BACKEND TASK SUBMISSION

RAJA-MANTRI-CHOR-SIPAHI

(Problem 2)

Name-Shreyas Menon(25BDS0102)

- Backend Framework used-Flask
- Database used-SQLite
- APIs were used for Postman to communicate with the backend code(for room creation, joining, role assignment and guessing).

  Flask is a lightweight Python framework suitable for REST APIs, and SQLite is a simple file-based database ideal for small applications and rapid development.

Steps followed-

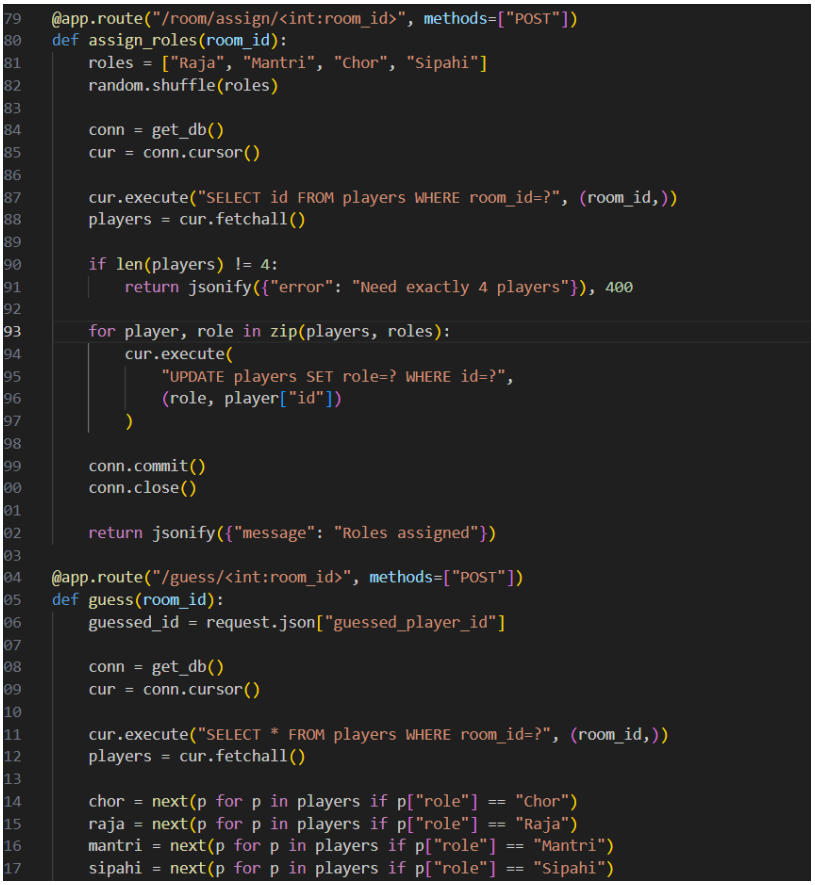
1)Source Code-

```python
from flask import Flask, request, jsonify
import sqlite3
import random

app = Flask(__name__)

DB = "game.db"

# ---------- DATABASE ----------
def get_db():
    conn = sqlite3.connect(DB)
    conn.row_factory = sqlite3.Row
    return conn

def init_db():
    conn = get_db()
    cur = conn.cursor()

    cur.execute("""
    CREATE TABLE IF NOT EXISTS rooms (
        id INTEGER PRIMARY KEY AUTOINCREMENT
    )
    """)

    cur.execute("""
    CREATE TABLE IF NOT EXISTS players (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        room_id INTEGER,
        name TEXT,
        role TEXT,
        points INTEGER DEFAULT 0
    )
    """)

    conn.commit()
    conn.close()

init_db()
```

