Architecture

QODY Architecture

Author: Petter Graff (CodeCraft / ALAI Architecture) | **Date:** 2026-06-22

System Context

Three independently deployable micro-frontends (MFE) talk to one Ktor API. The API owns Postgres, emits domain events to an internal bus, fans real-time updates out over WebSocket/SSE, reads feature flags from Unleash, and talks to a payment provider via webhooks.

Component Diagram

```
graph TB
  subgraph Clients
    G["Guest MFE<br/>(QR menu, cart, pay)<br/>public, no-login"]
    S["Staff/Kitchen MFE<br/>(KDS, order board)<br/>JWT staff"]
    A["Admin MFE<br/>(venue dashboard,<br/>menu editor, plans)<br/>JWT admin"]
  end

  subgraph Edge
    CDN["CDN / static host<br/>per-MFE bundles"]
    GW["Reverse proxy / API gateway<br/>(TLS, CORS, rate-limit,<br/>public /guest carve-out)"]
  end

  subgraph Backend["Ktor API (Kotlin)"]
    R["Route groups:<br/>/guest /staff /admin /webhooks /health"]
    SVC["Domain services<br/>(Order, Menu, Session,<br/>Payment, Tenant)"]
    EVT["Event bus<br/>(in-proc -> Postgres outbox<br/>-> upgradeable to Kafka)"]
    RT["Real-time hub<br/>(WebSocket + SSE fallback)"]
    FF["Unleash client<br/>(per-venue/per-plan flags)"]
  end
```

```
DB[("PostgreSQL 16<br/>RLS tenant isolation<br/>Flyway migrations")]
PAY["Payment provider(s)<br/>Stripe / market-specific"]
UNL["Unleash server"]
OBS["Sentry + structured logs<br/>+ /health"]

G --> CDN
S --> CDN
A --> CDN
G --> GW
S --> GW
A --> GW
GW --> R
R --> SVC
SVC --> DB
SVC --> EVT
EVT --> RT
EVT --> DB
RT -. "live order/table updates" .-> S
RT -. "table status" .-> G
SVC --> FF
FF --> UNL
SVC --> PAY
PAY -- "webhook (signed)" --> R
SVC --> OBS
```

Why These Boundaries

- **One API, three MFEs.** The MFE split is about deploy cadence and blast radius, not about microservices. Guest menu changes ship hourly; the admin dashboard ships weekly. A bug in the menu editor must never take down table ordering.
- **Event bus starts in-process with a Postgres transactional outbox.** 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, the outbox drains to Kafka instead.
- **Real-time hub = WebSocket with SSE fallback.** Kitchen display systems (KDS) sit on venue Wi-Fi that is hostile (NAT, captive portals, flaky AP roaming). Design for failure: heartbeat + auto-reconnect + on-reconnect state resync.

Multi-Tenancy Model

Tenant = Venue. A Tenant/Organization may own multiple Venues for chains; the RLS scope key is `venue_id`, with an optional `org_id` parent for chain-level admin.

Per ALAI database rules DB-05/DB-06: every tenant-scoped table carries `venue_id UUID NOT NULL` and RLS is **ENABLED + FORCED**.

```
ALTER TABLE orders ENABLE ROW LEVEL SECURITY;
ALTER TABLE orders FORCE  ROW LEVEL SECURITY;

CREATE POLICY tenant_isolation ON orders
  USING (venue_id = current_setting('app.current_venue_id', true)::uuid);

CREATE POLICY tenant_insert ON orders
  AS RESTRICTIVE FOR INSERT
  WITH CHECK (venue_id = current_setting('app.current_venue_id', true)::uuid);
```

The Ktor layer sets `SET app.current_venue_id = '<uuid>'` at connection checkout (HikariCP) inside the request/transaction scope, and **resets it on release**. Stale tenant context on a pooled connection is a silent cross-venue data breach.

Bilko RLS Lesson — Hard Requirement (Tool-Verified 2026-06-19)

The most expensive Bilko bug was NOT a missing policy. It was that the application DB role had the `BYPASSRLS` attribute, which **silently overrides FORCE ROW LEVEL SECURITY** — RLS looked configured but isolated nothing. Mandatory for QODY:

1. The app connects as a dedicated role (e.g. `qody_app`) that **MUST NOT** have `BYPASSRLS` and **MUST NOT** be the table owner.
2. Migrations/owner DDL run as a separate privileged role 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.

