Technical Test

Please read these instructions carefully, they contain useful information to help you complete the test successfully.

Purpose

This test will ask you to implement features on a microservice as you would in your daily work with us.

The micro-service is responsible for managing the loyalty program, and is using an event-driven approach by listening to AMQP messages and reacting accordingly.

Prerequisites

As this is a NodeJS project, you will of course need to install **Node** (>= 8.5), **npm** and **nvm**.

It also needs a **Rabbitmq** server and a **Mongo** database to run locally. The easier way to do this is to run them in **Docker** containers.

Here are the few steps you would have to follow if you don't know how to setup this :

1. If you don't have one, create a Docker hub account here https://hub.docker.com/

2. Login on your terminal

```
> docker login
Username: your-docker-id
Password: your-password
```

3. Pull images, and create the containers with correct binded ports

```
# RabbitMQ
> docker pull rabbitmq:3-management
> docker create --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

# Docker
> docker pull mongo:3.6
> docker create --name mongo -p 27017:27017 mongo:3.6
```

4. Now, just start the containers with the name you gave before

```
> docker start rabbitmq

> docker start mongo
```

Additional notes

Message producer

A “producer” is provided (have a look at the ***producer*** folder), and simulates a production environment by generating user events you're supposed to react to.

To keep this test short enough, only 3 events are published: **user signup**, **ride created** and **ride completed**.
To use it, you will need an **AMQP** server (like **RabbitMQ**) running locally or in a Docker container (cf ***Prerequisites***).

You can get further details in the producer’s ***README.md*** file, you should quickly read it as we think it would help you understand this test philosophy.

Skeleton

We won't ask you to implement the whole microservice, as it would take time and wouldn't reflect how you fit in our organization, as we developed tools to easily create microservice skeletons.

Instead, you are given a project as if you worked with us: the architecture and services instantiation are already implemented. Also, some people already worked on the project so you may find features already implemented...

It gives you a real opportunity to add useful features and show us how you work ;)

Tests

We think that our code should always be unit-tested, and we'll expect you to follow this guideline. Some tests are already written for the provided skeleton so don't hesitate to use/copy/paste them to save time!

Even if you don't have the time to have a total coverage, show us that you share our philosophy by testing the main features and exceptions/errors.

TODOs

You'll have to implement some features from scratch, but sometimes you'll just have to improve existing code. To simplify this, look for TODO comments, they will give you indications.

Required Goals

Implement loyalty points earning

Task

Improve the worker to handle queue events and update the riders loyalty status and points.

Rules

Each rider has a status. The status is computed from the number of rides completed according to the following rule:

- ➔ Bronze***:*** **0** ⩽ nb rides ≺ **20**
- ➔ Silver: **20** ⩽ nb rides ≺ **50**
- ➔ Gold: **50** ⩽ nb rides ≺ **100**
- ➔ Platinum: **100** ⩽ nb rides

When a rider finishes a ride, he gains an amount of loyalty points. The amount of points is computed with a multiplier according to the following rule:

- ➔ Bronze: **1€ = 1 point**
- ➔ Silver: **1€ = 3 points**
- ➔ Gold: **1€ = 5 points**
- ➔ Platinum: **1€ = 10 points**

Examples

If a bronze rider paid 15€, he earns 15 loyalty points.
If a gold rider paid 8€, he earns 40 loyalty points.

Important note

Here, you may find:

- a handler already implemented but with missing test(s), it's up to you to add any test you think may be useful.
- tests that are written on a handler but break because the corresponding feature isn’t implemented (somebody said TDD?)

Provide data to other microservices

Task

Improve the API so you microservice can handle requests from other microservices when needed.

Rules

1. If a user uses some of his points to pay for a ride, the "ride-handling" micro-service should be able to call your microservice to **remove** them .

2. For specific marketing operations, we would need to get, for a given user and a given loyalty status, the average amount he spent on all the rides he made with this loyalty status.
   Ex: how much did Jack (a premium Platinum client) spent on his rides when he was a Bronze user ?

Important note

An API should always handle authorizations. For example, we don't want anybody to be allowed to modify a user's point value... But this is not the purpose of this test, and it is not necessary to consider this here.

Data model

Here, no task is clearly given. We just want you to make sure that your data is correctly handled, as for us it is one of the factor of a good, maintainable and scalable microservice.

Hints

We don't have limitations on databases size, thus feel free to manipulate and store any data you think would be useful.

We don't like to directly manipulate data in the database. Model's helpers are the way we do it.

"LIVE" Testing

Your solution should run smoothly:

- worker and server should start without crashing
- tests should pass without errors

To make sure of this, here are the command to start your services.

```
> cd back
> npm i
```

Start the worker

```
> npm start:loyalty_worker
```

Start the server

```
> npm start
```

Whenever you're ready to "live test" your code, start your worker to handle producer's published messages.

Your api can be tested by starting the server and calling the route you defined on ***https://localhost:8080/api***