Backend Implementation of Parking_System

github : Park_Vision_Backend

INTRODUCTION

The **Parking System API** is a backend solution designed to manage a parking lot system efficiently. The API provides functionalities to handle the management of parking lots, floors, rows, parking slots, user registration, vehicle management, and various related features. This system is intended for both users and administrators to ensure smooth parking operations and facilitate parking space availability and management.

Features of the Parking System:

- **User Registration and Login** : Users can register and log in to access the system. They can view available parking slots, park their vehicles, and manage their accounts.
- **Parking Lot Management**: Admins can add, update, or delete parking lots. Each parking lot consists of multiple floors, rows, and parking slots.
- **Floor, Row, and Slot Management**: The system supports defining the structure of each parking lot, including floors, rows, and parking slots, allowing for efficient space utilization.
- **Parking Session Management**: Users can initiate and end parking sessions, keeping track of parking durations, payment, and available slots.
- **RESTful API**: The system uses REST API routes to allow users and administrators to interact with the backend.

Technologies Used:

- **Backend Framework**: Built using **Flask**, a lightweight web framework for Python, which simplifies the creation of RESTful APIs.
- **Database**: The system uses **MySQL** for storing data related to users, parking lots, floors, rows, slots, and parking sessions.
- **Containerization**: The application is **dockerized** to ensure portability, ease of deployment, and consistency across different environments. Docker images are used to run the application inside isolated containers.
- **CI/CD Integration**: GitHub Actions are used for continuous integration and deployment (CI/CD) pipelines, automating testing, building, and deployment of the app to production.
- **Testing**: The system is tested using the **pytest** framework to ensure reliable and bug-free operation, covering various API routes and functionalities.

🗀 Project Structure

```
🗀 Parking_system/
|-- 🗀 .git/              # Git version control directory
|-- 🗀 .github/ └── 🗀 workflows/ └── 🗎 cicd.yaml
|-- 🗎 .gitignore         # Git ignore file for excluding files from repo
|-- 🗎 app.py             # Main Flask app with routes and models
|-- 🗎 Dockerfile         # Instructions to build Docker image
|-- 🗎 requirements.txt         # Python dependencies
|-- 🗎 run.py             # Script to run the Flask app
|-- 🗎 test_app.py        # Pytest test suite for the application
```

Understanding Tables and Schemas

MySQL: Create Database

- Bash
  - **mysql -u your_username -p**
- sql
  - **CREATE DATABASE parking_system;**
  - **USE parking_system;**

The database is designed to handle parking lot management, user registrations, parking sessions, and parking space availability. The schema is broken down into several tables, each serving a specific purpose within the system.

here are multiple ways to create database tables and populate them with initial data in this project:

- **Manually Create Tables Using SQL Script**

1)parkinglots_details

```
CREATE TABLE parkinglots_details (
    parking_id INT PRIMARY KEY AUTO_INCREMENT,
    parking_name VARCHAR(255),
    city VARCHAR(100),
    parking_location VARCHAR(255),
    address_1 VARCHAR(255),
    address_2 VARCHAR(255),
    latitude DOUBLE,
    longitude DOUBLE,
    physical_appearance VARCHAR(255),
```

```
    parking_ownership VARCHAR(100),
    parking_surface VARCHAR(50),
    has_cctv VARCHAR(10),
    has_boom_barrier VARCHAR(10),
    ticket_generated VARCHAR(50),
    entry_exit_gates TEXT,
    weekly_off VARCHAR(50),
    parking_timing VARCHAR(50),
    vehicle_types VARCHAR(255),
    car_capacity INT,
    two_wheeler_capacity INT,
    parking_type VARCHAR(50),
    payment_modes VARCHAR(255),
    car_parking_charge VARCHAR(50),
    two_wheeler_parking_charge VARCHAR(50),
    allows_prepaid_passes VARCHAR(10),
    provides_valet_services VARCHAR(10),
    notes TEXT,
    total_slots INT,
    available_slots INT
);


2) floors
CREATE TABLE floors (
    floor_id INT PRIMARY KEY AUTO_INCREMENT,
    parking_id INT,
    floor_number VARCHAR(50),
    total_slots INT,
    available_slots INT,
    FOREIGN KEY (parking_id) REFERENCES parkinglots_details(parking_id)
);
3) parking_rows
CREATE TABLE parking_rows (
    row_id INT PRIMARY KEY AUTO_INCREMENT,
    parking_id INT,
```

```
    row_number VARCHAR(50),
    floor_id INT,
    FOREIGN KEY (parking_id) REFERENCES parkinglots_details(parking_id),
    FOREIGN KEY (floor_id) REFERENCES floors(floor_id)
);
```

4) slots

```
CREATE TABLE slots (
    slot_id INT PRIMARY KEY AUTO_INCREMENT,
    parking_id INT,
    row_id INT,
    slot_number VARCHAR(50),
    is_available TINYINT(1) DEFAULT 1,
    FOREIGN KEY (parking_id) REFERENCES parkinglots_details(parking_id),
    FOREIGN KEY (row_id) REFERENCES parking_rows(row_id)
);
```

5) users

```
CREATE TABLE users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(100) UNIQUE,
    password VARCHAR(255),
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

6) parkingsessions

```
CREATE TABLE parkingsessions (
    session_id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT,
    parking_id INT,
    slot_id INT,
    entry_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    exit_time TIMESTAMP,
    car_number VARCHAR(20),
    payment_status VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (parking_id) REFERENCES parkinglots_details(parking_id),
    FOREIGN KEY (slot_id) REFERENCES slots(slot_id)
);
```

- **Automatically Create Tables Using SQLAlchemy**

Instead of manually creating table schemas, you can automate the process by adding the following code to your app.py file:

