Marcus Rappenborg Kjærsgaard
21-11-2025

Software Requirements Specification (SRS) | version 2.1

Multi-Currency Wallet Simulator

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to specify the functional and non-functional requirements for a multi-currency wallet simulator.

The system is intended for use in an educational setting, where it serves as a basis for implementation, automated testing, and demonstration of software testing techniques.

The document is aimed at stakeholders such as:

- Developers implementing the system.
- Testers designing and executing tests.
- Instructors and examiners evaluating the implementation and test approach.

1.2 Scope

The system simulates a simple multi-currency wallet for a single logical user. It allows the user to:

- Create and view wallets in supported currencies.
- View current balances for each wallet.
- Perform deposits into wallets.
- Perform withdrawals from wallets, subject to balance constraints.
- Perform exchanges between wallets, including cross-currency exchanges.
- View a transaction history for the user's wallets.

The system operates purely on simulated data. It does **not** connect to real banks or payment providers, and it does not handle real money or sensitive payment information.

The following are explicitly out of scope:

- User authentication and authorization.
- Multiple user accounts or access control between users.
- Currencies other than the supported set defined in this document.
- Real financial integrations (e.g. card payments, bank transfers).

1.3 Definitions

For the purposes of this document, the following terms are used:

- **Wallet:** A record that holds a balance in exactly one supported currency and has an associated status (active, frozen, closed).

- **Transaction:** A single operation that changes, or attempts to change, wallet balances. The system supports deposits, withdrawals, and exchanges as transaction types. Each transaction has a status (completed or failed).
- **Balance:** The numeric amount of money currently held in a wallet, expressed in the wallet's currency.
- **Supported currency:** One of the currencies that the system is allowed to use for wallets and transactions. In this system, the supported currencies are Danish krone (DKK), euro (EUR), and US dollar (USD).
- **Wallet status:** The lifecycle state of a wallet. The system distinguishes at least the states active, frozen, and closed. The permitted transitions and their effects are described in later sections.
- **Invalid input:** A request that violates the system's input constraints, such as using an unsupported currency, referencing a non-existent wallet, providing a non-numeric amount, or providing an amount that is not strictly greater than zero.
- **External exchange-rate API:** The external HTTP-based service used by the system to obtain currency exchange rates between the supported currencies.
- **Completed transaction:** A transaction that passes validation and is fully applied, resulting in updated balances.
- **Failed transaction:** A transaction that is rejected due to invalid input or an error (for example, an external API failure). Failed transactions do not change any wallet balances but are still recorded with a failure status.

2. System Overview and Constraints

2.1 System Context

The system is a web-based multi-currency wallet simulator used by a single logical user through a standard desktop web browser.

The user interacts with a browser-based user interface. The user interface communicates with a backend service implemented as a Flask-based REST API over HTTP using JSON request and response bodies. The backend stores wallet and transaction data in a local relational database and, when needed, calls an external exchange-rate API to obtain currency conversion rates between the supported currencies.

Apart from the external exchange-rate API, the system does not integrate with any other external systems.

2.2 Architecture and Technology Constraints

The system shall be implemented using the following architecture and technology constraints:

- **Frontend**

- Implemented as a web user interface using HTML5, CSS, and JavaScript.
- Runs in a standard desktop web browser.
- Communicates exclusively with the backend REST API over HTTP/HTTPS.

- **Backend**
  - Implemented in Python using the Flask framework.
  - Exposes a REST-style JSON API for all wallet and transaction operations.
  - Responsible for input validation, business rules, persistence, and integration with the external exchange-rate API.
- **Database**
  - Uses SQLite as the primary relational data store.
  - Stores wallets, transactions, and any related domain data.
- **Caching**
  - The system may use a caching mechanism for exchange rates.
  - Redis may be introduced in later iterations as an external cache store; the initial version shall be able to function without Redis.
- **External exchange-rate API**
  - The backend integrates with a public HTTP-based exchange-rate API that provides rates between DKK, EUR, and USD.
  - The exact provider may be configurable, but the API must return machine-readable exchange rates (for example, as JSON) over HTTPS.

No other architectural styles or frameworks are required or assumed beyond those specified above.

