switch.

Engineering Challenge.

Table of Contents

Intro

First of all, congratulations for reaching this step of the hiring process. The idea with this challenge is to evaluate your analytical and technical skills.

The Project

Currently the service that handles the **payments** and **refunds** it's a "monolithic application" with a single relational database. For scalability reasons Switch wants to improve those services architecture.

Your main goal is to replace this "monolithic application" with an architecture based on microservices using Django as the main web application framework and PostgreSQL as the main relational database.

Goals

- Develop a Django application for the **payments** microservice;
- Develop a Django application for the **refunds** microservice
- Both microservices should have their independent **PostgreSQL database**;
- Both microservices and databases should be "dockerized" and run with **docker-compose** (e.g. docker-compose up will run the **payment** microservice, **refund** microservice, **payment** database, **refund** database and make all the required **database migrations**);
- **Don't use** any REST frameworks, just pure Django;

Requirements

In this section we will define to you the requirements for the microservices implementation.

Payments

- Create a data model that represents the Payment entity with the following attributes:

- **payment_id:** represents the unique identifier of the payment. Properties and constraints are:
  - String
  - Unique
  - Auto generated on save
  - Maximum size of 200 characters
  - Cannot be null
- **amount:** the numeric value of the payment amount. Properties and constraints are:
  - Float
  - Cannot be null
- **payment_method:** payment method type. Properties and constraints are:
  - String
  - Maximum size of 50 characters
  - Cannot be null
  - Only accepts the following values: **credit_card**, **mbway**
- **created_at:** date when the payment was created. Properties and constraints are:
  - Datetime field
  - Auto generated on save
  - Cannot be null
- **status:** the current status of the payment. When a payment is created on the database with success **the status should be success**, otherwise if some exceptions occur **the status should be error**. After that the status can only be changed to **settled**. Properties and constraints are:
  - String
  - Maximum size of 20 characters
  - Cannot be null
  - Only accepts the following values: **success**, **error, settled**
- **settled_at:** date when the payment status changed to **settled.** Properties and constraints are:
  - Datetime field
  - Can be null
- **settled_amount:** the total amount settled for the payment. Properties and constraints are:
  - Float

        - Can be null
    - Each **payment method has its set of additional parameters**:

      Example 1: Credit Card

      ```
      {amount: 10, method: 'credit_card', number: '4111111111111111', name: 'John Doe', expiration_month: '10', expiration_year: '2042', cvv: '123'}
      ```

      Example 2: MBWAY

      ```
      {amount: 10, method: 'mbway', phone_number: '910000000'}
      ```

- Implement a RESTful API that supports the following requirements:
    - The API url pattern must be **/api/payments**;
    - List all the payments. Should return only the **base** attributes;
    - Get the details of a specific payment. Should return **all the instance attributes** (the base attributes and the set of additional parameters);
    - Search payments by **payment_id**, **amount** between ranges (e.g. amount > 10, amount < 20, amount > 2 and amount < 5), **payment method**, **status**, **created_at** and **settled at** between ranges (e.g. created_at > 2019/06/07)
    - Create new payments depending on the **available** payment methods. We can't manipulate (on create or update) the attributes **payment_id**, **created_at**, **status**, because they are the responsibility of the application;
    - Mark a payment as **settled** by setting the settled date and settled amount. If a payment is already settled we **can't mark it again as settled** (change the date or amount);
    - Delete payments;
- The Payments service **must run on port 1111**;

Refunds

- Create a data model that represents the Refund entity with the following attributes:
    - **refund_id:** represents the unique identifier of the refund. Properties and constraints are:
        - String
        - Unique
        - Auto generated

        - Maximum size of 200 characters
        - Cannot be null
    - **payment_id:** represents the unique identifier of an existent payment. Properties and constraints are:
        - String
        - Maximum size of 200 characters
        - Cannot be null
    - **created_at:** date when the refund was created. Properties and constraints are:
        - Datetime field
        - Auto generated on save
        - Cannot be null
    - **amount:** the total amount that we want to refund to a specific payment. Properties and constraints are:
        - Float
        - Cannot be null
- Implement a RESTful API that supports the following requirements:
    - The API url pattern must be **/api/refunds**;
    - List all the refunds;
    - Search refunds by **refund_id** and **payment_id**;
    - Get the detail of a specific refund;
    - Create refunds. The sum of all refunds of a specific payment **cannot exceed** the original payment amount. Note that **we can do multiple refunds** for the same payment;
- The Refunds service **must run on port 2222**;

Deliverables

- A Bitbucket private git repository with all the commits made during the development of this project. This repository should be shared from the beginning of development with the following users: andre.pires@switchpayments.com, filipe.gomes@switchpayments.com, david@switchpayments.com, simao@switchpayments.com, andre@switchpayments.com, and pedro.campos@switchpayments.com.
- A complete README file describing the API methods and requests examples (e.g. cURL requests, Postman requests)
- The project should be run by just:
    - (1) Cloning the repo;

    - (2) docker-compose up.
- Completely independent docker services as per described ahead.

Bonus Points

1. The Payments and Refunds shouldn't use each other's REST APIs, using Kafka or RabbitMQ as a data pipeline.
2. For security reasons the refunds microservice should only commit definitely refunds after X minutes (configurable in the application) of the received request.

Next Steps

There are no right or wrong answers. We want to be able to evaluate as much as possible how you tackle a problem and, after its completion, kickstart a f2f conversation with one of our lead engineers, where we will discuss:

- Scope and requirements understanding, and clear communication regarding expectations and deadlines;
- Work breakdown, execution and delivery;
- Bootstrap an application in Django;
- RESTful API design;
- Code structure;
- App deployment using Docker;
- Microservices interaction;
- Real world impact of sync/async transactions;
- What would be necessary to bring your project to production.