Architecture Decision Records (ADR)

All architectural decisions and rationale

Architecture Decision Record — ADR-013

Architecture Decision Record — ADR-013

> **Project:** Drop **ADR Number:** ADR-013 **Title:** Neonomics as Open Banking Aggregator for AISP/PISP **Version:** 1.0 **Date:** 2026-02-23 **Author:** Petter Graff, Senior Enterprise Architect **Status:** Proposed **Reviewers:** Alem Bašić (CEO), John (AI Director)

Document History

Version	Date	Author	Changes
0.1	2026-02-23	Petter Graff	Initial draft

ADR Numbering Scheme

Convention: `ADR-{NNN}-{short-slug}.md` — e.g., `ADR-013-neonomics-open-banking-aggregator.md`
Store in: `docs/architecture/adr/`

---

1. Context

1.1 Situation

Drop is a PSD2 pass-through payment application (ADR-003) that requires AISP (read bank balances) and PISP (initiate payments from user's bank) capabilities to function in production. The

current MVP uses mock AISP/PISP — the `NEXT_PUBLIC_SERVICE_MODE=mock` flag returns simulated balances and payment confirmations without contacting real banks.

To go live (Phase 2), Drop must connect to the actual Open Banking APIs of Norwegian and Nordic banks (DNB, SpareBank 1, Nordea, Handelsbanken, etc.) using the Berlin Group NextGenPSD2 standard. Two strategic approaches exist: direct ASPSP integration (per-bank) or aggregator integration (single API covering all banks).

The previous planned provider, Swan (a BaaS provider), has been deprecated and removed from the codebase — it was not aligned with the PSD2 pass-through model (Swan offered embedded banking, not TPP-style AISP/PISP).

1.2 Forces & Constraints

Technical forces:

- Drop must integrate with 10+ Norwegian banks to provide meaningful coverage — each bank has slightly different PSD2 API implementations despite the Berlin Group standard
- eIDAS QWAC and QSeal certificates (required for direct ASPSP integration) take 4-8 weeks to obtain from Buypass or Commfides
- Drop's current backend is a single Hono API — minimal engineering bandwidth for multi-bank integration maintenance

Business forces:

- Time-to-market is critical; Phase 2 Open Banking MVP must launch within 3-4 months of Finanstilsynet license/agent arrangement
- Each direct bank integration requires a bilateral agreement and developer portal onboarding (2-6 weeks each)
- Revenue model depends on remittance transaction volume — delayed Open Banking = zero transaction revenue

Compliance & regulatory:

- Finanstilsynet PISP/AISP license not yet obtained; Drop may operate as an agent under a licensed PSP while license is pending
- GDPR: Data processed by aggregator requires a DPA (Data Processing Agreement) with the aggregator
- PSD2 RTS: SCA (Strong Customer Authentication) must be ASPSP-side — aggregator must support scaRedirect flow (not screen-scraping)

Existing decisions that constrain this:

- ADR-003: PSD2 pass-through model — constrains us to regulated TPP AISP/PISP, not BaaS wallet

- ADR-005: Monolith-first — constrains us to a single integration point, not microservice-per-bank

1.3 Problem Statement

> **We need to decide:** Which Open Banking connectivity strategy should Drop adopt for Phase 2 production AISP/PISP — direct per-ASPSP integration or a single Open Banking aggregator?

---

2. Decision

We will: Use **Neonomics** as the primary Open Banking aggregator for both AISP (balance reads) and PISP (payment initiation) in the Norwegian and Nordic market.

Rationale (summary): Neonomics provides a single REST API endpoint covering 90%+ of Norwegian banks under the Berlin Group NextGenPSD2 standard, handles eIDAS certificate management and bank onboarding, and is a Norwegian company with strong local regulatory relationships — reducing Drop's time-to-market from 6-9 months (direct per-bank) to 6-8 weeks (aggregator contract + integration).

---

3. Alternatives Considered

Option A: Neonomics Aggregator ? Selected

Description: Contract with Neonomics (Norwegian Open Banking aggregator, Oslo HQ) for a single HTTPS REST API covering Norwegian + Nordic banks. Neonomics holds its own PISP/AISP registration with Finanstilsynet and eIDAS certificates — Drop operates as an agent or technology partner under Neonomics' regulatory umbrella initially.

Pros:

- Nordic focus: Deep coverage of DNB, SpareBank 1, Nordea, Sbanken, Handelsbanken, Skandiabanken (90%+ Norwegian market)
- Norwegian company: Strong FSA relationships, Norwegian language support, local regulatory expertise
- Single API: One integration maintains coverage across all banks — bank API updates handled by Neonomics

- eIDAS certs handled by Neonomics: Removes 4-8 week cert procurement from Drop's critical path
- Agent arrangement possible: Drop can operate under Neonomics' license while applying for own license

Cons:

- Per-transaction cost: Aggregator charges per API call / transaction — reduces margin compared to direct ASPSP integration at scale
- Data through third party: All AISP data transits Neonomics infrastructure — requires GDPR DPA
- Neonomics dependency: If Neonomics is acquired or raises prices, switching is a 2-3 month project

Cost/Effort: Contract negotiation 2-4 weeks; technical integration 2-4 weeks — total 6-8 weeks
Risk: MEDIUM — Neonomics is a funded startup (not a Tier-1 bank); business continuity risk mitigated by Tink as backup

---

Option B: Direct Per-ASPSP Integration

Description: Integrate directly with each Norwegian bank's PSD2 developer portal: DNB API, SpareBank 1 Open Banking, Nordea Open Banking, etc. Each bank has its own onboarding process, bilateral agreement, and API specifics.

Pros:

- Lower per-transaction cost at scale: No aggregator margin once integrated
- Data stays bilateral: No third-party aggregator processes user financial data
- Full control: No dependency on aggregator's uptime or pricing

Cons:

- Time: Each bank takes 2-6 weeks to onboard — covering 5 banks = 10-30 weeks minimum
- Ongoing maintenance: Each bank independently updates their API — Drop must track all changes
- eIDAS certificates: Drop must obtain QWAC + QSeal independently (4-8 weeks, ~€3,000-5,000/year)
- Finanstilsynet license: Cannot make direct ASPSP calls without own license or agent arrangement — blocks Phase 2

Cost/Effort: 6-12 months engineering time + bank agreements + cert procurement **Risk:** HIGH — Timeline risk is critical; direct integration too slow for Phase 2 target

Why not chosen: Timeline incompatible with Phase 2 launch target; Direct integration is the long-term (Phase 3+) optimization once scale justifies the margin savings.

Option C: Tink (Visa) Aggregator

Description: Use Tink (acquired by Visa in 2022), the largest European Open Banking aggregator with 6,000+ banks across the EU.

Pros:

- Broadest coverage: 6,000+ banks — future-proof for any European expansion
- Visa backing: Financial stability, enterprise SLAs

Cons:

- Non-Norwegian HQ: Less Nordic specialization; support in English only
- Enterprise pricing: Higher minimum spend than Neonomics
- GDPR: Data processed in Sweden (EU) — adequate, but Neonomics processes in Norway

Cost/Effort: Similar to Neonomics — 6-8 weeks integration **Why not chosen:** Neonomics preferred for Phase 2 Norwegian launch due to local regulatory relationships and Norwegian bank specialization. Tink retained as Backup/Phase 3 (EU expansion) option.

