IMPLEMENTATION ARCHITECTURE

Bank-Account application consists of two containers which are both needed in order to properly run the application and expose the HTTP RESTful APIs.

1. REDIS-SERVER CONTAINER

It exposes Redis-Server on port 6379 in order to guarantee the application persistence functionality, exploiting two relevant Hash-based caches:

- Account Cache (Map of <ID, Account>): all the accounts;
- Transaction Cache (Map of <ID, Transaction>): all the executed transactions. Three types of transaction are allowed: Deposit, Withdrawal, or Transfer. Different data models are used for them.

2. BANK-ACCOUNTS CONTAINER

It exposes the application APIs on port 8080, which are useful to:

- Manage Accounts:
    - save: it requires *name*, *surname*, *pin*, and *amount*; this operation will automatically assign a sequential *ID* to the new account.
    - update: it requires *ID*, *name*, *surname* and *pin*; it is not allowed to change the amount using the update API (this is an assumption, the only way to change the amount is executing a transaction).
    - delete: it requires the *ID* of the account to be deleted.
    - get-all: it does not require any input.
    - get-by-id: it requires the ID of the account.
- Manage Transactions: the allowed operations are the following ones:
    - save: it requires different input based on the transaction type to be performed. For DEPOSIT and WITHDRAWAL, *accountId*, *amount* and *pin* are needed. For TRANSFER, the one extra field needed is the *beneficiary accountId*. This operation will automatically assign a sequential ID to the new transaction.
    - get-all: it does not require any input.
    - get-by-id: it requires the *ID* of the transactions.
    - get-all-by-account-id: it requires the *ID* of the account.
    - others operations (delete and update) are not allowed on transaction because they could lead to inconsistency within the application. In fact, the account amount variations would be tampered and become confusing.

2.1 LAYERS

Internally, the Bank-Accounts container is divided in different layers:

1. controller layer: it provides the RestController and some logic to validate input (as DTO) and convert it from/to the internal model;
2. dao layer: it is in charge of persistence within the application and it is composed by managers and repository; the former communicates directly with the controller, while the latter exploits a Redis-Client (in Spring it is called RedisTemplate) to interact with the Redis-Server.

2.2 EXCEPTION

Custom Exceptions are defined in a dedicated package and they are largely used within the application to identify the proper error-cause and send the right error code/message to the

user. Exceptions are propagated and managed only by the top layer (controller) thanks to a dedicated method annotated with @ExceptionHandler.

2.3 TRANSACTIONALITY

Saving a transaction (either Deposit, Withdrawal or Transfer) involves at least two operations on both account and transaction caches. In order to guarantee consistency within the application, these operations must be performed in a transactional way. Transactionality is guaranteed using the RedisTemplate and its low-level "*execute()*" method.

2.4 CONNECTIONS

Redis-Template is the Spring Boot component (from dependency "*spring-boot-starter-data-redis*") which is used to connect to Redis-Server. Redis-Template is based on a Redis Connector which in the first implementation was Jedis library (3.3.0). Anyway, during tests it was found that a relevant issue is in place with the capability of releasing connections from the pool, which become exhausted after some requests. In particular, once the number of HTTP requests exceeds the number of connections in the pool, a timeout is always raised due to "*JedisExhausedPoolException : Could not get a resource since the pool is exhausted*". This issue is solved changing the connector and using Lettuce instead.

3. OVERALL ARCHITECTURE

The following diagram shows the overall communication architecture.