```
’’’
with app.app_context():
    db.create_all()
’’’
```

The db.create_all() method scans your SQLAlchemy models and creates the corresponding tables in the database if they don’t already exist. Wrapping it inside with app.app_context(): ensures that the application context is active, which is necessary for SQLAlchemy to access the Flask app’s configuration and database connection.

- **Database Seeding via SQL Dump** or simply **Importing a SQL Dump File**

You can also **use a raw SQL script to populate data into your tables**. For this project, a sample data file is provided:
Parking-system dump.sql : Follow link to download

You can execute this SQL file manually using a database client (such as MySQL Workbench or DBeaver) or from the terminal:

bash >>> mysql -u your_username -p your_database_name < Parking_system_dump.sql

First, create the database manually then this command **automatically creates the tables and inserts data** into them, based on the SQL statements in the .sql file.

Parking System API (app.py)

This **Flask-based Parking System API** allows users to register, book parking slots, and manage parking infrastructure—including lots, floors, rows, and individual parking slots. The application uses **MySQL** as the backend database and leverages **SQLAlchemy ORM** for efficient database interaction. The system is designed following the **modular application factory pattern** to support better scalability, maintainability, and testing.

Key Technologies Used

- **Flask** – Lightweight web framework for building RESTful APIs.
- **Flask-SQLAlchemy** – ORM tool for handling database models and queries.
- **MySQL** – Relational database system used for storing application data.
- **Application Factory Pattern** – Initializes the app using a configurable create_app() function.
- **REST API Design** – Provides full CRUD operations on all parking-related data entities.

Configuration & Initialization

- The app uses create_app() to configure and return a Flask app instance.
- The MySQL URI is dynamically constructed using username, password, host, and DB name.
- db.create_all() is used within the app context to generate tables if they don't exist.

Database Models

➢ **ParkingLot**
Stores metadata about each parking location like name, location, CCTV availability, capacities, and charges.
➢ **Floor**
Each parking lot can have multiple floors with a defined number of total and available slots.
➢ **Row**
Each floor contains several rows identified by a row number.
➢ **Slot**
Represents an individual parking slot, linked to a row. Tracks availability (is_available).
➢ **ParkingSession**
Records parking activity including car number, user, slot, and timestamps for entry/exit.
➢ **User**
Holds user data for registration and login.

API Endpoints

🔑 Public Routes

- GET / – Welcome page with endpoint documentation.

🚗 Parking Lot

- GET /parkinglots/ – List all parking lots.
- POST /parkinglots/ – Add a new parking lot.

🏢 **Floors**

- GET /floors/ – List all floors.
- GET /floors/<parking_id> – List floors in a specific parking lot.
- POST /floors/ – Add a floor to a parking lot.

🚪 **Rows**

- GET /rows/ – List all rows.
- GET /rows/<floor_id> – List rows in a specific floor.
- POST /rows/ – Add a row to a floor.

🅿️ **Slots**

- GET /slots/ – List all slots.
- GET /slots/<row_id> – List available slots in a row.
- POST /slots/ – Create a new slot.

🎟️ **Parking Sessions**

- GET /sessions/ – View all parking sessions.
- POST /Park_car/ – Book a slot.
- PUT /Remove_car/<session_id>/exit – Release a slot (mark exit time and set is_available to true).

👤 **User Management**

- POST /users/register – Register a new user.
- POST /users/login – Login with email and password.
- GET /users/ – List all users.
- GET /users/<user_id> – Get a specific user's details.

Setting Up the Project Environment

Before running the application, you need to set up your environment and install the required dependencies.

Step 1: Create a Virtual Environment (if needed)

It is recommended to use a virtual environment to manage project dependencies. If you haven't already created a virtual environment, you can do so with the following steps:

- **Create the Virtual Environment**
  In your project directory, run the following command to create a new virtual environment:

```
>>> python -m venv venv
```

This will create a venv directory where the virtual environment will be stored.

- **Activate the Virtual Environment**
  After creating the virtual environment, activate it with the appropriate command based on your operating system:
  - **For Windows:**

