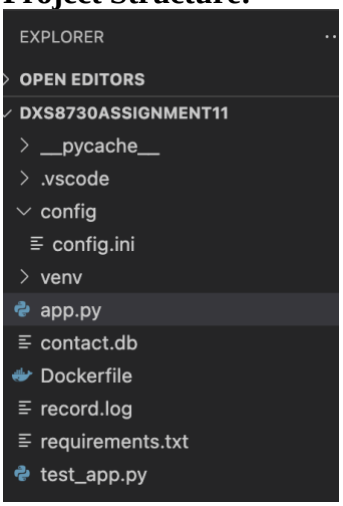
SECURE PROGRAMMING
ASSIGNMENT 11: INPUT VALIDATION

NAME: DANNASRI SRINIVASAN
ID: 1001698730

INTRODUCTION:
Created a python flask RestAPI connects with SQLite database. Integrated with docker container to build and tested my application through Postman.

Language: Python 3.9
IDE: Visual Studio Code
Build: Through Docker
Test IDE: Postman
DB: SQlLite3

Project Structure:



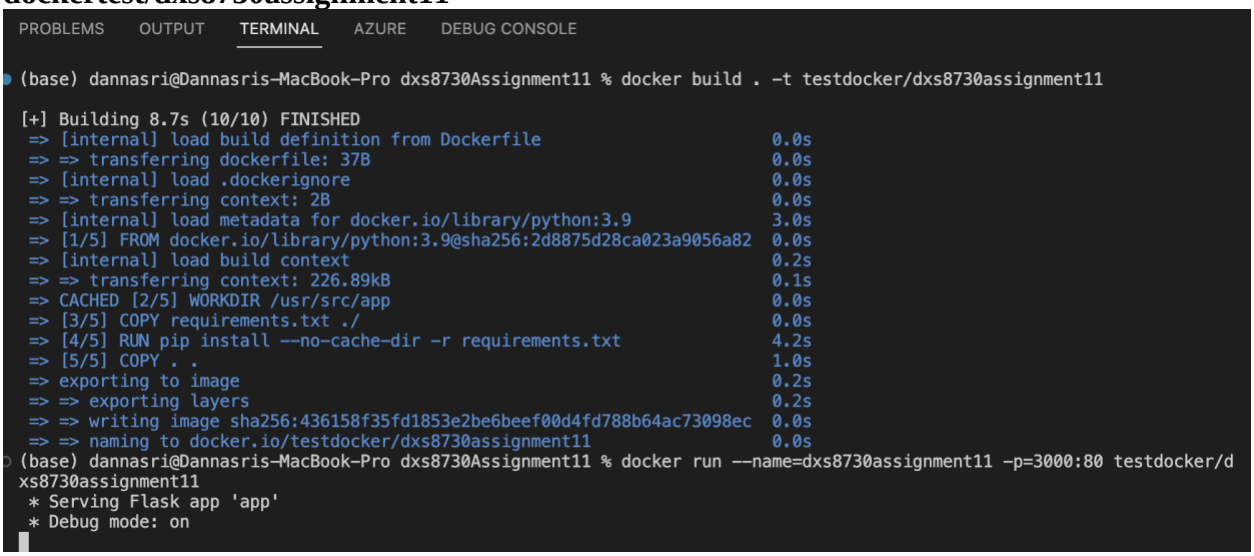
BUILDING AND RUNNING SOFTWARE:
I used docker containers to build and run the software.

My Docker File:

```
FROM python:3.9

WORKDIR /usr/src/app
COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY . .
EXPOSE 80
CMD [ "python", "./app.py" ]
```

Building through Docker:
Creating Docker Image: **docker build . -t dockertest/dxs8730assignment11**

Creating Docker Container: **docker run --name=dxs8730assignment11 -p=3000:80 dockertest/dxs8730assignment11**



Running with Docker Container:
Started with port: 3000.
URL: http://localhost:3000/

ASSUMPTIONS:

1. Both name and phone number are string elements.
2. Name should be unique before adding to the database.
3. We can enter name based on our requirements under acceptable and not acceptable format
4. We can enter phone number based on our requirements under acceptable and not acceptable format
5. Multiple users can have same phone number
6. Input should be passed in the JSON format with “name” and ‘’phonenumber” parameters.
7. Record needs to be inserted before trying to delete either by name or phone.
8. Delete by name deletes single record as name is unique
9. Delete by number deletes multiple records if exists as phone number is not unique many user can have same number.
10. Token needs to generate before any call and pass it in the header

PROS OF MY APPROCH:

