ZNotice — System Architecture

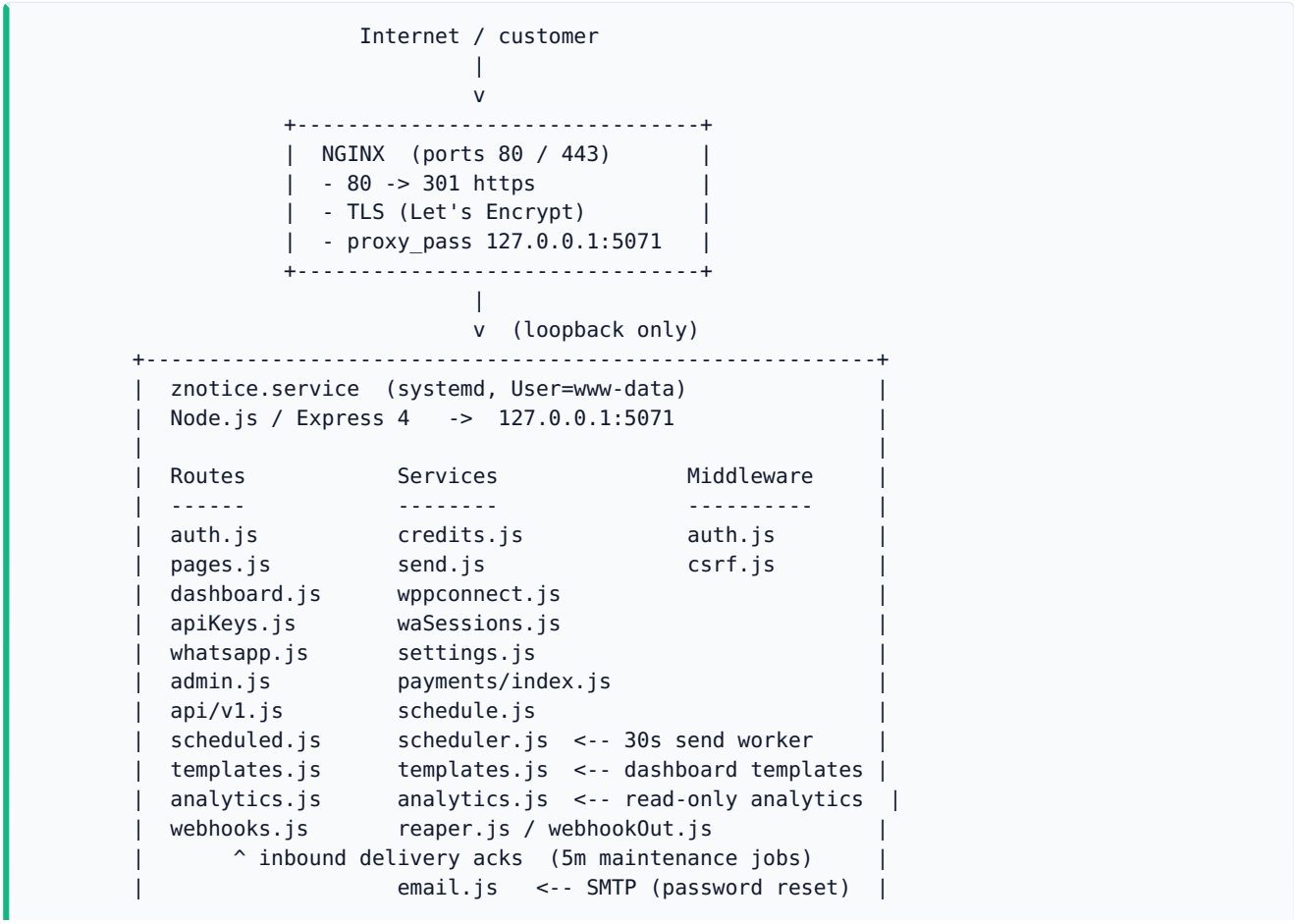
1. Overview

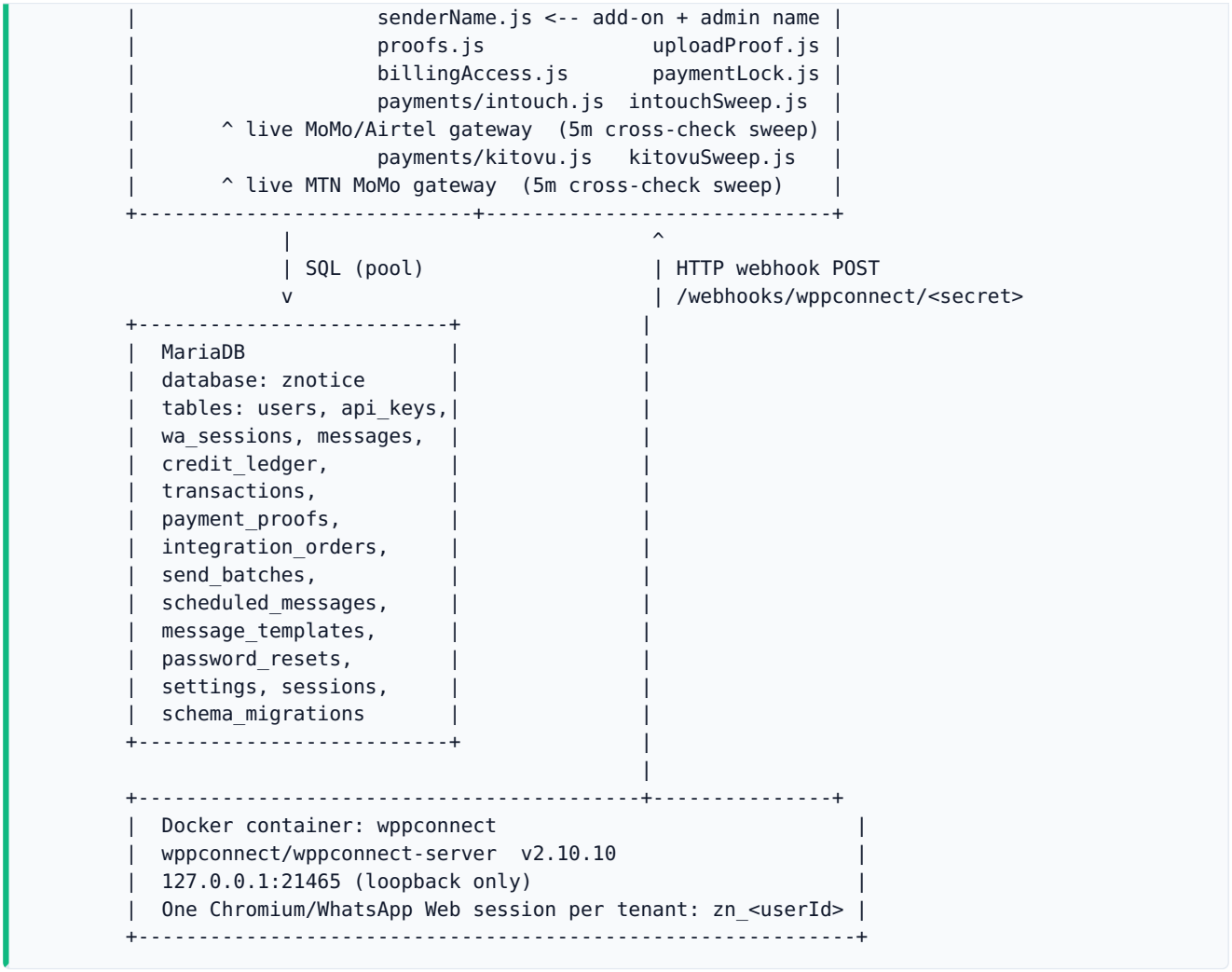
ZNotice is a single-node, multi-tenant SaaS built as a classic three-tier web application. NGINX terminates TLS and reverse-proxies to a Node.js/Express application. The application stores all state in MariaDB and delegates the actual WhatsApp protocol work to a locally-hosted WPPConnect Server process running in Docker.

Key properties:

- **Single-node deployment.** The app is bound to `127.0.0.1:5071` and only reachable through NGINX. There is no horizontal scaling; in-memory state (rate-limit buckets, settings cache, WPPConnect bearer-token cache) is per-process.
- **Tenant isolation by user id.** Every customer has exactly one WhatsApp session row, one credit ledger, and many API keys.
- **Prepaid credits are the source of truth for billing.** The `credit_ledger` table is authoritative; a balance is `SUM(delta)`.
- **Payments via pluggable adapters.** A payment adapter interface supports a manual offline flow (`manual`) and two live mobile-money gateways (`intouch`, Rwanda MTN MoMo / Airtel Money; `kitovu`, Rwanda MTN MoMo), added without touching routes.

