drop-backend-migration-plan

Drop Backend Migration Plan

Hono/TypeScript ? Kotlin/Ktor

Authored by: Petter Graff (Software Architect Agent) **Date:** 2026-03-29 **MC Task:** #5124 **Status:** READY FOR REVIEW — Requires CEO approval before any build action (ZAKON #2)

Executive Summary

Five previous attempts failed because this migration was treated as a single monolithic task. It is not one task. It is eight tasks — each independently deliverable, each with its own acceptance criteria, each runnable on a separate agent thread.

The core architectural principle here is **strangler fig**: the new Ktor service grows alongside the existing Hono service. Traffic is routed per-endpoint via nginx (or AWS load balancer at the path level). At no point is the old service down while the new one is being built. Cutover is a DNS/nginx config change, not a big-bang deployment.

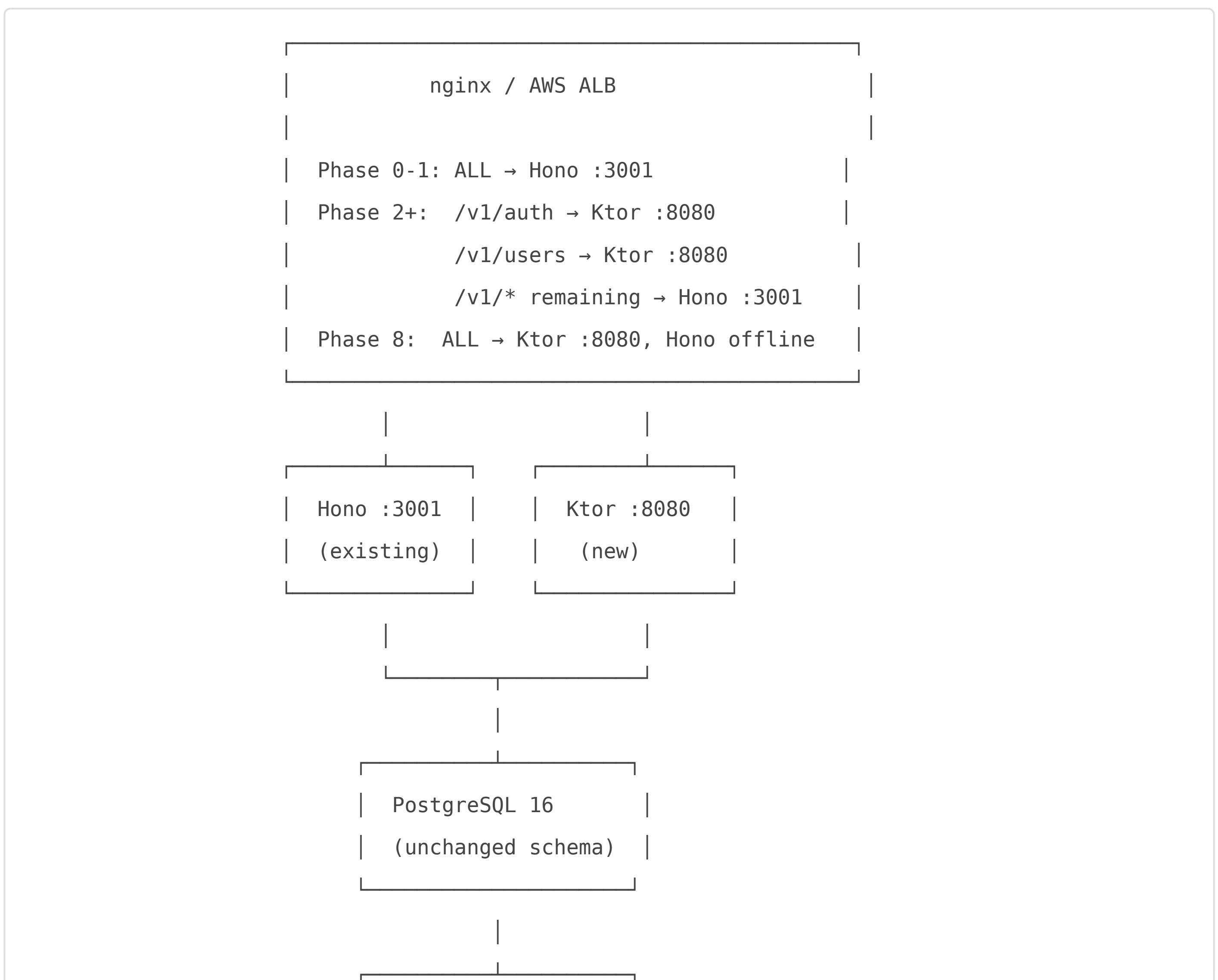
The database does not change. Same 24 PostgreSQL 16 tables, same schemas. Flyway replaces Drizzle for migrations going forward, but the existing tables are adopted as-is via a Flyway baseline migration. This is the key insight: decoupling the DB migration from the API migration means each phase can be independently verified against a live database.