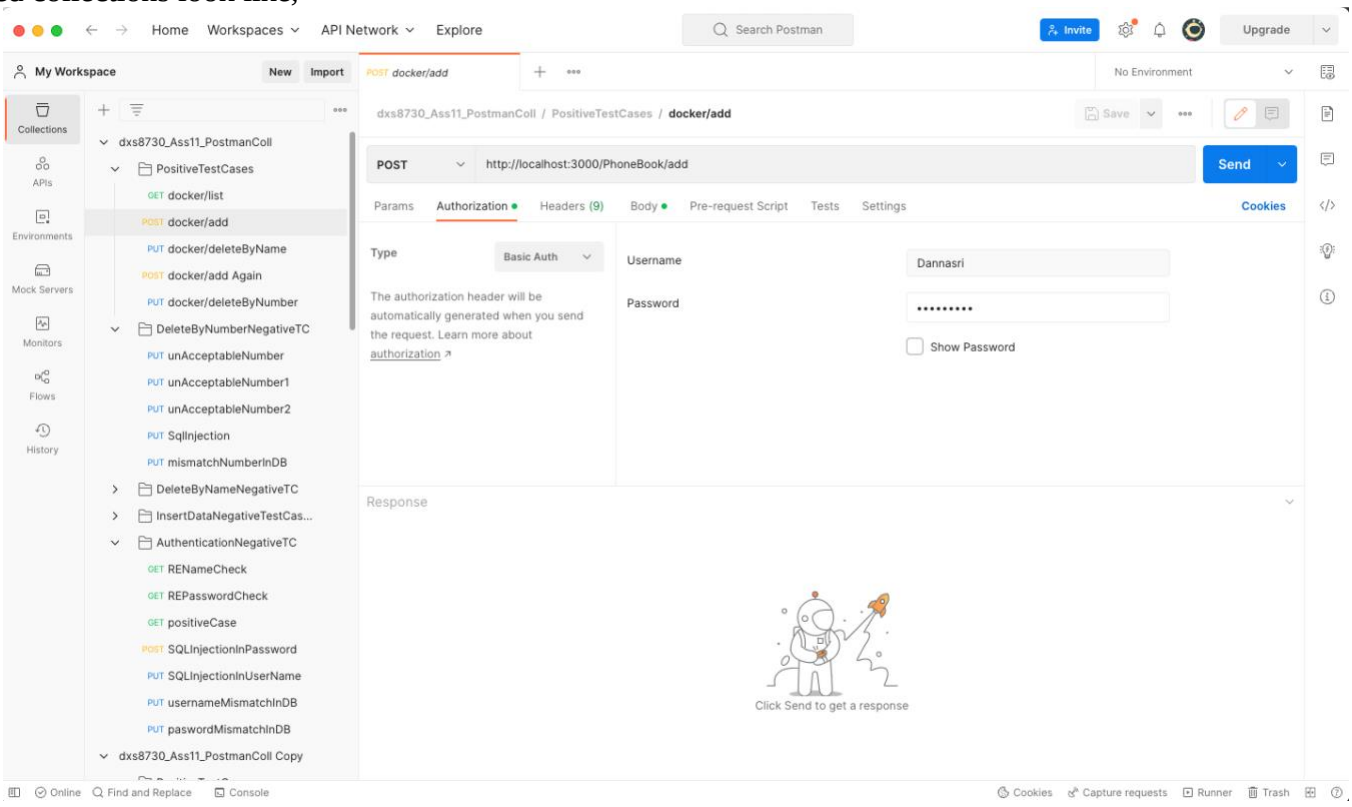
1. Usage of prepared statements reduces the SQL injection attack.
2. It allows only acceptable format for name and phone parameter, which reduces SQL attacks like droping the database.
3. Works for all input (name, phone) which specified in the requirements.pdf
4. Usage of authentication and authorization make it more secured.
5. Logging all operations makes it easier to understand the failure and debug the application.

CONS OF MY APPROCH:

1. Response time varies while testing in local and docker. Testing through docker gives less response time.
2. Program might fail for unknown formats
3. Tested through postman which is not much user friendly.
4. Created a basic database table with no proper key definitions.

