Migration Strategy

Migration Strategy: SQLite to PostgreSQL

> **STATUS: COMPLETED (2026-03-03)** This document describes the completed migration from the old dual-driver architecture to PostgreSQL-only. The migration is done. Current architecture: PostgreSQL 16 (all environments), Drizzle ORM. See ADR-014 for the authoritative current state.

Version: 1.0 **Date:** 2026-02-21 **Status:** Completed — migration done per ADR-014 **Owner:** Database Architect

---

Overview

> **HISTORICAL NOTE:** The dual-driver architecture and `better-sqlite3` dependency described in this document have been removed. The codebase now uses Drizzle ORM with PostgreSQL 16 exclusively. `db.ts` and `USE_PG` no longer exist. See ADR-014.

This document captures the migration plan that was executed when transitioning from SQLite (development) + PostgreSQL (production) to PostgreSQL 16 in all environments. It is preserved as a historical record.

Migration Execution Flow

```
flowchart TD
    A[Phase 1: Prepare] --> B[Phase 2: Schema Migration]
```

```
B --> C[Phase 3: Data Migration]
C --> D[Phase 4: Validation]
D --> E{All checks pass?}
E -->|Yes| F[Phase 5: Cutover]
E -->|No| G[Fix issues]
G --> D
F --> H[Phase 6: Post-migration]

subgraph "Phase 1: Prepare"
    A1[Provision PostgreSQL instance]
    A2[Configure DATABASE_URL]
    A3[Create shadow database for testing]
    A4[Backup SQLite database file]
end

subgraph "Phase 2: Schema"
    B1[Run PostgreSQL schema DDL]
    B2[Create indexes]
    B3[Verify constraints]
end

subgraph "Phase 3: Data"
    C1[Export SQLite data as INSERT statements]
    C2[Transform data types]
    C3[Load into PostgreSQL]
    C4[Reset sequences]
end

subgraph "Phase 4: Validate"
    D1[Row count comparison]
    D2[Checksum validation]
    D3[Application smoke tests]
    D4[Run full test suite against PG]
end

subgraph "Phase 5: Cutover"
    F1[Set DATABASE_URL in production]
    F2[Deploy application]
    F3[Verify health endpoint]
end
```

```
    subgraph "Phase 6: Post-migration"
        H1[Monitor error rates]
        H2[Monitor query performance]
        H3[Archive SQLite file]
    end
```

Data Type Mapping

The dual-driver layer already handles SQL syntax differences. The schema migration must map SQLite types to PostgreSQL equivalents:

SQLite Type	PostgreSQL Type	Tables Using It	Notes
`TEXT`	`TEXT`	All tables	Direct mapping, no change
`TEXT PRIMARY KEY`	`TEXT PRIMARY KEY`	All except `exchange_rates`	Same behavior
`INTEGER` (boolean)	`BOOLEAN` or `INTEGER`	`sessions.revoked`, `notifications.read`, `settings.push_enabled`, `settings.email_enabled`, `bank_accounts.is_primary`, `consents.granted`, `users.sanctions_cleared`	Keep as INTEGER for dual-driver compat, or convert to BOOLEAN in PG-only mode
`INTEGER` (currency)	`BIGINT`	`transactions.amount`, `transactions.fee`, `transactions.send_amount`, `transactions.receive_amount`, `bank_accounts.balance`, `spending_limits.amount`	Use BIGINT for amounts in minor units to prevent overflow
`INTEGER PRIMARY KEY AUTOINCREMENT`	`SERIAL PRIMARY KEY`	`exchange_rates.id`	Only auto-increment in schema
`INTEGER` (unix timestamp)	`INTEGER`	`rate_limits.reset_at`	Unix epoch, no conversion needed
`REAL`	`DOUBLE PRECISION`	`transactions.exchange_rate`, `merchants.fee_rate`	Direct mapping
`TEXT DEFAULT (datetime('now'))`	`TEXT DEFAULT CURRENT_TIMESTAMP`	All `created_at`, `updated_at` columns	Handled by `adaptSqlForPg()` in db.ts

SQLite-specific SQL Adaptations (already implemented in `db.ts:46-52`)

SQLite SQL	PostgreSQL Equivalent	Handler
`INSERT OR IGNORE INTO`	`INSERT INTO ... ON CONFLICT DO NOTHING`	`runIgnore()` function
`INSERT OR REPLACE INTO`	`INSERT INTO ... ON CONFLICT (col) DO UPDATE SET`	`runUpsert()` function
`datetime('now')`	`CURRENT_TIMESTAMP`	`adaptSqlForPg()` regex
`?` placeholders	`$1, $2, ...` positional	`convertPlaceholders()`
`randomblob(32)`	`gen_random_bytes(32)`	Schema-level change (merchants.qr_hmac_key default)
`hex()`	`encode(..., 'hex')`	Schema-level change

