LAB 3 – Postman and APIs

In this lab you will be creating a Flask APP that adds, deletes and updates users in SQLite DB via API calls. Then you will get introduced to Postman, an advanced API client testing. You will be using Git and Github practices learned in the previous lab while building your code.

Create the Flask APP

REST API (or RESTful API), stands for Representational State Transfer — Application Programming Interface, allows apps to interact with resources via HTTP protocols. Such interactions are normally CRUD operations which implements specific business logics. So, an app running on either web, desktop or mobile requests a resource via HTTP methods and the response is returned normally in JSON format for interoperability. In our use case, our business logic includes adding user to our database — Create operation, getting list of users or a single user — Read operation, Updating users — Update operation and deleting users — Delete operation. All these will be done via HTTP requests.

Let's define our REST API endpoints:

#	Endpoints	HTTP Method	Description
1	http://localhost:5000/api/users	GET	Get the list of all users in the database
2	http://localhost:5000/api/users/<user_id>	GET	Get a single user record that matches the id
3	http://localhost:5000/api/users/add	POST	Create a new user record
4	http://localhost:5000/api/users/update	PUT	Update a user record
5	http://localhost:5000/api/users/delete/<user_id>	DELETE	Delete a user record

The table above shows the API endpoints for the User management app. The endpoints are pretty straightforward to understand and they perform database operations.

Setup a new project using VS Code and initiate your virtual environment, install flask and sqlite3

pip install db-sqlite3

Initiate git for this project, and create a repo on Github to host this project's code.

Now that we have the basic setups out of the way, let's take a look at the source codes. First, we'll focus on database implementation, then we'll focus on the REST API implementation. Basically, we need to create our database and table. Then, write functions for each of our business logic.

```
#!/usr/bin/python
import sqlite3

def connect_to_db():
    conn = sqlite3.connect('database.db')
    return conn

def create_db_table():
    try:
        conn = connect_to_db()
        conn.execute('''
            CREATE TABLE users (
                user_id INTEGER PRIMARY KEY NOT NULL,
                name TEXT NOT NULL,
                email TEXT NOT NULL,
                phone TEXT NOT NULL,
                address TEXT NOT NULL,
                country TEXT NOT NULL
            );
        ''')

        conn.commit()
        print("User table created successfully")
    except:
        print("User table creation failed - Maybe table")
    finally:
        conn.close()
```

The code snippet above implements the User database. The implementation is pretty straightforward. First we define a function that connects to db i.e. connect_to_db. This function creates the database if it doesn't exist and returns a connection object. This object will be required to perform other database operations. The *create_db_table* function connects to the database and creates table. The user table contains several fields including user_id (a unique identifier), name and email. The "changes" to the db is saved when the *commit* function is executed. Next, let's see the implementations of our business logics.

```
def insert_user(user):
    inserted_user = {}
    try:
        conn = connect_to_db()
        cur = conn.cursor()
        cur.execute("INSERT INTO users (name, email, phone,
address,
                    country) VALUES (?, ?, ?, ?, ?)",
(user['name'],
                    user['email'], user['phone'], user['address'],
                    user['country']) )
        conn.commit()
        inserted_user = get_user_by_id(cur.lastrowid)
    except:
        conn().rollback()

    finally:
        conn.close()

    return inserted_user
```

The code snippet above defines a function that adds a new user into the database table. The function takes a user object as an argument. The user object is simply a python dictionary as shown in the code snippet below.

```
user = {
    "name": "John Doe",
    "email": "jondoe@gamil.com",
    "phone": "067765434567",
    "address": "John Doe Street, Innsbruck",
    "country": "Austria"
}
```

Notice that the *user_id* is not inserted. This is because it's automatically generated. A *cursor* object is normally use to execute CRUD operations as it facilities traversal/processing over database records. In addition, we return the just inserted user by calling another function that returns a user given the user_id. We'll see this function implementation next.

```
def get_users():
    users = []
    try:
        conn = connect_to_db()
        conn.row_factory = sqlite3.Row
        cur = conn.cursor()
        cur.execute("SELECT * FROM users")
        rows = cur.fetchall()

        # convert row objects to dictionary
        for i in rows:
            user = {}
            user["user_id"] = i["user_id"]
            user["name"] = i["name"]
            user["email"] = i["email"]
            user["phone"] = i["phone"]
            user["address"] = i["address"]
            user["country"] = i["country"]
            users.append(user)

    except:
        users = []

    return users


def get_user_by_id(user_id):
    user = {}
    try:
        conn = connect_to_db()
        conn.row_factory = sqlite3.Row
        cur = conn.cursor()
        cur.execute("SELECT * FROM users WHERE user_id = ?",
                    (user_id,))
        row = cur.fetchone()

        # convert row object to dictionary
        user["user_id"] = row["user_id"]
        user["name"] = row["name"]
        user["email"] = row["email"]
        user["phone"] = row["phone"]
        user["address"] = row["address"]
        user["country"] = row["country"]
    except:
```

```
            user = {}

    return user
```