Comparison Matrix

Criterion	Weight	Option A: Neonomics (Selected)	Option B: Direct ASPSP	Option C: Tink
Time-to-market	5	5	1	4
Norwegian bank coverage	5	5	3	4
Per-transaction cost (3yr)	3	3	5	3
GDPR compliance complexity	3	4	5	4
Regulatory / license path	4	5	2	4
Vendor stability	3	3	5	5
Engineering effort	4	5	1	4
Weighted Total		**131**	**77**	**112**

Score: 1 (poor) to 5 (excellent)

---

4. Consequences

4.1 Positive Consequences

- Phase 2 Open Banking live within 8 weeks of contract signing vs. 6-12 months for direct integration
- No eIDAS certificate management burden until Drop obtains its own Finanstilsynet license
- Single endpoint to maintain — bank API changes are Neonomics' responsibility
- Norwegian regulatory expertise from partner reduces compliance risk

4.2 Negative Consequences

- Aggregator per-transaction fee reduces remittance margin by ~0.1-0.3% at scale — *Mitigation: Renegotiate pricing at 10K+ monthly transactions; plan direct integration for Phase 3*
- GDPR DPA with Neonomics required before any AISP data can transit their infrastructure — *Mitigation: DPA negotiated as part of commercial contract*
- Vendor concentration risk — *Mitigation: Document Tink integration as a 6-week fallback migration path*

4.3 Neutral / Secondary Effects

- Drop's Hono API adds a single Neonomics client module (`lib/openbanking/neonomics.ts`) — clean encapsulation means provider can be swapped
- AISP and PISP are separate Neonomics API product lines — may have different pricing tiers

4.4 Technical Debt Created

- The `mock-swan.ts` file (deprecated Swan mock) must be removed and replaced with a `mock-openbanking.ts` compatible with the Neonomics API schema — plan in Phase 2 sprint 1
- *Acceptable because:* Mock removal is low-risk (test infrastructure only) and unblocks clean integration

---

5. Compliance Impact

Regulation	Impact	Notes
GDPR	MEDIUM	AISP balance data and PISP payment data transit Neonomics — GDPR DPA required; Neonomics is EU/EEA entity
PSD2 (Betalingstjenesteloven)	HIGH	Neonomics holds PISP/AISP registration; Drop operates as agent/technology partner until own license obtained
AML (Hvitvaskingsloven)	LOW	Transaction monitoring remains Drop's responsibility regardless of aggregator
DORA	MEDIUM	Neonomics is a critical third-party ICT provider — must be documented in Drop's ICT risk management framework

Data residency implications: Neonomics processes data in Norway/EEA — no cross-border transfer to non-adequate countries.

---

6. Performance Impact

Metric	Before (mock)	After (Neonomics production)	Source
AISP balance read latency	~10ms (mock)	~200-500ms (Neonomics + ASPSP)	Neonomics SLA + Berlin Group typical
PISP initiation latency	~10ms (mock)	~300-800ms (Neonomics + ASPSP)	Neonomics SLA
PISP SCA redirect latency	N/A (mock)	User-dependent (BankID at ASPSP)	External
Availability	N/A (mock)	99.5% SLA (Neonomics)	Neonomics commercial SLA

Performance testing plan: Load test AISP balance reads with 100 concurrent users against Neonomics sandbox before production launch.

---

7. Migration / Implementation Notes

7.1 Migration Plan

```
Phase 2a (Weeks 1-2): Contract + setup
  - [ ] Sign Neonomics commercial contract
  - [ ] Sign GDPR DPA
  - [ ] Obtain Neonomics sandbox API credentials
  - [ ] Create lib/openbanking/neonomics.ts client module

Phase 2b (Weeks 3-4): AISP integration (balance reads)
  - [ ] Implement AISP consent flow (POST /v1/consents)
  - [ ] Implement balance read (GET /v1/accounts/{id}/balances)
  - [ ] Update bank_accounts table (add consent_id, consent_expires_at columns)
  - [ ] Integration test against Neonomics sandbox with DNB test account

Phase 2c (Weeks 5-6): PISP integration (payment initiation)
  - [ ] Implement PISP payment initiation (POST /v1/payments/sepa-credit-transfers)
  - [ ] Implement SCA redirect flow + payment status polling
  - [ ] Add payment webhook receiver for async status updates
  - [ ] Integration test: full remittance flow against Neonomics sandbox

Phase 2d (Weeks 7-8): Production readiness
  - [ ] Switch NEXT_PUBLIC_SERVICE_MODE from 'mock' to 'production'
  - [ ] Production credentials (Neonomics production API key) in AWS Secrets Manager
  - [ ] Remove deprecated mock-swan.ts
  - [ ] Monitoring: Sentry alerts for Neonomics API errors
  - [ ] Circuit breaker: 3 failures in 60s → 60s cooldown
```

7.2 Rollback Strategy

Can we roll back? YES — Feature flag `NEXT_PUBLIC_SERVICE_MODE` reverts to `mock` mode instantly

Rollback steps:

1. Set `NEXT_PUBLIC_SERVICE_MODE=mock` in AWS Secrets Manager
2. Restart App Runner instances — rollback complete in < 5 minutes
3. Users see cached balance; payments queue for retry when production mode re-enabled

7.3 Feature Flags

Flag	Purpose	Default
`NEXT_PUBLIC_SERVICE_MODE`	`mock` = simulated AISP/PISP; `production` = Neonomics live	`mock`

Flag	Purpose	Default
`OPEN_BANKING_PROVIDER`	Future: switch between Neonomics and Tink	`neonomics`

---

8. Related ADRs

ADR	Relationship	Notes
ADR-003	Prerequisite	PSD2 pass-through model requires AISP/PISP provider
ADR-005	Constrained by	Single monolith means single aggregator integration point
ADR-007	Related	BankID used for Drop login SCA; ASPSP-side SCA (for PISP) is separate via Neonomics scaRedirect

---

9. Review Date

Next review: 2026-08-23 (6 months post-decision) or when Neonomics pricing changes by > 50%

Review trigger conditions:

- If Drop processes > 10,000 monthly transactions: evaluate direct ASPSP integration economics
- If Neonomics raises per-transaction pricing by > 50%: evaluate Tink migration
- If Neonomics experiences > 3 outages in 30 days: activate Tink fallback

Superseded by: — *(fill in if this ADR is later superseded)*

Approval

Role	Name	Date	Signature
Author	Petter Graff	2026-02-23	
Tech Lead	John (AI Director)		
Security (if compliance impact)			

Role	Name	Date	Signature
CEO	Alem Bašić		

ADR-001: Consolidate Backends

ADR-001: Consolidate to Single Backend

Status: Accepted **Date:** 2026-02-12 **Deciders:** John (AI Director), Alem (CEO) **Category:** Architecture

---

Context

The Drop codebase contained two competing middleware implementations creating confusion about which was authoritative:

1. **`lib/middleware.ts`** -- A simple, monolithic file used by all 24 API routes. Provided cookie-based JWT auth (`requireAuth`, `requireMerchant`), basic rate limiting via SQLite-backed `rate_limits` table, IP extraction from `X-Forwarded-For`, and standardized JSON error responses (`jsonError`).
2. **`lib/middleware/` directory** -- A more robust, modular implementation with `auth-middleware.ts` (Bearer token auth, in-memory rate limiting with `X-RateLimit-*` headers), `error-handler.ts` (typed `AppError` class with predefined error constructors), and `validation.ts` (comprehensive input validation: phone, amount, IBAN, PIN, email, currency, sanitization). This directory was **completely unused** by any route.

Additionally, **FontelePay** (an earlier project iteration) resided at `src/fontelepay/` with its own `.git/`, `node_modules/`, and independent codebase, blurring project boundaries.

The `SOURCE-STATUS.md` analysis confirmed that Drop source code had never been properly committed to version control. A clean rebuild using build artifacts as specification was the recommended path.

```
graph LR
    subgraph before["Before (Dual Middleware)"]
        routes["24 API Routes"] --> mw1["lib/middleware.ts<br/>(cookie JWT, basic rate limit)"]
        unused["UNUSED"] --> mw2["lib/middleware/<br/>(Bearer JWT, typed errors, validation)"]
```

```
    end

    subgraph after["After (Consolidated)"]
        routes2["24 API Routes"] --> mw3["lib/middleware.ts<br/>(cookie JWT, persistent rate
limit,<br/>CSRF, session revocation)"]
        routes2 --> val["lib/middleware/validation.ts<br/>(input validation, sanitization)"]
    end

    before -->|"ADR-001"| after
```

Decision

1. **Consolidate middleware** into a single implementation, incorporating the best parts of both:
   - Cookie-based JWT auth from `lib/middleware.ts` (matches current route expectations)
   - Typed errors and input validation from `lib/middleware/` directory
   - Persistent rate limiting via `rate_limits` SQLite table (replacing in-memory `Map`)
2. **Remove duplicate code** after consolidation to eliminate confusion about which implementation is authoritative.
3. **Separate FontelePay** from Drop's `src/` directory (see ADR-002).
4. **Remove dead code** including any orphaned backend variants and unused files.

Consequences

Positive

- Single source of truth for middleware behavior
- Eliminates developer confusion about which middleware to use
- Typed errors (`AppError`, `Errors.*`) improve debugging and error reporting
- Input validation (`validateName`, `sanitizeText`, `validateAmount`) applied consistently
- Clean separation between Drop and FontelePay projects
- Reduced codebase size and maintenance burden

Negative

- All 24 API routes needed import path updates during rebuild
- Risk of regression if middleware behavior changed subtly during consolidation
- FontelePay separation required updating any shared references

Risks

- **Behavioral drift:** Consolidated middleware may behave differently from original in edge cases. Mitigation: test suite covering auth, transactions, cards, and merchants.
- **Import breakage:** Route import paths change. Mitigation: rebuild approach writes routes fresh against consolidated middleware.

