# Brief Report: Analysis and Roadmap for the Yape Code Challenge

By José Antonio Portilla Fernández, Systems Engineer.

## Business Requirements

A solution must be made to save transactions. It is required that every time a financial transaction is created, this solution must validate every transaction with an anti-fraud microservice which, in turn, must send a message back to update the transaction status and only then send a response to the client that consulted the transactions service in the first place.

The solution currently manages three transaction statuses:

1. PENDING
2. APPROVED
3. REJECTED

It is also required that every transaction with a value greater than 1000 is rejected. It was also pointed out that the system would be under high volume scenarios where it will have a huge number of writing and reading for the same data at the same time.

## Disclaimer – Assumptions

It is never a good idea to make assumptions in any project, a scrum master or a product owner should be consulted if any doubts appear in the development process. Even before that, almost everything should have been as clear as possible during the project planning and subsequent sprint planning. Due to the lack of a present Product Owner, I am forced to make certain assumptions to be able to develop the solution.

1. The provided resource number 1 lists all the information that must be sent to be able to create a transaction:

```
{
  "accountExternalIdDebit": "Guid",
  "accountExternalIdCredit": "Guid",
  "tranferTypeId": 1,
  "value": 120
}
```

2. The provided resource number 2 lists all the information needed by the client:

```
{
  "transactionExternalId": "Guid",
  "transactionType": {
    "name": ""
  },
  "transactionStatus": {
    "name": ""
  },
  "value": 120,
  "createdAt": "Date"
}
```

3. The names provided imply that these are transactions made from cards, either debit or credit.

4. The name transactionExternalId seems to be the identifier of the transaction itself once it is inserted.

5. The structure of the resources, especially number 2, imply that there is more data and tables stored. At the very least 2 more tables that hold data for transaction types and transaction statuses. Such data should never be hard coded. In this exercise no data is going reside in the code, but the data base structure will be simplified in the case to not overcomplicate both development of this small proof of concept and its subsequent review.

6. Due to how this assignment was worded, this solution for transactions would presumably also be simply a micro service, would be under the protected layer of the network architecture and would later need another exposed server that would function as a bridge with the web or mobile app requesting such operations.

7. The suggested tool GraphQL is an API mostly used for communicating a front end with a backend. Communication between microservices should be more lightweight.

8. I assume that the condition: “every transaction with a value greater than 1000 should be rejected” is belongs to the anti-fraud service and it is the only thing it should do for now, nothing more. I added a common-sense condition to not accept negative values either, since the system seems to be managing types of transactions. One of this nature would be a withdrawal or outward transfer.

9. The first resource seems to be information for insertion but one of the variables sent has a typo: tranferTypeId, a missing letter s. I assume it is transferTypeId.

10. Even if I fix the supposed typo, the response required when getting the information of a transaction is, according to resource 2, transactionTypeId. I assume they refer to the same data because there is no data type of that name that acts as a column for our table. Even though the response could be configured to answer with a different variable name we have to remember that it is a bad practice to do these adjustments in the backend service. The front end is the one that should change the name of the variable in its request to transactionTypeId to avoid any kind of confusion.

11. The fact that in resource one there are two fields for different types of cards is really confusing. A transaction can only be from either one of them, I assume that a business rule might be that only one of those fields should be full. If both are filled it should give an error. Since I don’t know much detail about this specific business case, what is the end objective of this would-be project I cannot make an accurate assumption. For this reason I am allowing both fields to be filled and I am not going to establish any restriction on both whatsoever at the moment.

12. I have included in the code a script to fill utility tables (status and transaction types) so you don’t have to worry about the set up beyond executing 2 commands.

## Chosen Technologies

The only restriction given is that the solution must use the Java programming language. This freedom led me to choose the following technologies, ranging from design patterns to architectures, dependencies, packages and tools:

1. Microservices Architecture: First and foremost, I decided to follow this architecture to be able to create small services that can be developed, deployed, and scaled independently. This also allowed me to decouple each service since each one focuses on a specific and different business and logic role. This will also allow in the future for each microservice to deploy its own security measures according to specific needs. Besides it was heavily implied that I should use this type of architecture.

2. Event-Driven Architecture (EDA): It was heavily implied, thanks to the description and the very same docker compose file provided that this should be used with the help pf Apache Kafka. Services will communicate asynchronously through events, and will help with decoupling and scalability.

3. Layered Architecture: I could argue that I am also using a bit of the layered architecture since the antifraud service and the transaction service can be considered to belong to different layers. The transaction service will use the antifraud service but it won’t ever be the case vice versa.

4. Concurrency Pattern Producer-Consumer: I will coordinate the actions of asynchronous requests that have elements that produce data and others that consume it.

5. Java Spring Boot: It not only has the benefit of dependency injections, but it also has an embedded server that will help accelerate the development and testing.

6. Lombok dependency: It will help reduce boilerplate code and accelerate development.

7. GraphQL API: It is a point of debate and investigation on my part whether or not a classic REST service is preferable, but as with most cases it depends on various factors that will be discussed later in the final chapter of this report. For now, let’s say that it not only was a suggested tool in the instructions, but it also has the well-known benefit of reducing over-fetching and under-fetching, which will help in this high stress scenario.