```
.\venv\Scripts\activate
```

You should see the virtual environment's name (venv) in your terminal prompt, indicating that it is active.

Step 2: Install Dependencies
Once the virtual environment is activated, you can install the required dependencies from the requirements.txt file.

1. **Install Dependencies**
   Run the following command to install the dependencies:
   >>> pip install -r requirements.txt
   This will install all the necessary packages listed in requirements.txt.

Step 3: Verify Installation
You can verify that all dependencies are installed correctly by checking the installed packages:

>>> pip list

Running the Application

To start the application, follow these steps:

1. **Run the Application**
   In your terminal, navigate to the project directory and run the following command:
   >>> python run.py

2. **Access the Application**
   The application will be available at:

   >>> http://0.0.0.0:5000

File Structure Overview

- **app.py**
  This is the main application file. It defines the application's models and route logic. It follows the application factory pattern and exposes a create_app() function that sets up and returns a Flask app instance.

- **run.py**
  The run.py file serves as the **entry point** for the application. It imports the create_app() function from app.py, creates the app instance, and starts the server.

Why Use run.py?

The run.py file helps to separate the application creation logic from the execution logic

User-Friendly Interface

When you navigate to this URL (http://0.0.0.0:5000), you will be redirected to a browser page that displays the available **GET API endpoints**. The interface is designed to be user-friendly, allowing you to easily interact with the API by clicking the relevant links for each endpoint. This makes it easier to explore and test the available functionality without needing to manually type URLs.

Interacting with POST and PUT API Endpoints: For testing and interacting with the **POST** and **PUT** API endpoints, we recommend using tools such as **Thunder Client**, **Postman**, or **curl**. These tools allow you to send HTTP requests with the appropriate data payloads.

Using Thunder Client (VS Code Extension)

1. **Install Thunder Client**

   Thunder Client is a lightweight REST API client extension for Visual Studio Code. You can easily install it from the **VS Code Extensions Marketplace**.

2. **Making POST or PUT Requests**

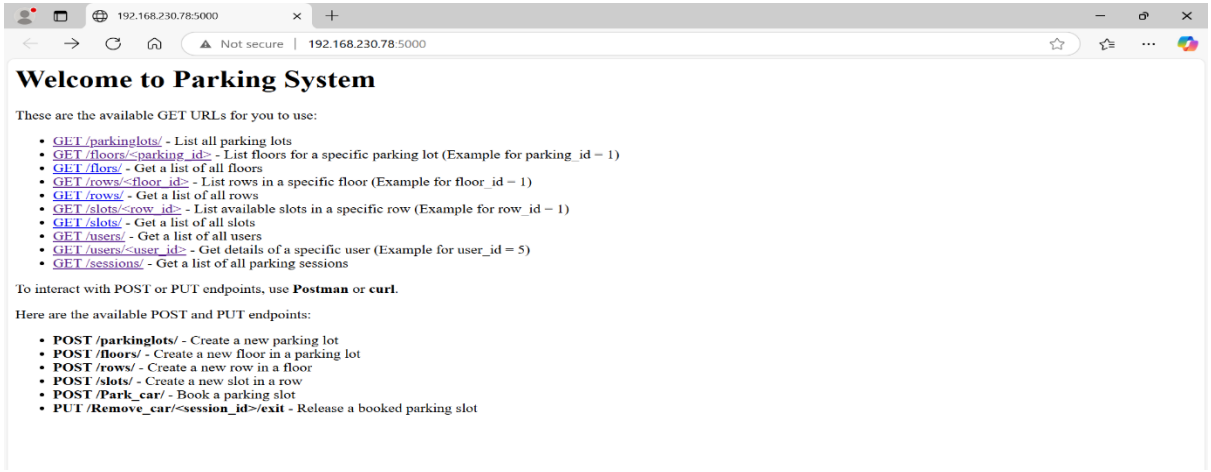
   After installation, open Thunder Client and create a new request. Select the request type (POST or PUT), enter the appropriate endpoint URL (e.g., http://0.0.0.0:5000/your-endpoint), and provide any necessary data in the body (in JSON format).

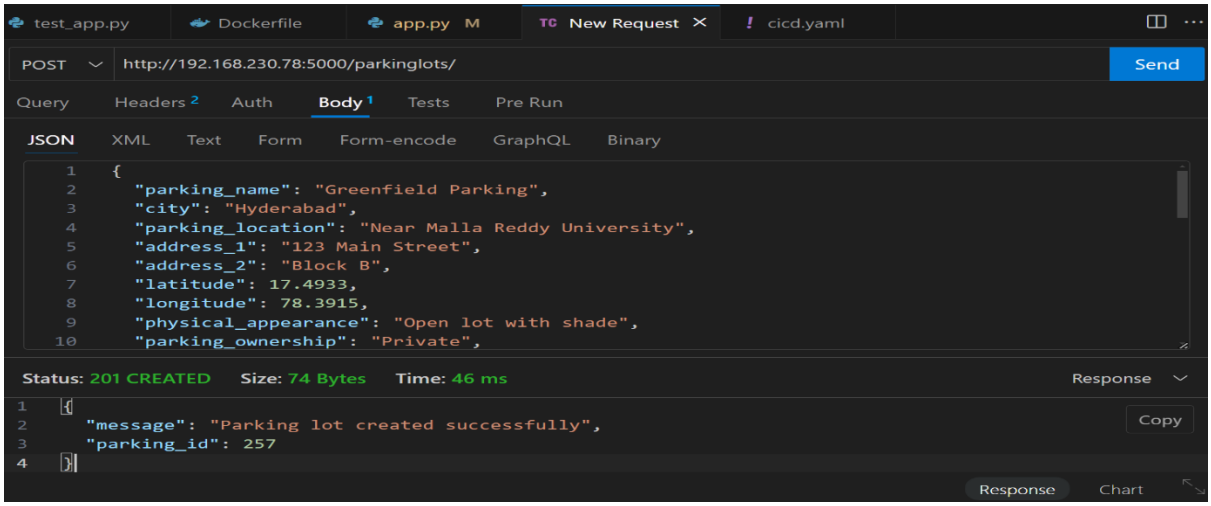
Screenshots of API Endpoint Results

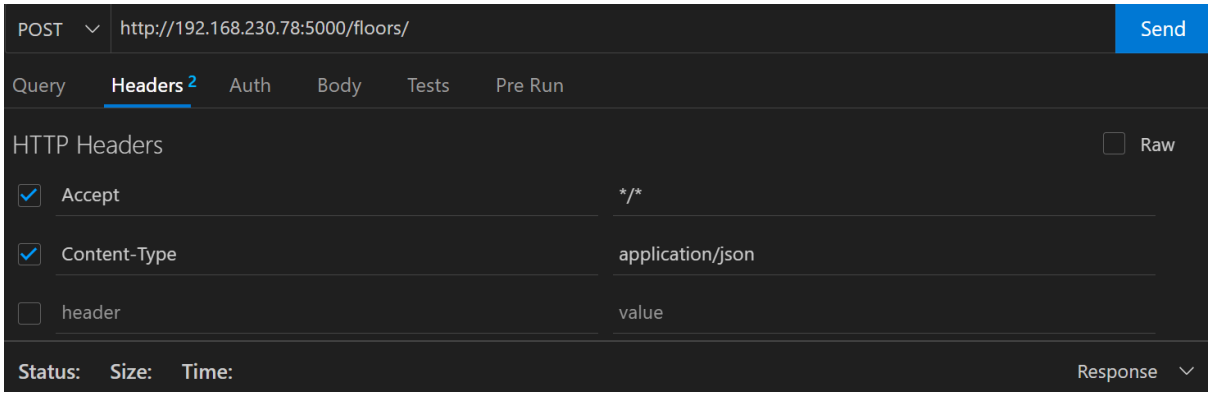
Below are a few screenshots capturing the results of various API endpoints. These images demonstrate the responses returned by the API for different HTTP methods (GET, POST, PUT).

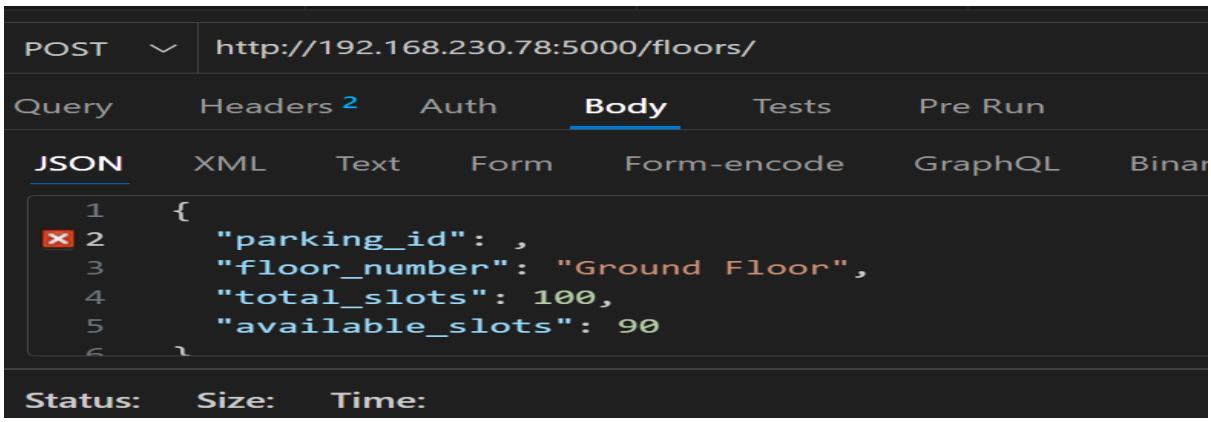
1. **GET API Endpoint Response**
2. **POST API Endpoint Response**
3. **PUT API Endpoint Response**

