Internal Admin Module Specification: Herethere Loyalty App

Contents

1 Purpose of the Internal Admin Module

The internal admin module serves as a centralized dashboard for the developer team to:

- Monitor and manage all merchants using Herethere Loyalty.
- View aggregated platform analytics (e.g., total merchants, orders processed, points issued).
- Perform administrative actions (e.g., suspend accounts, adjust points, manage plans).
- Troubleshoot issues (e.g., view API logs, audit logs).
- Support internal operations without relying on the Shopify admin panel.

This module is distinct from the merchant-facing Shopify admin panel, accessible only to the team via secure authentication (e.g., admin user login) and hosted outside the Shopify embedded app context (e.g., a standalone web app at `admin.herethere.com`).

2 Advice for Designing the Internal Admin Module

2.1 Key Features

Based on the requirement for an overview and management of merchants, the following features are recommended, tailored to the needs and informed by the database schema (`herethere_full_schema.sql`):

2.1.1 Overview Dashboard

- **Metrics**: Display high-level statistics:
  - Total merchants (`merchants` table count).
  - Active merchants (e.g., with recent orders or points transactions).
  - Total orders processed (`usage_records.order_count` sum).
  - Total points issued/redeemed (`points_transactions` sum).
  - Revenue from paid plans (`plans.base_price` and `usage_records`).
- **Charts**: Visualize trends (e.g., merchant growth, points activity) using `Chart.js`.
- **Recent Activity**: Show recent merchant signups, API errors (`api_logs`), or audit events (`audit_logs`).

2.1.2 Merchants Management

- **List Merchants**: Table of all merchants with columns for:
  - Merchant ID (`merchants.merchant_id`).
  - Shopify domain (`merchants.shopify_domain`).
  - Plan (`merchants.plan_id`, joined with `plans.name`).
  - Order count (`usage_records.order_count` for current period).

  - Status (e.g., active, suspended, trial).
  - Created/updated dates (`merchants.created_at`, `updated_at`).
- **Search/Filter**: Search by domain, email, or plan; filter by status or order count.
- **View Merchant Details**: Drill down into a merchant's profile:
  - Store details (`merchants.brand_settings`, `language`).
  - Customer count (`customers` count per `merchant_id`).
  - Points transactions (`points_transactions` summary).
  - Integrations (`integrations` list).
  - API logs (`api_logs` for debugging).
- **Manage Merchants**:
  - Activate/suspend accounts (add `status` column to `merchants`).
  - Upgrade/downgrade plans (`merchants.plan_id`).
  - Reset API token (`merchants.api_token`).
  - Adjust points balances for customers (`customers.points_balance` via `points_transactions`).
  - Delete merchant (soft delete with `status` or hard delete with cascading constraints).

2.1.3 Admin Users Management

- **List Admins**: Table of admin users (`admin_users`) with `username`, `email`, `created_at`.
- **Manage Admins**: Add/edit/delete admin accounts, update passwords (hashed with `bcrypt`).
- **Role-Based Access**: Basic roles (e.g., superadmin, support) using `admin_users.metadata` (JSONB).

2.1.4 Audit and Logs

- **Audit Logs**: View actions by admin users (`audit_logs.action`, `target_table`, `target_id`).
- **API Logs**: Monitor Shopify API calls (`api_logs.route`, `status_code`, `timestamp`) for debugging.
- **Import Logs**: Track data imports (`import_logs.success_count`, `fail_reason`).

2.1.5 Platform Settings

- **Global Configurations**: Manage default plans (`plans`), points expiry rules (`program_settings.exp` or feature flags (`merchants.features_enabled`).
- **Email Templates**: Edit default email templates (`email_templates`) for all merchants.
- **Integrations**: Monitor platform-wide integration health (`integrations.status`).

2.2 Implementation Approach

- **Separation**: Host the internal admin module as a standalone web app (e.g., `admin.herethere.com`) separate from the Shopify embedded app to avoid Shopify's embedded app restrictions and simplify authentication.
- **Authentication**:
  - Use `admin_users` table for login with `username`/`password` (`bcrypt` hashing).
  - Implement JWT-based authentication for secure API access.
  - Optionally integrate with OAuth2 (e.g., Auth0) for SSO in the future.
- **Tech Stack**:
  - *Backend*: Use `NestJS` (TypeScript) for admin APIs, leveraging existing `merchants`, `admin_users`, `api_logs`, etc., tables from `herethere_full_schema.sql`. Integrate with `PostgreSQL` for data storage and `Redis` for caching overview metrics.
  - *Frontend*: Build a new `Vite + React` (TypeScript) app with `Tailwind CSS` (no Polaris, as it's not Shopify-embedded) for a clean, developer-focused UI. Use `Chart.js` for dashboard visualizations.
  - *Database*: Use existing `PostgreSQL` schema with JSONB fields (e.g., `merchants.brand_settings`, `admin_users.metadata`) and add a `status` column to `merchants` for activation/suspension.
  - *Deployment*: Host on a VPS (e.g., Ubuntu) using `Docker Compose` for `NestJS`, `PostgreSQL`, `Redis`, and the `React` frontend. Use `Nginx` as a reverse proxy to serve frontend assets and route API requests to `NestJS`.
  - *Security*: Restrict access via IP whitelisting, VPN, or role-based permissions in `admin_users.metac`
- **Scalability**:
  - Cache overview metrics (e.g., merchant count, points issued) in `Redis` to reduce `PostgreSQL` load.
  - Use `PostgreSQL` indexes on `merchants(shopify_domain)`, `api_logs(merchant_id, timestamp)` for fast queries.
  - Paginate merchant/customer lists to handle 5,000+ merchants.
- **Monitoring**:
  - Log admin actions in `audit_logs` for traceability.
  - Integrate with tools like `Sentry` for error tracking.