2.3 Execution Environment

The reference execution environment for development, demonstration, and testing is:

- **Hardware**
  - A MacBook with an Apple M2 processor.
- **Operating System**
  - macOS Sequoia 15.6.1.
- **Web Browsers**
  - Safari version 26.0 or later macOS.
  - Google Chrome version 142.0.7444.176 or later macOS.
- **Containerization**
  - The system may be run directly on macOS (using a local Python environment and SQLite database file) or inside a Docker container.

- When Docker is used, a provided Docker image and tag shall be used so that the environment is reproducible for testing.

Performance and usability requirements stated later in this document are to be understood with respect to this reference environment.

3. Functional Requirements

3.1 Domain and Supported Currencies

3.1.1 Supported Currencies

The system shall support the following currencies:

- Danish krone (DKK)
- Euro (EUR)
- US dollar (USD)

Any attempt to create a wallet or perform a transaction using a currency other than DKK, EUR, or USD shall be treated as invalid input.

3.1.2 Wallets and Ownership

The system models wallets for a single logical user.

Each wallet shall have:

- A unique identifier.
- A currency, which shall be one of the supported currencies.
- A balance in that currency.
- A status (for example, active, frozen, closed), as defined in section 3.2.

All wallets belong to the same logical user. The system shall not implement user authentication, registration, or access control between different users.

3.1.3 Balances

The balance of a wallet represents the amount of money held in that wallet, expressed in the wallet's currency.

- Balances shall be stored as numeric values with at least two decimal places of precision.
- The system shall ensure that balances cannot become negative as a result of any successful transaction.

3.2 Wallet Lifecycle and Status

3.2.1 Wallet Creation

The system shall allow the user to create a new wallet by selecting one of the supported currencies.

When a new wallet is created:

- The wallet's currency shall be set to the selected supported currency.
- The wallet's initial status shall be **active**.
- The wallet's initial balance shall be:
  - 0, or
  - The amount of a validated initial deposit, if such a deposit is provided and passes the deposit validation rules defined in section 3.3.1.

If wallet creation fails due to invalid input (for example, unsupported currency or invalid initial deposit), no wallet shall be created.

3.2.2 Wallet States

A wallet shall be in exactly one of the following states:

- **Active:** The wallet can be used for all supported transactions (deposits, withdrawals, exchanges), subject to validation rules.
- **Frozen:** The wallet is visible but cannot be used as source or target for any new transaction.
- **Closed:** The wallet is in a final state. It is visible for historical purposes, but no further transactions shall be applied to it.

3.2.3 Allowed State Transitions

The system shall support the following state transitions:

- active → frozen
- frozen → active
- active → closed
- frozen → closed

The system shall not allow any state transitions from closed to another state. Any attempt to change the state of a closed wallet shall be treated as invalid input.

The system shall not allow any state transitions other than those explicitly listed above.

3.2.4 Effect of Status on Operations

The system shall enforce the following rules:

- Deposits, withdrawals, and exchanges are allowed only for **active** wallets.

- A wallet in the **frozen** or **closed** state shall not be used as the source or target of any new transaction.
- Any request that attempts to perform a transaction on a frozen or closed wallet shall be treated as invalid input and shall not change any balances or wallet states.

The system may provide simple user interface actions to change a wallet's status between the allowed states. The exact user interface layout is not specified in this document.

3.3 Transactions

The system shall support three types of transactions:

- Deposit
- Withdrawal
- Exchange (including same-currency and cross-currency exchanges)

For all transaction types, the system shall:

- Validate the request according to the rules for that transaction type.
- Either:
  - Apply the transaction fully and record it as completed, or
  - Reject the transaction, leave all balances unchanged, and record it as failed.

3.3.1 Deposits

A deposit increases the balance of a wallet by a specified amount.

Validation rules

A deposit request shall be considered valid if all the following conditions are met:

- The referenced wallet exists.
- The referenced wallet is in the **active** state.
- The deposit amount is numeric.
- The deposit amount is strictly greater than 0.
- The deposit currency matches the wallet's currency.

If any of these conditions are not satisfied, the deposit request shall be treated as invalid input.

Effects on success

If a deposit request is valid:

- The system shall increase the wallet's balance by the deposit amount.
- The system shall record a completed deposit transaction.

Effects on failure

If a deposit request is invalid or fails for any reason:

- The system shall not change the balance of any wallet.

