

Input Validation Report

Code Description:

- This project is a FastAPI-based RESTful API used for securely maintaining the records of a phonebook. It uses the Python FastAPI made by Siddhrajsinh Pradumansinh Solanki as base for the starter API. The API has the following features:
 - **GET, POST, PUT Operations:** Authenticated users can retrieve, create, and delete phonebook entries.
 - **Input Validation:** All phone and name inputs are validated using regex patterns to ensure valid data is supplied.
 - **Role-based Authentication:**
 - **“read”:** Users with this role can only view records, not update them.
 - **“readwrite”:** Users with this role can both view records and update them.
 - **Audit Logging:** Every major API call is logged in a log file with the designated time stamp, user action, and their utilized token for accountability. Each app run will have its own audit log. These logs are saved to disk via mounting in Docker.
 - **Database Persistence:** Changes made to a database will persist to disk, allowing other containers to access and retain previous records, so long as the database is not deleted. This is accomplished via mounting in Docker.

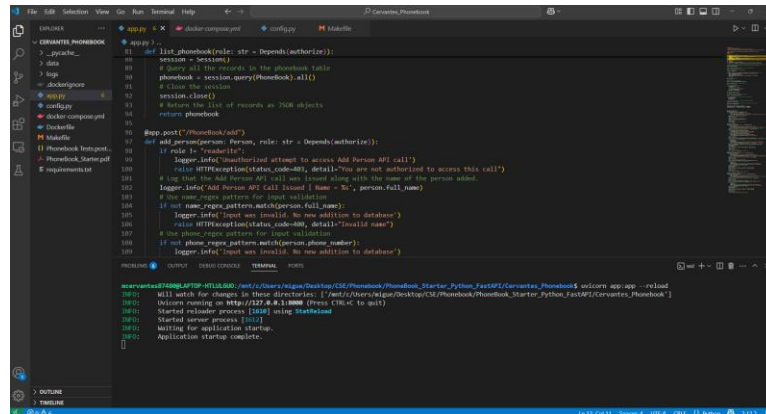
Tech Stack Used:

- **Backend:**
 - Python, FastAPI, Uvicorn
- **Database:**
 - SQLite, SQLAlchemy
- **Testing Tools:**
 - Postman
- **Containerization & Building:**
 - Docker, Docker Compose, Makefile

Instructions for Building and Running Software and Unit Tests:

- **Running the app:**
 - Open Visual Studio.
 - In the top left corner, click on “File” and then select “Open Folder”.
 - Navigate to the directory where the code (app.py) is located and select it.
 - The app.py code should now be displayed, as should the directory it is located in.
 - Open a new terminal by click the “Terminal” tab at the top and clicking “New Terminal”.
 - Ensure the terminal type is Ubuntu (WSL)
 - Install the required libraries by running this command in the terminal: **pip install -r requirements.txt**
 - After installing the required libraries run the app by typing the command in the terminal: **uvicorn app:app --reload**
 - Upon running the app, the /data and /logs directories will be made in the current directory in order to store the database and logs files that are generated. The output to be expected can be seen below.

- **WARNING:** Ensure that these directories are not deleted manually (File Explorer), but rather by running: **make clean**
- Failing to do so will result in the app being unable to recreate those folders again automatically. This can be fixed by recreating those directories manually.

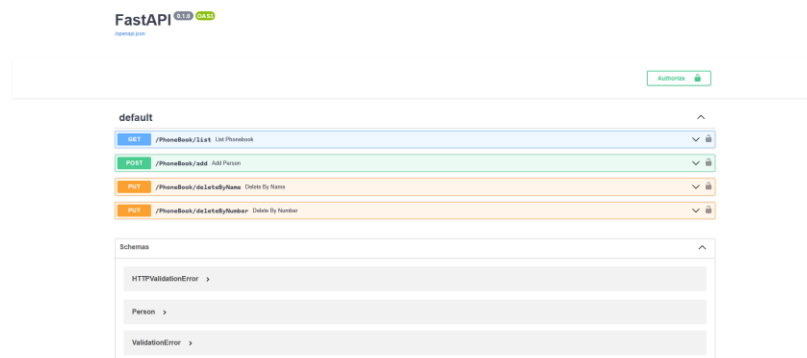


```
def list_phonenumber(rule: str = Depends(authorize)):
    session = Session()
    # Query all the records in the phonebook table
    phonebook = session.query(Phonebook).all()
    # Close the session
    session.close()
    # Return the list of records as JSON objects
    return phonebook

@app.post("/phonebook/add/")
def add_person(person: Person, rule: str = Depends(authorize)):
    if rule is "Unauthorized":
        logger.info("Unauthorized attempt to access Add Person API call")
        raise HTTPException(status_code=401, detail="You are not authorized to access this call")
    # Log that the Add Person API call was issued along with the name of the person added
    logger.info(f"Add Person API call issued [ Name = {person.full_name}, person.full_name]
    # Use name_regex pattern for input validation
    if not name_regex.pattern.match(person.full_name):
        logger.info("Input was invalid. No new addition to database")
        raise HTTPException(status_code=400, detail="Invalid name")
    # Use phone_regex pattern for input validation
    if not phone_regex.pattern.match(person.phone_number):
        logger.info("Input was invalid. No new addition to database")
        raise HTTPException(status_code=400, detail="Invalid phone number")
    # Add the person to the database
    session.add(person)
    session.commit()
    return person
```

