ADRs

QODY Architecture Decision Records (ADRs)

Architecture Decision Records document key architectural choices made for QODY. Each ADR captures the context, decision, and consequences of significant technical decisions.

ADR-001: RLS/BYPASSRLS Fail-Closed Guard

Status: ACCEPTED | **Date:** 2026-06-22 | **Author:** Petter Graff (CodeCraft)

Context

The Bilko product suffered a silent cross-tenant data breach where the application DB role (`bilko_admin`) had the `BYPASSRLS` attribute, which silently overrides `FORCE ROW LEVEL SECURITY`. RLS policies looked configured but isolated nothing. This was discovered late and required extensive remediation.

Decision

QODY will implement a **fail-closed RLS role verification** that runs at application startup, before any HTTP server initialization:

1. The app connects as a dedicated role (`qody_app`) that **MUST NOT** have `BYPASSRLS` and **MUST NOT** be the table owner
2. Migrations/owner DDL run as a separate privileged role (`qody_flyway`) used only by Flyway, never by the running app
3. CI startup-validation query (fail-closed) on every boot:

```
SELECT rolname, rolbypassrls FROM pg_roles WHERE rolname = 'qody_app';
-- must return rolbypassrls = false, or the app refuses to start
```

4. RLS isolation E2E test (Proveo): create two venues, set context to venue A, assert venue B's orders are invisible AND uninsertable

The `/health` endpoint also exposes RLS role status and returns HTTP 500 if BYPASSRLS is active.

Consequences

Positive:

- Silent cross-tenant data breach is impossible — the app refuses to start if misconfigured
- RLS role status is observable at runtime via `/health`
- Proveo validation provides continuous regression protection

Negative:

- Slightly more complex DB role setup (two roles instead of one)
- Startup self-check adds ~100ms to boot time (acceptable)

Validation: Phase 0 Proveo validation PASS (Test 6 — Bilko breach reproduced and guarded against).

ADR-002: Payment Provider Strategy — Provider Abstraction Layer

Status: ACCEPTED | **Date:** 2026-06-22 | **Author:** Markos Zachariadis (Finverge)

Context

QODY targets multiple markets with different payment ecosystems:

- **BiH/Balkans:** Stripe (international cards), Monri (local PSP with BAM settlement)
- **Norway:** Vipps MobilePay (90% adoption), Stripe (international cards)

Locking into a single provider creates risk (downtime, pricing changes, market-specific requirements).

Decision

Implement a **Payment Gateway Abstraction** pattern with a provider-agnostic interface:

```
interface PaymentGateway {
    suspend fun createPaymentIntent(request: PaymentIntentRequest): PaymentIntentResponse
    suspend fun confirmPayment(intentId: String): PaymentConfirmationResponse
    suspend fun refund(paymentId: String, amount: Money): RefundResponse
    suspend fun handleWebhook(payload: String, signature: String): WebhookEvent
}

// Implementations:
class StripeGateway : PaymentGateway { /* Stripe-specific logic */ }
class VippsGateway : PaymentGateway { /* Vipps-specific logic */ }
class MonriGateway : PaymentGateway { /* Monri-specific logic */ }

// Factory for per-venue routing:
class PaymentGatewayFactory(private val config: PaymentConfig) {
    fun forVenue(venueId: UUID): PaymentGateway {
        return when (config.getProviderForVenue(venueId)) {
            PaymentProvider.STRIPE -> StripeGateway(config.stripe)
            PaymentProvider.VIPPS -> VippsGateway(config.vipps)
            PaymentProvider.MONRI -> MonriGateway(config.monri)
        }
    }
}
```

The database stores `venues.payment_provider_id` to allow per-venue provider selection.

Consequences

Positive:

- No vendor lock-in — can switch providers or support multiple simultaneously
- Market-specific providers (Vipps for Norway, Monri for BiH) can coexist
- A/B testing of providers is possible
- Future-proof for new providers (e.g., if BiH launches instant payments)

Negative:

- Abstraction adds complexity (must support lowest-common-denominator API)
- Provider-specific features (e.g., Stripe Radar fraud detection) require careful interface design

Alternatives Considered:

- **Stripe-only:** Rejected — insufficient for Norway (Vipps required) and BiH (Monri preferred for local recognition)
- **Third-party payment orchestration (e.g., Primer.io):** Rejected — adds dependency + cost + not proven in BiH/Balkans

