Bradoo - Python Developer

This document contains a problem we use to evaluate candidate skills. Read the proposal, and implement it the best you can, based on your understanding of it.

Problem proposal

In this test, you will develop a basic double-entry accounting system for depicting a retailer selling process. If you aren't aware of the basics of accounting, we will detail the basics below.

In accounting, every money transaction, ie, every time a certain amount of money flows, it's registered in at least two accounts, the account(s) from where the money comes from, and the account(s) to where the money goes; this is the principle of double-entry bookkeeping. Every time you subtract an amount of money from one account, you need to transfer that money into another account. This is done with positional information: either you are doing a debit into an account or you are doing a credit into an account.

In the double-entry system, you need to ensure that for a movement of money, the sum of both credits and debits equals zero. You can have any number of credits and debits into a move, if you ensure that the sum of all debits and credits equals zero, so you should have at least a debit and a credit inside a move.

For this problem, you are tasked with the responsibility of creating a system to create and manage the accounts, and to create the account moves that actually realizes the money transfer between the accounts, affecting their balances.

Modelling

In this test, you will develop a basic double-entry accounting system for depicting a retailer selling dynamic. For this, our system will consists of three objects:

Account: This model represents a money sum that is labeled with a name. This can be any storage of value within the company, like one bank account, the amount of sales of the year, the amount of tax due.

For our simplified accounting system, we will propose four types of accounts:

a) Liquidity Account - It will represent our company's bank accounts, that is real money
b) Receivable Account - an account that states that you have money to receive in the future
c) Revenue Account - an account that states revenue from our company sales, It basically represents the products that was sold to our customers
d) Expense Account - an account that states expenses for our company, like bought products and personal payroll

An account should also have a name.

Move Component: A move component holds data describing a change in an account's balance. It can have either a debit or a credit amount, never both. Credit and debit are always non negative.

Move: A move describes one or more flow of values between accounts through at least two move components. When posted, a move is guaranteed to have all its components credit sum equals to all its components debit sum. A move may have any number of components given our constraints. Also, a move component always has a related move.

That is, an account system is a closed system in total value, you can never generate a credit or debit amount without a counterpart in the system. A move should also have a datetime when it was posted (our accountant may set a past date time too) and a description field for our accountant control. If a move is deleted, its move components should also be.

As any move component necessarily relates to an account, our system will also give us the balance of each account. To compute the balance, we sum up the debit minus credit amount of all the related move components, this is for a *Liquidity Account*, *Receivable Account* or *Expense Account*. For a *Revenue Account*, we sum up the credit minus debit amount of all the related move components.

A use case scenario

During a sale, our company will be invoicing for the sold products. In our accountant's point of view, this makes our revenue account increases in it's balance, as sold products makes revenue, and also makes our receivable account increases in it's balance (note that our customer may not have paid for his order yet: through our invoice, we assume that our customer will be paying our company for the checked out goods).

When the customer finally pays for his order, based on his invoice, in our accountant's point of view the past increase in the receivable account will be balanced by a new move component that decreases the same account (this tells us that the customer doesn't own our company money anymore). Through the received customer money, we also increase some of our company liquidity account.

So the total balance change in your accounts was an increase in a liquidity account, telling us that our company has more money at its disposal, and an increase in a revenue account, telling us that products were sold.

Note that our customer could have paid our company with multiple payment methods, that could increase multiple liquidity accounts. Our accountant also could keep multiple revenue accounts, like one account for electronics sales revenue and another for housekeeping sales revenue. So that if our customer had bought a SSD and a scrub sponge, we would have an increase in both of the revenue accounts, summing up the total invoice amount.

The thing is, we can keep your account system increasingly complex, and this may give our company data to analyze which departments of products give the most profit (we could have expense accounts for depicting the products' cost), or which departments are increasing in sales through time.

In this dynamic we can notice two main moves between our accounts:

1) A move when we invoiced our sale, having two components, one that makes a debit in a receivable account (receivable accounts balance is debit minus credit) and one that makes a credit in a revenue account (revenue account balance is credit minus debit).
2) A move when the customer pays for our sale, having two components, one makes a credit in the same receivable account as the former, and one makes a debit in a liquidity account (liquidity account balance is debit minus credit)

Features

Given our description, you are expected to deliver a server-client application that will let our accountant:

1) List, create, edit and delete (watch out for our constraint about that) Accounts.
2) See the current balance of any account, and also its balance at a certain date.
3) See all the move components related to an account (you may take this view to show the account balance too).
4) List, create, edit and delete a move with its components, with datetime and description inputs. The move total amount should also be displayed.
5) Search moves through posted datetime and description.

Notes:

1) All List views should be paginated, our accountant doesn't like scrolling up and down a web page.
2) Your server side app should use Django or Flask. Also, all server calls should be done through a Json API. ie. The client application will consume the server side through an API.
3) You should use any RDMS using its own server. (eg. postgres, but not sqlite)

4) Your frontend should be a SPA, using any framework/library (eg. react, vue, bootstrap), or none at all.

How to submit your exam

1. Create a public or private git repository
2. Commit your code and any instructions needed to run the project into the repository
3. Deploy your project on a hosting service (we recommend Heroku);
4. Send a email to renato.sabo@bradootech.com and ravi.luz@bradootech.com with:
    - Your name
    - The address of the git repository
    - The location and required credentials where you deployed the project

Other Project Requirements

- Every text or code must be in English
- Write the project documentation containing:
    - Description;
    - Installing (setup) and testing instructions;
    - If you provide a docker solution for setup, ensure it works without docker too.
    - Brief description of the work environment used to run this project (Computer/operating system, text editor/IDE, libraries, etc).
- Provide API documentation (in English)
- Variables, code and strings must be all in English.

Bonus

- Provide a fully functional docker image of your application. We usually distribute our code in containers whenever possible
- Provide postman collection for the API part of this exam

Recommendations

- Practice the 12 Factor-App concepts;
- Use SOLID design principles;
- Use git best practices, with clear messages (the commit text can be in PT-BR)