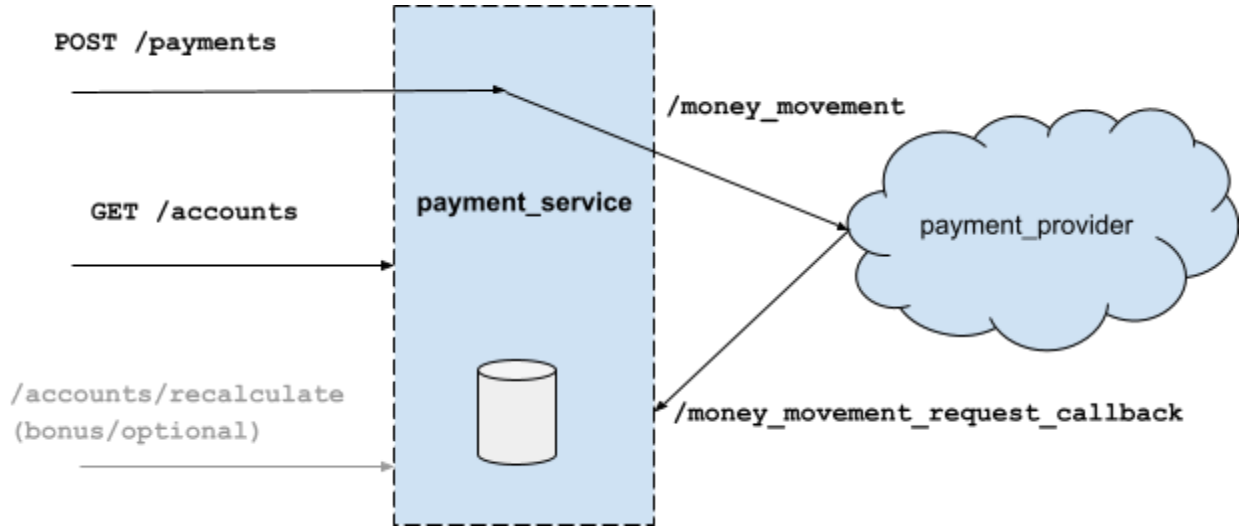
Cushion Coding Project: Payment Service

The goal of this project is **to design and build a small backend of a payment service**. No UI will be necessary, and we provide the schemas for all components involved. Result: a small running service that implements a few basic endpoints.

We've designed this project to be similar to what we do in real life and not contain any algorithmic puzzles.

Please use the language and the environment you are comfortable with. We can accommodate popular languages (see last section for details).



We will ask you to upload the code for this project (to get a running `payment_service`) into AWS Beanstalk, and will provide all accounts and help to do that.

1. `payment_provider`

This is a service that we provide for you as a black box, live service running at: https://bxzggmcc.cushionai.com

You can think of this service as a "payment provider" such as Stripe, that takes your calls to move money on behalf of customers. It is very simple and has only one endpoint and one callback from it.

The purpose of `payment_provider` is to accept posts to the **`/money_movement/`** endpoint. As per the documentation below, it takes:

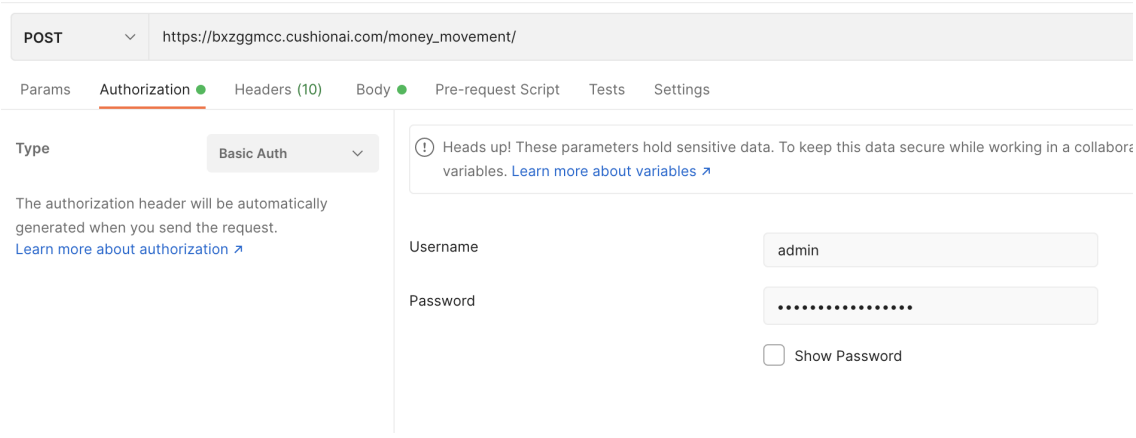
- `amount_cents` (could be negative)
- `money_movement_id` - this is a unique ID that you can create to identify this payment, so that later you can match the callback to it,
- `time_to_settle_seconds` - this is the maximum time you can expect a callback from `payment_provider` that will contain the results of that money movement request.
    - Your money movement request can change status (e.g., **settle** to `status='posted'` - which must be reflected in account's `settled_balance`), or even change amounts (that happens with real payments providers!), any number of times until the `time_to_settle_seconds` expires.
- other fields (`account_id, merchant_name`) are mostly just for bookkeeping, you'd use them in real life to identify the particular customer account and the payment destination, for example, a merchant.

