Architecture Document: Wallet Service

Version: 1.1

Date: February 10, 2025

Author: Garel Silva
Contact: +55 19 9 9969 0160
email: garel_vinicius@hotmail.com

1. Introduction

This document describes the proposed architecture for developing a digital wallet service (Wallet Service) responsible for managing users' funds. The project aims to provide high availability, transactional consistency, scalability, auditability, and security for financial operations, while also publishing events to integrate with other ecosystem components.

1.1 Objectives

- Provide a service for creating and managing wallets linked to users.
- Enable operations such as depositing, withdrawing, and transferring funds.
- Ensure that balances and transactions are stored and queried reliably and auditable.
- Integrate securely with other modules, ensuring authentication, permissions, and validations.
- Publish events (via a message broker) for reporting, notifications, anti-fraud, etc.
- Expose reactive endpoints (using Spring WebFlux + R2DBC) for wallet manipulation and queries.

1.2 Scope

- Creation and maintenance of wallets and associated users.
- Transactional operations: deposits, withdrawals, and transfers.
- Balance queries (both current and historical).
- Reliable persistence (atomic transactions, audit logs).

- Security: authentication, authorization, access control, and business rule validations.
- Event publishing for asynchronous notifications and data sharing.

2. Requirements

2.1 Functional Requirements

- Create a wallet for a user, starting with a zero balance.
- Deposit funds into a wallet.
- Withdraw funds from a wallet, validating the available balance.
- Transfer funds between two wallets.
- Retrieve the current balance of a wallet.
- Retrieve a historical balance (at a specific date/time).

2.2 Non-Functional Requirements

- **Reliability:** Prevent data loss or inconsistencies that could compromise the platform.
- **Scalability:** Handle a high volume of concurrent transactions.
- **Auditability:** All operations must be fully traceable and reconstructable.
- **Reactivity:** Use a non-blocking framework (Spring WebFlux) and reactive persistence (R2DBC).
- **Event Publishing:** Use a message broker (Kafka/RabbitMQ) to integrate with external services.
- **Security:**
    - **Authentication:** Ensure that only authenticated users can perform operations.
    - **Authorization/Permissions:** Appropriately control access to wallets and data (e.g., users can only operate on their own wallets).
    - **Validation:** Enforce input validation and consistency of business rules.

3. High-Level Architecture Overview

3.1 Diagram

Below is an ASCII diagram illustrating the hybrid reactive architecture. It provides transactional persistence for local consistency and event publishing to external consumers, including a security layer:

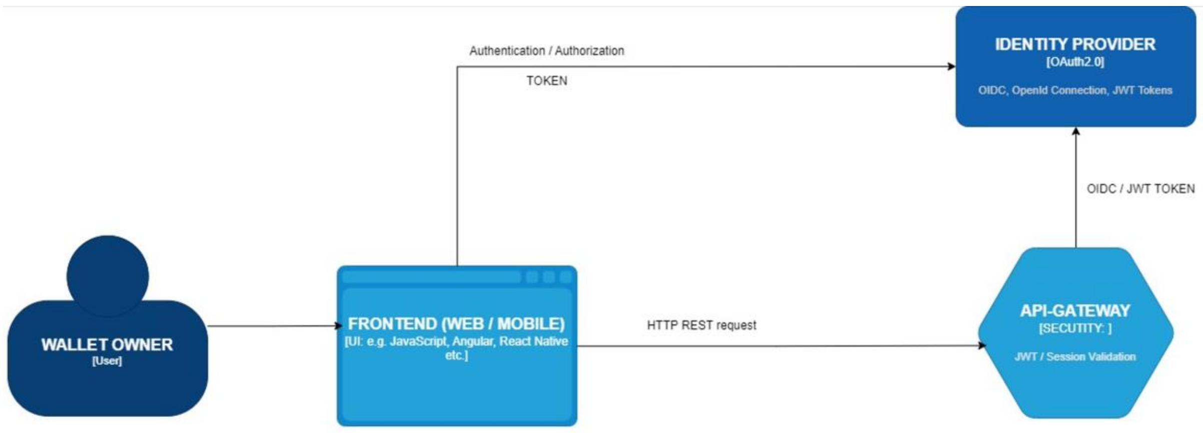
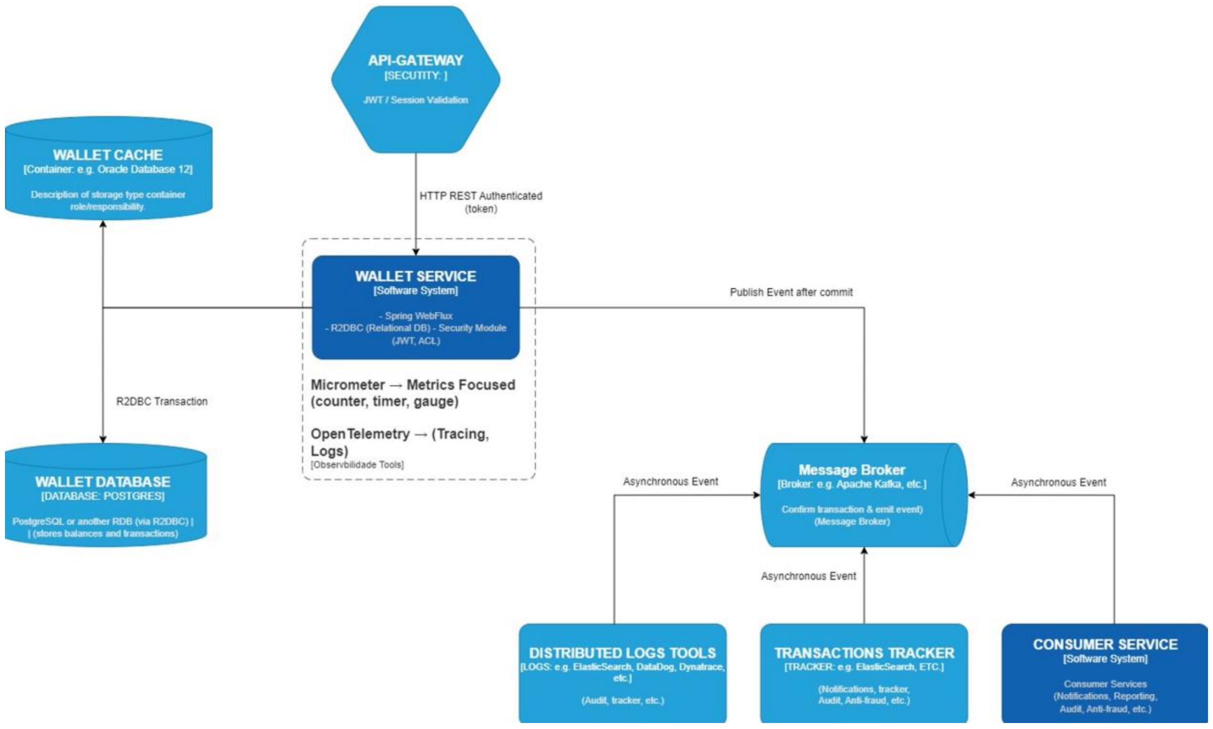


Diagram 1: Layer 1 High Level



API-GATEWAY
[SECUTITY:]
JWT / Session Validation
WALLET CACHE
[Container: e.g. Oracle Database 12]
Description of storage type container role/responsibility.
HTTP REST Authenticated (token)
WALLET SERVICE
[Software System]
- Spring WebFlux
- R2DBC (Relational DB) - Security Module (JWT, ACL)
Micrometer → Metrics Focused (counter, timer, gauge)
OpenTelemetry → (Tracing, Logs)
[Observbilidade Tools]
Publish Event after commit
R2DBC Transaction
WALLET DATABASE
[DATABASE: POSTGRES]
PostgreSQL or another RDB (via R2DBC) | | (stores balances and transactions)
Message Broker
[Broker: e.g. Apache Kafka, etc.]
Confirm transaction & emit event)
(Message Broker)
Asynchronous Event
Asynchronous Event
Asynchronous Event
DISTRIBUTED LOGS TOOLS
[LOGS: e.g. ElasticSearch, DataDog, Dynatrace, etc.]
(Audit, tracker, etc.)
TRANSACTIONS TRACKER
[TRACKER: e.g. ElasticSearch, ETC.]
(Notifications, tracker, Audit, Anti-fraud, etc.)
CONSUMER SERVICE
[Software System]
Consumer Services
(Notifications, Reporting, Audit, Anti-fraud, etc.)

