Paloma Asset Management — Wiring Up Guide

For: Future Functional Wiring, Maintenance, and Feature Expansion

---

# Table of Contents

1. 1. File Directory & Roles

* 2. High-Detail Wiring Logic for Each File
* - Component/Handler Function
* - Key Methods, Patterns, Data Flow
* - Principles for Expansion

---

# 1. File Directory & Component Roles

|  |  |  |
| --- | --- | --- |
| File Name | Role / Type | Brief Description |
| AddAssetModal.jsx | Modal Component | UI for adding new asset, collects form data, triggers save |
| TransferModal.jsx | Modal Component | UI for bulk transferring asset locations, collects location input |
| ModalsContainer.jsx | Modal Router | Central logic for all modals (add, edit, transfer, QR, delete) |
| FlyHQ.jsx | Feature Page (React) | Asset HQ page: Table, filters, selection, CRUD logic, modals |
| index.js | Express API Server | All backend REST endpoints: Asset API, job planner, activity |
| assets.js | DB Handler/ORM | PostgreSQL asset logic: get, add, update, delete, transfer |

---

# 2. High-Detail Wiring Logic

## A. AddAssetModal.jsx

Type: Modal UI Component

Purpose: Presents form to collect asset data (PPC#, SN, name, category, location, status). Handles validation, then triggers onSave(payload) with form values.

* Key Wiring Patterns:
* Controlled State: Each input is useState-controlled. Changing values updates local state immediately.
* Validation: On save, all required fields are checked. If missing, alert() is used.
* Output Format: onSave is called with a well-structured object ({ id, sn, name, category, location, status }).
* ID Construction: The id field is always built as PPC <number>, enforcing a consistent database key.
* Reset: On successful save, all fields are cleared to defaults.

Expansion Principle:

* All modal submitters should validate, build a DB-aligned payload, and pass upward using onSave. Keep form state strictly internal, but push save/cancel decisions to the parent.

## B. TransferModal.jsx

Type: Modal UI Component

Purpose: Allows user to bulk-transfer asset(s) to a new location. Accepts selected asset IDs, outputs new location for transfer.

* Key Wiring Patterns:
* Local Shadow State: Location is managed locally (localLocation), synced from parent prop when modal opens.
* Bulk Selection: Renders all selected IDs for review. Parent passes selectedIds array.
* Upstream Events: onLocationChange(newLoc) — used for real-time sync with parent. onTransfer() — fires when ready, parent handles DB/API.
* Form Logic: Save only enabled if at least one asset selected and location is chosen.

Expansion Principle:

* Whenever you add bulk actions, let the parent own the master state; use modal-local state for inputs, sync up on open/close, and trigger changes upwards with callbacks.

## C. ModalsContainer.jsx

Type: Modal Router (React Pattern)

Purpose: Centralizes all modal logic, mounts/unmounts each modal component as needed. Passes handlers and data from parent (FlyHQ).

* Key Wiring Patterns:
* Visibility Props: e.g. showAddModal, setShowAddModal, showTransferModal, etc.
* Action Callbacks: e.g. onAddAsset, onEditAsset, onDeleteConfirm, onTransfer
* Data Propagation: Passes option lists, asset data, and selected IDs to relevant modals.
* State Ownership: Parent (FlyHQ) owns all modal open/close and data state; ModalsContainer just coordinates.

Expansion Principle:

* When adding new modals (or bulk actions), always update ModalsContainer to centralize their show/hide logic, and manage all parent/child state flow in one place.

## D. FlyHQ.jsx

Type: Main Feature Page (Asset Manager)

Purpose: Master controller for the asset management dashboard: Loads assets, activity logs, controls filters, sorting, search, pagination. Owns all CRUD logic, selection, and modal state. Triggers API requests for add/edit/delete/transfer.

* Key Wiring Patterns:
* Hooks: Uses custom hooks (useAssets, useActivityLog, useFilteredPaginated) for data-fetching, filtering, and log management.
* Modal State: Holds all modal visibility and working asset data (add, edit, delete, transfer).
* CRUD Methods: Add: Calls fetch('POST', /api/assets) with new asset payload. Edit: Calls fetch('PUT', /api/assets/:id) with updated fields. Delete: Calls fetch('DELETE', /api/assets/:id). Transfer: Calls fetch('POST', /api/assets/transfer) with { assetIds, newLocation }.
* Table Selection: Handles single and multi-select (with selectedAssetIds), provides those IDs to modals/actions.
* Option Generation: Dynamically builds options for form selects by extracting unique values from current assets.

Expansion Principle:

* Always wire new CRUD actions to align with DB schema. Make sure all state/data for modals lives in FlyHQ, and pass handlers down to modals via ModalsContainer. Any new feature requiring asset data should use/extend the hooks provided.

