Flask API’s

Developing APIs in Python can be done easily with popular web frameworks like **Flask** or **FastAPI**. Here’s a step-by-step guide using both:

**1. Using Flask**

Flask is a lightweight and widely-used framework for building APIs. It’s simple and easy to set up.

**Step 1: Install Flask**

Install Flask via pip:

bash

Copy code

pip install Flask

**Step 2: Create a Basic Flask API**

Create a file, say app.py:

from flask import Flask, jsonify, request

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

# Example data

items = [

{"id": 1, "name": "Item 1", "description": "This is item 1"},

{"id": 2, "name": "Item 2", "description": "This is item 2"}

]

# GET endpoint

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

def get\_items():

return jsonify(items)

# POST endpoint

@app.route('/api/items', methods=['POST'])

def add\_item():

new\_item = request.json

items.append(new\_item)

return jsonify(new\_item), 201

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Step 3: Run the Flask API**

bash

Copy code

python app.py

This starts a server at http://127.0.0.1:5000, and you can test the API endpoints:

* **GET** /api/items: Fetches all items.
* **POST** /api/items: Adds a new item to the list.

**2. Using FastAPI**

FastAPI is a high-performance framework for building APIs with support for automatic validation, asynchronous programming, and automatic generation of documentation.

**Step 1: Install FastAPI and Uvicorn (ASGI server)**

bash

Copy code

pip install fastapi uvicorn

**Step 2: Create a Basic FastAPI API**

Create a file, say main.py:

python

Copy code

from fastapi import FastAPI, HTTPException

from pydantic import BaseModel

from typing import List

app = FastAPI()

# Define a data model

class Item(BaseModel):

id: int

name: str

description: str = None

# Example data

items = [

Item(id=1, name="Item 1", description="This is item 1"),

Item(id=2, name="Item 2", description="This is item 2")

]

# GET endpoint

@app.get("/api/items", response\_model=List[Item])

def get\_items():

return items

# POST endpoint

@app.post("/api/items", response\_model=Item, status\_code=201)

def add\_item(item: Item):

items.append(item)

return item

# GET endpoint with path parameter

@app.get("/api/items/{item\_id}", response\_model=Item)

def get\_item(item\_id: int):

for item in items:

if item.id == item\_id:

return item

raise HTTPException(status\_code=404, detail="Item not found")

**Step 3: Run the FastAPI API**

Use uvicorn to run the server:

bash

Copy code

uvicorn main:app --reload

This starts a server at http://127.0.0.1:8000, and you can test the API endpoints:

* **GET** /api/items: Fetches all items.
* **POST** /api/items: Adds a new item.
* **GET** /api/items/{item\_id}: Fetches an item by ID.

FastAPI also provides auto-generated interactive API documentation:

* **Swagger UI**: http://127.0.0.1:8000/docs
* **ReDoc**: http://127.0.0.1:8000/redoc

**3. Testing Your API**

Once the API is up and running, you can use tools like **Postman**, **curl**, or **HTTPie** to send requests and test the endpoints.

**4. Advanced Features**

Both Flask and FastAPI support advanced features like authentication, middlewares, async requests, and database integration (e.g., with SQLAlchemy or Tortoise ORM).

Both frameworks make it straightforward to build APIs in Python, with **FastAPI** being the go-to choice for more performant and production-ready APIs due to its async capabilities and automatic data validation.

**Client**

**How many ways we can call the python api give some code example GET PUT POST DELETE**

In Python, there are multiple ways to call an API, particularly for the HTTP methods GET, POST, PUT, and DELETE. The most popular libraries for this task are **requests** and **http.client**. Below are examples for each method using these libraries.

**Method 1: Using the requests Library**

The requests library is the most commonly used library for making HTTP requests in Python because it is easy to use and provides a simple interface.

**Install requests (if not already installed)**

bash

Copy code

pip install requests

**1. GET Request**

python

Copy code

import requests

response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

print(response.status\_code) # Status code

print(response.json()) # JSON response content

**2. POST Request**

python

Copy code

import requests

data = {"title": "New Post", "body": "This is the content", "userId": 1}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=data)

print(response.status\_code)

print(response.json())

**3. PUT Request**

python

Copy code

import requests

data = {"title": "Updated Post", "body": "Updated content", "userId": 1}

response = requests.put('https://jsonplaceholder.typicode.com/posts/1', json=data)

print(response.status\_code)

print(response.json())

**4. DELETE Request**

python

Copy code

import requests

response = requests.delete('https://jsonplaceholder.typicode.com/posts/1')

print(response.status\_code)

print(response.json()) # Some APIs might return an empty response for DELETE

**Method 2: Using http.client (Standard Library)**

The http.client library is part of Python’s standard library and doesn’t require any additional installation. It provides a lower-level interface compared to requests.

