**Restful APIs with Flask**

**Returning the JSON instead of text:**

from flask import Flask, jsonify

import flask\_cors

app = Flask(\_\_name\_\_)

@app.route('/')

def greet():

return 'hello world'

@app.route('/sample\_page')

def sample\_page():

# return 'Back to home'

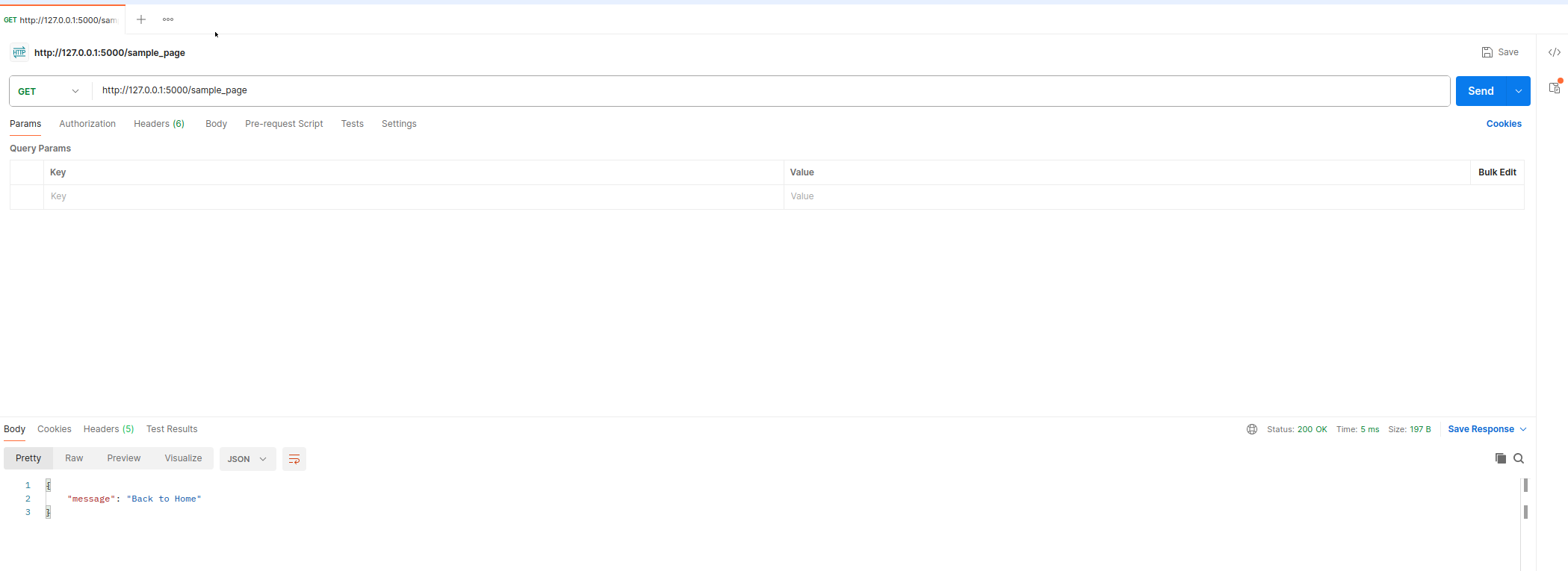
return jsonify(message='Back to Home')

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug = True)

In the above code there is a function called sample\_page() in which will return the json format of the string instead of normal text. JSONIFY does this modification.

The above code will be running in the localhost i.e. <http://127.0.0.1:5000> and this can be tested by using POSTMAN. Go to postman and set the new file as GET method and pass the above URL so that it gives the response.



**HTTPS status codes:**

All web applications are based on request-response mechanism. Browser requests the resource from the site and the site responses. Typically, it’s a HTML document with links, CSS videos, files, images etc. what else is needed to render the page.

Requests and responses both have headers i.e. headers are the metadata that describe the characteristics of request and response and one of the key features of headers is the **status code**. The status codes in HTTP will tell whether the request is successful or not. Status code 200 means the request is successful and okay.

