Complaint API Application

Author: Alessandro F. Martins

Version: 0.1b

Introduction

This is an application for submission to *Reclame Aqui,* which consists on a simple Restful API application to manage complaints, which includes:

- Full CRUD capabilities: create, read, update and delete complaints
- Three search modes for the GET method: exact match, "*like*" match (using parts of the desired attribute value, case insensitive) and distance range search
- Exact and "*like*" searches can be combined
- Google Maps- and MongoDB native-based geolocation capabilities
- JSON-based document format, MongoDB-based persistence
- Dockerfile for building containerized application.

Installation

Prerequisites

The following software was used in the elaboration of this application:

- Python 3.6.5
- pip 10.0.1
- Nginx 1.14.0
- virtualenv 16.0.0 (plain version)
- MongoDB 3.6.5 (plain version) and 3.4.1 (dockerized version)
- Linux: Ubuntu 18.01 (plain version) and Alpine 3.7 (dockerized version)

Python packages versions used in Complant API application can be found in the *requirements.txt* file.

Installation steps

1. Clone this repository to a working directory (let's say, *app_dir*):

```
$ mkdir app_dir
$ cd app_dir
$ git clone https://github.com/alessandro-f-martins/complaint_api_app .
```

2. Create a Python virtual environment for this project (please refer to any related tutorials on the Internet. There is a number of ways to organize virtual environments, which depend on system organization, so please feel free to choose the one which is best for you).
3. Install the needed Python packages using the provided `requirements.txt` :

```
$ pip install -r requirements.txt
```

4. Modify the following configuration files. They contain references to the docker internal directory structure (`/usr/src/app` for the main application directory), and should be changed for running in your environment:
   - `app_vars.cfg` :

     ```
     line 1: # . ../../venv/bin/activate # --> change it to point to the 'activate' script
     of your virtual environment
     ```

   - `runApp.sh` :

     ```
     line 2: export APP_DIR=/usr/src/app # --> change it to point to your app_dir
     ```

   - `http/nginx.conf` :

     ```
     line 2: pid /usr/src/app/http/nginx.pid; # --> change it to point to your app_dir
     line 41: access_log /usr/src/app/http/access.log; # --> change it to point to your
     app_dir
     line 42: error_log  /usr/src/app/http/error.log # --> change it to point to your
     app_dir
     ```

   - `db/init/mongod.conf` :

     ```
     line 3:    path: "/usr/src/app/db/log/mongod.log" # --> change it to point to your
     app_dir
     line 9:    dbPath: "/usr/src/app/db/data" # --> change it to point to your app_dir
     ```

5. Go to the *api_dir* directory and run the application with `sudo` :

   ```
   $ cd app_dir
   $ sudo ./runApp.sh &
   ```

   Once running, the server listens on HTTP port 80.
6. To deactivate the application, run `closeApp.sh` with `sudo` :

   ```
   $ sudo ./closeApp.sh
   ```

Configuration

Complaint API Application configuration is done in two ways:

- in the plain version: in the *app_vars.cfg* file
- in the dockerized version: in the *ENV* section of *Dockerfile*

The file holds configuration environment variables for the Flask and MongoDB subsystems and for the application itself.

Logging

You can activate/deactivate application logging in the configuration files (please see *Configuration*):

```
export API_LOG_FILE=$LOG_DIR/api-log.log
export API_LOG_ACTIVE=1 # --> change it to 0 to deactivate logging
```

Testing

There are some examples of testing data in the `app_dir/tests` directory. They can be run using the `testscript.sh` curl-based Bash script:

```
$ ./testscript.sh
```

During development, Postman was used for ease of operation. Please feel free to analyze the `tests/entrynn.txt` test data files to produce test data of your own.

About the application

The API Routing and Data Engine

The Complaint API application is based upon *Flask* and *Flask-RESTful* frameworks to provide all the underpinnings of request routing, HTTP method association, argument parsing and REST object manipulation. For more details, please refer to Flask-RESTful website.

Flask-PyMongo was also used for providing connectivity to the MongoDB database.

The *Complaint* document format

The *complaint* JSON document has the following format:

```
{
    "complain_id" : 1,
    "title" : "string",
    "description" : "string",
    "locale" : {
        "address" : "string",
        "complement" : "string",
        "vicinity" : "string",
```

```
        "zip" : "string",
        "city" : "string",
        "state" : "string",
        "country" : "string",
        "geo_location" : {
            "type" : "Point",
            "coordinates" : [0.0, 0.0]  # float(longitude), float(latitude)
        }
    },
    "company" : "string"
}
```

Two points to mention about the `locale.geo_location` attribute:

- Its type is *GeoJSON*, a format used for encoding a variety of geographic data structures. For more information, please refer to GeoJSON.org.
- The document shown above is how it is stored in the MongoDB database. It is omitted in any REST operations, as they are produced and stored for geolocation purposes only (see *Working with Geolocation* section).