**1. GET Request**

python

Copy code

import http.client

import json

conn = http.client.HTTPSConnection("jsonplaceholder.typicode.com")

conn.request("GET", "/posts/1")

response = conn.getresponse()

data = response.read()

print(response.status)

print(json.loads(data))

**2. POST Request**

python

Copy code

import http.client

import json

conn = http.client.HTTPSConnection("jsonplaceholder.typicode.com")

payload = json.dumps({"title": "New Post", "body": "This is the content", "userId": 1})

headers = {'Content-Type': 'application/json'}

conn.request("POST", "/posts", payload, headers)

response = conn.getresponse()

data = response.read()

print(response.status)

print(json.loads(data))

**3. PUT Request**

python

Copy code

import http.client

import json

conn = http.client.HTTPSConnection("jsonplaceholder.typicode.com")

payload = json.dumps({"title": "Updated Post", "body": "Updated content", "userId": 1})

headers = {'Content-Type': 'application/json'}

conn.request("PUT", "/posts/1", payload, headers)

response = conn.getresponse()

data = response.read()

print(response.status)

print(json.loads(data))

**4. DELETE Request**

python

Copy code

import http.client

conn = http.client.HTTPSConnection("jsonplaceholder.typicode.com")

conn.request("DELETE", "/posts/1")

response = conn.getresponse()

data = response.read()

print(response.status)

print(data) # Some APIs might return an empty response for DELETE

**Method 3: Using urllib.request (Standard Library)**

The urllib library is also part of Python’s standard library, although it’s less convenient for handling JSON payloads. It’s typically used for simpler HTTP requests.

**1. GET Request**

python

Copy code

import urllib.request

import json

with urllib.request.urlopen("https://jsonplaceholder.typicode.com/posts/1") as response:

data = json.loads(response.read().decode())

print(data)

**2. POST Request**

python

Copy code

import urllib.request

import json

data = json.dumps({"title": "New Post", "body": "This is the content", "userId": 1}).encode('utf-8')

req = urllib.request.Request("https://jsonplaceholder.typicode.com/posts", data=data, method="POST")

req.add\_header("Content-Type", "application/json")

with urllib.request.urlopen(req) as response:

print(json.loads(response.read().decode()))

**3. PUT Request**

python

Copy code

import urllib.request

import json

data = json.dumps({"title": "Updated Post", "body": "Updated content", "userId": 1}).encode('utf-8')

req = urllib.request.Request("https://jsonplaceholder.typicode.com/posts/1", data=data, method="PUT")

req.add\_header("Content-Type", "application/json")

with urllib.request.urlopen(req) as response:

print(json.loads(response.read().decode()))

**4. DELETE Request**

python

Copy code

import urllib.request

req = urllib.request.Request("https://jsonplaceholder.typicode.com/posts/1", method="DELETE")

with urllib.request.urlopen(req) as response:

print(response.status) # Some APIs might return a 204 status for DELETE

Each method above shows how to make GET, POST, PUT, and DELETE requests in Python using different libraries. The requests library is recommended for most projects due to its simplicity and extensive support for various HTTP functionalities.

**which one is most used now a days**

The **requests library** is by far the most commonly used method for making HTTP requests in Python today. It’s popular because of its simplicity, readability, and ease of handling JSON data, which is standard for RESTful APIs.

Here's why requests is preferred:

1. **Ease of Use**: The syntax is straightforward, making it easy for developers to make requests with minimal code.
2. **Automatic JSON Handling**: requests makes it easy to work with JSON responses, providing .json() to directly parse JSON content.
3. **Extensive Documentation**: It's well-documented and widely supported within the Python community.
4. **Community Support**: Due to its popularity, there's an abundance of resources, examples, and community support available online.
5. **Error Handling**: requests provides intuitive error handling, with built-in methods to handle response statuses and exceptions.

Example with requests for clarity:

python

Copy code

import requests

# GET request

response = requests.get('https://jsonplaceholder.typicode.com/posts/1')

if response.status\_code == 200:

print(response.json())

# POST request

data = {"title": "New Post", "body": "This is the content", "userId": 1}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=data)

if response.status\_code == 201:

print(response.json())

**Alternative: httpx**

While requests is still the most popular, **httpx** is gaining traction, especially for asynchronous programming. httpx offers both synchronous and asynchronous API calls, which makes it a versatile choice for modern applications where concurrency and performance are key.

However, requests remains the go-to for most standard API needs.