Project: Secure Phone Book REST API

Student: Vineet Kurapati (ID: 1002154505)
Instructor: Thomas L. "Trey" Jones

Introduction

This report documents the development and features of the Secure Phone Book REST API, created as part of the [Course Name/Number Here] assignment under the guidance of Thomas L. "Trey" Jones at the University of Texas at Arlington. The primary goal of this project was to enhance a basic phone book API by implementing robust input validation using regular expressions, incorporating essential security features (authentication, authorization, audit logging), ensuring data persistence, and packaging the solution using Docker. This document details the implementation approach, provides instructions for building and running the application and its tests, discusses assumptions made during development, and evaluates the pros and cons of the chosen solution, fulfilling the reporting requirements outlined in the assignment specification.

1. Instructions for Building and Running Software and Unit Tests

This section provides detailed instructions for installing dependencies, building the application, running the API server, and executing the automated unit tests using both Docker (recommended) and local setup methods. It also includes instructions for manual testing using curl.

Prerequisites:

- **For Docker:** Docker Engine and Docker Compose installed and running.
- **For Local:** Python 3.9+, pip package manager.
- Git (for cloning the repository).
- curl command-line tool (for manual API testing).

Option 1: Docker Installation & Running (Recommended & Required Method)

This method uses Docker and Docker Compose to build and run the application in a containerized environment, ensuring consistency and managing dependencies.

1. **Download the Source Code**

2. **Verify Dockerfile:** Ensure the Dockerfile is present in the project's root directory alongside docker-compose.yml. This file defines the container image build process.
3. **Build and Run Containers:** From the project's root directory, run:

docker-compose up --build

- --build: Forces Docker Compose to build the image using the Dockerfile.
- This command builds the image, creates necessary volumes for data persistence (./data mapped to /app/data, ./logs mapped to /app/logs), and starts the API server on port 8000.

4. **Run in Detached Mode (Optional):** To run the containers in the background:
   docker-compose up --build -d

5. **Accessing the API:**
   - API Endpoint: http://localhost:8000
   - Interactive API documentation (Swagger UI): http://localhost:8000/docs
   - Alternative API documentation (ReDoc): http://localhost:8000/redoc
6. **Viewing Logs:**
   - View all logs: docker-compose logs
   - Follow logs in real-time: docker-compose logs -f
7. **Stopping the Application:**
   docker-compose down

8. **Checking Container Status:**
   docker-compose ps

Option 2: Local Installation & Running

This method involves setting up a local Python environment.

1. **Download the Source Code**

2. **Create and Activate Virtual Environment:**
   python -m venv venv
   source venv/bin/activate # Linux/macOS
   # or venv\Scripts\activate # Windows

3. **Install Dependencies:**
   pip install -r requirements.txt

4. **Initialize the Database:** This creates the phonebook.db file (or the file specified by DATABASE_FILE env var) and sets up the necessary tables and default users.
   python init_db.py

5. **Run the Application Server:**
   uvicorn main:app --reload

   - --reload: Enables auto-reloading when code changes are detected (useful for development).
   - The API will be accessible at http://localhost:8000.
6. **Deactivate Virtual Environment (When finished):**
   deactivate

Running Unit Tests

Tests verify API functionality, validation, security, and requirements adherence.

- **Using Docker:**

  1. If using the main docker-compose.yml, ensure containers are running (docker-compose up -d).
  2. Execute tests inside the api container:
     docker exec <container_name_or_id> python -m pytest test_phonebook.py -v

- **Running Locally:**
  1. Ensure you are in the activated virtual environment with dependencies installed.
  2. **Set up Test Environment:** Creates test-specific database and directories.
     python test_setup.py

  3. **Run Tests:**
     python -m pytest test_phonebook.py -v

API Testing with Curl (Manual)

You can manually test the API endpoints using curl once the application is running (either locally or via Docker).

