

Zapier — Developer Documentation

Version: 0.2.0 · **Last updated:** 2026-07-28

Metered API platform: per-endpoint pricing, per-tier multipliers, usage ledger, Stripe metered billing, purchase-order invoicing, a company admin console, a customer portal with TOTP 2FA, and a Zapier integration. This document is the full technical reference. For operations, see `USER-MANUAL.md`.

Table of contents

1. Architecture
 2. Module reference
 3. Pricing engine
 4. Request lifecycle
 5. Persistence (SQLite)
 6. Billing pipeline
 7. API reference
 8. Zapier app
 9. Testing
 10. Configuration
 11. Extending the system
-

1. Architecture

Stack: Node 20 · TypeScript (strict) · Express 4 · better-sqlite3 · express-openapi-validator · swagger-ui-express · stripe SDK · Zapier Platform (core ^19) · jest + supertest (root) and mocha (zapier-app).

Express app (src/app.ts)		
Browser	/admin	static admin UI (admin/)
Browser	/admin/api	adminAuth → adminRouter (src/admin.ts) pricing · customers · invoices · reports · zapier status
Browser	/portal	static portal SPA (portal/)
Browser	/portal/api	portalRouter (src/portal.ts) signup/login+2FA · me · usage · invoices · reload · prefs
Browser	/docs	swagger-ui (openapi.yaml)
Client	/v1/*	apiKeyAuth (src/auth.ts) OpenAPI request validation meter() (src/meter.ts) quoteCall() (src/pricing.ts) UsageRepo.record()
SqliteUsageRepo (usage_entries)	SqliteCustomerRepo (customers)	SqlitePricingStore (price_endpoints, invoices,

```

                                tiers)                invoice_lines)
        SqliteSessionRepo (portal_sessions)

                                zappier.db (SQLite)

Cron (daily 06:17 ET)
"Zappier billing · report usage"

src/jobs/report-usage.ts  reportMonthlyUsage()
    computeBillableCents / computeDelta (src/billing/stripe.ts)
    SqliteBillingReportRepo (billing_reports, job_locks)
    stripe.billing.meterEvents.create    Stripe

```

Design rules the codebase follows:

- **Ports & adapters:** `UsageRepo`, `CustomerRepo`, `PricingStore`, `MeterEventClient` are interfaces with in-memory adapters (tests) and SQLite / Stripe adapters (production). Nothing outside `src/db/` and the job's `main()` touches SQL or the Stripe SDK.
- **Live pricing reads:** the `PricingContext` getters in `buildApp` read the store on every quote, so admin edits apply without a restart.
- **Money in integer cents everywhere.** No floats cross a boundary except multipliers, which are applied once and rounded (`Math.round`).

2. Module reference

File	Responsibility	Key exports
<code>src/index.ts</code>	Production entry: opens SQLite, wires repos, listens	—
<code>src/app.ts</code>	<code>buildApp(deps)</code> — full Express wiring; <code>StoredItem</code> ; <code>DEFAULT_CUSTOMERS</code>	<code>buildApp</code> , <code>AppDeps</code>
<code>src/auth.ts</code>	Customer model + x-api-key middleware	<code>Customer</code> , <code>CustomerRepo</code> , <code>apiKeyAuth</code> , <code>InMemoryCustomerRepo</code>
<code>src/pricing.ts</code>	Pricing domain: rules, tiers, quote algorithm, store port	<code>PriceRule</code> , <code>TierConfig</code> , <code>Quote</code> , <code>quoteCall</code> , <code>PricingStore</code> , <code>DEFAULT_RATE_CARD</code> , <code>DEFAULT_TIERS</code>
<code>src/meter.ts</code>	Per-request metering middleware	<code>meter(endpointId, repo, pricing)</code>
<code>src/usage.ts</code>	Usage ledger domain + summaries	<code>UsageEntry</code> , <code>UsageSummary</code> , <code>summarize</code> , <code>UsageRepo</code> , <code>InMemoryUsageRepo</code>
<code>src/admin.ts</code>	Admin API (<code>/admin/api</code>) + admin-key guard	<code>adminAuth</code> , <code>adminRouter</code>
<code>src/billing/credit.ts</code>	Monthly credit application	<code>applyMonthlyCredit</code> , <code>BilledSummary</code>
<code>src/billing/stripe.ts</code>	Stripe-facing math + client port	<code>METER_EVENT_NAME</code> , <code>MeterEventClient</code> , <code>computeBillableCents</code> , <code>computeDelta</code> , <code>reportUsage</code>
<code>src/billing/reload.ts</code>	Portal reloads via Stripe <code>PaymentIntents</code>	<code>stripePaymentClient</code> , <code>hasRealStripeKey</code>