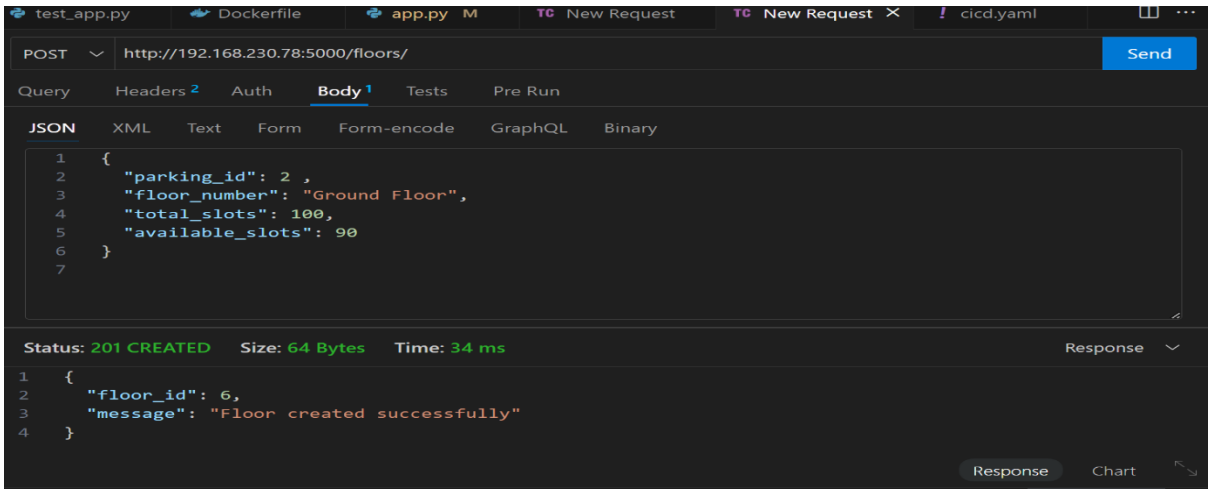
These screenshots show the structure of the response, including the returned data, status codes, and_headers.

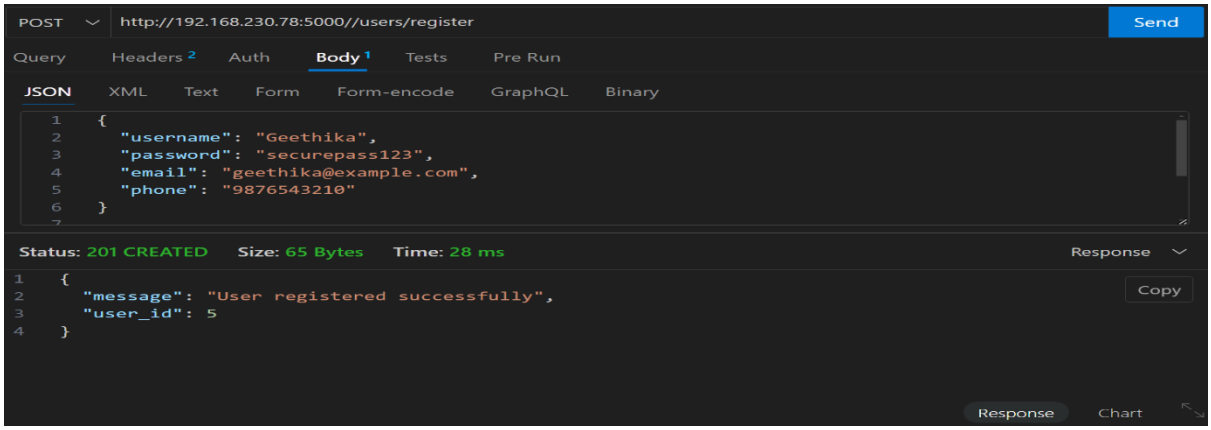


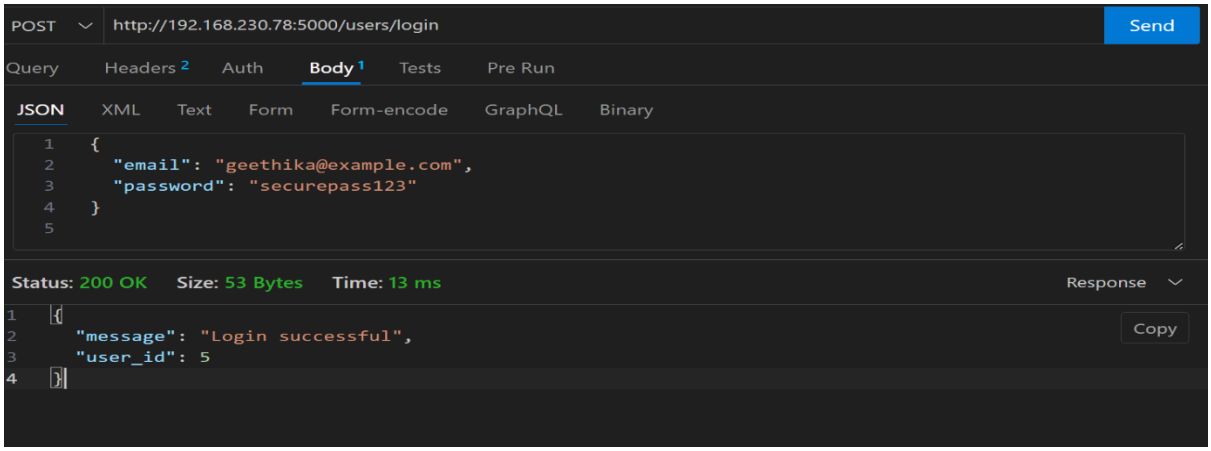

test_app.py
Dockerfile
app.py M
New Request
cicd.yaml
POST
http://192.168.230.78:5000/parkinglots/
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
{
"parking_name": "Greenfield Parking",
"city": "Hyderabad",
"parking_location": "Near Malla Reddy University",
"address_1": "123 Main Street",
"address_2": "Block B",
"latitude": 17.4933,
"longitude": 78.3915,
"physical_appearance": "Open lot with shade",
"parking_ownership": "Private",
Status: 201 CREATED
Size: 74 Bytes
Time: 46 ms
Response
{
"message": "Parking lot created successfully",
"parking_id": 257
}
Copy
Response
Chart

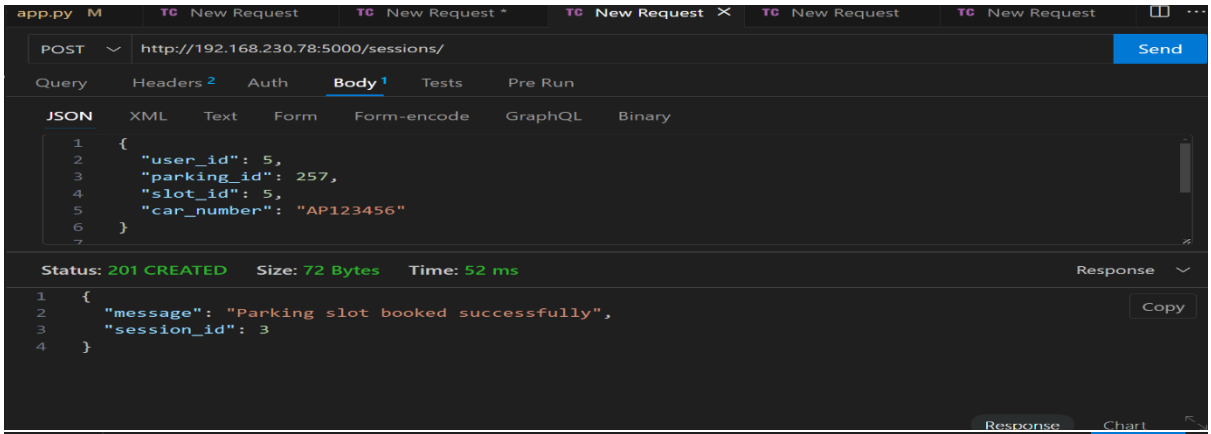

POST
http://192.168.230.78:5000/floors/
Send
Query
Headers 2
Auth
Body
Tests
Pre Run
HTTP Headers
Raw
Accept
/
Content-Type
application/json
header
value
Status:
Size:
Time:
Response

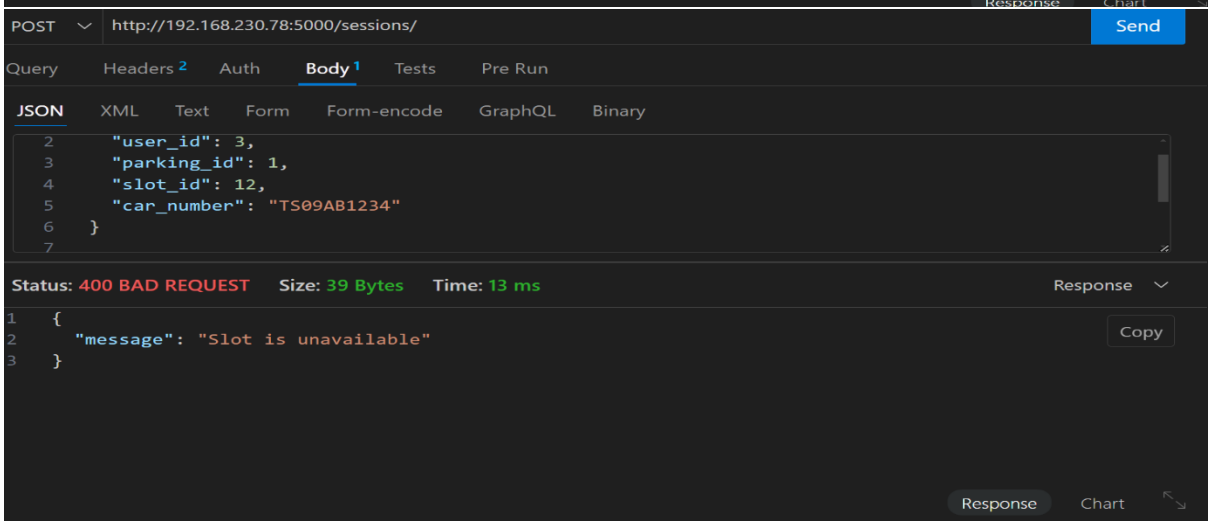

POST
http://192.168.230.78:5000/floors/
Query
Headers 2
Auth
Body
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
{
"parking_id": ,
"floor_number": "Ground Floor",
"total_slots": 100,
"available_slots": 90
}
Status:
Size:
Time:

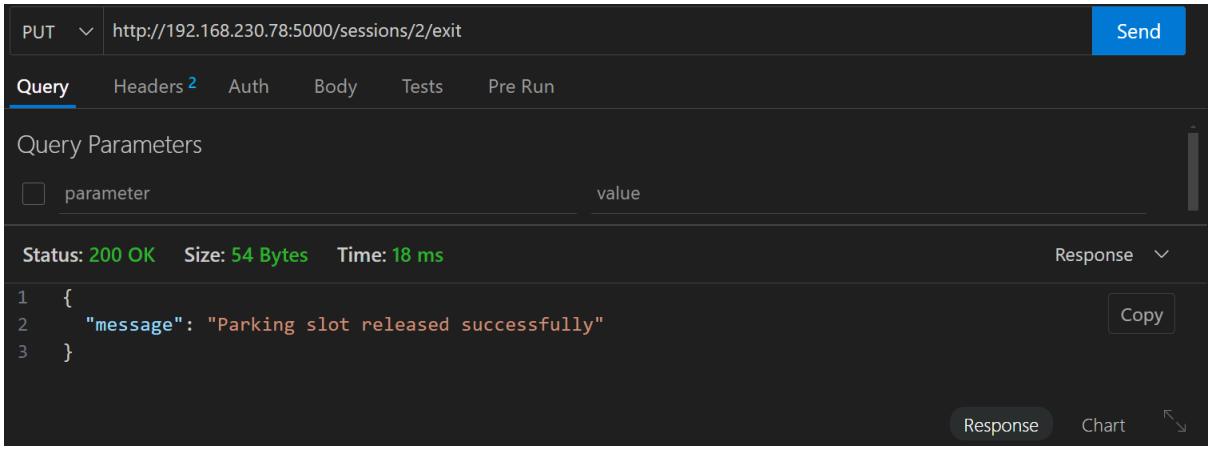

test_app.py
Dockerfile
app.py M
New Request
New Request
cicd.yaml
POST
http://192.168.230.78:5000/floors/
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
{
"parking_id": 2 ,
"floor_number": "Ground Floor",
