Problem Statement – Rabobank Customer Statement Processor

**Functionality:** Rabo bank every month gets Customer transaction statements records. These records are sent in two formats CSV and XML File and these needs to be validated against two major features: -

1. All the transaction references which are numeric values needs to be validated.
2. The End balance has to be validated to check its correct ness against the current mutation.
3. When their validations fail the failure has to be generated as a consolidated report which will need to be sent to the stake holders.

**Components in this Application: -**

Ideally this application should have been build using SOAP Web Service which can validate the XML Schema against a canonical entity and then convert them into POJO components, but with the current time limit in mind we will have to use Rest Services which will allow us to use application/xml as the content-type.

On ideal grounds this application should have a proper user credentials where user will be validated against a LDAP or Active Directory component and if the application is stateless to provide additional security we can always use a Spring Security Oauth2 Framework where tokens could have invalidated and this framework is mobility friendly.

Additionally, there should be a user identity or user security related tables in the database which should have actions which will be related to controls along with roles that will be related with these set of authorization entities which will have relationship with security tables by which the successful login of an user will be diverted with ldap data to fetch corresponding complete user profile from the database and then stored in cache as an in memory user profile or stored in an IMDB if the application is stateless.

Ideally when Customer tries to perform any operation either he has to redirected to Login module if not logged in case with token this could be judged by the expiry time or validity of the token.

This actually should be activated by SSO module and the core security module should exist as a micro-service with its own database

On call Login he will pass through the Authentication Filter which allow him to pass through the Authentication providers post his / her credentials is validated and then the provider through the Authentication and Resourcing Server will generate a token which will be passed on to the User.

If the Application is stateless for every operation that the user performs the token validation to allow him to flow to the business logic layer.

* There should be gateway or Front Controller which will divert according to the Content Type to the right processor ->

1. Namely if the Content Type is CSV then it should direct or delegate the input to the CSV Processor Coarse Component.
2. If the Content Type is XML, then it should be delegated to the XML Processor Component.

The important components in the Coarse Grained CSV Component should include the following - Fine Grained Component

1. CSV Data Validation Component to Check whether all the required fields are available and it is well formed.
2. CSV Data Conversion Component to Java Bean or Pojo
3. Business Logic layer which will call the Repository to perform the validation and also update the data mutation performed using the Repository layer and the end balance if It found correct in the below steps.
4. It should ideally validate the existence of the Account or whether it is in the cold or active.
5. It should also check the uniqueness of the transaction reference with a common component which will perform this.
6. It should also check whether the end balance is valid by refereeing to the transactions which have been subsequently executed on the day.
7. If the end balances and the transaction references do not carry uniqueness or not correct then it should call the report module which will create a PDF document or a JSP Page with the report in our case, we can return a JSON array which can act as the values carrying reporting data.

The important components in the Coarse Grained XML Component should include the following - Fine Grained Component

1. XML Schema and Data Validation Component to Check whether all the required fields are available and it is well formed.
2. XML Data Conversion Component to Java Bean or Pojo
3. Business Logic layer which will call the Repository to perform the validation and also update the data mutation performed using the Repository layer and the end balance if It found correct in the below steps.
4. It should ideally validate the existence of the Account or whether it is in the cold or active.
5. It should also check the uniqueness of the transaction reference with a common component which will perform this.
6. It should also check whether the end balance is valid by refereeing to the transactions which have been subsequently executed on the day.
7. If the end balances and the transaction references do not carry uniqueness or not correct then it should call the report module which will create a PDF document or a JSP Page with the report in our case, we can return a JSON array which can act as the values carrying reporting data.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Technology Choice: -**

Although this could be achieved with different platforms and technology stacks, I have chosen Spring Boot as the base Framework.

Spring Boot is light weight and it allows a Spring Application to be deployed as a Fat Jar / war with a Tomcat Application Server, which could be replaced with other server modules or separately deployed as a war in other Application Server.

With Spring Boot -> Spring brings the concept of micro services to implementation in Spring Framework and provides an opinionated structure using maven / gradle as build tool where different spring modules and their dependencies can be injected into local repository as fat spring jars saving the developer to download the spring needed dependency as further dependencies.

