Architecture

Tok technical architecture, Kotlin/Ktor, bank adapter pattern, GCP infrastructure

Tech Stack

Tech Stack

Tok is a Kotlin-native backend built for reliability and financial-grade security.

Core Stack

Layer	Technology	Notes
Language	**Kotlin**	JVM-based, coroutine-native
HTTP Framework	**Ktor**	Kotlin-idiomatic, coroutines-native routing
Dependency Injection	**Koin**	Lightweight, Kotlin-first DI
Database	**PostgreSQL**	Primary data store
ORM	**Exposed** (Kotlin SQL framework)	Type-safe SQL DSL
Connection Pooling	**HikariCP**	High-performance JDBC pool
DB Migrations	**Flyway**	Version-controlled schema migrations
Job Scheduling	**Quartz Scheduler + coroutines**	Bank sync scheduling
Serialization	**kotlinx.serialization**	Native Kotlin JSON
Build	**Gradle (Kotlin DSL)**	Multi-module project

Security & Encryption

Concern	Technology
Token encryption	**AES-256-GCM**
Key management	**GCP Cloud KMS** (HSM-backed)
PSD2 mTLS (QWAC)	DigiCert or GlobalSign certificate
CSRF protection	Cryptographic random `state` parameter per consent
Secret storage	**GCP Secret Manager**

Token encryption flow:

```
1. Receive OAuth token from bank API
2. Call GCP Cloud KMS generateDataKey (DEK + encrypted DEK)
3. Encrypt token with DEK (AES-256-GCM, random IV)
4. Store: encrypted_dek + iv + ciphertext in PostgreSQL
5. DEK discarded from memory after use
```

QWAC private key is stored in GCP Cloud KMS HSM — never extracted to filesystem.

Testing

Tool	Purpose
Kotest	Primary test framework (BDD-style)
MockK	Kotlin-idiomatic mocking
Testcontainers	Ephemeral PostgreSQL for integration tests

Cloud Infrastructure — GCP

Service	Purpose
Cloud Run	API server deployment (serverless containers)
Cloud SQL	Managed PostgreSQL
Cloud KMS	HSM-backed key management for OAuth tokens
Secret Manager	QWAC certs, API credentials

Data residency: `europe-north1` (Finland) — covers EU/GDPR requirements for Croatian data, and PDPL-equivalent requirements for Serbian data.

API Design

Aspect	Choice
Style	REST + OpenAPI 3.1
Auth	API keys (server-to-server) + OAuth2 (PSD2 consent flows)

Aspect	Choice
Multi-tenant	Organisation-scoped — each client = one organisation
Rate limiting	Per-organisation, tiered: Free / Pro / Enterprise

Core endpoints:

- `GET /accounts` — list bank accounts
- `GET /transactions` — fetch transactions (with date range filters)
- `POST /consents` — initiate PSD2 consent flow
- `POST /payments` — initiate payment (PISP — Phase 2)

Project Structure

```
Tok/
├── api/                        # Ktor API server (Gradle module)
│   └── src/
│       ├── main/kotlin/io/tokapi/
│       │   ├── Application.kt       # Ktor entry point
│       │   ├── adapters/             # BerlinGroupAdapter, BilateralAdapter
│       │   ├── consent/              # PSD2 consent management
│       │   ├── routes/               # Ktor routing
│       │   ├── services/             # Business logic
│       │   ├── models/               # Domain models + Exposed tables
│       │   └── plugins/              # Auth, rate-limit, logging, serialization
│       └── test/kotlin/io/tokapi/
├── sdk-kotlin/                 # Kotlin client SDK (for Bilko, Drop)
├── sdk-node/                   # Node.js client SDK (for third parties)
├── shared/                     # Shared domain types
├── docs/                       # Documentation
├── infrastructure/
│   ├── docker-compose.yml
│   └── terraform/              # GCP infrastructure as code
├── design/figma/
├── build.gradle.kts            # Root Gradle build
├── settings.gradle.kts         # Multi-module config
└── Dockerfile
```

SDKs

SDK	Language	Package
`sdk-kotlin/`	Kotlin	`io.tokapi:sdk-kotlin`
`sdk-node/`	TypeScript	`@tokapi/sdk`
`packages/sdk-python/`	Python 3.10+	`tokapi-sdk`

Bank Adapter Pattern

Bank Adapter Pattern

The bank adapter pattern is a core architectural principle in Tok. Every bank integration goes through an abstract `BankAdapter` interface, isolating the rest of the system from bank-specific API differences.

Why This Pattern?