2. Component diagram





Supporting host services (not on the request path):

- **certbot.timer** — renews the Let's Encrypt certificate.
- **/etc/cron.d/znotice-backup** — daily DB + app/secret archive (02:30).
- **/etc/cron.d/znotice-health** — health probe every 5 minutes.
- **logrotate** — weekly rotation of the two ops log files.

In-process background jobs (started from `server/index.js`, no extra ports):

- **Scheduler** (`server/services/scheduler.js`) — every **30s** plus an immediate kick after a successful enqueue; claims due `scheduled_messages` rows and dispatches them through `services/send.js`.
- **Reaper** (`server/services/reaper.js`) — every **5 minutes** plus one pass ~30s after boot; reconciles stale WhatsApp sessions, resets interrupted scheduler claims, prunes old scheduled rows/batches, and deletes expired plus stale anonymous sessions.
- **Webhook worker** (`server/services/webhookOut.js`) — every **30s** plus a kick; delivers queued customer webhooks.
- **Intouch sweep** (`server/services/intouchSweep.js`) — every **5 minutes**; cross-checks pending `intouch` transactions older than **2 minutes** against the gateway's `gettransactionstatus` and confirms any that have settled (heals lost/early callbacks).
- **Kitovu sweep** (`server/services/kitovuSweep.js`) — every **5 minutes**; cross-checks pending `kitovu` transactions older than **2 minutes** against the gateway's `GET /merchant-transaction/payment/:externalId` and confirms any that have settled (heals lost/early callbacks).

Outbound email (`server/services/email.js`) is **not** a background job: it is called inline (fire-and-forget) by the password-reset routes and is inert until `SMTP_HOST` is configured.

Usage analytics (Phase 7) is **not** a background job: it is computed synchronously per request from read-only aggregate queries over the existing tables and adds no worker, queue or cache.

3. Ports and processes

Port	Binding	Process	Purpose
80	`0.0.0.0`	NGINX	ACME challenge + HTTPS redirect
443	`0.0.0.0`	NGINX	Public HTTPS entry point
5071	`127.0.0.1`	`node server/index.js` (systemd `znotice`)	Application HTTP server
21465	`127.0.0.1`	Docker container `wppconnect`	WPPConnect WhatsApp engine
3306	localhost	MariaDB	Application database

Process	Manager	Unit / name	User
ZNotice app	systemd	`znotice.service`	`www-data`
WPPConnect	Docker	container `wppconnect`	container
MariaDB	systemd	`mariadb.service`	`mysql`
NGINX	systemd	`nginx.service`	`www-data`

systemd unit (`/etc/systemd/system/znotice.service`)

- `After=network.target mariadb.service`
- `Type=simple`, `User=www-data`, `Group=www-data`
- `WorkingDirectory=/var/www/znotice.consultbofa.com`
- `EnvironmentFile=/var/www/znotice.consultbofa.com/.env`
- `ExecStart=/usr/bin/node server/index.js`
- `Restart=always`, `RestartSec=5`
- Hardening: `NoNewPrivileges=true`, `PrivateTmp=true`

NGINX site (`/etc/nginx/sites-available/znotice.consultbofa.com`)

- Port 80: serves `/.well-known/acme-challenge/` from `/var/www/html`, then redirects everything else to HTTPS.
- Port 443: TLS 1.2/1.3, `client_max_body_size 50m`, security headers (`X-Content-Type-Options`, `X-Frame-Options`, `Referrer-Policy`), and a proxy to `http://127.0.0.1:5071` with WebSocket upgrade headers and a 3600s read/send timeout.
- The TLS certificate is issued with the webroot method and NGINX is configured manually (no certbot NGINX installer).

4. Database schema

Charset `utf8mb4` / collation `utf8mb4_unicode_ci`, engine InnoDB. Migrations live in `server/migrations/` and are applied exactly once (tracked in `schema_migrations`). The `sessions` table is created automatically by `express-mysql-session`.

`users`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`name`	VARCHAR(120)	
`email`	VARCHAR(190) UNIQUE	lowercased on registration
`password_hash`	VARCHAR(255)	bcrypt, cost 12
`role`	ENUM('user','admin')	default `user`
`status`	ENUM('active','suspended')	default `active`
`accepted_terms_at`	TIMESTAMP NULL	set on registration (Phase 2)
`sender_name_until`	DATETIME NULL	dedicated sender-name add-on expiry, UTC (Phase 10); `NULL` = never purchased. Active while `> NOW()`
`created_at`, `updated_at`	TIMESTAMP	

`api_keys`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`label`	VARCHAR(120)	default `Default`
`key_prefix`	VARCHAR(16)	first 12 chars of the key, for display
`key_hash`	CHAR(64) UNIQUE	SHA-256 hex of the full key
`last_used_at`	TIMESTAMP NULL	refreshed on each authenticated request
`revoked_at`	TIMESTAMP NULL	soft delete
`created_at`	TIMESTAMP	

`wa_sessions`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED UNIQUE FK -> users	one session per user
`session_name`	VARCHAR(64) UNIQUE	`zn_<userId>`

Column	Type	Notes
`status`	ENUM('disconnected','pending','connected','error')	default `disconnected`
`phone_number`	VARCHAR(32) NULL	connected WID digits
`profile_name`	VARCHAR(60) NULL	last WhatsApp profile/push name pushed through ZNotice (Phase 9); `NULL` = never set here
`connected_at`	TIMESTAMP NULL	first connection time
`last_qr_at`	TIMESTAMP NULL	last QR emission
`created_at`, `updated_at`	TIMESTAMP	

`messages`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`api_key_id`	BIGINT UNSIGNED NULL FK -> api_keys	`ON DELETE SET NULL`; null for dashboard sends
`to_number`	VARCHAR(32)	normalized digits
`body`	TEXT NULL	message text / media caption
`media_url`	VARCHAR(500) NULL	
`status`	ENUM('queued','sent','delivered','read','failed')	default `queued`
`wpp_message_id`	VARCHAR(190) NULL	engine message id (indexed)
`error`	TEXT NULL	sanitized failure detail
`credits_charged`	INT UNSIGNED	1 on success, 0 on failure
`created_at`, `updated_at`	TIMESTAMP	

Indexes: `(user_id, created_at)`, `wpp_message_id`, `api_key_id`, `(user_id, status)` from Phase 2, and `(created_at)` from Phase 7 (migration `007_analytics_indexes.sql`, for platform-wide date-range scans).

`credit_ledger` — authoritative balance

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`delta`	INT	signed (+ topup/refund, − charge)
`balance_after`	INT	running balance at write time

Column	Type	Notes
`reason`	ENUM('topup','purchase','message','refund','admin_adjust','bonus')	
`ref_type`	VARCHAR(40) NULL	e.g. `transaction`, `message`, `api_key`, `dashboard`, `admin`
`ref_id`	BIGINT UNSIGNED NULL	
`note`	VARCHAR(255) NULL	
`created_at`	TIMESTAMP	

Balance = `COALESCE(SUM(delta), 0)` for a user.

`transactions`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`provider`	VARCHAR(40)	default `manual`
`provider_ref`	VARCHAR(190) NULL UNIQUE	`ZN-XXXXXXXX` (Phase 3 unique key)
`amount_cents`	INT UNSIGNED	
`currency`	CHAR(3)	default `USD`
`credits`	INT UNSIGNED	credits to grant (0 for integration fee)
`status`	ENUM('pending','paid','failed','refunded')	default `pending`
`meta`	JSON NULL	`{ type: 'credits'
`created_at`	TIMESTAMP	
`paid_at`	TIMESTAMP NULL	

Indexes: `(user_id, created_at)` from Phase 1 and `(status, paid_at)` from Phase 7 (migration `007_analytics_indexes.sql`, for paid-revenue scans).

`payment_proofs` (Phase 12, migration `012_payment_proofs.sql`)

Metadata for customer-uploaded proof-of-payment files (screenshots/receipts) attached to a `pending` manual transaction. The bytes live on disk (see §8(i)); this table stores only metadata. `original_name` is display-only and is never used as a path; `stored_name` is the generated on-disk filename; `mime` is **server-sniffed** (the client value is never trusted).

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`transaction_id`	BIGINT UNSIGNED FK -> transactions	`ON DELETE CASCADE`

Column	Type	Notes
`original_name`	VARCHAR(255)	sanitized client name (control chars/path stripped), display only
`stored_name`	VARCHAR(128)	generated `<16 hex>.<ext>`, ASCII only
`mime`	VARCHAR(100)	server-sniffed MIME (`image/jpeg` , `image/png` , `image/webp` , `application/pdf`)
`size_bytes`	INT UNSIGNED	byte length of the stored file
`created_at`	TIMESTAMP	

Index: `idx_proof_tx (transaction_id, created_at)`, plus the FK `fk_payment_proofs_tx` -> `transactions(id) ON DELETE CASCADE`. Rows cascade-delete with their transaction; files live under `<UPLOADS_DIR>/payment-proofs/<transaction_id>/<16hex>.<ext>` (see §8(i)).