References

- ADR-002: Separate FontelePay -- Companion decision for FontelePay extraction
- Middleware Documentation -- Current middleware reference
- Security Architecture -- Security controls using consolidated middleware
- Original source: `comms/decisions/ADR-001-consolidate-backends.md`

ADR-002: Separate FonTelePay

ADR-002: Separate FontelePay from Drop Repository

Status: Accepted **Date:** 2026-02-12 **Deciders:** John (AI Director) **Category:** Architecture

---

Context

FontelePay is an earlier iteration of the payment concept that predates the Drop rebrand. It resided at `src/fontelepay/` inside the Drop product directory with:

- Its own `.git/` repository with independent commit history
- Its own `node_modules/` (527 entries)
- Its own `.claude/` configuration directory
- Separate documentation, R&D research, and project structure
- Multiple subdirectories: frontend, backend, mobile, AI, infrastructure, security, marketing, sales, support, legal, design, rnd, team

This created several problems:

Problem	Impact
Project confusion	Developers could not distinguish "Drop source" from "FontelePay source"
Git conflicts	Nested `.git/` directory caused submodule-like behavior without proper configuration
Build interference	FontelePay's `node_modules/` could interfere with Drop's build process
Size bloat	FontelePay's 527 dependency packages inflated the Drop directory

```
graph TB
    subgraph before["Before: Nested Repository"]
        drop_dir["Drop/"]
```

```
        drop_dir --> src["src/"]
        src --> drop_app["drop-app/ (Drop)"]
        src --> fp["fontelepay/ (FontelePay)"]
        fp --> fp_git[".git/ (separate repo)"]
        fp --> fp_nm["node_modules/ (527 pkgs)"]
        fp --> fp_claude[".claude/"]
        fp --> fp_rnd["rnd/ (market research)"]
    end

    subgraph after["After: Clean Separation"]
        products["ALAI/products/"]
        products --> drop_clean["Drop/<br/>Clean codebase"]
        products --> fp_separate["FontelePay/<br/>Preserved independently"]
    end

    before -->|"ADR-002"| after
```

Decision

Move FontelePay out of the Drop directory to its own location:

1. Move `src/fontelepay/` to `~/ALAI/products/FontelePay/` (or archive)
2. Preserve FontelePay's git history by keeping its `.git/` intact during move
3. Keep only a reference document in Drop pointing to FontelePay's new location
4. Copy relevant research documents (particularly `rnd/mobilebank-research/`) to Drop's `rnd/` directory if directly relevant

Consequences

Positive

- Clean Drop directory with only Drop-related source code
- No nested git repository confusion
- Smaller directory footprint for Drop
- Clear project boundaries for all developers and AI agents
- FontelePay research preserved independently for future reference

Negative

- Any scripts or references pointing to `src/fontelepay/` paths break
- Need to verify no Drop code depends on FontelePay modules
- Historical context of FontelePay as Drop's predecessor may be lost if not documented

Risks

- **Reference breakage:** Scripts or docs pointing to old FontelePay path. Mitigation: search-and-replace across codebase during migration.
- **Lost context:** FontelePay's role as Drop's predecessor forgotten. Mitigation: this ADR documents the relationship.

References

- ADR-001: Consolidate Backends -- Companion decision for middleware cleanup
- ADR-003: PSD2 Pass-through Model -- Architectural direction that superseded FontelePay's wallet model
- Original source: `comms/decisions/ADR-002-separate-fontelepay.md`

ADR-003: PSD2 Pass-Through Model

ADR-003: Adopt PSD2 Pass-through Model (No Wallet)

Status: Accepted **Date:** 2026-02-12 **Deciders:** Alem (CEO), John (AI Director) **Category:** Architecture

---

Context

The original Drop codebase implemented a **wallet model** where:

- Users had a local balance stored in the `users` database table
- Users could "top up" their wallet via `/api/users/top-up` (no payment verification)
- Transactions deducted from local balance
- Drop effectively held customer funds

This wallet model had significant regulatory implications under Norwegian law:

Aspect	Wallet Model (EMI)	Pass-through Model (PISP/AISP)
License type	E-money Institution (EMI)	PISP/AISP registration
Norwegian law	Finansforetaksloven	Betalingstjenesteloven
Initial capital	350,000 EUR	20,000-50,000 EUR
Timeline to license	12-18 months	6-12 months
Fund safeguarding	Required (segregated accounts or insurance)	Not needed
PCI-DSS scope	Full (card data stored)	Minimal (no card data)

The alternative PSD2 pass-through model positions Drop as a Payment Initiation Service Provider (PISP) and Account Information Service Provider (AISP) where Drop **never holds customer funds**.

```
graph LR
    subgraph wallet["Wallet Model (Rejected)"]
        user1["User"] -->|"Top-up"| drop_wallet["Drop Wallet<br/>(holds funds)"]
        drop_wallet -->|"Pay"| merchant1["Merchant"]
        drop_wallet -->|"Send"| receiver1["Receiver"]
    end

    subgraph passthrough["Pass-through Model (Adopted)"]
        user2["User"] -->|"PISP: Initiate payment"| bank["User's Bank<br/>(holds funds)"]
        bank -->|"Execute transfer"| merchant2["Merchant"]
        bank -->|"Execute transfer"| receiver2["Receiver"]
        drop_pt["Drop<br/>(orchestrator)"] -.->|"AISP: Read balance"| bank
        drop_pt -.->|"PISP: Initiate"| bank
    end

    classDef rejected fill:#FFCDD2,stroke:#C62828
    classDef adopted fill:#C8E6C9,stroke:#2E7D32

    class user1,drop_wallet,merchant1,receiver1 rejected
    class user2,bank,merchant2,receiver2,drop_pt adopted
```

Decision

Drop adopts the PSD2 pass-through model. Specifically:

1. **No wallet:** Remove all local balance, top-up, and fund-holding functionality
2. **AISP for balance:** User sees their bank account balance via Open Banking API (read-only). The `bank_accounts.balance` field stores a cached AISP read -- not a Drop-held balance
3. **PISP for payments:** Remittance and QR payments are initiated from the user's own bank account via Open Banking payment initiation with SCA
4. **No card storage:** Cards feature gated behind feature flags (all default `false`); future card issuance via PCI-compliant partner only
5. **BankID for SCA:** Strong Customer Authentication via Norwegian BankID replaces email+password for all financial operations

Code Impact

Feature	Wallet Model (removed)	Pass-through Model (current)

Balance	Local `balance` column in `users` table	`bank_accounts.balance` = cached AISP read from bank
Top-up	`/api/users/top-up` endpoint	Removed -- no top-up needed
Remittance	Deduct from local balance	`POST /api/transactions/remittance` triggers PISP
QR Payment	Deduct from local balance	`POST /api/transactions/qr-payment` triggers PISP
Cards	Stored locally (PAN, CVV in DB)	Feature-flagged; future partner integration (token-only)
Auth	Email + password (single factor)	BankID OIDC for SCA
Transaction	Local DB update only	Local record + bank payment confirmation

```
sequenceDiagram
    participant User
    participant Drop
    participant BankID
    participant Bank
    participant Recipient

    Note over User,Recipient: PSD2 Pass-through Remittance Flow
    User->>Drop: Initiate remittance (amount, recipient)
    Drop->>Drop: Fee disclosure (0.5%)
    Drop->>User: Show total cost + exchange rate
    User->>Drop: Confirm payment
    Drop->>BankID: SCA challenge (amount + payee)
    BankID->>User: Authenticate (BankID app)
    User->>BankID: Approve
    BankID->>Drop: SCA confirmed
    Drop->>Bank: PISP: Initiate payment
    Bank->>Bank: Debit user account
    Bank->>Drop: Payment status: processing
    Drop->>Drop: Record transaction (status: processing)
    Bank->>Recipient: Transfer funds (SEPA/SWIFT)
    Bank->>Drop: Payment status: completed
    Drop->>Drop: Update transaction (status: completed)
    Drop->>User: Notification: transfer complete
```

Consequences

Positive

- Lower regulatory barrier to market entry (PISP/AISP vs EMI license)
- Faster licensing timeline (6-12 months vs 12-18 months)
- Lower capital requirements (20-50K EUR vs 350K EUR)
- No PCI-DSS card data storage obligations
- No fund safeguarding requirements (no funds to protect)
- Simpler security model -- Drop cannot lose customer funds
- Users keep their money in their trusted bank until payment execution

Negative

- Dependent on banking partner / BaaS provider for Open Banking API access
- User experience may be slower (bank confirmation for each payment vs instant local deduction)
- Cannot offer instant transfers (limited by bank processing times: 1-2 days SEPA, 2-4 days SWIFT)
- Revenue model changes: no float income from held funds
- BankID integration adds complexity and requires BankID Norge partnership

Risks

- **Banking partner dependency:** If no Norwegian bank provides Open Banking access, Drop cannot function. Mitigation: SpareBank1 already pitched; Swan (BaaS) as backup provider.
- **UX friction:** Each payment requires bank authentication via SCA. Mitigation: BankID app provides smooth mobile flow; consider session-based consent for repeat payments within limits.
- **Corridor coverage:** PISP may not support all 30+ target countries directly. Mitigation: use licensed remittance partner for non-SEPA corridors.

