|  |  |
| --- | --- |
| Jira epic title | Implement Account Balance Retrieval and Transfer Functionality |
| Jira Ticket | ACC-001 Implement Account Balance Retrieval  ACC-002 Implement Transfer Functionality |
| Target Development complete date | Acc-001 : 04/08/2024 Acc-002 : 05/08/2024 |
| Target Release date | Acc-001 : 07/08/2024 Acc-002 : 07/08/2024 |
| Backend Developer | Fai Man Kwong |
| Web Developer | TBC |
| Project Manager | TBC |
| Business Analysis | TBC |

Background:  
The account-manager service for acmebank provides functionalities to retrieve account balances and transfer funds between accounts. This document outlines the architecture, setup instructions, API endpoints, service layer, validation logic, testing, and project structure.

**Repository Structure**

|  |
| --- |
| Plain Text account-manager/ ├── src/ │ ├── main/ │ │ ├── java/com/acmebank/accountmanager/ │ │ │ ├── controller/ # REST Controllers │ │ │ ├── exception/ # Custom Exceptions │ │ │ ├── model/ # Data Models │ │ │ ├── repository/ # JPA Repositories │ │ │ ├── service/ # Service Layer │ │ │ ├── validator/ # Validation Logic │ │ │ └── AccountManagerApplication.java # Main Spring Boot Application │ │ └── resources/ │ │ ├── application.properties # Spring Boot Configuration │ │ └── data.sql # Initial Data │ │ └── scheme.sql # create scheme │ └── test/ │ └── java/com/acmebank/accountmanager/ │ └── validator/ # Spock Tests │ └── service/ # Spock Tests ├── build.gradle # Gradle Build File └── README.md # Project Documentation |

**Database Schema**

The database uses an H2 in-memory database with the following schema for the account table:

sql

Copy

|  |
| --- |
| SQL CREATE TABLE IF NOT EXISTS account (  id BIGINT PRIMARY KEY,  balance DECIMAL(19, 2),  currency VARCHAR(3) DEFAULT 'HKD',  created\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  modified\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP ); |

**Prepopulated Accounts**

* **Account 1:** ID: 12345678, Balance: 1,000,000 HKD
* **Account 2:** ID: 88888888, Balance: 1,000,000 HKD

**API Endpoints**

**Get Account Balance**

**Request:**

* **Method:** GET
* **URL:** /api/v1/accounts/balance
* Request Param :
* Long accountId ;
* **Sample** **Response:**

|  |
| --- |
| JSON {  "accountId": 12345678,  "balance": 1000000.00,  "currency": "HKD" } |

**Transfer Funds**

**Request:**

* **Method:** POST
* **URL:** /api/v1/accounts/transfer
* **Body:**

|  |
| --- |
| JSON {  "fromAccountId": 12345678,  "toAccountId": 88888888,  "amount": 5000.00 } |

* **Response:**

|  |
| --- |
| JSON {  "status": "success",  "message": "Transfer completed successfully" } |

Exception :

|  |  |
| --- | --- |
| AccountNotExistException | When the Input account ID does not exist |
| InsufficientBalanceException | When the balance is insufficient to complete the transfer |
| TransferFailCheckingException | Validating the input parameters for the transfer operation |

TransferFailCheckingException Detail:

|  |  |  |
| --- | --- | --- |
| **Validation Check** | **Condition** | **Exception Message** |
| Invalid fromAccountId | fromAccountId is null or less than 1 | Invalid fromAccountId: + fromAccountId |
| Invalid toAccountId | toAccountId is null or less than 1 | Invalid toAccountId: + toAccountId |
| fromAccountId and toAccountId same | fromAccountId is equal to toAccountId | Sender and receiver cannot be the same |

**Service Layer**

**AccountService**

* **Methods:**
* BigDecimal getBalance(Long accountId)
* void transferFunds(Long fromAccountId, Long toAccountId, BigDecimal amount)

**AccountRepository**

* **Extends:** JpaRepository<Account, Long>
* **Custom Queries:** None required for this implementation.

**Validation**

**AccountTransferValidation**

* Ensures fromAccountId and toAccountId are valid and not the same.
* Ensures sufficient balance in the source account before transferring.

|  |
| --- |
| Java @Override public void validate(ValidateAccount validateAccount) throws AccountNotExistException {  Long fromAccountId = validateAccount.getCurrentAccountId();  Long toAccountId = validateAccount.getToAccountId();  if (fromAccountId == null || fromAccountId < 1) {  throw new TransferFailCheckingException("Invalid fromAccountId: " + fromAccountId);  }  if (toAccountId == null || toAccountId < 1) {  throw new TransferFailCheckingException("Invalid toAccountId: " + toAccountId);  }  if (fromAccountId.equals(toAccountId)) {  throw new TransferFailCheckingException("Sender and receiver cannot be the same");  } } |

**Technical Summary**

**Technical Summary of AccountValidatorStrategySpec**

The AccountValidatorStrategySpec class is a Spock test specification designed to validate different aspects of account operations within a banking system. It focuses on three main validation strategies:

1. **Account Existence Validation**
2. **Account Balance Validation**
3. **Account Transfer Validation**

These validations are performed using the following classes:

* AccountExistenceValidator
* AccountBalanceValidator
* AccountTransferValidator

**Key Components and Test Cases**

**Account Existence Validation**

* **Purpose:** Ensure that the account exists and the fromAccountId is valid.

**Account Balance Validation**

* **Purpose:** Ensure that the transfer amount is valid.

**Account Transfer Validation**

* **Purpose:** Ensure that transfer details are valid, including fromAccountId, toAccountId, and that the sender and receiver are not the same.

The AccountValidatorStrategySpec class comprehensively tests the account validation logic, ensuring that:

* Accounts must exist before performing operations.
* Transfer amounts must be valid and greater than zero.
* Transfer operations must have valid sender and receiver details, ensuring no self-transfers.

By covering these scenarios, the test class ensures that the account validation mechanisms are robust, preventing invalid operations and ensuring data integrity within the banking system.

**AccountServiceImpl Test Cases**

This section provides an overview of Spock test cases for the AccountServiceImpl class, focusing on validating the getBalance and transfer methods. These tests ensure the correct behavior of the account service under various conditions, including successful operations and failure scenarios.

**Technical Aspects**

* **Spock Framework:** Utilizes Spock's BDD-style testing and data-driven testing capabilities.
* **Unroll Annotation:** Used to run multiple data-driven tests within a single test method, improving clarity and reducing redundancy.
* **Exception Handling:** Tests focus on ensuring appropriate exceptions are thrown for invalid conditions, enhancing the robustness of the validation logic.