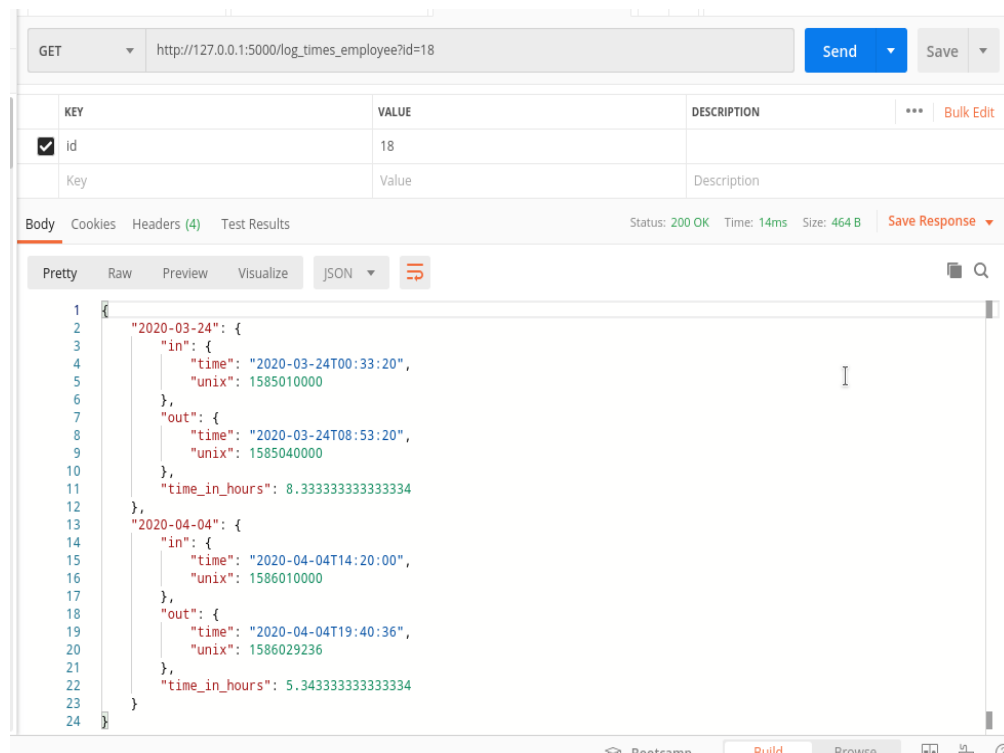


Lab 5, part 1

Intro

My RFID application is a monolith made of two important parts. The first part is RFID reader itself, which reads data and saves them to database, and the second one is a Flask server being a simple interface to fetch/edit user data in the application. The API is of REST type. Every information is saved in PostgreSQL database.

The application purpose is to track employees' working time. Each employee has the id card (called 'badge') with RFID number encoded in it. Each employee has one's own card so based on card uid (unique identifier) it is possible to track when employee comes and leaves work. The working time is generated into such report obtained from the application api:



Saved data:

The application has exactly two tables *users* and *card_logs*.

Users table stores name, last name and assigned badge uid of an employee. Uid is reflected as a string. Moreover the table is indexed with automatically incremented field *id*. The table declaration looks like this:

```
"CREATE TABLE IF NOT EXISTS users(id SERIAL PRIMARY KEY,u_name VARCHAR(20), l_name VARCHAR(20), uid VARCHAR(100));"
```

The *card_logs* stores the uid and time of card log. The time is stored as unix timestamp. Timestamp is the easiest way to store time very precisely and is easily parsable by python. The declaration of the table looks like this:

```
"CREATE TABLE IF NOT EXISTS card_logs(id SERIAL PRIMARY KEY,uid VARCHAR(100), log_time BIGINT);"
```

Read data:

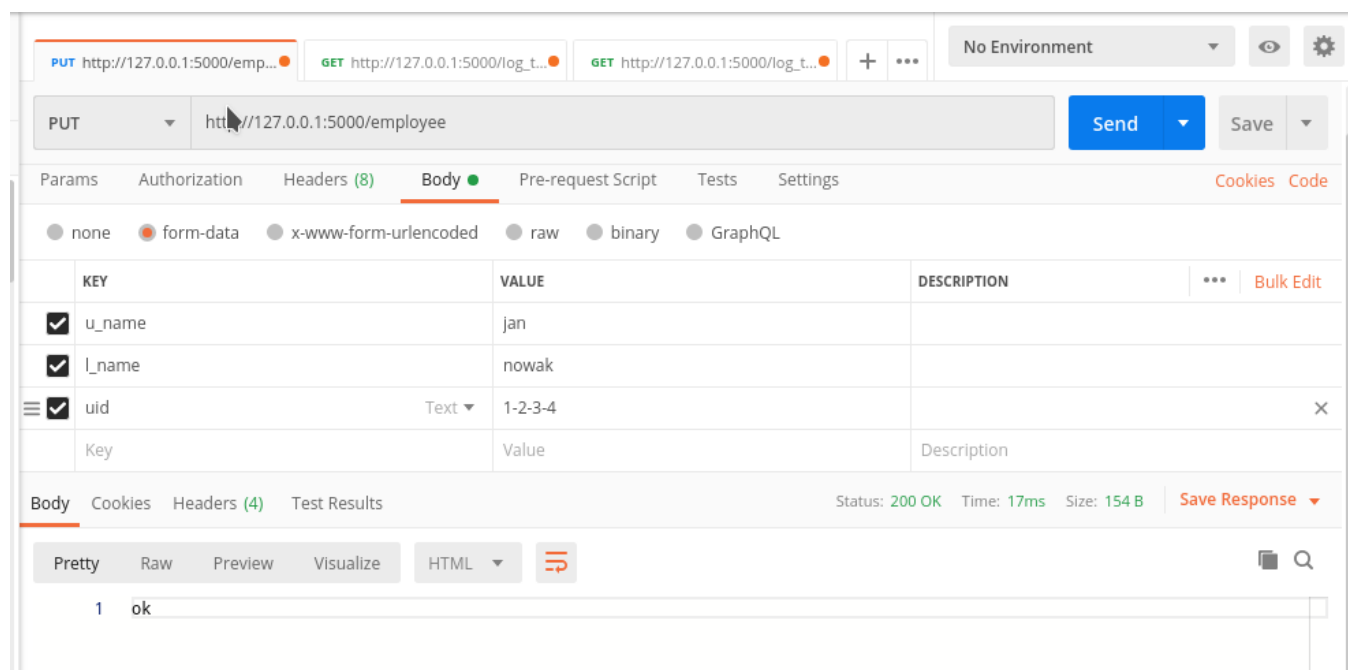
The data is read using MFRC522 driver the reader stops reading if CTRL+C or red button (on RaspberryPi) is pressed. Each time, when the badge is read, the reading time (as unixtime) and badge uid is saved to the database as new record.

NOTE: This part I haven't tested due to lack of proper equipment

Employee management

In order to manage employees the simple server is running. This part will be separated in part 2 of the laboratory. There are the following endpoints:

PUT /employee – adds new employee. It is required to specify in JSON, the following fields: *u_name*, *l_name*, *uid*. From those fields employee name, last name and card uid are read. Example using Postman:



GET /employee – gets the list of all employees

GET http://127.0.0.1:5000/emp...

GET http://127.0.0.1:5000/log_t...

GET http://127.0.0.1:5000/log_t...

+

...

No Environment

👁

⚙

GET

http://127.0.0.1:5000/employee

Send

Save

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Code

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input type="checkbox"/>					
<input type="checkbox"/>					
<input type="checkbox"/>					
	Key	Text	Value	Description	

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 7ms

Size: 267 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

🔍

```
1 {
2   {
3     "id": 18,
4     "l_name": "nowak",
5     "u_name": "jan",
6     "uid": "1-2-3-4"
7   },
8   {
9     "id": 19,
10    "l_name": "kowalski",
11    "u_name": "jan",
12    "uid": "1-2-4-4"
13  }
14 }
```

PATCH /employee – by providing *e_id* (employee id as shown in GET) and new or empty *uid* the system edits the users *uid* (assigns new or leaves the field empty).

The image shows two screenshots of the Postman API client interface. The top screenshot displays a PATCH request to `http://127.0.0.1:5000/employee` with a status of 200 OK. The request body is a JSON object with `e_id` and `uid` fields. The bottom screenshot displays a GET response to `http://127.0.0.1:5000/employee` with a status of 200 OK. The response body is a JSON array containing two employee objects.