Status codes are useful because when there is a UI that is calling one endpoint, the calling program needs to know whether there are any problems with the request. Status code 404 means “not found” the requested data.

from flask import Flask, jsonify

import flask\_cors

app = Flask(\_\_name\_\_)

@app.route('/')

def greet():

return 'hello world'

@app.route('/sample\_page')

def sample\_page():

# return 'Back to home'

# explicitly specifying the status code, but it is optional

return jsonify(message='Back to Home'), 200

@app.route('/not found')

def not\_found():

return jsonify(message = 'Page not found'), 404

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug = True)

**URL Parameters:**

GET request can do more rather than getting simple address. They can also pass the data to the server for processing the requests, allowing to present the dynamic content to the users based on the URL parameters passed in.

<http://www.google.com>/ – a URL with no parameters, to add parameters =>

<http://www.google.com/?s=pavan> => the parameters can be passed by adding a **“?”** and passing the key-value pairs. This allows the frontend directly to pass the input to the endpoint.

Here is a simple code snippet to give the parameters to the endpoint =>

from flask import Flask, jsonify, request

# pass the parameters to the endpoint from the frontend directly or test the api using postman

@app.route('/param')

def parameters():

name = request.args.get("name")

age = int(request.args.get('age'))

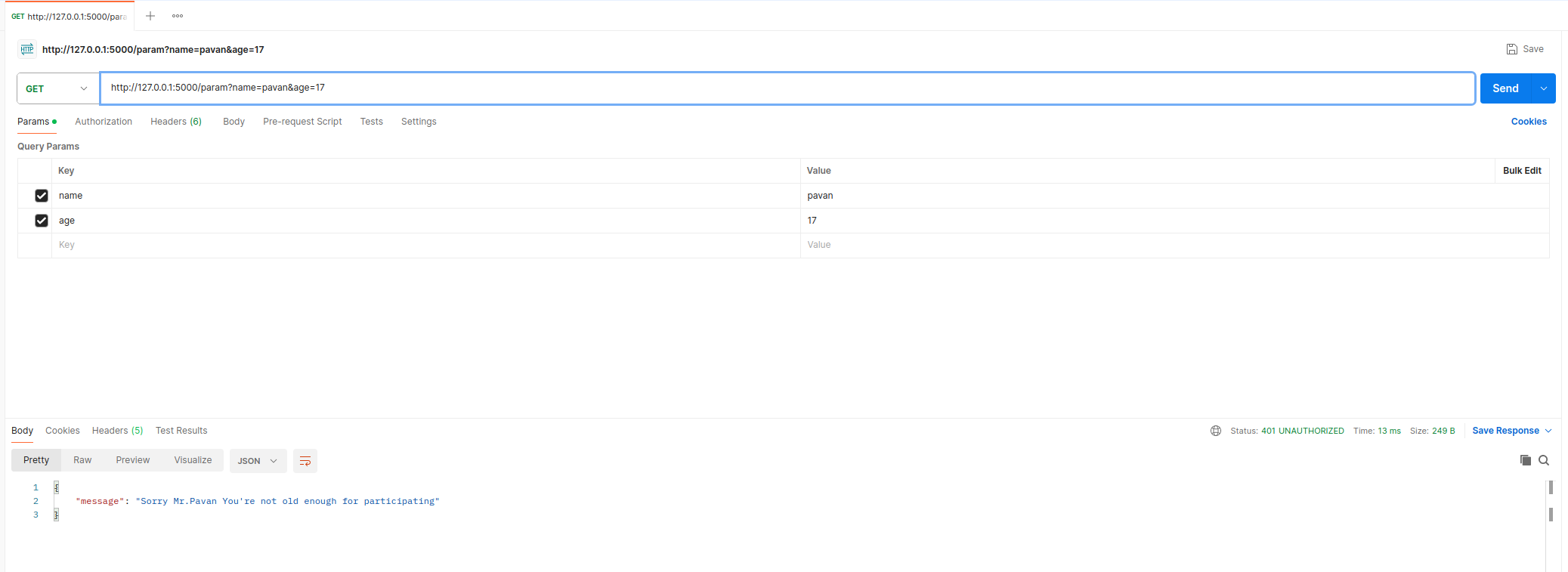
if age > 18:

return jsonify(message = "Mr." + name.capitalize() + ", You're eligible to vote."), 200

else:

return jsonify(message = "Sorry Mr." + name.capitalize() + " You're not old enough for participating"), 401

and now while checking in postman, after doing the GET request with the URL <http://127.0.0.1:5000/param> you need to pass the parameters in key-value pairs in the key and value column of postman then it will reflect as <http://127.0.0.1:5000/param?name=pavan&age=17> .



