Bhartiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

NAME	Utsav Avaiya, Ayush Bodade
UID	2021300005, 2021300015
BATCH	Comps A (A1)
PROJECT TITLE	DataWiz
EXPERIMENT NO.	4
DATE OF SUBMISSION	19th September, 2023

AIM

To create a RESTful web service to showcase the usage of HTTP methods (GET, POST, PUT, DELETE) for CRUD operations on a hypothetical resource, emphasizing their roles in creating, reading, updating, and deleting data in a RESTful API.

THEORY

What is API?

API, which stands for Application Programming Interface, serves as a vital framework of rules, protocols, and tools that facilitate communication between distinct software applications. APIs define the methods and data structures through which applications can request and exchange information, thus enabling seamless collaboration between them.

The primary purpose of APIs is to facilitate the integration of different software systems, allowing them to share both data and functionalities. These interfaces find widespread use in various domains such as web development, mobile app development, and various other software-related fields. APIs empower developers to access services, retrieve data from databases, interact with external resources, and accomplish a diverse range of tasks. Consequently, APIs play a fundamental role in the development of modern software applications that rely on external resources and services.

SOAP vs REST API

Aspect	SOAP API	REST API
Message Format	Uses XML for message format.	Uses various formats, primarily JSON.
Complexity	Generally considered more complex.	Simpler and more flexible.
Security	Offers various security mechanisms, including WS-Security.	Relies on HTTPS for security.
Performance	Tends to have more overhead due to XML and stricter standards.	Typically, faster, and more efficient.
State Management	Supports stateful communication.	Stateless; each request contains all needed information.
Error Handling	Provides built-in error handling and standardized fault messages.	Error handling often uses HTTP status codes and custom responses.
Usage	Commonly used in enterprise and strict standards environments.	Popular for web and mobile applications, especially public APIs.

CRUD Operations in REST API:

CRUD stands for Create, Read, Update, and Delete. These operations are the basic actions that can be performed on resources in a RESTful system:

1. Create (POST): In a REST API, creating a resource corresponds to the HTTP POST method. Clients can send data to the server to create a new resource. For example, when creating a new user in a web application, you might send a POST request with the user's details to the server.

2. Read (GET): Reading a resource is represented by the HTTP GET method. Clients can retrieve resource data from the server by sending a GET request. For instance, when fetching information about a specific product, you'd use a GET request to the product's URL.

3. Update (PUT/PATCH): Updating a resource can be done using the HTTP PUT or PATCH methods. PUT typically updates the entire resource, while PATCH updates only specific fields of the resource. For example, to modify a user's email address, you'd send a PUT or PATCH

request with the updated data to the user's URL.

4. Delete (DELETE): Deleting a resource is achieved using the HTTP DELETE method. Clients can send a DELETE request to a resource's URL to remove it from the system. Deleting a user account, for instance, would involve sending a DELETE request to the user's URL.

Here is an example of how these CRUD operations map to RESTful endpoints:

Create: POST /users

Read: GET /users/{id}

Update: PUT /users/{id} or PATCH /users/{id}

Delete: DELETE /users/{id}

API vs Web Service

Aspect	API	Web Service
Communication	A broader term that encompasses various methods for software components to communicate with each other.	A specific type of API that is accessible over a network, often using standard protocols like HTTP or SOAP.
Purpose	Used for defining a set of rules, protocols, and tools for building software applications and enabling interaction between them.	A type of API designed for remote communication between software systems or components.
Accessibility	Can be either public (open to external developers) or private (restricted to internal use).	Typically designed for external access by other applications or services, but can also be private.
Protocols	Can use various communication protocols, including HTTP, REST, SOAP, gRPC, and more.	Often associated with protocols like HTTP, SOAP, and REST, depending on the type of web service.
Data Format	Supports various data formats, such as JSON, XML, HTML, and plain text, depending on the API design.	Often uses standardized data formats like XML or JSON, depending on the web service type.