As time line is short and as we are not building an enterprise application here, I have gone with an embedded or in-memory database called H2 which could be brought in as a dependency through spring boot plugged-in maven which is configured using its pom.xml file.

This database could be run through running the Spring Boot Application with minimum configuration which could be put in the classpath: properties (normally /src/main/resources in project template / folder) file and even could be configured for different active profiles, namely the different environments.

Spring boot also comes with Actuators which are Spring modules that allows the application to be monitored again by using easy configuration in the same path mentioned above to check on the application health.

For CSV we can use an opencsv module which can be integrated with Spring boot but to be added as a separate dependency which will be used by the csv parsing components.

As we are using Spring Rest we can use Spring-Web which will allows the glued Rest Framework code to be used through the Controllers which acts as the Command Components gluing in the Business Components or Service component which are injected as interfaces where in run times the Spring Application context glued them in using the Cglib weaving and Spring own Dynamic Proxy Pattern of JDKDynamicProxy called through Spring AOP module.

For Data Integration Layer I have chosen Spring Data JPA as the tables have relationships or referential integrity which could be seamlessly handled through JPA which could be configured through those classpath: properties file to allow Hibernate as the JPA Vendor through which DDL or Schema Creation, DML, pagination wherever required along with creating the initial or seed data where necessary.

The Spring Data JPA also internally implements the Repository pattern which removes the data interaction code cluttering by implementing all the common methods through its JPARepository interface which is generics enabled again at run time made available through Cglib and Dynamic proxy pattern implementation.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Actual Implementation Steps: -**

1. First the needed Spring and H2 modules were weaved and downloaded as a project using Spring.io or Spring Initializer.
2. The project was then built using maven from command line and later updated through -> STS IDE – maven -> update project option
3. The IDE used was STS.
4. I started bottom up, after analyzing the requirement and gathering the clarification.
5. I created the database entities by writing a schema.sql (carries the ddl scripts) and data.sql -> the second for inserting the seed data, as I felt that there should be initial data for this application to do the necessary fetching and business logic validation.
6. Initially I had used Springs embedded Database builder to build the DDL and update the DML to ensure the data frame is coming up correctly, the Embedded database builder was glued in through the Spring boot application main class and configured as a bean.
7. Building of the above took time as there was a set of unforeseen errors which came in, as for every step I was manually testing and resolving the issues
8. This was later commented to inactive the code in favor of Spring Data JPA.
9. Once the first building block was completed I brought in the Data Entity building code through Spring POJO / Beans which were glued to Spring JPA using annotation including the relationships or association which were built in through the code.
10. For Each Entity and its relationship code was tested using the data.sql script which ensured the coded implementation is free of DDL and DML bugs.
11. The above came in with a set of defects as challenges and I have recently used more of Spring JDBC than Spring Data JPA, it took me some time to resolve this effect, once the initials were done, the velocity grew.
12. Once this was resolved completely, I had started to implement the data repository for data interaction which had generics enabled for the entities they were managing and the surrogate key.
13. I had to end here as the time limit came to end here.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**What remained pending: -**

I would have loved to take the Test Driven Approach by which initially I could have even mocked data to check right data interaction is happening before going into the actual business code.

With the transaction boundary coming through the test classes as they are proxying the service till they are built, remember this is testing the data interaction layer only as unit.

When the Services are built using the business logic, the transaction boundary would have been drawn through this service classes and individual method which is handling the validation would have been exposed as façade and the transaction boundary would have been drawn from here.

While building the services the TDD approach along with mocking of data and then passing through the service classes would have ensured that the business logic gets tested before the actual implementation is tested with the controller classes.

With Mock Framework we could even mock each of these components for a scenario where the complete functionality of these components are not clear from the requirements and we need to test the input and output comes in line with our thought process.

Now when the Rest Controllers are implemented, we will build the adjoining the validation layer which with opencsv and Jaxb will do the initial the data validation to understand the inputs are correct and then convert them into POJO carrying the data from the end customer.

The above will be passed into the service through the façade methods or Service API, which will do the necessary business logic check, data interaction to do the necessary validation.

This will then be converted into another POJO and returned as rest response with the right message where the success message or validation are configured in database.

Also using Jasper, JReport, XSLT using Velocity we can created PDF and send an URL link to the customer which will allow him / her to redirect to get the PDF rendered or downloaded through the browser.