**URL variables and conversion filters:**

Normal URLs work by adding the key values pairs after a question mark within the URL structure. This causes a mess like unattracted URLs in the application. So instead of using the key value pairs, modern APIs use a cleaner URL structure like <http://127.0.0.1:5000/pavan/17>.

Flask supports forgoing the normal URL syntax instead it allows to create rules for matching the variable parts of the URL. This means there can be variables in certain parts of the URL instead of hard coded ones. The URL variables become arguments in the endpoint function.

# pass the URL parameters as arguments instead of using the request

@app.route('/url\_var/<string:name>/<int:age>')

def url\_variables(name: str, age: int):

if age > 18:

return jsonify(message = "Mr." + name.capitalize() + ", You're eligible to vote."), 200

else:

return jsonify(message = "Sorry Mr." + name.capitalize() + " You're not old enough for participating"), 401

**Adding an ORM: SQLAlchemy**

Most of the time APIs would probably creating, reading, updating, and deleting records of some kind. The most common ways to do this is to use the relational database or else can use a no SQL database like Mongo db or any other type of object storage. Here SQLite database is used.

SQLite is a file-based database management system as no server is required. Most of the db like oracle, Postgres, MySQL are server based databases i.e. need to install and manage a database server on the development machine.

By using SQLite we can bypass this and just use a simple database file to store our data. No software installation is required. Here we use ORM (Object Relational Mapper) called SQLAlchemy in this case to deal with our database.

Relational databases typically use a query language called SQL (Structured Query Language) to manipulate structure and data in the database. But when using ORM the SQL is generated for the developer behind the scenes.

**Benefits of ORM:**

* It works with python classes and objects, not with SQL.
* Using an ORM also allows to switch the database easily.
* It allows to control the database’s structure with code which can be managed by a VCS like Git etc.
* Supports multiple db platforms.

Install SQLAlchemy by using PyCharm or by doing pip install. By using PyCharm =>

PyCharm 🡪 preferences 🡪 project interpreter 🡪 click + at bottom 🡪 search for Flask-SQLAlchemy 🡪 click on install package.

**Setting up SQLAlchemy:**

We need some configurations like importing SQLAlchemy and the datatypes from it. Then as it is the file based database management system, need to specify where to store the file.

from flask import Flask, render\_template, request

from flask\_sqlalchemy import SQLAlchemy

from sqlalchemy import Column, String, Integer, Float

import os

app = Flask(\_\_name\_\_)

#setting up the base directory where the db should be stored.

# Here the application is kept in the same folder itself.

base\_directory = os.path.abspath(os.path.dirname(\_\_file\_\_))

# Creating a database

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///' + os.path.join(base\_directory, 'hospital.db')

# Initializing the database

db = SQLAlchemy(app)

The above will create db called hospital.db in the present working directory.

**Creating the ORM model classes:**

Here the hospital database will be having the **patients id,** and **covid result.** Create various classes for this by saying it’s a Model i.e. db.model. The models will be like

#table for patients

class Patient(db.Model):

\_\_tablename\_\_ = 'patients' # It sets the table name when it generates the table, here the table name is "patients"

patients\_id = Column(Integer, primary\_key = True, unique=True)

first\_name = Column(String)

last\_name = Column(String)

phone\_num = Column(Integer, unique = True)

# table for covid result

class CovidResult(db.Model):

\_\_tablename\_\_ = 'covid results'

id = Column(Integer,primary\_key=True)

test\_result = Column(bool)

Now the models are created i.e. the database tables are created.

**Seeding database with Flask CLI:**

The database structure is present in the code, but we need a way to manage the db file. The script the creates the database, drops the database and to seed the database to test with some test data. The code looks like

# Flask CLI commands for using the database like creating the db, deleting it and testing the db using some test data

@app.cli.command('create\_db') # Here the command given inside paranthesis is used in terminal to follow instructions

def db\_create():

with app.app\_context():

db.create\_all()

print('Database creation is done!!!')