File	Responsibility	Key exports
src/accounts.ts	Portal identity: script passwords, RFC 6238 TOTP, sessions	hashPassword, verifyPassword, totp, verifyTotp, generateTotpSecret, totpUri, SessionRepo, InMemorySessionRepo
src/invoicing.ts	Invoice domain + generation	Invoice, InvoiceLine, buildInvoice, InvoiceRepo, InMemoryInvoiceRepo
src/reports.ts	Billing/usage aggregation + CSV	billingRows, usageTrend, toCsv, BillingRow, TrendPoint
src/portal.ts	Customer portal API (/portal/api)	portalRouter, PortalDeps, PaymentClient, PaymentResult
src/paths.ts	Installation-root resolution (cwd-independent)	PROJECT_ROOT
src/jobs/report-usage.ts	Billing job: delta reporting to Stripe	reportMonthlyUsage, firstOfMonthUtc, ReportUsageDeps
src/db/usage-repo.ts	SQLite adapter: usage_entries	SqliteUsageRepo
src/db/customer-repo.ts	SQLite adapter: customers (seeds when empty; idempotent column migrations)	SqliteCustomerRepo
src/db/pricing-store.ts	SQLite adapter: price_endpoints, tiers (seeds when empty)	SqlitePricingStore
src/db/billing-repo.ts	SQLite adapter: billing_reports, job_locks	SqliteBillingReportRepo, BillingReportRepo, JobLockRepo
src/db/invoice-repo.ts	SQLite adapter: invoices, invoice_lines	SqliteInvoiceRepo
src/db/session-repo.ts	SQLite adapter: portal_sessions	SqliteSessionRepo
openapi.yaml	Public API contract; drives validation and /docs	—
admin/	Dependency-free admin SPA (index.html, app.js)	—
portal/	Dependency-free customer portal SPA (index.html, app.js)	—

3. Pricing engine

Types (src/pricing.ts)

```

type PriceRule =
  | { kind: 'free' }
  | { kind: 'fixed'; fixedCents: number }
  | { kind: 'variable'; baseCents: number; perKbCents: number; perMbCents: number };

interface TierConfig {
  id: string; name: string;
  multiplier: number;           // e.g. 0.5 = 50% of list
  monthlyCreditCents: number;   // free included usage per month
  defaultRule?: PriceRule;      // fallback for endpoints with no rule

```

```

}

interface Quote {
  endpointId: string;
  listCents: number;           // before multiplier
  totalCents: number;         // after multiplier - this is what is recorded
  breakdown: { baseCents: number; metadataCents: number; attachmentCents: number };
}

quoteCall(pricing, tierId, endpointId, usage, multiplierOverride?)

1. Resolve tier — throws Unknown tier: <id> (mapped to 403 by meter).
2. Resolve rule: rate-card rule for endpointId, else the tier's defaultRule, else throw No price rule
   for <tier>/<endpoint> (403).
3. Compute breakdown:
   • free → all zeros.
   • fixed → baseCents = fixedCents.
   • variable → baseCents + perKbCents × ceil(metadataBytes/1024) + perMbCents ×
     ceil(attachmentBytes/1048576). Note the ceiling: 1 byte of metadata bills a full KB unit;
     attachments bill per started MB.
4. listCents = sum of breakdown; multiplier = multiplierOverride ?? tier.multiplier;
   totalCents = Math.round(listCents × multiplier).

```

`totalCents` (never `listCents`) is what the usage ledger records and what the billing pipeline sums.

4. Request lifecycle

For POST `/v1/storage`:

- `express.json()` parses JSON bodies (multipart handled by the validator's `multer` — 25 MB per-file cap).
- `apiKeyAuth` (`src/auth.ts`) — `x-api-key` → `Customer` on `req.customer`, else 401.
- `express-openapi-validator` checks the request against `openapi.yaml` (400 on violation). For `/v1/storage`, `parseMetadata` then JSON-parses the `metadata` form field into `res.locals.parsedMetadata` (400 on bad JSON).
- `meter('storage', usage, pricing)` (`src/meter.ts`):
 - measures `metadataBytes` (UTF-8 length of the JSON-stringified metadata) and `attachmentBytes` (sum of `multer` file sizes),
 - calls `quoteCall` — pricing errors become 403,
 - records a `UsageEntry` with `cents = quote.totalCents`,
 - stashes the quote in `res.locals.quote`.
- The route handler builds the `StoredItem` (in-memory list) and responds `{ id, quote }`.