That's it for `payment_provider`! It will send the callback to the IP of the originating request and port 80 no SSL (to keep things simple for a temporary service), so next, we will work on the actual `payment_service` and will start that service in AWS (help will be provided) so that the two services can talk to each other.

Detailed documentation:
Here is the OpenAPI definition of that endpoint in human-readable format:
https://bxzggmcc.cushionai.com/schema/
Here are also yaml and json definitions, just in case (e.g. to auto-generate a client)

If any of these servers prompt for a login, it is:
user: **admin**, password: **23f3ee13-635e819b**

You will also need to use these credentials via Basic Authentication when connecting to that service from your code. This is what it looks like in, for example, in Postman's Authorization tab (just for testing):



2. payment_service

This is the service that we ask you to build. It receives POSTs to /payments (from, for example a UI, **which we will not build here**), makes corresponding requests to `payment_provider` and processes the results - to keep track of account balances and account status (for "recalculate").

You will need to store data between the calls to do the math, so some kind of persistence will be necessary. One quick way to do this is to connect to `db.sqlite3` - a simple SQL database. Yes, it will disappear if something happens to the EC2 instance, but for the purposes of this task it should suffice. You are welcome to use any other DB engine (any SQL supported by e.g. RDS or even NOSQL supported by AWS), but you would have to configure it on your own and have it be available to `payment_service` on Beanstalk.

Detailed docs are below, but the most important parts of `payment_service` that need to be built are:

- POST to /payments - just to initiate a payment on a particular account
    - For simplicity, **we can assume just one account and one user in the system. No authentication is necessary, since it's just one user**.
    - This POST does some bookkeeping and calls the `payment_provider`.
- GET /accounts - just to print out the accounts (assume one for now), the following fields are critical on an account:
    - `balance` - this is the sum of all payments (positive or negative) that you have created for that account, whether those payments have been verified as `posted` by the `payment_provider`'s callback or not.
    - `settled_balance` - this is the sum of all payments that have been reported as `posted` by the `payment_provider`.
    - `status` - account can be current or delinquent. The status is refreshed when the optional `recalculate` endpoint (see below) gets called

- `money_movement_request_callback` - as we mentioned above, this is the resulting callback from the stuff you sent to the `payment_provider`. Within a certain time window that you specify (default 10 seconds) you can receive any number of callbacks for this money movement ID, with a different status (for example, "pending" changed to "posted"), or the provider can even adjust the amount of this payment.
    - you need to carefully record the results and update the corresponding account fields (`balance, settled_balance`), so that if someone calls the `/accounts` endpoint, you would report all the up-to-date amounts.
    - Please consider what happens if you receive multiple payment requests and need to handle multiple callbacks for each, and what parts of your code need to be prepared to handle that.
- BONUS(optional): `/accounts/recalculate` - this endpoint is given a time window (say, 10 seconds or 10 days, doesn't matter) and for that time window, if the account's

`settled_balance` has been permanently in the negative (since balances can be negative), you need to set its status to "`delinquent`". So you might need to store each payment that happened on that account, but that is up to you how to model that.

Detailed documentation on the schema for this `payment_service` is here:
https://app.swaggerhub.com/apis/one10/payment-service_rest_api/v1#/
(In addition to this human-readable link, Swaggerhub also allows you to export the schema in YAML, JSON and even auto-gen in some programming languages, which may or may not save some time.)

Please follow this schema carefully. The callback from `payment_provider` will come in this exact format, so if you don't follow the schema, it may not work, same for other endpoints.