8. JPA: The Java Persistence API will allow the quick set up of the table and entity I will use and will also give me the option of treating the retrieved information as a Java class immediately if need be.

9. PostgreSQL: A commonly used relational database.

## Basic High-Level Planned Software Workflow

1. Transaction Service - Transaction Creation: When a GraphQL request is received, createTransaction in YapeTransactionResolver is called and the initial creation of a PENDING transaction is done.

2. Transaction Service - Storage of a CompletableFuture: A CompletableFuture is created and stored in a ConcurrentHashMap keyed by the transaction ID itself to ensure it is unique.

3. Transaction Service - Communication – Send message to Kafka: The transaction request is sent to the transaction-events Kafka topic and identified with a unique key.

4. Transaction Service - Communication – Wait for Kafka Response: A method waits for the Kafka response by calling future.get(30, TimeUnit.SECONDS).

5. Anti-Fraud Service – Communication – Receive Kafka Message: Anti-Fraud Service listens to the stream and captures the messages with the right topic.

6. Anti-Fraud Service – Fraud Validation: The value of the transaction is evaluated.

7. Anti-Fraud Service – Communication – Send message to Kafka: Anti-Fraud Service sends the result of the evaluation with the proper unique key.

8. Transaction Service – Communication – Receive Kafka Message: The YapeTransactionListener receives the Kafka message with the unique transaction key and the right topic.

9. Completion of the CompletableFuture: The YapeTransactionListener calls handleKafkaResponse with the unique key, which completes the CompletableFuture associated with the transaction ID.

10. Return Response: Once the CompletableFuture is completed, the createTransaction method returns the response.

This design ensures the Transaction service waits for the response from Anti-Fraud Service before returning the final result.

## Roadmap for the Project

1. **Setup the Project Structure**

   - Create two Maven projects: transaction-service and anti-fraud-service.
   - Configure Spring Boot for both services.
   - Add dependencies for Lombok, PostgreSQL, Kafka, Zookeeper, GraphQL, and Spring Data JPA in the pom.xml files.
   - Create the docker file for each one.

2. **Configure PostgreSQL Database**

- Set up PostgreSQL in the docker compose file.
- Properly configure a database schema and table for the transaction service in the docker compose file.
- Add the PostgreSQL driver and configure the data source in application.properties file for the transaction service. Ensure that the driver is compatible with JPA.

3. **Implement the Transaction Service**

    - **Entity and Repository**: Define the YapeTransaction entity and create a JPA repository.
    - **Service Layer**: Implement the transaction creation, update, and retrieval logic.
    - **Controller - Resolver**: Create GraphQL endpoints for managing transactions.
    - **Kafka Integration**: Configure Kafka producer to send transaction events to the anti-fraud service with the right topic.
    - **Asynchronous functionality**: Ensure that the service doesn't respond until the transaction has been correctly validated.
    - **Docker Compatibility**: Configure the docker file so the docker composer can later compile the project with ease.

4. **Implement the Anti-Fraud Service**

    - **Service Logic**: Implement the logic to check if a transaction is fraudulent based on the value and conditions established about it in the project instructions.
    - **Kafka Integration**: Configure Kafka consumer to listen for transaction events and produce a response event with the right topic.
    - **Docker Compatibility**: Configure the docker file so the docker composer can later compile the project with ease.

5. **Event Communication via Kafka**

    - Set up Kafka and Zookeeper in the docker compose file.
    - Define topics for communication between the transaction service and anti-fraud service.
    - Double check both services are correctly configured to produce and consume events. Check application.properties

6. **Update Transaction Status Based on Anti-Fraud Response**

    - **Event Listener**: Implement Kafka listener in the transaction service to update the transaction status based on the response from the anti-fraud service.

## Further Steps and Recommendations

**Regarding Rest vs GraphQL**: Classic REST and GraphQL have both pros and cons, in this high stress scenario we need to compare the main ones.

- GraphQL: It reduces over-fetching and under-fetching, but Query parsing and execution can introduce additional overhead on the server. It does not have built-in caching; however, this functionality can be forced with different techniques.

- REST: Well-understood and straightforward to implement and it is easier to take advantage of the HTTP caching mechanisms, but clients might receive more or less data than needed.

My final recommendation for this matter and high volume scenarios would be to do realistic stress tests with both methods, REST vs GraphQL, this way we could see which one eats up more processing power and is more prone to errors. In this day and age, it is most likely that the services are going to be in the cloud, either AWS, Azure or GCP and this way we could not only see which method is best for this use case, but we can also check which one saves more money in each Cloud Service.

**Regarding Security:** Critical services, especially the ones that operate with databases should be behind a security layer in the network architecture. Network architecture and information backup is out of this scope, but I mention it regardless to show that is not something that I would omit in a production environment.

**Regarding reliability and service levels:** If this business case is one that has a high volume of transactions it would definitely need a load balancer to distribute the work and have backups in case of failure.