- **Base URL:** http://localhost:8000
- **1. Get Authentication Tokens:**
  - Get writer token:
    curl -X POST "http://localhost:8000/token" \
      -H "Content-Type: application/x-www-form-urlencoded" \
      -d "username=writer&password=writerpass"

- Get reader token:
  curl -X POST "http://localhost:8000/token" \
    -H "Content-Type: application/x-www-form-urlencoded" \
    -d "username=reader&password=readerpass"

- *Note:* Copy the access_token value from the JSON response (e.g., {"access_token": "your-token-here", "token_type": "bearer"}) and use it in subsequent requests where YOUR_TOKEN_HERE, YOUR_WRITER_TOKEN, or YOUR_READER_TOKEN is indicated.

- **2. List Entries (Requires any token):**
  curl -X GET "http://localhost:8000/PhoneBook/list" \
    -H "Authorization: Bearer YOUR_TOKEN_HERE"

- **3. Add Entry (Requires writer token):**
  - Add valid entry:
    curl -X POST "http://localhost:8000/PhoneBook/add" \
      -H "Authorization: Bearer YOUR_WRITER_TOKEN" \
      -H "Content-Type: application/json" \
      -d '{"name": "Bruce Schneier", "phoneNumber": "(703)111-2121"}'

  - Test invalid name (should return 400):
    curl -X POST "http://localhost:8000/PhoneBook/add" \
      -H "Authorization: Bearer YOUR_WRITER_TOKEN" \
      -H "Content-Type: application/json" \
      -d '{"name": "L33t Hacker", "phoneNumber": "(703)111-2121"}'

  - Test invalid phone (should return 400):
    curl -X POST "http://localhost:8000/PhoneBook/add" \
      -H "Authorization: Bearer YOUR_WRITER_TOKEN" \
      -H "Content-Type: application/json" \
      -d '{"name": "Valid Name", "phoneNumber": "invalid-phone"}'

- **4. Delete Entry by Name (Requires writer token):**
  # Note: URL encode the name if it contains spaces or special characters
  curl -X PUT "http://localhost:8000/PhoneBook/deleteByName?name=Bruce%20Schneier" \
    -H "Authorization: Bearer YOUR_WRITER_TOKEN"

- **5. Delete Entry by Number (Requires writer token):**
  # Note: URL encode the number if it contains special characters like parentheses
  curl -X PUT "http://localhost:8000/PhoneBook/deleteByNumber?number=%28703%29111-2121" \
  -H "Authorization: Bearer YOUR_WRITER_TOKEN"

- **Example Test Scenarios:**
  - **Test Authentication:**
    - Valid: curl -X POST "http://localhost:8000/token" -H "Content-Type: application/x-www-form-urlencoded" -d "username=writer&password=writerpass"
    - Invalid: curl -X POST "http://localhost:8000/token" -H "Content-Type: application/x-www-form-urlencoded" -d "username=writer&password=wrongpass" (Should fail)
  - **Test Authorization:**
    - Add with reader token: curl -X POST "http://localhost:8000/PhoneBook/add" -H "Authorization: Bearer YOUR_READER_TOKEN" -H "Content-Type: application/json" -d '{"name": "Should Fail", "phoneNumber": "123-4567"}' (Should fail with 403)
  - **Test Input Validation (Add Examples):**
    - curl -X POST "http://localhost:8000/PhoneBook/add" -H "Authorization: Bearer YOUR_WRITER_TOKEN" -H "Content-Type: application/json" -d '{"name": "Schneier, Bruce", "phoneNumber": "(703)111-2121"}'
    - curl -X POST "http://localhost:8000/PhoneBook/add" -H "Authorization: Bearer YOUR_WRITER_TOKEN" -H "Content-Type: application/json" -d '{"name": "O'\"Malley, John F.", "phoneNumber": "(703)111-2121"}' (Note: single quote escaped for shell)
    - curl -X POST "http://localhost:8000/PhoneBook/add" -H "Authorization: Bearer YOUR_WRITER_TOKEN" -H "Content-Type: application/json" -d '{"name": "Test User 1", "phoneNumber": "12345"}'
    - curl -X POST "http://localhost:8000/PhoneBook/add" -H "Authorization: Bearer YOUR_WRITER_TOKEN" -H "Content-Type: application/json" -d '{"name": "Test User 2", "phoneNumber": "+1(703)111-2121"}'
  - **Test Error Cases:**
    - Duplicate Add (409): Add an entry, then try adding the exact same entry again.
    - Delete Non-Existent (404): curl -X PUT "http://localhost:8000/PhoneBook/deleteByName?name=NonExistentUser" -H "Authorization: Bearer YOUR_WRITER_TOKEN"

2. Description of How The Code Works

This project implements a secure RESTful API for managing a phone book, built using Python and the FastAPI web framework. It adheres to the OpenAPI 3.0 specification provided (PhoneBook.json).