Why It Failed Before (Root Cause Analysis)

Failure Pattern	Root Cause	Fix in This Plan
Agent ran out of context window	22 routes + 24 tables = too much in one pass	Each phase has ≤4 route files

Failure Pattern	Root Cause	Fix in This Plan
API contracts broken	No explicit contract test suite defined upfront	Phase 0 builds contract tests FIRST
PII encryption silently wrong	AES-256-GCM reimplemented from scratch each time	Phase 0 documents the exact algorithm; Phase 2 validates it against known vectors
BankID OIDC stub wrong	Complex OAuth2 PKCE flow with state validation	Phase 1 is ONLY auth — nothing else
Agent improvised table mappings	No Exposed ORM specification	Phase 0 produces Exposed entity specs for all 24 tables
No parallel running strategy	Hono taken down too early	Strangler fig: both services run until Phase 8

System Architecture During Migration



```
                          │  Redis 7             │
                          │  (sessions + limits) │
                          └──────────────────────┘
```

Key constraint: Both services share the same PostgreSQL instance and Redis instance. Session tokens issued by Hono must be validatable by Ktor (same JWT secret, same session table). This is addressed in Phase 1.

Database: 24 Tables Mapped to Phases

Table	Phase	Notes
`users`	2	Core entity
`sessions`	1	JWT validation — needed from day one
`refresh_tokens`	1	Token refresh
`otp_codes`	1	Phone OTP
`oidc_states`	1	BankID OIDC state/nonce
`settings`	2	User settings
`user_preferences`	2	User prefs
`recipients`	3	Money movement
`transactions`	3	Core money table
`exchange_rates`	3	Rates
`bank_accounts`	4	AISP
`ob_consents`	4	Open Banking consents (Berlin Group)
`cards`	7	Feature-flagged, low priority
`spending_limits`	7	Cards related
`merchants`	7	QR merchants
`notifications`	6	Push notifications
`push_tokens`	6	Expo push tokens
`audit_log`	5	Compliance
`aml_alerts`	5	AML
`str_reports`	5	Suspicious Transaction Reports

Table	Phase	Notes
`screening_results`	5	PEP/sanctions
`consents`	5	GDPR consents
`data_access_requests`	5	GDPR Art. 20
`complaints`	5	PSD2 complaints
`withdrawal_requests`	5	Angrerett
`webhook_events`	6	Webhook dedup
`webhook_dlq`	6	Dead letter queue
`settlement_batches`	6	Financial settlement
`settlement_items`	6	Settlement line items
`reconciliation_reports`	6	Reconciliation
`reconciliation_discrepancies`	6	Reconciliation details
`circuit_breaker_state`	3	PISP/AISP circuit breakers
`disputes`	5	Transaction disputes

Phases

Phase 0: Foundation + Contract Baseline

Goal: Ktor skeleton project. All 24 Exposed entities. All API contracts documented as tests. PII encryption verified.

Duration estimate: L (4-6 days)

Files to create:

```
drop-api-ktor/
├── build.gradle.kts
├── settings.gradle.kts
├── gradle.properties
├── Dockerfile
├── src/main/kotlin/no/getdrop/api/
│   ├── Application.kt          ← Ktor entry point
│   ├── Routing.kt              ← Route registration (stubs)
```

```
│   ├── Database.kt            ← HikariCP + Exposed setup
│   ├── plugins/
│   │   ├── Serialization.kt
│   │   ├── Security.kt        ← JWT config (same secret as Hono)
│   │   └── Monitoring.kt      ← Sentry + OTel
│   └── db/
│       └── Tables.kt          ← Exposed Table objects (all 24)
├── src/main/resources/
│   ├── application.conf
│   └── db/migration/
│       └── V1__baseline.sql   ← Flyway baseline (existing tables, no-op)
└── src/test/kotlin/no/getdrop/api/
    ├── PiiEncryptionTest.kt   ← AES-256-GCM round-trip tests
    └── ContractBaselineTest.kt ← HTTP smoke test vs Hono (contract capture)
```

Key technical decisions for this phase:

1. Ktor version: 3.x (current stable as of 2026)
2. Exposed version: 0.54.x (current stable) — use DAO-style for complex entities, DSL-style for simple queries
3. Flyway baseline: `V1__baseline.sql` runs `SET search_path = public` + marks all existing tables as already migrated. No DDL changes.
4. PII encryption: `PiiEncryptionTest` must verify the Kotlin implementation produces the same ciphertext format `v1:<iv_hex>:<tag_hex>:<ciphertext_hex>` as the TypeScript implementation. Test vectors must be generated FROM the Hono system and validated against.
5. JWT: Both services use identical `JWT_SECRET` env var. `jose` library in Hono uses HS256 by default — Ktor must use the same algorithm and same claim structure.

