Program Practice – Design and Code task

Explanation Document

By

Krishna Kumar Gopal

Contents

[Tools and software used 3](#_Toc96422899)

[Functional Description: 3](#_Toc96422900)

[List of Restful Services 3](#_Toc96422901)

[Functional Description 5](#_Toc96422902)

[Architecture Diagram 7](#_Toc96422903)

[Code Architecture 7](#_Toc96422904)

[DB Schema Screenshots 8](#_Toc96422905)

[Best Practice Followed 9](#_Toc96422906)

[@Transactional 9](#_Toc96422907)

[Scheduler 9](#_Toc96422908)

[Validations 10](#_Toc96422909)

[Idempotency 10](#_Toc96422910)

[Used BigDecimal for Currency attributes 11](#_Toc96422911)

[Stored time in UTC format into the database 11](#_Toc96422912)

[Used complex database query to retrieve monthly status data from the database rather than filtering in java code. 11](#_Toc96422913)

[Used Java8 features 11](#_Toc96422914)

[Asynchronous 12](#_Toc96422915)

[Used Mockito and Junit for Unit testing: 13](#_Toc96422916)

[Built the code using Azure Devops Pipeline from GIT Repository: 13](#_Toc96422917)

[How would you handle account closures? 14](#_Toc96422918)

[Suggestions 15](#_Toc96422919)

[Event Driven Mechanism 15](#_Toc96422920)

[Using AWS Cloud Services -Serverless Architecture 15](#_Toc96422921)

[Using Microservices and Message Queues 16](#_Toc96422922)

[Improvements that can be done in existing code 17](#_Toc96422923)

# Tools and software used

I have used the below stack for the given assignment

|  |  |
| --- | --- |
| Web Framework | Spring Boot |
| ORM | Hibernate |
| Database | PostgreSQL |
| Unit Testing | Junit and Mockito |
| Build Tool | Maven |
| Webservice Testing Tool | postman |
| Programming Language | Java 11 |
| Code Repository | Github |
| CI/CD Pipeline | Azure Devops |
| GIT Code URL | https://github.com/krishnakumar9113/ProgramPractice |

## Functional Description:

### List of Restful Services

|  |  |
| --- | --- |
| Rest Service URL | Http Method |
| /account/process-account-opening | POST |
| Description  User can create a new account using this service  Body:   {         "bsb":"123456"   }  Sample Response:  {      "identification": 3802,      "bsb": 123456  } | |
| /transactions/create-transactions/{Identification} | POST |
| Description  User can create a new transaction for an existing account using this service  Body:   {         "txnAmount":100.00,         "txnType":"CRDT",         "txnRemarks":"Expense "   }  txnType can be “CRDT” /”DBIT” only.  Request Headers:  Idempotency-Key : <Time based UUID > eg: 959f9c81-c8d0-4a90-8af5-216581450610  This header needs to be unique for each transaction  Sample Response:  “Transaction created successfully”  Http Status: 201 CREATED | |
| /interest/process-account-end-of-day-balances | PUT |
| Description:  This service will calculate daily interest amount for each account based on their current balance, and credit the interest to their account. This happens only on first hit per day.  The interest amount calculation logic can be triggered only once per day, for other successive hits the interest calculation logic will not be executed, only the balance amount details will be listed.  The service response will contain all account details and their balance amount.  The balance amount shown also includes the current days interest.  Response will contain a list of account details like account balance with the Current date.  {      "balanceDate": "2022-02-19",      "AccountDetails": [          {              "bsb": 123456,              "identification": 2152,              "balance": 1900.52          },          {              "bsb": 123456,              "identification": 2452,              "balance": 200.03          },          {              "bsb": 123456,              "identification": 3802,              "balance": 100.01          }      ]  } | |
| /interest/calculate-monthly-interest/{month}/{Year} | PUT |
| Description:  This service will sum up interest amounts for the given month and year which was calculated for each day with respect to each account.  Response:  [      {          "identification": 2152,          "calculatedamt": 0.52      },      {          "identification": 3802,          "calculatedamt": 0.01      },      {          "identification": 2452,          "calculatedamt": 0.03      }  ]  Where identification is the account identification number | |