- Creating the Docker Image and Running via Docker

- A Dockerfile is utilized for image creation and a docker-compose file is utilized for ease of access in running Docker containers. This is used in conjunction with a Makefile for running the container with simplified commands.
- Firstly, ensure Docker desktop is running.
- Next, in a terminal run the command: **docker pull python:3.10-slim**
 - This will ensure that the library is up-to-date and is able to create the image successfully.
- Afterwards, in the terminal run the command: **make setup**
 - This creates the directories /data and /log which will be used to persist the database and log files to disk.
- To build and run the image, run the command: **make up**
 - This command creates and runs the Docker image as well as creates the necessary directories for mounting if they have not been created yet
- Once the image is running, open a browser window and navigate to: <http://127.0.0.1:8000/docs>
- What the website looks like can be seen below:



- To run any API call, you must authenticate with the appropriate token for either the “read” or “readwrite” role. The tokens for each role can be found in **config.py** listed as **API_KEYS**
- Additionally, provided is a Postman collection of numerous tests which are utilized to test each API call.
 - Ensure the docker container is up and running and that the database is empty before running the test collection via Postman.
 - The tests provide valid input for name and phone numbers as well as invalid input for name and phone numbers. Additionally, tests are provided to test each token authentication of each API call.
- To stop a container, either do so via Docker desktop or in terminal using Ctrl + C.
- In order to delete the container, run the command: **make down**
- In order to delete the /data and /logs directories safely run the command: **make clean**
 - **WARNING:** Doing so will delete the directories, thus deleting the current database and any existing log files.
- In order to perform a reset of the image, run: **make rebuild**
- If any new changes are made to the code or corresponding elements, the image must be rebuilt to take effect.

How it Works:

- The application is built using FastAPI, a modern, high-performance web framework for building RESTful APIs with Python. It runs inside a Docker container using Uvicorn as the ASGI server.
- Several configuration settings are set in config.py such as the API security keys, log file format, and regex patterns.
- SQLite is used for persistent storage of phonebook entries, with SQLAlchemy managing the database models and sessions.
- The API supports basic phonebook operations: adding a person, listing all entries, and deleting them by either name or phone number
- Token-based authentication is implemented using FastAPI's HTTPBearer scheme. Users must provide a valid token to interact with the API. Roles (read, readwrite) are assigned based on the token and restrict access to certain endpoints accordingly.
- Audit logging is enabled to log all access attempts, including both successful actions and failed attempts (e.g., unauthorized access, invalid input).
- Input validation is enforced via regular expressions defined in config.py
- **Regex Patterns:**
 - **NAME_PATTERN:** Validates input in the name field to prevent malformed or malicious input
 - Restricts names to a maximum of 100 characters and prevents repeated apostrophes or hyphens.
 - Disallows digits, SQL keywords (select, drop, etc.), HTML tags, etc.
 - Allows formats like First Last, (Last, First MI), or First Middle Last.
 - **PHONE_PATTERN:** Accepts a wide variety of international and North American formats
 - Ex: 12345, +1 (703) 123-1234, 011 1 703 111 1234, 123-1234, and even 12345.12345
 - Rejects invalid phone numbers and any attempts to inject scripts or SQL

Assumptions:

- **User Input is Untrusted:**
 - When designing this, I naturally assumed from a security perspective that we can not trust user input. In other words, all user input is treated as potentially malicious, and data is thoroughly

combed through to ensure that the data being entered is valid and safe. This is done through the use of regex patterns.

- **Static API Keys for Testing:**
 - API keys are hardcoded in config.py to streamline local development and testing. The assumption is that these tokens are only shared with trusted users. In a production environment, this would be replaced by dynamically generated tokens, secure key management, or OAuth.
- **Regex Limitations:**
 - Regex patterns are utilized to perform data validation on user input. These patterns, however, may not allow all valid input through and thus there is expected to be some possible loss in favor of input validation.
- **Utilizes Localhost/Docker:**
 - This implementation assumes that the application is run locally and not deployed to a public-facing server. Therefore, extra layers of security are not implemented here.

Pros/Cons:

- **Pros:**
 - **Strong Input Validation:**
 - Matching via regex patterns prevents malformed or potentially malicious data from entering the system, lowering the risk of injection attacks or database inconsistencies.
 - **Audit Logging:**
 - All important API interactions are logged and saved to disk allowing for previous logs to be viewed. These logs monitor access and report any API calls made along with the token utilized.
 - **Role-based Access Control:**
 - The use of tokens with distinct roles ensures that only users with proper permissions can perform sensitive actions
 - **Easy, Testable Design:**
 - By keeping the application simple and self-contained with FastAPI, SQLite, and Docker, it makes it easy to test and run without relying on external services.
 - **Clear and Maintainable Codebase:**
 - The code structure follows clean conventions with separation of concerns such as configurations in config.py, models via SQLAlchemy, security and authentication via tokens.
- **Cons:**
 - **Hardcoded Security Keys:**
 - API security keys are hardcoded in config.py making them insecure in production. These static keys can be leaked or reused, making them vulnerable. A more robust authentication mechanism would be preferable.
 - **Rigid Regex Rules:**
 - The regex patterns prioritize security over user flexibility. As a result, certain technically valid but uncommon entries may be rejected to reduce the risk of malformed or malicious inputs. This is the result of a tradeoff between security and user experience.
 - **Lack of Persistent User Management:**
 - Authentication is in the form of static security keys. There is no user account system or key expiration logic, which would be needed for scalability or real-world use.
 - **No Encryption/Security Layers:**

- Token-based access is basic and lacks encryption or verification steps. HTTPS, secure storage of tokens, or JSON Web Tokens would be better for a public-facing app. This also means that if an attacker gains access to a token, they can send requests via a URL that could alter the database.
- **Only Viable Locally:**
 - The app is intended to run locally via Docker and has not been configured for deployment to a public cloud or production-grade server
- **Limited Scalability:**
 - This app uses an in-memory implementation of SQLite which is ideal for local testing but is not suitable for access in a large-scale environment.