Daily Trade Reporting Engine

# Requirement

Requirement is to generate different reports for:

1. Incoming amount settled everyday
2. Outgoing amount settled everyday
3. Ranking of different entities

# Usecases

|  |  |
| --- | --- |
| **Usecase** | **Description** |
| **IncomingReport** | Set of instructions would be given as an input, these instructions can be both buying and selling instructions. System should calculate total USD amount settled for transaction type SELL. It should also group the amount date wise.  Report needs to be generated |
| **OutgoingReport** | Set of instructions would be given as an input, these instructions can be both buying and selling instructions. System should calculate total USD amount settled for transaction type BUY. It should also group the amount date wise.  Report needs to be generated |
| **RankIncomingEntities** | Rank all the entities based on the trade amount.  Input will be a list of instructions with both indicator types – BUY and SELL  System should calculate ranks for indicator type SELL  An entity with the highest sell amount should be ranked first, followed by other entities |
| **RankOutgoingEntities** | Rank all the entities based on the trade amount.  Input will be a list of instructions with both indicator types – BUY and SELL  System should calculate ranks for indicator type BUY  An entity with the highest buy amount should be ranked first, followed by other entities |

# Project Dependencies

Project is built on Java 8 features.

There is a restriction on the number of external jars that can be used to build reporting-engine. A max of two is only allowed. Hence, I have used only two (junit and slf4j)

|  |  |
| --- | --- |
| **Name** | **Description** |
| Maven | Used for build and running the application. Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework.  Invoking **mvn exec:java** on the command line will invoke the plugin which is configured to execute the class **com.jpmc.reportsystem.app.App** |
| Junit | For unit testing |
| Slf4j | For logging |

# Services

# All the services are loosely coupled and has no dependency on one another.

|  |
| --- |
| CalculationService  To calculate the total trade amount for incoming and outgoing transactions. It has two apis:   1. calculateTotalAmount    1. Takes list of client instructions and predicate as input    2. Instructions would be iterated using the predicate. Predicate would be either buy or sell    3. Calculates the trade amount for incoming and outgoing    4. Groups the date date wise 2. calculateDayWiseTotalAmount    1. To calculate the trade amount for a specific date |
| DataManipulationService  To manipulate the data and has one api exposed   1. To manipulate the settlement date, if the settlement date falls on a weekend for a particular transaction, then the same needs to be changed to the immediate working days. 2. This logic needs to be applied based on the partner 3. Reporting system has two partners configured currently    1. AED or SAR – for currency type AED    2. Default – for all other currencies 4. We have different partners configured within reporting application.    1. If the currency is of type – AED or SAR then the weekends would be Friday and Saturday    2. For all other currencies – weekend is Saturday and Sunday |
| RankingService  To calculate the ranking of all entities   1. Ranking service would calculate total trade amount for individual entities 2. Calculation would be separate for buying and selling entities 3. Entity with maximum trade amount would be ranked first |
| PartnerOperationalWindow – to identify operating windows of a configured partner   1. It has two implementation    1. **AEDPartnerOperationalWindow** – for AED/SAR currency type    2. **DefaultPartnerOperationalWindow** – for all other currencies 2. Has exposed three methods    1. getOperationWindow – fetches the list of operating windows of a specific partner    2. isSelectedDayWeekday – takes date as input and checks whether the input is a weekday or a weekend    3. findNextOperationDay – takes date as input and calculates the next operational day in case of weekend |
| DataReader DataReader has exposed following apis:   1. read - reading data from different source 2. close – close the resource 3. addConnectionDetails – url and resource to add a connection 4. isConnectionEstablished - to check the status of connection 5. isDone – flag to check whether the reader is done with the reading or not   DataReader has two implementation classes   1. **CSVDataReader** - This class provides an efficient way of reading the data as it will not read the complete set, instead in batches. This will boost the performance of the application as less data would be store in-memory 2. **MessageDataReader** – Mock implementation to show how easy it is to extend and use DataReader |
| DataWriter DataWriter has exposed following apis:   1. write- writing data to different destination 2. close – closing the resource   DataWriter has one implementation class   1. **ConsoleDataWriter** – For writing the data to console |

# Util

|  |
| --- |
| ReportingSystemResourceUtil Reporting system is reading all the values from the properties file. In an ideal scenario a external system like ZooKeeper should be used for managing the properties |
| ReportingSystemConstants All the constants are part of this file |

# Exception

|  |
| --- |
| ReportingSystemException All the exceptions are wrapped under ReportingSystemException |

# Orchestration

|  |
| --- |
| ReportingSystemProcessor ReportingSystemProcessor is the back bone of the reporting system engine. The main functionality of ReportingSystemProcessor is to orchestrate and call the independent services as per the need. All the dependencies are injected through constructors only. Calls are made to :   1. DataReader – for reading the incoming data 2. Calculation service – for calculation 3. Manipulation service – for fixing the settlement date 4. Ranking service – for calculating the ranks 5. DataWriter – for generating and writing reports |

# Tests

TDD has been followed and have tests for all the scenarios. Current code’s test coverage is 93.6 %

|  |
| --- |
|  |

# Running the application

|  |
| --- |
| App A plain simple class with no fancy logic. Just with a main method to run the processor   1. generate reports on console using command: **mvn exec:java** 2. run tests - **mvn clean install** |

Assumptions and Improvements

|  |
| --- |
| 1. **Calculation** - currently everything is happening within java code, we should be using some external tools like MAPLE, MATLAB, numpy a python library is also helpful 2. **File processing** - Large files needs to be broken down into smaller ones then process for better performance. Once processed, they should be moved to processed or error folders respectively 3. **Validation** - No validation is done on raw feed, I have assumed all the records will have complete set of data. There would be a possibility that few data are missing, in that case it should collect those data and report separately 4. **Using external frameworks** - External frameworks needs to be used:    1. Spring – for dependency injection and bean management    2. SpringBatch – for batch processing, scheduled service    3. ApacheCommonsCSV - Commons CSV reads and writes files in variations of the Comma Separated Value 5. **Different batches** - we should have different batch jobs for:    1. Break the large file and move smaller files to a folder where data manipulation service is polling    2. Changing the settlement date – this should be called first and once done it should be moved to a different folder on which the calculation and ranking batch job is polling    3. Calculation and Reporting – this can happen in parallel as both of them are independent. Once done, it should be moved to a different folder 6. **Package name** – I have used com.jpmc.reportsystem as the assignment was from JP Morgan |