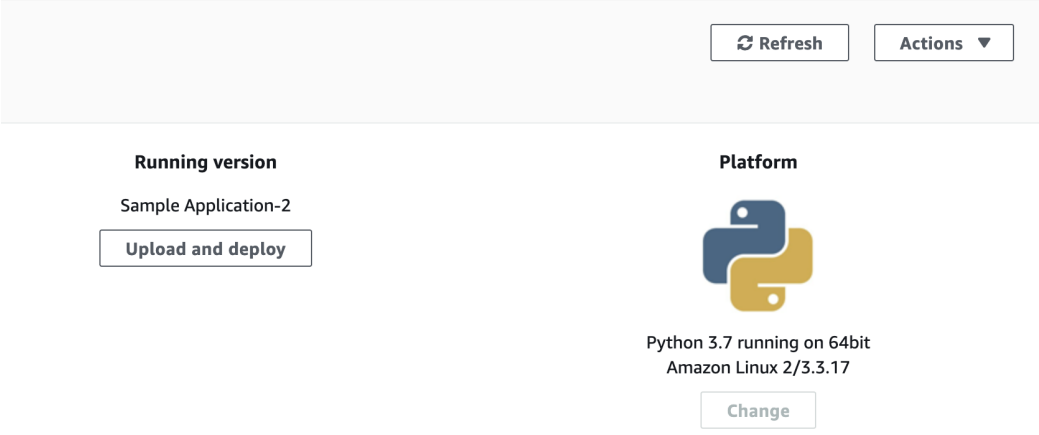
3. How to deploy and test

You can start creating the `payment_service` in your favorite environment on your local system/IDE, and use your favorite API tool (Postman, curl, etc) to make test calls to your `payment_service`.

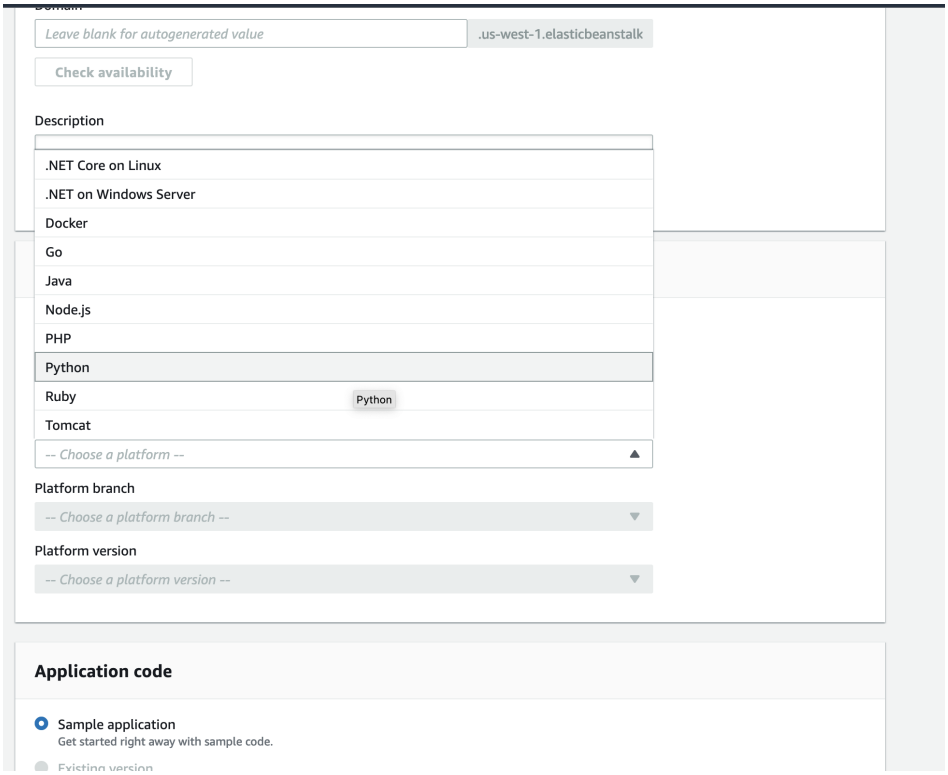
In order to test a real `money_movement_request_callback` when you are ready (in addition to local curl) we've prepared this live endpoint that will call you back at your requesting IP (see BA username/password from Section 1 of this doc for authentication):

https://bxzggmcc.cushionai.com/money_movement/

Since these two services will need to call each other, we ask you to deploy your code to AWS Beanstalk (with a port 80 listener, which comes by default). We will prepare all accounts and show how to do that. Deployment to Beanstalk can be as simple as zipping up code, specifying the main script (e.g., `payment_service_config.wsgi:application` for Python) and hitting the "Upload code" button:



Beanstalk can support other languages (see screenshot below). We can accommodate them, as long as your code works with the schemas that we've listed here.



For Python (configuration we tested was Python 3.7 running on 64bit Amazon Linux 2/3.3.17), you only need to include packages you need in `requirements.txt` at the top level with your code (see below) and Beanstalk will automatically install them during code upload.

If you choose a different stack, you'd have to research a bit how other platforms are configured on Beanstalk. There are always system commands that can be run via `.ebextensions` (also see below), but that is an advanced option that in most cases is not necessary.