2.2.1 Enhancements & Best Practices

- Add session expiry and refresh token logic for JWT to reduce risk from stolen tokens.
- Consider optional multi-factor authentication (MFA) for superadmin accounts.
- Enforce strong password requirements and periodic password rotation for admin users.
- Explicitly implement rate limiting for sensitive admin endpoints to prevent brute-force attacks.

- Make `audit_logs` append-only or store them to prevent tampering.
- Use environment variables or a secrets manager for sensitive config (JWT secrets, DB passwords).

2.3 Integration with Existing Stack

- **NestJS Backend**: Add admin-specific endpoints (e.g., `/admin/merchants`, `/admin/points/adjust`) to the existing `NestJS` server, secured with JWT middleware. Reuse existing services for Shopify integrations (`@shopify/shopify-app-express` for OAuth, webhooks) and Twilio SMS referrals.
- **Frontend**: Create a new `Vite + React` app (`admin-frontend`) alongside the merchant-facing frontend (dashboard, customer widget). Share reusable components (e.g., `Chart.js` visualizations, table components) where possible, but avoid `Polaris` for the admin module to maintain a distinct, non-Shopify UI.
- **Database**: Reuse `PostgreSQL` tables (`merchants`, `admin_users`, `api_logs`, `audit_logs`, `usage_records`, `points_transactions`) and add a `status` column to `merchants` for activation/suspension. Leverage `Redis` for caching dashboard metrics.
- **Deployment**: Deploy the admin module on the same VPS as the merchant app, using separate `Docker` containers for the admin frontend and backend. Configure `Nginx` to route `/admin/*` to the admin app and `/api/admin/*` to `NestJS` APIs.

2.3.1 Enhancements & Best Practices

- Structure the codebase so new admin features (e.g., new analytics, new integrations) can be added easily (modular design).
- Schedule automated backups for the admin module's data and document the restore process.

2.4 Security Considerations

- **Authentication**: Use `bcrypt` for password hashing (`admin_users.password`) and JWT for session management.
- **Authorization**: Implement role-based access in `admin_users.metadata` (e.g., `{ "role": "superadmin" }`).
- **Audit Trail**: Log all admin actions (e.g., merchant suspension, points adjustment) in `audit_logs`.
- **API Security**: Require JWT for all `/admin/*` endpoints, validate inputs, and rate-limit requests.
- **Data Privacy**: Mask sensitive data (e.g., `merchants.api_token`, `customers.email`) in the UI for non-superadmins.

2.4.1 Enhancements & Best Practices

- Log all access to sensitive data (not just actions) for compliance.

2.5 User Experience

- **Simplicity**: Design a clean, table-heavy UI with search, filters, and pagination for easy merchant management.
- **Non-Technical Access**: Provide a pre-built `Docker Compose` setup and one-click deployment scripts for the VPS to simplify hosting.
- **Documentation**: Include a README with login instructions, feature descriptions, and troubleshooting steps.

2.5.1 Enhancements & Best Practices

- Add bulk actions (e.g., suspend multiple merchants, adjust points for multiple customers) for efficiency.
- Consider a "login as merchant" feature for support/admins to troubleshoot merchant issues directly.
- Allow exporting merchant/customer data (CSV/Excel) for reporting or migration.

2.6 Potential Challenges

- **Learning Curve**: `NestJS`'s structure is mitigated by AI-generated code (e.g., via Grok) and the study plan for TypeScript/`NestJS`.
- **Data Volume**: 5,000+ merchants require efficient queries (solved with indexes, pagination, `Redis`).
- **Security**: Robust authentication/authorization is critical to prevent unauthorized access (solved with JWT, `audit_logs`).
- **Maintenance**: Admin module adds complexity, but modular design (separate frontend/backend) simplifies updates.

2.7 Documentation & Onboarding

2.7.1 Enhancements & Best Practices

- Provide a simple user guide for new admins, including screenshots and common workflows.
- Generate and maintain `OpenAPI/Swagger` docs for all admin endpoints.

2.8 Compliance & Privacy

2.8.1 Enhancements & Best Practices

- Ensure GDPR/CCPA support: ability to export/delete merchant/customer data on request.

2.9 Summary Table of Key Suggestions

Area	Suggestion
Auth/Security	JWT expiry/refresh, MFA, password policy, rate limiting, audit log integrity, secrets management
Monitoring	Alerts for suspicious activity, performance monitoring
UX	Bulk actions, impersonation, export/import
Maintenance	Modular code, automated backups
Docs	Admin user guide, `OpenAPI` docs
Compliance	GDPR/CCPA support, access logging

Table 1: Summary of Key Suggestions for Internal Admin Module