Acceptance criteria:

- [ ] `./gradlew build` succeeds with zero errors
- [ ] `./gradlew test` passes `PiiEncryptionTest` with 5+ test vectors from Hono
- [ ] Flyway baseline runs cleanly against dev PostgreSQL without modifying any table
- [ ] Ktor starts on port 8080 and returns 200 on `GET /health`
- [ ] All 24 Exposed Table objects defined with correct column types
- [ ] Contract baseline test captures all Hono endpoint response shapes

Dependencies: None. Can start immediately.

Risk: LOW. No traffic routing change. Ktor not yet in production path.

Phase 1: Health + Auth (JWT + BankID OIDC)

Route files migrated:

- `health.ts` → `HealthRoutes.kt`
- `auth.ts` → `AuthRoutes.kt`
- `admin-auth.ts` → `AdminAuthRoutes.kt`

Endpoints:

```
GET  /v1/health                    → HealthRoutes
GET  /v1/auth/bankid/initiate      → AuthRoutes (BankID OIDC initiation)
POST /v1/auth/bankid/callback      → AuthRoutes (BankID OIDC token exchange)
POST /v1/auth/send-otp             → AuthRoutes
POST /v1/auth/verify-otp           → AuthRoutes
POST /v1/auth/login                → AuthRoutes (demo/dev only)
POST /v1/auth/register             → AuthRoutes (demo/dev only)
POST /v1/auth/demo-login           → AuthRoutes
POST /v1/auth/demo-admin-login     → AuthRoutes
GET  /v1/auth/me                   → AuthRoutes
POST /v1/auth/logout               → AuthRoutes
POST /v1/auth/refresh              → AuthRoutes
POST /v1/auth/demo/login           → AuthRoutes
POST /v1/admin-auth/*              → AdminAuthRoutes
```

Tables: sessions, refresh_tokens, otp_codes, oidc_states (+ users read-only)

Duration estimate: L (5-7 days)

BankID OIDC implementation notes: The TypeScript implementation uses `initiateOIDC`, `exchangeAndVerify`, `findOrCreateUser`, `validateAndConsumeState`. These must be reimplemented in Kotlin using the same PKCE flow:

1. Generate `state` (random, stored in `oidc_states` with nonce + expiry)
2. Build authorization URL with `response_type=code`, `scope=openid profile`, `code_challenge` (S256)
3. Callback receives `code` + `state`, validate state from DB, exchange code for tokens via HTTP
4. Verify ID token signature (JWK endpoint from BankID OIDC discovery)
5. Extract `sub` (fødselsnummer hash) and user claims
6. `findOrCreateUser`: lookup by `national_id_hash`, create if new

Critical: The `state` + `nonce` validation must be byte-for-byte identical in behavior to the TypeScript version. Any difference here breaks mobile login flow permanently.

Traffic routing (nginx): After Phase 1 validation passes contract tests:

```
location /v1/health    { proxy_pass http://ktor:8080; }
location /v1/auth/     { proxy_pass http://ktor:8080; }
location /v1/          { proxy_pass http://hono:3001; }
```

Acceptance criteria:

- [ ] BankID OIDC full flow tested with staging BankID (not just stub)
- [ ] JWT tokens issued by Ktor are accepted by Hono's `authMiddleware` (shared secret)
- [ ] JWT tokens issued by Hono are accepted by Ktor's JWT plugin (shared secret)
- [ ] `POST /v1/auth/logout` revokes session in shared Redis AND `sessions` table
- [ ] OTP rate limiting: 5 OTP sends per hour per phone number
- [ ] Demo mode respected (`DROP_MODE=demo` env var bypasses BankID)
- [ ] Contract test: all Hono auth response shapes match Ktor auth response shapes exactly

Dependencies: Phase 0 complete.

Risk: HIGH. Auth is the most critical path. A JWT incompatibility breaks ALL authenticated routes. Mitigation: keep Hono auth route live alongside Ktor during test period. Only cut over to Ktor auth after 48h of parallel validation.