### Functional Description

1. User/Tester can create an Account using “/process-account-opening”
2. User/Tester can create transactions using “/transactions/create-transactions/{Identification}” for a specific account. They can trigger both Credit and debit transactions.
3. User/Tester will trigger “/interest/process-account-end-of-day-balances” service at the end of each day [Note: This service calculates interest amount only for the first trigger and respond with the balance amount, subsequent trigger will not calculate interest]

Interest is calculated based on Simple Interest formula

Simple Interest per day = (Balance Amount\* Percentage of interest)

Number of days in current year

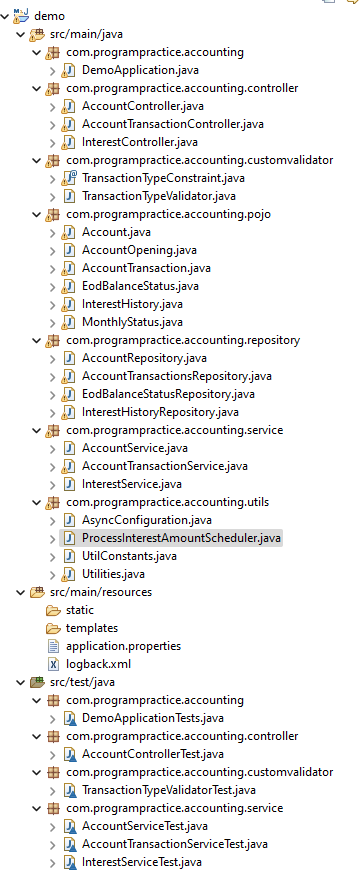
Assumptions: Percentage of interest= 5% =>0.05

The interest calculated will be credited to the account balance and an entry will be made in the transaction table.

1. User/Tester can check the total interest amount credited for a particular amount in a given month/year by retrieving the data from “Interest History“ Table using the “/interest/calculate-monthly-interest/02/2022” service.

# Architecture Diagram

## Code Architecture



Contain cron job Scheduler

Contain configuration for Thread executor used in Async operation

Junit Test cases

Contains Utility Functions and Constants used in the project

Service classes for implementing business logic for restful service

Repository classes for interacting with the databases.

Bean classes which define the properties used for database schemas, Restful apis request -body and response json structure

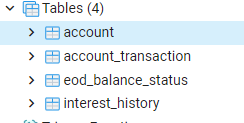
Validator classes for custom validation of properties in beans

Controller classes – to define the definition of RestFul Apis

Main java Class to Start the spring boot application

## DB Schema Screenshots

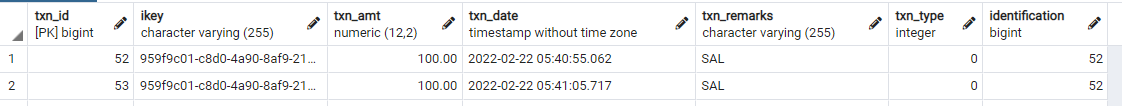
List of tables



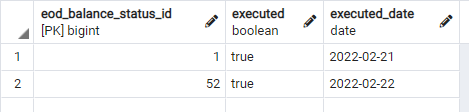
Account Table:



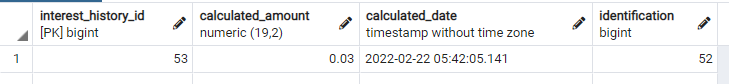
Account Transactions Table:



EOB Balance Status



Interest History Table

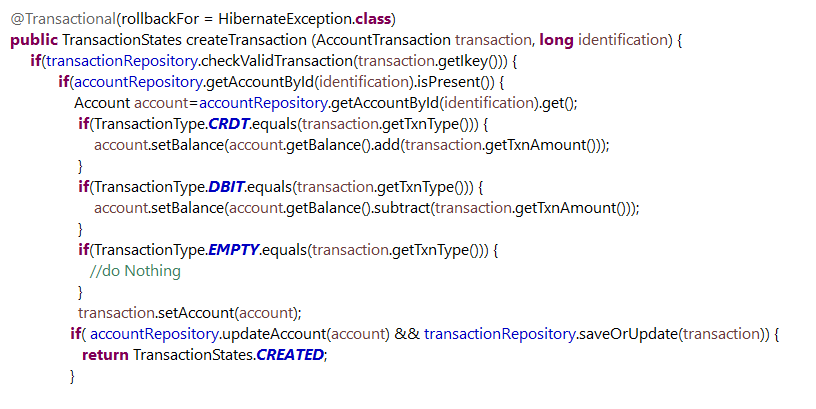


# Best Practice Followed

## @Transactional

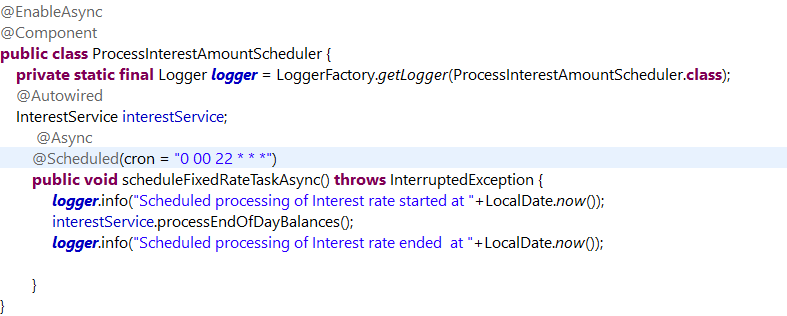
Used @Transactional annotation to process updation of multiple tables in a single transaction

Eg: While creating a record in Account transaction table. we will check for existing Idempotency-Key from “Account transaction” table and check for the account details from “account” table. Then based on the transaction type , we update the account balance and insert the transaction details in their respective tables.



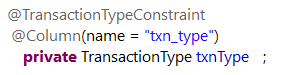
## Scheduler

Created a Cron job scheduler to execute the code for calculating the interest amount at the end of each day at 10 00 pm and credit the interest amount into each corresponding account.



If the interest amount is already calculated and credited to the account for that particular day, this job will run but will skip the interest calculation and updation logics.

## Validations

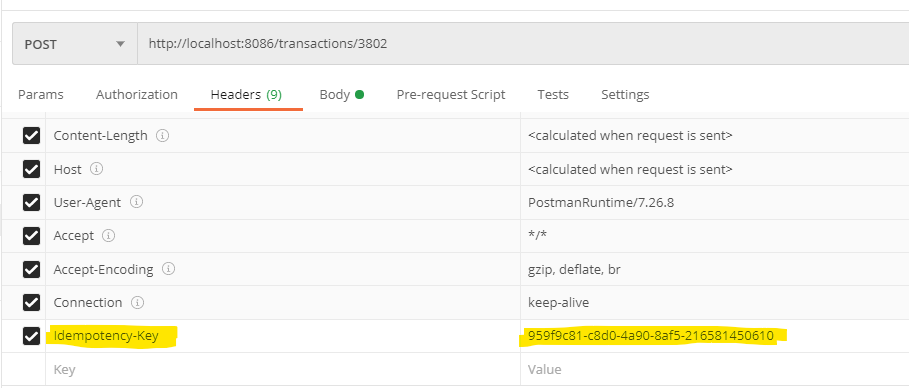


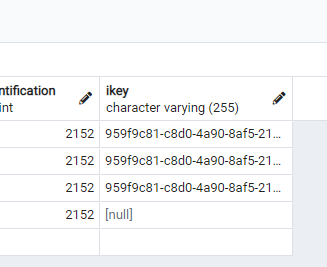
Used **custom validators** to validate Enums of transaction types



## Idempotency

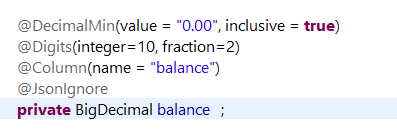
Time based UUID Key needs to be sent as request header for the “/transactions/create-transactions/{Identification}” service. If the transaction is successful, the key will be inserted along with the transaction record. So that a duplicate transaction can be avoided, if the user tries to repeat the transaction with same key.