- The system shall record a failed deposit transaction, including an appropriate error code.

3.3.2 Withdrawals

A withdrawal decreases the balance of a wallet by a specified amount.

Validation rules

A withdrawal request shall be considered valid if all the following conditions are met:

- The referenced wallet exists.
- The referenced wallet is in the **active** state.
- The withdrawal amount is numeric.
- The withdrawal amount is strictly greater than 0.
- The withdrawal amount is less than or equal to the current balance of the wallet.

If any of these conditions are not satisfied, the withdrawal request shall be treated as invalid input.

Effects on success

If a withdrawal request is valid:

- The system shall decrease the wallet's balance by the withdrawal amount.
- The system shall record a completed withdrawal transaction.

Effects on failure

If a withdrawal request is invalid or fails for any reason:

- The system shall not change the balance of any wallet.
- The system shall record a failed withdrawal transaction, including an appropriate error code.

3.3.3 Exchanges

An exchange transfers an amount from a source wallet to a target wallet. The source and target wallets may have the same currency or different currencies.

Validation rules

An exchange request shall be considered valid if all the following conditions are met:

- The source wallet exists.
- The target wallet exists.
- Both the source and target wallets are in the **active** state.
- The exchange amount is numeric.
- The exchange amount is strictly greater than 0.
- The exchange amount is less than or equal to the current balance of the source wallet.

If any of these conditions are not satisfied, the exchange request shall be treated as invalid input.

Same-currency exchanges

If the source and target wallets use the same currency:

- The system shall treat the exchange rate as exactly **1.00**.
- The system shall not call the external exchange-rate API for this transaction.
- The system shall decrease the source wallet's balance by the full exchange amount and increase the target wallet's balance by the same amount.

Cross-currency exchanges

If the source and target wallets use different currencies:

- The system shall obtain an exchange rate between the source and target currencies, either:
  - By calling the external exchange-rate API, or
  - By using a cached rate, if a suitable cached rate is available and considered recent enough according to system configuration.

If no valid rate can be obtained (for example, due to external API failure or invalid data), the exchange shall fail as described in section 3.3.4.

3.3.4 Exchange Rates, Rounding, and Caching

Use of exchange rates

For cross-currency exchanges, the system shall compute the credited amount in the target currency as:

- raw_amount = source_amount * rate
- credited_amount = rounded(raw_amount)

Rounding strategy

The system shall apply a consistent, deterministic rounding strategy for all credited amounts. The rounding strategy shall be:

- Round half up to 2 decimal places.

Example:

- Source amount: 100.00
- Exchange rate: 1.23456
- Raw amount: 123.456
- Credited amount (after rounding): 123.46

Another example:

- Raw amount: 100.005
- Credited amount (after rounding): 100.01

The system shall apply rounding once to the final credited amount in the target currency.

Caching of exchange rates

The system may cache exchange rates for reuse in later exchanges. The initial version may use in-memory caching or no caching. Later versions may use Redis as an external cache store.

If caching is used:

- Cached rates shall be considered valid only for a configurable maximum age.
- When a suitable cached rate is available and not expired, the system may use it instead of calling the external exchange-rate API.

External API failures

If the external exchange-rate API is unavailable, returns an error, or returns data that cannot be interpreted as a valid rate:

- The exchange transaction shall fail.
- No wallet balances shall be changed.
- The system shall record a failed exchange transaction, including an appropriate error code indicating an external service problem.

3.3.5 Effects on Success and Failure

For all transaction types (deposit, withdrawal, exchange), the system shall follow these rules:

- If the transaction passes validation and any required exchange-rate lookup succeed:
  - The resulting balance changes shall be applied atomically.
  - The transaction shall be recorded as completed.
- If the transaction is invalid or an error occurs:
  - No wallet balances or wallet states shall be changed.
  - The transaction shall be recorded as failed with an appropriate error code.

3.4 Transaction Recording and History

3.4.1 Transaction Recording

For each transaction (deposit, withdrawal, exchange), the system shall record at least the following information:

- A unique transaction identifier.
- Timestamp of when the transaction was processed.
- Transaction type (deposit, withdrawal, exchange).
- Source wallet identifier, if applicable.
- Target wallet identifier, if applicable.
- Input amount and currency.