Phase 2: Core Entities (Users, Settings, Sessions)

Route files migrated:

- `user.ts` → `UserRoutes.kt`
- `settings.ts` → `SettingsRoutes.kt`

Endpoints:

```
DELETE /v1/user/account            → UserRoutes (GDPR erasure request)
GET    /v1/user/export             → UserRoutes (GDPR Art. 20 data export)
PATCH  /v1/user/profile            → UserRoutes
GET    /v1/settings                → SettingsRoutes
PUT    /v1/settings                → SettingsRoutes
GET    /v1/settings/preferences    → SettingsRoutes
PUT    /v1/settings/preferences    → SettingsRoutes
```

Tables: users (write), settings, user_preferences, data_access_requests

Duration estimate: M (3-4 days)

PII encryption note: `DELETE /v1/user/account` is a soft-delete (sets `deleted_at`). The TypeScript version also creates a `data_access_requests` entry with `request_type=erasure`. This must be replicated exactly. Do not actually delete the row — AML 5-year retention requirement.

User data export (`GET /v1/user/export`) must decrypt `national_id_encrypted` using the PII key — this is the ONLY place in the system where decryption happens. Verify the Kotlin `PiiEncryption.decrypt()` function against Phase 0 test vectors.

Acceptance criteria:

- [ ] `PATCH /v1/user/profile` validates input (sanitization equivalent to TypeScript `sanitizeText`)
- [ ] `DELETE /v1/user/account` sets `deleted_at`, revokes all sessions, creates erasure record — verified in DB
- [ ] `national_id_encrypted` decryption produces correct plaintext (test vector from Phase 0)
- [ ] Settings CRUD operations match Hono response shapes exactly
- [ ] Rate limiting applied: user routes inherit global 100 req/min per IP

Dependencies: Phase 1 complete (needs JWT validation).

Risk: MEDIUM. PII handling is sensitive. Mitigation: Phase 0 test vectors validate decryption before this phase starts.

Phase 3: Transactions + Recipients + Exchange Rates

Route files migrated:

- `transactions.ts` → `TransactionRoutes.kt`
- `recipients.ts` → `RecipientRoutes.kt`
- `rates.ts` → `RateRoutes.kt`

Endpoints:

```
GET  /v1/transactions              → TransactionRoutes (list, paginated)
GET  /v1/transactions/analytics   → TransactionRoutes
GET  /v1/transactions/summary     → TransactionRoutes
POST /v1/transactions/remittance  → TransactionRoutes (PISP initiation — HIGH RISK)
```

```
POST /v1/transactions/qr-payment  → TransactionRoutes (PISP QR — HIGH RISK)
POST /v1/transactions/disclosure  → TransactionRoutes
GET  /v1/transactions/:id         → TransactionRoutes
GET  /v1/transactions/:id/receipt → TransactionRoutes
GET  /v1/recipients               → RecipientRoutes
POST /v1/recipients               → RecipientRoutes
PUT  /v1/recipients/:id           → RecipientRoutes
DELETE /v1/recipients/:id       → RecipientRoutes
GET  /v1/rates                    → RateRoutes (public, no auth)
GET  /v1/rates/:currency          → RateRoutes
```

Tables: transactions, recipients, exchange_rates, circuit_breaker_state

Duration estimate: L (6-8 days)

PISP implementation — critical notes: The remittance and QR payment routes are the highest-risk endpoints in the system. They initiate real money movement. The TypeScript implementation contains:

1. **Idempotency key validation** — `transactions.idempotency_key` + `uniqueIndex`. Kotlin must validate this BEFORE any external PISP call. HTTP 409 on duplicate.
2. **Rate lock** — `rate_lock_expires_at` column. If rate lock expired, re-fetch rate before proceeding.
3. **Circuit breaker** — `circuit_breaker_state` table + in-memory registry (`circuitBreakerRegistry`). Kotlin must implement the same state machine: CLOSED → OPEN (after N failures) → HALF_OPEN (after timeout) → CLOSED.
4. **PISP attempt tracking** — `pisp_attempts`, `pisp_timeout_count`, `pisp_last_attempt_at`.
5. **Refund tracking** — `refund_status`, `refund_amount_ore`, `refunded_at`.

All monetary amounts MUST remain in integer øre. No floating point arithmetic on money values.

Acceptance criteria:

- [ ] `POST /v1/transactions/remittance` returns 409 on duplicate idempotency key
- [ ] Circuit breaker transitions: CLOSED→OPEN after 5 failures, HALF_OPEN after 30s, CLOSED after 2 successes
- [ ] Amount calculations use integer arithmetic only (BigInteger/Long in Kotlin — no Double)
- [ ] `GET /v1/transactions` pagination matches Hono response shape (cursor-based or offset?)
- [ ] Exchange rates endpoint returns same structure as Hono (check `to_currency`, `rate`, `updated_at` field names)
- [ ] Contract tests pass for all 8 transaction endpoints

Dependencies: Phase 2 complete.