"total_slots": 100,
"available_slots": 90
}
Status: 201 CREATED
Size: 64 Bytes
Time: 34 ms
Response
{
"floor_id": 6,
"message": "Floor created successfully"
}
Response
Chart


POST
http://192.168.230.78:5000//users/register
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
{
"username": "Geethika",
"password": "securepass123",
"email": "geethika@example.com",
"phone": "9876543210"
}
Status: 201 CREATED
Size: 65 Bytes
Time: 28 ms
Response
Copy
{
"message": "User registered successfully",
"user_id": 5
}
Response
Chart


POST
http://192.168.230.78:5000/users/login
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
{
"email": "geethika@example.com",
"password": "securepass123"
}
Status: 200 OK
Size: 53 Bytes
Time: 13 ms
Response
Copy
{
"message": "Login successful",
"user_id": 5
}


app.py M
New Request
New Request *
New Request
New Request
New Request
POST
http://192.168.230.78:5000/sessions/
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
{
"user_id": 5,
"parking_id": 257,
"slot_id": 5,
"car_number": "AP123456"
}
Status: 201 CREATED
Size: 72 Bytes
Time: 52 ms
Response
Copy
{
"message": "Parking slot booked successfully",
"session_id": 3
}
Response
Chart


POST
http://192.168.230.78:5000/sessions/
Send
Query
Headers 2
Auth
Body 1
Tests
Pre Run
JSON
XML
Text
Form
Form-encode
GraphQL
Binary
"user_id": 3,
"parking_id": 1,
"slot_id": 12,
"car_number": "TS09AB1234"
}
Status: 400 BAD REQUEST
Size: 39 Bytes
Time: 13 ms
Response
Copy
{
"message": "Slot is unavailable"
}
Response
Chart



Not secure | 192.168.230.78:5000/users/

Pretty-print


