Design Decisions – Wallet Service

The Wallet Service is designed to manage user funds with full traceability and reliability. The solution enables the creation of wallets, processing of deposits, withdrawals, transfers, and querying of current and historical balances. Despite the project being time-boxed, the architecture and implementation follow professional software engineering standards to emulate a production-quality system.

Architectural Overview

The system is implemented using Java with Spring Boot, providing a clean and modular REST API. It is fully containerized with Docker, making it easily deployable via docker-compose.

Core Design Principles

1. RESTful API Design

Each operation (create wallet, deposit, withdraw, transfer, retrieve balances) is exposed as a RESTful endpoint. HTTP status codes and structured JSON responses are used for clarity and standardization.

2. Transactional Integrity

All money-related operations are executed within ACID-compliant transactions. Operations like transfer (which involves two wallets) are handled in a single transaction to prevent partial updates.

3. Auditability

Every financial operation (deposit, withdrawal, transfer) is persisted as a separate transaction record. Each wallet's balance is calculated based on these immutable records, supporting full traceability and historical lookup.

4. Historical Balance Retrieval (Statement)

Historical balance is implemented by querying transaction history up to a given timestamp. It returns a pageable statement containing all action made in this wallet in the given time This ensures accurate reconstruction of wallet states at any past point, essential for audits.

5. Domain-Driven Approach

Clear separation of **domain models**, **service layer**, and **persistence layer**. Business rules (e.g., disallowing negative balances) are enforced in the service layer.

6. Dockerized Deployment

The project has Postgres and kafka as dependencias. In order to run the application, use `docker-compose up -d` to run the necessary dependencies. Then, use `./mvnw spring-boot:run` to run the application.

Functional Coverage

Feature	Implemented	Notes
Create Wallet	✅	Each user can own one or more wallets.
Retrieve Current Balance	✅	Returns the latest balance computed from transactions.
Retrieve Historical Balance	✅	Returns balance at any past timestamp.
Deposit Funds	✅	Adds credit to the user’s wallet.
Withdraw Funds	✅	Deducts funds if sufficient balance exists.
Transfer Funds	✅	Atomic transfer between two user wallets.

Non-Functional Coverage

High Reliability

Transactional operations and error handling prevent inconsistent states.

Full Traceability

All transactions are logged in a persistent, auditable format.

Containerization

Easy to spin up/down with consistent environments using Docker.

Time Spent

Task	Estimated Time
Project setup and dependencies	1h
Designing entities and database	20 min
Implementing project	6h
Dockerizing and testing	1h
Documentation and polish	40min
Total Estimated Time	**9h**

Technologies Used

- Java 17
- Spring Boot 3.5
- Spring Data JPA
- PostgreSQL (via Docker)
- Kafka
- Pgadmin
- zookeeper
- Docker / Docker Compose

Test application

To test the application, access the Swagger UI at: http://localhost:8080/swagger-ui-custom.html. **Ensure the application is running before accessing this endpoint.** This interface provides a complete list of available endpoints and their expected inputs for testing and exploration.

Limitations

Security

No authentication or authorization mechanisms have been implemented. The system is currently open and unsecured — not suitable for production environments in its current state.

Testing

Automated tests (unit, integration, or end-to-end) have not yet been implemented. No stress, load, or scalability tests have been performed.

Scalability & Data Consistency

Kafka was integrated to support scalability through asynchronous processing. However, due to time constraints, known data consistency concerns were not addressed. Issues such as potential message loss, ordering, or duplication may occur in high-concurrency scenarios.

Design Notes

Several architectural improvements were identified during development like eventual consistency mechanisms, retries.These were not implemented in the current version due to limited development time but are recommended for future iterations.

Error handling

While a centralized error handling mechanism was planned, user-facing error messages are currently not being reported properly. Many exceptions return generic or uninformative responses (e.g., HTTP 500 without context), which may hinder debugging and user feedback. Error details are not consistently structured, which affects client-side parsing and UI error presentation.