Risk: VERY HIGH. Money movement. Mitigation: PISP routes are NOT cut over to Ktor until 5 days of parallel logging (same request sent to both, responses compared, no actual PISP calls doubled).

Phase 4: Banking (AISP, Open Banking Consents, Bank Accounts)

Route files migrated:

- `ob-consents.ts` → `ObConsentRoutes.kt`

Endpoints:

```
GET  /v1/ob-consents                       → ObConsentRoutes (list active consents)
POST /v1/ob-consents                       → ObConsentRoutes (create consent)
DELETE /v1/ob-consents/:id                → ObConsentRoutes (revoke consent)
GET  /v1/ob-consents/:id/accounts         → ObConsentRoutes (AISP: list bank accounts)
POST /v1/ob-consents/:id/refresh          → ObConsentRoutes (refresh AISP balance)
```

Tables: ob_consents, bank_accounts

Duration estimate: M (3-5 days)

PSD2 / Berlin Group compliance notes:

1. **90-day consent expiry** — enforced in `ob_consents.expires_at`. Ktor must reject AISP calls when `status != 'active'` OR `expires_at < now()`.
2. **4 AISP calls per day** — `access_count_today` counter + `last_access_date` for daily reset. This is a Berlin Group RTS requirement. The counter logic must be atomic (use PostgreSQL row-level lock or advisory lock to prevent race conditions).
3. **Consent revocation** — sets `status = 'revoked'`, `revoked_at = now()`. Also logs to `audit_log` with action `CONSENT_REVOKED`.

Acceptance criteria:

- [ ] AISP call rejected when consent expired — HTTP 403 with `consent_expired` error code
- [ ] AISP call rejected at count > 4/day — HTTP 429 with `daily_limit_exceeded`
- [ ] Daily counter resets at midnight (compared by `last_access_date` string, YYYY-MM-DD)
- [ ] Consent creation writes to `audit_log`
- [ ] Consent revocation writes to `audit_log`
- [ ] Contract tests pass for all 5 ob-consent endpoints

Dependencies: Phase 3 complete.

Risk: HIGH (PSD2 regulatory compliance). Mitigation: Parallel run against Hono for 72h before cutover.

Phase 5: Compliance (AML, KYC, GDPR, Consents, Disputes)

Route files migrated:

- `consents.ts` → `ConsentRoutes.kt`
- `complaints.ts` → `ComplaintRoutes.kt`
- `disputes.ts` → `DisputeRoutes.kt`
- `reports.ts` → `ReportRoutes.kt` (admin only)

Endpoints:

```
GET  /v1/consents                  → ConsentRoutes (list user consents)
POST /v1/consents                  → ConsentRoutes (record consent grant)
DELETE /v1/consents/:id           → ConsentRoutes (withdraw consent)
POST /v1/complaints                → ComplaintRoutes (submit complaint)
GET  /v1/complaints                → ComplaintRoutes (list user complaints)
GET  /v1/complaints/:id            → ComplaintRoutes
POST /v1/disputes                  → DisputeRoutes (submit dispute)
GET  /v1/disputes                  → DisputeRoutes (list user disputes)
GET  /v1/disputes/:id              → DisputeRoutes
GET  /v1/admin/reports/transactions → ReportRoutes (admin — CSV/JSON export)
GET  /v1/admin/reports/aml         → ReportRoutes (admin — AML summary)
```

Tables: consents, data_access_requests, complaints, disputes, audit_log, aml_alerts, str_reports, screening_results, withdrawal_requests

Duration estimate: L (5-7 days)

Compliance notes:

1. **PSD2 complaint SLA** — `complaints` must be acknowledged within 15 business days (EU PSD2 Art. 101). Ktor does not need to enforce this programmatically, but the `status` field transitions must be correct.
2. **AML audit trail** — ALL writes to `aml_alerts` and `str_reports` must produce `audit_log` entries. This is a Finanstilsynet requirement.

3. **GDPR consent** — `consents.withdrawn_at` is nullable. Withdrawal sets this timestamp. Do NOT delete the consent record (audit trail requirement).
4. **Disputes** — `sla_deadline` is 15 business days from submission. Kotlin must calculate this correctly (exclude Norwegian public holidays if the calculation is done here — check if Hono does this or delegates to a cron job).
5. **Reports** — CSV export must use the same column names and encoding as Hono. Frontend may parse this directly.