Implementation	Can be implemented in various programming languages and technologies.	Typically implemented using technologies and frameworks specific to web services, such as JAX-RS, ASP.NET Web API, or Flask.
Interaction	Can include any type of interaction between software components, from basic function calls to data retrieval.	Primarily designed for remote procedure calls (RPC) or exchanging structured data between systems.
Use Cases	Used for a wide range of purposes, including libraries, frameworks, and various software integrations.	Commonly used for exposing specific functionalities or data to external applications, often in a standardized manner.

POSTMAN:

Postman is a popular and powerful software tool used in the field of application development and API testing. It is primarily designed for:

1. **API Development and Testing**: Postman is used by developers and QA (Quality Assurance) professionals to design, develop, and test APIs (Application Programming Interfaces). APIs are a set of rules and protocols that allow different software applications to communicate with each other.

2. **HTTP Request Testing**: Postman simplifies the process of sending HTTP requests (GET, POST, PUT, DELETE, etc.) to a web server or API endpoint. Users can easily construct requests, set headers, pass parameters, and inspect responses.

3. **Automation**: It provides features for automating API testing and integration testing, allowing users to create collections of requests and run them in sequences or as part of a test suite. This is valuable for regression testing and continuous integration workflows.

4. **Documentation**: Postman can be used to generate API documentation that is easy to read and understand. This documentation can be shared with developers who need to integrate with the API.

5. **Mock Servers**: Postman allows users to create mock servers, which are simulated versions of APIs. Mock servers are useful for testing an API before it's fully developed or to simulate various responses for testing different scenarios.

6. **Collaboration**: Teams can collaborate on API development and testing by sharing collections of requests and test scripts. This helps in maintaining consistency and ensuring that everyone is using the same API definitions.

7. **Monitoring and Debugging**: Postman provides debugging tools to examine request and

response details, including headers, status codes, and JSON data. It helps identify and resolve issues during API development and testing.

8. **Environment Variables**: Postman allows users to set up and manage environment variables, making it easy to switch between different environments (e.g., development, testing, production) without changing request configurations manually.

9. **Security Testing**: Postman can be used for security testing of APIs by sending various types of requests and payloads to identify vulnerabilities like SQL injection or cross-site scripting (XSS).

Overall, Postman simplifies the process of interacting with APIs, making it a valuable tool for developers, testers, and anyone involved in building and maintaining software systems that rely on web services and APIs. It provides a user-friendly interface, automation capabilities, and collaboration features that streamline the API development and testing process.

STEPS TO MAKE A POSTMAN REQUEST FOR THE BELOW SAMPLE CODE:

To make the above request in Postman and test the /ask endpoint, you can follow these steps:

1. Open Postman: Launch the Postman application if it's not already open.
2. Create a New Request:
   2.1. Click on the "New" button in the upper-left corner to create a new request.
3. Name Your Request:
   3.1. Give your request a name, such as "Test /ask Endpoint."
4. Select the Request Type:
   4.1. In the request panel, choose the request type as "POST" from the dropdown list next to the URL field.
5. Set the Request URL:
   5.1. Enter the URL where your Flask application is running. For a local Flask app, it might be something like http://localhost:5000/ask.
6. Set the Request Body:
   6.1. Go to the "Body" tab in the request panel.
   6.2. Select the "raw" option.
   6.3. In the dropdown next to "raw," select "JSON (application/json)."
   6.4. Enter your test data (JSON payload) in the request body. In this case, it should look like this:

```
{
  "prompt": "what is the message for investors/stakeholders"
}
```

7. Send the Request:
   7.1. Click the "Send" button to send the POST request to your Flask /ask endpoint.
8. View the Response:
   8.1. Postman will display the response in the response panel below the request panel.
9. Check the Response:
   9.1. Verify that the response status code is 200 (OK).

9.2. Examine the response body to see if it matches your expectations based on the test case you defined in your Flask application.

10. Repeat and Test Different Cases:

10.1. You can repeat these steps to test different cases by modifying the request body or URL as needed.

SAMPLE CODE

The API has been written in Python, Flask using the Langchain Framework as the LLM base.