## E. index.js

Type: Express API Server (Backend Router)

Purpose: Exposes RESTful API endpoints for all core operations: asset CRUD, job planner, activity logs, uploads, authentication, etc.

* Key Wiring Patterns:
* Modular Routing: Each major resource is its own handler (assets, jobs, activity, sourcing, etc.), each endpoint is mapped to an async function.
* Asset Endpoints: GET /api/assets — fetch all, POST /api/assets — create new, PUT /api/assets/:id — update, DELETE /api/assets/:id — delete, POST /api/assets/transfer — bulk transfer.
* Handler Wiring: All routes delegate actual DB logic to assets.js methods.
* Activity Logging: On every CRUD, addActivityLog is called for audit trails.
* Session & Auth: Uses passport/Discord for authentication.
* Consistent Response Format: Always returns either { success: true } or { error: <message> }.

Expansion Principle:

* Whenever adding new resource types, always make a handler in this pattern: 1. Validate input 2. Delegate to DB handler (like assets.js) 3. Log activity if applicable 4. Return { success } or { error }

## F. assets.js

Type: Database Handler/ORM for Assets (PostgreSQL)

Purpose: Owns all direct DB operations for the assets table: query, insert, update, delete, transfer, and logs.

* Key Wiring Patterns:
* Pooling: Uses a shared pg.Pool for efficient DB connections.
* Method-per-Action: getAllAssets(), addAsset(asset, updatedBy), updateAsset(assetId, changes, updatedBy), deleteAsset(assetId, updatedBy), transferMultipleAssets(assetIds, newLocation, updatedBy).
* Safe Updates: Only allows column changes for approved fields.
* Bulk Updates: transferMultipleAssets uses WHERE id = ANY($3::text[]) for robust multi-asset updates.
* Activity Log: Dedicated methods for adding/fetching log entries.

Expansion Principle:

* When adding a new DB action, write a new function here with parameterized queries, then expose it in module.exports. Update the API server to call it.

## General Wiring Up Patterns for New Features

- Front-end (React):

- Keep all modal state and CRUD handlers at the feature page (e.g. FlyHQ).

- Pass handlers/data to modals via a container component.

- Validate all form data before submitting to backend.

- Always align API payloads with DB schema; remove empty/undefined fields before sending.

- Use effect hooks to keep UI in sync after DB changes (re-fetch assets/logs on add/edit/delete/transfer).

- Backend (Express/Node):

- Modularize each resource (assets, jobs, activity, etc.).

- Use async/await and try/catch in all endpoints.

- Log all mutating actions to an activity log for full traceability.

- When transferring/bulk-updating, validate inputs and check for existence before DB action.

- Always return a JSON result with { success } or { error }.

- Database (Postgres):

- Parameterize all queries to avoid SQL injection.

- Restrict updates to only allowed fields.

- For bulk actions, prefer WHERE ... = ANY($N::type[]) for arrays.

## Template for Adding a New Modal or CRUD Feature

1. Front-end Modal:

- Build form as a controlled React component.

- Validate all fields before save.

- On save, trigger parent handler with clean payload.

2. FlyHQ Handler:

- Write async handler: prepares API payload, sends request, updates local state, re-fetches data/logs.

3. API Endpoint:

- Add Express route for new action.

- Validate body, delegate to assets.js (or relevant DB file).

- Log action in activity log.

4. DB Handler:

- Add function in assets.js to handle new DB logic.

- Export it and call from API route.

## Summary Table: How Data Flows When Adding or Updating an Asset

|  |  |  |  |
| --- | --- | --- | --- |
| Layer | Action | File | Method / Function |
| User | Opens modal, fills out form, clicks Save | AddAssetModal.jsx | handleSubmit → onSave(payload) |
| UI Parent | Handles onAddAsset(payload) | FlyHQ.jsx | handleAddAsset (async) |
| API Call | Sends POST/PUT request to server | FlyHQ.jsx | fetch("POST", /api/assets) |
| Backend | Receives, validates, logs, updates DB | index.js | app.post("/api/assets") |
| DB Layer | Executes parameterized SQL on assets | assets.js | addAsset(asset, updatedBy) |
| Activity | Logs action to activity log | assets.js | addActivityLog(...) |
| Response | Sends success/failure to frontend | index.js | JSON { success } / { error } |
| UI Update | Fetches new data, updates table/log | FlyHQ.jsx | fetchAssets(), fetchActivityLogs() |

---

1. Q1: How do I add a new bulk action for assets that needs user input and DB changes?
2. Q2: What steps should I follow to safely update the assets schema and all wiring up the stack?
3. Q3: How does activity logging ensure traceability and how should it be maintained for all asset actions?