

Javohir (Ned)

Backend Developer & PostgreSQL DBA | Career Details

+82 10-8864-3848 | nediro883@gmail.com | github.com/kicheol83 | KakaoTalk open.kakao.com/o/sa2bEaRi | javohir.dev | Seodaemun-gu, Seoul

Experience

Backend Developer (remote, Tashkent)

2024.09 - 2025.08

Sigma Software

A global IT services company founded in Ukraine (2,000+ specialists in 12 countries). On a team of six (3 backend, 2 frontend, 1 QA) I owned the backend APIs and the database schema. Java, Kotlin, Spring Boot, PostgreSQL, Redis, Docker, GitHub Actions, AWS, Kubernetes.

- Order list API p95 1.4 s → 210 ms: added a composite index and merged N+1 queries into a single JOIN (Grafana p95 before/after)
- Redis cache cut repeated read queries to the primary database by about 45% (pg_stat_statements calls before/after)
- Took part in a Primary-Standby HA setup on streaming replication; failover to the standby completes in 5-10 s
- Designed order and invoice APIs for a European client in Java/Kotlin and Spring Boot, and introduced versioned Flyway migrations

Backend Developer & PostgreSQL DBA (remote, Tashkent)

2022.10 - 2024.05

LeverX Group

An international IT company building SAP-based software and web, mobile and cloud solutions (1,500+ employees). On a team of eight I owned the integration services, reporting APIs and the database. Node.js, NestJS, PostgreSQL, REST, Docker, Redis, AWS.

- Completed a PostgreSQL 13 → 15 major upgrade with logical replication and zero downtime
- Built an integration service for data exchange between internal systems, removing manual integration work
- Added pagination and batch export to large reports so they are processed in chunks without overloading the server
- Wrote unit and integration tests wired into CI, and optimized slow database queries

Backend & Database Developer (remote)

2021.11 - 2025.09

Freelance

Project-based work solving database performance, security and scaling problems for overseas clients (16 months in total).

UAE, Manufacturing · 2021.11 - 2022.04 (6 months)

Problem: As data grew, end-of-day production reports opened slowly and managers waited a long time.

Key work: Analyzed slow queries with EXPLAIN ANALYZE and rewrote them, added indexes, and introduced materialized views for the heaviest frequent reports.

Result: Report load time 12 s → 0.8 s (~15x); overall database query load dropped

Hong Kong, Restaurant / F&B · 2023.01 - 2023.04 (4 months)

Problem: Some database users in the ordering system had far more privileges than their jobs needed, a real security risk.

Key work: Redesigned permissions with RBAC, restricted table access with row-level security, and set up daily backups with a recovery procedure.

Result: Unneeded superuser connections 7 → 0; restore from backup tested at under 15 minutes

Bangladesh, AI services · WellDev Bangladesh Ltd. · 2025.03 - 2025.09 (6 months)

Problem: The table storing AI model results grew fast and INSERTs slowed down.

Key work: Split the table with monthly range partitioning and introduced an archiving strategy for old data, so live traffic touches only the partitions it needs.

Result: INSERT p95 320 ms → 45 ms (~7x); active data 180 GB → 22 GB

PG Insight

An open multi-tenant monitoring service: anyone signs up, adds their own PostgreSQL and sees only the databases they added

2026.08 - 2026.10

Role: Solo: design, backend, frontend, security, tests, deployment and operations (81 commits)

NestJS, TypeScript, Prisma, PostgreSQL, TimescaleDB, Socket.IO, React, Vite, Tailwind CSS, Recharts, Jest, Vitest, Docker, Caddy

What it proves

- Tenant isolation: a global guard checks the owner on every :targetId route and returns 404, not 403, for someone else's target; adding an unguarded mutating route fails a test
- SSRF protection: every IP in the DNS answer is checked and the connection is pinned to the verified IP (keeping TLS SNI); production rejected platform-db, 169.254.169.254 and 127.0.0.1
- Limits on arbitrary SQL: EXPLAIN goes through six layers of checks and runs READ ONLY with a 5 s statement_timeout; pg_sleep was cut off with 400 in 0.05 s and generate_series(1, 10000000000) in 5.4 s
- I measured the load on monitored databases: 0.06% of one core over 10 minutes, +0.3% container CPU
- Operations: SHA-tagged deploys, recorded only after a health check, and a two-way rollback drill

Problem

Slow queries, lock waits, bloat and replication lag in a production PostgreSQL have to be checked separately across tools and hand-written SQL. I brought them into one service, and because it is a public service that connects to other people's databases, tenant isolation, SSRF protection and limits on arbitrary SQL had to be there from the start.