GET `/v1/usage` is **not** metered; it summarizes the caller's month-to-date usage and applies the tier credit via `applyMonthlyCredit`.

5. Persistence (SQLite)

Single database file (`ZAPPIER_DB`, default `zappier.db`), WAL-agnostic `better-sqlite3`, all tables created with `CREATE TABLE IF NOT EXISTS` in the repo constructors. **Seeding rule:** `customers` and pricing tables seed from `DEFAULT_CUSTOMERS` / `DEFAULT_RATE_CARD` / `DEFAULT_TIERS` only when empty.

Table	Columns	Written by
usage_entries	id, customer_id, endpoint_id, cents, metadata_bytes, attachment_bytes, timestamp_ms	meter() on every priced call
customers	id PK, name, tier_id, api_key UNIQUE, stripe_customer_id, multiplier_override, billing_type, email, password_hash, totp_secret, totp_enabled, balance_cents, email_invoicing	Admin API, portal API
price_endpoints	endpoint_id PK, rule_json	Admin API
tiers	id PK, name, multiplier, monthly_credit_cents, default_rule_json	Admin API
invoices	id PK, customer_id, period, status, cents totals, billing_type, po_number, lifecycle timestamps	Admin API (generate/issue/paid + balance drawdown)
invoice_lines	invoice_id, endpoint_id, calls, cents	Invoice generation
portal_sessions	token PK, customer_id, created_ms, expires_ms	Portal auth
billing_reports	customer_id + period PK, reported_cents (cumulative), reported_at_ms	Billing job, after each successful meter event
job_locks	name PK, acquired_at_ms	Billing job run guard

New customer columns are added by **idempotent migrations** (`PRAGMA table_info` guard + `ALTER TABLE ADD COLUMN`) when the repo opens an older database — no manual migration step.

In tests, every repo is constructed over `:memory:` databases.

6. Billing pipeline

6.1 Math (`src/billing/stripe.ts`)

```
computeBillableCents(entries, monthlyCreditCents)
  = max(0,  $\Sigma$  entry.cents - monthlyCreditCents)
```

```
computeDelta(billable, previouslyReported)
  = max(0, billable - previouslyReported)
```

`reportUsage(client, stripeCustomerId, entries, monthlyCreditCents)` is the original whole-month reporter — **retained as public API**; the job uses the delta path instead.

6.2 The job (`src/jobs/report-usage.ts`)

`reportMonthlyUsage(deps)` per run:

1. **Lock:** `locks.tryAcquireLock('report-usage', 1h TTL)` — a single atomic `INSERT ... ON CONFLICT ... DO UPDATE ... WHERE acquired_at_ms <= now - ttl`. Failure aborts the run; release happens in `finally`. A crashed run's lock is taken over after the TTL.

2. **Window:** `since = firstOfMonthUtc(now)` (injectable via `deps.since` for tests); `period = since.toISOString().slice(0, 7)` (YYYY-MM).
3. Per customer with a `stripeCustomerId` (others skipped silently; unknown `tierId` warns and skips):
 - `entries = usage.listFor(customer.id, since)`
 - `billable = computeBillableCents(entries, tier.monthlyCreditCents)`
 - `prior = billingRepo.getReportedCents(customer.id, period)` (0 for a new month — periods are isolated by the composite PK)
 - `delta = computeDelta(billable, prior); delta <= 0 → log skip, continue`
 - `createMeterEvent({ eventName: 'zappier.api_cents', customerId: stripeCustomerId, value: String(delta), identifier: stripeCustomerId : {period} : {billable}})`
 - **only on success:** `upsertReportedCents(customer.id, period, billable)` — cumulative, not the delta.