UNIT TESTING:

I included postman collections named dxs8730Ass11.postman_collection.json
Can import in postman through File -> Import, Select the above postman collection and import.
Imported collections look like,



BONUS:

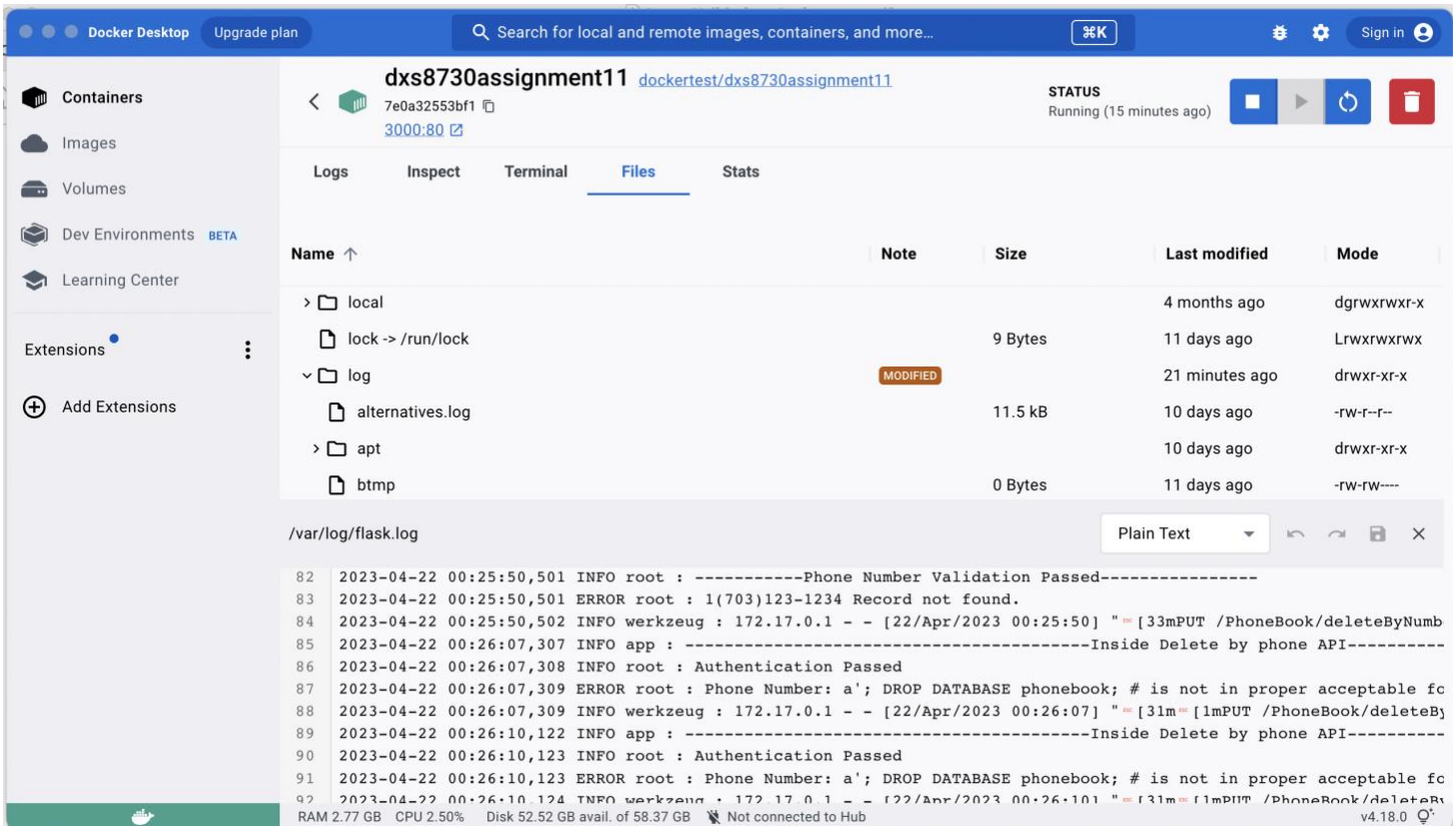
1. Used SQLite database to store and retrieve data
2. Used an API that supports parameterized queries.
3. Used basic authentication with token generation
4. Used Docker to build and run the application.

AUTHENTICATION:

For bonus authentication, I created a userdata table with name and password fields.
I added only one user to the userdata DB which is
name: Dannasri
password: Qwerty123
Other this credential everything else will fail.
In Postman: Authorization -> Basic Auth (copy paste above mentioned name and password)

AUDIT LOGGING:

My log file located under **docker files -> var/log/flask.log**



SAMPLE REQUEST AND RESPONSE:

Get All data:

Request:

URL: http://localhost:3000/PhoneBook/list

Method: GET

Authorization: Type: Basic Auth

Username: Dannasri

Password: Qwerty123

Response:

Status Code: 200


```
[
  {
    "name": "Cher",
    "phonenumber": "(703)111-2121"
  },
  {
    "name": "O'Malley, John F.",
    "phonenumber": "12345.12345"
  }
]
```


Insert Data:

Request:

URL: http://localhost:3000/PhoneBook/add

Method: POST

Authorization: Type: Basic Auth

Username: Dannasri

Password: Qwerty123

Body: raw -> JSON


```
{
```

```
  "name": "O'Malley, John F.",
  "phonenumber": "12345.12345"
}
```


Response:
Status Code: 200
Status Code: 404 (if data already exist)
Status Code: 400 (if name and number validation fails)


```
{
  "description": "Success",
  "message": "User Inserted successfully"
}
```


Delete by Name:

Request:
URL: http://localhost:3000/PhoneBook/deleteByName
Method: PUT
Authorization: Type: Basic Auth
Username: Dannasri
Password: Qwerty123
Body: raw -> JSON


```
{
  "name": "Cher"
}
```


Response:
Status Code: 200
Status Code: 404 (if data doesn’t exist)
Status Code: 400 (if name validation fails)


```
{
  "description": "Success",
  "message": "User deteted successfully"
}
```


Delete by PhoneNumber:

Request:
URL: http://localhost:3000/PhoneBook/deleteByNumber
Method: PUT
Authorization: Type: Basic Auth
Username: Dannasri
Password: Qwerty123
Body: raw -> JSON


```
{
  "phonenumber": "12345.12345"
}
```


Response:
Status Code: 200
Status Code: 404 (if data doesn’t exist)
Status Code: 400 (if name validation fails)


```
{
  "description": "Success",
```

```
    "message": "User deteted successfully"
}
```

DESCRIPTION OF HOW CODE WORKS:

Required libraries for the application:

app.py

```
from flask import Flask, request, jsonify
from configparser import ConfigParser
import logging, sqlite3, re ,os
```

Configuring Logger file:

```
logging.basicConfig(filename='/var/log/flask.log',
            level=logging.DEBUG, format='%(asctime)s %(levelname)s %(name)s : %(message)s')
```

My log file located under **docker files -> var/log/flask.log**
In the format configured timestamp, logging level, name(root/app/other) and the messages.
Used Info for success logs and error for error logs in the application.

Regular Expression Check:

1. **Name Check:**

```
def nameCheck(name):
    namecheck = re.compile("^[A-Z]\\'?([a-zA-Z]*?\\'?[a-zA-Z]*?\\,?[ ]?\\'?\\-?\\.?){1,3}$")
    return namecheck.match(name)
```

Starts with Uppercase character, followed by both lower and upper case.
\\,?[ ]?\\'?\\-?\\.? – allows single comma(,), single space([ ]), single apostrophe/single quote(‘), single hyphen(-), single period(.).
{1,3} – Both first and last name should be greater than 3 characters.

Note: Wrote based on the acceptable and not acceptable format provided in the requirements.pdf. Names which have apostrophe/single quote(‘) works only if we manually type apostrophe/single quote(‘), won’t work if we copy paste from the requirements.pdf. While copy pasting complier considering apostrophe/single quote(‘) as some other special character and returns false.

2. **Phone Number Check:**

```
def phoneCheck(phone):
    phoneCheck=re.compile("(^\\d{5}$)|(^\\d{5}\\.\\d{5}$)|(^\\+[1-9]{1,2}[ ]?\\(|^[1][ ]?\\(|^[0][1][1][ ][1]?[ ]?\\(?|^\\(?)([1-9]\\d{1,2})?\\)?[-]?(\\d{3})[-| ](\\d{4})$")
    return phoneCheck.match(phone)
```

Can contain digits 0 to 9. Allows special char like period(.), plus(+), single space([ ]), hypen(-). Allows min of 5 digit.
[1-9]\\d{1,2}- area code with 1-3 digits
(\\d{4}) - last four digits of phone number.

3. **Password Check:**

```
def passwordCheck(password):
    checkpass=re.compile("^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[a-zA-Z]).{8,32}$")
    return checkpass.match(password)
```

Does basic password check- minimum one uppercase, one lower case, one number and can also contain special characters

SQLITE DATABASE:

Connecting through Config:
I used config.ini to specify the database name and parsed using config parser and connected to SQLite database.
config/config.ini

```
[main]
appversion = 0.1.0
datasource = contact.db
```

app.py

```
#reading config.ini
CONFIG_PATH = os.path.join(os.path.dirname(__file__), 'config/config.ini')

def db_connect():
    config = ConfigParser()
#reading config_path
    config.read(CONFIG_PATH)
#connecting to database with the database name specified in the config.ini
    con = sqlite3.connect(config.get('main', 'datasource'))
    return con
```