- **Core Framework (main.py):** FastAPI provides the foundation, handling HTTP requests, routing, data serialization/deserialization, dependency injection, and automatic generation of interactive API documentation (Swagger UI at /docs, ReDoc at /redoc). Asynchronous capabilities (async def) are utilized for potentially I/O-bound operations.
- **API Endpoints (main.py):**
  - POST /token: Handles user authentication. Takes username and password (OAuth2PasswordRequestForm), verifies credentials against the database (authenticate_user), and returns a JWT Bearer token (create_access_token) upon success.
  - GET /PhoneBook/list: Retrieves all phone book entries, ordered by name. Requires a valid JWT token (any role).
  - POST /PhoneBook/add: Adds a new entry. Requires a valid JWT token with 'readwrite' role (require_write_permission dependency). Accepts a JSON body conforming to the PhoneBookEntry model. Performs input validation. Returns HTTP 200 on success, 400 on validation error, 409 if the name already exists.
  - PUT /PhoneBook/deleteByName: Deletes an entry based on the name query parameter. Requires 'readwrite' role. Returns 200 on success, 404 if the name is not found.
  - PUT /PhoneBook/deleteByNumber: Deletes an entry based on the number query parameter. Requires 'readwrite' role. Performs validation on the number format. Returns 200 on success, 404 if the number is not found.
- **Data Modeling and Validation (main.py - PhoneBookEntry class):**
  - Pydantic's BaseModel is used to define the structure of the expected request body for adding entries (name, phoneNumber).
  - @field_validator decorators implement custom validation logic for name and phoneNumber fields, enforcing the rules specified in the assignment using regular expressions and conditional checks.
  - **Name Validation:** Allows letters, spaces, single apostrophes, single hyphens, periods, and a comma (for "Last, First"). Disallows digits, multiple consecutive separators (space, ', -), potentially harmful characters (<>;()[]{}|\\"$), and names with more than 3 logical parts. It specifically checks against unacceptable examples like "Ron O"Henry", "L33t Hacker", "...", "select * ...", etc.
  - **Phone Number Validation:** Uses a comprehensive regex (pattern) to allow various formats: 5-digit extensions, North American formats (with/without country code, area code, parentheses, hyphens, spaces, dots), international formats (starting with +,

optional country code, area code in parentheses), and specific examples like "011..." or "12345.12345". It explicitly rejects numbers containing letters, those that are too short/long after cleaning, specific invalid country/area codes ("+01", "(001)"), and potentially harmful characters.
    - Validation failures raise ValueError, which is caught by a custom exception handler to return an HTTP 400 response.
- **Persistence (main.py, init_db.py):**
    - An SQLite database is used for storing user credentials (users table) and phone book entries (phonebook table). The database file path is configurable via the DATABASE_FILE environment variable (config.py).
    - init_db.py script handles the creation of tables and seeding initial users ('reader', 'writer') with hashed passwords. It drops existing tables first to ensure a clean state upon initialization.
    - Database interactions in main.py use the standard sqlite3 library. Connections are obtained via the get_db function.
    - **Parameterized Queries:** All SQL queries involving user-provided data (INSERT, SELECT WHERE, DELETE WHERE) use ? placeholders (e.g., c.execute("DELETE FROM phonebook WHERE name = ?", (name,))). This is a crucial security measure to prevent SQL injection vulnerabilities, as the database driver handles proper escaping of input values.
- **Authentication and Authorization (main.py, config.py):**
    - **Authentication:** Uses the OAuth2 Password Bearer flow. Clients post username/password to /token to receive a JWT. The python-jose library handles JWT creation (create_access_token) and decoding/validation (get_current_user). Passwords are securely hashed and verified using passlib with the bcrypt algorithm (pwd_context).
    - **Authorization:** Role-based access control (RBAC) is implemented. User roles ('read', 'readwrite') are stored in the users table and potentially included in the JWT payload (though the current implementation re-fetches the user from DB in get_current_user). FastAPI's dependency injection system is used:
        - Depends(get_current_active_user): Ensures a valid token is present and corresponds to an existing user for accessing any protected endpoint.
        - Depends(require_write_permission): An additional dependency applied to write-operations (add, deleteByName, deleteByNumber). It checks if the authenticated user has the 'readwrite' role; otherwise, it raises an HTTP 403 Forbidden error.
- **Configuration (config.py):** Centralizes application settings like database file path, JWT secret key, algorithm, token expiry time, user role names, and logging configurations. It reads values from environment variables with sensible defaults.
- **Audit Logging (main.py - setup_logging, logger):**

