Payment Terminals Service Project

Content:

This document describes and contains the following:

- Installation guide
- How to run tests
- Terminals API
- Accessing the API
- Sequence diagram
- Pre-code plan
- Future plan

Installation Guide

1. get source from:
   git clone https://github.com/AvshalomP/SmartPay.git
2. run makefile
3. activate HTTP server: ./runServer

How to run Unit Tests:

Each unit test can be executed separately as well as we can run all unit tests all together.

Run solo unit test:

- cd test directory
- make
- ./test <TEST_NAME>

Run all unit tests:

- cd test directory
- make runAll

Terminals API

Description:

This service is to manage payment terminal equipment and can do the following:

- Create new terminal
- Get specific terminal details
- Get all stored terminals' details

Resource – The service consists one resource, named "terminals", that is strict to the described below specs.

Threads (Modules) – The program has 2 main modules that are represented as threads

- Connection thread – to handle the new HTTP requests
- Equipment thread – to handle the Read/Write tasks to the DB (in our case linked list)

API specs:
The API consists of 3 endpoints for CRUD operations:

1.
URI: **/api/terminals/**
method: **POST**
Description: Create a new terminal
Content Type: Json
Authorization: Token 0Auth1 based in header.

Request:
```
{
        “card_types”: <list(string)>,
        “transaction_types”: <list(string)>
}
```

Response:
```
{
        “terminalID”: (number) - The created terminal ID,
        “error”: (string or null),
}
```

2.
URI: **/api/terminals/<terminal#>**
method: **GET**
Description: Read details of existing terminal
Content Type: Json
Authorization: Token 0Auth1 based in header.

Request:
```
{
        <empty>
}
```

Response:
```
{
        “terminalID”: (number) - The created terminal ID,
        “transactions”: [
        {
                “card_types”: <(string)>,
                “transaction_types”: <(string)>
        },
        {
                …
        }
        ]
        “error”: (string or null),
}
```

3.
URI: **/api/terminals/**
method: **GET**
Description: Read details of all existing terminals
Content Type: Json
Authorization: Token 0Auth1 based in header.

Request:
{
<empty>
}

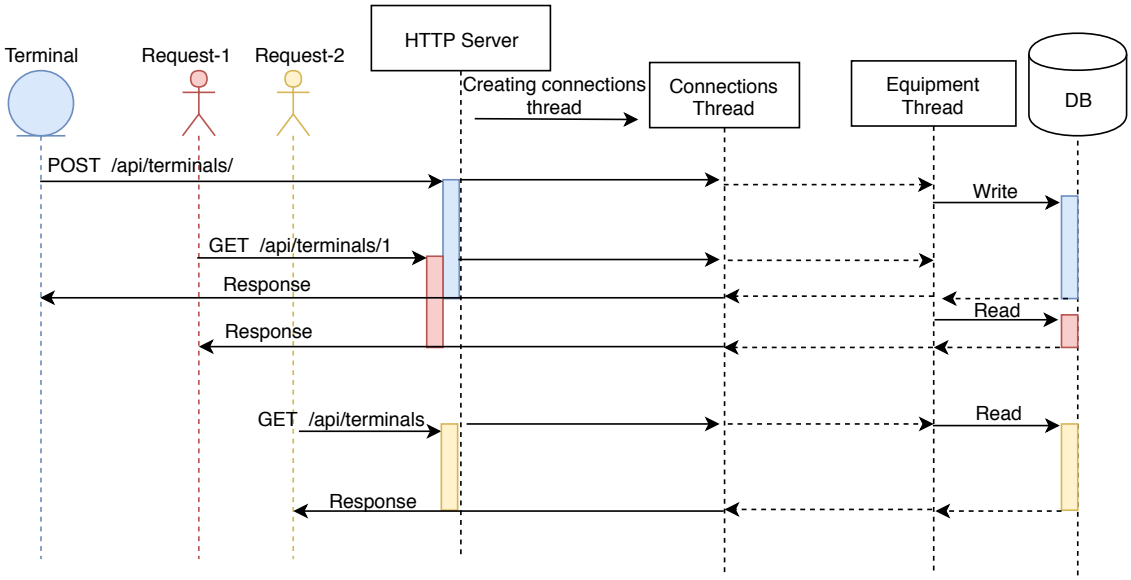
Response:
{

[
{
“terminalID”: (number) - The created terminal ID,
“transactions”: [
{
“card_types”: <(string)>,
“transaction_types”: <(string)>
},
]
},
“terminalID”: (number) - The created terminal ID,
“transactions”: [
…
]
}
]

Accessing the API

- Default server route will run on http://localhost:8888/ unless specified in arguments otherwise
- CRUD operations usage with curl:
  - POST - create new terminal:
    ```
    Curl -X POST -H "Content-Type: application/json" -d "{ \"key1\": \"value1\" }" http://localhost:8888/api/terminals/
    ```
  - GET - specific terminal:
    ```
    curl -X GET -H "Content-Type: application/json" http://localhost:8888/api/terminals/<term#>
    ```
  - GET - all terminals:
    ```
    curl -X GET -H "Content-Type: application/json" http://localhost:8888/api/terminals/
    ```

Sequence Diagram:



Pre-code Plan:

1. Building Hello World program to integrate both cunit and libmicrohttpd libraries.
2. Building skeleton to handle simple GET/POST with no implementation.
3. Implementing our 3 endpoints 1 by 1 along with unit test each implementation.
4. Making the program multithreaded and thread safe
5. Dealing with basic http authentication
6. Error handling

Future plan:

- Finish implementing writeToDb and performing end to end test with post.
- Adding PUT request to handle the updates from every transaction made in stored terminals.
- Error handling.
- Code robustness through improved unit-testing.
- Making the program multi-threaded in terms of thread per-connection and performing a read/write task to DB simultaneously – to improve efficiency.
- Add authentication to secure the http requests.
- Managing real DB for scalability.
- Adding DELETE option for removing terminal.