Admins in the admin interface. Updates take effect on new loaner loads immediately.

### INVENTORY INTAKE — THREE MODES

Reps can load inventory three ways depending on volume and context. All three modes require inventory type selection and produce the same structured record.

- **Manual single-item entry:** Flutter multi-step wizard — entity → manufacturer → location → kit ID/SKU → inventory type → photo → notes → confirm. State held in `GetX` controllers. Draft persisted via `shared_preferences` so backgrounding the app does not lose progress.
- **Kit loading:** Receive a pre-built tray as a unit. Same wizard flow with kit-level metadata. All items in the kit inherit the kit's type classification unless individually overridden.
- **Bulk upload:** Excel/CSV via downloadable templates. Server-side parsing with `openpyxl`, row-by-row `transaction.savepoint()` so bad rows fail individually without blocking valid rows. Packing slip photo required per upload batch.
- **Barcode/scan intake:** Continuous scanning via `mobile_scanner` (GS1 + HIBC formats). Session-level parameters (manufacturer, location, type, lot) set once, then scanned items inherit them.

Duplicate detection via in-memory `Set<String>` plus a pre-commit server-side `bulk_check`. Batches commit through a single `POST /inventory/bulk-receive` wrapped in `transaction.atomic()`.

- **Photo capture:** Required at intake on every mode. Camera via Flutter's `camera` package. Photos upload to S3 via presigned URLs; the S3 key is a non-nullable foreign key on the receive record — the database refuses the insert without it.

#### ON-HAND MANAGEMENT

Once loaded, inventory moves into the on-hand view — the rep's primary working surface.

- Search by kit name, manufacturer, location, SKU, or lot number.
- Filter by inventory type (loaned / consigned / owned), site, manufacturer, and status.
- Sort by name, manufacturer, and date received.
- Kit detail screen with full history timeline, all photos, and every status transition logged.
- Status system enforced server-side: every status change writes the status row, the audit entry, and the S3 photo reference inside a single `transaction.atomic()` block.

#### TRANSFER WORKFLOW

`IN-TRANSIT` is a row in the `InventorySite` table carrying `from_site_id` and `to_site_id` foreign keys. State transitions handled by `TransferService`. Auto-confirm triggers when a kit's location update matches its destination site. Supports MAIN→REP, REP→FACILITY, REP→REP, and REP→MAIN site flows.

#### RETURN TO OEM & UPS/FEDEX INTEGRATION

Adapter pattern in `shipping/providers/`. Abstract `ShippingProvider` base class implemented by `UPSProvider`; FedEx drops in later without changing call sites. UPS REST API called via `httpx` async client. OAuth tokens cached in `Redis` with a five-minute pre-expiry buffer and lazy refresh on first 401. Bulk multi-OEM returns handled by `BulkReturnService` — groups kits by `manufacturer_id`, fires parallel UPS calls via `asyncio.gather()`, emits one prepaid label per OEM. Label PDFs stored in S3 and emailed to the rep via `AWS SES`.

#### LOANER INVENTORY HANDLING & BRIDGE TO ORDERING

The loaner lifecycle in this module covers the inventory custody portion only — from load through return. The request and approval flow is the explicit bridge point to the Ordering module.

- `LoanerRequest` model with status enum: Requested → Approved → Shipped → In Use → Returned. Each transition creates the appropriate `InventorySite` record through `TransferService`.
- When a loaner is loaded, the manufacturer's policy is auto-applied via `LoanerPolicyService`. Expected return date is calculated and stored on the record.
- The `LoanerRequest` record is designed as the handoff point: when the Ordering module ships, it reads and writes this same record. No migration or data model change required at that time.
- Return condition captured at return (good / damaged). Return photo required before the return record commits.

#### INVENTORY DECREMENT AFTER USE

When inventory is used in surgery, the system decrements it automatically — no manual update required from the rep.

- A `UsageEvent` record is created when inventory is associated with a completed surgical case. The event references the rep, the inventory item, and the case.
- Decrement logic runs inside `transaction.atomic()` — the usage event and the inventory quantity update commit together or neither commits.
- Used inventory transitions to a terminal state in the status system and is no longer shown as available on-hand.
- All decrement events are written to the audit log, providing a complete traceability chain from intake to use.

#### GROUP & BULK ACTIONS

Multi-kit selection on the on-hand list driven by a `GetX` selection controller. Bulk transfer, bulk status update, and bulk return to OEM all available. Group segregation rules enforced server-side. Skipped-kit reasons returned as structured response payloads consumed by the Flutter UI.

