Bank Application Project - ING

Contents of the project

- Instructions on setting up project (this)
- Architecture diagram
- Class diagram
- Source code
- SQL file (for setting up database)

Setting up the database

- The SQL file: bankApp.sql is attached, that contains the database setup details, such as creation of database, and tables.
- Login to database terminal and run this .sql file.
- If database used is Postgres, it can be run by the command: \i *bankApp.sql*
    - Here, give the name of sql file with full path.
- It will create the following:
    - Database: ing
    - Table: account_info
        - This table holds the information of all customers, such as account number, customer name and account balance.
    - Table: acc_id_generator
        - This table is used to generate new account ID for a new customer.
    - Sequence: bank_seq
        - This sequence is for the account_info table.
    - Index: ing_index
        - Index has been applied to account_number field of account_info table, as this is the search criteria.

Setting up properties of database

- Open the file: src/main/resources/application.properties
- Change the database related settings as per the database you want to use.
    - Hostname
    - Driver name
    - JDBC URL
    - Password

Setting up project

- Unzip the shared folder and navigate to the project folder.
- If you wish to change the port, open the file: src/main/resources/application.properties, and change the value of server.port.
- Currently the port is set to 8085.
- Run - *mvn clean install*
- Go to the target folder and run – *java -jar bank-application-0.0.1.jar*
- This will start the spring boot application on the mentioned port (8085 here).

Running the project

- The spring boot application will expose the following APIs:
  - API: /addcustomer
    - URL: http://localhost:8085/addcustomer
    - Method: POST
    - Content-type: JSON(application/json)
    - Body: { "name": "<customer name>", "balance": "<opening balance>"}
      - Eg.: { "name": "John", "balance": "1000"}
    - Response:
      - Success case: 200 OK – New customer: <customer_name> added, with account number: <account_number>
      - Failure case: 500 Internal Server Error – Error message
  - API: /checkbalance
    - URL: http://localhost:8085/checkbalance
    - Method: POST
    - Content-type: JSON(application/json)
    - Body: <6-digit account number>
      - Eg.: 000007
    - Response:
      - Success case: 200 OK – Balance amount
      - Failure case: 500 Internal Server Error – Error message
  - API: /dotransaction
    - URL: http//localhost:8085/dotransaction
    - Method: POST
    - Content-type: JSON(application/json)
    - Body: {"accountID" : "<6-digit account number>", "amount" : "<transaction amount>", "type" : "<type of transaction>"}
      - Eg.: {"accountID" : "7878787", "amount" : "1000", "type" : "DEPOSIT"}
      - Valid keywords for transactions: DEPOSIT, WITHDRAWAL
    - Response:
      - Success case: 200 OK – Balance amount after transaction
      - Failure case: 500 Internal Server Error – Error message
      - Withdrawal amount is greater than available balance case: 200 OK - Not enough balance to withdraw <available_balance>

Technology stack

- Keywords: Spring 4.0, Spring boot, REST APIs, Java, JDBC Templates, Transaction Templates, Postgres database, Mockito, Spring runner, Indexing

- Sprint boot is used for running the application.
- The service layer has been written in Java.
- JDBC template has been used for maintaining concurrency and for database transactions.
- All the classes have been covered with Junit test cases that are written using Spring runner and Mockito. Code coverage is more than 90%.

- Postgres database has been used. However it can be replaced by any other database, as the settings are configurable.
- Indexing has been used for faster search.
- Transaction template has been used for committing and rollback in case of failure of transaction.
- Spring 4.0 annotations have been used for dependency injection.