- For exchanges, the credited amount and currency in the target wallet.
- Resulting balances for the affected wallets, for completed transactions.
- Transaction status (completed or failed).
- An error code or category, for failed transactions.

The system shall store this information in the database so it can be retrieved later.

3.4.2 History Views

The system shall provide a way for the user to view transaction history per wallet.

For a given wallet:

- The history view shall include all transactions where the wallet was either:
  - The source wallet, or
  - The target wallet.
- The history shall include both completed and failed transactions.
- Transactions in the history view shall be ordered by timestamp, with the **newest transactions first**.

The exact layout and styling of the history view in the user interface is not specified in this document, if the information listed above is accessible to the user.

4. Validation and Error Handling

4.1 Invalid Input

The system shall treat a request as **invalid input** if it violates any of the input constraints defined in this document. Invalid input includes, but is not limited to, the following cases:

- The specified wallet identifier does not exist.
- The specified wallet is not in a state that allows the requested operation (for example, frozen or closed when an active wallet is required).
- The specified currency is not one of the supported currencies (DKK, EUR, USD).
- The specified amount is missing, non-numeric, or cannot be parsed as a valid number.
- The specified amount is not strictly greater than 0, when a positive amount is required.
- The specified amount exceeds the available balance in the source wallet, when a balance constraint applies.
- Required fields in the request are missing or use an incorrect format.

For any request that is classified as invalid input:

- The system shall not change the balance of any wallet.
- The system shall not change the status of any wallet.

- If the request corresponds to a transaction (deposit, withdrawal, exchange), the system shall record a failed transaction with an appropriate error code.
- The backend API shall return a client error status code in the 4xx range (for example, HTTP 400 Bad Request).

4.2 Error Response Format

The backend REST API shall return error information in a consistent JSON format for both invalid input and server-side errors.

The error response body shall contain at least the following fields:

- errorCode – a short, machine-readable string that identifies the error category.
- message – a human-readable explanation of the error in simple English.

Example of an error response:

```
{
"errorCode": "INVALID_AMOUNT",
"message": "Amount must be greater than 0."
}
```

The system shall use stable errorCode values for common error categories, such as:

- INVALID_AMOUNT – the amount is missing, non-numeric, or not strictly greater than 0.
- INSUFFICIENT_FUNDS – the requested withdrawal or exchange amount exceeds the source wallet balance.
- INVALID_WALLET_STATE – the operation is not allowed for the current wallet status.
- WALLET_NOT_FOUND – the referenced wallet identifier does not exist.
- UNSUPPORTED_CURRENCY – a currency outside the supported set is used.

The frontend may display simplified or localized messages to the user, but the backend shall always return the structured error response described above.

4.3 External and Server Errors

Errors may occur due to server-side problems or failures in external services, such as the external exchange-rate API.

Examples of such errors include:

- The external exchange-rate API is unavailable or times out.
- The external exchange-rate API returns data that cannot be interpreted as a valid rate.
- Internal server errors, such as unexpected exceptions or database failures.

For any error of this type:

- The backend API shall return a server error status code in the 5xx range (for example, HTTP 500 Internal Server Error or HTTP 502 Bad Gateway).
- No wallet balances or wallet states shall be changed as a result of the failed operation.
- If the request corresponds to a transaction (for example, an exchange that requires an external rate), the system shall record a failed transaction with an error code indicating an external or internal error (for example, EXTERNAL_SERVICE_UNAVAILABLE or INTERNAL_ERROR).

The system shall not silently invent or guess exchange rates when the external exchange-rate API fails. If a valid rate cannot be obtained from the external API or a non-expired cache entry, the exchange shall fail.

4.4 Sensitive Data Handling

The system deals only with simulated wallet balances and does not require any sensitive payment information.

For the purposes of this system, **sensitive payment information** includes, for example:

- Payment card numbers, expiry dates, and security codes.
- Bank account numbers and bank identifiers.
- Personal identification numbers associated with financial accounts.

The system shall:

- Not accept sensitive payment information as input in any API or user interface.
- Not store sensitive payment information in the database.
- Not log sensitive payment information in application logs or client-side storage.