Top Screenshot: PATCH Request

Method: PATCH
URL: `http://127.0.0.1:5000/employee`
Status: 200 OK
Time: 17ms
Size: 154 B

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> <code>e_id</code>	18	
<input checked="" type="checkbox"/> <code>uid</code>	f-f-f-f	
<input type="checkbox"/>		
Key	Value	Description

Body: `ok`

Bottom Screenshot: GET Response

Method: GET
URL: `http://127.0.0.1:5000/employee`
Status: 200 OK
Time: 7ms
Size: 267 B

Body (JSON):

```
1 {
2   {
3     "id": 19,
4     "l_name": "kowalski",
5     "u_name": "jan",
6     "uid": "1-2-4-4"
7   },
8   {
9     "id": 18,
10    "l_name": "nowak",
11    "u_name": "jan",
12    "uid": "f-f-f-f"
13  }
14 }
```

DELETE /employee – by providing e_id to this endpoint, the employee is deleted

The screenshot shows a REST client interface with a DELETE request to `http://127.0.0.1:5000/employee`. The request has a query parameter `e_id` with the value `19`. The response is `200 OK` with a status of `200 OK`, a time of `15ms`, and a size of `159 B`. The response body is `deleted`.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> e_id	19	
<input type="checkbox"/>		
Key	Value	Description

Body: `deleted`

The screenshot shows a REST client interface with a GET request to `http://127.0.0.1:5000/employee`. The response is `200 OK` with a status of `200 OK`, a time of `8ms`, and a size of `205 B`. The response body is a JSON object:

```
{  "id": 18,  "l_name": "nowak",  "u_name": "jan",  "uid": "f-f-f-f"}
```

GET /health-check – this endpoint is only used to check whether server is running. If it is, it sends “health ok”.

The screenshot shows a REST client interface with a single tab for the endpoint `http://127.0.0.1:5000/health-check`. The request method is `GET`. The response status is `200 OK` with a time of `13ms` and a size of `161 B`. The response body is displayed in the 'Body' tab, showing the text `health ok` in 'Pretty' format.

KEY	VALUE	DESCRIPTION
Key	Text	Value
		Description

Body Cookies Headers (4) Test Results Status: 200 OK Time: 13ms Size: 161 B Save Response

Pretty Raw Preview Visualize HTML

```
1 health ok
```

GET /log_times_employee – this endpoint gets the report of when did the employee come to work and when did the one left the work

The screenshot shows a REST client interface with a single tab for the endpoint `http://127.0.0.1:5000/log_times_employee?id=18`. The request method is `GET`. The response status is `200 OK` with a time of `14ms` and a size of `464 B`. The response body is displayed in the 'Body' tab, showing a JSON object in 'Pretty' format.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Id	18	
Key	Value	Description

Body Cookies Headers (4) Test Results Status: 200 OK Time: 14ms Size: 464 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "2020-03-24": {
3     "in": {
4       "time": "2020-03-24T00:33:20",
5       "unix": 1585010000
6     },
7     "out": {
8       "time": "2020-03-24T08:53:20",
9       "unix": 1585040000
10    },
11    "time_in_hours": 8.333333333333334
12  },
13  "2020-04-04": {
14    "in": {
15      "time": "2020-04-04T14:20:00",
16      "unix": 1586010000
17    },
18    "out": {
19      "time": "2020-04-04T19:40:36",
20      "unix": 1586029236
21    },
22    "time_in_hours": 5.343333333333334
23  }
24 }
```

How to run the application

In order to run the application, you need to have PostgreSQL installed and running. The configuration regarding the database, user, host and passwords are stored in `db_management.py` file. You need to edit them before running to suit your set-up. Apart from it, all modules from `file_dependencies.txt` need to be installed using `pip` and `python` in version 3.6 or higher. Then, using terminal, use `cd` to go to directory where the application is, and `index.py` is stored. First, export an environment variable . In RaspberryPi or Linux machine;

```
$ export FLASK_APP=index.py
```

And afterwards

```
$ flask run
```

The application should be running like this:

```
* Serving Flask app "index.py"
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```