Simple Credit Card API

In this document, we describe a simplified fictional credit card API that simulates money transaction events. This home assignment is only supposed to handle purchase transactions, but it might have to be able to handle other use cases like billing or payments in the future.

Contact

If you should have any questions throughout this interview, feel free to send an email to andreas.maier@capitalone.com. We will try to get back to you as soon as possible. Please keep in mind that we are in the Pacific timezone.

Description

Create an API for a lightweight balance calculation of a credit card product. In order to track purchases for customers, we would like to create an API that is capable of tracking account balances via double-entry bookkeeping. No worries, we will only use a simplified version of the accounting logic here and provide all the rules for ledger distributions. We want to be able to send several transactions simulating a credit card to the server and then be able to query the server for lists of transactions or account balances, while double-entry bookkeeping assures that all our calculations are balanced.

Double-entry bookkeeping is governed by the accounting equation. If revenue equals expenses, the following (basic) equation must be true:

```
Assets = Liabilities + Equity
```

For our Credit Card API, we will use a simplified version of double-entry bookkeeping. It will use a set of predefined journals, ledgers, (explained below) and rules to accommodate our transactions of our accounts.

For this stage of the app we also only want to be able to create new accounts and handle purchases.

Expectations

- Use any language, framework, and database you are most comfortable with
- Data needs to be persisted in a database. Keeping the values in memory is not enough. It would be assumed that our API runs on multiple instances when deployed
- We would like to receive the code in a git repository
- Provide a detailed readme of the steps neccessary to run your solution (step-by-step guide or a container solution like docker are both okay)
- We value testing of software not only to avoid regression but also as a form of documentation.
- Please put more attention to future extension of your code, models, and architecture rather than

premature optimization
- It is okay to use floating point math in your code. (Real banking/accounting systems use high precision arithmetic that is not necessary for an example like this one)

Requirements

Journals

A journal is a list of transactions for a single account. We currently only need to store purchase transactions but we might want to be able to store different transaction types (for example payments) in the future.

For now all we want to store in a journal is:

- transaction id
- transaction type (e.g. purchase)
- timestamp (when we received the transaction request)
- amount

Ledgers

A ledger is a bank's accounting view of transactions. For the purpose of this exercise, this is a list of debit and credit entries. We will only use very simple predefined ledgers and allocation rules.

Whenever we handle a financial transaction (purchase, bill, payment) it will be distributed via rules into the correct ledger.

```
Example Ledger:

debit | credit
------+-------
100   |
      |   100
      |
```

We want to be able to store debits and credits for a ledger as well as its corresponding transaction id that triggered this allocation.

Account Creation

Account creation should initialize all the domain objects needed (the journal and its ledgers) for an account.

Purchases

Once we have an incoming transaction, it will be recorded in the journal. Accounting rules will then determine into which ledgers money will get allocated, and whether the transaction is a debit or a credit. In our case, we already specified a simple set of rules for allocating the transaction amount to the correct ledgers.

Note that in order to comply with the accounting equation (assets = liabilities + equity), the total debits must equal total credits for each transaction. The purchase case honors this balance requirement as it will apply the amount as a debit in one ledger and as a credit in another ledger.

Ledger allocation rules (for this product we have 2 ledgers in total per account/journal. A “cash-out” ledger and a “principal”)

```
purchase:
    - debit: "cash-out"
    - credit: "principal"
```

For example, the amount coming in through a purchase would get applied to the “cash-out” ledger (the money that we have to send to the merchant) as a debit, and to the “principal ledger” (the money that we get from the customer) as a credit.

Endpoint requirements:

```
/health
    GET:
        ensure the server is up and runnning
        returns 200
/accounts
    POST:
        creates a new account
        initializes the journal and ledgers required for an account
        returns the account id for further usage
        response body: {
            id: int or string (your preference)
        }
    GET: ( /accounts/:id )
        returns an existing account
        returns the outstanding principal for an account (being the sum of all the credit entr
        fetches all transactions for an account and returns them in a list ordered by time
        response body: {
            id: int or string
            principal: double
            transactions: [
                {
                    id: int or string (your preference)
                    type: string
                    timestamp: string
                    amount: double
                },
                ...
            ]
        }
```

```
/transactions
    POST:
        applies an incoming transaction
        records the transaction in the correct journal
        applies the given rule to allocate the transaction in the correct ledgers
```

All the endpoints should return status codes that make sense, even in error cases.

Example Flow

Here is an example that could also be used as acceptance criteria within an end-to-end test:

- create account1
- $200 purchase for account1
- create account2
- $2000 purchase for account2
- $500 purchase for account1
- $750 purchase for account1
- $2500 purchase for account2
- get account1, returns a balance of $1450 and the transactions list should have a length of 3 and be in the correct order
- get account2, returns a balance of $4500 and the transactions list should have a length of 2 and be in the correct order

(To extend this example, the database should have 2 journals, one for each account. There would be a total of 4 ledgers, 2 for each journal)

For clarity, this is how the ledgers would look like at the end of this example:

```
Account 1:                          | Account 2:
                                    |
cash-out:          principal:       |   cash-out:          principal:
debit | credit     debit | credit   |   debit | credit     debit | credit
------+-------     ------+-------   |   ------+-------     ------+-------
200   |                  |    200   |   2000  |                  |   2000
500   |                  |    500   |   2500  |                  |   2500
750   |                  |    750   |         |                  |
```

Closing Notes

And that’s it. It is just a REST API with a little server and a database to persist the values. Have fun!