`integration_orders`

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`transaction_id`	BIGINT UNSIGNED NULL FK -> transactions	`ON DELETE SET NULL` (Phase 3)
`status`	ENUM('requested','paid','in_progress','delivered','cancelled')	default `requested`
`amount_cents`	INT UNSIGNED	default 2000
`contact_channel`	VARCHAR(120) NULL	`"<channel>: <value>"`
`requirements`	TEXT NULL	includes optional preferred stack
`admin_notes`	TEXT NULL	internal notes
`created_at` , `updated_at`	TIMESTAMP	

`send_batches`

One row per bulk job (and per dashboard-created schedule, which is always a batch). Counters are advanced atomically by the scheduler as items are delivered; `message` / `media_url` store the **original** template (per-recipient bodies live on `scheduled_messages`).

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`api_key_id`	BIGINT UNSIGNED NULL FK -> api_keys	`ON DELETE SET NULL` ; null for dashboard jobs
`label`	VARCHAR(120) NULL	optional batch label

Column	Type	Notes
`message`	TEXT NULL	original template, placeholders unresolved
`media_url`	VARCHAR(500) NULL	
`scheduled_at`	DATETIME	UTC due time
`status`	ENUM('scheduled','processing','completed','cancelled')	default `scheduled`
`total_count` / `sent_count` / `failed_count` / `cancelled_count`	INT UNSIGNED	counters, updated by the scheduler
`created_at`, `updated_at`	TIMESTAMP	
`completed_at`	DATETIME NULL	set when terminal

Indexes: `(user_id, created_at)`, `(status, scheduled_at)`.

`scheduled_messages`

The durable per-recipient send queue. `body` is **already placeholder-resolved at enqueue time**; `next_attempt_at` drives retry backoff. A row may belong to a batch (bulk / dashboard) or stand alone (a single API schedule with `scheduled_at`, `batch_id IS NULL`).

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`api_key_id`	BIGINT UNSIGNED NULL FK -> api_keys	`ON DELETE SET NULL`
`batch_id`	BIGINT UNSIGNED NULL FK -> send_batches	`ON DELETE CASCADE`
`recipient_name`	VARCHAR(120) NULL	substituted into `{{name}}`
`to_number`	VARCHAR(32)	normalized digits
`body`	TEXT NULL	resolved message text
`media_url`	VARCHAR(500) NULL	
`scheduled_at`	DATETIME	UTC due time
`next_attempt_at`	DATETIME NULL	retry backoff gate
`status`	ENUM('pending','sending','sent','failed','cancelled')	default `pending`; `sending` is the claim state (migration 006)
`attempts`	TINYINT UNSIGNED	default 0; terminal after 3
`last_error`	VARCHAR(500) NULL	sanitized; e.g. `interrupted_by_restart`
`message_id`	BIGINT UNSIGNED NULL FK -> messages	set on success
`created_at`, `updated_at`	TIMESTAMP	

Indexes: `(status, next_attempt_at, scheduled_at)`, `(user_id, created_at)`, `(batch_id)`.

`message_templates` (Phase 11, migration `011_message_templates.sql`)

Per-user saved message snippets used to prefill the dashboard WhatsApp and scheduled forms (`?template=<id>`). Private to the owner; never exposed through API v1.

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`name`	VARCHAR(80)	trimmed, internal whitespace collapsed; unique per user
`body`	TEXT	message text/caption; required, ≤4096 chars
`media_url`	VARCHAR(500) NULL	optional; `http(s)` only, control chars rejected
`created_at`, `updated_at`	TIMESTAMP	`updated_at` bumps on edit (drives list order)

Indexes: `UNIQUE(user_id, name)` (`uq_message_templates_user_name`), plus the `user_id` foreign key. All service queries filter on `user_id` in the `WHERE` clause, so a foreign id affects 0 rows and `resolvePrefill` returns `null`.

`settings`

Column	Type	Notes
`key`	VARCHAR(80) PK	
`value`	TEXT NULL	
`updated_at`	TIMESTAMP	

Seeded keys: `price_per_1000_credits_cents` (1000), `integration_fee_cents` (2000), `currency` (USD), `default_country_code` (250), `send_rate_per_min` (60), `max_concurrent_sessions` (15), `manual_payment_instructions`, `billing_email`, the Phase 6 queue caps `bulk_max_recipients` (1000), `schedule_max_days_ahead` (90) and `scheduled_max_pending` (5000), the Phase 10 sender-name add-on keys `sender_name_price_cents` (1000) and `sender_name_days` (30), the Phase 11 template cap `templates_max_per_user` (200), the Phase 14 IntouchPay exchange rate `intouch_rwf_rate` (float, default `1477.29`, RWF per USD at checkout), and the Phase 15 KitovuPay exchange rate `kitovu_rwf_rate` (float, default `1477.29`, RWF per USD at checkout).

`schema_migrations`

Column	Type	Notes
`filename`	VARCHAR(190) PK	applied migration file name
`applied_at`	TIMESTAMP	

`sessions` (managed by `express-mysql-session`)

Stores Express session data (`znotice.sid` cookie), created automatically on first use. Contains session state including `userId` and `csrfToken`. `sessions.expires` holds UNIX **seconds** (not milliseconds); the reaper relies on that when pruning expired/anonymous rows.

`password_resets` (Phase 8, migration `008_password_resets.sql`)

Single-use, 30-minute password-reset tokens. Only the SHA-256 hash is stored; the raw token exists solely in the emailed link.

Column	Type	Notes
`id`	BIGINT UNSIGNED PK AI	
`user_id`	BIGINT UNSIGNED FK -> users	`ON DELETE CASCADE`
`token_hash`	CHAR(64) UNIQUE	SHA-256 hex of the raw token
`expires_at`	DATETIME	UTC; 30 minutes after issue
`used_at`	TIMESTAMP NULL	set on consumption (single-use)
`created_at`	TIMESTAMP	

Index: unique `token_hash`, plus `user_id`. The token is claimed with a conditional `UPDATE ... WHERE used_at IS NULL AND expires_at > NOW()` inside the password-change transaction (see §8(f)).

5. Session naming and WhatsApp engine integration

- **Session name:** `zn_<userId>` (e.g. user 42 -> `zn_42`). Enforced unique in `wa_sessions.session_name` and one row per user.
- **Engine auth:** WPPConnect does not accept the raw `SECRET_KEY`. The app computes a per-session bearer token as `bcrypt(sessionName + SECRET_KEY)` (cost 10), URL-safe encoded (`/` → `_`, `+` → `-`), and caches it in memory (`services/wppconnect.js`). The engine validates with `bcrypt.compare(session + SECRET_KEY, presented)`.
- **Webhooks are registered per session** via the `webhook` field in the `start-session` body — there is no `set-webhook` endpoint. The app passes `https://znotice.consultbofa.com/webhooks/wppconnect/<WEBHOOK_SECRET>`.
- **Engine endpoints used** (v2.10.10): `start-session`, `qrcode-session`, `status-session`, `close-session`, `send-message`, `send-file`, `get-phone-number`, `check-number-status/{phone}` and `change-username` (Phase 9 — `POST /api/{session}/change-username` `{ name }`, which the engine maps to `client.setProfileName(name)`; called only while the session is `connected`).
- **Status mapping:** engine `CONNECTED` → `connected`; `QRCODE`, `INITIALIZING`, `PHONECODE` → `pending`; `CLOSED`, `DISCONNECTED`, empty → `disconnected`; anything else → `error`.

6. Payment adapter design

`server/services/payments/index.js` is a provider registry. Each adapter must implement a stable interface so routes never change when a gateway is added:

Method	Contract
`createCheckout({ user, amountCents, credits, meta, conn? })`	Inserts a `pending` transaction, returns `{ transactionId, reference, instructions, provider }`. The optional `conn` joins a caller transaction (defaults to the shared pool).
`markPaid(transactionId, adminUserId)`	Idempotently settles a payment; grants credits for credit top-ups, advances a linked integration order to `paid`, or extends the sender-name entitlement. The shared `settlePaid()` helper accepts an optional **credit-ledger note override** so adapters (e.g. `intouch`) can tag the ledger entry distinctly.
`markFailed(transactionId, adminUserId)`	Idempotently marks a pending transaction failed.
`handleWebhook(req)`	Reserved for async gateways; `manual` returns `{ ok: false }`. `intouch` unwraps the `jsonpayload`, cross-checks `gettransactionstatus`, and returns a pure ACK. `kitovu` cross-checks `GET /merchant-transaction/payment/:externalId` and returns a pure ACK.