    - Python's standard logging module is configured to write timestamped logs to a file (logs/audit.log by default, path configurable).
    - A RotatingFileHandler is used to limit log file size (10MB) and keep backups (5 files), preventing excessive disk usage.
    - Log entries are generated for significant events: successful logins, failed logins, database initialization, API requests (via middleware or explicit logging), successful data additions/deletions (including the specific data), validation errors, HTTP exceptions, and unhandled errors.
- **Database Backup (main.py - backup_database):**
    - A function backup_database is called after successful add or delete operations.
    - It copies the current database file (DATABASE_FILE) to the data/backups directory with a timestamped filename (e.g., phonebook_YYYYMMDD_HHMMSS.db).
    - It implements a simple retention policy, keeping only the 5 most recent backup files based on modification time and deleting older ones.
- **Rate Limiting (main.py - simple_rate_limiter):**
    - A simple in-memory rate limiter is implemented as a FastAPI dependency (Depends(simple_rate_limiter)).
    - It tracks request timestamps per client IP address (or a fixed identifier for the test client) within a sliding window (RATE_LIMIT_WINDOW seconds).
    - If the number of requests exceeds RATE_LIMIT_COUNT within the window, it raises an HTTP 429 Too Many Requests error. *Note: This is not suitable for production environments with multiple processes.*
- **Error Handling (main.py - Exception Handlers):**
    - Custom exception handlers are defined using @app.exception_handler for RequestValidationError, ValueError, HTTPException, and generic Exception.
    - These handlers log the error details and return structured JSON responses with appropriate HTTP status codes (400, 404, 403, 409, 429, 500), preventing stack traces from leaking to the client.
- **Testing (test_phonebook.py, test_setup.py):**
    - pytest is used for automated testing. test_setup.py prepares the environment for local testing.
    - test_phonebook.py contains a comprehensive suite of tests covering:
        - Authentication (valid/invalid login).
        - Authorization (read vs. readwrite roles).
        - Endpoint functionality (list, add, delete).
        - Input validation (using assignment's acceptable/unacceptable examples for names and phones).
        - Edge cases (deleting non-existent entries, adding duplicates).
        - Security vulnerabilities (SQL injection attempts, XSS attempts in input fields).

        - Audit logging checks (verifying log file existence and content).
        - Token handling (simulated expiration/invalidity).
    - A Postman collection (phonebook_api.postman_collection.json) is also provided for manual API interaction and testing.
- **Containerization (Dockerfile, docker-compose.yml):**
    - The Dockerfile defines the steps to build a container image containing the application and its dependencies.
    - docker-compose.yml orchestrates the running of the application container, managing port mappings, volume mounts for persistent data/logs, and environment variables.