Diagram 2: Layer 2 High Level

Financial Operations: Deposits, withdrawals, and transfers are performed atomically in the database via R2DBC.
Events: Published after each successful transaction commit, enabling asynchronous integration with reporting, notifications, etc.
Security: Uses tokens (JWT, OAuth2/OIDC) for authentication and in-service authorization (wallet-level permissions).

4. Data Flow and Security

- The user obtains an access token (JWT) from an Identity Provider (e.g., Keycloak, Auth0, etc.).
- The client (e.g., a frontend application) sends HTTP requests to the Wallet Service along with the token.
- The Wallet Service validates the token (signature, expiration, scopes) and checks permissions (e.g., whether the user can operate on this wallet).
- Upon successful validation, it initiates a reactive transactional flow to process the deposit, withdrawal, or transfer in the database.
- After the transaction is confirmed, the service publishes an event to the message broker.
- Consumer services process these events independently (reporting, notifications, anti-fraud, etc.).

5. Main Components

5.1 Wallet Service (Reactive Application)

HTTP Layer (Spring WebFlux):

- Typical endpoints:
  - POST /wallets (create wallet)
  - POST /wallets/{id}/deposit
  - POST /wallets/{id}/withdraw
  - POST /wallets/{sourceId}/transfer/{destId}
  - GET /wallets/{id}/balance
  - GET /wallets/{id}/balance?date=... (historical balance)

Security Middleware:

- Validates JWT tokens and extracts claims (user ID, permissions).

Business Layer:

- Handles wallet creation, deposit, withdrawal, and transfer logic. Includes balance checks, business rules, and safeguards against inconsistencies.

Persistence Layer (R2DBC):

- **Main tables:**
  - **User:** Stores local user information or an external ID if integrated with an identity provider.
  - **Wallet:** Fields: wallet_id, user_id, current_balance, status, etc.

    - **Transaction:** Fields: transaction_id, wallet_id, type (DEPOSIT, WITHDRAW, TRANSFER), amount, timestamp, etc.
- Atomic transactions ensure that both transaction records and wallet balance updates occur together.

5.2 Relational Database (PostgreSQL, MySQL, etc.)

- Supports transactions with a high isolation level (e.g., REPEATABLE READ, SERIALIZABLE).
- Constraints and foreign keys ensure referential integrity among User, Wallet, and Transaction.
- Indexes in the Transaction table (e.g., by wallet_id and date) optimize historical balance queries.

5.3 Message Broker (Kafka/RabbitMQ)

- Receives events from the Wallet Service after transaction commits.
- Stores messages for consumer services (notifications, auditing, reporting, etc.).
- Ensures decoupling between the wallet microservice and other functionality.

5.4 Consumer Services (Asynchronous)

- **Notifications:** Send emails or push messages about transactions.
- **Reporting:** Generate usage statistics and financial volume data.
- **Anti-Fraud, External Audit, etc.:** Each service processes events according to its logic, without impacting the Wallet Service's critical flow.

5.5 Security / Authorization Module

- **Token Validation:** Checks token signature, expiration, and scopes.
- **Authorization:** Ensures only the wallet owner (or an admin) can perform certain commands.
- **Access Rules (simple example):**
    - wallet.user_id == token.user_id → the user can manage deposits/withdrawals/transfers on that wallet.
- **Replay Protection:** Track a nonce or requestId to avoid duplicate requests.

6 Transaction Tracing and Correlation

6.1 Unique Identifier: *requestTransactionId*

To ensure complete traceability of every operation within the Wallet Service, the **requestTransactionId** identifier has been implemented. This field is generated for each request and serves as a robust hash that aggregates relevant user information (e.g., userId), the timestamp of the request, and a random component (nonce) to guarantee both uniqueness and unpredictability.