References

- System Context (C4 Level 1) -- Shows Drop's external system relationships
- Open Banking Integration -- AISP/PISP integration specification
- Security Architecture -- Security controls for pass-through model
- Compliance Status -- Regulatory compliance tracking

- Roadmap -- Phase 2 banking integration plan
- Original source: `comms/decisions/ADR-003-psd2-passthrough-model.md`

ADR-004: JWT HTTPOnly Cookies

ADR-004: JWT Storage in httpOnly Cookies

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director) **Category:** Security

---

Context

Drop is a financial application handling payment initiation, bank account data, and personal information. Secure token storage is critical -- token theft enables full account takeover including payment initiation from the victim's bank account.

The two primary options for JWT storage in browser-based SPAs are:

Option	XSS Risk	CSRF Risk	Implementation Complexity
`localStorage`	**HIGH** -- any XSS payload can read tokens	None	Low
`httpOnly cookie`	**None** -- JavaScript cannot access	Medium -- requires CSRF protection	Medium

Given that Drop processes financial data and operates under PSD2, XSS-based token theft would be catastrophic -- an attacker could initiate payments from a user's bank account. CSRF is a more constrained attack vector with well-understood mitigations.

The mobile app (Expo SDK 54) uses Bearer tokens stored in `AsyncStorage` since cookies are not practical for native apps, but the attack surface is fundamentally different (no XSS in native context).

Decision

Store JWTs in httpOnly cookies for the web application. Use Bearer tokens for the mobile API.

Web cookie configuration (`auth.ts:48-54`):

Property	Value	Rationale
`httpOnly`	`true`	Prevents JavaScript access, eliminates XSS token theft
`secure`	`true` (production)	HTTPS-only transport
`sameSite`	`"Lax"`	CSRF defense (allows BankID redirect back)
`maxAge`	604,800 (7d)	Session lifetime
`path`	`"/"`	Full application scope

> **Implementation note:** The actual implementation uses `maxAge=604800` (7d) and `SameSite=Lax` (changed from the originally specified strict/24h to support BankID OIDC redirect flows).

CSRF protection layers:

1. `sameSite: "Lax"` -- browser refuses to send cookie on cross-origin POST requests
2. Origin header validation against allowed origins whitelist (`app.ts:23-30` CORS middleware)
3. CSRF token generation available (`generateCsrfToken()`) for additional protection

```
graph TB
    subgraph localStorage["localStorage (Rejected)"]
        xss["XSS Attack"] -->|"document.cookie<br/>or localStorage.getItem()"| steal["Token
Stolen"]
        steal --> takeover["Account Takeover<br/>+ Payment Initiation"]
    end

    subgraph httpOnly["httpOnly Cookie (Adopted)"]
        xss2["XSS Attack"] -->|"Cannot access<br/>httpOnly cookie"| blocked["BLOCKED"]
        csrf["CSRF Attack"] -->|"Cross-origin request"| samesite["sameSite: strict<br/>BLOCKED
by browser"]
    end

    classDef danger fill:#FFCDD2,stroke:#C62828
    classDef safe fill:#C8E6C9,stroke:#2E7D32
```

```
    class xss,steal,takeover danger
    class blocked,samesite safe
```

Consequences

Positive

- XSS cannot steal authentication tokens (critical for fintech)
- `sameSite: strict` provides strong CSRF protection with minimal implementation overhead
- React's built-in output escaping + CSP headers provide defense-in-depth
- Aligns with OWASP recommendations for secure session management
- Session revocation via `sessions` table allows server-side token invalidation

Negative

- Slightly more complex CSRF handling compared to Bearer tokens
- Cookie-based auth requires different handling for server-side requests (SSR)
- Cannot share tokens across subdomains without `sameSite` adjustment
- Mobile app requires separate Bearer token flow (dual auth pattern)

Risks

- **CSP bypass:** If CSP includes `unsafe-inline` or `unsafe-eval`, XSS risk increases even with httpOnly cookies (attacker could make API calls from victim's browser). Mitigation: tighten CSP with nonce-based script loading for production.

References

- Security Architecture -- Full security controls documentation
- Authentication System -- Auth flow implementation details
- Middleware Documentation -- CSRF and auth middleware
- ADR-007: BankID OIDC Auth -- Authentication provider decision
- OWASP Session Management Cheat Sheet

ADR-005: Monolith First

ADR-005: Monolith-First Architecture

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director), Alem (CEO) **Category:** Architecture

---

Context

Drop needs to go from concept to demo-ready product as quickly as possible. The team is small (1 human CEO + AI agents), and the initial scope is well-defined: 10 screens, ~24 API endpoints, single SQLite database.

Three architecture patterns were considered:

Pattern	Deployment Complexity	Dev Speed	Scaling	Team Size Fit
Microservices	High (orchestration, service mesh, API gateway)	Slow (interface contracts, distributed testing)	Excellent	Large teams
Modular monolith	Low (single deploy unit)	Fast (shared code, simple debugging)	Good (extract later)	Small teams
Serverless functions	Medium (cold starts, state management)	Medium	Good (per-function)	Any

The current scope (7 core pages + API routes) does not justify the operational overhead of microservices. The codebase is ~24 API route files and ~19 database tables -- well within what a single deployment can handle.

```
graph TB
    subgraph monolith["Drop Monolith (Current)"]
        nextjs["Next.js 15<br/>App Router"]
        nextjs --> pages["Frontend Pages<br/>(10 screens, React 19)"]
        nextjs --> api["API Routes<br/>(24 endpoints)"]
```

```
        api --> db["SQLite / PostgreSQL<br/>(19 tables)"]
    end

    subgraph future["Future Extraction (If Needed)"]
        web["Web Frontend<br/>(Next.js)"]
        mobile_api["Mobile API<br/>(Hono v4)"]
        payment_svc["Payment Service<br/>(PISP orchestration)"]
        banking_svc["Banking Service<br/>(AISP integration)"]
        shared_db["PostgreSQL<br/>(shared or per-service)"]
    end

    monolith -->|"Extract when:<br/>- Team grows to 5+ devs<br/>- 10K+ concurrent users<br/>- Independent deploy needed"| future

    classDef current fill:#C8E6C9,stroke:#2E7D32
    classDef future_style fill:#E3F2FD,stroke:#1565C0

    class nextjs,pages,api,db current
    class web,mobile_api,payment_svc,banking_svc,shared_db future_style
```

Decision

Start as a modular monolith. Extract microservices only when scaling demands it.

The monolith is structured with clear module boundaries to make future extraction feasible:

Module	Responsibility	Files
`auth/`	Authentication (BankID OIDC, JWT, sessions)	`api/auth/*`, `lib/auth.ts`
`transactions/`	Remittance and QR payment processing	`api/transactions/*`
`merchants/`	Merchant registration and dashboard	`api/merchants/*`
`cards/`	Card management (feature-flagged)	`api/cards/*`
`compliance/`	GDPR, AML, complaints	`api/consents/*`, `api/complaints/*`, `api/user/*`
`lib/`	Shared: middleware, DB, validation, feature flags	`lib/*`

Extraction triggers (when to consider splitting):

- Team grows beyond 5 developers working on different modules simultaneously
- Single-digit millisecond response time required for specific endpoints
- Independent deployment cadence needed (e.g., payment processing updated hourly, auth monthly)
- Database contention from concurrent writes exceeds SQLite/single-PostgreSQL capacity

Consequences

Positive

- Fastest path from concept to working demo
- Simple deployment: single Docker container on AWS App Runner
- No distributed system complexity (no service discovery, circuit breakers, distributed tracing)
- Easy debugging: single process, single log stream, single database
- Shared code: middleware, validation, and DB access used consistently across all routes
- Low operational cost at current scale

Negative

- All modules must deploy together (no independent deployment)
- Single point of failure (if the monolith crashes, everything is down)
- Scaling is all-or-nothing (cannot scale payment processing independently)
- Module boundaries are convention-based, not enforced by process isolation

Risks

- **Boundary erosion:** Without process isolation, module boundaries may erode over time. Mitigation: clear file organization, code review, and this ADR as a reminder.
- **Scaling ceiling:** Monolith will hit throughput limits at high concurrency. Mitigation: PostgreSQL handles concurrent writes; App Runner auto-scales horizontally.