Any logs produced by the system shall contain only technical information needed for debugging and auditing (such as wallet identifiers, transaction identifiers, error codes, and timestamps) and shall not contain sensitive payment information.

The system shall use stable errorCode values for common error categories, such as:

- INVALID_AMOUNT – the amount is missing, non-numeric, or not strictly greater than 0.
- INSUFFICIENT_FUNDS – the requested withdrawal or exchange amount exceeds the source wallet balance.
- INVALID_WALLET_STATE – the operation is not allowed for the current wallet status.
- WALLET_NOT_FOUND – the referenced wallet identifier does not exist.
- UNSUPPORTED_CURRENCY – a currency outside the supported set is used.

The frontend may display simplified or localized messages to the user, but the backend shall always return the structured error response described above.

4.3 External and Server Errors

Errors may occur due to server-side problems or failures in external services, such as the external exchange-rate API.

Examples of such errors include:

- The external exchange-rate API is unavailable or times out.
- The external exchange-rate API returns data that cannot be interpreted as a valid rate.
- Internal server errors, such as unexpected exceptions or database failures.

For any error of this type:

- The backend API shall return a server error status code in the 5xx range (for example, HTTP 500 Internal Server Error or HTTP 502 Bad Gateway).
- No wallet balances or wallet states shall be changed as a result of the failed operation.
- If the request corresponds to a transaction (for example, an exchange that requires an external rate), the system shall record a failed transaction with an error code indicating an external or internal error (for example, EXTERNAL_SERVICE_UNAVAILABLE or INTERNAL_ERROR).

The system shall not silently invent or guess exchange rates when the external exchange-rate API fails. If a valid rate cannot be obtained from the external API or a non-expired cache entry, the exchange shall fail.

4.4 Sensitive Data Handling

The system deals only with simulated wallet balances and does not require any sensitive payment information.

For the purposes of this system, **sensitive payment information** includes, for example:

- Payment card numbers, expiry dates, and security codes.
- Bank account numbers and bank identifiers.
- Personal identification numbers associated with financial accounts.

The system shall:

- Not accept sensitive payment information as input in any API or user interface.
- Not store sensitive payment information in the database.
- Not log sensitive payment information in application logs or client-side storage.

Any logs produced by the system shall contain only technical information needed for debugging and auditing (such as wallet identifiers, transaction identifiers, error codes, and timestamps) and shall not contain sensitive payment information.

5. Non-functional requirements

This section defines quality constraints that guide testing beyond functional correctness. They are scoped to an exam project (not production hardening) and are written to be measurable through the planned deliverables (Playwright, JMeter, SonarCloud, usability test design).

5.1 Performance (JMeter)

The API must remain responsive and stable under a small controlled workload in the reference environment (Docker).

- **Scope**: POST /api/wallets/{id}/deposit, POST /api/wallets/{id}/withdraw, POST /api/wallets/exchange, GET /api/wallets, GET /api/wallets/{id}/transactions.
- **Baseline load target**: during a steady load test, the system must maintain < 1% HTTP 5xx errors and a 95th percentile response time ≤ 500 ms.
- **Stress/spike observation**: during stress/spike tests, results are measured and reported (response time + error rate) to document the breaking point and recovery behavior.
- **Evidence**: results documented via JMeter report/screenshots.

5.2 Reliability & Robustness

The system must behave predictably on invalid input and must protect persisted state.

- Input validation: malformed or missing request fields must return 4xx responses with a JSON error payload following the system's error format.
- Business rule failures: domain rule violations must return a 4xx response and must not produce partial state updates.
- Transaction integrity: failed operations must not mutate wallet balances; successful operations must persist both balance changes and a corresponding transaction record.

5.3 Security

Security is addressed through preventive constraints and automated checks.

- No secrets in source control: tokens/keys (including Sonar tokens and API configuration) must be provided via environment variables / CI secrets and must not be committed to the repository.
- Static security signal: static analysis must report 0 vulnerabilities for the submitted version (as measured by SonarCloud).
- Sensitive data constraint: the system must not store or log sensitive payment data (e.g., card numbers, CVV).

5.4 Usability

A usability test must be designed but not executed.

- The deliverable must include scenarios and tasks covering core workflows (create wallet, deposit, withdraw, exchange, view transactions).
- The design must include both:
- Performance measures (e.g., time-on-task, error rate, task completion)
- Preference measures (e.g., perceived ease of use, clarity)

5.5 Frontend Quality

A minimal frontend must exist to support key user flows and automated UI testing.