Design details:

- **Registry:** `providers = { manual, intouch, kitovu }`; `getProvider(name)` resolves an adapter and throws `UNKNOWN_PROVIDER` otherwise.
- **References:** `manual` uses `ZN-` + 8 characters from an unambiguous alphabet (no `0/O/1/I/L`), collision-retried up to 5 times against the unique `provider_ref` index. `intouch` uses `ZNI-<id>-<8hex>` (a distinct prefix, since the merchant account is shared with another project whose refs use a different prefix). `kitovu` uses a **6-digit numeric** `externalId` (the gateway's lookup key), checked for uniqueness in `provider_ref` before sending and retried on a `DUPLICATE_EXTERNAL_ID` rejection.
- **Idempotency:** settlement uses a single conditional `UPDATE ... WHERE id = ? AND status = 'pending'` as the concurrency gate. Exactly one caller wins `pending → paid`. If the subsequent credit add fails, the transition is rolled back to `pending` so an admin can retry.
- **Credit top-ups:** `credits == amount_cents` ($0.01/credit).
- **Integration service fee:** `credits = 0`; settling it flips the linked `integration_orders` row from `requested` to `paid`.
- **Sender-name add-on (Phase 10):** `credits = 0`, `meta = { type:'sender_name', days }`. Settlement extends `users.sender_name_until` from `GREATEST(COALESCE(sender_name_until, NOW()), NOW())` by `days`, where `days` is clamped to **1..365** (explicit finite check; falls back to the `sender_name_days` setting when unset). The atomic `pending → paid` gate means concurrent approvals cannot double-extend.
- **Settlement compensation:** if any post-transition work (credit add, integration-order update, or sender-name entitlement extension) throws, the transaction is reverted to `pending` (`paid_at = NULL`) and the error rethrown, so an admin can retry without losing the payment.
- **Instructions templating:** the admin-editable `manual_payment_instructions` string supports `{{reference}}`, `{{amount}}`, and `{{credits}}` placeholders.

Intouch adapter (Phase 14) — `server/services/payments/intouch.js`

The first live gateway. It uses the global `fetch` (no new dependencies) and a **30s timeout** on every call.

- **Auth.** A SHA256 hex digest over `username + account_number + partner_password + timestamp` (raw concatenation, UTC `YYYYMMDDHHMMSS`). Only the digest is sent; the account number and partner

password are never transmitted in cleartext. Credentials come from `.env` (`INTOUCH_USERNAME`, `INTOUCH_ACCOUNT_NUMBER`, `INTOUCH_PARTNER_PASSWORD`).

- **Endpoints.** `POST /requestpayment/` (async, returns `Pending` `1000`), `POST /gettransactionstatus/` (no amount), and `POST /getbalance/` (ops) against the live `https://www.intouchpay.co.rw/api` or sandbox `https://developer.intouchpay.co.rw/api/v1/sandbox` base (selected by `INTOUCH_MODE`). Every request carries an `X-Trace-Id` header.
- **Phone normalization.** Rwandan numbers are normalized to `2507XXXXXXXX` before `requestpayment`.
- **Currency & amount.** Checkout converts the USD `amount_cents` to **whole RWF** at `intouch_rwf_rate` (admin-editable float, default `1477.29`, `settings.getFloat()`); the RWF amount and rate are written into the transaction `meta` alongside `{ currency:'RWF', amount_rwf, rate, intouch:{ phone, tx_id, provider, statusdesc } }`. The `transactions` row itself stays in USD (`amount_cents`), so the ledger and analytics remain USD-only.
- **Confirmation.** Settlement only proceeds when the cross-checked status is settled **and** the returned RWF amount and currency match the transaction exactly (whole-RWF, off-by-one blocked).

Intouch webhook — callback / cross-check / ACK

The provider calls `POST /payments/intouch/webhook`, mounted **before** the session/CSRF middleware and requiring **no auth** — the callback is **unsigned**, so it is never trusted on its own. The request body carries the payload wrapped in a `jsonpayload` field (an object or a JSON string).

1. The handler unwraps `jsonpayload` and looks up the local transaction.
2. It always **cross-checks ground truth** by calling `gettransactionstatus` (3 bounded retries) before settling.
3. It responds with a **pure ACK** regardless of outcome: `200 {"message":"success","success":true,"request_id":"<uuid>"}`. Malformed, unknown or mismatched payloads are logged and still ACKed `200` — no validation detail is leaked and there is no retry storm.
4. A terminal failure (amount/currency mismatch, gateway error) is recorded in `meta.failure_reason` and the transaction stays `pending` (or is marked `failed`), where the sweep or an admin status check can recover it.

Intouch sweep — `server/services/intouchSweep.js`

Every **5 minutes** the sweep selects pending `intouch` transactions older than **2 minutes**, cross-checks each via `gettransactionstatus`, and confirms the ones that have settled. This heals callbacks that were lost or arrived before the transaction was fully written. It shares the same settle path and strict amount/currency matching as the webhook, so a late callback and the sweep cannot double-credit.

Kitovu adapter (Phase 15) — `server/services/payments/kitovu.js`

The second live gateway. It uses the global `fetch` (no new dependencies) and a **30s timeout** on every call.

- **Auth.** Every POST carries `x-api-key` (merchant API key), `x-api-timestamp` (Unix **milliseconds as a string**, `Date.now().toString()`) and `x-api-signature` = HMAC-SHA256 (lowercase hex) over the compact JSON envelope `{"method":"POST","path":"<path>","body":{...}}` keyed with the API secret. The HTTP body is the same body object serialized with `JSON.stringify` (compact, no escaped slashes/unicode); envelope key order is method, path, body. GET (status) needs only `x-api-key`. Credentials come from `.env` (`KITOVUPAY_API_KEY`, `KITOVUPAY_API_SECRET`).

- **Endpoints.** `POST /merchant-transaction/merchant/collection/mobile_money` (async, body `{ amount, payment_rail:'mtn', sender_account_issuer:'mtn', sender_account_name, sender_account_number, description, callbackUrl, externalId }`) against the UAT `https://gateway.uat.kitovupay.com` or live `https://gateway.kitovupay.com` base (selected by `KITOVUPAY_MODE`). Status is `GET /merchant-transaction/payment/:externalId` → `{ reference_id, status, reason }`. 429 responses are retried with bounded exponential backoff (3 attempts, 250ms/500ms).
- **External ID.** `externalId` is a **6-digit numeric string**, unique per merchant account. It is checked for uniqueness in `transactions.provider_ref` before sending; a `DUPLICATE_EXTERNAL_ID` rejection allocates a fresh ref and retries (bounded). `provider_ref` stays `NULL` until a collection is sent.
- **Phone normalization.** Rwandan numbers are normalized to `2507XXXXXXXX`.
- **Description.** Sanitized to plain text (letters/numbers/spaces) — symbols like `#` break the USSD push.
- **Currency & amount.** Checkout converts the USD `amount_cents` to **whole RWF** at `kitovu_rwf_rate` (float, default `1477.29`, `settings.getFloat()`); the RWF amount and rate are written into the transaction `meta` alongside `{ currency:'RWF', amount_rwf, rate, kitovu:{ phone, tx_id, status, requested_at } }`. The `transactions` row stays in USD.
- **Confirmation.** Settlement only proceeds when the cross-checked status is `APPROVED`; when a webhook supplies an amount it must match the transaction exactly (whole-RWF, off-by-one blocked).

Kitovu webhook — callback / cross-check / ACK

The provider calls `POST /payments/kitovu/webhook`, mounted **before** the session/CSRF middleware and requiring **no auth** — the callback is **unsigned**, so it is never trusted on its own. The body is `{ transactionId, externalId, status, amount, message }`.

1. The handler looks up the local transaction by `externalId`.
2. It always **cross-checks ground truth** by calling `GET /merchant-transaction/payment/:externalId` (3 bounded retries) before settling.
3. It responds with a **pure ACK** regardless of outcome: `200 {"message":"success","success":true,"request_id":"<uuid>"}`. Malformed, unknown or mismatched payloads are logged and still ACKed `200`.
4. A terminal failure (amount mismatch, gateway error) is recorded in `meta.failure_reason` and the transaction stays `pending`, where the sweep or an admin status check can recover it.

Kitovu sweep — `server/services/kitovuSweep.js`

Every **5 minutes** the sweep selects pending `kitovu` transactions older than **2 minutes** (with `meta.kitovu.requested_at` set), cross-checks each via the status endpoint, and confirms the ones that settled. This heals callbacks that were lost or arrived early. It shares the same settle path and strict amount matching as the webhook, so a late callback and the sweep cannot double-credit.