#### BROTHER BLUETOOTH PRINTER INTEGRATION

Extends the existing SurgiScribe Brother integration through the Brother iPrint&Label SDK. Print queue with offline support via a local queue table in `sqflite`, drained when the printer reconnects.

#### REP INVENTORY DASHBOARD

Intentionally minimal. The dashboard gives reps exactly what they need to act — not a full analytics surface.

- Overdue loaners — any loaner past its expected return date, surfaced immediately on open.
- Late fees owed — calculated from the manufacturer's loaner policy against each overdue record.
- Basic inventory status indicators — on-hand count by site, any items flagged for attention.
- Single endpoint: `GET /inventory/dashboard/`. Flutter renders cards through the existing SurgiScribe shared widget library. Tap-throughs deep-link into filtered on-hand views.

#### AUDIT TRAIL

Every action — load, transfer, status change, return, decrement — writes a permanent, uneditable record. This is a compliance and accountability feature, not an afterthought.

- Single `AuditLog` table indexed on `(model_name, record_id, timestamp)` for sub-100ms history queries. `UPDATE` and `DELETE` privileges revoked at the database role level.
- Every service method calls `log_audit()` inside the same `transaction.atomic()` block as the action it records. The log entry and the action commit together or neither does.
- Photos are part of the audit trail. Every photo is stored in S3 with EXIF preserved — server timestamp, S3 `LastModified`, and embedded EXIF provide three independent witnesses on every action.
- Audit history visible from any kit detail screen. Managers see their team's audit history. Entity Admins see their full entity. Platform Admins see everything.

#### MOBILE-FIRST INTERFACE

The interface is designed for field use — simple, fast, and forgiving of the conditions reps actually work in.

- Mobile-first Flutter layout. Every screen is designed for one-handed use in a hospital corridor before it is adapted for desktop.
- Desktop mirrors mobile. The Flutter Web build shares the same Dart codebase. What works on the phone works on the laptop — no separate frontend to maintain.
- Scan and photo workflows optimized for speed — continuous scan mode, minimal taps between actions, full-width camera in landscape.
- Responsive breakpoints via `LayoutBuilder` and `MediaQuery`. Tablet landscape activates two-column layouts and master/detail views.

#### OFFLINE SUPPORT

Offline capability is included as a reliability layer, not a headline feature. Photos write to the app documents directory immediately. S3 upload queue managed by `workmanager` on Android and `BGTaskScheduler` on iOS, with exponential backoff retry. Pending upload count exposed as a `GetX` observable on the home screen so reps know what's still queued.

---

3.5 WORK AREA FIVE

## Integration Testing & Deployment

#### HOW IT'S BUILT

- Backend tests through `pytest` and `pytest-django`, with fixtures generated by `factory_boy`. Unit tests target permission classes, status rules, loaner policy enforcement, and decrement logic.
- The RBAC matrix — every endpoint × every role × cross-entity wrong-user scenarios — written as parameterized `pytest` tests, executed in CI before any merge. Permission holes get caught before they reach staging.
- Flutter end-to-end coverage through the `patrol` package on the critical flows: receive (all three modes), transfer, return to OEM, decrement, bulk update.
- UPS sandbox interactions replayed through `pytest-recording` cassettes — label generation, OEM account routing, error responses, and idempotency replays.
- Deployment through the existing `GitHub Actions` CI pipeline to the current AWS environment. Zero-downtime migration patterns. Post-deploy smoke tests verify audit log writes, dashboard endpoint, and decrement behavior.

---

3.6 WORK AREA SIX

## Project Coordination

#### HOW IT'S RUN

- Weekly demos against a working branch deployed to staging. Recorded walkthroughs via `Loom` for async review.
- Sprint planning every two weeks, with task tracking in `Linear` or the issue tracker SurgiScribe already uses.

- Blockers surfaced in writing within 24 hours — no decisions held hostage by calendar availability.
- Code review on every merge, with the validation trail visible to the client.

THREE PRINCIPLES THAT RUN THROUGH THE BUILD

1. **Invariants enforced at the PostgreSQL and Django model layer**, never at the Flutter UI. Permission rules, inventory type requirements, and policy enforcement cannot be bypassed by a bug above the data layer.
2. **Every state change is one `transaction.atomic()` block** that includes its audit row and any photo reference. Partial states are impossible by construction.
3. **Every external integration sits behind an adapter class.** When FedEx, a different printer, or a new ERP arrives in a later module, no code in this engagement needs to be rewritten.