Khanh Nguyen Cong Tran
1002046419
CSE 4382-001
Thomas L. "Trey" Jones

Instructions for Building and Running Software and Unit Tests:

The application can be run within a docker container or run locally on your device.

To run the application on docker:

1. Build the docker image:
   ```
   docker build -t cong-phonebook .
   ```
2. Run the docker image to create the docker container
   ```
   docker run -d -p 8000:8000 cong-phonebook
   ```

To run the application locally:

1. Download the python libraries
   ```
   pip install -r requirements.txt
   ```
2. Run the application
   ```
   uvicorn app:app --host 0.0.0.0 --port 8000
   ```

To check on the audit log:

1. Find the docker container ID:
   a. `docker ps`
2. Enter the docker container
   a. `docker exec -it CONTAINER_ID bash`
3. Inspect the database for the audit logs
   a. `sqlite3 phonebook.db "SELECT * FROM audit_log;"`

Unit testing was done through Postman.

- The collection.json and environment.json is provided in the Postman folder within the project.

1. Open the import Postman Collections and Environment.
2. Navigate to the **Unit Test** folders in the **Phonebook API TEST**
3. Find the **TestData** folder in the **project**
    a. It is separated into different categories such as valid data, invalid data and custom made data.
4. Right-click on a specific **Unit Test folder** and clicke **Run Folder**.
    a. Some folders will have it to where you will need testData as input.
    b. Click on the **Unit Test** and in the description should tell you whether a file is needed and which file in particular to run for this test case.
5. Press **Run PhoneBook API Tests**

Description of My Code

I will go through `app.py`, `Dockerfile`, and `requirement.txt` files. I will go through the significant parts of each file in sequence, top to bottom. If a part was not mentioned here but was in the cdoe, it was likely already present within the codebase (from the zip file download) and not worth mentioning again.

`app.py`

1. REGEX
    a. I developed both regular expressions for names and phone numbers. I broke up each component of each REGEX for the specific case they are handling. I understand they are both extremely long but it works so.
2. Audit Log
    a. I added another table specifically for audit logs. Every API request will log the information and can be easily be obtained within the `phonebook.db`
3. API_KEYS
    a. I added the proper API authentication keys for read/write roles (“admin-key”) and just read roles (“read-key”).
    b. If no authentication key is provided or a key that was provided that isn't one of the two above, the user is not able to make any request.

4. `get_api_key`
    a. This is an async function taht is called everytime a API request is called. This ensures that for every API request is called, the proper authentication key is provided to perform the action. Otherwise, an error is thrown and the request is not fulfilled.
5. Added validation to Person Model
    a. Using the `@validator` call, I can perform validation checks on the passed in name and phone number. If one of the two attributes do not pass their respected REGEX match, it will throw an error adn the person model class will not be created.
6. Home Page
    a. A basic home page that you have to add the `/doc` to acces the application
7. Middleware
    a. This auto logs timestamps of when requests are called and important information necessary for audit logging.
8. /PhoneBook/list
    a. Same as the original code but using parameterized queries.
9. /PhoneBook/add
    a. I added normalization to phone numbers because there was an issue with deleting certain phone numbers with “+”, “(“, “)” and other non-digit but valid characters. This also helps with keep phone numbers unique because normalizations get rid of all the possible variables of given the same phone numbers.
    b. I parameterized the SQL query in this one as well.
10. /PhoneBook/deleteByName
    a. Same as before but with parameterized queries
11. PhoneBook/deleteByNumber
    a. Same as before but with parameterized queries and normalizations of phone numbers and that normalized phone number is what we use to filter out the database.

Dockerfile

1. Added more environment variables
    a. Have the `APP_HOME` to my `/app` location
    b. Have the PORT=8000
2. Set the work directory
    a. Set it to `APP_HOME`

3. Install system dependencies:
    a. I did this to do debugging and check the audit logs. In a real application, this would pose security risk so this should be avoided probably.
4. Expose the PORT

`requirements.txt`

1. No changes were made.

Assumptions I Have Made (10 points)

1. The API keys aren't hardcored and are securely managed and not exposed to unauthorized parties who have access to the source code
2. Phone numbers should be normalized. So ++1(800)-123-4568 is the same as 1(800)-123-4578 because they are both the same phone number, purely based on the ordering, regardless of the extra symbols.
3. User Input Validation: Assumes that user input (names and phone numbers) is validated correctly using regex patterns.
4. Docker is already installed and properly configured on the host machine to hand create a docker image and docker container.
5. Port 800 is available and not being used.
6. The application expects the database to be created in /app and that the container has write permissions there.
7. The application assumes all requests are made with proper API keys and valid JSON payloads.

Pros/Cons of My Approach (5 points)

Pros:

- Normalization of the phone number:
    - I am storaging the normalizd version of a phone number in the database
    - This allows for only unique phone numbers to be present and prevent same phone numbers with extra characters from being recognized as the same phone number
- Using Postman to Test:
    - It is quick and easy to do.
    - It is also testing from a separate application so this prevents any type of bias whereas I was testing within my application.

    - Since Postman is sending requests externally, I can test the application from its rawest form and from the perspective of an actual user.
- Audit Logs cannot be accessed from the user
    - Audit logs are stored within the database but you need access to the database in order to send a SQL query to output the audit log.
- The Docker container does not have root access
    - I specifically did not give the Docker container root access because I did not want any attackers to be able to enter the docker container and already have a root request to run malicious commands.
    - I only gave the docker container the basic libraries to be functional and to output the audit logs and that is it.

Cons:

- The API keys, API key header, and database models are exposed in the codebase.
    - It is better practice to keep the API keys and API key header in some type of secrets manager
    - The database model should be within its folder / file instead of being within the app.py file.
- This entire thing is ran in Python
    - No further explanation.
- SQLite is only good for small applications. If we scale this to the millions and billions, another database should be much better.