Acceptance criteria:

- [ ] Consent withdrawal sets `withdrawn_at`, does not delete row
- [ ] Complaint submission produces `audit_log` entry
- [ ] AML alert creation produces `audit_log` entry with `severity` mapped correctly
- [ ] Dispute `sla_deadline` calculation matches Hono output (test with known input date)
- [ ] CSV export column names match Hono exactly
- [ ] All endpoints require appropriate auth (`authMiddleware` or `adminMiddleware`)
- [ ] Contract tests pass for all compliance endpoints

Dependencies: Phase 4 complete.

Risk: MEDIUM. These are compliance-critical but not on the hot transaction path. Errors here are serious but not immediately money-losing.

Phase 6: Operations (Notifications, Cron, Webhooks, Metrics, Settlement)

Route files migrated:

- `notifications.ts` → `NotificationRoutes.kt`
- `cron.ts` → `CronRoutes.kt`
- `metrics.ts` → `MetricsRoutes.kt`
- `webhooks.ts` → `WebhookRoutes.kt`

Endpoints:

```
GET  /v1/notifications             → NotificationRoutes (list, paginated)
POST /v1/notifications/mark-read  → NotificationRoutes
DELETE /v1/notifications/:id      → NotificationRoutes
POST /v1/push-tokens              → NotificationRoutes (register Expo push token)
DELETE /v1/push-tokens/:token     → NotificationRoutes
```

```
GET  /v1/cron/retention          → CronRoutes (data retention — IP-restricted)
GET  /v1/cron/rates              → CronRoutes (FX rate refresh)
GET  /v1/cron/settlement         → CronRoutes (daily settlement)
GET  /v1/cron/reconciliation     → CronRoutes (daily reconciliation)
GET  /v1/metrics                 → MetricsRoutes (Prometheus format)
POST /v1/webhooks/payment        → WebhookRoutes (banking partner webhook)
POST /v1/webhooks/partner/*      → WebhookRoutes
```

Tables: notifications, push_tokens, webhook_events, webhook_dlq, settlement_batches, settlement_items, reconciliation_reports, reconciliation_discrepancies

Duration estimate: L (5-7 days)

Webhook security — critical: The TypeScript implementation uses `timingSafeEqual` for HMAC signature validation. Kotlin MUST use a constant-time comparison for webhook secrets. Java's `MessageDigest.isEqual()` is NOT constant-time. Use `org.bouncycastle` or implement explicitly.

Settlement notes: `runDailySettlement` and `runDailyReconciliation` are called from cron routes. The Kotlin implementation must use database transactions (Exposed `transaction {}` block) for settlement batch creation — partial writes must roll back.

Cron route security: TypeScript version uses IP allowlist check (`getClientIp` + rate limit). Kotlin must replicate this — cron routes are NOT behind JWT auth, they are behind IP filtering. This is a security boundary.

Metrics: Prometheus `/v1/metrics` endpoint — use Ktor + `micrometer-prometheus` (compatible with `prom-client` metric names if configured correctly). Verify Grafana dashboard still works after cutover.

Acceptance criteria:

- [ ] Webhook HMAC validation uses constant-time comparison
- [ ] Duplicate webhook (`webhook_events.webhook_id` unique constraint) returns 200, not 500
- [ ] Dead letter queue write happens on processing failure
- [ ] Settlement batch creation is transactional (rollback on partial failure)
- [ ] Cron routes return 403 for non-allowlisted IPs
- [ ] Prometheus metrics endpoint returns valid exposition format
- [ ] Push token registration is idempotent (same token, same user = upsert not duplicate)

Dependencies: Phase 5 complete.

Risk: MEDIUM. Operational, not user-facing critical path.

Phase 7: Admin + Merchants + Cards + Remaining

Route files migrated:

- `admin.ts` → `AdminRoutes.kt`
- `merchants.ts` → `MerchantRoutes.kt`
- `cards.ts` → `CardRoutes.kt`
- `withdrawal.ts` → `WithdrawalRoutes.kt`
- `openapi.ts` → `OpenApiRoutes.kt`

