Banking demo Application:

Approach we applied into current implementation:

1. We can create multiple account one at a time by providing parameters
   1. Account number : can’t be null and duplicate
   2. Balance : should not be negative
2. We can get the account details by sending the account id to
   1. /v1/account/{accountid}
   2. This will return the account details as
      1. Account Number
      2. Amount
3. There is two Models and one Entity class to persist data to DB
   1. Account : accountNumber, Amount
   2. TRANSACTION\_DETAIL\_TBL With multiple tranx Audit related fields

To perform any transaction between account at least two account should persist in DB with no Negative amount and debited account should have balance > then 0.

We have precheck conditions for account number Exist and amount >0 validation if both account details are available and our debit amount is not grater then the total available balance in account then we can allow the transaction to be performed.

Else we are throwing appropriate Exception either balance is low or account details not available.





@Transactional

Used for transaction management, it provides the feature like we can add attribute in that like

attributes rollbackFor or rollbackForClassName to rollback the transactions, and the attributes noRollbackFor or noRollbackForClassName to avoid rollback on listed exceptions

As it’s an spring boot application we do not need to Enable @EnableTransactionManagement annotation in main application.

@Transactional handles both Jpa and hibernate transaction handeling and it will reduce our boilerplate code and code management like rollback, commit etc.

**Transaction with kafak:**

There are several other ways to manage the Transaction in SpringBoot.

Once approach for multithreading environment is by using the Kafka messaging system which can accept multiple transaction in messaging queue.

In Spring Boot, enabling Kafka transactions can be done in the consumer and producer configuration parameters (at the very least).

We can use the ChainedKafkaTransactionManager if you want to [synchronize the database transaction with the Kafka transaction](https://docs.spring.io/spring-kafka/reference/html/#transaction-synchronization)

When an exception occurs, all published messages and database operations that happened within a transaction are rolled back. When the consumer restarts (and assuming errors are fixed), all operations within the transaction will only be committed if all succeeds. No multiple publishing of the same notifications. No repeated database inserts.

**Exception handling:**

1.In this application I have configured the exception handling through the @controlleradvice for global exception handling.

2.we have created our own custom exception handling class with appropriate code and error messages.

Definitely there will be more better way to structure the exception handling, but here I am trying to give an brooder picture how we can apply exception handling.

**Thinks we can structure our project more proper way:**

We should have the seprate dto class for request and response and seprate Entity which can talk to db for persistence

Dto level we can have validation classes for field validation or we can configure the variable level validation check also

These are my basic observation, but yes we can go for much better approach to develop things in much proper way.

Thank you

Anil kumar