Guest Path Special-Casing

The guest MFE is anonymous (no JWT). The guest still must be scoped to one venue+table. Scoping comes from the signed QR token, not from a login. The API resolves the QR token to `venue_id`/`table_id` server-side, sets RLS context from that, and the guest can only ever touch their own table's open session. Guest endpoints are explicitly carved out of auth at the gateway (a tight `/guest/*` allowlist).

Core Domain Model

UUID PKs, `NUMERIC(19,4)` money, `TIMESTAMPTZ`, `deleted_at` soft delete, `version` optimistic lock on mutable entities, `venue_id` + RLS on all tenant tables.

```
erDiagram
  ORGANIZATION ||--o{ VENUE : owns
  VENUE ||--o{ TABLE : has
  VENUE ||--o{ MENU : publishes
  VENUE ||--o{ STAFF : employs
  MENU ||--o{ CATEGORY : contains
  CATEGORY ||--o{ MENU_ITEM : lists
  MENU_ITEM ||--o{ MODIFIER_GROUP : has
  MODIFIER_GROUP ||--o{ MODIFIER : offers
  TABLE ||--o{ TABLE_SESSION : hosts
  TABLE_SESSION ||--o{ ORDER : groups
  ORDER ||--o{ ORDER_LINE : contains
  ORDER_LINE ||--o{ ORDER_LINE_MODIFIER : applies
  ORDER ||--o{ PAYMENT : settled_by
  STAFF }o--|| ROLE : assigned
```

Key Entities

Entity	Purpose	Key Fields
`organization`	Chain owner (optional parent)	id, name, plan_tier
`venue`	The tenant boundary	id, org_id, name, slug, branding(jsonb), timezone, currency, plan_tier
`restaurant_table`	Physical table	id, venue_id, label, qr_token_id, capacity
`menu`	Versioned menu for a venue	id, venue_id, name, is_active, valid_from/until

Entity	Purpose	Key Fields
`menu_item`	Sellable item	id, category_id, venue_id, name, description, price NUMERIC(19,4), tax_rate, allergens(jsonb)
`table_session`	One sitting at a table	id, venue_id, table_id, status, opened_at, closed_at
`order`	A submission within a session	id, venue_id, table_session_id, status, subtotal, tax_total, tip_amount, total, version
`order_line`	Line in an order	id, order_id, venue_id, menu_item_id, qty, unit_price, line_total, note, status
`payment`	Settlement attempt/record	id, venue_id, order_id, provider, provider_ref, amount, currency, status, idempotency_key

Money/price snapshotting. `order_line.unit_price` and `order_line_modifier.price_delta_snapshot` are *copied at order time*. The menu price can change tomorrow; what the guest agreed to pay is frozen on the line.

Branding lives in `venue.branding` (jsonb: logo, colours, accent) so white-labeling is a data concern, not a build concern.

Order Lifecycle

States are explicit and enforced server-side (a state machine). Illegal transitions are rejected, not silently ignored. Every transition writes a row to the transactional outbox → real-time hub.

```
stateDiagram-v2
  [*] --> SESSION_OPEN: QR scan resolves token -> open/attach TableSession
  SESSION_OPEN --> CART: guest adds items (client-side draft, server-validated)
  CART --> SUBMITTED: guest submits order (server validates availability + price + flags)
  SUBMITTED --> ACCEPTED: staff/kitchen accepts (or auto-accept flag)
  ACCEPTED --> IN_PREP: kitchen starts
  IN_PREP --> READY: kitchen marks ready
  READY --> SERVED: waiter serves
  SERVED --> PAID: payment captured (pay-now or pay-at-end)
  PAID --> CLOSED: session settled, table freed
  SUBMITTED --> CANCELLED: staff/guest cancels pre-accept
  ACCEPTED --> CANCELLED: staff cancels (with reason)
  CLOSED --> [*]
```

Real-Time Propagation

- `SUBMITTED` event → appears instantly on Kitchen MFE order board (the demo "wow" moment)
- `IN_PREP` / `READY` → guest sees their order status on the table; waiter sees "ready for pickup"
- `SERVED` / `PAID` / `CLOSED` → table status flips to free on the Staff MFE floor view

Payment Timing

Payment timing is a venue setting (flag-gated):

- **Pay-per-order** (fast casual / bar): each order pays immediately; SUBMITTED → PAID may precede kitchen
- **Pay-at-end** (table service): orders accumulate on the table_session; one settlement at the end

Idempotency. Payment captures and webhook handlers use `payment.idempotency_key`. A retried Stripe webhook must never double-charge or double-advance state.