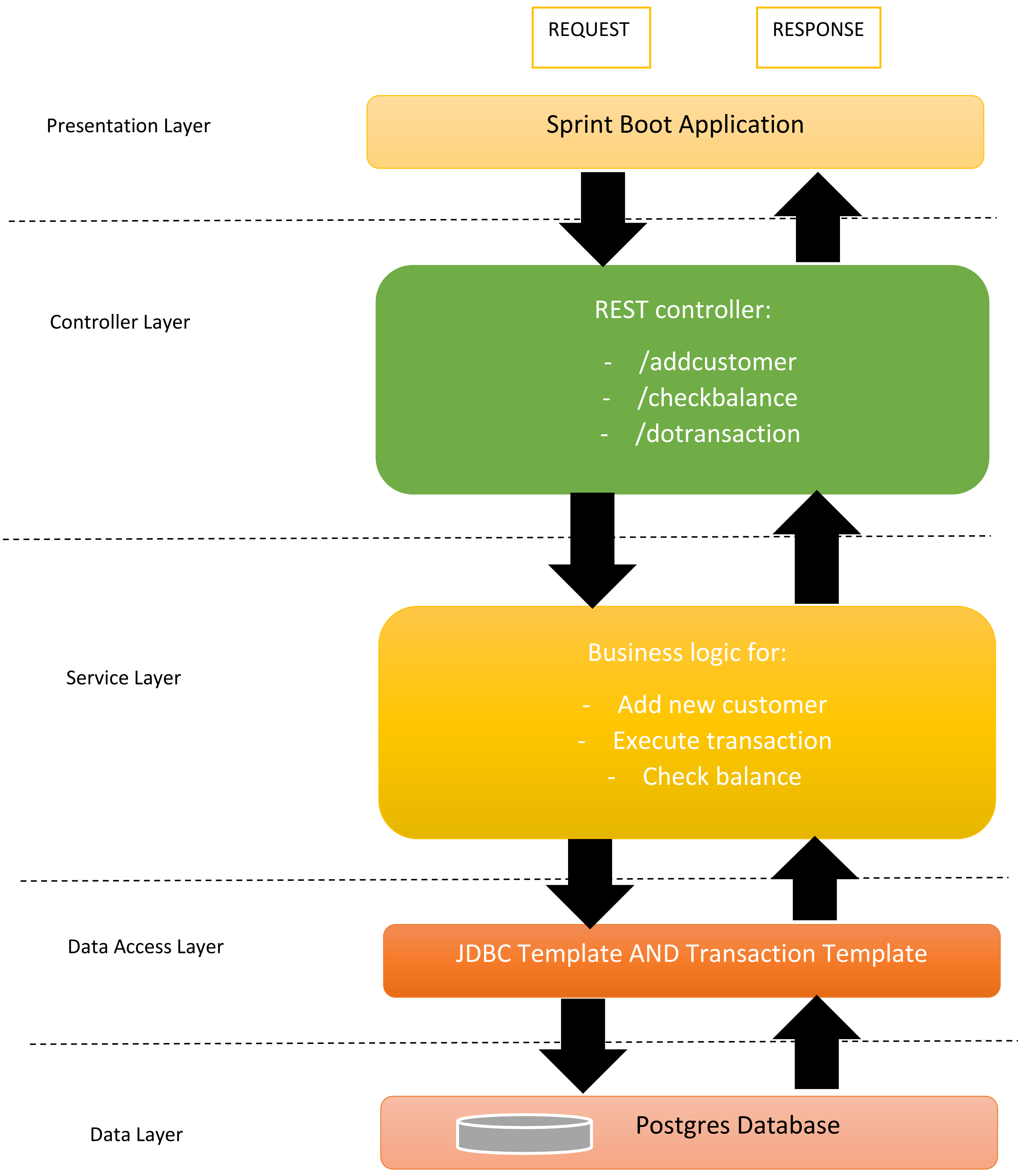
Description

- This project is built on MVC based architecture.
- It consists of 5 layers:
    - Presentation layer – runs the Spring boot application on the system, on the port 8085.
    - Controller layer - exposes the REST APIs to the user. These APIs can be consumed in front-end code, as well as other Java applications. It passes the user request to the service layer, and returns the response to user, with a suitable HTTP status code.
    - Service layer – contains the business logic for executing bank operations. The business logic lies in this layer. It takes the request from controller, process it, passes it to the data access layer, finally returns the response from data access layer to the controller layer.
    - Data Access layer – takes care of interactions with the data layer. Takes the request from service layer, executes it and sends the response back. JDBC templates and Transaction templates have been used here. Transaction templates take care of roll back of transaction, in case some exception occurs.
    - Data layer – stores the data. Postgres has been used for storing all the data.

Assumptions

- The account number is a 6-digit numeric value.
- A table called acc_id_generator is used for getting the next available value for account ID.
- It is then made into a 6-digit string, by introducing padding of 0 to the left of the string. Hence 1 becomes 000001.
- For /checkbalance, if account ID it entered incorrectly, an error will be displayed.
- TransactionType enum class has been defined for two kinds of transactions – DEPOSIT and WITHDRAWAL. It can be extended to other kinds of transactions as well.

Architecture Diagram


REQUEST
RESPONSE
Presentation Layer
Sprint Boot Application
Controller Layer
REST controller:
- /addcustomer
- /checkbalance
- /dotransaction
Service Layer
Business logic for:
- Add new customer
- Execute transaction
- Check balance
Data Access Layer
JDBC Template AND Transaction Template
Data Layer
Postgres Database