Payment Processing Service

Independent project · 2026 · Solo · Public repository

Asynchronous payment processing that accepts a request in one call, charges the provider in the background, and never charges the same order twice.

Links: Repository · System design doc

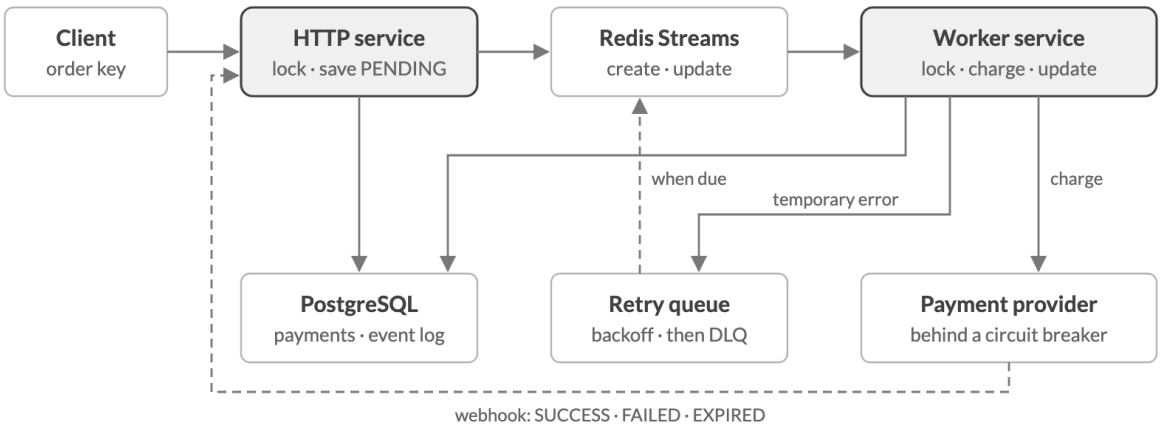
Why

When an API calls the payment provider directly, the client waits on the provider's 1–3 second response, and a slow or failing provider takes the API down with it. I wanted to build the reliable version: the API stays fast, provider failures are retried without the client, and a double-clicked "Pay" button cannot charge twice.

What it does

- Takes a payment request, saves it as PENDING and answers right away; a worker calls the payment provider afterwards.
- Supports virtual accounts, e-wallets and cards behind a provider interface; the first adapter is written for Xendit's API.
- Receives provider webhooks and moves the payment to SUCCESS, FAILED or EXPIRED.
- Keeps an append-only log of every state change and provider call, with the raw request and response.

Key flow



1. The client sends a payment with an order key. The HTTP service locks that key, checks for an existing payment, saves a new one as PENDING and publishes it to a Redis stream.
2. The worker picks it up, locks the payment, and charges the provider.
3. If the provider fails with a temporary error, the job is scheduled for a retry with exponential backoff. After 5 tries the payment is marked FAILED and the message goes to a dead-letter queue.
4. The provider later calls the webhook. The HTTP service checks the callback token, publishes the event, and the worker updates the status and writes the event log.

Key decisions

Decision	Why / trade-off
Separate HTTP and worker services	Each scales on its own load (traffic vs. queue depth), and a worker crash does not take the API down. The cost is two services to deploy.
Redis Streams instead of Kafka	Redis was already needed for locks, so no extra infrastructure. Consumer groups give at-least-once delivery; the cost is a lower throughput ceiling.
Retry queue in a Redis sorted set	The library's built-in retry blocks the consumer for the whole backoff. A sorted set lets it move on; a scheduler re-publishes jobs when due.
Idempotency at two levels	A Redis lock on the order key stops concurrent duplicates; a unique constraint in PostgreSQL is the final safety net. Same key with a different amount is rejected as a conflict.
Circuit breaker around the provider	When the failure rate passes 50%, calls fail fast without touching the network, and the retry queue absorbs the load until the provider recovers.
Only temporary errors are retried	5xx and timeouts are retried; 4xx and decode errors are marked permanent, so a bad request does not burn retries.
Provider behind an interface	The domain never sees provider-specific types; a new provider is one new implementation. Raw requests and responses are still kept for the audit log.

Non-goals

- Automatic reprocessing from the dead-letter queue: it is logged for a person to review.
- Refunds: a payment only moves from PENDING to SUCCESS, FAILED or EXPIRED.

Outcome

- A system design doc covering data flows, database design, locking, retries and graceful shutdown.
- Lesson: retry only what can succeed, or a bad request still makes five provider calls.

Code pointers

- internal/usecase/payment.go: idempotent create, charging and webhook handling.
- pkg/retryqueue/scheduler.go: the scheduler that re-publishes due retries under a lock.
- pkg/psp/xendit/executor.go: circuit breaker and error classification around the provider call.

Stack

Go · PostgreSQL · Redis Streams · Watermill · redsync · gobreaker · Docker