StoreFlow — A1.3 Authentication, Authorization & Security Deep Dive

1. Overview

This document provides a deep dive into authentication, authorization, and security for StoreFlow. It defines how users and customers authenticate, how permissions are enforced, how multi-tenant isolation is guaranteed, and which security controls protect the system from common attack vectors.

Scope:

• Merchant/staff authentication

• Customer account model (optional)

• Role & permission model

• Multi-tenant isolation

• Policies & middleware

• Session & cookie security

• Rate limiting & brute-force mitigation

• CSRF, CORS, and request integrity

• Logging, auditing, and security monitoring

2. Identity Types in StoreFlow

StoreFlow has two main identity categories:

1) Merchant Identities (Dashboard Users)

• Owner

• Store Manager

• Staff

2) Customer Identities (Storefront Users)

• Guest Customers (no login)

• Registered Customers (optional accounts)

Key principles:

• Merchant identities are strictly bound to a single merchant.

• Customer identities are scoped by merchant (customer in Store A is independent of Store B under another merchant).

• The dashboard is never exposed to customers.

• Customers use different auth flows and are fully separated from merchant auth.

3. Merchant Authentication – Username/Password

Merchant & staff login uses username/password.

Data:

• users.username (unique per system)

• users.password_hash (bcrypt)

Login Flow:

1. User submits username + password.

2. System finds user by username.

3. System verifies password using bcrypt.

4. If valid:

- Set authenticated session.

- Load merchant_id from user record.

- Determine accessible stores.

- If > 1 store, redirect to store-selection.

- If = 1 store, bind store_id automatically.

5. If invalid:

- Increment login attempt counter for rate limiting.

- Return generic error message.

Security:

• No disclosure if username exists or not.

• Password is never logged.

• Using Laravel's built-in authentication guard (session-based).

4. Customer Authentication – Optional Accounts

Customers can check out as guest or create an account.

Guest:

• No password.

• Identified by email + mobile during order.

• Can track orders using public_id + email combination (future).

Registered:

• Same customer record, but with password_hash set.

• Login via storefront login page (separate route & guard).

• Allows:

- viewing order history

- faster checkout via pre-filled details

- loyalty points visibility

Security:

• Distinct auth guard (e.g., 'customer') separate from 'web' merchant guard.

• Never mix session spaces between merchant and customer auth.

• Customer login constrained to a single merchant context.

5. Role Model & Permission Matrix

Merchant user roles:

OWNER:

• Bound to a merchant.

• Capabilities:

- Full CRUD on stores

- Manage all users (create, edit, deactivate)

- Configure shipping, loyalty, and themes

- View all orders for merchant

- View all audit logs

STORE MANAGER:

• Bound to specific store(s) via store_users.

• Capabilities:

- Manage products for assigned stores

- Manage shipping config for assigned stores

- Manage loyalty config for assigned merchant (if allowed)

- View + update orders in assigned stores

- View audit logs for assigned stores only

STAFF:

• Bound to specific store(s).

• Capabilities:

- View current orders in Operations view

- Update order statuses within allowed transitions

- Print receipts/invoices

- No access to:

- global merchant settings

- shipping/loyalty settings

- user management

Permission enforcement:

• Implemented via Laravel Policies + Gates.

• Fine-grained ability checks (e.g., viewAnyOrders, updateOrderStatus, manageProducts).

6. Multi-Tenant Isolation Rules

Tenant isolation is enforced at multiple layers:

Database:

• All core tables include merchant_id.

• Many include store_id.

• Queries always include merchant_id filters.

• Store filters added for store-specific data.

Application:

• Middleware attaches merchant_id and store_id from user session.

• Controllers do not accept merchant_id directly from incoming payloads.

• Policies validate that target objects belong to current merchant/store.

Authorization:

• A user cannot:

- access another merchant's store or orders

- access stores they are not linked to (unless owner)

- see or operate on orders outside their assigned stores.

Defense-in-depth approach:

• DB FK constraints ensure store rows always belong to merchant.

• Policies operate on retrieved models.

• Controllers avoid exposing cross-tenant identifiers in URLs where possible.