@app.cli.command('delete\_db')

def db\_drop():

with app.app\_context():

db.drop\_all()

print('Database is Deleted :(')

@app.cli.command('test\_db')

def db\_seed():

with app.app\_context():

result\_1 = CovidResult(id = 789, test\_result = True)

result\_2 = CovidResult(id = 987, test\_result = False)

result\_3 = CovidResult(id = 321, test\_result = False)

# Need to add the records to the database

db.session.add(result\_1)

db.session.add(result\_2)

db.session.add(result\_3)

test\_user\_1 = Patient(patients\_id = 321, first\_name = 'Ravindra', last\_name = 'Chahal', phone\_num = 6549873210)

test\_user\_2 = Patient(patients\_id = 789, first\_name = 'Rohit', last\_name = 'Rayudu', phone\_num = 1234567890)

test\_user\_3 = Patient(patients\_id = 987, first\_name = 'Virat', last\_name = 'Singh', phone\_num = 9876543210)

db.session.add(test\_user\_1)

db.session.add(test\_user\_2)

db.session.add(test\_user\_3)

# After adding the records, need to save the changes by commiting it, otherwise db will be running but nothing will be seen inside database

db.session.commit()

print('Database Seeded')

The above code is having creation of the database, deleting it, and seeding the database with some test data.

Remember that **adding** and **committing** is must after writing the test data.

Now go to terminal and as per the above code to create database use **flask create\_db**, this will create the database, after that run **flask test\_db** so that the database is seeded.

Install DB browser for SQLite to view the database that is created.

**Retrieving the list of patients:**

The most common pattern found in web development is pulling the list of things from a database, displaying them in a table or grid and then providing the ability to drill down to view details, add new records, update existing records, delete the records if not needed etc. this is the most common thing the APIs do.

Here we will be doing the above operations starting with an endpoint can be used to display list of patients that are in the hospital database.

In SQLAlchemy 🡪

* To query all elements in a table 🡪 **<table\_name>.query.all()**
* # Retriving the list of patients from the database
* # so to this rule "methods" are added so that whatever specified inside methods like GET or POST etc. only they are accepted other methods will throw error
* @app.route('/patients', methods = ['GET'])
* def patients():
* list\_of\_patients = Patient.query.all()
* return jsonify(data = list\_of\_patients)

but the above code will return the 500 internal server error when tested using postman.

**NOTE: Errors in flask applications look little different than the errors in other languages. Flask is built on top of another library called “Werkzeug”. It is a python web services gateway or WSGI in short. Werkzeug is responsible for actually serving up our data in response to the endpoint definitions that are made in flask. Flask is usually used to develop traditional template-based HTML sites.**

The above code gives the error when tested with postman saying that “**TypeError: Object of type Patients is not JSON serializable”**. What it actually means is that, Flask.JSON can’t convert the data that is passed into the function into the actual JSON. The actual JSON serializers actually work with python dictionaries but in the above code is not a python dictionary. This can be solved in different ways, but the easiest way is to use the existing library called **Marshmallow**.

**Serializing SQLAlchemy results with Marshmallow:**

The process of converting an object into textual representation of that object is called **Serialization** and the process of converting text back into an object is called **Deserialization.**

**Marshmallow** is a 3rd party serialization library for python.

# Creating Schemas using marshmallow instances

class PatientSchema(ma.Schema):

class Meta:

fields = ('patients\_id', 'first\_name', 'last\_name', 'phone\_num')

class ResultSchema(ma.Schema):

class Meta:

fields = ('id', 'test\_result')

patient\_schema = PatientSchema()

patients\_schema = PatientSchema(many=True) # Here two schemas are created because if there is only one entry it can deserialize using patient\_schema, if more entries it can use patients\_schema

result\_schema = ResultSchema()

results\_schema = ResultSchema(many=True)

This is the route that gives the json representation of the list of patients

# Retriving the list of patients from the database

# so to this rule "methods" are added so that whatever specified inside methods like GET or POST etc. only they are accepted other methods will throw error

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

def patients():

list\_of\_patients = Patient.query.all()

# return jsonify(data = list\_of\_patients) # returns a typeerror saying TypeError: Object of type Patient is not JSON serializable. To do this use "Marshmallow"

result = patients\_schema.dump(list\_of\_patients)

return jsonify(result)

**API Security:**

Most APIs involve user registrations and logins, you track the users and grant them privileges. So in the above Patients API need to allow registration. In order to register a new patient, delete or edit the patient details you must be logged in.