- Flask is used for creating a backend server(basically so we can run our python code on a webserver which stays online and waits for requests)
- Request is used for reading data sent from postman and jsonify sends a response as JSON
- Sqlite3 is for the SQL database
- get_db() opens a connection to the SQLite database and returns it so other functions can read and write data.
- Innit_db is a function that creates the required databse tables if they do not already exist
- The database initialization function[init_db()] is called at the start to ensure the backend is ready before handling requests.

```python
# ---------- ROUTES ----------
@app.route("/")
def home():
    return "Backend running (Python + SQLite)"

@app.route("/room/create", methods=["POST"])
def create_room():
    conn = get_db()
    cur = conn.cursor()
    cur.execute("INSERT INTO rooms DEFAULT VALUES")
    conn.commit()
    room_id = cur.lastrowid
    conn.close()
    return jsonify({"room_id": room_id})

@app.route("/room/join", methods=["POST"])
def join_room():
    data = request.json
    room_id = data["room_id"]
    name = data["player_name"]

    conn = get_db()
    cur = conn.cursor()

    cur.execute("SELECT COUNT(*) FROM players WHERE room_id=?", (room_id,))
    if cur.fetchone()[0] >= 4:
        return jsonify({"error": "Room full"}), 400

    cur.execute(
        "INSERT INTO players (room_id, name) VALUES (?, ?)",
        (room_id, name)
    )

    conn.commit()
    player_id = cur.lastrowid
    conn.close()

    return jsonify({"player_id": player_id})
```

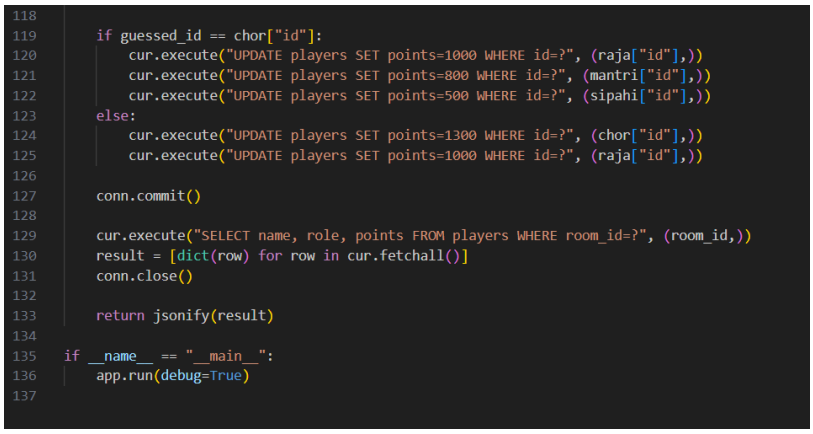
- Here we are using routes which allow external clients such as Postman or a frontend application to trigger specific backend logic through HTTP requests.
- basically, when a route is triggered(called) a specific task will be done
- the / route checks if the backend is running
- the second route will basically create the new gameroom at the start of the game(when we trigger it with a POST request in postman or any other method)
- similarly there are roots when a player wants to enter a game,etc..
- routes basically define the step-by-step flow of the game, from room creation to final score calculation
- conn and cur are variables initialized to handle database operations, where conn represents the database connection and cur is used to execute SQL queries.
- Data=request.json and then room_id-data[“room id”] basically reads the JSON data sent by the client to extract inputs like name etc..



```python
@app.route("/room/assign/<int:room_id>", methods=["POST"])
def assign_roles(room_id):
    roles = ["Raja", "Mantri", "Chor", "Sipahi"]
    random.shuffle(roles)

    conn = get_db()
    cur = conn.cursor()

    cur.execute("SELECT id FROM players WHERE room_id=?", (room_id,))
    players = cur.fetchall()

    if len(players) != 4:
        return jsonify({"error": "Need exactly 4 players"}), 400

    for player, role in zip(players, roles):
        cur.execute(
            "UPDATE players SET role=? WHERE id=?",
            (role, player["id"])
        )

    conn.commit()
    conn.close()

    return jsonify({"message": "Roles assigned"})

@app.route("/guess/<int:room_id>", methods=["POST"])
def guess(room_id):
    guessed_id = request.json["guessed_player_id"]

    conn = get_db()
    cur = conn.cursor()

    cur.execute("SELECT * FROM players WHERE room_id=?", (room_id,))
    players = cur.fetchall()

    chor = next(p for p in players if p["role"] == "Chor")
    raja = next(p for p in players if p["role"] == "Raja")
    mantri = next(p for p in players if p["role"] == "Mantri")
    sipahi = next(p for p in players if p["role"] == "Sipahi")
```

- Here we first assign the roles randomly using the random() function
- Validation checks are used to enforce game rules such as maximum players and required player count.
- In short, the role assignment route randomly assigns roles to players and validates that exactly four players are present.