7. Capacity guard

WhatsApp sessions are RAM-heavy (roughly 300–500 MB per active Chromium session). The operator sets `max_concurrent_sessions` (default **15**) in admin settings. The guard is implemented in `server/routes/whatsapp.js`:

- It counts rows in `wa_sessions` with status `pending` or `connected`.

- If the count is below the cap, the request is allowed.
- If the cap is reached, a user is still allowed through **only if their own session is already counted** (so a user mid-pairing can keep polling or reconnecting to their own QR).
- Otherwise the connect/poll route returns `409 at_capacity` with a friendly message. The public API `/api/v1/send` is not gated by this guard — it only requires the user's own session to be `connected`.

8. Core flows

(a) Customer signup → payment → credit top-up

```
Browser ──POST /register (name,email,password,accept_terms)──▶ auth.js
   auth.js: validate, bcrypt hash, INSERT users(accepted_terms_at=NOW())
   establishSession() → session.regenerate() → session.userId
        │
        ▼
Browser ──POST /dashboard/billing/checkout {amount,method}──▶ dashboard.js
   validate $1.00–$500.00 → amountCents
   payments.getProvider('manual').createCheckout({amountCents, credits=amountCents})
        │  INSERT transactions(status=pending, provider_ref='ZN-XXXXXXXX', credits=N)
        ▼
   redirect → GET /dashboard/billing/pay/:id
        │  render manual_payment_instructions with {{reference}} etc.
        ▼
Customer pays offline (bank / mobile money) quoting ZN-XXXXXXXX
        │
        ▼
Admin ──POST /admin/transactions/:id/paid──▶ admin.js → payments.markPaid(id)
   1. UPDATE transactions SET status='paid' WHERE id=? AND status='pending'  (gate)
   2. if credits > 0: credits.addCredits(reason='topup', ref=transaction)
        └─ ledger row (+credits, balance_after) inside a user-row-locked tx
   3. if meta.type='integration': integration_orders requested → paid
   On credit-add failure: roll back to pending for safe retry.
```

(b) QR connect

```
Dashboard ──POST /dashboard/whatsapp/connect──▶ whatsapp.js
   waSessions.ensure(userId)            → row created status=disconnected
   checkCapacity(userId, wa)            → 409 at_capacity if full and not owner
   if wa.status='connected': probe engine; if still connected, short-circuit
   else wpp.startSession('zn_<id>', { webhook: <secret URL>, waitQrCode:true })
        │  engine returns status + qrcode (data URL)
        ▼
   normalizeStatus() → persist pending/connected/error + last_qr_at
   respond { ok, status, qr, phone_number }
        │
        ▼
UI polls GET /dashboard/whatsapp/qr every 3s
   wpp.getStatus() → status-session
   if pending and no QR: wpp.getQrCode() → qrcode-session (PNG/JSON → data URL)
   if connected: fetch get-phone-number, persist phone + connected_at
        │
        ▼
User scans QR in WhatsApp → engine fires status-find webhook
   /webhooks/wppconnect/<secret> → waSessions.updateBySessionName(status=connected)
```

(c) API send pipeline with atomic credit charge + refund

```
Client ──POST /api/v1/send {phone,message,media_url?}──▶ api/v1.js
   requireApiKey: hash(SHA-256) lookup → active user? → req.apiUser/req.apiKey
        │
        ▼  services/send.js performSend()
   1. settings.ensure()  → default_country_code (250), send_rate_per_min (60)
   2. checkRate(rateKey='api:<keyId>')  → 429 rate_limited + retry_after
   3. validateInput()   → phone normalize + message/media checks → 400
   4. user must exist (404) and be active (403)
   5. wa_sessions.status must be 'connected' → 409 whatsapp_not_connected
   6. chargeCredits(userId, amount=1)   ← atomic:
         BEGIN; SELECT id FROM users WHERE id=? FOR UPDATE;
         balance = SUM(delta); if balance < 1 throw INSUFFICIENT_CREDITS;
         INSERT credit_ledger(delta=-1, balance_after); COMMIT
         → 402 insufficient_credits { balance }
   7. engine call: wpp.sendMessage() or wpp.sendMedia() (send-file)
        ├─ success → INSERT messages(status='sent', wpp_message_id, credits_charged=1)
        │            respond { ok, id, message_id, credits_remaining }
        └─ failure → INSERT messages(status='failed', error=<sanitized>, credits_charged=0)
                     addCredits(+1, reason='refund', ref=message)
                     respond 502 { ok:false, error:'send_failed', detail }
```

(d) Delivery webhook status updates

```
Engine ──POST /webhooks/wppconnect/<secret>  (mounted before session + CSRF)
   secretOk(): timing-safe compare of path token or x-znotice-webhook-secret
        │  401 invalid_webhook_secret on mismatch
        ▼
   event routing (routes/webhooks.js):
     onack        → look up messages by wpp_message_id, advance status
     qrcode       → wa_sessions.status='pending', last_qr_at=NOW()
     status-find  → map engine status → connected / disconnected
     closesession / logoutsession → status='disconnected'
     onmessage    → log only, never charge
        │
        ▼
   Ack → status mapping (monotonic; never downgrades):
     1 sent · 2 delivered · 3 read · 4 read · negative = failed
     STATUS_RANK = { queued:0, sent:1, delivered:2, read:3, failed:4 }
   The handler always returns HTTP 200 so webhook processing can never
   fail the engine's delivery.
```

(e) Scheduled & bulk enqueue → dispatch → completion

```
Client ──POST /api/v1/bulk {recipients[], message, scheduled_at?}──▶ api/v1.js
   requireApiKey → schedule.createBulk()
     settings.ensure(): bulk_max_recipients, schedule_max_days_ahead,
                        scheduled_max_pending
     parseScheduledAt(scheduled_at)   → 400 invalid_scheduled_at
                                        (bad / < now+60s / > horizon)
     normalizeRecipients()            → 400 invalid_recipients (any bad entry)
                                        dedupe → duplicates_removed
     resolveTemplate() per recipient  ({{name}}, {{phone}}; unknown tokens kept)
     account active? session connected? balance ≥ total? → 403 / 409 / 402
     BEGIN;
       SELECT id FROM users WHERE id=? FOR UPDATE   ← serializes the cap check
       pending = COUNT(scheduled_messages WHERE status='pending')
       if pending + total > scheduled_max_pending → ROLLBACK 429 queue_full
       INSERT send_batches(status='scheduled', total_count=N)
       INSERT scheduled_messages × N (status='pending')
```

```
    COMMIT
  scheduler.kick() → 202 { batch_id, total, duplicates_removed,
                           scheduled_at, credits_balance }
       │
       ▼  every 30s (services/scheduler.js runOnce; in-flight guard)
  SELECT due pending rows (ROW_NUMBER per user, ≤25/user, global LIMIT 50)
    for each row:
      rate-limit check (user:<id>, send_rate_per_min) → defer +1 min if over
      CLAIM: UPDATE ... SET status='sending' WHERE id=? AND status='pending'
             (0 rows → skip; a cancel can only ever target 'pending')
      executeSend(): account/session checks, media fetch, charge 1 credit,
                     engine call
        ├─ 200  → item 'sent', message_id; batch sent_count++  (atomic)
        ├─ retryable (409 not connected, 502 engine unreachable / HTTP 5xx):
        │        attempts++; release back to 'pending' (+1 min / +5 min);
        │        failed at 3 attempts
        └─ other → item 'failed' (the pipeline already refunded the credit)
      bumpBatch() → status='processing'
      maybeCompleteBatch(emit=true): single conditional UPDATE
        (status IN ('scheduled','processing') AND no open item)
        → 'completed' (or 'cancelled'); the winner emits `batch.completed`
        once, only for a scheduler-driven batch with 0 cancellations
```

Crash recovery: the reaper resets `sending` rows older than **10 minutes** to `failed` / `last_error='interrupted_by_restart'`, because a claim that outlived the process may already have reached WhatsApp. This keeps dispatch **at-most-once** — a crash can drop a message but never duplicate it.