- Core UI flows: create wallet, list wallets, deposit, withdraw, exchange, view transactions.
- E2E automation: Playwright tests must cover at least one complete happy path and one negative path through the UI.
- Lighthouse: Lighthouse scores must be measured and reported for the frontend, with target minimums:
  - Performance ≥ 80
  - Accessibility ≥ 90
  - Best Practices ≥ 80

6. Data Model Overview

This section gives a brief overview of the main data structures. It is not a full database schema, but describes the core entities and relationships needed for implementation and testing.

6.1 Wallet

A **wallet** represents a balance in one supported currency for the single logical user.

Each wallet record shall contain at least:

- **Id:** A unique identifier for the wallet.
- **Currency**: The wallet's currency. Must be one of: DKK, EUR, USD.
- **Balance**: The current balance of the wallet, stored as a numeric value with at least two decimal places of precision.
- **Status**: The current status of the wallet. Must be one of: active, frozen, closed.
- **createdAt**: Timestamp indicating when the wallet was created.
- **updatedAt**: Timestamp indicating when the wallet was last updated (for example, after a transaction or status change).

The system may maintain additional technical fields (such as internal versioning or audit information) if needed, if the behaviour defined in this document is preserved.

6.2 Transaction

A **transaction** records an operation that changes, or attempts to change, wallet balances.

Each transaction record shall contain at least:

- **id**: A unique identifier for the transaction.
- **Timestamp**: Timestamp indicating when the transaction was processed.
- **Type**: The transaction type. Must be one of: deposit, withdrawal, exchange.
- **sourceWalletId**: Identifier of the source wallet, if applicable:
  - Required for withdrawals and exchanges.
  - Optional or null for deposits.
- **targetWalletId**: Identifier of the target wallet, if applicable:
  - Required for exchanges.
  - Optional or null for deposits and withdrawals.
- **amount**: The input amount specified by the user, expressed in the source wallet's currency (or in the wallet's currency for deposits and withdrawals).
- **creditedAmount**: For exchanges, the amount credited to the target wallet after conversion and rounding, expressed in the target wallet's currency. For deposits and withdrawals, this may be equal to amount or omitted if redundant.
- **Status**: The transaction status. Must be one of: completed, failed.
- **errorCode**: or failed transactions, a short code describing the reason (for example, INVALID_AMOUNT, INSUFFICIENT_FUNDS, EXTERNAL_SERVICE_UNAVAILABLE). For completed transactions, this field may be null.

The system may store additional information, such as the exchange rate used for an exchange, for audit or debugging purposes.

6.3 Relationships

The main relationships between wallets and transactions are:

- A wallet can be referenced as the **sourceWalletId** in zero or more transactions.
- A wallet can be referenced as the **targetWalletId** in zero or more transactions.
- A single transaction may reference:
  - one source wallet (for withdrawals and exchanges),
  - one target wallet (for exchanges),
  - or neither (for some deposit representations if the wallet is implied).

There is no direct relationship between wallets representing different users, as the system models only a single logical user.

7. Out-of-Scope

The following features and concerns are explicitly out of scope for this system:

- **Real financial integrations**
  - No integration with real banks, payment providers, or card processors.
  - No handling of real money or actual financial accounts.
- **User management and security features**
  - No user registration, login, or authentication mechanisms.
  - No authorization or access control between multiple users.
  - No password management or account recovery features.
- **Additional currencies and instruments**
  - No currencies other than DKK, EUR, and USD.
  - No support for financial instruments such as stocks, cryptocurrencies, or loans.
- **Localization and internationalization**
  - No requirement to support multiple languages or locale-specific formats beyond basic numeric formatting used in the reference environment.
- **Production-grade hardening and deployment**
  - No requirements for production-scale scalability, high availability, or failover.
  - No detailed logging, monitoring, or alerting beyond what is needed for development and testing.
- **Advanced usability and accessibility features**
  - No formal accessibility compliance (such as WCAG) is required, beyond basic keyboard and mouse usability for the target users.

These exclusions are intended to keep the system focused and manageable for the intended educational and exam context.