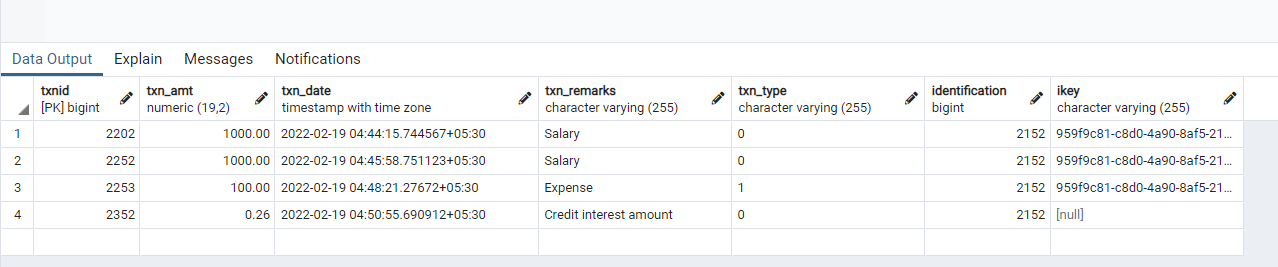
## Used BigDecimal for Currency attributes

Used BigDecimal Datatype for storing financial data like balance amount and interest amount.



## Stored time in UTC format into the database

Time stamp is stored in UTC format in the database.



## Used complex database query to retrieve monthly status data from the database rather than filtering in java code.

String jpql = "select e.account.identification as identification ,sum(e.calculatedamt) as calculatedamt from InterestHistory e where date\_part('month',e.calculatedDate) = :month and date\_part('year',e.calculatedDate) = :year GROUP BY e.account.identification";

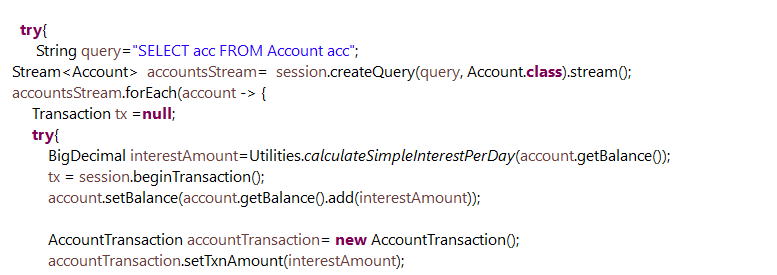
Query query= session.createQuery(jpql).setParameter(":month",month).setParameter(":year",year)

By this way, the computation happens in database itself and send only the required data to the spring boot application thus reducing the latency.