Architecture

Collectors gather metrics from each database every 3 seconds to 5 minutes and write them to TimescaleDB (10 hypertables), while live pages query the target directly. Users, targets, alerts and audit logs live in a platform DB (PostgreSQL 16) managed with Prisma 7. Global NestJS 11 guards (Throttler → JWT → TargetAccess) check ownership on every `:targetId` route, and target queries run as raw SQL on a per-target pool (max 5). There are two roles, admin (the first account) and user (up to 3 own targets, public addresses only), and the UI is in Korean, English and Uzbek.

Key decisions

- An open multi-tenant service, not a demo: Open sign-up, with each user seeing only the targets they registered. / Trade-off: Tenant isolation, SSRF protection, quotas and rate limits became mandatory.
- Ownership checks in a global guard, not in controllers: Global guards in the order Throttler → JWT → TargetAccess check every `:targetId` route. / Trade-off: Routes keyed by other IDs need the `@TargetAccess` decorator.
- 404, not 403, for someone else's target: Without permission the response is the same 404 as for a target that does not exist. / Trade-off: Existence has to be checked separately for admins too.
- Resolve the host and pin to the IP: Every IP in the DNS answer is checked with `net.BlockList`, and the connection goes straight to the verified IP. / Trade-off: TLS SNI has to be passed by hand, and self-hosting needs the `TARGET_ALLOW_PRIVATE_HOSTS` option.
- Keep EXPLAIN, but READ ONLY + 5 s + a function denylist: After six layers of checks it runs in a READ ONLY transaction with `statement_timeout` and `lock_timeout`. / Trade-off: Time and function limits can reject some legitimate queries.
- New targets default to `sslmode=require`: Switched new targets and the database default to require (47287b4). / Trade-off: For a database without TLS the user has to choose disable explicitly.
- Migrations are never rolled back: Every migration is additive (add first, clean up later). / Trade-off: Every migration has to stay compatible with the previous code.

Key numbers

- Queries sent to the target in 10 minutes: 2,430
- Load on the target (share of one CPU core): 0.06 %
- Average query execution time: 0.14 ms
- Collected data lag: 1.4 s
- Time to block `pg_sleep` inside EXPLAIN: 0.05 s
- Automated tests (188 backend + 49 frontend): 237

Incident

After an API restart no target was monitored; the overhead measurement showed 0 queries from PG Insight.

Cause: The log showed Starting collection for 0 active target(s): the orchestrator started in `onModuleInit` before the pools had connected, and data from 14:23 to 17:25 UTC was lost.

Fix: Moved startup to `onApplicationBootstrap` (6246ce2); both targets are collected right after a restart.

Prevention: Added a lifecycle test that reproduces the bug and also checks that paused targets are skipped; the new tests were confirmed to fail on the old code.

Tests and quality

- Tests: 237
- CI: GitHub Actions (워크플로 준비, billing 문제로 현재 중단)
- Jest, Supertest, Vitest, 실제 PostgreSQL 16

Outcome

Collection load on a monitored database was 2,430 queries and 353.4 ms of total execution in 10 minutes, about 0.06% of one CPU core. I found 5 bugs on real production databases (LedgerCore, FlashDrop) and 3 more in code review and tests, and reproduced each one with a test before fixing it. Other cases: a real-time broadcast error that stopped the collectors (b9825e4), a RENAME syntax error in the lock query (b9825e4), connection metrics stamped with the session start time (d587ce6, 891 wrong rows deleted), paused targets turning themselves back on after a restart (41569c7), SSL settings lost on a password change and prefer not actually encrypting (41569c7, 47287b4), and a first rollback drill where the script stopped safely because the old image was missing.

