Bank Application

Solution Document
By Vivek Malhotra

1 Entity Relationship

Below is the entity relationship diagram for Bank Application. Firstly we describe the entities.

1.1 Entities

Account – This entity defines the details of an account. It has properties like account id, account reference, balance, currency and transaction legs. Account class has oneToMany relationship with AccountTransactionLeg. Besides being JPA entity it has two convenience methods:

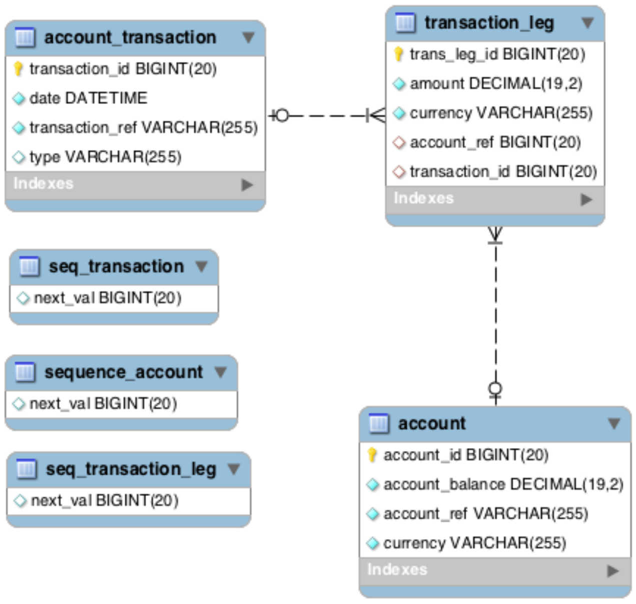
- *public void performTransaction(BigDecimal transactionAmount)* - This method performs a transaction on an Account. It first checks if the transaction will lead to negative balance. Successful transactions are carried out.
- *public Money getBalance()* – This method returns the balance in the account

AccountTransaction – This entity denotes transactions involving atleast 2 or more accounts. AccountTransactionLeg denotes this individual transaction leg on a particular account. This class also has some convenience method.

- *public Transaction toTransactionValueObject()* – This method converts the AccountTransaction object on which it is called to the value object Transaction. It also converts all the AccountTransactionLeg present inside the AccountTransaction to the value object TransactionLeg.

AccountTransactionLeg – This defines a transaction performed on one account. Each AccountTransaction should have atleast two AccountTransctionLeg.

1.2 ER Diagram



2 Technology stack

I based my solution on:

- Java 1.7
- Spring boot 1.4 release for IOC(Inversion of Control) and Dependency Injection.
- JPA as object relation model framework
- JUnit and Mockito for unit test cases
- h2 database for persistence in test
- Spring boot test for integration tests
- MySql database for use in production
- Maven 3.x for project build management

2.1 Spring boot

Even though Spring framework is the go-to Dependency Injection framework and lets us configure all components like

- Persistence and data – Spring data
- Spring Security
- Web using Spring MVC
- And many more

Annotations in spring have matured that you have to write minimum set of xml-based configurations. What spring still lags is you still have to bring all these components together and is less intuitive. For example, we still have to configure the dataSource. Many a times we are just copying the configuration xml and tweaking them for our requirements. We also have to make sure that versions of different

components are compatible with each other in the project build management tool of our choice.
Reason we chose Spring boot was just to avoid all these configurations and add more intuition to our application. Spring boot provides

- Default configurations to most of the components without writing any single xml.
- It also gives you the facility to override them through application.properties. You can also create properties for different profiles and activate them with @ActiveProfiles("profilename”) based on whether we are running application in production or writing integration tests.
- Minimalist dependencies in the project management tool. The version of dependencies is inherited based on Bills of Material pattern. We only need to define the version of Parent pom.

```
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-
parent</artifactId>
		<version>1.4.4.RELEASE</version>
		<relativePath />
	</parent>
```

Dependencies used by us in our application:

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-
jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.apache.commons</groupId>
			<artifactId>commons-lang3</artifactId>
			<version>${apache.common.version}</version>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
```

2.2 H2 database
We use H2 database as it is an embedded database with support of sequences and JPA and has a small footprint for our integration test classes. It is also the default database that Spring boot uses for running tests as our classes are marked with @SpringBootTest.

3 Application design

3.1 Important classes besides those already present in project

3.1.1 AccountServiceImpl
Implementation of AccountService. Important methods:

- *createAccount(String accountRef, Money amount)* – Method to create an account with provided reference and initial balance denoted by {@link Money}. It will create a modifying transaction.
- *Money getAccountBalance(String accountRef)* - Method to get account balance based on account reference.

3.1.2 TransferServiceImpl
Implementation of TransferService. This is a spring @Service and will create a read only transaction for all its methods by default. Important methods are:

- *public void transferFunds(TransferRequest transferRequest)* - Perform a multi-legged transaction on 2 or more accounts. The transaction is modifying and will be carried out with isolation level as REPEATABLE_READ. It guarantees that any data read was committed at the moment it is read. It also guarantees that any data that was read cannot change. if the transaction reads the same data again, it will find the previously read data in place, unchanged, and available to read.It will rollback the transaction if any exception is thrown in this block.
- *List<Transaction> findTransactions(String accountRef)* - This method finds the transactions for an account reference and returns a list of Transaction value object.
- *Transaction getTransaction(String transactionRef)* - This method gets an instance of Transaction based on the transaction reference.

3.1.3 AccountRepository
This is a JpaRepository and contains two functions

- *save(Account account)* – this saves an Account object and returns the managed entity.
- *findByAccountRef(String accountRef)* – this is to find an account with an account ref.

3.1.4 TransactionRepository
This is a JpaRepository and contains three functions

- *Save(AccountTransaction)* – this saves the account transaction.

- *findByAccountRef(String accountRef)* – This a @Query based on JPQL to get all the AccountTransaction associated with an account reference. Since one Account can have multiple AccountTransactionLegs. Also AccountTransaction has OneToMany relationship with AccountTransactionLeg Thus we use Inner joins to fetch all AccountTransactions of an Account reference.

```
SELECT DISTINCT trans FROM AccountTransaction trans
INNER JOIN FETCH trans.transactionLegs legs INNER JOIN
FETCH legs.account acc WHERE acc.accountRef =
:accountRef
```

- *findByTransactionRef(String ref)* – This is to find a transaction based on transaction reference.

3.1.5 AccountAlreadyExistsException

This business exception is thrown when creating an account with reference if it already exists.

3.1.6 BankFactoryImpl

It implements BankFactory and ApplicationContextAware interface. It implements the setApplicationContext(ApplicationContext appContext) method to set static instance applicationContext. This is then used to get the accountService and transferService bean.

4 Application Setup

4.1 Unit tests

We have written unit tests of individual JPA Entities, JPA repositories and Business services in their respective packages.

4.2 Integration Tests

We have written integration tests for JPA repositories and Business services using SpringBootTest and h2 as embedded database by default.

4.2.1 Configuration file

The configuration properties for tests is located in
<PROJ_DIR>/src/test/resources/application-test.properties.
Important properties:

- spring.jpa.hibernate.use-new-id-generator-mappings - this is set to true to use the new IdentifierGenerator instead of the deprecated SequenceHiLoGenerator for generating sequences.

4.2.2 Use MySQL database for tests

Uncomment the datasource properties

```
## datasource settings if using mysql for integration tests
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdb
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

4.3 Database in Production

Use the following DDL script to setup the mysql database for production
<PROJ_DIR>/src/main/resources/bank_ddl.sql

Configuration file is placed in <PROJ_DIR>/src/test/resources/application-test.properties.