## 

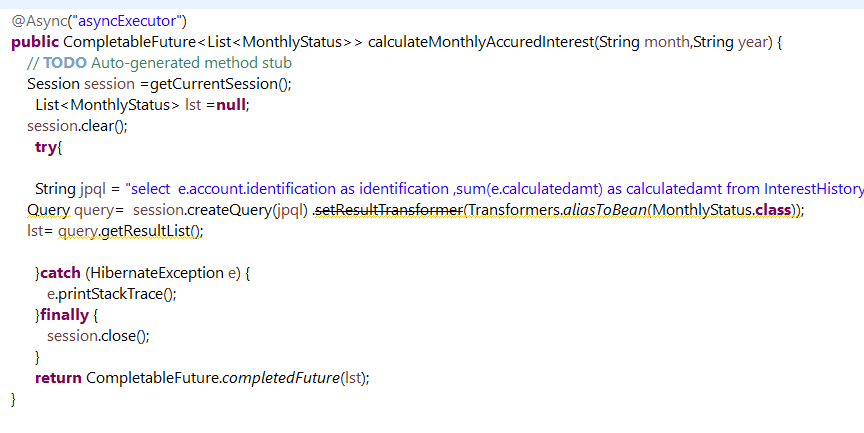
## Used Java8 features

* Streams

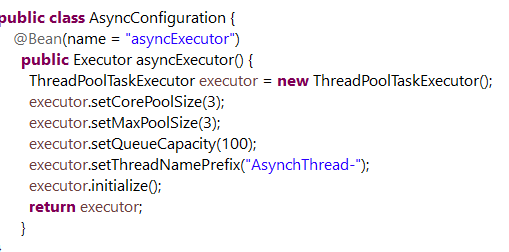


## Asynchronous

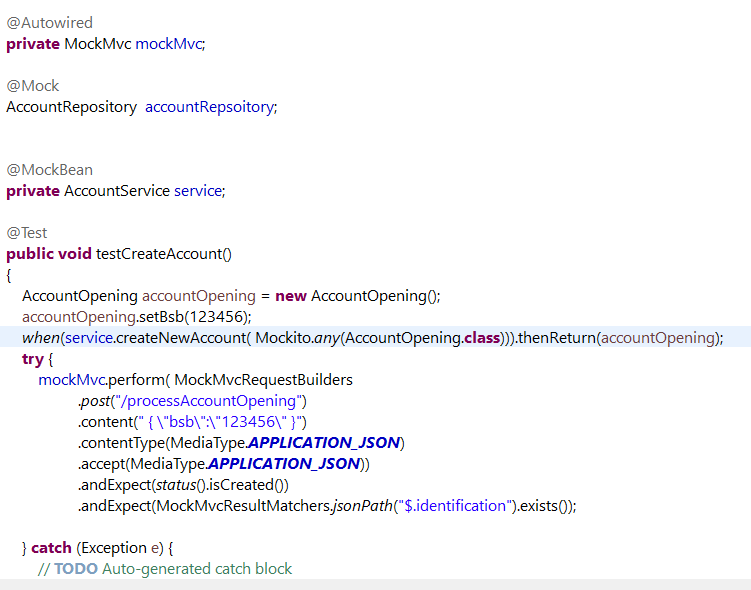
Used @Async annotation to execute a method/Function in an asynchronous way.



Async Configuration



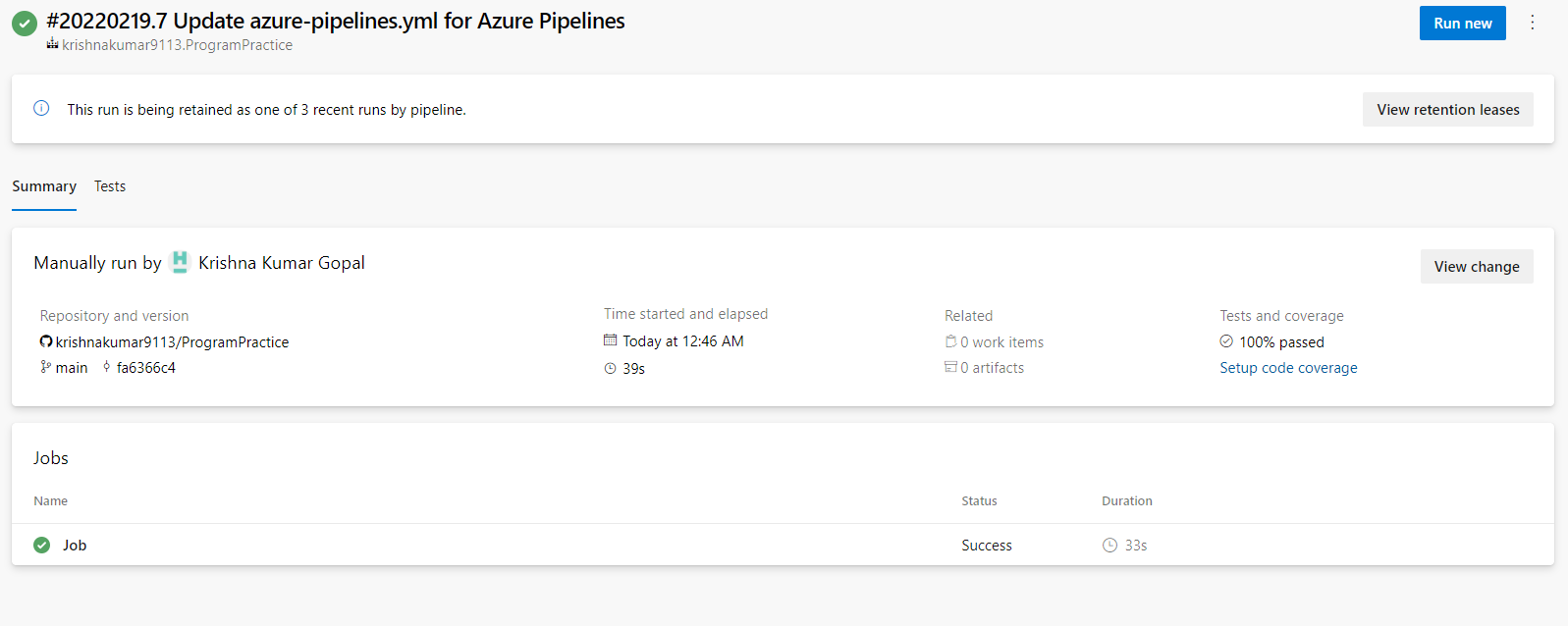
## Used Mockito and Junit for Unit testing:



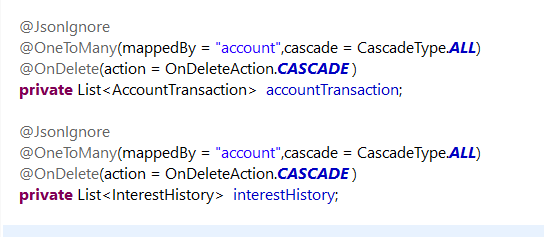
## Built the code using Azure Devops Pipeline from GIT Repository:

I have built the code using Maven from Azure Devops CI/CD Pipeline along with the unit test cases.