pginsight.javohir.dev | github.com/kicheol83/pg-insight

LedgerCore

A PostgreSQL double-entry ledger where money is never created or lost and no transfer is ever applied twice

2026.08 - 2026.10

Role: Solo: design, backend, database, tests, load testing, deployment (53 commits)

NestJS, TypeScript, PostgreSQL, node-pg-migrate, zod, React, Vite, Vitest, k6, Docker, Caddy, GitHub Actions

What it proves

- Correctness under concurrency: ordered row locks gave 0 deadlocks and 0 faults across 12/12 load runs, and a reversal race was found and fixed with a test
- Invariants in the database: debits = credits is checked by a trigger at COMMIT, with imbalance 0 both under load and in a database restored from backup
- Measurement-driven optimization: a single-statement balance read cut p95 from 24.9 to 13.4 ms in a fixed-rate A/B
- Finding causes with evidence: a 10-second stall under load was traced by lining up Postgres and API log timestamps, and the cause turned out to be synchronous logging
- Operations: SHA-tagged deploys, rollback in about 1 s, daily backups with verified restores

Problem

A system that records money moving between accounts has one requirement: never be wrong. Whether requests arrive at the same instant, the client retries, or the process dies halfway, money must not appear or vanish and no transfer may be applied twice.

Architecture

An API (a NestJS 10 modular monolith) handles users and accounts, deposits and withdrawals, transfers, reversals, balances, history and reconciliation. A separate worker process from the same codebase leases outbox events with FOR UPDATE SKIP LOCKED and sends them as HMAC-signed webhooks. Invariants are checked by deferred constraint triggers in PostgreSQL 16, so a bug in application code cannot bypass them. An operations console (React + Vite) shows accounts, transfers and outbox delivery.

Key decisions

- ADR-0001 Money as BIGINT minor units, strings in the API: Store BIGINT minor units, wrap them in a bigint-based Money type and serialize as strings at every boundary. / Trade-off: Clients must read amounts as strings, in exchange for no floating-point errors.
- ADR-0002 No balance column; balances derived from entries: Balances are always the sum of ledger_entries, with a covering index to bound the cost. / Trade-off: Read cost grows with history; snapshots get added once a measurement calls for them.
- ADR-0003 Raw SQL instead of an ORM: The pg driver with raw SQL, and node-pg-migrate for migrations. / Trade-off: Type safety is guaranteed by integration tests rather than the compiler.

- ADR-0004 Modular monolith: accounts, ledger, idempotency and outbox modules in one process and one database. / Trade-off: A module never touches another module's tables directly.
- ADR-0005 Invariants in deferred constraint triggers: Triggers at COMMIT check at least two entries, debits = credits, total = transaction amount and matching currencies. / Trade-off: Errors surface at COMMIT rather than at the statement.
- ADR-0006 READ COMMITTED + FOR UPDATE in id order: Account row locks are always taken in ascending id order. / Trade-off: Every new write path must follow the same order.
- ADR-0007 Monthly partitions for the outbox only, none for the ledger: Monthly partitions by created_at plus a DEFAULT partition, applied to outbox_events only. / Trade-off: Creating partitions ahead of time becomes an operational task.

Key numbers

- Valid load-test runs (0 deadlocks, 0 faults): 12 / 12
- Plain transfer throughput (50 VUs): 249 req/s
- Plain transfer p95: 228 ms
- Hot-account transfer p95 (40 VUs → 1 account, 0 409s, 0 deadlocks): 896 ms
- Mixed workload throughput (30 readers + 10 writers): 884 req/s
- Balance read p95 (fixed-rate A/B, after): 13.4 ms
- Restore from backup (14/14 tables, imbalance 0): 0.68 s
- Automated tests: 392

Incident

A full benchmark produced three 500s and three 409s; the ledger stayed balanced.

Cause: Postgres raised lock_timeout at 04:00:52.7, but the API logged its response 10 seconds later, with a burst of 500s within 40 ms: the Node process itself had stalled. The cause was logging every request at info level; on Windows, writes to a stdout pipe are synchronous and the PowerShell redirection could not keep up.