Creating Phonebook table:
Contains 2 columns: NAME and PHONENUMBER.

```
def create_db_table():
    app.logger.info("Info log information")
    try:
        conn = db_connect()
        conn.execute('''
            CREATE TABLE phonebook (
                name TEXT NOT NULL,
                phonenumber TEXT NOT NULL
            );
        ''')

        conn.commit()
        logging.info("Phonebook table created successfully")
    except:
        logging.info("Phonebook table exists already or not created successfully")
    finally:
        conn.close()
```

Creating Userdata table:
Contains 2 columns: NAME and PASSWORD.

```
def create_user_table():
    try:
        conn = db_connect()
        conn.execute('''
```

```
            CREATE TABLE userdata (
                name TEXT NOT NULL,
                password TEXT NOT NULL
            );
        '')

        conn.commit()
        logging.info("Phonebook table created successfully")
    except:
        logging.info("Phonebook table exists already or not created successfully")
    finally:
        conn.close()
```

Getting all data from the table:

```
def get_data():
    users = []
    try:
#calling db_connect to establish the connection
        conn = db_connect()
        conn.row_factory = sqlite3.Row
        cur = conn.cursor()
#using select query to get all data from the table
        cur.execute("SELECT * FROM phonebook")
        rows = cur.fetchall()
        if len(rows) > 0:
            logging.info("listing all records in the database")
           #Iterating through all rows and storing name and phone in the dictionary
            for i in rows:
                user = {}
                user["name"] = i["name"]
                user["phonenumber"] = i["phonenumber"]
                logging.info("Name: %s  Phone Number: %s", user["name"], user["phonenumber"])
# appending using dictionary to the user list
                users.append(user)
        else:
#if table has no data return empty user list
            users = []
    except:
        users = []
        logging.error("Data not retrieved from Phonebook table")
    return users
```

Inserting a record to the table:

1. Before inserting did name and phone number check through the method which we created already. If name and phone number are not in acceptable return JSON response with 400 bad request and failure message to let user know name and phone are not in proper format.

```
if nameCheck(name):
        logging.info("----------Name Validation Passed----------")
        if phoneCheck(phone):
```

```
                logging.info("-----------Phone Number Validation Passed---------")
```

```
else:
        logging.error("Phone: %s is not in proper acceptable format.", phone)
        return jsonify({'description': 'Failure','message' : 'Phone Number is not in proper acceptable format'}), 400
    else:
        logging.error("Name: %s is not in proper acceptable format.", name)
        return jsonify({'description': 'Failure','message' : 'Name is not in proper acceptable format'}), 400
```

2. After name and phone check, I checked for data exists in table or not with the name specified by the user using select query. If the name already exist, returning 404 user already exists.
3. Then performing insert operation using insert query

```
cur.execute("INSERT INTO phonebook (name, phonenumber) VALUES (?, ?)", (name,
                        phone) )
            conn.commit()
            logging.info("Name: %s and Phone Number: %s added successfully", name, phone)
                return jsonify({'description': 'Success','message' : 'User Inserted successfully'}), 200
```

Deleting the record by user name:

Same like insert above, I checked name validation, record exist or not, then performed delete query with user name

```
conn.execute("DELETE from Phonebook WHERE name = ?", (name,))
            conn.commit()
            logging.info("%s Record Deleted Successfully.", name)
            return jsonify({'description': 'Success','message' : 'User deteted successfully'}), 200
```

Deleting the record by user phone number:

Same like insert above, I checked phone number validation, record exist or not, then performed delete query with user phone number

```
conn.execute("DELETE from Phonebook WHERE phonenumber = ?", (phone,))
            conn.commit()
            logging.info("%s, %s Record Deleted Successfully.", name, phone)
            return jsonify({'description': 'Success', 'message' : 'User deteted successfully'}), 200
```

Note: All database queries such as inert, delete by name, delete by phone and search through name all done through prepared statements to avoid and reduce the effect of SQL injection.

API'S

1. **Get users API:**

First check for the token in the authentication section, check with name and password check from above else returns Authentication failed. Calls get_data() method which has select query and returns the users list.

2. **Insert Users API:**

First check for the token in the authentication section, check with name and password check from above else returns Authentication failed. Calls insert_data() method which has all name and phone valid check, check for existing record, and insert query to insert user to the db and returns the success or failure message.

3. **Delete User by Name API:**

First check for the token in the authentication section, check with name and password check from above else returns Authentication failed. Calls delete_data_byName() method which has name valid check, check for existing record, and delete query to delete user from the db and returns the success or failure message.

4. Delete User by Phone Number API:

First check for the token in the authentication section, check with name and password check from above else returns Authentication failed. Calls delete_data_byPhoneNumber() method which has phone number valid check, check for existing record, and delete query to delete user from the DB and returns the success or failure message.