7. Policies & Middleware Design

MIDDLEWARE LAYERS:

auth (web):

• Validates that a user is authenticated as a merchant user.

tenant:

• Reads merchant_id and store_id from session.

• Ensures store belongs to merchant.

• Aborts with 403 if mismatch.

role:

• Optional middleware to short-circuit based on user.role (owner/manager/staff).

• Mostly handled via policies.

POLICIES:

StorePolicy:

• view/manage stores based on user.role == owner.

OrderPolicy:

• viewAny: user belongs to merchant & to store.

• view: user authorized for that specific store.

• updateStatus: role is manager or staff for store.

UserPolicy:

• Only owner can manage users within merchant.

All policies:

• Must assume that if policy is executed, merchant_id in route/session is trusted.

• Must double-check entity.merchant_id == user.merchant_id.

8. Session & Cookie Security

StoreFlow uses Laravel's session-based auth.

Security settings:

• SESSION_DRIVER = 'redis' or 'database' in production.

• SESSION_SECURE_COOKIE = true (HTTPS-only).

• SESSION_SAME_SITE = 'lax' or 'strict' for dashboard.

• SESSION_HTTP_ONLY = true to prevent JS access.

Cookie Scope:

• Merchant dashboard and customer storefront may run under different subdomains.

• Each can have isolated session cookies to avoid leakage.

Session Lifecycle:

• Regenerate session ID on login and logout.

• Invalidate sessions on password change or role changes (eventual).

• Shorter dashboard session TTLs (for higher security).

• "Remember me" disabled for merchant/staff; optional for customers only.

9. Rate Limiting & Brute-Force Protection

Rate limiting is applied via Laravel's RateLimiter:

Merchant Login:

• 5 attempts per minute per username + IP.

• After threshold, lock account for short duration (e.g., 5–15 minutes).

• Log suspicious patterns in audit/log systems.

Customer Login:

• Slightly more permissive but still rate-limited (10/minute).

• CAPTCHAs may be optionally introduced (future).

Public Endpoints:

• Shipping quote endpoint: limit per IP per minute.

• Checkout endpoint: limited to prevent abuse (e.g., card testing attack in future Stripe integration).

Implementation:

• RateLimiter::for('login', ...) definitions.

• Throttling middleware applied to routes.

10. CSRF, CORS & Request Integrity

CSRF:

• Laravel's built-in CSRF protection enabled on all dashboard and customer POST/PUT/DELETE routes.

• Tokens bound to user sessions.

• For public API endpoints (e.g., shipping quote), CSRF is not required if they are stateless and under separate domain.

CORS:

• Dashboard CORS locked to internal domains only.

• Storefront API CORS limited to allowed store domains.

• Preflight responses configured to allow required headers/methods only.

Request Integrity:

• All sensitive actions (login, checkout, configuration updates) enforced over HTTPS.

• No mixed-content allowed.

• Input validation & sanitization on every incoming request layer via Laravel validation.

11. Logging, Auditing & Security Monitoring

LOGGING:

• Central structured logs for:

- logins (success/failure)

- admin changes (settings, roles, shipping, loyalty)

- security-relevant events (excessive rate-limiting, suspicious behavior)

AUDIT LOGS:

• audit_logs table persists semantic events:

- who (user_id)

- what (entity, entity_id, action)

- when (created_at)

- context (meta_json)

SECURITY MONITORING:

• Optional integration with alerting tools:

- Alert if many failed logins for one merchant.

- Alert if mass updates to shipping rates.

- Alert if many orders are cancelled within a narrow window.

This layered approach allows support, forensics and compliance reviews to be done reliably.

12. Data Protection & Secure Coding Practices

DATA PROTECTION:

• Passwords: bcrypt via password_hash fields.

• API keys (Stripe, SMTP): never stored in DB, only environment variables or secret store.

• Potential encryption for sensitive fields (e.g., customer mobile) using Laravel's encryption helpers (future).

SECURE CODING:

• Use parameterized queries via Eloquent or Query Builder.

• Never concatenate untrusted input into raw SQL.

• Validate all request data.

• Avoid exposing internal IDs where not required; use public_id tokens.

• Regularly update dependencies and framework versions.

• Disable debug mode in production.