Fix: With LOG_LEVEL=warn the same scenario gave 0 faults, 0 409s and 37% more throughput. lock_timeout 3 s, statement_timeout 5 s and idle_in_transaction_session_timeout 10 s released the locks, and timeouts that had returned 500 INTERNAL_ERROR now return 503 DATABASE_TIMEOUT (retryable: true) (8c30dba).

Prevention: Wrote three timeout → 503 tests against a real Postgres and pg-pool, and recorded in the limitations that production needs an async logger or sampling.

Tests and quality

- Tests: 392
- CI: GitHub Actions (커밋 메시지, 포맷, lint, typecheck, Postgres 서비스로 테스트, 빌드; billing 문제로 현재 중단)
- Vitest, supertest, 실제 PostgreSQL 16 (mock 없음), k6 + bench.ps1

Outcome

All 12 load-test runs on the final code were valid, with 0 faults, 0 deadlocks, 0 contention and a balanced ledger after every run. Cutting the balance read to a single statement lowered p95 from 24.9 ms to 13.4 ms in a fixed-rate A/B. Before trusting the numbers I also found and fixed a race where the second of two concurrent reversals returned 422 instead of 409 (ccee39b), tests that never ran on a clean clone because .env.test was gitignored (e0e6884), and __ITER and teardown bugs in the k6 scripts (0f76291).

ledger.javohir.dev | github.com/kicheol83/ledger_core

Smart Farm

Real-time greenhouse IoT monitoring and automatic irrigation that closes the loop from ESP32 sensors through MQTT, a pipeline and rules to the pump and

2026.01 - 2026.10

back to the device

Role: Solo: backend, frontend, ESP32 firmware, deployment and operations (171 commits, in production since 2026-10-04)

NestJS, TypeScript, GraphQL, MongoDB, Redis, BullMQ, MQTT, Socket.IO, React, Vite, Leaflet, ESP32, MicroPython, Jest, Docker, Caddy

What it proves

- Proving with measurements that IoT data is not lost: 24/24 through a 132 s MongoDB outage and 7/7 through a 20 s backend stop; the losses on the old code (1 of 12, 1 per deploy) were measured first
- A security audit and fixes: more than 12 issues, including ADMIN escalation at sign-up, arbitrary field writes, cross-member access, unlimited OTP guessing and a hard-coded JWT key, closed after first checking production for abuse
- Closing the loop between device and backend: RELAY_ON/OFF goes out over MQTT, pump run time is capped on both the server (30 min) and the firmware (1800 s), and an automation rule never fires twice at once
- Engineering discipline: all 89 new tests were confirmed to fail on the old code, and wrong hypotheses and insufficient tests are recorded too

Problem

Greenhouse sensor readings drop out or spike, device clocks can be wrong, and no reading may be lost while the database or backend is briefly down. When the soil dries out the pump should start before anyone looks at a screen, and it must never keep running and flood the greenhouse because the network dropped. Nobody should ever reach someone else's farm data.

Architecture

A real ESP32 (DHT22, capacitive soil moisture, BH1750 light, water level, relay + 12 V pump) and a demo simulator in production publish over the same path to Mosquitto (TLS 8883, QoS 1). A NestJS 11 IoT pipeline checks the device API key, payload shape, topic-to-deviceId match, sensor ownership and calibration, passes through a Redis buffer (retries, DLQ), stores raw readings plus a 90-day-TTL MongoDB time series, and builds hourly, daily and monthly aggregates. It computes per-sensor Z-score anomalies (≥ 2 warning, ≥ 3 critical), a plant health index and Open-Meteo weather (10-minute cache); when an automation rule changes an actuator, RELAY_ON/OFF goes to the device over MQTT. The dashboard updates live over GraphQL (51 resolvers) and Socket.IO, in Korean, English and Uzbek.