PostgreSQL Schema DDL

The PostgreSQL schema must be created separately from the SQLite schema since `CREATE TABLE IF NOT EXISTS` syntax is shared but defaults and functions differ:

```
-- PostgreSQL schema for Drop
-- Run once when provisioning production database

CREATE TABLE IF NOT EXISTS users (
    id TEXT PRIMARY KEY,
    email TEXT UNIQUE NOT NULL,
    password_hash TEXT NOT NULL DEFAULT 'EIDONLY',
    auth_provider TEXT DEFAULT 'bankid',
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    phone TEXT,
    date_of_birth TEXT,
    kyc_status TEXT DEFAULT 'pending' CHECK(kyc_status IN ('pending','approved','rejected')),
    role TEXT DEFAULT 'user' CHECK(role IN ('user','merchant')),
    risk_level TEXT DEFAULT 'low' CHECK(risk_level IN ('low','medium','high')),
    pep_status TEXT DEFAULT 'not_checked' CHECK(pep_status IN
('not_checked','clear','match','pending_review')),
    sanctions_cleared INTEGER DEFAULT 0,
    kyc_method TEXT CHECK(kyc_method IN ('bankid','document','simplified')),
    kyc_verified_at TEXT,
    national_id_hash TEXT,
    deleted_at TEXT,
```

```
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_users_national_id ON users(national_id_hash) WHERE
national_id_hash IS NOT NULL;

CREATE TABLE IF NOT EXISTS recipients (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    name TEXT NOT NULL,
    country TEXT NOT NULL,
    currency TEXT NOT NULL,
    bank_account TEXT NOT NULL,
    bank_name TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_recipients_user ON recipients(user_id);

CREATE TABLE IF NOT EXISTS merchants (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    business_name TEXT NOT NULL,
    org_number TEXT UNIQUE NOT NULL,
    address TEXT,
    bank_account TEXT NOT NULL,
    fee_rate DOUBLE PRECISION DEFAULT 0.01,
    status TEXT DEFAULT 'active',
    qr_hmac_key TEXT NOT NULL DEFAULT encode(gen_random_bytes(32), 'hex'),
    created_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS transactions (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    type TEXT NOT NULL CHECK(type IN ('remittance','qr_payment')),
    status TEXT DEFAULT 'processing' CHECK(status IN ('processing','completed','failed')),
    amount BIGINT NOT NULL,
    currency TEXT DEFAULT 'NOK',
    fee BIGINT DEFAULT 0,
    recipient_id TEXT REFERENCES recipients(id),
    merchant_id TEXT REFERENCES merchants(id),
```

```
    send_amount BIGINT,
    send_currency TEXT,
    receive_amount BIGINT,
    receive_currency TEXT,
    exchange_rate DOUBLE PRECISION,
    purpose_code TEXT,
    created_at TEXT DEFAULT CURRENT_TIMESTAMP,
    completed_at TEXT,
    idempotency_key TEXT
);
CREATE UNIQUE INDEX IF NOT EXISTS idx_tx_idempotency ON transactions(idempotency_key) WHERE
idempotency_key IS NOT NULL;
CREATE INDEX IF NOT EXISTS idx_transactions_user ON transactions(user_id);

CREATE TABLE IF NOT EXISTS exchange_rates (
    id SERIAL PRIMARY KEY,
    from_currency TEXT DEFAULT 'NOK',
    to_currency TEXT NOT NULL,
    rate DOUBLE PRECISION NOT NULL,
    updated_at TEXT DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS bank_accounts (
    id TEXT PRIMARY KEY,
    user_id TEXT NOT NULL REFERENCES users(id),
    bank_name TEXT NOT NULL,
    account_number TEXT NOT NULL,
    iban TEXT,
    balance BIGINT DEFAULT 0,
    balance_synced_at TEXT,
    currency TEXT DEFAULT 'NOK',
    is_primary INTEGER DEFAULT 0,
    connected_at TEXT DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX IF NOT EXISTS idx_bank_accounts_user ON bank_accounts(user_id);

-- Remaining tables follow the same pattern...
-- See full DDL in migration script
```

Sequence/Auto-increment Migration

Only one table uses auto-increment: `exchange_rates`.