---

ADR-003: Outbox vs Kafka — Start with Transactional Outbox, Upgrade Path to Kafka

Status: ACCEPTED | **Date:** 2026-06-22 | **Author:** Petter Graff (CodeCraft)

Context

QODY needs to propagate order state transitions (e.g., `SUBMITTED` → `ACCEPTED`) to:

- Real-time hub (WebSocket/SSE) for kitchen display and guest table updates
- Potentially other services in the future (e.g., analytics, notifications)

Two architectural patterns exist:

1. **Transactional outbox:** Write event to Postgres `outbox` table in the same transaction as the state change; a dispatcher drains the outbox
2. **Kafka:** Publish event directly to Kafka topic; consumers subscribe

Decision

Start with a **Postgres transactional outbox** for Phase 1/2. Order state transitions write the state change AND the outbox row in the same DB transaction (no lost events, no dual-write inconsistency). A dispatcher drains the outbox to the real-time hub.

When a venue chain needs cross-service scale (Phase 3), the outbox drains to Kafka instead — same producer contract, zero domain-code rewrite.

Rationale

- **No premature distribution:** QODY starts as a monolith. Kafka is distributed-systems tax we don't need yet
- **Transactional guarantees:** Outbox pattern ensures exactly-once semantics without dual-write complexity

- **Mechanical sympathy:** Earn Kafka, do not cargo-cult it. Distribute only proven seams
- **Upgrade path is clean:** When outbox drains to Kafka instead of in-memory hub, producer code is unchanged

Consequences

Positive:

- Simple: Postgres transactions we already understand
- No Kafka infra cost/complexity in MVP
- Exactly-once delivery guaranteed by DB transaction
- Clear upgrade path when scale justifies Kafka

Negative:

- Outbox dispatcher must poll the `outbox` table (adds DB load)
- Not suitable for cross-service pub/sub until Kafka is added

Alternatives Considered:

- **Kafka from day one:** Rejected — premature optimization, adds infra complexity for MVP
- **Direct in-memory event bus:** Rejected — no durability, lost events on crash

---

ADR-004: Pay-Now vs Pay-at-End — Both, Flag-Gated

Status: ACCEPTED | **Date:** 2026-06-22 | **Author:** Markos Zachariadis (Finverge)

Context

Hospitality venues have different payment timing preferences:

- **Fast casual / bar:** Pay immediately after ordering (pay-per-order)
- **Table service:** Multiple rounds of ordering, pay once at the end (open tab / pay-at-end)

Different markets and venue types require different flows.

Decision

Support **both payment timing models**, flag-gated per venue:

1. **Pay-per-order** (Phase 1 MVP): Guest submits order → immediate payment → order goes to kitchen only after payment succeeds
2. **Pay-at-end** (Phase 2): Guest orders multiple times → orders accumulate on the `table_session` → one settlement at the end when guest requests bill

The order lifecycle state machine supports both — the only difference is *when* the `PAID` transition fires and whether it targets `order` or `table_session`.

Flag: `qody.payment.pay_at_end` (venue-level, Unleash).

Consequences

Positive:

- Flexible: supports both fast-casual and table-service venues
- Market-specific: BiH bars prefer pay-now; Norwegian cafes prefer pay-at-end
- Same backend state machine handles both flows

Negative:

- Slightly more complex payment logic (two paths)
- Reconciliation must handle both `order.total_paid` and `table_session.total_paid`

Alternatives Considered:

- **Pay-now only:** Rejected — does not support table-service venues (major market segment)
- **Pay-at-end only:** Rejected — fast-casual venues need immediate payment to avoid fraud risk

---

Future ADRs (To Be Written)

- **ADR-005:** Fiscalization Strategy — Non-Fiscal MVP vs ESET Integration
- **ADR-006:** AI Tier-Router Architecture — Ollama-First Cost Control
- **ADR-007:** Multi-Language Translation — Pre-Computed Cache vs On-Demand
- **ADR-008:** Real-Time Hub — WebSocket vs SSE Fallback Strategy
- **ADR-009:** Feature Flag Enforcement — API-Level vs UI-Only
- **ADR-010:** Deploy Verification — ACA 0%-Traffic Trap Mitigation

---

Revision #1
Created 2026-06-22 15:52:06 UTC by John
Updated 2026-06-22 15:52:07 UTC by John