Key decisions

- Message buffer: a hash plus two zsets instead of a Redis list: `iot:buffer:messages` (hash), `iot:buffer:inflight` (zset, processing start), `iot:buffer:retry` (zset, next attempt), `iot:buffer:dead` (DLQ list); ack is an `O(1)` `hdel + zrem` by id. / Trade-off: The retry logic is our own, hence 10 tests and a check against a real redis-server.
- A `done[]` list per reading: Successful sensors from `Promise.allSettled` go into `done[]` and are skipped on retry; retries keep the original `receivedAt`.
- Telling authorization errors from database errors: Only `UnauthorizedException` drops the message; every other error goes to retry.
- Persistent MQTT session + connecting in `onApplicationBootstrap`: `clean:false` with a stable `clientId` (`MQTT_CLIENT_ID` distinguishes a second instance), connecting in `onApplicationBootstrap` after every `onModuleInit`.
- Error logging is best-effort: Logging is wrapped in `try/catch` and only warns on failure, without touching the retry count.

- Ownership checks: a global interceptor + @OwnedArgs: A global OwnershipInterceptor finds the resource from the argument name and walks the chain up to farm.memberId; ambiguous arguments are mapped with @OwnedArgs({ 'input.fieldId': 'fieldMap' }). / Trade-off: Every request performs an extra ownership lookup.
- Pump limit on both sides: The server sets autoOffAt to at most 30 minutes, and the firmware switches itself off after 1800 s + 60 s grace based on ticks_ms.
- Automation rules: claiming the cooldown atomically: The cooldown condition sits in the findOneAndUpdate filter, so only one claims the rule.
- A fallback for recordedAt: If the device time is more than 24 hours old or more than 5 minutes ahead, the server uses its receive time, and second-resolution timestamps are detected automatically; the firmware retries NTP every hour.

Key numbers

- Messages saved through a 132 s MongoDB outage (DLQ 0): 24 / 24
- Messages delivered through a 20 s backend stop: 7 / 7
- Rows saved in the 10 minutes after the IoT deploy (0 rejected): 320
- Security probes rejected: 5 / 5
- New tests (each confirmed to fail on the old code): 89
- Maximum pump run time (server): 30 min
- Longest retry interval (30 / 60 / 120 s, DLQ after 3): 120 s
- MongoDB collections: 40

Incident

Sending memberRole: "ADMIN" in the sign-up mutation created an ADMIN member (probe before).

Cause: CreateMemberInput had a memberRole field and memberModel.create(input) stored the whole input; MemberUpdateInput for profile updates also accepted role, status, email and password. The same audit found the member list open without a token ({ "total": 7 }) because @Roles(MANAGER) had no @UseGuards (RolesGuard), and the buffer admin resolvers open because the guard ignored class-level @Roles.

Fix: Removed the field from the input and pass only allowed fields to create and \$set (22a760bd); added the guard and read handler and class together with getAllAndOverride, so an invalid token gets 401.

Prevention: Before fixing I checked production for abuse (all 7 members WORKER/ACTIVE, 0 privileged, 0 suspicious calls). member-security.spec.ts has 13 tests, 12 of which fail on the old code, and after the deploy all 5 external probes are rejected.

Tests and quality

- Tests: 89
- Jest 30, ts-jest, 실제 redis-server, Mosquitto 2.0.18, MicroPython stub harness

Outcome

An audit on 2026-10-10-11 produced nine fix commits, each with a before/after measurement. With MongoDB stopped for 132 s, 24/24 messages were saved (6 retries, DLQ 0); with the backend stopped for 20 s, 7/7 arrived, including the one sent while it was down, which came from the broker queue. On the old code a 58 s outage lost 1 of 12 messages and a deploy restart lost 1. The security audit found and closed more than 12 vulnerabilities: becoming ADMIN by sending memberRole at sign-up (22a760bd), writing arbitrary fields via profile update, reaching other members' farms, sectors and alerts (c2c5ca26), unlimited OTP guessing and Math.random OTPs, a JWT key hard-coded in the code (a5a1b62e), and an open GraphQL playground in production. Before fixing, I checked production for signs of abuse (7 members, 0 privileged accounts, 0 suspicious calls). I also fixed the backend turning the pump on only in the database without sending the device any command (3f3eacb2) and ESP32s sending 1970 timestamps before NTP sync. When my first hypothesis

(errors from heartbeat) was wrong, the logs pointed to the real source (/sensors), and I recorded that a 45 s outage never exercised the error path because of the driver's 30 s wait.

