Secure Phone Book REST API Report

This report describes the implementation of a secure Phone Book REST API as per the requirements. It covers the functionality of the code, the design of regular expressions for input validation, assumptions made during development, and the pros and cons of the chosen approach.

How the Code Works

The Phone Book REST API is built using **FastAPI**, a modern Python web framework, and uses **SQLAlchemy** for database interactions. The API uses dependency injection to interface with important functions such as Logger, Auth, Database, and Models. This interface allows for each endpoint to use these functions as "per-request plugins." Dependency injection is supported by FastAPI and allows modularity for the API and future expansion. The API provides the following endpoints:

- **GET /PhoneBook/list**: Lists all entries in the phone book.
- **POST /PhoneBook/add**: Adds a new person to the phone book.
- **PUT /PhoneBook/deleteByName**: Deletes a person by their full name.
- **PUT /PhoneBook/deleteByNumber**: Deletes a person by their phone number.

Note: The POST /token endpoint is not listed since its middleware for OAuth.

Key Components Summary - see design.md for more details

1. **Database**:
    - **SQLAlchemy ORM** interacts with an SQLite database (`phonebook.db`) to store phone book entries.
    - The schema is defined in `models.py` , with the `PhoneBook` model representing the table structure.
2. **Authentication and Authorization**:
    - **JWT (JSON Web Tokens)** are used for authentication via the `/token` endpoint, issuing tokens upon valid credentials.
    - Two roles exist: **Read** (list entries) and **ReadWrite** (list, add, delete entries).
    - Hardcoded users are implemented with hashed passwords using the `passlib` library.
3. **Input Validation**:
    - **Pydantic models** with custom validators enforce input validation using regular expressions.
    - Names and phone numbers are validated before processing or storage.
4. **Audit Logging**:
    - Actions (token, add, delete, list) are logged in `audit.log` with timestamps and operation details.
5. **Error Handling**:
    - Custom handlers manage validation errors, returning HTTP status codes like 400 (bad request) or 404 (not found).

API Flow Summary - see design.md for more details

1. **Authentication**:
    - Users authenticate via `/token` with a username and password, receiving a JWT token.
    - The token is included in the `Authorization` header for protected endpoints.
2. **Endpoint Access**:
    - JWT tokens are validated per request, with role-based access control enforcing permissions.
3. **Database Operations**:
    - Parameterized queries via SQLAlchemy prevent SQL injection.
    - Inputs are validated before database interactions.
4. **Logging**:
    - Significant actions are recorded in `audit.log` for auditing.

Design of Regular Expressions

Regular expressions validate the `full_name` and `phone_number` fields in the `Person` model, balancing flexibility and integrity.

Name Validation

- **Pattern**: `^[a-zA-Z.,'\u2019 -]+$`
- **Explanation**:
    - Permits letters (upper/lowercase), periods, commas, apostrophes (including curly as shown in examples), spaces, and hyphens.
    - Prevents two consecutive apostrophes or curly.
    - Limits name parts to three (e.g., first, middle, last).
    - Limits each name part to one hyphen.
- **Flexibility**:

- Supports names like "O'Malley," "John-Smith," or "Bruce Wayne Schneier."
- All provided examples are successfully sorted as valid or invalid.

Phone Number Validation

- **Patterns**:
  - **Basic**: `^[+\d()-. ]+$`
  - **Extension**: Used for formats like `12345`
  - **North American**: Formats like `(703) 111-2121`, `123-1234`, `+1(703) 111-2121`.
  - **International**: Starts with `+` followed by country code and number.
  - **IDD**: International direct starts with `011`
  - **Danish**: Formats like `12 34 56 78` or `1234 5678`.
  - **General**: Allows spaces, dots, or hyphens.
- **Explanation**:
  - Basic permits only allowed characters (`+`, digits, parentheses, hyphens, periods, spaces).
  - Enforces digit length between 5 and 15.
  - Matches must be one of the regional and format-specific patterns.
- **Flexibility**:
  - Accommodates diverse international and local phone number formats.
  - All provided examples are successfully sorted as valid or invalid.

Assumptions Made

1. **User Management**:
   - Hardcoded users suffice for a test environment that balances the scope of requirements versus a full system. However, the project was designed to try to allow for the expansion of requirements for a full, dynamic user management system.
2. **Database**:
   - SQLite is adequate, assuming low concurrency and small data scale. If used in production as a Docker container, the dynamically created db would need to have its permissions restricted to the 'appuser' from the Dockerfile.
3. **Security**:
   - The JWT secret key is a current best practice and HTTPS is assumed (though not enforced).
4. **Input Data**:
   - Names and phone numbers fit the defined regular expressions; extreme cases are unhandled.
5. **Logging**:
   - File-based logging (`audit.log`) is sufficient. No log rotation was included.

Pros and Cons of the Approach

Pros

1. **Security**:
   - Regular expressions and Pydantic prevent injection attacks.
   - JWT and role-based access control secure the API.
   - Parameterized queries eliminate SQL injection risks.
2. **Modularity**:
   - FastAPI and SQLAlchemy enable easy feature extensions.
3. **Testing**:
   - Pytest unit tests validate functionality; Docker simplifies setup.
4. **Audit Logging**:
   - Tracks user actions for security auditing.

Cons

1. **Hardcoded Users**:
   - A mock development object; a user management system is needed.
2. **Database**:
   - SQLite limits scalability and concurrency.
3. **Security Features**:
   - Lacks rate limiting, IP whitelisting, or HTTPS enforcement.
4. **Error Handling**:
   - Basic; lacks detailed messages and logging.
5. **Performance**:
   - Unoptimized for large datasets.

Conclusion

The secure Phone Book REST API meets the assignment requirements, showcasing input validation, authentication, authorization, and logging. Built with FastAPI and SQLAlchemy, it is functional and extensible, though production use would require enhancements like a robust database and user management system.

References

1. https://fastapi.tiangolo.com/
2. https://github.com/sumanentc/python-sample-FastAPI-application
3. https://dassum.medium.com/building-rest-apis-using-fastapi-sqlalchemy-uvicorn-8a163ccf3aa1
4. https://regex101.com/
5. https://medium.com/@ramanbazhanau/mastering-sqlalchemy-a-comprehensive-guide-for-python-developers-ddb3d9f2e829
6. https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/
7. https://www.propelauth.com/post/a-practical-guide-to-dependency-injection-with-fastapis-depends