// POST REQUEST

app1.py

```
from flask import Flask, request, jsonify
from werkzeug.utils import secure_filename
import os
from langchain.llms import OpenAI
from langchain.embeddings import OpenAIEmbeddings
from langchain.document_loaders import PyPDFLoader
from langchain.vectorstores import Chroma
from langchain.agents.agent_toolkits import (
    create_vectorstore_agent,
    VectorStoreToolkit,
    VectorStoreInfo
)

app = Flask(__name__)

# Set API key for OpenAI Service
os.environ['OPENAI_API_KEY'] = 'your_openai_api_key'

# Create an instance of OpenAI LLM
llm = OpenAI(temperature=0.1, verbose=True)
embeddings = OpenAIEmbeddings()

# Define the directory where uploaded files will be stored
UPLOAD_FOLDER = 'uploads'
ALLOWED_EXTENSIONS = {'pdf'}

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/upload', methods=['POST'])
def upload_file():
    if 'file' not in request.files:
```

```
        return jsonify({'error': 'No file part'})

    file = request.files['file']

    if file.filename == '':
        return jsonify({'error': 'No selected file'})

    if not allowed_file(file.filename):
        return jsonify({'error': 'Invalid file extension'})

    filename = secure_filename(file.filename)
    file_path = os.path.join(UPLOAD_FOLDER, filename)
    file.save(file_path)

    return jsonify({'message': 'File uploaded successfully', 'file_path': file_path})

@app.route('/ask', methods=['POST'])
def ask_question():
    file_path = request.form.get('file_path')

    loader = PyPDFLoader(file_path)
    pages = loader.load_and_split()
    store = Chroma.from_documents(pages, embeddings, collection_name='uploaded_report')

    vectorstore_info = VectorStoreInfo(
        name="uploaded_report",
        description="User-uploaded annual report as a PDF",
        vectorstore=store
    )

    toolkit = VectorStoreToolkit(vectorstore_info=vectorstore_info)
    agent_executor = create_vectorstore_agent(
        llm=llm,
        toolkit=toolkit,
        verbose=True
    )

    prompt = request.form.get('prompt')
    response = agent_executor.run(prompt)

    return jsonify({'response': response})

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

Test Case:

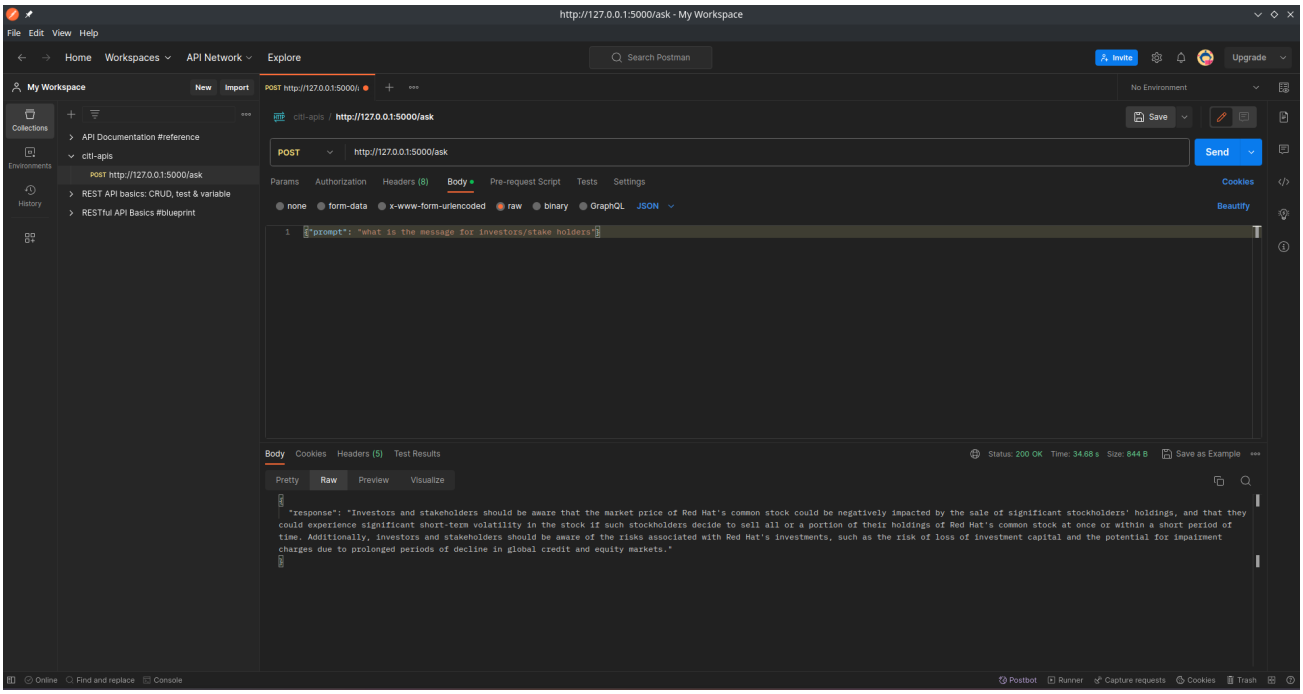
//POST TEST_CASE

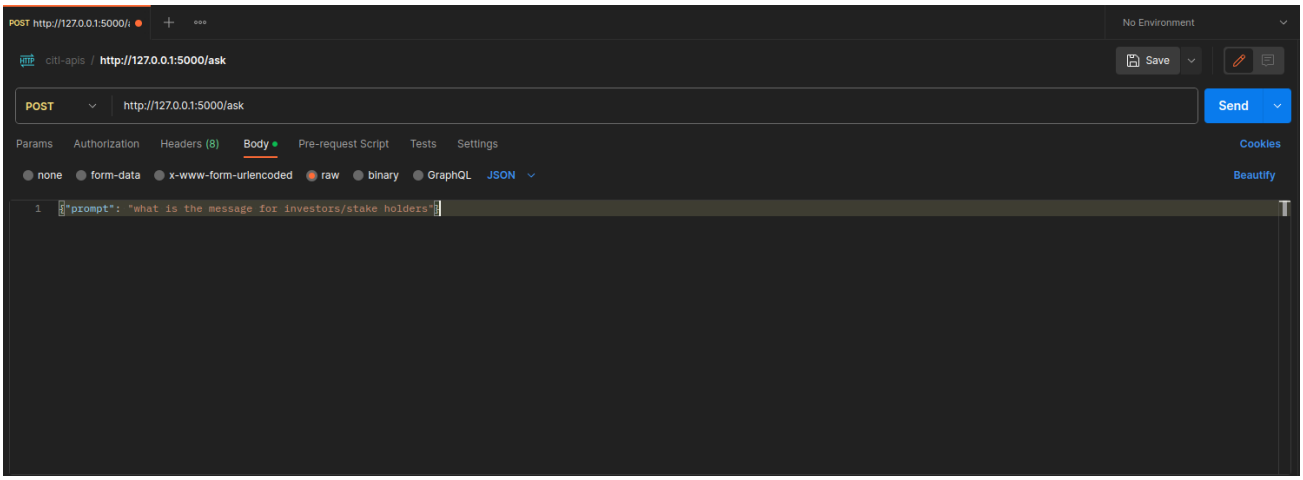
test_app1.py

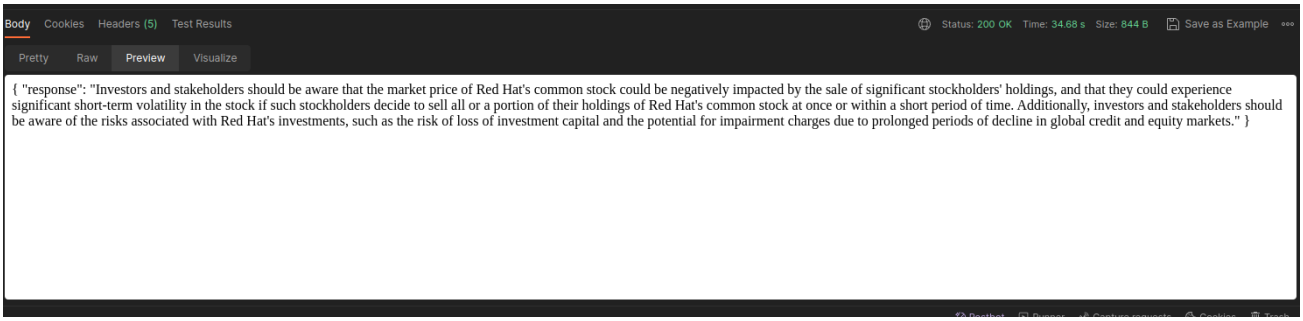
```
import requests