(f) Password reset (Phase 8)

```
Visitor ──POST /forgot-password {email}──► auth.js  (forgotLimiter 5/15m/IP)
  validate address (≤190 chars, email regex); look up active user
  if found:
    DELETE password_resets WHERE user_id=? AND used_at IS NULL   -- invalidate prior
    INSERT password_resets(user_id, token_hash=sha256(raw), expires_at=NOW()+30m)
    email.send(passwordResetEmail(resetUrl))   -- fire-and-forget, never blocks
  always redirect → /forgot-password?sent=1    -- identical for all addresses
       │
       ▼
User clicks  GET /reset-password?token=<raw>  ──► auth.js
  SELECT ... WHERE token_hash=sha256(raw) AND used_at IS NULL AND expires_at>NOW()
  found → render form (valid); else → 400 invalid-link page
       │
       ▼
User submits POST /reset-password {token,password,confirm}  (resetLimiter 10/15m/IP)
  validate password (≥8 chars, ≤72 bytes, confirm match)
  bcrypt.hash(password, 12)
  BEGIN;
    UPDATE password_resets SET used_at=NOW()
      WHERE id=? AND used_at IS NULL AND expires_at>NOW()   -- ATOMIC CLAIM
    affectedRows !== 1 → ROLLBACK + 400 invalid-link (race/expiry loser)
    UPDATE users SET password_hash=?
    DELETE password_resets WHERE user_id=? AND id<>? AND used_at IS NULL
  COMMIT;
  destroyAllSessions(userId)          -- sign out everywhere (exact $.userId match)
  regenerate current session if it belongs to the reset user
  email.send(passwordChangedEmail())  -- fire-and-forget
  flash success → redirect /login
```

`server/services/email.js` is the only outbound-email component. It reads `SMTP_*` from the environment per call, reuses a transport keyed by host/port/ secure/user plus a short non-reversible password fingerprint (so a rotated `SMTP_PASS` rebuilds the transport), never throws, and logs only a sanitized error code. With no `SMTP_HOST` it returns `email_not_configured` and the request still succeeds.

(g) Sender name + dedicated sender-name add-on (Phases 9–11)

```
Dashboard → WhatsApp ──POST /dashboard/billing/sender-name──▶ dashboard.js
   settings.ensure(): sender_name_price_cents (1000), sender_name_days (30)
   BEGIN;
     SELECT id FROM users WHERE id=? FOR UPDATE          -- serialize per user
     existing pending sender_name tx? → ROLLBACK + redirect to it
     INSERT transactions(status='pending', credits=0,
                         provider_ref='ZN-XXXXXXXX',
                         meta={type:'sender_name', days})   -- same tx
   COMMIT → redirect /dashboard/billing/pay/:id
        │
        ▼
Customer pays offline quoting ZN-XXXXXXXX
        │
        ▼
Admin ──POST /admin/transactions/:id/paid──▶ payments.markPaid(id)
   1. UPDATE transactions SET status='paid' WHERE id=? AND status='pending' (gate)
   2. meta.type='sender_name' →
        UPDATE users
           SET sender_name_until =
             DATE_ADD(GREATEST(COALESCE(sender_name_until, NOW()), NOW()),
                      INTERVAL ? DAY)
         WHERE id=?                                     -- days clamped 1..365
      failure → revert tx to 'pending' (paid_at=NULL), rethrow
        │
        ▼
Dashboard → WhatsApp ──POST /dashboard/whatsapp/sender-name──▶ whatsapp.js
   wa_sessions.ensure(userId)
   wa.status !== 'connected'                → flash + redirect (no engine call)
   senderName.isSenderNameActive(userId)    → active? (sender_name_until > NOW())
       inactive → flash + redirect to Billing  -- entitlement gate BEFORE engine
   validateSenderName(req.body.sender_name)  → control chars/line breaks rejected;
                                               whitespace collapsed; 1..25 chars
   wpp.setProfileName('zn_<id>', name)      → POST /api/{session}/change-username
   waSessions.update(userId, { profileName: name })
   flash success → redirect /dashboard/whatsapp
```

The entitlement check (`services/senderName.js`) is shared by the route and the WhatsApp view, so the form and the server enforce the same rule. On the **customer** route the engine is never called when the session is disconnected or the add-on is inactive. Engine failure detail is logged server-side only; the customer sees a generic message. `wa_sessions.profile_name` is display state — the authoritative profile name lives on the WhatsApp account itself.

Admin sender-name override (Phase 11) — deliberate entitlement bypass

```
Admin → Sessions ──POST /admin/sessions/:id/sender-name──▶ admin.js
   parse id (safe integer > 0; else "Invalid session")
   SELECT wa_sessions JOIN users WHERE w.id=?  → 404-style flash if missing
   validateSenderName(req.body.sender_name)    → same rules as the customer route
   session.status must be 'connected'          → flash if not (NO engine call)
   wpp.setProfileName(session.session_name, name)
       engine offline → generic "engine offline" flash
```

```
       other failure  → log detail server-side, generic flash (never raw error)
   waSessions.update(session.user_id, { profileName: name })
```

Unlike `POST /dashboard/whatsapp/sender-name`, the admin route does **not** call `isSenderNameActive` — it is support tooling, so a platform admin can correct/override a tenant's profile name without the tenant owning the paid add-on. This is the **only** sender-name entitlement bypass; the customer route and its paywall are unchanged. The route is behind `requireAdmin` and the global CSRF middleware, and it never surfaces raw engine errors.

(h) Message templates & dashboard prefill (Phase 11)

Templates are a **dashboard-only** convenience: a user saves named message bodies (plus an optional media URL) and prefills the send forms from them. There is no API v1 surface, no background worker and no change to the send pipeline — `{{name}}` / `{{phone}}` placeholders are still resolved at enqueue/dispatch time by the existing scheduler/bulk code.

```
Dashboard → Templates ──POST /dashboard/templates──▶ routes/templates.js
   (mounted BEFORE the /dashboard catch-all; requireAuth + global CSRF)
   validateTemplate(name ≤80, body ≤4096, media http(s) ≤500, no control chars)
   createForUser(userId, tpl):
     BEGIN; SELECT id FROM users WHERE id=? FOR UPDATE;      -- serialize per user
            SELECT COUNT(*) FROM message_templates
              WHERE user_id=? FOR UPDATE;                    -- lock the scanned rows
            count >= templates_max_per_user → ROLLBACK limit_reached
            INSERT message_templates(...)  (ER_DUP_ENTRY → duplicate_name)
     COMMIT
         │
         ▼
Prefill (server-side, no JS)
   GET /dashboard/whatsapp?template=<id>  ─▶ resolvePrefill(userId, id)
   GET /dashboard/scheduled?template=<id> ─▶ resolvePrefill(userId, id)
     parseTemplateId: digits only, safe integer > 0 (else null)
     SELECT body, media_url WHERE id=? AND user_id=? LIMIT 1
       owned → { body, mediaUrl } rendered into the form
       foreign / invalid / missing → null (no information leak; blank form)
```

`routes/templates.js` also handles `POST /:id` (update) and `POST /:id/delete` (delete); both enforce ownership in the SQL `WHERE` clause, so a foreign id affects 0 rows and flashes "Template not found". The list is bounded by `listForUser(userId, 500)`. The per-user cap is read from the `templates_max_per_user` setting (default 200) inside the create transaction, so concurrent creates for one user cannot overrun it.

(i) Payment proof uploads (Phase 12)

A manual-payment proof is a screenshot/receipt a customer attaches to a `pending` transaction from the pay page. Metadata is written to `payment_proofs`; the bytes are written to disk under `<UPLOADS_DIR>/payment-proofs/`. There is no API v1 surface and no background worker. Types are JPG/PNG/WebP/PDF, max **5 MB** each, max **5 per transaction** and max **25 files / 50 MB per user**.

Middleware order in `server/index.js` (this is load-bearing — see below):

```
app.use('/dashboard/billing/proof', proofUploadLimiter, multerSingle)
        │  proofUploadLimiter: 30 POSTs / 15 min, keyed by `user:<id>` else
        │  `ip:<ip>`; non-POST requests skip. Over limit → error flash + 302
        │  to /dashboard/billing/pay/:id.
```

```
        │  multerSingle = uploadProof.single('proof') (server/middleware/
        │  uploadProof.js): memoryStorage, limits { fileSize: 5 MB, files: 1,
        │  fields: 1, fieldSize: 1024, parts: 3 }. Multipart only; other
        │  content types pass straight through to the routers.
        │  Parses the hidden `_csrf` field into req.body — nothing hits disk.
        ▼
app.use(verifyCsrf)                    ← global CSRF check now sees req.body._csrf
        ▼
app.use('/dashboard', requirePayment)  ← locked users may still upload (billing path)
        ▼
routes/dashboard.js  POST /billing/proof/:id
   validateProofBuffer(req.file.buffer) → magic-byte sniff + ≤5 MB
   safeOriginalName(req.file.originalname) / makeStoredName(mime) → <16hex>.<ext>
   BEGIN;
     SELECT ... FROM transactions WHERE id=? FOR UPDATE      -- row lock
       not owned / not pending → 404 or flash
     SELECT COUNT(*) FROM payment_proofs WHERE transaction_id=? FOR UPDATE
       assertProofLimit(count)                               -- ≤5 per transaction
     getUserProofUsage(userId, conn) → assertUserProofQuota  -- ≤25 / ≤50 MB
     INSERT payment_proofs(...)
   COMMIT; release
   writeProof()                          ← file write OUTSIDE the lock (H4)
     failure → best-effort delete the committed row + unlink partial file
   flash success → redirect /dashboard/billing/pay/:id
```

Why multer precedes CSRF. The global `verifyCsrf` reads `req.body._csrf`, but `express.urlencoded` cannot parse a `multipart/form-data` body. Parsing the upload first (into memory) populates `req.body` so the hidden `_csrf` field is visible and the normal synchronizer-token check still applies. The rate limiter runs before multer so an unauthenticated/abusive client cannot make the server buffer files, and because the storage is `memoryStorage`, nothing touches disk until the route has validated CSRF, ownership, transaction status, size and magic bytes. The DB row is committed before the file write (H4) so the transaction row lock is never held across disk I/O; serving returns 404 during the brief row-exists-file-missing window.