- The guessing route processes the Mantri's guess, applies the game scoring rules, updates the database, and returns the final results.



```python

    if guessed_id == chor["id"]:
        cur.execute("UPDATE players SET points=1000 WHERE id=?", (raja["id"],))
        cur.execute("UPDATE players SET points=800 WHERE id=?", (mantri["id"],))
        cur.execute("UPDATE players SET points=500 WHERE id=?", (sipahi["id"],))
    else:
        cur.execute("UPDATE players SET points=1300 WHERE id=?", (chor["id"],))
        cur.execute("UPDATE players SET points=1000 WHERE id=?", (raja["id"],))

    conn.commit()

    cur.execute("SELECT name, role, points FROM players WHERE room_id=?", (room_id,))
    result = [dict(row) for row in cur.fetchall()]
    conn.close()

    return jsonify(result)

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

```

- Here we write the code for the guessing part
- We get the guessing simulation to work and then calculate the points

In conclusion, the backend uses Flask routes to manage game flow, SQLite for data storage, and Postman for testing the API endpoints.

All API endpoints were tested using Postman by sending HTTP requests and verifying the JSON responses. This allowed the backend functionality to be validated without requiring a frontend.

2)command window configuration

```
C:\Users\Menon>cd Desktop\raja_mantri_sql

C:\Users\Menon\Desktop\raja_mantri_sql>dir
 Volume in drive C is Windows
 Volume Serial Number is 5AB2-DD55

 Directory of C:\Users\Menon\Desktop\raja_mantri_sql

13-12-2025  13:27    <DIR>          .
13-12-2025  13:26    <DIR>          ..
13-12-2025  13:28             3,628 app.py
               1 File(s)          3,628 bytes
               2 Dir(s)  111,099,351,040 bytes free

C:\Users\Menon\Desktop\raja_mantri_sql>python app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
```

Here, I first navigated to a folder called raja_mantri_sql which I had created earlier.

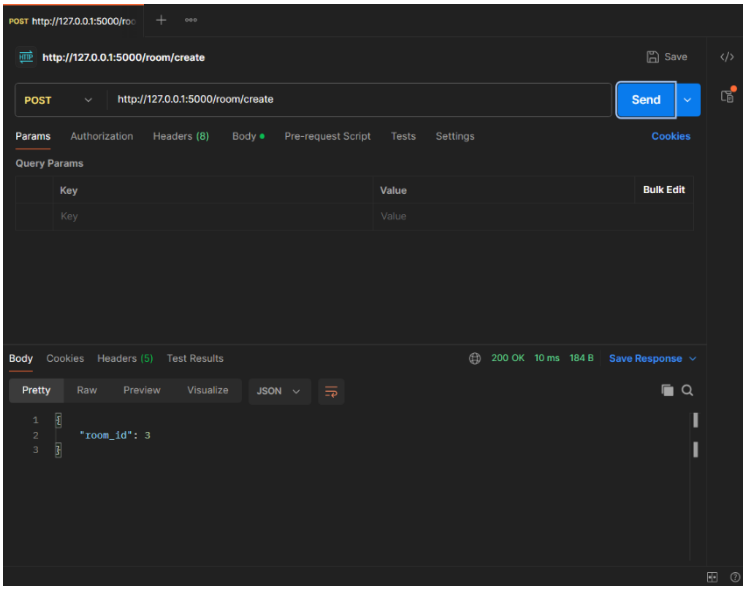
Inside this folder I created a file called app.py which contains the source code.

By using the command python app.py I basically executed the backend source code and started the Flask development server.

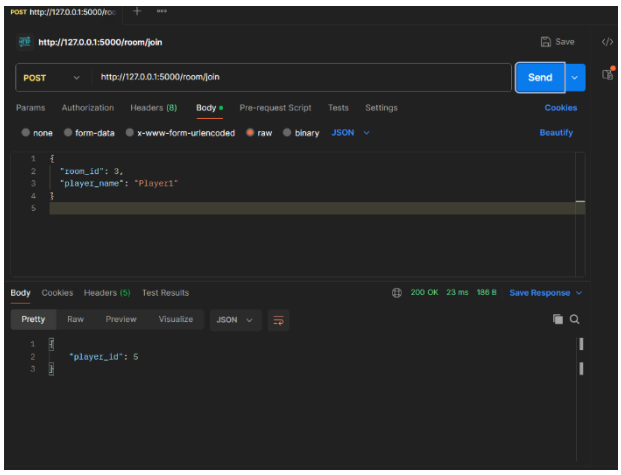
The API endpoints were then available at locally at http://127.0.0.1:5000 so that I could test it using Postman