6.2 Generation, Propagation, and Storage Mechanism

- **Generation:**
  When a request is received, the entry layer (Spring WebFlux) calls an identifier generator that employs a hash algorithm (such as SHA-256) to combine:
    - User data (e.g., userId),
    - Request timestamp,
    - A random value (nonce).

This combination results in the **requestTransactionId**, which acts as the “fingerprint” of the operation.

- **Propagation:**
  Once generated, the **requestTransactionId** is injected into the request metadata and propagated through all layers of the system:
    - **HTTP Layer:** Included in the request header to trace incoming traffic.
    - **Business and Persistence Layers:** Passed along to transaction records and audit logs.
    - **Event Publishing:** Incorporated into messages sent to the message broker (Kafka/RabbitMQ), enabling correlation between operations and events consumed by other services.
- **Storage:**
  Every operation recorded in the **Transaction** table—as well as in structured audit logs—stores the associated **requestTransactionId**. This allows:
    - Identification of the origin and complete flow of the transaction.
    - Facilitation of investigating inconsistencies or failures by correlating distributed logs.

6.3 Architectural Benefits and Justifications

- **Complete Auditability:**
  The inclusion of the **requestTransactionId** in every record enables reconstructing the entire transaction path, thereby simplifying audits and security investigations. Every operation can be traced from the initial request through to its publication in the message broker.

- **Diagnosis and Debugging:**
  In cases of failure or unexpected behavior, the identifier facilitates the correlation of logs across various system layers (HTTP, business logic,

database, and messaging). This accelerates issue diagnosis and incident resolution.

- **Integration with Distributed Tracing Systems:** The solution integrates seamlessly with tracing tools such as Jaeger, Zipkin, or OpenTelemetry. This integration provides end-to-end visibility of transactions, enabling detailed performance monitoring and the identification of bottlenecks or failures within distributed environments.

- **Reduction of Risks and Inconsistencies:** Ensuring that each operation has a unique, traceable identifier minimizes the risk of duplication or loss of critical information. In high-concurrency scenarios, this approach strengthens data consistency and transactional integrity.

- **Standardization and Monitoring:** The systematic inclusion of **requestTransactionId** in logs, events, and transaction records enables the creation of customized dashboards and alerts. This facilitates real-time monitoring of transaction flows and proactive responses to anomalies.

6.4 Future Considerations and Improvements

- **Automated Log Correlation:** Standardizing the **requestTransactionId** facilitates integration with centralized log management solutions (such as the ELK Stack or Grafana Loki), enhancing the analysis and correlation of events.

- **Enhanced Monitoring Mechanisms:** The identifier can be leveraged to generate detailed metrics, allowing for in-depth monitoring of each processing step within a transaction, thereby improving overall system observability.

- **Enhancement in Hash Generation:** Periodic reviews of the hash algorithm and the components used (userId, timestamp, nonce) should be conducted to ensure the mechanism remains secure and performant as environmental conditions and business requirements evolve.

By incorporating this tracing solution, the Wallet Service not only reinforces its transactional integrity and reliability but also aligns with best practices in auditability and monitoring, providing a solid foundation for the scalability and security of financial operations.

7. Key Flows (with Security and Validations)

Note: In addition to the transactional flow described for each operation (deposit, withdrawal, transfer, balance update), an event-driven process via Kafka (using a queue) has been considered to ensure higher volume, consistency, and reliability in processing. Thus, each operation generates an event that is queued and consumed, allowing for scalability and robustness in handling large volumes of concurrent transactions.

7.1 Create Wallet

Request:

POST /wallets + Authentication Header (Bearer Token).

Process:

- Validate the token and check permission to create a wallet (e.g., any authenticated user may do so).
- Insert a record into the Wallet table (balance = 0), associated with the user_id from the token.
- Commit the transaction.
- (Optional) Publish a “WalletCreatedEvent.”

7.2 Deposit

Request:

POST /wallets/{id}/deposit + token.