Banks across Croatia, Serbia, and BiH use different API standards:

- **Croatia:** Berlin Group NextGenPSD2 (standardised, all HUB-registered banks)
- **Serbia (EU groups):** Berlin Group (UniCredit, Raiffeisen, NLB)
- **Serbia (domestic):** No central standard — bilateral per bank (AIK, OTP Serbia, Banca Intesa)
- **BiH:** Bilateral agreements only (no PSD2 mandate)

The adapter pattern hides this complexity behind one interface.

Abstract Interface

```
interface BankAdapter {
    /** Initiate OAuth consent flow — returns redirect URL */
    suspend fun initiateConsent(
        organizationId: String,
        callbackUrl: String
    ): ConsentRequest

    /** Exchange auth code for access + refresh tokens */
    suspend fun exchangeCode(code: String, state: String): OAuthTokens

    /** Refresh access token using refresh token */
    suspend fun refreshToken(refreshToken: String): OAuthTokens
```

```
    /** Fetch transactions for a date range */
    suspend fun fetchTransactions(
        accessToken: String,
        accountId: String,
        fromDate: LocalDate,
        toDate: LocalDate
    ): List<BankTransaction>

    /** Fetch account balance */
    suspend fun fetchBalance(
        accessToken: String,
        accountId: String
    ): BigDecimal

    /** Revoke consent */
    suspend fun revokeConsent(accessToken: String, consentId: String)
}
```

Implementations

BerlinGroupAdapter

Implements the **Berlin Group NextGenPSD2** standard.

Used for:

- All Croatian banks (registered with HUB — min. v1.3.8)
- EU bank groups in Serbia: UniCredit, Raiffeisen, NLB

Standard endpoints:

```
Consent URL:  {baseUrl}/v1/consents
Auth URL:     {baseUrl}/v1/oauth/authorize
Token URL:    {baseUrl}/v1/oauth/token
Accounts:     GET {baseUrl}/v1/accounts
Transactions: GET {baseUrl}/v1/accounts/{accountId}/transactions
              ?dateFrom={ISO}&dateTo={ISO}
```

Required headers:

```
Authorization: Bearer {access_token}
X-Request-ID:  {uuid}          ← unique per request
PSU-IP-Address: {user-ip}      ← required by some banks
```

Auth flow: OAuth 2.0 Authorization Code Grant + SCA redirect.

BilateralAdapter

Implements **per-bank custom REST** integrations for banks without a central standard.

Used for:

- Domestic Serbian banks (AIK, OTP Serbia, Banca Intesa Serbia)
- BiH banks under bilateral agreements
- Any bank that does not adopt Berlin Group

Each bilateral bank gets its own `BilateralAdapter` subclass with custom field mapping — the interface contract remains identical.

Bank Registry

```
val BANK_REGISTRY: Map<String, BankAdapterConfig> = mapOf(
    // Croatia (Berlin Group)
    "addiko-hr"    to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://oapideveloper.addiko.hr"
    ),
    "erste-hr"     to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://developers.erstegroup.com"
    ),
    "hpb-hr"       to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://openbanking.hpb.hr"
    ),
    "otp-hr"       to BankAdapterConfig(
        adapter = "BerlinGroup",
```

```
        baseUrl = "https://api.otpbanka.hr"
    ),
    "pbz-hr"       to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://apiportal.pbz.hr"
    ),
    "raiffeisen-hr" to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://sandbox.rba.hr"
    ),
    "zaba-hr"      to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://developer.unicredit.eu"
    ),
    // Serbia — EU groups (Berlin Group)
    "nlb-rs"       to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://developer.nlbkb.rs"
    ),
    "unicredit-rs" to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://developer.unicredit.eu"
    ),
    "raiffeisen-rs" to BankAdapterConfig(
        adapter = "BerlinGroup",
        baseUrl = "https://api.rbinternational.com"
    ),
    // Domestic Serbian banks — added as bilateral agreements are established
)
```

Transaction Normalization

Regardless of which adapter is used, all transactions are normalized to the internal `BankTransaction` format before being stored. This is the adapter's primary responsibility.

