PROJECT WALKTHROUGH & DETAILS

1) **command_parser.py**
    a. The first step was to start building the command_parser.py to get the valid commands to implement and store them in a queue.
    b. Then we take run the valid commands and run them. For each command we save the output and metadata in the Command object in database.
2) **main.py**
    a. get_command_output: We format the database query results to JSON format.
    b. process_commands: Changing the method of getting the commands file.

```
file = request.files.get('filename')
filename = file.filename
```

    c. Processing the commands through a simple Queue by calling the function directly rather than using multiprocessing. This is because on running the following section, the browser returns an empty output at: http://localhost:8080/commands

```
processes = [Process(target=process_command_output, args=(queue,)) for num in range(2)]
for process in processes:
    process.start()
for process in processes:
    process.join()
```

Therefore, the following code gives me the correct output in the browser

```
queue = Queue()
get_valid_commands(queue, filename)
process_command_output(queue)
```

3) **Dockerfile**: After doing a sanity test of the code by testing if the output is appearing at http://localhost:8080/commands. Next step was to start building a Dockerfile.

IMPLEMENTATION STEPS USING PYTHON VIRTUALENV

Please follow the following steps for implementation of the project:

1) Run python main.py
2) Run the following commands in order to pass commands.txt to filename:
    - curl -X DELETE http://localhost:8080/database
    - curl -X POST http://localhost:8080/database
    - curl -F "filename=@commands.txt" http://localhost:8080/commands
3) Open a browser and go to http://localhost:8080/commands in order to see the metadata stored in database in JSON format.

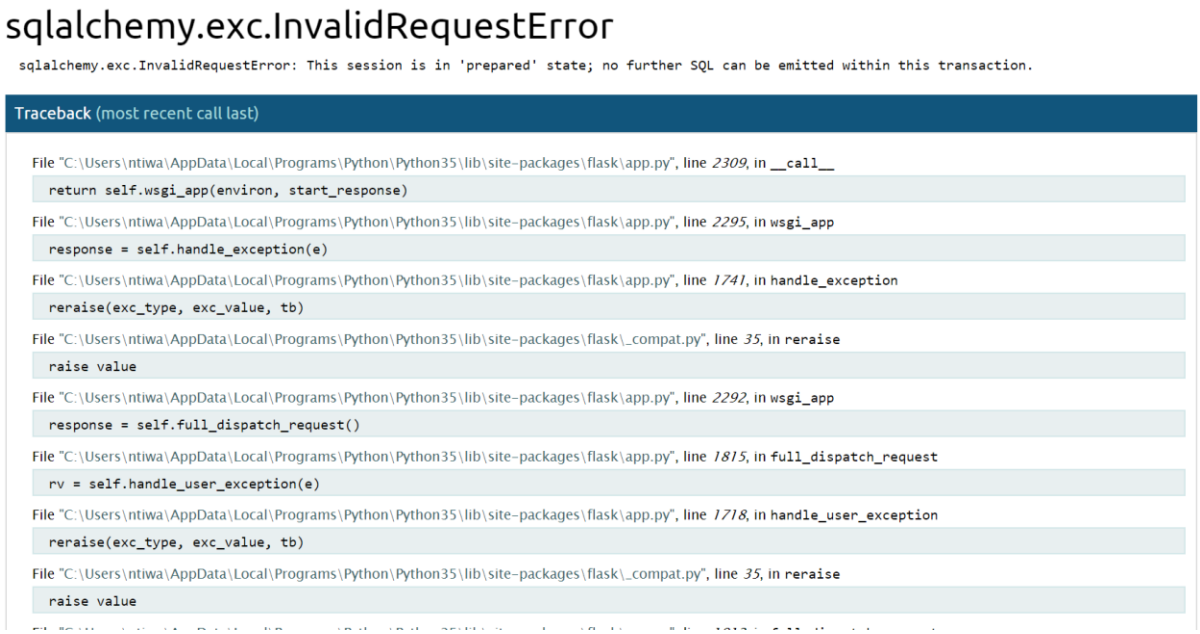
IMPLEMENTATION STEPS USING DOCKER

Please follow the following steps for implementation of the project:

4) Build the Dockerfile to create the image.
5) Run the image to start the local flask server.
6) Run the following commands in order to pass commands.txt to filename:
    - curl -X DELETE http://dockerMachineIP:8080/database
    - curl -X POST http://dockerMachineIP:8080/database
    - curl -F "filename=@commands.txt" http://dockerMachineIP:8080/commands
7) Open a browser and go to http://dockerMachineIP:8080/commands in order to see the metadata stored in database in JSON format.

TESTING:

A command file called command_test.txt contains erroneous commands that do not give any output. On running only invalid commands, the results are as follows.



The code therefore fails on running only invalid commands, there is no population of the database and it gives an Invalid Request error.

ADDITIONAL DETAILS:

- The first Implementation steps running on the virtual environment runs successfully and the results have been recorded in browser_output_8080.PNG and in query_results.json

- On implementation using Docker container by running the Dockerfile created, the image is built successfully. However, on running the image, there is an empty output after the commands are successfully parsed through the command processing service.
- Possible explanation of disparate behaviour:
- Due to the flow of the Dockerfile, it is possible that the database is not getting populated with the values in command_parser.py before getting queried in main.py leading to the empty output.
- However, this is the result and estimate of testing on a Windows 10 environment using a Docker Toolbox.

BONUS TASKS ADDITIONAL DETAILS:

The solutions to certain bonus tasks have been outlined in the document Bonus_tasks_implementation. The implementation code snippets have been provided. However, I did not include the code with the project scripts for the fear of breaking the running application that I had built.