```
{
  "users": [
    {
      "created_at": "Mon, 28 Apr 2025 00:23:18 GMT",
      "email": "john@example.com",
      "phone": "1234567890",
      "user_id": 1,
      "username": "john_doe"
    },
    {
      "created_at": "Mon, 28 Apr 2025 00:23:18 GMT",
      "email": "jane@example.com",
      "phone": "0987654321",
      "user_id": 2,
      "username": "jane_smith"
    },
    {
      "created_at": "Mon, 28 Apr 2025 00:24:20 GMT",
      "email": "preethika@gmail.com",
      "phone": "8919654223",
      "user_id": 3,
      "username": "preethika"
    },
    {
      "created_at": "Sun, 27 Apr 2025 19:03:41 GMT",
      "email": "mark@example.com",
      "phone": "9876543987",
      "user_id": 4,
      "username": "mark"
    },
    {
      "created_at": "Thu, 01 May 2025 06:19:37 GMT",
      "email": "geethika@example.com",
      "phone": "9876543210",
      "user_id": 5,
      "username": "Geethika"
    }
  ]
}
```


Not secure | 192.168.230.78:5000/sessions/

Pretty-print


```
{
  "sessions": [
    {
      "car_number": "NY1234",
      "entry_time": "Sun, 27 Apr 2025 19:10:59 GMT",
      "exit_time": "Sun, 27 Apr 2025 19:14:55 GMT",
      "parking_id": 1,
      "session_id": 1,
      "slot_id": 1,
      "user_id": 3
    },
    {
      "car_number": "AP233",
      "entry_time": "Sun, 27 Apr 2025 19:12:55 GMT",
      "exit_time": "Thu, 01 May 2025 07:09:48 GMT",
      "parking_id": 1,
      "session_id": 2,
      "slot_id": 2,
      "user_id": 1
    },
    {
      "car_number": "AP123456",
      "entry_time": "Thu, 01 May 2025 06:25:31 GMT",
      "exit_time": null,
      "parking_id": 257,
      "session_id": 3,
      "slot_id": 5,
      "user_id": 5
    }
  ]
}
```

About test_app.py

The test_app.py file contains a comprehensive suite of automated tests for the Parking System application. It is built using **pytest**, a widely-used Python testing framework. These tests ensure that each part of the application — such as user registration, login, and the creation of parking structures — works as expected.

Purpose of the Test File

The goal of test_app.py is to:

- Validate that the application behaves correctly under different scenarios.
- Detect issues early in development.
- Provide confidence during deployment that core features are functioning properly.

Test Environment Setup

The test file creates a temporary version of the Flask application, specifically configured for testing. It uses a lightweight, in-memory SQLite database so that no real data is affected. The database tables are freshly created at the start of testing to ensure a clean and isolated environment.

What is Being Tested

1. **Application Availability**
   It checks if the home (welcome) page of the application is accessible and returns the expected content.

2. **User Registration and Login**
   Tests are written to simulate the process of registering a new user and then logging in with valid credentials.

3. **Parking Lot Functionality**
   The tests verify that parking lots can be created and correctly listed by the API.

4. **Floor, Row, and Slot Management**
   Each of these features is tested to ensure that:
   - They can be created with valid data.
   - The application correctly retrieves them when requested.

Why These Tests Matter

- They simulate real-world API usage through a test client.
- They help catch errors quickly by running in a controlled and repeatable test environment.
- They improve application reliability and make future changes safer and easier to manage.

To run your test suite using pytest, use the following command in your terminal:

- pytest test_app.py

If you want more detailed output (like showing which tests passed or failed), use the -v (verbose) flag:

- pytest -v test_app.py

Make sure you're in the virtual environment and inside your project directory when you run these commands.

Dockerization of the Parking System Application

To make the application platform-independent and easy to deploy, we containerized it using Docker. Dockerfile defines the setup for the Docker image and the instructions needed to run the Flask application inside a container.

Explanation of the Dockerfile

1. **Base Image**
   The Docker image is built using a minimal Python base: **python:3.12-slim**
   This keeps the container lightweight while still providing everything needed to run Python applications.

2. **Working Directory**
   The container's working directory is set using: **WORKDIR /app**
   All subsequent operations like copying files or installing packages are done inside this /app folder.

3. **System Package Installation**
   To support **mysqlclient**, several system-level packages are installed:

   - gcc
   - default-libmysqlclient-dev
   - pkg-config
   - build-essential

   These are installed using apt-get commands, ensuring all necessary build tools and libraries are present.

4. **Copying Source Code**
   Your local project files are transferred into the container using: **COPY . /app**
   This makes the entire Flask application accessible inside the container.

5. **Installing Python Dependencies**
   All Python libraries listed in **requirements.txt** are installed with:
   pip install --no-cache-dir -r requirements.txt
   This sets up the Flask app and its dependencies without caching to reduce image size.

6. **Exposing Flask Port**
   The container declares that it will use port: **5000**