Internal format fields:

Field	Type	Source

`externalId`	String	Bank's own transaction ID (dedup key)
`bookingDate`	LocalDate	Berlin Group `bookingDate`
`valueDate`	LocalDate	Berlin Group `valueDate`
`amount`	NUMERIC(19,4)	Normalized — never float
`currency`	CurrencyCode	ISO 4217
`direction`	inbound/outbound	Derived from credit/debit indicator
`creditorIban`	String?	Berlin Group `creditorAccount.iban`
`debtorIban`	String?	Berlin Group `debtorAccount.iban`
`remittanceInfo`	String?	`remittanceInformationUnstructured`
`source`	String	`open_banking` (vs `manual` or `csv_import`)

Adding a New Bank

1. Determine which adapter applies (Berlin Group or bilateral)
2. Add entry to `BANK_REGISTRY` with `baseUrl`, `authUrl`, `scopes`
3. If bilateral — implement custom `fetchTransactions()` field mapping
4. Register sandbox credentials and test against bank sandbox
5. Add bank to UI: logo, display name, supported countries
6. Test deduplication against `externalId + bankAccountId` unique constraint

Development Rules

Development Rules

Seven mandatory rules for all Tok development. These rules exist because financial data and PSD2 compliance leave no room for shortcuts.

Rule 1 — Consent Immutability

PSD2 consent records are append-only audit trails.

Never update or delete a consent record. Each state change creates a new log entry:

```
consent_created → consent_exchanged → consent_active
                                    → consent_expired (90 days)
                                    → consent_revoked (user action)
```

All consent events are logged to `LoggedAction` (append-only). This is a legal compliance requirement under PSD2.

Rule 2 — Token Encryption Mandatory

AES-256-GCM + GCP Cloud KMS for ALL OAuth tokens. No exceptions.

```
CORRECT:  Store tokens via Cloud KMS envelope encryption → encrypted_dek + iv + ciphertext in
DB
WRONG:    Store raw tokens in DB or env vars
WRONG:    Store tokens in logs, files, or memory beyond request lifecycle
```

QWAC private key must also live in GCP Cloud KMS HSM — signing is done via Cloud KMS API, the key is never extracted.

Rule 3 — Bank Adapter Pattern

Every bank integration goes through the abstract `BankAdapter` interface.

```
// Correct
class BerlinGroupAdapter : BankAdapter { ... }
class BilateralAdapter   : BankAdapter { ... }

// Wrong — never call bank HTTP endpoints directly from services/routes
```

The adapter is the only layer that knows about bank-specific API formats. Services above it work only with normalized `BankTransaction` objects.

Rule 4 — Deduplication via externalId

`externalId` (bank's own transaction ID) + `bankAccountId` = unique constraint.

Duplicate imports are silently skipped — this is intentional, not an error. Never create duplicate transactions.

```
ALTER TABLE bank_transactions
ADD CONSTRAINT uq_bank_account_external
UNIQUE (bank_account_id, external_id);
```

Rule 5 — Money = NUMERIC(19,4)

Never use float or double for financial amounts.

```
// Correct
val amount: BigDecimal  // Kotlin
// Correct in DB
amount NUMERIC(19,4)

// Wrong
val amount: Double  // loses precision
val amount: Float   // loses precision
```

All amounts from bank APIs must be parsed to `BigDecimal` before storage. Amount equality comparisons use exact decimal matching.

Rule 6 — CSRF on Consent

Every OAuth consent flow must include a cryptographically random `state` parameter.

```
// Correct
val state = java.security.SecureRandom()
    .generateSeed(32)
    .toHexString()
// Store in server-side session (NOT cookie, NOT localStorage)
// Validate on callback — reject if mismatch
// One-time use — invalidate after successful exchange

// Wrong
val state = UUID.randomUUID().toString()  // too predictable
val state = "fixed-string"               // completely insecure
```

Rule 7 — 90-Day Consent Tracking

Every `BankConnection` must have automated expiry monitoring.

PSD2 (EBA RTS Art. 10) requires re-authentication every 90 days.

Mandatory implementation:

- `consentValidUntil` field on every `BankConnection`
- Daily cron: check all active connections
- 14 days before expiry: send email to org admin
- On expiry: set `consentStatus = 'expired'`, pause sync jobs
- UI: show "Bank feed paused — click to reconnect"
- One-click re-connect flow (user re-does SCA, new tokens stored)

Without this, bank feed silently breaks for ALL users when consents expire simultaneously.

Summary

Rule	Enforcement
1. Consent immutability	Code review — no UPDATE/DELETE on consent tables
2. Token encryption	No raw token strings in code/DB/logs
3. Bank adapter pattern	No direct HTTP bank calls outside adapter layer
4. Deduplication	DB unique constraint enforced
5. Money = NUMERIC(19,4)	No float/double for amounts anywhere
6. CSRF on consent	State parameter required in every consent initiation
7. 90-day tracking	Daily cron + email notification mandatory