# wefox Group engineer Challenge

Technical Test Inc. is a clothing company that recently had a big commercial success. As a result, the number of transactions has dramatically increased, and the old system doesn't scale to the required transaction rate on peak hours.

For this reason the system has been redesigned and one of the key pieces is the new payment processor. This new microservice will consume payments from the message broker and process them at its own pace. In addition, it will communicate via REST API with third parties for validation and logging. Your goal is to implement this microservice.

## Specifications

Technical Test Inc. has two types of payments online (web) and offline (CRM/Shopping center).

All the valid payments should be stored in the payments database, regardless of the payment type. Furthermore, each time we store a payment in the database we should update the account information with the last payment date.

In case of error, we would like to **keep ALL** the information and try not to lose it. For that we need to log all the errors in Log System via a REST call.

### CRM/Shopping Payments (offline)

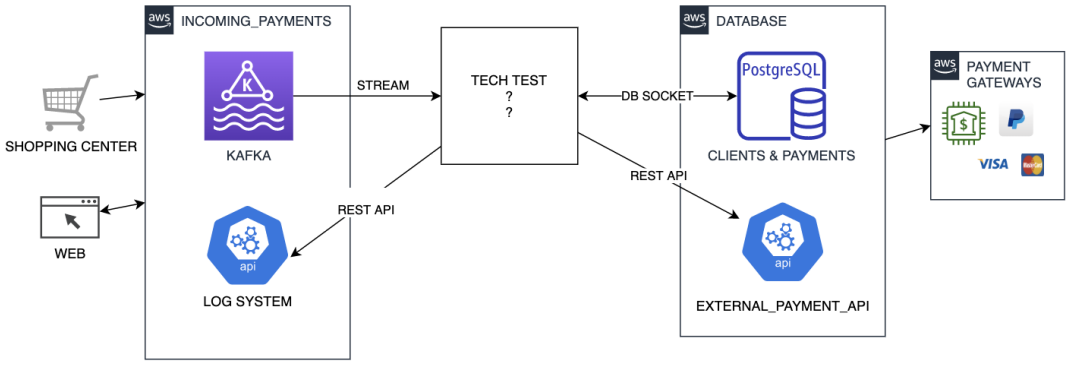
There is no need to do payment validation, all the payments are done with cash or prevalidated in the shop.

### Third party Payments (online)

We need to validate against some third-party provider (PayPal, Stripe, credit, etc.)

Currently, we have some gateway taking care of that, we only need to connect via REST API and send the request to validate the payment. `2XX OK` response means a valid payment. Otherwise, it's invalid, and we should treat it like an error.

## Architecture overview




As we can see it's a client-server problem which involves reading the data from the stream, interacting with some REST API, store the data in the database and log errors.

## Infrastructure

The required infrastructure is already provided. All you need to do is to install Docker Engine and to use docker-compose to start the services.

To start the services run `docker-compose up`. Use `-d` to start in detached mode and run the containers in the background.

For a more detailed information about docker-compose commands refer to the official documentation: Overview of Docker Compose.

### Database

PostgreSQL (plain auth, port 5432)
All the schemas and initial data import are stored in .sql files inside the Docker database folder.

### Event streaming

Kafka (plain auth, port 9092 in internal Docker network or 29092 in external)
There are two topics in Kafka that we should read:

- "online": process data as online payment
- "offline": process data as offline payment

The message structure is the same as the payment data model. (see below)

### Log system

Log REST API and gateway REST API use the same endpoint to keep it simple (in the architecture overview they are separated). (port 9000 without SSL).

## Endpoints

### To use in the implementation

- `http://localhost:9000/payment`: To check if the payment is valid with third-party providers. Only accepts JSON with POST request.

```
curl -i --header "Content-Type: application/json" \
  --request POST \
  --data "{'payment_id': 'fdf50f69-a23a-4924-9276-9468a815443a', 'account_id': 1,
'payment_type': 'online','credit_card': '12345','amount':12}" \
  http://localhost:9000/payment
```

- `http://localhost:9000/log`: To store error logs. Only accepts JSON with POST request.

```
curl -i --header "Content-Type: application/json" \
  --request POST \
  --data "{'payment_id': 'fdf50f69-a23a-4924-9276-9468a815443a', 'error_type':
'network', 'error_description': 'Here some description'}" \
  http://localhost:9000/log
```

**Other endpoints**

- `http://localhost:9000/start` : Starts the technical test. Before starting, it deletes all the payments in the PostgreSQL database, resets logs and starts to produce payments data in the Kafka stream.

```
curl localhost:9000/start
```

Once we start the emission, the producer will emit around 1000 elements in 1-2 minutes through the Kafka stream and stop. If you want to restart the emission from the beginning, simply call the start endpoint again.

If you have problems with database cleanup on "restarts", use some pg admin tool like *Postico* in order to clear all payments table before executing the start command again.

- `http://localhost:9000/logs` : Simply visualizer of error log values.

```
curl localhost:9000/logs
```

- `http://localhost:9000` : basic HTML website with buttons to start the test or view the logs if you don't want to use cURL.

## Data model

Some models can be checked with the provided *.sql* schemes.

*Account model* (account database table)

- account_id : primary key autoinc Integer
- email: String
- birthdate: Date
- last_payment_date: String with last payment date
- created_at: Timestamp with creation date

*Payment model* (payment endpoint and payments database table)

- payment_id : unique String identifier
- account_id : foreign key to the accounts table
- payment_type: text enum (String), possible values are:
  - "online": String value that represents that payment should be verified through gateway API
  - "offline": payment stored in the database without verification
- credit_card: String with some payment information
- amount: Integer with price amount
- created_at: Timestamp with creation date. Only in database

*Error model* (check log endpoint JSON)

- payment_id : unique identifier (string)
- error: text enum (String), possible values are:
  - "database": for all related to database operations
  - "network": for all related to online
  - "other": rest
- error_description : some extra info that you can give some overview of error like number, description, etc.

## Delivery

The technical test is expected to be solved in 4-12 hours depending on the expertise/complexity of the solution.

Unless it has been explicitly requested to solve the problem in a specific language, you are free to use the following programming languages: java, node, python, kotlin and scala. If you would like to use a different language, please contact us before.

To submit your solution please create a private Github repository, commit your code and share it with us. You will be provided with the Github user with whom you should share it.

We like clean code and scalable solutions :)

If you are running out of time you can explain what else would you implement and list improvements in *README*.

For the right prioritizing, logs REST API problem part must be the last one to be implemented.

Feel free to modify all docker-compose system, add or modify tables, add new docker services-machines, etc.