# Set the API URL
base_url = 'http://localhost:5000'  # Replace with your server URL if needed

# Upload a file
files = {'file': open('./sample.pdf', 'rb')}
upload_response = requests.post(f'{base_url}/upload', files=files)

if upload_response.status_code == 200:
    file_path = upload_response.json().get('file_path')

    # Ask a question
    data = {
        'file_path': file_path,
        'prompt': 'What are the profits in the latest year'
    }
    ask_response = requests.post(f'{base_url}/ask', data=data)

    if ask_response.status_code == 200:
        response_data = ask_response.json()
        print('Response:', response_data['response'])
    else:
        print('Error:', ask_response.text)
else:
    print('Error:', upload_response.text)
```

Sample.pdf : https://github.com/ayushbodade/citl-project-api/blob/main/tests/sample.pdf

OUTPUT:


http://127.0.0.1:5000/ask - My Workspace
File Edit View Help
Home Workspaces API Network Explore
Search Postman
Invite Upgrade
My Workspace New Import
Collections Environments History
API Documentation #reference
citi-apis
POST http://127.0.0.1:5000/ask
REST API basics: CRUD, test & variable
RESTful API Basics #blueprint
POST http://127.0.0.1:5000/
No Environment
citi-apis / http://127.0.0.1:5000/ask
Save
POST
http://127.0.0.1:5000/ask
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
1 {"prompt": "what is the message for investors/stake holders"}
Body Cookies Headers (5) Test Results
Status: 200 OK Time: 34.68 s Size: 844 B Save as Example
Pretty Raw Preview Visualize
{
"response": "Investors and stakeholders should be aware that the market price of Red Hat's common stock could be negatively impacted by the sale of significant stockholders' holdings, and that they could experience significant short-term volatility in the stock if such stockholders decide to sell all or a portion of their holdings of Red Hat's common stock at once or within a short period of time. Additionally, investors and stakeholders should be aware of the risks associated with Red Hat's investments, such as the risk of loss of investment capital and the potential for impairment charges due to prolonged periods of decline in global credit and equity markets."
}
Online Find and replace Console
Postbot Runner Capture requests Cookies Trash


POST http://127.0.0.1:5000/
No Environment
citi-apis / http://127.0.0.1:5000/ask
Save
POST
http://127.0.0.1:5000/ask
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
1 {"prompt": "what is the message for investors/stake holders"}


Body Cookies Headers (5) Test Results
Status: 200 OK Time: 34.68 s Size: 844 B Save as Example
Pretty Raw Preview Visualize
{ "response": "Investors and stakeholders should be aware that the market price of Red Hat's common stock could be negatively impacted by the sale of significant stockholders' holdings, and that they could experience significant short-term volatility in the stock if such stockholders decide to sell all or a portion of their holdings of Red Hat's common stock at once or within a short period of time. Additionally, investors and stakeholders should be aware of the risks associated with Red Hat's investments, such as the risk of loss of investment capital and the potential for impairment charges due to prolonged periods of decline in global credit and equity markets." }

CONCLUSION

The experiment successfully demonstrated the fundamental principles of RESTful web services, highlighting the crucial roles played by HTTP methods such as GET, POST, PATCH, and DELETE in performing CRUD (Create, Read, Update, Delete) operations and used the different features of the POSTMAN service to test the created CRUD APIs using the POSTMAN application.