Idempotency guarantees (reviewed design):

- *Re-run safety:* second run with same usage → delta 0 → no Stripe call.
- *Mid-month growth:* only the increase is sent; the identifier embeds the new cumulative billable, so legitimate growth is never deduped away.
- *Crash between Stripe success and ledger write:* retry sends a byte-identical event; Stripe drops it via the **identifier** (uniqueness enforced within a rolling 24 h window).
- *Partial failure:* customer A's ledger write commits before customer B is attempted; B's failure leaves A correctly recorded.

CLI: `npx ts-node src/jobs/report-usage.ts` (guarded by `require.main === module`; loads `.env` via `dotenv` inside `main()`). Scheduled by the Kimi cron job “Zappier billing · report usage to Stripe” (17 6 * * *, America/New_York), which runs this command daily and reports the outcome.

7. API reference

Public API (/v1, auth: x-api-key)

Defined in `openapi.yaml`; interactive docs at `/docs`.

Operation	Method & path	Price	Notes
<code>status</code>	GET <code>/v1/status</code>	free	Health + quote echo
<code>transform</code>	POST <code>/v1/transform</code>	fixed	<code>{text}</code> → <code>{output: TEXT, quote}</code>
<code>storage</code>	POST <code>/v1/storage</code>	variable	multipart: <code>metadata</code> (JSON string), <code>attachments[]</code> (25 MB/file) → <code>{id, quote}</code>
<code>storage-list</code>	GET <code>/v1/storage</code>	free	Caller's stored items
<code>usage</code>	GET <code>/v1/usage</code>	unmetered	Month-to-date summary with credit applied

Error envelope: `{ "error": string }` with 400 (validation/metadata), 401 (bad key), 403 (unknown tier / no price rule), 413 (file over 25 MB).

Admin API (/admin/api, auth: x-admin-key or login session)

Route	Purpose
POST /login	{username, password} → session token (accounts: ADMIN_USER/ADMIN_KEY, DEMO_ADMIN_USER/DEMO_ADMIN_PASSWORD)
GET /pricing	{ rateCard, tiers }
PUT /endpoints/:id	Upsert a PriceRule (validated: 400 on bad shape)
DELETE /endpoints/:id	Remove a rule (endpoint becomes 403 for tiers without a default rule)
PUT /tiers/:id	Upsert a TierConfig
DELETE /tiers/:id	Remove a tier
GET /customers	List customers without API keys
POST /customers	Create {name, tierId} → full customer incl. generated apiKey (201, shown once)
PUT /customers/:id	Patch name / tierId / multiplierOverride / stripeCustomerId / billingType / email
POST /invoices/generate	{period, customerId?, poNumber?} — drafts per customer with usage; regenerating replaces drafts, skips issued/paid → {generated, skipped}
GET /invoices	Filters: customerId, period, status
GET /invoices/:id	JSON, or print-ready HTML with ?format=html
POST /invoices/:id/issue	draft → issued (PO gets 30-day due date). Prepaid drawdown: if the customer's balanceCents fully covers billableCents, the balance is deducted and the invoice is saved as paid instead
POST /invoices/:id/paid	issued → paid
GET /reports/billing	from/to/customerId/billingType filters; JSON rows or format=csv
GET /reports/usage-trend	bucket=day week, same range filters
GET /zapier/status	Zapier app dir presence, version, triggers, creates

Portal API (/portal/api, auth: Bearer session)

Route	Purpose
POST /signup	{name, email, password 8} → 201 {token, customer}. New customer on free with instant API key; an email match on a passwordless (admin-created) customer claims that account; 409 when the email already has a password
POST /login	{email, password, totpCode?} → {token, customer}; 401 totp_required when 2FA is on and the code is missing/wrong
POST /logout	Deletes the session
GET /me	Public profile — never includes passwordHash/totpSecret
POST /api-key	Regenerates the API key (old key dies immediately)
GET /usage	Month-to-date summary with tier credit applied
GET /pricing	Live rate card + tiers for the pricing page
GET /invoices · GET /invoices/:id	Own invoices only (others 404); ?format=html print view
POST /2fa/setup	Generates + stores a TOTP secret (not yet enabled) → {secret, uri, qr} (QR as PNG data URL via qrcode)

Route	Purpose
POST /2fa/enable · POST /2fa/disable POST /reload	{code} verified against the stored secret {amountCents} integer \$1–\$10,000 via the injected <code>PaymentClient</code> — dev client credits instantly; Stripe client returns a <code>clientSecret</code> and credits on confirmation
PUT /email-invoicing	{enabled} preference

Sessions live in `portal_sessions` (7-day TTL) and survive restarts. `src/accounts.ts` implements scrypt hashing (`scrypt:N:r:p:salt:hash`, timing-safe compare) and RFC 6238 TOTP (HMAC-SHA1, 30 s step, 6 digits, ± 1 step window) with no external crypto dependency.