The code snippet above implements the feature to retrieve user(s) from the database.
The *get_user* function returns the list of all users from the database, while the *get_user_by_id* takes in a *user_id* as an argument and returns the user record matching the *user_id*. Let's see how to update a user next.

```
def update_user(user):
    updated_user = {}
    try:
        conn = connect_to_db()
        cur = conn.cursor()
        cur.execute("UPDATE users SET name = ?, email = ?, phone =
                    ?, address = ?, country = ? WHERE user_id =?",
                    (user["name"], user["email"], user["phone"],
                    user["address"], user["country"],
                    user["user_id"],))
        conn.commit()
        #return the user
        updated_user = get_user_by_id(user["user_id"])

    except:
        conn.rollback()
        updated_user = {}
    finally:
        conn.close()

    return updated_user
```

The code snippet implements the feature to update user record in the database. It takes in a user object as an argument, execute a SQL query to update the user with the new values and finally returns the updated user by calling the *get_user_by_id* function as seen previously. What if we want to remove or delete a user from the database? Let's see how to do that next.

```
def delete_user(user_id):
    message = {}
    try:
        conn = connect_to_db()
        conn.execute("DELETE from users WHERE user_id = ?",
                     (user_id,))
        conn.commit()
```

```
            message["status"] = "User deleted
successfully"
    except:
        conn.rollback()
        message["status"] = "Cannot delete user"
    finally:
        conn.close()

    return message
```

The code snippet above implements the delete user feature. We define a function that takes in a user id as an argument, remove the user from the database (SQL DELETE query) and return a message back.

Let's see how to make these function available via REST API endpoints next.

Run the code and make sure the database is created. You will notice a database.db file added to your VS code files

Once done, commit your code to Github.

REST API implementation

The REST api endpoints expose the database functions created in the previous section. The table below maps the database functions to the API endpoints.

Create a new git branch, pull the DB code and edit it to add the REST code.

User management API endpoints with database function

```
from flask import Flask, request, jsonify #added to top of file

from flask_cors import CORS #added to top of file

app = Flask(__name__)

CORS(app, resources={r"/*": {"origins": "*"}})


@app.route('/api/users', methods=['GET'])

def api_get_users():

   return jsonify(get_users())


@app.route('/api/users/<user_id>', methods=['GET'])
```

```
def api_get_user(user_id):
    return jsonify(get_user_by_id(user_id))


@app.route('/api/users/add',  methods = ['POST'])
def api_add_user():
    user = request.get_json()
    return jsonify(insert_user(user))


@app.route('/api/users/update',  methods = ['PUT'])
def api_update_user():
    user = request.get_json()
    return jsonify(update_user(user))


@app.route('/api/users/delete/<user_id>',  methods = ['DELETE'])
def api_delete_user(user_id):
    return jsonify(delete_user(user_id))


if __name__ == "__main__":
    #app.debug = True
    #app.run(debug=True)
    app.run() #run app
```

The code snippet above implements our API endpoints. First we import the necessary packages. The API endpoints receive JSON objects as request and also return JSON objects. For this reason, we convert Python dictionary to JSON using the jsonify package. Next we create the Flask app and configure the app to allows access to our endpoints from any ip-address using CORS.

In order to create an endpoint, we first define the route. Let's take the first endpoint as an example. We define the root as '/api/users' and set the method to GET. The default method is GET so it is also fine to leave out the method parameter.

The root (or endpoint) specifies an api function i.e. api_get_users which uses the database function get_user to return the list of users in our database. Notice that, a JSON object is returned. The other endpoints are straightforward.

Test using your browser the Get methods.

Once tested successfully, merge with main branch and commit.

Postman

Go over the Postman first steps tutorial by covering the sections Overview till Create your first workspace

https://learning.postman.com/docs/getting-started/first-steps/overview/

learn how to send requests in Postman

https://learning.postman.com/docs/sending-requests/requests/

Exercise

1- Create a collection named “Falsk user app”
2- Add the requests that test our app (get, delete, etc.)
3- Create examples for each request (hint: choose the save example after running a request)
4- Save the URL as an environment variable and use it in your collection.