farm.javohir.dev | github.com/kicheol83/smart-farm

FlashDrop

First-come coupons with zero over-issue: four concurrency strategies implemented and measured with k6

2026.09 - 2026.10

Role: Solo: design, backend, tests, load testing, frontend, deployment and operations (21 commits)

Java, Spring Boot, Spring Data JPA, PostgreSQL, Redis, Lua, Flyway, Bucket4j, JUnit 5, Testcontainers, k6, Docker, Caddy, Cloudflare

What it proves

- Proving the problem with numbers: the no-lock version issued 300 coupons instead of 50 and remaining_quantity stopped at 46, not 0
- Three correct solutions: the other strategies let exactly 50 of 300 requests through, with exactly 50 rows in the database
- Explaining the result: I found why the optimistic lock beat FOR UPDATE under this load; FOR UPDATE queues even the 250 requests that will be rejected
- Measurement discipline: warm-up, median of three, runs partly refused by the TCP backlog marked invalid automatically; when re-measuring overturned the conclusions I fixed the README
- Operations: SHA-tagged deploys and rollbacks, daily backups with restore tests, and a deploy guard added after conflict markers reached production

Problem

Three hundred concurrent requests hit 50 coupons, and not a single extra coupon may be issued. A plain read, check, write flow issues 300 coupons instead of 50 because of lost updates. I implemented the same business rule four ways to measure the balance between correctness, throughput and latency under real load.

Architecture

Caddy serves the static frontend and forwards /api/* to Spring Boot. Requests first pass a Bucket4j rate limiter whose state lives in Redis. Each strategy has its own endpoint, entity and table. The Redis strategy checks the remaining count and the set of users who already claimed inside one Lua script, and writes to PostgreSQL in the background with @Async.

Key decisions

- Redis: DECR + SADD in Lua, database write via @Async: SISMEMBER → GET → DECR + SADD run in one claim.lua, so the hot path has neither Postgres nor a check-then-act gap. / Trade-off: If the process dies between Redis and the database, the coupon can be missing from the database.
- Optimistic retry split into two beans: OptimisticClaimService (retries, max 20) → OptimisticClaimAttempt (@Transactional, @Version) / Trade-off: One piece of logic is split across two classes.
- Separate tables and endpoints per strategy: Each of the four strategies has its own campaign / claim tables, entities and endpoint. / Trade-off: Code and migrations are duplicated (8 tables).
- The no-lock version kept on purpose: Kept an endpoint where the lost update happens as is, without a CHECK constraint on its table. / Trade-off: A real product would never ship such an endpoint.
- Idempotency and the stock floor in the database: UNIQUE(campaign_id, user_id), CHECK (remaining_quantity >= 0): UNIQUE on every claim table and the V9 CHECK on the three correct strategies; production rejected an update setting remaining_quantity = -1. / Trade-off: The Redis strategy checks in two places, Redis and the database.

- Rate-limit key is the userId: A Bucket4j + Redis (Lettuce) token bucket keyed by userId, so the limit is shared across instances. / Trade-off: The client can change it, so production needs it combined with the IP.

Key numbers

- Redis Lua throughput: 873 req/s
- Redis Lua p95: 178 ms
- SELECT FOR UPDATE throughput: 188 req/s
- SELECT FOR UPDATE p95: 1,465 ms
- Optimistic lock throughput: 331 req/s
- Optimistic lock p95: 831 ms
- No lock: coupons issued out of 50: 300
- Rollback command time: 1.2 s
- Database restore from backup (11/11 tables): 0.22 s

Incident

After a deploy on 2026-10-09 the flashdrop.javohir.dev UI broke and the page showed a raw <<<<<<< HEAD marker.

Cause: The local repository was behind GitHub, so the push was rejected; the git pull merge conflicted in four files and the markers were committed unresolved. Caddy served the frontend straight from the git working directory, so the pull went live immediately, and a backend rollback could not undo it.

Fix: Restored frontend/index.html from the last working commit, rewrote the READMEs, removed accidentally committed IDE build output (bin/), added it to .gitignore and redeployed (7ac0754).