References

- Architecture Document -- Section 1.2: Architecture Style
- System Context (C4 Level 1) -- High-level system view
- Container Diagram (C4 Level 2) -- Internal container structure
- ADR-012: AWS App Runner -- Deployment target for monolith
- Martin Fowler, "MonolithFirst" (2015)

ADR-006: SQLite to PostgreSQL

ADR-006: SQLite for Development, PostgreSQL for Production

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

Context

Drop requires a database strategy that supports rapid local development while being production-ready for a financial application. The key tension is between developer velocity (zero-config local setup) and production reliability (concurrent writes, ACID transactions, managed backups).

Database	Local Setup	Concurrent Writes	Managed Services	Financial Compliance
SQLite	Zero config, file-based	Poor (single writer)	None	Not suitable for production
PostgreSQL	Docker required	Excellent (MVCC)	AWS RDS, Aurora	Industry standard for fintech
MySQL	Docker required	Good	AWS RDS, Aurora	Common but less feature-rich

Drop's data model includes 19 tables with foreign keys, transactions requiring atomicity (balance deduction + transaction record), and compliance tables needing reliable concurrent access for audit logging. SQLite handles development workloads but cannot support concurrent writes from multiple App Runner instances.

Decision

Use SQLite (`better-sqlite3`) for development and PostgreSQL for production, with a dual-driver abstraction layer.

Driver detection is automatic based on environment:

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

When `DATABASE_URL` is set (production), PostgreSQL is used. Otherwise, SQLite with WAL mode is used.

```
graph LR
    subgraph dev["Development"]
        app_dev["Drop App"] --> dal["Database Access Layer<br/>(db.ts)"]
        dal --> sqlite["SQLite<br/>(better-sqlite3)<br/>./data/drop.db"]
    end

    subgraph prod["Production"]
        app_prod["Drop App (x N)"] --> dal2["Database Access Layer<br/>(db.ts)"]
        dal2 --> pg["PostgreSQL<br/>(AWS RDS)<br/>DATABASE_URL"]
    end

    dal -.->|"Same API:<br/>query(), getOne(),<br/>run(), transaction()"| dal2
```

See ADR-010: Dual Database Driver for the abstraction layer details.

Consequences

Positive

- Zero-config local development: `npm run dev` just works, no Docker needed for DB
- Production-grade concurrent access with PostgreSQL MVCC
- AWS RDS provides automated backups, point-in-time recovery (critical for financial data)
- Same application code runs against both databases via abstraction layer
- SQLite WAL mode provides good read performance during development

Negative

- SQL compatibility layer adds complexity (see ADR-010)
- Subtle behavioral differences between SQLite and PostgreSQL (e.g., type coercion, datetime handling)
- Cannot test PostgreSQL-specific features locally without Docker
- Must test against both databases in CI

Risks

- **SQL dialect drift:** A query that works in SQLite may fail in PostgreSQL. Mitigation: dual-driver abstraction normalizes SQL; CI tests against both.
- **Performance characteristics differ:** SQLite is faster for single-connection workloads. Mitigation: performance testing against PostgreSQL before production launch.

References

- ADR-010: Dual Database Driver -- Abstraction layer implementation
- Database Schema -- Full schema documentation
- Database Design -- Database architecture decisions
- Migration Strategy -- SQLite to PostgreSQL migration plan

ADR-007: BankID OIDC Auth

ADR-007: BankID as Sole Authentication Provider

Status: Accepted **Date:** 2026-02-21 **Deciders:** Alem (CEO), John (AI Director) **Category:** Security

---

Context

Drop is a financial application operating under PSD2 in Norway. PSD2 mandates Strong Customer Authentication (SCA) for payment initiation and account access. SCA requires two of three factors: knowledge (something you know), possession (something you have), and inherence (something you are).

Authentication options considered:

Option	SCA Compliant	KYC Built-in	Norwegian Coverage	Implementation
BankID	Yes (possession + knowledge/biometric)	Yes (national ID verified)	~4.5M users (90%+ adult pop.)	OIDC standard
Vipps Login	Partial (depends on config)	Partial (phone-verified)	~4.3M users	OIDC standard
Email + Password	No (single factor)	No	Universal	Simple
Email + OTP	Partial (possession)	No	Universal	Medium

BankID provides the strongest combination: SCA compliance (BankID app = possession, PIN = knowledge, biometric = inherence), built-in identity verification (national ID / fodselsnummer), and near-universal adoption in Norway. Using BankID as the sole auth provider eliminates the need for a separate KYC step -- identity is verified at login.

The original Drop codebase used email + password authentication, which is inadequate for PSD2 compliance and provides no identity verification.

Decision

Use BankID OIDC as the sole authentication provider for Drop. Remove email/password login.

Authentication architecture:

Platform	Flow	Token Storage	Token Lifetime
Web (Next.js BFF)	BankID OIDC redirect flow	httpOnly cookie (`drop_token`)	7 days
Mobile (Expo)	BankID OIDC with deep link callback	AsyncStorage (Bearer token)	7 days

User creation is automatic on first BankID login:

1. Parse `pid` (fodselsnummer, 11 digits) from BankID ID token
2. Hash `pid` with SHA-256 for storage (`national_id_hash` column)
3. Check for existing user by `national_id_hash`
4. If new: create user with `kyc_status = 'approved'`, `kyc_method = 'bankid'`
5. Verify age >= 18 from `pid` birthdate encoding

```
sequenceDiagram
    participant User
    participant Drop as Drop (BFF)
    participant BankID as BankID OIDC

    User->>Drop: GET /api/auth/bankid
    Drop->>Drop: Generate state + nonce
    Drop->>Drop: Set bankid_state cookie
    Drop->>User: Redirect URL to BankID

    User->>BankID: Authenticate (app/code device)
    Note over User,BankID: SCA: possession (device) +<br/>knowledge (PIN) or inherence
(biometric)
    BankID->>User: Redirect to callback with code

    User->>Drop: GET /api/auth/bankid/callback?code=&state=
    Drop->>Drop: Verify state vs cookie
    Drop->>BankID: Exchange code for tokens
    BankID->>Drop: ID token + access token
    Drop->>Drop: Verify ID token (JWKS)
    Drop->>Drop: Extract pid, verify age >= 18
    Drop->>Drop: Find or create user
    Drop->>Drop: Create session, set JWT cookie
```

```
    Drop->>User: 302 Redirect to /dashboard
```

Deprecated endpoints (return 410 Gone):

- `POST /auth/login` -- replaced by BankID OIDC
- `POST /auth/register` -- automatic via BankID
- `POST /auth/verify-otp` -- not needed

Consequences

Positive

- Full PSD2 SCA compliance out of the box
- Identity verification (KYC) built into authentication -- no separate KYC step for basic verification
- Near-universal adoption in Norway (~4.5M BankID users)
- Eliminates password-related attack vectors (credential stuffing, brute force, phishing)
- National ID hash enables user deduplication across auth providers (Vipps in Phase 2)
- Industry-standard OIDC protocol -- well-documented, well-supported

Negative

- Users without BankID cannot use Drop (excludes some demographics: very young, recent immigrants)
- Dependency on BankID infrastructure availability
- BankID integration requires BankID Norge agreement and certificate
- Development requires mock OIDC flow (`BANKID_MOCK=true`) since real BankID needs production credentials
- More complex auth flow compared to email/password

Risks

- **BankID outage:** If BankID is down, no one can log in. Mitigation: Vipps Login planned as Phase 2 fallback (`auth_provider` field supports multiple providers).
- **Demographic exclusion:** Users without BankID (e.g., new residents) cannot register. Mitigation: Vipps Login + Sumsub manual KYC as alternatives in Phase 2.