3. Assumptions Made

- **Development Environment:** The primary development and testing environment is assumed to be Linux-based or compatible with Docker Desktop on macOS/Windows. File paths and commands might need minor adjustments for native Windows execution outside Docker.
- **Input Validation Scope:** The validation rules for names and phone numbers are based on the specific examples and descriptions in the assignment PDF. While aiming for flexibility (e.g., various international/US phone formats), they might not cover every conceivable valid format globally or handle highly unusual edge cases. The focus is on meeting the assignment's criteria and preventing common invalid/malicious inputs.
- **Name Structure:** Name validation assumes standard Western naming conventions (First, Middle, Last) or "Last, First" formats, allowing for common variations (initials, hyphens, apostrophes). It restricts complexity (max 3 parts) based on assignment examples, which might exclude some valid but complex names.
- **Security Context:** Security measures (input validation, parameterized queries, AuthN/AuthZ) are implemented based on common web security principles and the assignment's focus areas. It's assumed this level of security is sufficient for the assignment's scope, not necessarily for a high-risk production environment without further hardening.
- **Rate Limiting Implementation:** The choice of a simple, in-memory rate limiter assumes a single-instance deployment scenario typical for assignments. It's understood this approach is not scalable for production.
- **Default Configuration:** The default values in config.py (especially SECRET_KEY) are for development/testing convenience. It's assumed these would be overridden with secure, environment-specific values in a real deployment.
- **Database Choice:** SQLite was chosen as recommended for the bonus and its portability/simplicity, assuming its performance characteristics are adequate for the expected load during testing and grading.
- **Dependency Stability:** Assumed that the Python package versions listed in requirements.txt are stable and compatible with each other and the Python version used (e.g., 3.10 in the sample Dockerfile).

- **Docker Availability:** Assumed the grading environment has Docker and Docker Compose readily available and configured correctly.
- **Dockerfile Presence:** Assumed the provided Dockerfile is functional and correctly defines the build process.

4. Pros/Cons of Approach

Pros:

- **Modern & Efficient Framework:** FastAPI offers excellent performance, asynchronous support, automatic data validation via Pydantic, dependency injection, and built-in interactive documentation, significantly streamlining development and improving code quality.
- **Strong Typing & Validation:** Pydantic models enforce data schemas and enable clear, robust input validation directly tied to the data structures, reducing boilerplate code and catching errors early. The custom validators effectively implement the assignment's specific rules.
- **Comprehensive Security:** Implements multiple layers of security: secure password hashing (bcrypt), standard JWT-based authentication, role-based authorization, prevention of SQL injection via parameterized queries, and input validation designed to mitigate common XSS and other injection vectors.
- **Clear Separation of Concerns:** Configuration (config.py), database initialization (init_db.py), core application logic (main.py), and tests (test_phonebook.py) are well-separated, improving maintainability.
- **Automated Testing:** The extensive pytest suite covers functionality, validation rules (including assignment examples), security aspects, and edge cases, providing high confidence in the application's correctness and robustness.
- **Auditability & Monitoring:** Detailed audit logging provides essential visibility for tracking usage and troubleshooting. Log rotation prevents uncontrolled growth. The basic rate limiter adds a layer of abuse prevention.
- **Bonus Requirements Met:** Successfully integrates SQLite for persistence and utilizes parameterized queries, fulfilling the bonus criteria.
- **Ease of Setup (with Docker):** Docker and Docker Compose simplify the setup and ensure a consistent running environment across different machines, meeting a key assignment requirement.
- **Database Backup:** Includes a simple, automated database backup mechanism, adding a layer of data safety.

Cons:

- **Scalability Limitations:**
    - The in-memory rate limiter is not suitable for multi-process or distributed deployments.

    - SQLite can become a bottleneck under high concurrent write loads compared to dedicated client/server databases.
- **Basic Backup Strategy:** The file-copy backup method is rudimentary. It lacks features like point-in-time recovery or integration with database-specific backup tools. Retention is based solely on file count/time.
- **Regex Complexity/Limitations:** While comprehensive, the regex patterns for names and phones are complex and might be difficult to maintain or extend. There's always a risk of either being too strict (rejecting rare valid formats) or too lenient (allowing cleverly disguised invalid inputs).
- **JWT Handling:** While standard, storing JWTs purely in client-side storage can have security implications (e.g., XSS stealing tokens). More advanced session management or token handling strategies (like refresh tokens stored securely) might be used in production but are beyond this assignment's scope.
- **Configuration Security:** Storing the SECRET_KEY directly in config.py (even with environment variable override) might not meet stringent security standards; secrets management systems are preferred in production.