Prevention: deploy.sh now uses git grep to scan upstream for conflict markers and stops the deploy if it finds any, and merges only with --ff-only (a2dd204). A hidden side effect turned up too: while fixing the conflict, the READMEs (ko/en/uz) had reverted to the old cold-run table; I caught it when comparing the portfolio text with the repo and restored the re-measured results (a29c567).

Tests and quality

- Tests: 4
- JUnit 5, Testcontainers (PostgreSQL 17, Redis 7), virtual threads, k6, bench.ps1

Outcome

The no-lock version issued 300 coupons instead of 50 (remaining_quantity stayed at 46, not 0), while the other three let exactly 50 of 300 requests through, leaving exactly 50 rows in the database. Redis beat FOR UPDATE by 4.6x on throughput and 8.2x on p95. Figures are the median of three valid runs after one warm-up; runs where success + sold-out added up to fewer than 300 (some requests refused by the TCP backlog) were marked invalid automatically and rerun. The rate limit let 6 of 12 requests from one userId through and answered 6 with 429 (5 tokens + greedy refill).

flashdrop.javohir.dev | github.com/kicheol83/flashdrop

LiveCommerce

A flash-sale platform for live broadcasts: 1,000 purchase attempts for 100 items, zero oversells 2026.09 - 2026.10

Role: Solo: architecture, seven services, frontend, tests, load testing, deployment and operations (127 commits)

Kotlin, Spring Boot, Spring Cloud Gateway, Spring Data JPA, PostgreSQL, Apache Kafka, WebSocket (STOMP), Flyway, Toss Payments, OpenTelemetry, Jaeger, Next.js, TypeScript, Tailwind CSS, JUnit 5, k6, Docker, Caddy

What it proves

- Correctness: 1,000 attempts for 100 items, repeated three times, with 0 oversells, 0 payment/sales mismatches and 0 HTTP errors; five invariants checked after every run
- Fault tolerance: stopping Kafka or the payment database for real lost 0 events and charged nobody twice (chaos tests)
- Measurement-driven speed-ups: sell-out 1 min 27 s → about 23 s, order creation p95 5.72 s → 2.68 s; the connection-pool hypothesis was rejected by measurement
- Closing a production business risk with evidence: one account could lock up all stock with unpaid orders; a partial unique index fixed it (from 15 → 0 to 15 → 5)
- Edge protection: rate limits on login, sign-up and orders, with the client IP taken from the trusted proxy chain so a forged X-Forwarded-For cannot dodge the 429 (verified in production)
- Operations: SHA-tagged deploys, health-gated rollback, a two-way drill in production, and a JWT key that keeps tokens valid across restarts

Problem

A limited sale during a live broadcast packs orders into a few seconds, and stock, payment and delivery live in different services. In a flash sale, selling out is normal, so charging first turns every sell-out into a refund and a PG fee, and one stopped service can lose events or double-charge.

Architecture

api-gateway (Spring Cloud Gateway) verifies JWTs, rewrites identity headers, and handles rate limiting and the WebSocket proxy. The Saga orchestrator in order-service holds stock for ten minutes before taking payment (AWAITING_STOCK → AWAITING_PAYMENT → PAYMENT_CONFIRMING → CONFIRMING_STOCK → COMPLETED, with a five-minute payment window). inventory (hold, confirm, release, expiry sweep), payment (Toss confirm and cancel with an Idempotency-Key), live (STOMP chat, live stock, viewer count), auth (RS256 JWT, JWKS, refresh token rotation) and delivery (shipping, HMAC courier webhooks) each own a PostgreSQL database and read one another's data only through APIs or Kafka events. Events leave through each service's transactional outbox together with a traceparent, so the whole Saga stays one Jaeger trace across Kafka.