References

- Authentication System -- Full auth implementation documentation

- BankID OIDC Integration -- Integration specification
- ADR-004: JWT httpOnly Cookies -- Token storage decision
- ADR-003: PSD2 Pass-through -- SCA requirement origin
- Security Architecture -- Session management details
- BankID Norge OIDC documentation

ADR-008: Hono API Framework

ADR-008: Hono v4 for Mobile API

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director) **Category:** Backend

---

Context

Drop has two client platforms with different API needs:

Platform	Auth Pattern	Token Storage	API Style	Deployment
Web (Next.js)	Cookie-based JWT via BFF	httpOnly cookie	Next.js API Routes (collocated)	Vercel / App Runner
Mobile (Expo)	Bearer token	AsyncStorage	REST API (separate process)	App Runner

Next.js API Routes work well for the web BFF pattern (server-side rendering + API in one deployment), but mobile needs a lightweight, standalone REST API with Bearer token authentication and mobile-specific concerns (deep link callbacks, longer token lifetimes).

Frameworks considered for the mobile API:

Framework	Performance	TypeScript	Edge Compatible	Bundle Size	Ecosystem
Hono v4	Excellent (minimal overhead)	First-class	Yes (Workers, Deno, Bun)	~14KB	Growing fast
Express 5	Good (mature)	Requires @types	No (Node-only)	~200KB	Massive
Fastify 5	Excellent (schema validation)	Good (built-in types)	No (Node-only)	~300KB	Large
Elysia	Excellent (Bun-native)	First-class	Bun only	~20KB	Small

Hono was selected for its TypeScript-first design, minimal overhead, and edge compatibility. The mobile API runs as a separate Hono server on App Runner alongside the Next.js web app.

Decision

Use Hono v4 for the mobile REST API. Keep Next.js API Routes for the web BFF.

```
graph TB
    subgraph clients["Client Platforms"]
        web["Web Browser<br/>(Next.js SSR + CSR)"]
        mobile["Mobile App<br/>(Expo SDK 54)"]
    end

    subgraph backend["Backend Services"]
        nextjs_api["Next.js BFF<br/>API Routes (/api/*)<br/>Cookie auth, SSR"]
        hono_api["Hono v4 API<br/>REST (/v1/*)<br/>Bearer auth, mobile-optimized"]
    end

    subgraph shared["Shared Layer"]
        db["Database Access (db.ts)"]
        bankid["BankID OIDC (bankid.ts)"]
        validation["Validation (validation.ts)"]
    end

    web --> nextjs_api
    mobile --> hono_api
    nextjs_api --> db
    nextjs_api --> bankid
    hono_api --> db
    hono_api --> bankid
    nextjs_api --> validation
    hono_api --> validation
```

Both APIs share the same database access layer, BankID integration, and validation utilities. The difference is in auth pattern and deployment:

Aspect	Next.js API Routes	Hono v4 API
Base path	`/api/`	`/v1/`
Auth	Cookie JWT (httpOnly)	Bearer token (Authorization header)

Aspect	Next.js API Routes	Hono v4 API
Token lifetime	7 days	7 days
BankID callback	HTTP redirect to `/dashboard`	JSON response with token
Rate limiting	SQLite-backed (persistent)	Database-backed (SQLite `rate_limits` table, persistent)
Deployment	Vercel or App Runner	App Runner (standalone)

Consequences

Positive

- Mobile API is lightweight and fast (Hono ~14KB, minimal middleware overhead)
- TypeScript-first with excellent type inference for request/response
- Edge-compatible runtime means future flexibility (Cloudflare Workers, Deno Deploy)
- Clear separation between web BFF (cookie auth) and mobile API (Bearer auth)
- Shared business logic prevents code duplication

Negative

- Two API servers to maintain (Next.js + Hono)
- Two deployment targets on App Runner
- Shared library updates must be tested against both frameworks
- Smaller ecosystem compared to Express (fewer middleware packages)

Risks

- **Diverging behavior:** Same endpoint implemented twice may behave differently. Mitigation: shared database access layer and validation utilities ensure consistent business logic.
- **Hono ecosystem maturity:** Hono is newer than Express/Fastify. Mitigation: Hono v4 is stable and backed by Cloudflare; core routing and middleware are well-tested.

References

- Container Diagram (C4 Level 2) -- Shows both API containers
- Authentication System -- Web vs mobile auth flows
- API Reference -- Next.js API endpoints

- ADR-005: Monolith First -- Overall architecture approach
- Hono v4 documentation: hono.dev

ADR-009: Feature Flag System

ADR-009: Custom Feature Flag System

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director) **Category:** Backend

---

Context

Drop needs feature flags for several reasons:

1. **Gradual rollout:** Cards feature requires a card issuing partner before activation -- must be gated
2. **Kill switches:** Ability to disable features instantly in production if compliance or operational issues arise
3. **Development:** Feature-in-progress code can be merged to main without being user-visible
4. **A/B testing:** Future capability for comparing payment flows

Feature flag approaches considered:

Approach	Cost	Complexity	Server+Client	Targeting	Audit
Custom (env vars)	Free	Low	Yes (NEXT_PUBLIC_)	No	Via deploy history
LaunchDarkly	$10/seat/mo	Medium	Yes	Yes (per-user)	Yes
Unleash (self-hosted)	Free (OSS)	High (infra)	Yes	Yes	Yes
ConfigCat	Free tier	Low	Yes	Yes	Yes

At Drop's current stage (pre-launch, no production users), a custom system based on environment variables provides exactly what is needed: server+client flag availability, zero operational overhead, and type-safe TypeScript integration. User-level targeting is not needed until there are users to target.

Decision

Implement a custom feature flag system using `NEXT_PUBLIC_FF_*` environment variables, with type-safe TypeScript wrappers for server and client access.

Architecture:

```
graph TB
    subgraph env["Environment Variables"]

vars["NEXT_PUBLIC_FF_VIRTUAL_CARDS=false<br/>NEXT_PUBLIC_FF_PHYSICAL_CARDS=false<br/>NEXT_PUBLIC_FF_NOTIFICATIONS=true<br/>..."]
    end

    subgraph server["Server-Side (API Routes)"]
        isEnabled["isEnabled('virtualCards')"]
        featureGate["featureGate('physicalCards')"]
        getAllFlags["getAllFlags()"]
    end

    subgraph client["Client-Side (React)"]
        useFlag["useFeatureFlag('notifications')"]
        useFlags["useFeatureFlags()"]
    end

    vars -->|"Build-time inline<br/>(NEXT_PUBLIC_ prefix)"| server
    vars -->|"Build-time inline<br/>(NEXT_PUBLIC_ prefix)"| client

    featureGate -->|"Returns 404<br/>if disabled"| api_route["API Route<br/>(e.g., POST /api/cards/[id]/physical)"]
```

Flag registry (`feature-flags.ts:27-36`):

Flag	Env Var	Default	Purpose
`virtualCards`	`NEXT_PUBLIC_FF_VIRTUAL_CARDS`	`false`	Virtual card issuance
`physicalCards`	`NEXT_PUBLIC_FF_PHYSICAL_CARDS`	`false`	Physical card ordering
`cardDetails`	`NEXT_PUBLIC_FF_CARD_DETAILS`	`false`	Card detail view
`cardFreeze`	`NEXT_PUBLIC_FF_CARD_FREEZE`	`false`	Card freeze/unfreeze

Flag	Env Var	Default	Purpose
`cardPin`	`NEXT_PUBLIC_FF_CARD_PIN`	`false`	Card PIN management
`spendingLimits`	`NEXT_PUBLIC_FF_SPENDING_LIMITS`	`false`	Spending limit controls
`notifications`	`NEXT_PUBLIC_FF_NOTIFICATIONS`	`true`	Push notifications
`merchantDashboard`	`NEXT_PUBLIC_FF_MERCHANT_DASHBOARD`	`true`	Merchant dashboard

Server-side API route protection via `featureGate()`:

```
const gate = featureGate("physicalCards");
if (gate) return gate;  // Returns 404: "Feature not available"
```

Client-side conditional rendering via `useFeatureFlag()`:

```
const cardsEnabled = useFeatureFlag("virtualCards");
if (!cardsEnabled) return null;
```

Consequences

Positive

- Zero infrastructure cost and operational overhead
- Type-safe TypeScript API prevents flag name typos at compile time
- Works on both server (API routes) and client (React hooks) via `NEXT_PUBLIC_` prefix
- `featureGate()` provides consistent 404 behavior for disabled API endpoints
- Flags are immutable per deployment (changed via environment variable update + redeploy)
- All card-related features safely gated while awaiting card issuing partner

Negative

- No per-user targeting (all users see the same flags)
- Flag changes require redeployment (not runtime-configurable)
- No built-in audit trail of flag changes (relies on deployment history)
- No gradual percentage-based rollout capability
- `NEXT_PUBLIC_` prefix exposes flag names to client (but values are public anyway)

Risks

- **Stale flags:** Flags left enabled/disabled long after they should be changed. Mitigation: feature tracking system (`features.ts`) monitors implementation status; quarterly flag cleanup reviews.
- **Build-time lock-in:** Flags are inlined at build time, so the same build cannot have different flag values. Mitigation: acceptable for current deployment model (one build per environment).

