Lab Assignment 5: Writing a web service

In this assignment you will expand on the small web service we saw in class. In particular you will have to read and understand existing code, and extend/adjust it to meet your goals.

This assignment is more complicated. In particular, we will also be introduced to maintaining a project via git. You should start by cloning the project via the following steps:

- Open up the terminal to the location you want to work on. We will automatically created a subfolder in there.
- Do git clone https://github.com/skiadas/messaging-flask.git. This should download the whole project folder and create a folder called messaging-flask right where you are.
- Go into that folder via cd messaging-flask.
- You can now open the whole folder in SublimeText via subl .. There are a number of files in this folder that you need to be acquainted with.
- First you will find a specs folder with three documentation files. We have seen some of those files in class already. They describe the project in some detail. You will in particular need to study the resources.md file that contains information about the different routes and their expected results.
- The files you will work on are all in the app folder.
- You will want to have GitKraken open on this project. When you have completed a solution for one of the problems, you want to review the changes in GitKraken, and then stage them and create a commit. The message of the commit should include the number of the assignment problem that the changes in this commit address.
- In order to use the server, you will need to tell it about your database. You will do so by creating a file called keys.json in the app folder. It should look like this:
  json { "USERNAME": "username here", "PASSWORD": "database password here", "SERVER": "vault.han
- You will need to have two terminals open, both at the messaging-flask folder. The one terminal will run a "server". You start that server with the command python app/messaging.py. You periodically will want to shut the server down with Ctrl-C and then restart it.
- The other terminal is your "client" that you can use to test your server. You will start a python session with python3, then run import requests to load a very popular requests package. Now you can perform requests with commands like r = requests.get('http://127.0.0.1:5000/users/haris').
- In the app folder are the files you will need to work on and add your code. You will want to study those files and try to understand what they are doing right now.

- messaging.py is the main controller file that kickstarts everything. It starts a "Flask" application, sets up some database information, and then contains a list of the various routes, along with the methods that handle them. When you need to implement a new route, or change the fundamental behavior of a new route, you will need to work on this file.
- message.py is a helper file that is meant to contain methods useful when processing messages. You can access it from within messaging.py via the message variable. If you need to do some validation of your inputs, or some other work related to your data, this is a good place to put that.
- db.py is a helper file that implements the various database interactions. You will be writing some sqlalchemy code in here. Whenever you need to get something from or write something to the database, you want to write the corresponding function here, and access it from messaging.py via the db variable prefix.

Specific problems follow. You should only attempt these after you have a good understanding of how the existing code in the file works.

1. The handling of GET users/<username>/messages currently lets the system handle the case where there is an error during the database access. This results in inappropriate error messages being sent back to the client. Adjust the code so that it handles database-access errors, similarly to how the corresponding POST implementation does.

   - You will need to locate the parts of the relevant db.py function that actually call the database, and protect them around a try-except block. Examples of such blocks should appear elsewhere in the file.
   - You will need to return a specific value from that db.py function back to the route handler in messaging.py to indicate that database error, and you would need to add some code in that handler to detect that case and return a suitable 500 reply.

2. Add the ability to set an "order" and "direction" to the users/<username>/messages GET request.

   - This would require changes to the function in message.py that validates the query parameters, and changes to the function in db.py that prepares the query.
   - You should decide what the default direction should be (and perhaps it might depend on the field chosen for order, for instance descending for dates but ascending for subject lines?).
   - You can start by finding the route in messaging.py and the function that implements it, and what other functions *it* calls.
   - You can read a bit for how to implement ordering in sqlAlchemy here[1].
   - Make sure to read in resources.md for the acceptable values for order.
   - You can test your method with something like requests.get('http://127.0.0.1:5000/users/bob/

[1] http://docs.sqlalchemy.org/en/rel_0_9/core/tutorial.html#ordering-grouping-limiting-offset-ing

3. Implement the function that handles the route for deleting a user.

    - It should return an appropriate error code if the username is too long.
    - It should call a db.py function that you will write, which does the actual deletion. It should account for errors on that process (for instance if a connection cannot be established).
    - It should "delete" the user by deleting all messages that have that user in the "from" or "to" fields.

4. Implement the function that handles the route for getting a specific message.

    - It should return a suitable error code if there are database problems.
    - It should return a suitable error code if the message id doesn't exist.
    - It should return the object described in resources.md.

5. (Make sure you have committed your work on 3 before doing this one). Add a suitable link for a "reply" for replying to the message in the reply to a GET requests for a message.

    - This new "reply" field would contain in it two fields: The url to the POST request for creating a new message and a "content" field that contains a sample of what the resulting request should contain.
    - Make sure to automatically include default values for the "from" and "to" fields of the reply message.
    - Make sure to include a setting for the reply_to field.
    - Make sure to automatically "fix" the subject line, by adding a "Re:" in front of the subject line if it doesn't already exist.
    - This all requires work only in messaging.py.
    - This section of the Flask Quickstart[2] may help.

6. Implement the handling of POST requests on a message, which allows the user to set the "read" status for the message.

    - It should expect the request body to be a JSON document with a "read" field with value true or false, and nothing else. You can do this via a new validation function in the message.py file.
    - A suitable error code should be returned if the above validation failed.
    - You will need to implement a suitable db.py function that writes to the database.
    - Make sure to handle with a suitable error the case of database problems.
    - On success you should return a suitable return code for a successful no-content reply.

7. Implement the handling of DELETE requests on a message.

    - You should return a suitable error code if the message doesn't exist.
    - You will need to implement a suitable function in db.py that deletes a message. It must first find all messages that have a reply_to field pointing to this message we are about to delete, and set the reply_to field to null for them. After that the message can be deleted.

[2] http://flask.pocoo.org/docs/0.11/quickstart/#url-building

- On success it should return a suitable code for no-content reply.

8. Implement the handling of GET requests for a message tag.
    - You should return a suitable response code if the message doesn't exist.
    - You should return a suitable response code if the message does not have the specific tag.
    - You should return a suitable response code if the message does have the tag.
9. Implement the handling of DELETE requests for a message tag.
    - You should return a suitable response code if the message doesn't exist.
    - You should return a suitable response code if the message does not have the specific tag.
    - You should return a suitable response code if the message does have the tag, and also delete the tag from the database.
    - You should also make sure to return a suitable response code if there is a database-access error.

When you are ready to submit, make sure you have all your changes saved via commits, then run the following:

```
git format-patch v1..master --stdout > yourName-assignment5.patch
```

Of course use your name in the filename at the en of that line. This will create this new file, which contains all the information about the change you have done. Email this file to me in order to submit.