Endpoints:

```
GET  /v1/admin/audit               → AdminRoutes (paginated audit log)
GET  /v1/admin/users               → AdminRoutes (user list)
PATCH /v1/admin/users/:id/kyc     → AdminRoutes (KYC status update)
GET  /v1/admin/aml                 → AdminRoutes (AML alert list)
PATCH /v1/admin/aml/:id           → AdminRoutes (AML alert update)
GET  /v1/admin/circuit-breakers   → AdminRoutes (circuit breaker status)
POST /v1/admin/circuit-breakers/reset → AdminRoutes
GET  /v1/merchants                 → MerchantRoutes
POST /v1/merchants                 → MerchantRoutes
GET  /v1/merchants/:id             → MerchantRoutes
PUT  /v1/merchants/:id             → MerchantRoutes
POST /v1/merchants/:id/qr         → MerchantRoutes (QR code generation)
GET  /v1/cards                     → CardRoutes (feature-flagged)
POST /v1/cards                     → CardRoutes (feature-flagged)
PUT  /v1/cards/:id/freeze         → CardRoutes (feature-flagged)
POST /v1/withdrawal                → WithdrawalRoutes (angrerett)
GET  /v1/openapi.json              → OpenApiRoutes
```

Tables: (all remaining) cards, spending_limits, merchants

Duration estimate: M (3-5 days)

Cards note: Cards are feature-flagged off in production. Implement the routes but gate them behind `feature.cards.enabled` config flag. This mirrors the TypeScript feature flag behavior.

QR HMAC note: `merchants.qr_hmac_key` is a 32-byte hex secret per merchant. QR code validation uses HMAC-SHA256 over `merchantId:amount:timestamp`. Kotlin must replicate this exactly — any difference breaks the mobile QR scanner.

Admin portal middleware: `adminPortalMiddleware` in TypeScript uses a separate session table (`admin_sessions`) and MFA verification. Verify the admin session schema exists in the DB — it may be in a migration not yet in `schema.ts`.

Acceptance criteria:

- [ ] QR HMAC validation produces identical results to TypeScript for test vectors
- [ ] Cards routes return 404 when `feature.cards.enabled=false`
- [ ] Admin MFA check enforced (`mfa_required` error if MFA enabled but not verified)
- [ ] Audit log pagination is cursor-based (consistent with Hono behavior)
- [ ] OpenAPI spec served at `/v1/openapi.json`

Dependencies: Phase 6 complete.

Risk: LOW-MEDIUM. Admin and feature-flagged routes are not on critical user path.

Phase 8: Cutover

Goal: Switch all traffic to Ktor. Decommission Hono.

Duration estimate: S (1-2 days + monitoring period)

Cutover checklist:

Pre-cutover (must all pass):

- [ ] All phases 1-7 have been running in parallel for minimum 48h each
- [ ] Zero contract test failures across all endpoints
- [ ] Prometheus metrics match between Hono and Ktor for equivalent traffic
- [ ] No errors in Sentry from Ktor (baseline noise only)
- [ ] Load test with k6 smoke script passes on Ktor
- [ ] PII decryption test passes on production PII_ENCRYPTION_KEY (test vector from a real user record — use a test account)
- [ ] BankID OIDC full flow tested on staging with real BankID

Cutover procedure:

1. Enable maintenance mode for 5 minutes (display "Vi jobber" in app)
2. Update nginx config: ALL traffic → Ktor :8080
3. Remove Hono from nginx upstream (keep container running, just off traffic)
4. Wait 15 minutes, monitor Sentry + Grafana

5. If zero critical errors: disable maintenance mode
6. If errors: roll nginx back to Hono (5 minute downtime max)
7. After 72h stable: decommission Hono container and TypeScript API codebase

Rollback SLA: Under 5 minutes (nginx config change only, no DB change needed).

Acceptance criteria:

- [ ] All k6 smoke tests pass on Ktor with 0% error rate
- [ ] p99 latency within 20% of Hono baseline
- [ ] Zero 5xx errors in first 30 minutes post-cutover
- [ ] BankID login works end-to-end on mobile
- [ ] At least one successful remittance transaction through Ktor

---

Risk Register

Risk	Probability	Impact	Mitigation
JWT incompatibility breaks all auth	MEDIUM	CRITICAL	Phase 0 and 1 validate cross-service token validation before any traffic cut
PII decryption produces wrong plaintext	LOW	CRITICAL	Phase 0 builds test vectors from Hono output; Phase 2 validates
PISP idempotency race condition	MEDIUM	HIGH	Use PostgreSQL advisory lock or `INSERT ... ON CONFLICT` with row-level lock
BankID OIDC state validation mismatch	MEDIUM	HIGH	Phase 1 tests against real staging BankID, not stub
Settlement batch partial write	LOW	HIGH	Exposed `transaction {}` block required; tested explicitly
Webhook HMAC timing attack	LOW	MEDIUM	Constant-time comparison enforced in Phase 6 acceptance criteria
Cards QR HMAC mismatch	LOW	MEDIUM	Phase 7 test vectors from TypeScript
Cron routes exposed without IP filter	MEDIUM	MEDIUM	Replicated in Phase 6; tested with IP outside allowlist

Risk	Probability	Impact	Mitigation
Berlin Group 4/day AISP race condition	MEDIUM	MEDIUM	Atomic counter update with PostgreSQL row lock in Phase 4
Monetary float arithmetic	LOW	CRITICAL	All amounts in Long/BigInteger; enforced by code review in Phase 3

Agent Assignment Model

Each phase runs as a single dedicated builder agent. DO NOT combine phases.