References

- Feature Flags Documentation -- Full API reference and flag listing
- API Reference -- Routes using `featureGate()`
- Security Architecture -- Feature flags section
- ADR-005: Monolith First -- Single deployment model

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

ADR-011: Expo Mobile Framework

ADR-011: Expo SDK 54 for Mobile App

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director), Alem (CEO) **Category:** Mobile

---

Context

Drop requires a mobile app for iOS and Android. The mobile app is the primary interface for remittance and QR payments -- users scan QR codes with their phone camera and approve payments via BankID on the same device.

Mobile framework options considered:

Framework	Cross-Platform	Code Sharing (Web)	OTA Updates	Camera/QR	BankID Integration	Dev Experience
Expo SDK 54	iOS + Android	High (React shared)	Yes (EAS Update)	expo-camera	expo-web-browser	Excellent
React Native (bare)	iOS + Android	High (React shared)	Manual	react-native-camera	Custom deep links	Good
Flutter	iOS + Android	None (Dart vs TS)	No native OTA	camera plugin	Custom deep links	Good
Native (Swift/Kotlin)	Separate codebases	None	App Store only	Native APIs	Native SDKs	Platform-specific

Key factors in the decision:

1. **Code sharing:** Drop's web app uses React 19. Expo enables sharing React components, hooks, types, and business logic between web and mobile.

2. **BankID flow:** Mobile BankID authentication requires opening a secure browser (`expo-web-browser`) and handling deep link callbacks (`drop://auth/callback`). Expo provides both natively.
3. **QR scanning:** Core feature requires camera access. `expo-camera` provides this with barcode scanning built in.
4. **OTA updates:** Financial apps need rapid hotfix deployment. Expo Application Services (EAS) provides over-the-air JavaScript bundle updates without App Store review.
5. **Team capacity:** AI-driven development team benefits from a single language (TypeScript) across all platforms.

Decision

Use Expo SDK 54 with managed workflow for the Drop mobile app.

```
graph TB
    subgraph mobile["Mobile App (Expo SDK 54)"]
        screens["Screens<br/>(10 screens matching web)"]
        hooks["Shared Hooks<br/>(useAuth, useTransactions)"]
        types["Shared TypeScript Types"]

        screens --> camera["expo-camera<br/>(QR scanning)"]
        screens --> browser["expo-web-browser<br/>(BankID auth)"]
        screens --> notif["expo-notifications<br/>(push alerts)"]
        screens --> storage["AsyncStorage<br/>(Bearer token)"]
        screens --> linking["expo-linking<br/>(deep links: drop://)"]
    end

    subgraph web["Web App (Next.js 15)"]
        web_screens["Screens<br/>(10 screens)"]
        web_hooks["Shared Hooks"]
        web_types["Shared TypeScript Types"]
    end

    subgraph backend["Backend"]
        hono["Hono v4 API<br/>(/v1/* - Bearer auth)"]
        nextjs["Next.js BFF<br/>(/api/* - Cookie auth)"]
    end

    hooks -.->|"Shared React code"| web_hooks
    types -.->|"Shared types"| web_types
```

```
    mobile --> hono
    web --> nextjs

    classDef expo fill:#E3F2FD,stroke:#1565C0
    classDef web_style fill:#C8E6C9,stroke:#2E7D32
    classDef backend_style fill:#FFF3E0,stroke:#E65100

    class screens,hooks,types,camera,browser,notif,storage,linking expo
    class web_screens,web_hooks,web_types web_style
    class hono,nextjs backend_style
```

Key Expo Modules Used

Module	Purpose	Drop Feature
`expo-camera`	Camera access + barcode scanning	QR payment scanning
`expo-web-browser`	Secure in-app browser	BankID OIDC authentication
`expo-notifications`	Push notification handling	Transaction alerts, payment receipts
`expo-linking`	Deep link handling (`drop://`)	BankID callback, notification deep links
`@react-native-async-storage`	Persistent key-value store	Bearer token storage
`expo-secure-store`	Encrypted storage	Sensitive data (future biometric)
`expo-local-authentication`	Biometric auth	App unlock (Phase 2)

Mobile-Specific Auth Flow

The mobile BankID flow differs from web:

1. `GET /v1/auth/bankid/initiate?platform=mobile` returns `{ redirectUrl, state }`
2. Open BankID in `expo-web-browser` (secure, isolated browser)
3. BankID redirects to `drop://auth/callback?code=&state=`
4. `expo-linking` catches the deep link
5. `POST /v1/auth/bankid/callback` exchanges code for Bearer token
6. Token stored in `AsyncStorage` (7-day lifetime)

Consequences

Positive

- Single language (TypeScript) across web, mobile, and backend
- High code reuse: shared types, hooks, and validation logic with web app
- OTA updates via EAS enable rapid hotfixes without App Store review cycle
- Managed workflow eliminates native build complexity
- `expo-camera` provides built-in barcode scanning for QR payments
- `expo-web-browser` provides secure BankID integration
- Large React Native ecosystem for additional modules

Negative

- Expo managed workflow limits access to some native APIs (can eject if needed)
- App size larger than pure native (~25MB vs ~5MB)
- JavaScript bridge performance for heavy computation (not a concern for Drop's use case)
- Must use Expo-compatible packages (some React Native packages require ejection)
- EAS build service adds to CI costs

Risks

- **Expo SDK upgrade breakage:** Major Expo SDK upgrades can break packages. Mitigation: managed workflow handles most upgrades; test thoroughly before upgrading.
- **App Store rejection:** Financial apps face stricter App Store review. Mitigation: ensure compliance with App Store Review Guidelines section 3.1 (payments) and 5.1 (privacy).
- **Performance on low-end devices:** React Native may lag on older Android devices. Mitigation: minimal animations, lazy loading, optimized list rendering.

References

- ADR-008: Hono API Framework -- Mobile API backend
- Authentication System -- Mobile BankID flow
- Component Overview (C4 Level 3) -- Mobile app components
- ADR-007: BankID OIDC Auth -- Authentication provider
- Expo documentation: docs.expo.dev

ADR-012: AWS App Runner Deploy

ADR-012: AWS App Runner for Deployment

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director), Alem (CEO) **Category:** Infrastructure

---

Context

Drop needs a deployment target for its backend services (Next.js BFF and Hono mobile API). The deployment platform must support Docker containers, auto-scaling, HTTPS termination, and be cost-effective at low initial traffic with the ability to scale.

Deployment options considered:

Platform	Container Support	Auto-scaling	Min Cost	Operational Overhead	Cold Start
AWS App Runner	Yes (ECR/source)	Automatic	~$5/mo (min instances)	Very low	Warm (min instance)
AWS ECS/Fargate	Yes (ECR)	Manual config (target tracking)	~$10/mo (Fargate)	Medium (task defs, services, ALB)	Warm
AWS Lambda	Yes (container image)	Automatic (per-request)	~$0 (free tier)	Low	Cold start problem
Vercel	No (serverless functions)	Automatic	Free tier	Very low	Cold start for API
Railway	Yes (Dockerfile)	Automatic	~$5/mo	Very low	Warm
Fly.io	Yes (Dockerfile)	Automatic	~$5/mo	Low	Warm

Key factors:

1. **WebSocket/long connections:** App Runner supports them; Lambda does not (29s timeout)
2. **PostgreSQL connectivity:** App Runner runs in VPC, can connect to RDS; Lambda requires NAT gateway ($32/mo)
3. **Operational simplicity:** App Runner is "push container, get HTTPS endpoint" -- no load balancer, target group, or service mesh to configure
4. **Cost at scale:** App Runner pricing is straightforward (vCPU-hour + memory-hour); ECS/Fargate pricing is similar but with more configuration
5. **AWS ecosystem:** PostgreSQL on RDS, secrets in Secrets Manager, logs in CloudWatch -- all in same account

Vercel was used for the landing page (static) and is excellent for Next.js, but its serverless function model is not ideal for the Hono API or long-running database connections.