   This is the default port on which the Flask app runs, allowing external access.

7. **Environment Variable (Optional)**
   An environment variable is defined: **ENV NAME=ParkVision**
   This can be used by the application internally if needed.

8. **Running the Application**
   The container launches the application using: **CMD ["python", "run.py"]**
   This runs the Flask server defined in the run.py file.

Once the **Dockerfile** is created, follow these steps to build the Docker image, test it locally, and push it to Docker Hub

Step 1: Build the Docker Image

Use the Docker CLI to build an image from your Dockerfile.

docker build -t your_Docker_Hub_**usernam**/Image_name:latest .

docker build -t preethika1801/parkvision:latest .

- -t tags the image with your Docker Hub username and image name.
- . indicates the Dockerfile is in the current directory.

Step 2: Test the Docker Image Locally

You can run the image locally to test if your app works as expected.

docker run -p 5000:5000 preethika1801/parkvision:latest

- This maps your local machine's port 5000 to the container's port 5000.
- Open a browser and go to http://localhost:5000 to check the app.

Step 3: Log In to Docker Hub

Before pushing the image, log in to Docker Hub: docker login

- Enter your Docker Hub **username** and **password** when prompted.

Step 4: Push the Image to Docker Hub

Push your built image to Docker Hub:

docker push preethika1801/parkvision:latest

- This uploads the image to your Docker Hub repository, making it accessible from any system with Docker.

Pull and Run the Image Anywhere : Once your image is pushed to Docker Hub, it becomes public (if not private) and can be used on **any machine with Docker installed**.

Try It Online with Play with Docker : Play with Docker is a free online platform where you can test Docker containers without installing anything locally.

How to use:

1. Go to https://labs.play-with-docker.com/
2. Log in with your Docker Hub account.
3. Click **"Start"** to open a new terminal instance.
4. In the terminal:

docker pull preethika1801/parkvision:latest

docker run -p 5000:5000 preethika1801/parkvision:latest

5. Click on the "5000" port link in the top tab bar to access your running Flask app.

CI/CD Pipeline for Parking System

File Path: .github/workflows/cicd-pipeline.yaml

This GitHub Actions workflow automates the process of:

- Building the Docker image for the parking system
- Pushing the image to Docker Hub

Triggers: The workflow triggers on:

· A push to the main or master branch.
· A pull request targeting main or master.

Job : build_and_push:

· Checkout the code: The actions/checkout@v2 action pulls your code from the repository.
· Docker login: Uses the Docker login action to authenticate with Docker Hub using your stored secrets (DOCKER_USERNAME and DOCKER_PASSWORD).
· Build Docker image: The docker build command builds your Docker image based on the Dockerfile in your repository, tagging it as preethika1801/parking-system:latest.
· Push Docker image: After the build, the image is pushed to Docker Hub under the preethika1801/parking-system repository with the latest tag.

GitHub Secrets Setup for Docker Authentication

To securely authenticate with Docker Hub, you need to store your Docker Hub credentials as **GitHub Secrets**. This allows GitHub Actions to access them during the CI/CD workflow.

Steps to Add GitHub Secrets:

1. Open Your GitHub Repository: Navigate to your repository on GitHub.
2. Go to Settings: Click on the Settings tab at the top of the repository page.
3. Access Secrets: Scroll down the left sidebar, and under the Security section, click on Secrets and variables. Choose Actions to manage repository secrets.
4. Add New Secrets: Click on New repository secret.
5. Enter the Secret Information: Add the following secrets

· DOCKER_USERNAME — Your Docker Hub username (e.g., preethika1801).
· DOCKER_PASSWORD — Your Docker Hub password or access token for authentication.

6. Save the Secrets: After entering the secret names and values, click Add secret to save.

Why Set These Secrets?

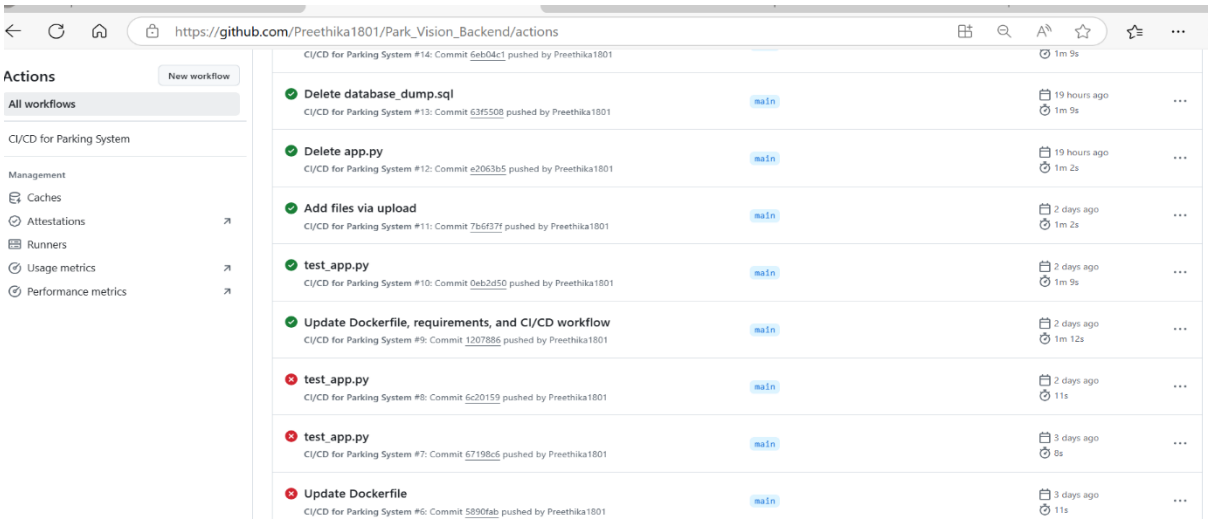
- DOCKER_USERNAME: The username associated with your Docker Hub account. This is used to log in during the CI/CD pipeline.
- DOCKER_PASSWORD: This can either be your Docker Hub password or, preferably, a Docker Hub access token (recommended for security).

Once you've added these secrets, the CI/CD pipeline will be able to authenticate with Docker Hub and securely build and push your Docker image whenever there is a push or PR to main or master.

Check CI/CD Pipeline Status

1. Commit & Push Changes to GitHub (to main or master).
2. Go to GitHub Repository and click on the Actions tab.
3. View Workflow Runs: Check if the build is successful (green check) or failed (red cross).
4. Click on a Run for detailed logs if there's an error.

This helps monitor your build and deployment process.



Track CI/CD Pipeline Progress

1. Click on the Orange Processing Button (or spinning icon) to view the current status of the pipeline.
2. See Step-by-Step Progress: The button will show which step (like build, push, etc.) the pipeline is currently processing.
3. Real-Time Feedback: You can monitor the ongoing actions and know if it's in the build phase, login, or any other part of the workflow.

This helps you track the exact point in the pipeline and monitor real-time progress.