wefox Group engineer Challenge

Description

Technical Test Inc. is a clothing store that needs to have a new centralized payment system through data stream queues.

Currently, they have two types of payments online (web) and offline (CRM/Shopping center).

Once a payment it's done needs to be processed.

CRM/Shopping Payments (offline):

There is no need to do checks, all payments are done with cash or prevalidated on the shop so we need only to store payment info in the payments database.

Third party Payments (online):

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

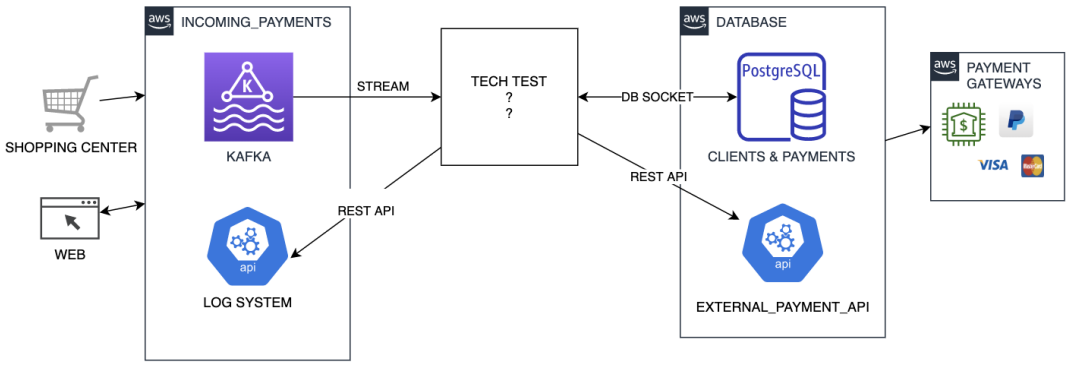
Currently, we have some gateway taking care of that, we only need to connect via REST API and if the response it's `2XX OK` we should take the payment info and store it in the payments database (same as offline).

Each time we store some payment in the database we should update the account information with the last payment date.

For us, it's very important that if some error occurs we **keep ALL** information and try not to lose it. In case of an error we need to inform with REST API service to log all errors.

Technical stuff

Architecture overview



As we can see it's client-server problem about reading info from the stream, interacting with some REST API, store info in the database and log some information in case of error.

All the infrastructure it's based on docker-compose so we only need to start services.

Problem solution:

We are aimed to develop some program that connects to infrastructure and solves explained the logical problem.

Unless it has been explicitly requested to solve the problem in a specific language, we allow to do a technical test using one of the following programming languages: java, node, python, kotlin and scala but if you want to use another one, please contact us before.

Once the infrastructure it's up and running and the solution program connected we can start Apache Kafka producer by using start endpoint (explained in the endpoints section), which will start to emit in Kafka stream all the payments.

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

Problem delivery:

Open a repo in GitHub/GitLab/Bitbucket, send it to us, so we can see the progress and create an application that solves what is requested.

The technical test it's expected to be resolved in 4-12 hours as minimum value depending on the expertise/complexity solution.

We provide some *README* to fill with execution instructions, notes and things to improve, please fill it before sending the solution.

The code must be developed in order to be clear and scalable.

If you are running out of time that wants to spend in technical test other solutions or improvements can be explained in *README*.

For prioritize, logs REST API problem part must be the last thing to do.

Infraestructure:

We provide some docker-compose file with all needed information.

Basic commands:

- Create and start Docker containers. Starts development server: `docker-compose up`. Use `-d` to start in detached mode: run container in the background.
- Stop and remove Docker containers, networks, images, and volumes created by *up*: `docker-compose down`.
- Starts existing containers for a service: `docker-compose start`.
- Restart existing containers for a service: `docker-compose restart`.
- Stops running containers without removing them: `docker-compose stop`.
- Display and listen log output from services: `docker-compose logs -f [--tail=<last-lines>] [SERVICE...]`.

For doubts about docker-compose refer to the official documentation: Overview of Docker Compose.

Examples are using localhost but if you deploy your solution inside the Docker network remember to use correct hostnames. All ports are exposed to internal or external networks in some way.

- Database it's based on PostgreSQL (plain auth, port 5432)

    - All scheme tables and initial import are stored in .sql files inside the Docker database folder.
- Stream-based in 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 it's the same as the payment data model.

Log REST API and gateway REST API uses the same endpoint in order to keep it simple (in the picture are separated). (port 9000 without SSL).

Endpoints:

- `http://localhost:9000/payment`: To check if payment it's valid with third-party providers. Only accepts JSON's with POST Request.

Data structure and type can be checked with the provided *.sql* payment structure.

cURL example with JSON structure:

```
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's with POST Request.

cURL example with JSON structure:

```
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 technical test by deleting all payments in the PostgreSQL database, reset logs and starts to produce payments data in the Kafka stream.

You can start or restart it from following cURL command.

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

Once we start emissions, the producer will emit around 1000 elements in 1-2 minutes through the Kafka stream and will stop emitting finalizing tests. If you want to restart from the beginning simply load the start endpoint again and it should clean the database and start to emit from beginning.

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 point to start or logs endpoints if you don't want to use cURL

Data models:

Some models can be checked with 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_decription : some extra info that you can give some overview of error like number, description, etc.