Key decisions

- Hold stock first, then charge (hold-then-pay): Hold stock for ten minutes, open a five-minute payment window, and confirm the hold once payment is confirmed. / Trade-off: Stock stays locked for ten minutes, which made a one-open-order-per-buyer rule necessary.
- Orchestration, not choreography: order-service coordinates the Saga, and a recovery scheduler resumes stuck steps every five seconds. / Trade-off: order-service becomes the central point.
- Transactional outbox, PUBLISHED only after the broker acks: State changes and events are written in one transaction; the relay sends in parallel and marks only the leading run of acknowledged events as PUBLISHED. / Trade-off: Duplicates can happen, so every consumer is idempotent.
- Toss called outside the transaction, result written with the outbox: With TransactionTemplate, Toss confirmation is called outside, and the result plus outbox event are written in one transaction. / Trade-off: If the result is not recorded, the recovery scheduler retries with the same Idempotency-Key.
- Reject sold-out early with a lock-free scalar read: Read one number before locking and reject at zero. The entity is not read first, because Hibernate's first-level cache would return a stale copy at the next FOR UPDATE and open the door to oversells. / Trade-off: A buyer may see sold-out while stock is being returned, but oversells remain impossible.
- One open order: a partial unique index: A partial unique index on (member_id, product_id) for open statuses (V5); a violation returns 409 OPEN_ORDER_EXISTS with the existing orderId. / Trade-off: Buyers must pay or cancel the old order before a new one, and the UI takes them back to it.
- Rate limits in gateway memory, orders keyed by user: Login 10/min and sign-up 5/hour by IP, orders 20/min by user; the IP is the right-most public address in the XFF chain. / Trade-off: It is only accurate with one gateway, and per-IP login limits do not stop attacks on one account from many IPs.

- JWT key in .env, kid as an RFC 7638 thumbprint: JWT_SIGNING_KEY is loaded from the server's .env (chmod 600) and the same key always yields the same kid; without the key, compose refuses to start. / Trade-off: Rotating the key briefly rejects older tokens.

Key numbers

- Oversells (100 items, 1,000 attempts, all 3 runs): 0
- Sold = charged: 100 / 100
- Order creation p95: 2.68 s
- Payment submission p95: 0.63 s
- Sell-out time: 23 s
- Stock update delivery (500 viewers): 99.2 %
- Viewer-count messages reduced: 98 %
- Chaos tests: lost events / double charges: 0

Incident

In production on 2026-10-09, lc-choard-check.sh created unpaid orders back to back from one test account: 2×201 and stock 15 → 0. A single user could shut everyone else out of the broadcast's whole stock for the ten-minute hold.

Cause: The Saga opened a separate hold for every POST /api/orders, and no rule anywhere said one buyer, one open order.

Fix: Migration V5: a partial unique index on (member_id, product_id) for the open statuses (AWAITING_STOCK, AWAITING_PAYMENT, PAYMENT_CONFIRMING, CONFIRMING_STOCK). The controller checks first: under a race, the DataIntegrityViolationException is identified by the index name and answered with 409 OPEN_ORDER_EXISTS plus the existing orderId, and the frontend resumes that order.

Prevention: On a local Postgres, the second of two concurrent transactions waited 1.5 s and was rejected; a production re-check gave 1×201 + 409 with stock stopping at 5. An integration test asserts exactly one 201 out of ten concurrent requests, and k6 flash-sale.js now gives each VU its own user (USERS ≥ VUS is checked).

Tests and quality

- Tests: 46
- JUnit 5 (통합 24, 카오스 3, gateway 단위 14, SigningKeys 5), 실행 중인 전체 스택에 실제 HTTP, k6 3종

Outcome

Three runs of 1,000 purchase attempts for 100 items (200 VUs) each ended with 0 oversells, 0 mismatches between payments and units sold, and 0 HTTP errors, with five invariants checked through the admin API after every run. Measuring the bottlenecks cut the sell-out time from 1 min 27 s to about 23 s, and a connection-pool hypothesis was rejected by measurement. In production I closed a hole where one account could hold the entire stock (15 → 0) with two unpaid orders, using a partial unique index (15 → 5); fixed a JWT key problem where, after an auth restart, the same token got different answers from different services (401/200) (0eeaf2c); and added limits to login and ordering, which had none (085ded0). I also tracked down an outbox transaction holding locks for minutes when the broker was unreachable (881afda), consumers missing new partitions after a partition increase, a k6 invariant that showed 101 sold because a hold from a previous run expired, and 14 failing integration tests caused by another project's container occupying port 8080.

live.javohir.dev | github.com/kicheol83/livecommerce-saga