There are several login and user registration pre-built plugins like **Flask-Login**, which handles logging in, logging out, and session management. Similarly there is **Flask-User,** which handles the user registration, login, logout, and role based security. But need to learn about how these plugins work with the different ORMs.

For API projects, its better to use **JWT (JSON Web Tokens)**. Most of the times the app talking to the API is going to be a web app written in React or Angular. It could also be a mobile or a desktop app.

JSON is a universal format to represent the data and it’s supported in most of the languages., making it easy to mix technologies on frontend versus backend. **JWT is an open standard for authenticating and verifying the data exchanges between two parties**.

The Flask-JWT library works by adding the decorators to the routes that are to be protected like user registration or login etc.

**Registering new users:**

Here in the new patient registration, assume that in the frontend HTML forms are used. So use the requests method to get the form using **request.form[‘<element>’]** and if the phone number exists already the registration will be denied or else remaining fields like first\_name, last\_name, patients\_id are entered.

@app.route('/registration', methods=['POST'])  
def registration():  
 # Assume that that calling routes using the HTML forms  
 # Registration is in the form format, now we need to give the fields to register the user  
 phone\_num = request.form['phone\_num'] # grabbing the phone number from the submitted form  
  
 # See whether the user is already registered or not  
 test = Patient.query.filter\_by(  
 phone\_num=phone\_num).first() # this is going to call the database and will be looking for the single user with the phone number that is just passed in  
 if test:  
 return jsonify(message='The patient with phone number already exist')  
 else:  
 fname = request.form['first\_name']  
 lname = request.form['last\_name']  
 pid = request.form['patients\_id']  
 patient = Patient(patients\_id=pid, first\_name=fname, last\_name=lname, phone\_num=phone\_num)  
 db.session.add(patient)  
 db.session.commit()  
 return jsonify(message='Patient Registration successful'), 201 # the status code in 200s means it is successful

**Authenticating users and passing the token:**

Now there is ability to register the new users and need a way for the existing users to login. A route needs to be added for this.

Before this setting up the JWT and its imports 🡪

from flask\_jwt\_extended import jwt\_required, create\_access\_token, JWTManager

