Banking Transaction Simulation Platform using Java

Project Documentation

A Multi-User, Multi-Account Banking System Implementation

Batch - 3

December 5, 2025

Abstract

This document details the design, implementation, and analysis of a Java-Based Simple Banking Transaction Simulator. The system simulates core banking functionalities including user registration, account management (Savings and Current), deposits, withdrawals, inter-account transfers, transaction logging via JDBC, and automated email alerts for low balances. The system employs a user-centric model, allowing one user to possess multiple accounts. It features a dynamic command-line interface (CLI) tailored to the user's account status and provides both text-based and PDF-formatted account reports. The project adheres to core Java principles without external frameworks, focusing on robust exception handling, data persistence, and user experience.

Contents

1 Introduction

1.1 Project Statement

This project aims to create a basic banking transaction simulator using core Java features. It incorporates exception handling for error management, JDBC for transaction logging, and integrates with text files and email APIs for balance alerts. The system processes deposits, withdrawals, and transfers while maintaining account balances, providing an educational tool to understand banking operations and financial flows.

1.2 Objectives

- Implement accurate simulation of banking transactions with real-time balance updates.
- Robustly handle errors for invalid operations (e.g., overdrafts).
- Generate basic reports on transaction history and account summaries.
- Provide notifications based on balance thresholds.
- Demonstrate understanding of Java fundamentals, JDBC, and email integration.

2 System Architecture and Design

2.1 High-Level Architecture

The system follows a layered architecture, separating presentation, business logic, and data access concerns.

Presentation Layer
`Main.java` (CLI Interface)
Business Logic Layer
`UserService`, `AccountService`, `TransactionService`, `AlertService`, `ReportGenerator`, `PDFReportGenerator`
Data Access Layer
`TransactionLogger`, `DatabaseInitializer`
Persistence Layer
MySQL Database (`users`, `accounts`, `transactions`)

2.2 Key Components

- **User Management**: Handles user registration and authentication.
- **Account Management**: Manages user accounts (Savings/Current), linking them to a user ID.
- **Transaction Processing**: Handles deposits, withdrawals, and transfers with validation.
- **Data Persistence**: Uses JDBC to interact with a MySQL database.
- **Reporting**: Generates text and PDF reports based on account activity.
- **Alerting**: Sends email notifications based on account balance thresholds.

3 Modules Implemented

3.1 Module 1: Account Management Engine

- **Functionality**: User registration, login, account creation (Savings/Current), account selection.
- **Storage**: MySQL-persistent via `users` and `accounts` tables.
- **Features**:
  - 4-digit numeric account IDs (e.g., 1000, 1001, ...) starting from 1000.
  - Email + password authentication for users. Email must be unique.
  - Each user can have up to one Savings and one Current account.
  - Dynamic user menu based on account ownership (e.g., "Create Savings" if only Current exists).
  - Admin access (`admin@bank.com` / `admin123`).
  - Custom exceptions: `DuplicateAccountException`, `AccountNotFoundException`.

3.2 Module 2: Transaction Processing System

- **Operations**: Deposit, Withdraw, Transfer (between accounts).
- **Validation**:
  - Positive amounts only.
  - Sufficient funds check for withdrawals/transfers.
  - Valid account IDs (must exist).
  - Prevents transfers to self (optional enhancement).
- **Exception Handling**:
  - `InsufficientFundsException`
  - `AccountNotFoundException`
- **Logging**: Transactions are logged in the `transactions` table via `TransactionLogger`.

3.3 Module 3: Reporting and Text File Integration Hub

- **Reports**: Per-account text files (`reports/account_1000.txt`) and PDF files (`reports/account_100`
- **Content**:
  - Account summary (ID, type, owner, balance, threshold).
  - Transaction history fetched from the database.
  - Timestamped, human-readable format.
- **PDF Generation**: Uses iText 7 library to create professional-looking PDF statements.

3.4 Module 4: Balance Alert Tracker

- **Rule**: Alert if balance < threshold (customizable per account, default $100.00).
- **Notification**:
  - Email sent to the account owner's registered email address.
  - Uses Gmail SMTP with App Password for authentication.
- **Customization**: Users can update the threshold value for their accounts.

4 Implementation Details

4.1 Database Schema

4.1.1 Users Table

```
CREATE TABLE users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL,        -- Username is NOT unique
    email VARCHAR(100) NOT NULL UNIQUE,   -- Email must be unique
    password VARCHAR(255) NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Listing 1: SQL: users table (username is NOT unique)

4.1.2 Accounts Table

```
CREATE TABLE accounts (
    account_id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT NOT NULL,
    account_type ENUM('SAVINGS', 'CURRENT') NOT NULL,
    balance DECIMAL(15,2) NOT NULL DEFAULT 0.00,
    threshold DECIMAL(15,2) NOT NULL DEFAULT 100.00,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
) AUTO_INCREMENT = 1000;
```

Listing 2: SQL: accounts table

4.1.3 Transactions Table

