Basic Banking Backend Application Requirements Document

Overview

This document outlines the requirements for developing a basic banking backend application using Spring Boot and MySQL. The application will provide basic banking functionalities such as account management, transaction processing, and balance inquiries.

Functional Requirements

1. User Management
   - Create a new user account.
   - Authenticate a user.
   - Retrieve user details.

2. Account Management
   - Create a new bank account.
   - Retrieve account details.
   - Close a bank account.

3. Transaction Management
   - Deposit money into an account.
   - Withdraw money from an account.
   - Transfer money between accounts.
   - Retrieve transaction history for an account.

4. Balance Inquiry
   - Retrieve the current balance of an account.

Basic Banking Backend Application Requirements Document

Non-Functional Requirements

1. Security

   - Use JWT for securing APIs.

   - Encrypt sensitive data.

2. Performance

   - Ensure the application can handle a high number of concurrent requests.

3. Scalability

   - Design the application to support horizontal scaling.

4. Maintainability

   - Write clean, modular, and well-documented code.

5. Database

   - Use MySQL for persistent storage.

   - Ensure the database schema supports indexing for efficient queries.

API Endpoints

User Management

1. Create User

   - Endpoint: POST /api/users

   - Request Body:

```
{
  "username": "string",
  "password": "string",
  "email": "string"
}
```

- Response: 201 Created

2. Authenticate User

   - Endpoint: POST /api/auth/login

   - Request Body:

```
{
  "username": "string",
  "password": "string"
}
```

- Response: 200 OK (JWT Token)

3. Get User Details

   - Endpoint: GET /api/users/{userId}

   - Response: 200 OK

```
{
```

```
  "id": "long",
  "username": "string",
  "email": "string"
}
```

Account Management

1. Create Account

   - Endpoint: POST /api/accounts

   - Request Body:

```
{
  "userId": "long",
  "initialDeposit": "double"
}
```

- Response: 201 Created

2. Get Account Details

   - Endpoint: GET /api/accounts/{accountId}

   - Response: 200 OK

```
{
  "id": "long",
  "userId": "long",
```

```
  "balance": "double",
  "status": "string"
}
```

3. Close Account
   - Endpoint: DELETE /api/accounts/{accountId}
   - Response: 200 OK

Transaction Management

1. Deposit
   - Endpoint: POST /api/accounts/{accountId}/deposit
   - Request Body:

```
{
  "amount": "double"
}
```

- Response: 200 OK

2. Withdraw
   - Endpoint: POST /api/accounts/{accountId}/withdraw
   - Request Body:

```
{
```

```
  "amount": "double"
}
```

- Response: 200 OK

3. Transfer

  - Endpoint: POST /api/accounts/transfer

  - Request Body:

```
{
  "fromAccountId": "long",
  "toAccountId": "long",
  "amount": "double"
}
```

- Response: 200 OK

4. Transaction History

  - Endpoint: GET /api/accounts/{accountId}/transactions

  - Response: 200 OK

```
[
  {
    "id": "long",
```

```
    "accountId": "long",
    "type": "string",
    "amount": "double",
    "timestamp": "datetime"
  }
]
```

Balance Inquiry

1. Get Balance
   - Endpoint: GET /api/accounts/{accountId}/balance
   - Response: 200 OK

```
{
  "accountId": "long",
  "balance": "double"
}
```

Database Schema

```
Users Table
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
```

```
    email VARCHAR(100) NOT NULL UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

```
Accounts Table
CREATE TABLE accounts (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    balance DOUBLE DEFAULT 0,
    status VARCHAR(20) DEFAULT 'ACTIVE',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

```
Transactions Table
CREATE TABLE transactions (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    account_id BIGINT NOT NULL,
    type VARCHAR(20) NOT NULL,
    amount DOUBLE NOT NULL,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES accounts(id)
```

```
);
```

Entity Relationships

- User to Account: One-to-Many (One user can have multiple accounts)
- Account to Transaction: One-to-Many (One account can have multiple transactions)

API Security

- Use JWT tokens for securing endpoints.
- All sensitive data such as passwords should be hashed using a secure algorithm (e.g., BCrypt).
- Use HTTPS for secure communication.

Conclusion

This document outlines the basic structure and requirements for a banking backend application using Spring Boot and MySQL. The application will provide essential banking operations with secure and efficient handling of user and account data.