Aspect	SQLite	PostgreSQL	Migration Step
Type	`INTEGER PRIMARY KEY AUTOINCREMENT`	`SERIAL PRIMARY KEY`	Schema DDL change
Sequence reset	N/A (built into rowid)	`SELECT setval('exchange_rates_id_seq', (SELECT MAX(id) FROM exchange_rates))`	After data load
Gap behavior	Gaps allowed	Gaps allowed	No difference

JSON Handling

Aspect	SQLite	PostgreSQL	Impact
Storage type	TEXT (plain string)	TEXT (could use JSONB)	No change needed for compatibility
JSON columns	`audit_log.details`, `aml_alerts.details`, `str_reports.details`, `screening_results.match_details`	Same columns, stored as TEXT	Keep as TEXT for dual-driver compatibility
Querying JSON	Not used (JSON is stored, not queried in SQL)	Could use `->>` / `@>` operators	Future optimization: add JSONB indexes for audit log queries
Validation	None (application layer)	Could add CHECK with `jsonb` cast	Future enhancement

Decision: Keep JSON columns as TEXT for now. Converting to JSONB is a future optimization that would break dual-driver compatibility.

Date/Time Handling

Aspect	SQLite	PostgreSQL	Migration
Default value	`datetime('now')`	`CURRENT_TIMESTAMP`	Handled by `adaptSqlForPg()`
Storage format	ISO 8601 TEXT	ISO 8601 TEXT (not TIMESTAMP type)	No conversion needed

Aspect	SQLite	PostgreSQL	Migration
Timezone	UTC (application convention)	UTC (application convention)	Consistent
Date arithmetic	`datetime('now', '-3 days')`	`CURRENT_TIMESTAMP - INTERVAL '3 days'`	Only used in seed data, not production queries

Note: All timestamps are stored as TEXT in ISO 8601 format (`YYYY-MM-DDTHH:MM:SS`) in both databases. This is intentional for dual-driver compatibility. A future PostgreSQL-only optimization could convert to `TIMESTAMPTZ`.

Migration Checklist

Pre-Migration

- ☐ Provision PostgreSQL instance (AWS RDS or equivalent)
- ☐ Configure connection pooling (built-in `pg.Pool`, max connections TBD)
- ☐ Set `DATABASE_URL` environment variable
- ☐ Create shadow database for testing
- ☐ Backup current SQLite file: `cp data/drop.db data/drop.db.backup.$(date +%s)`
- ☐ Run full test suite against SQLite (baseline)
- ☐ Review all raw SQL queries for SQLite-specific syntax (should be none -- all go through `db.ts`)

Schema Migration

- ☐ Run PostgreSQL DDL script to create all 19 tables
- ☐ Create all indexes (11 indexes defined in `db.ts` schema)
- ☐ Verify all CHECK constraints are active
- ☐ Verify all foreign key constraints are active
- ☐ Test `initDb()` function with `DATABASE_URL` set

Data Migration

- ☐ Export SQLite data using `sqlite3 drop.db .dump` or custom export script
- ☐ Transform `INTEGER PRIMARY KEY AUTOINCREMENT` to `SERIAL`

- [ ] Transform `randomblob()` defaults to `gen_random_bytes()`
- [ ] Load data into PostgreSQL
- [ ] Reset `exchange_rates_id_seq` sequence
- [ ] Verify row counts match per table

Validation

- [ ] Row count comparison (all 19 tables)
- [ ] Spot-check 10 records per table for data integrity
- [ ] Run application smoke tests:
  - [ ] Login (BankID flow)
  - [ ] View dashboard (bank accounts, balance)
  - [ ] List transactions
  - [ ] Create remittance
  - [ ] Create QR payment
  - [ ] View notifications
  - [ ] Update settings
- [ ] Run full test suite with `DATABASE_URL` set
- [ ] Verify `GET /api/health` returns `db: "connected"` with acceptable latency

Cutover

- [ ] Set `DATABASE_URL` in production environment
- [ ] Deploy application
- [ ] Verify health endpoint
- [ ] Monitor error rates for 1 hour
- [ ] Monitor query latency for 1 hour

Post-Migration

- [ ] Archive SQLite file
- [ ] Update documentation to reflect PostgreSQL as primary
- [ ] Consider PostgreSQL-specific optimizations (JSONB, TIMESTAMPTZ, partial indexes)
- [ ] Configure automated backups (pg_dump cron or RDS snapshots)

Rollback Procedure

```
flowchart TD
    A[Issue detected in PostgreSQL] --> B{Is it data corruption?}
    B -->|Yes| C[Stop application immediately]
    B -->|No| D{Is it a query/performance issue?}
    D -->|Yes| E[Fix query and redeploy]
    D -->|No| F[Investigate further]

    C --> G[Remove DATABASE_URL env var]
    G --> H[Redeploy application]
    H --> I[Application falls back to SQLite]
    I --> J[Investigate PostgreSQL issue offline]
    J --> K[Fix and retry migration]
```

Rollback is simple: Remove or unset the `DATABASE_URL` environment variable. The application immediately falls back to SQLite. This is the primary advantage of the dual-driver architecture.