Proof routes:

Route	Auth	Behaviour
`POST /dashboard/billing/proof/:id`	owner (`requireAuth`)	Upload one proof while the transaction is `pending`.
`GET /dashboard/billing/proof/:id/:proofId`	owner	Stream the file; foreign/missing → `404`.
`POST /dashboard/billing/proof/:id/:proofId/delete`	owner	Delete while `pending`; ownership + status re-asserted in the `DELETE`.
`GET /admin/transactions/:id/proof/:proofId`	`requireAdmin`	Admin stream of any proof; missing → `404`.

Serving headers (both owner and admin routes; MIME comes from the DB row, never the client):

```
Content-Type: <sniffed mime>
Content-Disposition: inline; filename="proof-<proofId>.<ext>"
Cache-Control: private, no-store
Content-Security-Policy: default-src 'none'; sandbox
```

Proofs remain readable by the owner and admins after `paid` / `failed` for audit; upload/delete are restricted to `pending`.

Storage layout and permissions (`server/services/proofs.js`):

```
<UPLOADS_DIR>/payment-proofs/<transaction_id>/<16hex>.<ext>
```

- `UPLOADS_DIR` (env) overrides `config.paths.uploads`; the default is `<app root>/uploads`. `config.paths.uploads` is read at call time, so an override (e.g. in tests) is honoured.
- Directories are created `0o750`, files written `0o640`.
- `uploads/` is **outside** `public/`, so proof files are only reachable through the authenticated Node routes above.
- Stored filenames are always generated (`crypto.randomBytes(8)` → `<16 hex>.<ext>`) and re-validated against `^[0-9a-f]{16}\.(jpg|png|webp|pdf)$` before any path join, so a user-controlled name can never traverse or select a path.

(j) Billing lock for unpaid users (Phase 13)

A non-admin account with no `paid` transaction and no positive credit balance is confined to the payment flow. The gate is mounted at `/dashboard` immediately after `verifyCsrf` and before every dashboard router:

```
app.use(verifyCsrf);
app.use('/dashboard', requirePayment);          // middleware/paymentLock.js
app.use('/dashboard/account', accountRouter);   // allowed for locked users
app.use('/dashboard/webhooks', ...);            // blocked for locked users
... every other /dashboard/* router ...
app.use('/dashboard', dashboardRouter);         // /billing/* lives here
app.use('/admin', adminRouter);                 // unaffected
```

`requirePayment` (`server/middleware/paymentLock.js`) delegates the decision to `server/services/billingAccess.js`:

- **Unlock rule.** An account is unlocked when it is an `admin`, **or** has `EXISTS(SELECT 1 FROM transactions WHERE user_id=? AND status='paid')`, **or** `COALESCE(SUM(credit_ledger.delta), 0) > 0`. Pending/failed transactions alone do not unlock. `getBillingAccess(userId)` resolves this in one query and looks up the most recent `pending` transaction only when locked.
- **Anonymous** requests pass through; the downstream `requireAuth` owns the login redirect (a single auth gate).
- **Admins** always pass (`res.locals.billingLocked = false`).
- **Locked + allowed path** passes with `res.locals.billingLocked = true`.
- **Locked + any other path** sets an error flash and `302`s to `/dashboard/billing/pay/<pendingTransactionId>` (most recent pending transaction) or `/dashboard/billing` when there is none.
- **Allowed prefixes** (relative to the `/dashboard` mount): `/billing`, `/account`, `/integration`. `/integration` is included because the integration request form (`POST /dashboard/integration/request`) creates a pending $20 payment and is shown on the Billing page (hardening H3). `isAllowedBillingPath` is pure and defensive: non-string/empty input is never allowed, so malformed input can only lock down.

`res.locals.billingLocked` is consumed by `partials/dashnav.ejs` (renders only the Billing + Account tabs plus a "Locked — payment required" chip) and by `dashboard/billing.ejs` (amber "your account is locked" banner). `/api/v1/*` is **intentionally not gated**: it is API-key authenticated and a zero balance already fails with `402 insufficient_credits`. `/admin/*` is unaffected.

Admin proof visibility. The admin overview pending-payments query (`routes/admin.js`) returns `proof_count` via a correlated subquery and `views/admin/index.ejs` renders an "N proof(s)" badge (or "No proof"), so an admin can spot transactions with uploaded proof before approving. `GET /admin/transactions` attaches each transaction's proofs in one extra query (no N+1) and the table renders a "Proofs (N)" column linking each file.

(k) IntouchPay instant credit top-up (Phase 14)

Credit top-ups only — the sender-name add-on and integration service remain manual. The customer pays by approving a USSD prompt on their phone; the transaction is auto-confirmed, with no admin approval or proof upload.

```
Browser ──POST /dashboard/billing/intouch/checkout {amount,phone}──▶ dashboard.js
   validate $1.00–$500.00 → amountCents
   convert amountCents → whole RWF at intouch_rwf_rate (settings.getFloat())
   normalize phone → 2507XXXXXXXX
   payments.getProvider('intouch').createCheckout({...})
        │  INSERT transactions(status=pending, provider='intouch',
        │    provider_ref='ZNI-<id>-<8hex>', amount_cents=USD,
        │    meta={currency:'RWF', amount_rwf, rate, intouch:{phone}})
        ▼
   POST /requestpayment/  →  tx_id (async, Pending 1000)
        │  store meta.intouch.tx_id
        ▼
   redirect → GET /dashboard/billing/pay/:id  (Intouch branch, no proof UI)
        │
        ▼
Customer approves the USSD prompt on their MTN MoMo / Airtel Money phone
        │
        ├─(path 1) InTouch POSTs /payments/intouch/webhook (unsigned, jsonpayload)
        │     webhook: unwrap → cross-check gettransactionstatus (3 retries)
        │              → settled + exact RWF/currency match? settlePaid()
        │              → always ACK 200 {"message":"success",...}
        │
        └─(path 2) intouchSweep (every 5m) cross-checks pending rows > 2m old
             → confirm settled ones (heals lost/early callbacks)

   settlePaid(): pending → paid (atomic gate) → credits.addCredits(note override)
        │  credit_ledger (+amount_cents, reason=topup)
        ▼
Browser ──POST /dashboard/billing/intouch/:id/status──▶ manual cross-check
   (optional) re-checks status; shows "auto-confirmed by InTouch" once paid
```

Key properties:

- **Auto-confirmation only.** An Intouch transaction never goes through the manual Approve/Reject buttons — the admin table shows a provider chip and a muted "auto-confirmed by InTouch" hint. The only trust anchor is the `gettransactionstatus` cross-check, never the unsigned callback.
- **Strict matching.** Settlement requires the gateway's returned status to be settled and the RWF amount + currency to equal the transaction's `meta` exactly (whole RWF, off-by-one blocked).
- **Re-request.** `POST /dashboard/billing/intouch/:id/start` re-sends the USSD prompt for a still-pending transaction.

- **No refunds via the processor.** Intouch has no reversal endpoint, so refunds stay local-ledger only (an admin credit adjustment), exactly like `manual`.
- **Idempotency.** The same conditional `pending → paid` gate as `manual` applies, so webhook + sweep racing a single settlement cannot double-credit.