8. Zapier app

`zapier-app/` — Zapier Platform (core ^19), CommonJS, mocha tests.

File	Purpose
<code>index.js</code>	App definition; wires auth, trigger, action
<code>authentication.js</code>	API-key auth; test call against <code>/v1/status</code>
<code>triggers/new_item.js</code>	Polling trigger: GET <code>/v1/storage</code> , newest first, dedupe by id
<code>creates/store_data.js</code>	Action: multipart POST <code>/v1/storage</code> (form-data), fields: metadata JSON + optional files
<code>test/</code>	mocha suite (4 tests): auth, trigger, action

Publish flow: `zapier login` → `zapier push` → invite users / submit for review. The app’s base URL must point at a publicly reachable deployment of the API server.

9. Testing

```
npm test          # jest, repo root - 165 tests / 22 suites
npx tsc --noEmit  # type gate
cd zapier-app && npm test  # mocha - 4 tests
```

Conventions:

- **TDD** throughout; every module has in-memory adapters so tests never touch disk or network.
- HTTP tests use **supertest** against `buildApp()` with in-memory repos.
- SQLite tests use `:memory:` databases.
- Stripe is faked by implementing `MeterEventClient`; the idempotency suite (`tests/report-usage-idempotency.test.`) simulates growth, re-runs, month rollover, Stripe throws, lock contention, and partial failure.
- The job is tested via the injectable `reportMonthlyUsage(deps)` — never by executing `main()`.

10. Configuration

Env var	Default	Used by
PORT	3000	<code>src/index.ts</code>
ZAPPIER_DB	<code><root>/zappier.db</code>	<code>src/index.ts</code> , billing job
ADMIN_KEY	<code>admin-dev-key</code>	<code>adminAuth()</code>
ADMIN_USER	<code>admin</code>	Admin login

Env var	Default	Used by
DEMO_ADMIN_USER / DEMO_ADMIN_PASSWORD	demo / \$\$\$\$Admin###	Demo admin login
STRIPE_SECRET_KEY	—	Billing job; portal reloads when it starts with <code>sk_</code> (otherwise a dev payment client credits instantly)

Both `src/index.ts` and the billing job load `.env` from the installation root (`src/paths.ts PROJECT_ROOT`) — never from the process `cwd` — so the compiled server and the job run from any working directory. `.env` is git-ignored (`chmod 600`); `.env.example` documents the shape. Git identity is configured `repo-local`; `.gitignore` covers `node_modules/`, `dist/`, `.env`, `zapper.db*`.

11. Extending the system

Add an API endpoint: 1. Add the path + `operationId` to `openapi.yaml` (validation & docs follow automatically). 2. Add the route in `src/app.ts`, wrapping the handler with `meter('<operationId>', usage, pricing)`. 3. Add a rate-card rule (admin UI or PUT `/admin/api/endpoints/<operationId>`) — otherwise tiers without a `defaultRule` get 403. 4. Write the failing test first; keep `npm test + tsc` green.

Add a customer type: admin UI or PUT `/admin/api/tiers/:id({name, multiplier, monthlyCreditCents, defaultRule?})`.

Swap the storage backend: implement `UsageRepo` / `CustomerRepo` / `PricingStore` against your database and pass them to `buildApp({...})` — no other code changes. Same for `BillingReportRepo`/`JobLockRepo` in the job.

Change the billing cadence: the job is safe at any frequency (`delta + ledger + lock`). The Kimi cron job controls scheduling; update its cron expression to change cadence.

Known intentional limitations: stored items are in-memory (restart clears them; usage ledger is unaffected); `reportUsage` is retained but superseded by the delta path; Stripe identifier dedup covers a rolling 24 h window; `releaseLock` is not owner-scoped (harmless at this job's runtime); admin session tokens are in-memory (portal sessions are persisted); Stripe reloads credit the balance only after payment confirmation (no webhook endpoint yet — dev client credits instantly); email invoicing stores the preference but sending requires SMTP wiring (deferred); PO invoices with partial prepaid coverage are not partially paid by design.