Decision

Use AWS App Runner for backend deployment. Keep Vercel for the landing page (static site).

```
graph TB
    subgraph edge["Edge Layer"]
        cf["Cloudflare<br/>DNS + CDN + WAF + DDoS"]
    end

    subgraph aws["AWS (eu-north-1)"]
        subgraph apprunner["App Runner"]
            nextjs["Next.js BFF<br/>Web app + API routes<br/>(1-10 instances)"]
            hono["Hono API<br/>Mobile REST API<br/>(1-10 instances)"]
        end

        subgraph data["Data Layer"]
            rds["RDS PostgreSQL<br/>(production DB)"]
            secrets["Secrets Manager<br/>(JWT_SECRET, BANKID creds)"]
        end

        subgraph monitoring["Monitoring"]
            cw["CloudWatch<br/>(logs, metrics)"]
        end
    end
```

```
    subgraph vercel["Vercel"]
        landing["Landing Page<br/>getdrop.no<br/>(static)"]
    end

    cf --> nextjs
    cf --> hono
    cf --> landing
    nextjs --> rds
    hono --> rds
    nextjs --> secrets
    hono --> secrets
    nextjs --> cw
    hono --> cw

    classDef edge_style fill:#FFF3E0,stroke:#E65100
    classDef aws_style fill:#E3F2FD,stroke:#1565C0
    classDef vercel_style fill:#F3E5F5,stroke:#6A1B9A

    class cf edge_style
    class nextjs,hono,rds,secrets,cw aws_style
    class landing vercel_style
```

App Runner Configuration

Setting	Value	Rationale
Region	`eu-north-1` (Stockholm)	Closest AWS region to Norway; GDPR data residency
Source	ECR (Docker image)	Pushed by GitHub Actions CI/CD
CPU	1 vCPU	Sufficient for current load
Memory	2 GB	Room for Node.js heap + DB connections
Min instances	1	Eliminates cold start; ~$5/mo baseline
Max instances	10	Auto-scales based on concurrent requests
Port	3000 (Next.js), 3001 (Hono)	Default Node.js ports
Health check	`GET /api/health` (also available at `/v1/health`)	Returns DB connectivity status

Setting	Value	Rationale
Auto deploy	Yes (on ECR push)	CI/CD pushes new image, App Runner deploys

Deployment Pipeline

```
graph LR
    push["git push"] --> gha["GitHub Actions"]
    gha --> build["Docker build<br/>+ TypeScript check<br/>+ Lint + Test"]
    build --> ecr["Push to ECR"]
    ecr --> apprunner["App Runner<br/>auto-deploy"]
    apprunner --> health["Health check<br/>GET /api/health"]
    health -->|"Healthy"| live["Live traffic"]
    health -->|"Unhealthy"| rollback["Auto-rollback<br/>to previous version"]
```

Consequences

Positive

- Minimal operational overhead: no load balancers, target groups, or service meshes to manage
- Automatic HTTPS with AWS-managed TLS certificates
- Auto-scaling based on concurrent requests (0 config beyond min/max instances)
- VPC connectivity to RDS PostgreSQL without NAT gateway
- Automatic rollback on failed health checks
- CloudWatch integration for logs and metrics out of the box
- Cost-effective: ~$5/mo baseline with 1 min instance, scales linearly

Negative

- Less configurability than ECS/Fargate (no custom networking, task placement, or sidecar containers)
- Limited to HTTP/HTTPS workloads (no TCP/UDP services)
- Newer AWS service with fewer community resources and examples
- No built-in blue/green deployment (App Runner does rolling updates). **Note:** `deployment-architecture.md` describes blue/green as aspirational — it would need custom implementation.
- Vendor lock-in to AWS (container is portable, but App Runner config is not)

Risks

- **App Runner regional availability:** Service may not be available in all regions. Mitigation: `eu-north-1` (Stockholm) is supported.
- **Scaling latency:** New instances take 30-60 seconds to provision. Mitigation: maintain 1 min instance for baseline traffic; pre-scale before expected traffic events.
- **Cost at scale:** App Runner pricing can exceed ECS/Fargate for high-throughput workloads. Mitigation: evaluate migration to ECS/Fargate if monthly compute exceeds $200.

References

- Deployment Architecture -- Full deployment topology
- System Context (C4 Level 1) -- Infrastructure components
- ADR-005: Monolith First -- Single deployment model
- ADR-008: Hono API Framework -- Mobile API deployment
- AWS App Runner documentation

ADR Overview

Architecture Decision Records (ADRs)

Project: Drop -- Fintech Payment App **Last updated:** 2026-03-03 **Maintainer:** Standards Architect

---

What are ADRs?

Architecture Decision Records capture significant technical decisions made during Drop's development. Each ADR documents the context, the decision itself, and its consequences -- providing a historical record of *why* the system is built the way it is.

ADRs are immutable once accepted. If a decision is reversed, the original ADR is marked **Superseded** and a new ADR is created referencing it.

ADR Index

ADR	Title	Status	Date	Category
ADR-001	Consolidate to Single Backend	Accepted	2026-02-12	Architecture
ADR-002	Separate FontelePay from Drop Repository	Accepted	2026-02-12	Architecture
ADR-003	Adopt PSD2 Pass-through Model (No Wallet)	Accepted	2026-02-12	Architecture
ADR-004	JWT Storage in httpOnly Cookies	Accepted	2026-02-21	Security
ADR-005	Monolith-First Architecture	Accepted	2026-02-21	Architecture

ADR	Title	Status	Date	Category
ADR-006	SQLite for Dev, PostgreSQL for Production	Superseded by ADR-014	2026-02-21	Database
ADR-007	BankID as Sole Authentication Provider	Accepted	2026-02-21	Security
ADR-008	Hono v4 for Mobile API	Accepted	2026-02-21	Backend
ADR-009	Custom Feature Flag System	Accepted	2026-02-21	Backend
ADR-010	Dual Database Driver Abstraction	Superseded by ADR-014	2026-02-21	Database
ADR-011	Expo SDK 54 for Mobile App	Accepted	2026-02-21	Mobile
ADR-012	AWS App Runner for Deployment	Accepted	2026-02-21	Infrastructure
ADR-014	PostgreSQL-Only Architecture (Drizzle ORM)	Accepted	2026-02-26	Database

ADR Lifecycle

```
stateDiagram-v2
    [*] --> Proposed : Author drafts ADR
    Proposed --> Accepted : Team reviews and approves
    Proposed --> Rejected : Team rejects proposal
    Accepted --> Deprecated : No longer relevant
    Accepted --> Superseded : New ADR replaces this one
    Rejected --> [*]
    Deprecated --> [*]
    Superseded --> [*]
```

ADR Template

Use this template when proposing a new ADR. Save as `ADR-NNN-short-title.md` in this directory.

```
# ADR-NNN: Title

**Status:** Proposed | Accepted | Deprecated | Superseded by [ADR-XXX](ADR-XXX-title.md)
**Date:** YYYY-MM-DD
**Deciders:** [Names and roles]
**Category:** Architecture | Security | Database | Backend | Frontend | Mobile |
Infrastructure

---

## Context

What is the issue that we are seeing that motivates this decision?
Include technical background, constraints, and forces at play.

## Decision

What is the change that we are proposing and/or doing?
State the decision clearly and concisely.

## Consequences

### Positive
- List benefits of this decision

### Negative
- List drawbacks and trade-offs

### Risks
- List risks and their mitigations

## References

- Link to related ADRs, documents, or external resources
- Link to implementation PRs or tasks
```

Guidelines for Proposing ADRs

1. **When to write an ADR:** Any decision that affects the system architecture, technology choices, security model, or data model and would be hard to reverse.
2. **Scope:** One ADR per decision. If a decision has multiple parts, consider splitting into separate ADRs.
3. **Numbering:** Use sequential three-digit numbers (001, 002, ...). Never reuse numbers.
4. **Review process:** Draft as `Proposed`, share with the team. Once approved by the AI Director (John) and/or CEO (Alem), change status to `Accepted`.
5. **Superseding:** When reversing a decision, create a new ADR and update the old one's status to `Superseded by [ADR-XXX]`. Never delete old ADRs.
6. **Context matters:** Future readers need to understand *why* the decision was made. Include constraints, alternatives considered, and the reasoning.

Cross-References

- Architecture Document -- Main architecture overview
- System Context (C4 Level 1) -- System context diagram
- Compliance Status -- Regulatory compliance tracking