Please find the attached screenshot for reference.



## How would you handle account closures?

* 1. We can use CascadeType.REMOVE to remove accounts and its child data for our database since we use only @OneToMany relationships.
  2. It is not a good practice to use CascadeType.REMOVE if we use many to many associations. The cascading may remove more records than we intend to do.
  3. For many to many relationships, we need to implement the removal of child entities logic by ourself

# Suggestions

## Event Driven Mechanism

## Using AWS Cloud Services -Serverless Architecture

API Gateway- Configured with API Key and Mutual TLS 

Calculate Monthly Interest Request Validator

Account Opening Schema Validator 

Process Account EOD balance schema Validator

Transaction Schema Validator 

SQS



SQS

SQS

TXNSQS FIFO

Process Account End of Day Balance Lambda  

Account Opening Lambda

Transaction Processor Lambda 

* Transaction Processor Lambda

EOD balance status

Calculate Monthly Interest Lambda

SQS

Transaction

Put transaction message in TXN-SQS-FIFO queue for adding interest amount to balance.

Interest History

Account Processor 

SNS – AWS Simple Notification Service

AWS Cloud Watch event to trigger cron job -marked as 

Account

EMAIL or SMS as per requirement

1. “Calculate Monthly interest” lambda does not have any write operations hence it can read the data from read replica of the database.
2. If the user does not trigger “/interest/process-account-end-of-day-balances”, we can trigger the same as a cron job using AWS cloud watch.
3. Instead of having multiple schema validator for each API we can also configure a single schema validator lambda from the API Gateway too.

**Explanation:**

* + API Gateway can authenticate and authorize the incoming requests and forward it to corresponding Request/Schema Validator lambda, which checks the format of JSON request schema.
  + If the JSON message is in valid format, then lambda puts the message into the Message Queue. Once the message lands in Message Queue, an SQS event is triggered.
  + The next level of processor lambdas poll for the message queues and picks the messages.
  + The processor lambda can process these messages/ requests by interacting with dynamo dB tables.
  + Then the processor lambda can also send a message to a SNS topic which can trigger Email/SMS as per need. [Marked in thick blue in the above diagram]
  + AWS CloudWatch event also can be implemented to trigger “/processEnd of day balance service” as a cron job. [Marked with AWS cloud watch symbol ]
  + Process Account End of Day Balance Lambda - This lambda will
    - check If the EOD balance is already calculated, else it will trigger the update interest logic inside itself.
    - It will fetch the account balance from account table and calculate the interest for the present day.
    - This lambda will then update interest history table, and put a message into “TXNSQS FIFO” (marked in yellow) which will create an entry in transaction table for credit and update the interest amount into the Account balance.

## Using Microservices and Message Queues

For the given accounting application scenario. similar to aws cloud resources shown in the above diagram, Event driven microservice architecture can also be created by substituting / combining Lambdas into simple microservices and SQS queues with Message queues like Active MQ.

Microservices can be created using Spring boot and containerized using Docker, which can be orchestrated via Kubernetes.

## Improvements that can be done in existing code

1. More Junit test cases can be added
2. Idempotency can be achieved in “/account/process-account-opening” by having a column for “EmailId” as unique column in Account table. So that the user cannot submit the form multiple times with same emailID.
3. Since I had less time due to my current project assignments, I am not able to follow Coding standards strictly. Variable names and method names can be modified as per java standards. Code quality can be verified and improved using Sonar Lint also.
4. Validations can be improved at controller level for scenarios like account balance going less than 0.00. For now, bean level validation is implemented.

@DecimalMin(value = "0.00", inclusive = **true**)

@Digits(integer=10, fraction=2)

@Column(name = "balance")

**private** BigDecimal balance ;