Process:

- Validate the token and confirm that the user has permission for wallet {id}.
- Start a transaction.
- Insert a record into the Transaction table (type DEPOSIT).
- Update current_balance by adding the deposit amount.
- Commit the transaction.
- Publish a “TransactionCreatedEvent.”

7.3 Withdraw

Request:

POST /wallets/{id}/withdraw + token.

Process:

- Validate the token and verify wallet ownership.
- Check if current_balance is greater than or equal to the requested amount.
- Start a transaction and insert a record into the Transaction table (type WITHDRAW).
- Update current_balance by subtracting the withdrawal amount.
- Commit the transaction.
- Publish the corresponding event.

7.4 Transfer

Request:

POST /wallets/{sourceId}/transfer/{destId} + token + amount.

Process:

- Validate the token and confirm the user has permission to move funds from sourceId.
- Check if there is sufficient balance in sourceId.
- Within a single transaction:
  - Insert a debit transaction for sourceId.
  - Update the source wallet balance.
  - Insert a credit transaction for destId.
  - Update the destination wallet balance.
- Commit the transaction.
- Publish a “TransferCompletedEvent” or similar.

7.5 Balance and Historical Queries

Request:

GET /wallets/{id}/balance + token.

Authorization check:

- token.user_id == wallet.user_id (or admin).
- Return the current_balance.

Request:

GET /wallets/{id}/balance?date=... + token.

Process:

- Perform the same access checks.
- Compute the historical balance by summing all transactions up to the specified date/time (or using snapshots).

8. Reliability and Security Mechanisms

- **Atomic Transactions:** Updating balances and inserting into the Transaction table must happen together to guarantee consistency.

- **Isolation:** Use a high isolation level (e.g., SERIALIZABLE) to avoid incorrect reads or negative balances under high concurrency.
- **Audit Logs:** Transaction table, plus logs for all critical operations (timestamps, user info).
- **JWT Token Validation:** Ensures only authenticated users reach the endpoints.
- **Authorization (RBAC or ABAC):**
  - **RBAC (Role-Based Access Control):** e.g., admin vs. regular user roles.
  - **ABAC (Attribute-Based Access Control):** e.g., wallet.user_id == token.user_id.
- **Replay/Idempotency Protection:** Use a unique requestId for sensitive operations to avoid duplicates upon client retries.
- **Encryption in Transit:** Use HTTPS for communication between the client and the service, and preferably between internal services as well.
- **Error & Fraud Monitoring:** Suspicious transactions can be flagged to anti-fraud services (via events).

9. Observability and Monitoring

- **Structured Logs:** Record all calls with key data (transactions, IDs, amounts).
- **Metrics:** Use Micrometer & Actuator to track response time, throughput, errors, etc.
- **Health Checks:** (e.g., /actuator/health) to validate the Wallet Service's availability.
- **Alerts:** Integrated into observability tools (Prometheus, Grafana, Kibana) for anomalies or error spikes.

10. Final Considerations

This reactive and hybrid architecture (transactional persistence + event publishing) has been expanded to include security (token-based authentication, authorization rules, and business validations) and risk mitigation measures to prevent data loss or unauthorized access.

Key Benefits:

- **Financial Consistency:** Each operation occurs within atomic transactions in the database.
- **Security and Privacy:** JWT/OAuth2 ensures only authorized entities can manipulate wallets.
- **Auditability:** Transaction logs and events enable comprehensive tracking and auditing.

- **Scalability:** Spring WebFlux + R2DBC efficiently handles high traffic with minimal resource blocking.
- **Decoupling:** Events allow other services (notifications, reporting, anti-fraud) to function independently without burdening the main service.

Approval:

Architecture and Security Teams

Next Steps:

- Deploy and configure the Identity Provider (Keycloak, Auth0, etc.) for issuing JWT/OAuth2 tokens.
- Adjust the Wallet Service to validate tokens and apply authorization rules.
- Set up CI/CD pipelines, provision the relational database (in reactive mode) and the message broker.
- Develop consumer microservices (notifications, reporting, etc.) that process the published events.