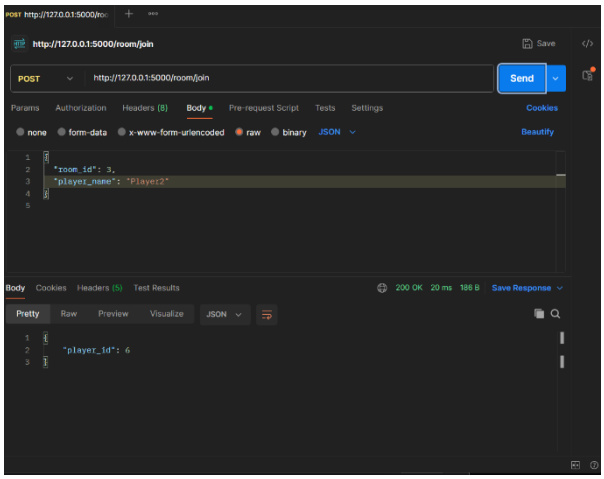
3)Testing it in postman

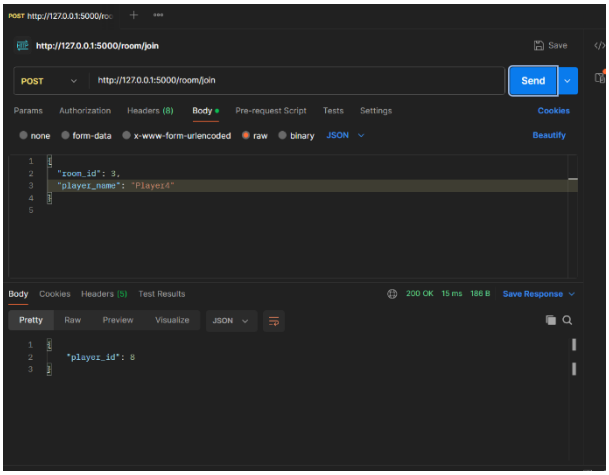
i)Creating the room



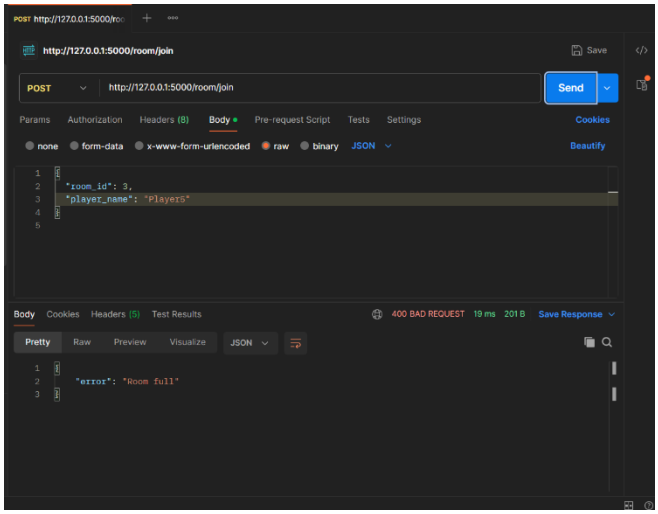
ii)Screen shots of the players joining the room



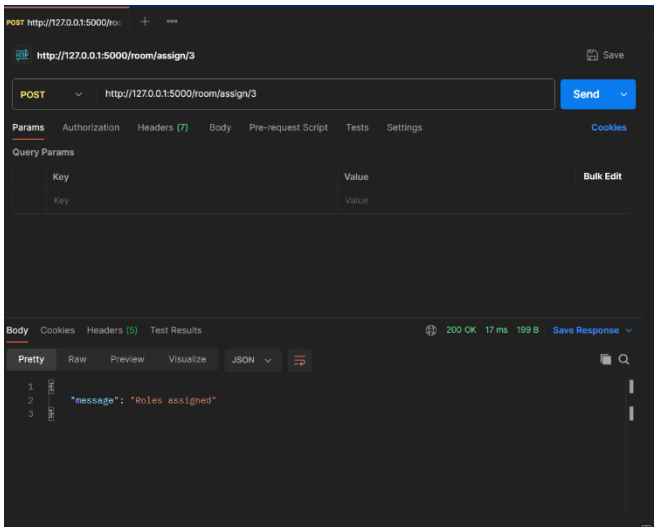




it shows error when more than 4 players join



iii)assigning the roles



iv)guessing and giving points