(l) KitovuPay instant credit top-up (Phase 15)

Credit top-ups only — the sender-name add-on and integration service remain manual. The customer pays by approving an MTN MoMo prompt on their phone; the transaction is auto-confirmed, with no admin approval or proof upload.

```
Browser ──POST /dashboard/billing/kitovu/checkout {amount,phone,name?}──▶ dashboard.js
   validate $1.00–$500.00 → amountCents
   convert amountCents → whole RWF at kitovu_rwf_rate (settings.getFloat())
   normalize phone → 2507XXXXXXXX
   payments.getProvider('kitovu').createCheckout({...})
        │  INSERT transactions(status=pending, provider='kitovu',
        │    provider_ref=NULL, amount_cents=USD,
        │    meta={currency:'RWF', amount_rwf, rate, kitovu:{phone:null,...}})
        ▼
   requestForTransaction: allocate unique 6-digit externalId
        │  (checked in transactions.provider_ref; retry on DUPLICATE_EXTERNAL_ID)
   POST /merchant-transaction/merchant/collection/mobile_money (signed)
        │  store provider_ref=externalId, meta.kitovu.{phone,tx_id,status,requested_at}
        ▼
   redirect → GET /dashboard/billing/pay/:id  (Kitovu branch, no proof UI)
        │
        ▼
Customer approves the MTN MoMo prompt on their phone
        │
        ├─(path 1) KitovuPay POSTs /payments/kitovu/webhook (unsigned)
        │     webhook: cross-check GET /merchant-transaction/payment/:externalId (3 retries)
        │               → APPROVED (+ exact RWF amount when present)? settlePaid()
        │               → always ACK 200 {"message":"success",...}
        │
        └─(path 2) kitovuSweep (every 5m) cross-checks pending rows > 2m old
              → confirm settled ones (heals lost/early callbacks)

   settlePaid(): pending → paid (atomic gate) → credits.addCredits(note override)
        │  credit_ledger (+amount_cents, reason=topup, note='kitovu top-up verified')
        ▼
Browser ──POST /dashboard/billing/kitovu/:id/status──▶ manual cross-check
   (optional) re-checks status; shows "auto-confirmed by KitovuPay" once paid
```

Key properties:

- **Auto-confirmation only.** A Kitovu transaction never goes through the manual Approve/Reject buttons — the admin table shows a provider chip and a muted "auto-confirmed by KitovuPay" hint. The only trust anchor is the status cross-check, never the unsigned callback.
- **Strict matching.** Settlement requires the gateway's status to be `APPROVED`; a webhook-supplied amount must equal the transaction's `meta` exactly (whole RWF, off-by-one blocked).
- **Re-request.** `POST /dashboard/billing/kitovu/:id/start` re-sends the prompt for a still-pending transaction.
- **No refunds via the processor.** KitovuPay has no reversal endpoint in scope, so refunds stay local-ledger only (an admin credit adjustment), exactly like `manual`.
- **Idempotency.** The same conditional `pending → paid` gate as `manual` applies, so webhook + sweep racing a single settlement cannot double-credit.

9. Usage analytics (Phase 7)

Analytics is a **read-only aggregation layer** over the existing `messages`, `credit_ledger`, `transactions` and `users` tables. It performs no writes, keeps no state, and has no background worker. All queries are parameterized and all day buckets are UTC.

Components

Piece	Location	Notes
Service	`server/services/analytics.js`	`customerDaily`, `customerSummary`, `adminDaily`, `adminSummary`, `topUsers`. Coerces MySQL string aggregates to numbers and zero-fills each window to exactly `days` points.
Customer route	`server/routes/analytics.js`	Mounted at `/dashboard/analytics` (before the `/dashboard` catch-all), `requireAuth`, fixed 30-day window.
Admin route	`server/routes/admin.js` (`GET /admin/analytics`)	Behind `requireAdmin`, fixed 30-day window.
API route	`server/routes/api/v1.js` (`GET /api/v1/usage`)	API-key auth; `?days=` default 30, clamped 1–90.
Views	`server/views/dashboard/analytics.ejs`, `server/views/admin/analytics.ejs`	Summary cards, charts, funnel, tables and empty states.
Chart partial	`server/views/partials/svg-bars.ejs`	Reusable server-rendered inline SVG bar chart (grouped bars when a second series is present).

Metric semantics

- **Window:** a `days`-long window starts at `UTC_DATE() - INTERVAL (days-1) DAY` (inclusive), so it ends on today and always yields exactly `days` points.
- **Completed statuses:** `sent`, `delivered`, `read`, `failed`. `queued` messages are excluded from every customer count and from every `credits_spent` sum (`credits_spent = SUM(credits_charged)` over completed statuses only).
- **Admin `messages` / `active_users`:** count **all** message rows (including `queued`); `active_users` is `COUNT(DISTINCT user_id)`.
- **Revenue:** `revenue_cents` and `credits_sold` come from `transactions WHERE status='paid'`, grouped by `DATE(paid_at)`.
- **Point-in-time figures:** `credits_outstanding = SUM(delta)` over the whole ledger and `pending_payments = COUNT(*)` of pending transactions are **not** windowed.
- **Failure reasons:** the customer page lists up to 5 error strings for `failed` messages in the last 30 days; blank/NULL errors are normalized to `unknown`.

Charts are CSP-safe

The charts are plain inline SVG generated server-side by `partials/svg-bars.ejs` using **presentation attributes only** — no inline `<script>`, no `style` attributes and no third-party chart library. This is what keeps the strict Content-Security-Policy (`script-src 'self'`, `style-src-attr 'none'`) intact while still rendering interactive tooltips via SVG `<title>` elements.

Indexes

Migration `server/migrations/007_analytics_indexes.sql` adds two indexes (both idempotent and guarded via `information_schema`, so the file is safe to re-run):

- `messages(created_at)` (`idx_messages_created`) — platform-wide date-range scans (admin analytics).
- `transactions(status, paid_at)` (`idx_tx_status_paid`) — paid-revenue scans.

Customer-scoped queries use the pre-existing `messages(user_id, created_at)` index. Because the migration is index-only, it changes no data.

Input hardening

`days` is clamped to `1..90` both at the HTTP routes and again inside the service (`clampDays`), and `topUsers` clamps `limit` to `1..50`. This is defense in depth: a direct service call can never produce an `INTERVAL NaN DAY` SQL error or allocate an unbounded day-key array.

10. Test suite layout

The suite uses Node's built-in `node:test` runner — there are **no** new test dependencies.

```
test/
  unit/                       # no database; pure-function and in-memory coverage
    send.test.js              # normalizePhone + validateInput
    schedule.test.js          # parseScheduledAt / parseRecipientLines / resolveTemplate
    webhook.test.js           # webhookOut.sign (HMAC vectors) + worker surface
    netguard.test.js          # netGuard URL/address classification
    analytics.test.js         # clampDays + read-only service surface
    safeNext.test.js          # open-redirect defenses
    senderName.test.js        # validateSenderName (trim/collapse/control/1-25)
    templates.test.js         # validateTemplate + parseTemplateId + service surface
    proofs.test.js            # sniffMime + proof path/limit helpers (Phase 12)
    billingAccess.test.js     # isAllowedBillingPath + billingRedirectTarget (Phase 13)
    intouch.test.js           # auth digest, RWF math, phone normalize, webhook ACK (Phase 14)
    kitovu.test.js            # signature vector, RWF math, phone normalize, webhook parse (Phase
15)
  integration/                # DB-backed; run against an isolated _test database
    helpers/db.js             # MUST be required first: safety guard + seeds + lock
    credits.test.js           # ledger atomicity / row-lock serialization
    schedule.test.js          # createScheduled / createBulk / pending cap
    scheduler.test.js         # dispatch, retry/backoff, completion
    passwordReset.test.js     # reset lifecycle, single-use race, session destroy
    reaper.test.js            # retention + session pruning selectivity
    senderName.test.js        # engine call + profile_name persistence/failure
    senderNameAddon.test.js   # settlement/renewal, days clamp, rollback, gate, checkout race
    templates.test.js         # CRUD scoping, per-user cap race, duplicate name, prefill
    adminSenderName.test.js   # admin override: bypass, connected gate, persistence
    paymentProofs.test.js     # proof CRUD, caps/quota, admin proof route, multipart CSRF
    billingAccess.test.js     # unlock rule (paid tx / positive balance) over the DB
    paymentLock.test.js       # lock redirect, allowed paths, admin/anonymous bypass
    intouch.test.js           # checkout → webhook → cross-check → settle, sweep healing (Phase
14)
    kitovu.test.js            # checkout → signed request → webhook → settle, sweep (Phase 15)
```

Commands (from the app root):

Command	Scope
`npm test`	164 unit tests, no DB
`npm run test:integration`	174 DB-backed tests
`npm run test:all`	both

`test/integration/helpers/db.js` refuses to run against production: it requires `NODE_ENV !== 'production'`, a `TEST_DB_NAME` ending in `_test`, and a test name different from the `.env` `DB_NAME`. It forces `NODE_ENV=test` and `DB_NAME=$TEST_DB_NAME` before `server/config.js` is loaded, applies the migrations, truncates business tables between tests, and holds a MariaDB advisory lock (`GET_LOCK('znotice_test_suite_lock')`) for each file's lifetime so concurrent `node --test` workers cannot truncate each other's data. See OPERATIONS.md §13 for the one-time test-DB setup.