Rollback Scenario	Action	Downtime
Schema issue in PostgreSQL	Unset `DATABASE_URL`, redeploy	~2 minutes (deploy time)
Data integrity issue	Unset `DATABASE_URL`, redeploy with SQLite backup	~2 minutes
Performance regression	Unset `DATABASE_URL`, optimize PG offline	~2 minutes
Partial migration failure	Drop PostgreSQL schema, fix script, retry	No production impact (still on SQLite)

Key constraint: Rollback only works if no new data has been written to PostgreSQL that does not exist in SQLite. In practice, this means the migration window should be short and the SQLite database should be read-only during cutover.

Zero-Downtime Migration Using Dual-Driver

The dual-driver architecture enables a phased migration with zero downtime:

```
sequenceDiagram
    participant App as Application
    participant SQLite as SQLite (current)
    participant PG as PostgreSQL (new)

    Note over App,SQLite: Phase A: Normal operation (SQLite)
    App->>SQLite: All reads/writes

    Note over App,PG: Phase B: Provision and schema
    App->>SQLite: All reads/writes (unchanged)
    Note right of PG: Create schema, indexes

    Note over App,PG: Phase C: Data migration
    App->>SQLite: All reads/writes (unchanged)
    Note right of PG: Bulk load from SQLite export

    Note over App,PG: Phase D: Final sync + cutover
    App->>SQLite: Brief read-only mode
    Note right of PG: Delta sync (new records since bulk load)
    Note over App: Set DATABASE_URL
    App->>PG: All reads/writes

    Note over App,PG: Phase E: Production on PostgreSQL
    App->>PG: All reads/writes
    Note left of SQLite: Archived as backup
```

Total downtime: Only during Phase D final sync + deploy, estimated at 2-5 minutes for the current data volume.

Testing Approach

Shadow Database Testing

Before production migration, run the full application against a shadow PostgreSQL database:

1. **Provision shadow PG:** Same version and configuration as production target
2. **Run schema creation:** Execute PostgreSQL DDL
3. **Load production-like data:** Export SQLite demo data, transform, load

4. **Run test suite:** `DATABASE_URL=<shadow> npm test`
5. **Run integration tests:** Full API flow tests against shadow
6. **Load testing:** Verify query performance under expected load

Data Integrity Checks

```
-- Row count comparison (run against both databases)
SELECT 'users' as tbl, COUNT(*) as cnt FROM users
UNION ALL SELECT 'transactions', COUNT(*) FROM transactions
UNION ALL SELECT 'bank_accounts', COUNT(*) FROM bank_accounts
UNION ALL SELECT 'recipients', COUNT(*) FROM recipients
UNION ALL SELECT 'merchants', COUNT(*) FROM merchants
UNION ALL SELECT 'sessions', COUNT(*) FROM sessions
UNION ALL SELECT 'notifications', COUNT(*) FROM notifications
UNION ALL SELECT 'settings', COUNT(*) FROM settings
UNION ALL SELECT 'exchange_rates', COUNT(*) FROM exchange_rates
UNION ALL SELECT 'cards', COUNT(*) FROM cards
UNION ALL SELECT 'spending_limits', COUNT(*) FROM spending_limits
UNION ALL SELECT 'rate_limits', COUNT(*) FROM rate_limits
UNION ALL SELECT 'audit_log', COUNT(*) FROM audit_log
UNION ALL SELECT 'aml_alerts', COUNT(*) FROM aml_alerts
UNION ALL SELECT 'str_reports', COUNT(*) FROM str_reports
UNION ALL SELECT 'screening_results', COUNT(*) FROM screening_results
UNION ALL SELECT 'consents', COUNT(*) FROM consents
UNION ALL SELECT 'data_access_requests', COUNT(*) FROM data_access_requests
UNION ALL SELECT 'complaints', COUNT(*) FROM complaints;
```

Cross-References

- **Dual-driver implementation:** `src/drop-api/src/lib/db.ts`
- **Database schema:** DATABASE-SCHEMA.md
- **Database design:** database-design.md
- **Data architecture:** data-architecture.md
- **Deployment architecture:** deployment-architecture.md
- **Roadmap Phase 2:** ROADMAP.md (PostgreSQL migration is Phase 2)

Revision #10
Created 2026-02-21 05:59:06 UTC by John
Updated 2026-05-23 10:57:14 UTC by John