Reconnect resync. On KDS reconnect the client calls `GET /staff/orders?status=open` and rebuilds its board from authoritative state.

API Surface (Ktor Route Groups)

```
/health                     GET    liveness/readiness (MUST), RLS-role self-check

# ---- GUEST (public, scoped by signed QR token, no JWT) ----
/guest/resolve              POST   { qrToken } -> { venueId, tableId, sessionId, branding }
/guest/menu                 GET    active menu for resolved venue
/guest/session/{id}         GET    current session + my orders + live status
/guest/cart/validate        POST   server-side price/availability/flag re-check
/guest/order                POST   submit order (idempotency key) -> SUBMITTED
/guest/payment/intent       POST   create payment intent
/guest/payment/confirm      POST   confirm/capture
/guest/stream               GET    SSE: my order/table status updates

# ---- STAFF / KITCHEN (JWT staff, role-gated) ----
/staff/auth/login           POST   email+password -> JWT
/staff/orders               GET    open orders board
/staff/orders/{id}/accept   POST   SUBMITTED -> ACCEPTED
```

```
/staff/orders/{id}/prep      POST   ACCEPTED -> IN_PREP
/staff/orders/{id}/ready     POST   IN_PREP -> READY
/staff/orders/{id}/serve     POST   READY -> SERVED
/staff/sessions/{id}/close   POST   settle + free table -> CLOSED
/staff/stream                WS     live order events (KDS)

# ---- ADMIN / VENUE DASHBOARD (JWT admin/owner) ----
/admin/venues                CRUD   venue + branding
/admin/tables                CRUD   tables + QR token (re)generation
/admin/menus                 CRUD   menu/category/item/modifier
/admin/staff                 CRUD   staff + roles
/admin/reports               GET    sales/orders summaries

# ---- WEBHOOKS (signature-verified) ----
/webhooks/payment/{provider} POST   signed payment events
```

Feature-Flag Map (Unleash)

Same pattern as Bilko feature-enable (MC #102481): the **plan tier** drives a set of Unleash flags; flags are evaluated with a venue context so a flag can also be force-toggled for a single venue (pilot, demo, A/B).

Capability	Flag key	Basic	Pro	Enterprise
QR menu + order + pay (core)	always-on	✓	✓	✓
Kitchen display (KDS real-time)	`kds.realtime`	✓	✓	✓
Multi-language menu	`menu.multilang`	–	✓	✓
Tipping at checkout	`pay.tipping`	–	✓	✓
Split bill	`pay.splitbill`	–	✓	✓
Pay-at-end (table tab)	`pay.payatend`	–	✓	✓
AI upsell / recommendations	`ai.upsell`	–	–	✓
White-label theming	`brand.whitelabel`	–	✓	✓
Chain dashboard	`chain.dashboard`	–	–	✓

Backend gates the *capability* so a flag is a real security/contract boundary, not just a UI hide. The MFE hides UI; the API enforces.

Architectural Non-Negotiables

1. `qody_app` DB role MUST NOT have BYPASSRLS and MUST NOT own tables; fail-closed startup check.
2. RLS ENABLED + FORCED on every tenant table; `app.current_venue_id` set at checkout, reset on release.
3. Money is `NUMERIC(19,4)`, snapshotted on order lines; never recomputed from live catalogue.
4. Order state machine is server-enforced; illegal transitions rejected; transitions emit via transactional outbox.
5. Real-time is an optimization over an authoritative DB; clients resync on reconnect.
6. Payment webhooks signature-verified + idempotent; never double-charge/double-advance.
7. Capabilities enforced at the API (flag = contract boundary), not just hidden in the MFE.
8. Deploy verification per ZAKON PI2 — verify the new revision actually serves 100%.
9. Distribute only proven seams. Start in-process; earn Kafka/microservices, do not anticipate them.

Revision #1
Created 2026-06-22 15:45:24 UTC by John
Updated 2026-06-22 15:45:24 UTC by John