```
CREATE TABLE transactions (
    id VARCHAR(36) PRIMARY KEY, -- UUID
    account_id INT NOT NULL,    -- Links to the affected account
    amount DECIMAL(15,2) NOT NULL,
    type VARCHAR(20) NOT NULL,  -- 'DEPOSIT', 'WITHDRAWAL', '
   TRANSFER_OUT', 'TRANSFER_IN'
    status VARCHAR(10) NOT NULL,-- 'SUCCESS', 'FAILED'
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES accounts(account_id) ON DELETE
   CASCADE
);
```

Listing 3: SQL: transactions table

4.2 Key Classes and Logic

4.2.1 Account.java

Listing 4: Account Model

```
public class Account {
    private int accountId;
```

```
    private int userId; // Links to User
    private String ownerName; // From User table
    private String email;     // From User table
    private String password;  // From User table
    private AccountType accountType; // SAVINGS or CURRENT
    private BigDecimal balance;
    private BigDecimal threshold;

    // Constructor, Getters, Setters, toString()
    // ...
}
```

4.2.2 Transaction Logging

Listing 5: Transaction Logging

```
// In TransactionLogger.java
public void logTransaction(int accountId, BigDecimal amount, String type, S
    String sql = "INSERT INTO transactions (id, account_id, amount, type, s
    // Execute prepared statement
}
// For transfers, two entries are made (one OUT, one IN if successful)
```

4.2.3 Dynamic Menu Logic (Excerpt)

Listing 6: Dynamic User Menu

```
// In Main.java -> showUserMenu()
try {
    List<Account> userAccounts = accountService.getAccountsByUserId(loggedI
    int numAccounts = userAccounts.size();
    // ... logic to determine hasSavings, hasCurrent ...
    if (numAccounts == 2) { // Has both
        // Show: Select Account, Logout
    } else if (numAccounts == 1) { // Has one
        if (hasSavings) {
            // Show: Create Current, Select Account, Logout
        } else { // hasCurrent
            // Show: Create Savings, Select Account, Logout
        }
    } else { // Has none
        // Show: Create Account, Logout
    }
} catch (SQLException e) { /* Handle error */ }
```

5 Sample Workflow

5.1 End-to-End Test Scenario

1. **Register User**: Alice (`alice@example.com`, password: `pass123`).
2. **Login**: Alice logs in successfully.
3. **Create Savings Account**: Alice creates a Savings account with $1000 initial balance.
4. **Deposit**: Alice deposits $400 into her Savings account (Balance: $1400).
5. **Create Current Account**: Alice creates a Current account.
6. **Transfer**: Alice transfers $200 from Savings to Current.
7. **Generate Report**: Alice selects her Savings account and generates a PDF report.
8. **Admin View**: Admin logs in and views all users, accounts, and transactions.

5.2 Console Output Snippet

```
=== WELCOME TO BANKING SIMULATOR ===
1. Register
2. Login
0. Exit
Choose: 1
Username: alice
Email: alice@example.com
Password: pass123
    Registered! Your username: alice

Choose: 2
Email: alice@example.com
Password: pass123
    Logged in as: alice

=== HI, alice ===
1. Create Account (Savings/Current)
2. Logout
Choose: 1
Account Type (SAVINGS/CURRENT): SAVINGS
Initial balance: 1000
    SAVINGS account created! ID: 1000

=== HI, alice ===
1. Create Current Account
2. Select Account
3. Logout
Choose: 1
Account Type (SAVINGS/CURRENT): CURRENT
Initial balance: 500
    CURRENT account created! ID: 1001
```

6 Security and Best Practices

- **SQL Injection Prevention**: All database queries use `PreparedStatement` with parameterized inputs.
- **Configuration Management**: Database and email credentials are stored in `config.properties`, which should be added to `.gitignore`.
- **Exception Handling**: Comprehensive use of custom and standard exceptions for robust error management.
- **Logging**: `java.util.logging` is used for application-level logging.
- **Email Authentication**: Uses Gmail App Passwords for secure SMTP authentication.

7 Conclusion

This project successfully delivers a robust, educational banking simulator using core Java technologies. It fulfills all stated outcomes with added sophistication through a multi-user, multi-account model, dynamic user interface, and professional reporting. The system demonstrates a solid understanding of Java fundamentals, JDBC, exception handling, and user experience design, making it suitable for academic evaluation and portfolio inclusion.

Appendix A: Tools and Technologies

- **Language**: Java 21 (JDK 21)
- **Database**: MySQL 8.0+
- **Libraries**: `mysql-connector-java:8.0.33`, `javax.mail:1.6.2`, `itextpdf:7.2.5`
- **IDE**: IntelliJ IDEA Community Edition
- **Build**: Maven (POM provided)

Appendix B: GitHub Repository Structure

```
BankingSimulator/
 config.properties
 pom.xml
 src/
    main/
       java/
           com/
               bank/
                   Main.java
                   ConfigLoader.java
                   exception/
                   model/
```

```
                  service/
                  util/
   resources/
       schema.sql
reports/           # Generated text reports
reports_pdf/       # Generated PDF reports (if using DigiBankProject structure)
lib/               # (If not using Maven)
```