app.config['JWT\_SECRET\_KEY'] = '\*\*\*\*\*\*' # Change this to UUID

# Initializing the web tokens  
jwt = JWTManager(app)

After initializing the JWT, add the route for logging the users

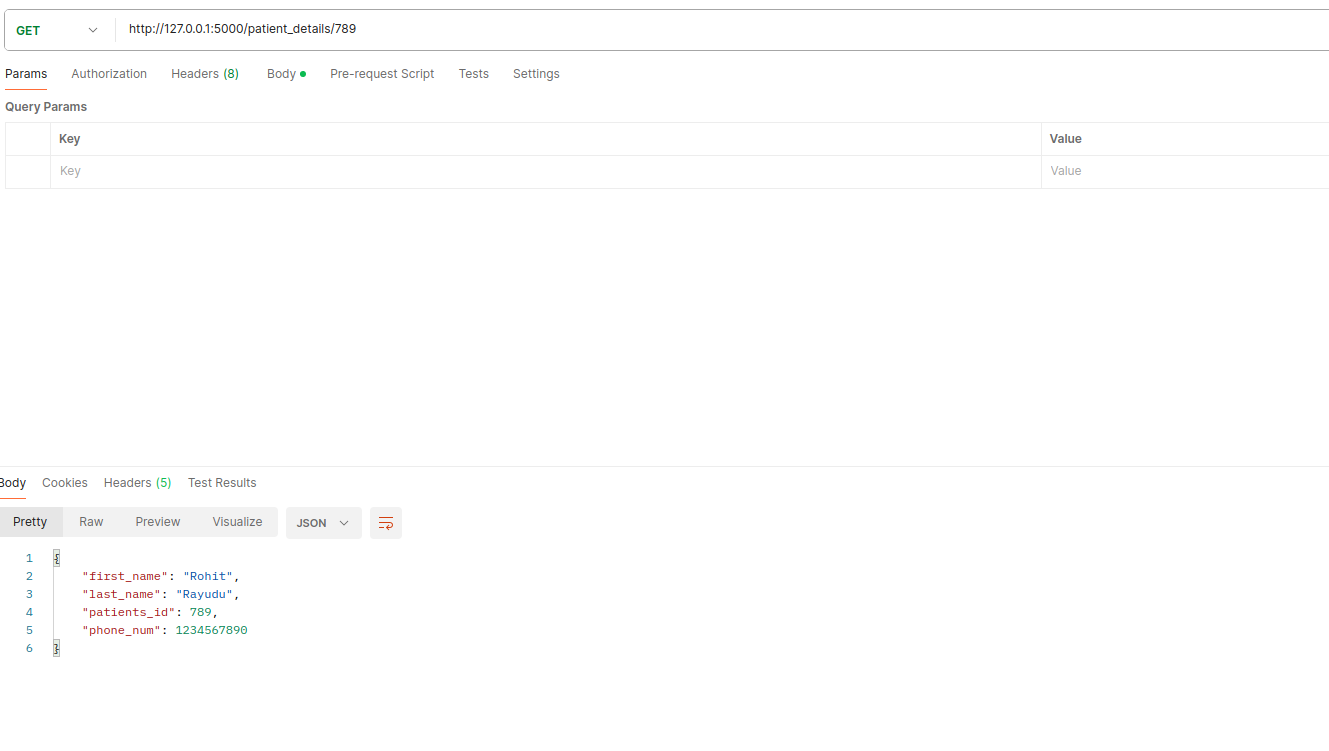
@app.route('/login', methods=[  
 'POST']) # The method is set to POST but this is controversial because POST is usually associated with creating the new records, but that is not done in login as we are dealing with existing users.  
def login():  
 if request.is\_json: # If the request is a Json then grab the phone number and patients name  
 phone\_no = request.json['phone\_num'] # used when it is in Json format  
 else:  
 phone\_no = request.form['phone\_num'] # used when the request is in a html form format  
  
 test patient = Patient.query.filter\_by(  
 phone\_num=phone\_no).first() # checking whether there is a match by phone number  
 if test\_patient: # if the patient is logged in then give the web token  
 access\_token = create\_access\_token(  
 identity=phone\_no) # identity means, how the user/patient is being identified. Here they are identified using phone number  
 return jsonify(message='Login Succeeded', access\_token=access\_token)  
 else:  
 return jsonify(message='Login Failed'), 401 # 401 status code means permission denied

**CRUD Operations:**

Retrieve the patient details using patients id 🡪

@app.route('/patient\_details/<int:patient\_id>', methods=['GET'])  
def patient\_details(patient\_id):  
 patient = Patient.query.filter\_by(patients\_id=patient\_id).first()  
 if patient:  
 result = patient\_schema.dump(patient)  
 return jsonify(result)  
 else:  
 return jsonify(message='Patient not found'), 404

After testing with postman, able to **GET** the details of patient if exists



Adding patients with **POST** method 🡪

@app.route('/add\_patient', methods=['POST'])  
def add\_patient():  
 phone\_num = request.form['phone\_num'] # grabbing the phone number from the submitted form  
  
 # See whether the user is already registered or not  
 test = Patient.query.filter\_by(  
 phone\_num=phone\_num).first() # this is going to call the database and will be looking for the single user with the phone number that is just passed in  
 if test:  
 return jsonify(message='The patient with phone number already exist')  
 else:  
 fname = request.form['first\_name']  
 lname = request.form['last\_name']  
 pid = request.form['patients\_id']  
 patient = Patient(patients\_id=pid, first\_name=fname, last\_name=lname, phone\_num=phone\_num)  
 db.session.add(patient)  
 db.session.commit()  
 return jsonify(message='Patient Registration successful'), 201

It is same as the user registration, now the important part is to secure this route 🡪

Protecting the endpoint is very easy by adding a decorator called **“jwt\_required”**, so after adding this decorator to POST method, test the endpoint using postman again and you may see 401 unauthorized which means Missing authorization header.

So first you need to login in and then copy the access token that is generated while logging in and then go to the authorization tab in postman and add the token and again click the send button then u will be able add the new patient to the db.