The *complain* module and REST functionalities

The *complain* module, under the *resources* package, contains the definitions of the `Complain` and `ComplainList` classes. The first one contains methods for handling URIs which are related to a single, unique complaint object, and therefore have the *id* of this object as its last part:

```
$ curl http://myserver/complain/5   # HTTP GET: Retrieves complaint object whose id is 5

$ curl -X DELETE http://myserver/complain/9   # HTTP DELETE: Removes complaint object whose id
is 9

$ curl -d 'complain={"company":"Umbrella%20Corp."}' -X PATCH \ http://localhost/complain/5 #
HTTP PATCH: Modifies the company attribute of complaint object whose id is 5

$ curl -d "@new_entry.txt" -X PUT http://localhost/complain/3  # HTTP PUT: Replaces the whole
content of complaint object whose id is 3 by the complain document contained in file
"new_entry.txt" (see below)
```

The second class (`ComplainList`) contains methods for handling URIs which don't relate to a specific object, either potentially bringing a complete list of complaint documents in the database, a list that match a query string, or creating a new object with a system-assigned *id*:

```
$ curl -d "@new_entry.txt" -X POST http://localhost/complain  # HTTP POST: Replaces the whole
content the company attribute of complaint object whose id is 3 by the complain document
contained in file "new_entry.txt" (see below)

$ curl http://localhost/complain # HTTP GET: Retrieves all complaints in the database

$ curl http://localhost/complain?title=Complain%201 # HTTP GET: Retrieves complaints whose
'title' attribute matches "Complain 1" exactly (whole text, case sensitive)

$ curl http://localhost/complain?city_like=paulo # HTTP GET: Retrieves complaints made in a
city which contains "paulo" in its name (partial text, case insensitive)
```

For reference, here is the content of a *new_entry.txt* file, as used by the *PUT* and *POST* examples above:

```
complain={
 "title" : "Complaint 4",
 "description" : "I am the fourth complaint",
 "locale" : {
   "address": "Av. Paulista, 1500",
   "complement": "3rd floor, room 345",
   "vicinity": "Bela Vista",
   "zip": "01310-100",
   "city": "São Paulo",
   "state": "SP",
   "country": "Brazil"
 },
 "company" : "Damage Inc."
}
```

Of course, these are `curl` -based usage examples, and the user should call the API methods according to the REST conventions of the client language or application.

Querying with the GET method

As mentioned, there are three query modes that can be made with the query string-using GET method:

- *Exact match queries*: using the name of any attribute directly in the search query string
- *"like" queries*: appending "_like" to the name of any attribute the search query string
- *Distance Range Search queries*: using the following query string fields:
  - *id*: id of the complaint to be queried
  - *within_radius*: radius in *meters* (range) where to look for other complaints

  Therefore, the following REST method call:

  ```
  $ curl http://localhost/complain?id=5&within_radius=3000
  ```

  returns all complaints that were made within a 3km-radius from where the complaint whose id is 5 was issued (excluding complaint #5).

Please refer to *The complain module and REST functionalities* section for usage examples.

Working with Geolocation

Complaint API Application uses Google Maps API to get the geographic coordinates of a complaint and, as such, relies on the availability of this service to provide this functionality.

The latitude and longitude values are obtained from a concatenation of the *locale.address* , *locale.city*, *locale.state* and , *locale.country* attributes, so they are the only relevant ones for geolocation determination.

You can activate/deactivate the geolocation functionality in the configuration files (please see *Configuration*):

```
export USE_GEOLOC=1  # --> change it to 0 to deactivate geolocation
```

NOTE: in the present version, modifying any *locale* attribute requires the whole *locale* sub-document to be rewritten and sent (see *Under work* section).

Handling address mistakes

When retrieving the [longitude, latitude] values of a query Google Maps API tries to identify the geographic center of the *minimal coherent fragment* of the full provided address, from the smallest identifiable region to the largest. For instance, if the provided query has the following content:

```
locale: {
  "address": "xxxx",
  "complement": "",
  "vicinity": "Morumbi",
  "zip": "12345-123",
  "city": "São Paulo",
  "state": "SP",
  "country": "Brasil"
}
```

, the Google Maps API will ignore the *address* attribute and provide the coordinates of the geographic center of São Paulo City:

```
{'lat': -23.5505199, 'lng': -46.63330939999999}
```

If all relevant query attributes are incorrect, the result is a spurious, but syntactically valid geolocation, resulting from the attempt to find a *Point of Interest* whose name matches the incorrect string:

```
locale: {
  "address": "xxxx",
  "complement": "",
  "vicinity": "Morumbi",
  "zip": "12345-123",
  "city": "xxxxx",
  "state": "xxxx",
  "country": "xxxx"
}
```

The result from the related query is:

```
{'lat': 38.573334, 'lng': -121.507788}
```

, which, by accident, matches the geolocation of the *XXXX Audio Systems* store, in Sacramento, CA, USA.

This behavior makes it extremely difficult (if not impossible) to apply correct exception handling in the case of wrong addresses, so assuring the correct address in needed *before* calling the related *POST*, *PUT* or *PATCH* methods.

Dockerized version

An example Docker image of the application is provided in Docker Hub:

```
$ docker pull afmartins/complain_api_app
$ docker run -p80:80 afmartins/complain_api_app
```

This image was built using the provided *Dockerfile*. It is self-contained for testing purposes only, as it is not currently using Docker Volumes for data persistence and sharing among Docker instances (Services). (see *Under Work* section).

Under work

There are still some functionalities we wish to improve further:

- Include attributes such *created_at:* and *complainer_name:* .
- Allow modification of *locale* single attributes when using *PATCH* method.
- Currently, the dockerized application provided as example keeps everything inside its own container, including its database files, as it is meant for testing. In a production environment, Docker Services with Docker Volumes should be used, so data may be persisted and multiple containerized instances of the application may persist and have access to the data. Please feel free to change the provided *Dockerfile* accordingly.