Phase	Agent Type	Priority	Notes
0	Builder (Sonnet)	H	Unblocks all other phases
1	Builder (Sonnet)	H	Auth — pair with Validator immediately
2	Builder (Sonnet)	H	After Phase 1 validated
3	Builder (Sonnet)	H	Highest risk, separate Validator required
4	Builder (Sonnet)	H	PSD2 compliance — Validator required
5	Builder (Sonnet)	M	Can start after Phase 4
6	Builder (Sonnet)	M	Can start after Phase 5
7	Builder (Sonnet)	M	Can start after Phase 6
8	John + Alem	H	Cutover is an operational decision, not a build task

Each builder agent MUST:

1. Read this plan first (ZAKON #18)
2. Read the specific route file(s) for their phase
3. Read the relevant Exposed ORM documentation from Phase 0 output
4. Run contract tests after every endpoint implementation
5. NOT proceed to the next endpoint if contract test fails

Parallel Running Strategy — nginx Configuration

Stage 1 (after Phase 1 cutover):

```
upstream hono  { server localhost:3001; }
upstream ktor  { server localhost:8080; }

location /v1/health     { proxy_pass http://ktor; }
location /v1/auth/      { proxy_pass http://ktor; }
location /v1/           { proxy_pass http://hono; }
```

Stage 3 (after Phase 3 cutover):

```
location /v1/health        { proxy_pass http://ktor; }
location /v1/auth/         { proxy_pass http://ktor; }
location /v1/user/         { proxy_pass http://ktor; }
location /v1/settings/     { proxy_pass http://ktor; }
location /v1/transactions/ { proxy_pass http://ktor; }
location /v1/recipients/   { proxy_pass http://ktor; }
location /v1/rates/        { proxy_pass http://ktor; }
location /v1/              { proxy_pass http://hono; }
```

Continue pattern through Phase 7. nginx location blocks are ordered most-specific first.

AWS App Runner note: If running on AWS App Runner (current infrastructure), the nginx layer may need to be added as an intermediate ECS task or ALB listener rule. The principle is the same — route by path prefix.

Effort Summary

Phase	Effort	Cumulative
0 — Foundation	L (4-6 days)	4-6d
1 — Auth	L (5-7 days)	9-13d
2 — Core Entities	M (3-4 days)	12-17d
3 — Transactions	L (6-8 days)	18-25d

Phase	Effort	Cumulative
4 — Banking/AISP	M (3-5 days)	21-30d
5 — Compliance	L (5-7 days)	26-37d
6 — Operations	L (5-7 days)	31-44d
7 — Admin + Cards	M (3-5 days)	34-49d
8 — Cutover	S (1-2 days)	35-51d

Total: 35-51 builder-days. With parallel Sonnet agents running 2 phases simultaneously (where dependencies allow), wall-clock time is approximately 3-4 weeks.

S = 1-2 days | M = 3-5 days | L = 5-8 days

What the Next Agent (Phase 0 Builder) Must Do First

1. Read `/Users/makinja/ALAI/products/Drop/src/shared/db/schema.ts` (full file)
2. Read `/Users/makinja/ALAI/products/Drop/src/shared/crypto/pii-encryption.ts`
3. Read `/Users/makinja/ALAI/products/Drop/src/drop-api/src/server.ts` (route registration order)
4. Create `/Users/makinja/ALAI/products/Drop/src/drop-api-ktor/` directory
5. Initialize Gradle project with Ktor 3.x + Exposed + Flyway + Kotest dependencies
6. Implement all 24 Exposed Table objects (matching column names exactly)
7. Implement `PiiEncryption.kt` — AES-256-GCM with same `v1:<iv>:<tag>:<ciphertext>` format
8. Write `PiiEncryptionTest.kt` with test vectors generated from the TypeScript implementation
9. Write Flyway `V1__baseline.sql` that does nothing (baseline only)
10. Verify `./gradlew test` passes before marking phase done

Plan authored after reading: BUILD-BLUEPRINT.md, all 22 route files, schema.ts (full), middleware/auth.ts, and lumiscare-to-lobby-migration-plan.md for architectural patterns.

Revision #2
Created 2026-04-02 12:37:59 UTC by John
Updated 2026-05-31 20:05:26 UTC by John