ADR-010: Dual Database Driver

ADR-010: Dual Database Driver Abstraction

Status: SUPERSEDED by ADR-014: PostgreSQL-Only Architecture (2026-03-03) **Date:** 2026-02-21
Deciders: John (AI Director) **Category:** Database

Context

Per ADR-006, Drop uses SQLite for development and PostgreSQL for production. This creates a challenge: the application code must work correctly against both database engines, which have different SQL dialects, parameter binding, and transaction semantics.

The naive approach -- maintaining two separate codebases or using an ORM -- has drawbacks:

Approach	Type Safety	SQL Control	Performance	Complexity
Raw SQL per driver	Low (string SQL)	Full	Optimal	High (2x code)
ORM (Prisma/Drizzle)	High	Limited	Good (with overhead)	Medium
Thin abstraction layer	Medium	Full	Optimal	Low

An ORM would add a dependency, a build step (Prisma generate), and limit SQL flexibility for complex compliance queries. A thin abstraction layer provides the best balance: same SQL syntax where possible, automatic translation where not.

Decision

Implement a thin database abstraction layer (`db.ts`) that exposes a unified API and transparently converts SQL between SQLite and PostgreSQL dialects.

Driver detection at startup:

```
const USE_PG = !!process.env.DATABASE_URL;
```

```
graph TB
    subgraph app["Application Code"]
        routes["API Routes"]
        routes -->|"query(), getOne(),<br/>run(), transaction()"| dal["Database Access Layer<br/>(db.ts)"]
    end

    subgraph dal_internals["Abstraction Layer Internals"]
        dal --> detect{"DATABASE_URL<br/>set?"}
        detect -->|"Yes"| pg_driver["PostgreSQL Driver<br/>(pg pool)"]
        detect -->|"No"| sqlite_driver["SQLite Driver<br/>(better-sqlite3)"]

        dal --> convert["SQL Converter"]
        convert -->|"? → $1,$2..."| pg_driver
        convert -->|"datetime('now') →<br/>CURRENT_TIMESTAMP"| pg_driver
    end

    subgraph databases["Databases"]
        pg_driver --> pg["PostgreSQL<br/>(production)"]
        sqlite_driver --> sqlite["SQLite<br/>(development)"]
    end
```

Unified API

Function	Signature	Purpose
`query<T>`	`(sql, params?) -> Promise<T[]>`	SELECT, returns array of rows
`getOne<T>`	`(sql, params?) -> Promise<T | null>`	SELECT, returns first row or null
`run`	`(sql, params?) -> Promise<{changes}>`	INSERT/UPDATE/DELETE
`runIgnore`	`(sql, params?) -> Promise<{changes}>`	INSERT OR IGNORE / ON CONFLICT DO NOTHING
`runUpsert`	`(sql, conflictCol, updateCols, params?) -> Promise<{changes}>`	INSERT OR REPLACE / ON CONFLICT DO UPDATE
`transaction<T>`	`(fn) -> Promise<T>`	Atomic transaction wrapper
`initDb`	`() -> Promise<void>`	Schema creation + seed data
`getDriver`	`() -> "pg" | "sqlite"`	Current driver type

SQL Translation Rules (`db.ts:50-59`)

SQLite Syntax	PostgreSQL Equivalent	Handled By
`?` placeholders	`$1, $2, $3, ...`	Automatic in `query()`/`run()`
`INSERT OR IGNORE INTO`	`INSERT INTO ... ON CONFLICT DO NOTHING`	`runIgnore()`
`INSERT OR REPLACE INTO`	`INSERT INTO ... ON CONFLICT (col) DO UPDATE SET`	`runUpsert()`
`datetime('now')`	`CURRENT_TIMESTAMP`	Automatic in SQL string
`INTEGER AUTOINCREMENT`	`SERIAL`	Schema initialization
`TEXT` dates	`TIMESTAMPTZ`	Schema initialization

Consequences

Positive

- Application code is database-agnostic -- same queries work against both engines
- Zero-config local development (SQLite), production-grade in deployment (PostgreSQL)
- No ORM overhead or code generation step
- Full SQL control for complex compliance queries (joins across audit tables)
- Transparent parameter binding conversion
- Transaction semantics unified across both drivers

Negative

- SQL must be compatible with both dialects (no PostgreSQL-specific features like arrays, JSON operators, CTEs with RETURNING)
- Subtle behavioral differences may cause bugs (e.g., SQLite type affinity vs PostgreSQL strict typing)
- `runUpsert()` API is slightly awkward compared to native SQL
- Cannot use advanced PostgreSQL features (partial indexes, LISTEN/NOTIFY, materialized views) through the abstraction

Risks

- **Silent data differences:** SQLite may accept data that PostgreSQL rejects (e.g., inserting text into INTEGER column). Mitigation: CI tests against both databases.

- **Transaction isolation:** SQLite uses serialized transactions (one writer), PostgreSQL uses MVCC. Code that works under SQLite serialization may have race conditions under PostgreSQL MVCC. Mitigation: explicit row locking (`FOR UPDATE`) in critical paths like balance deduction.

References

- ADR-006: SQLite to PostgreSQL -- Database strategy decision
- Database Schema -- Table definitions for both dialects
- Migration Strategy -- Data migration plan
- Database Design -- Database architecture

Revision #7
Created 2026-02-21 05:59:01 UTC by John
Updated 2026-05-23 10:56:51 UTC by John