## Feb 11, 2024 | [Elective Assesment Exam](https://www.google.com/calendar/event?eid=MmRhaGxsODlrNGhlODBtNWt1cDZpMDNpbDMgYXBzYXIuMTkxNzA0QG5jaXQuZWR1Lm5w)

## Feb 12, 2024 | Elective Board Exam

## APSARABK\_191704

* 1.Multiples of 3 or 5 [Project\_Euler]
* 2. Fibonacci Problem [Project\_Euler]
* 3. Python Virtual Environments, basic commands.
* 4. Modern Architecture for Web Applications. Why this works well?
* 5. Database - CRUD queries
* 6. REST Framework. GET/PUT/DELETE resource using REST.
* 7. API Document
* 8. HTTP Status Codes and categorization
* 9. HTTP Methods
* 10. Simple Authentication using Token.
* 11. POST vs GET
* 12. HTTP vs HTTPS
* 13. Database: JOIN Query for two tables with example.
* 14. URLs/Routes in flask with example.
* 15. POST Request in flask using HTML vs via API request
* 16. JSON
* 17. Database Connection example using flask and any db library for Mysql.

**Answers :**

**Qn1.If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3,5,6, and 9. The sum of these multiples is 23.Find the sum of all the multiples of 3 or 5 below 1000**

**SOLUTION:  
  
total\_sum = 0**

**# Loop through numbers from 1 to 999 (excluding 1000)**

**for i in range(1, 1000):**

**# Check if the number is a multiple of 3 or 5**

**if i % 3 == 0 or i % 5 == 0:**

**# Add the multiple to the total sum**

**total\_sum += i**

**# Print the result**

**print("The sum of multiples of 3 or 5 below 1000 is:", total\_sum)**

**Initialization: initializing total\_sum to 0. This variable will be used to store the sum of multiples.**

**Loop through numbers: The for loop iterates through numbers from 1 to 999 (excluding 1000). This is done using the range(1, 1000) statement.**

**Check for multiples: Inside the loop, the if statement checks whether the current number (i) is a multiple of 3 or 5 using the modulo (%) operator. If the condition is true, it means the number is a multiple, and it is included in the sum.**

**Update total\_sum: If the condition is true, the current number (i) is added to the total\_sum.**

**Print the result: After the loop completes, the code prints the final sum of multiples below 1000.**

**OR   
  
def sum\_of\_multiples(limit):**

**# Initialize the sum**

**total\_sum = 0**

**# Iterate through numbers below the limit**

**for num in range(limit):**

**# Check if the number is a multiple of 3 or 5**

**if num % 3 == 0 or num % 5 == 0:**

**# Add the multiples to the sum**

**total\_sum += num**

**return total\_sum**

**# Set the limit to 1000**

**limit = 1000**

**# Call the function and print the result**

**result = sum\_of\_multiples(limit)**

**print("The sum of all multiples of 3 or 5 below", limit, "is:", result)**

**Function Definition:**

**def sum\_of\_multiples(limit):: Defines a function named sum\_of\_multiples that takes a parameter limit.**

**Initialization:**

**total\_sum = 0: Initializes a variable total\_sum to store the sum of multiples.**

**Loop Through Numbers:**

**for num in range(limit):: Iterates through numbers from 0 to limit - 1. The variable num takes each value in this range during each iteration.**

**Check for Multiples:**

**if num % 3 == 0 or num % 5 == 0:: Checks if the current number (num) is a multiple of 3 or 5 by using the modulo operator (%). If the remainder is 0 for either condition, the number is a multiple.**

**Add to Sum:**

**total\_sum += num: If the current number is a multiple of 3 or 5, it is added to the total\_sum.**

**Function Return:**

**return total\_sum: After iterating through all numbers below the limit, the function returns the calculated sum of multiples.**

**Setting the Limit:**

**limit = 1000: Sets the limit to 1000 for this specific case.**

**Function Call and Print Result:**

**result = sum\_of\_multiples(limit): Calls the function with the specified limit and stores the result in the variable result.**

**print("The sum of all multiples of 3 or 5 below", limit, "is:", result): Prints the result along with a descriptive message.**

**Qn2.Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2,the first 10 numbers will be :  
   
 1,2,3,5,8,13,21,34,55,89,........**

**By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.**

**Solution:  
  
# Initialize the first two terms of the Fibonacci sequence**

**a, b = 1, 2**

**# Initialize the total sum of even-valued terms**

**total\_even\_sum = 0**

**# Continue the loop until the Fibonacci term exceeds 4 million**

**while b <= 4000000:**

**# Check if the current Fibonacci term (b) is even**

**if b % 2 == 0:**

**# If it's even, add it to the sum**

**total\_even\_sum += b**

**# Update the Fibonacci sequence by moving to the next two terms**

**a, b = b, a + b**

**# Print the result**

**print("The sum of even-valued terms in the Fibonacci sequence below 4 million is:", total\_even\_sum)**

**OR**

**def sum\_even\_fibonacci(limit):**

**# Initialize variables for the first two terms of the Fibonacci sequence**

**a, b = 1, 2**

**# Initialize the sum of even-valued terms**

**even\_sum = 0**

**while a <= limit:**

**# Check if the current term is even**

**if a % 2 == 0:**

**# Add the even term to the sum**

**even\_sum += a**

**# Update the Fibonacci sequence by moving to the next two terms**

**a, b = b, a + b**

**return even\_sum**

**# Set the limit to four million**

**limit = 4000000**

**# Call the function and print the result**

**result = sum\_even\_fibonacci(limit)**

**print("The sum of even-valued terms in the Fibonacci sequence below", limit, "is:", result)**

**Initialization: Initializing a and b to the first two terms of the Fibonacci sequence (a=1 and b=2), and sum\_even to 0.**

**While Loop: The while loop continues until the current Fibonacci term (b) exceeds 4 million.**

**Check for Even Terms: Inside the loop, the if statement checks whether the current Fibonacci term (b) is even using the modulo (%) operator. If it's even, the term is added to the sum\_even.**

**Update Fibonacci Sequence: The Fibonacci sequence is updated in each iteration by moving to the next two terms (a becomes the old b, and b becomes the sum of the old a and b).  
Print Result: After the loop completes, the code prints the final sum of even-valued terms in the Fibonacci sequence below 4 million.**

**Python Virtual Environment**

**A virtual environment is a self-contained directory that contains a specific Python interpreter and its standard library, along with any additional packages or modules needed for a particular project. It helps in managing dependencies and isolating project-specific libraries, ensuring that different projects can have their own dependencies without interfering with each other.**

**We use virtualenv to create a virtual environment in Python.**

**Here's why virtual environments are important:**

**1. Isolation: Virtual environments keep each project's dependencies separate, so changes in one project don't affect others.**

**2. Dependency Management: Different projects can use different library versions, avoiding conflicts and enabling work on multiple projects with distinct requirements.**

**3. Clean Environments:Virtual environments offer a fresh start for each project, preventing clashes between library versions and maintaining a tidy system-wide Python installation.**

**4. Consistent Dev Environment: Virtual environments help maintain similarity between development and production setups, minimizing deployment issues.**

**Basic Commands in python virtual environment:**

**Installing virtualenv:**

**$ pip install virtualenv**

**Test your installation:**

**$ virtualenv --version**

* **mkvirtualenv**
* **rmvirtualenv**
* **workon**
* **deactivate**

**1. mkvirtualenv - Make Virtual Environment:**

**- Command: `mkvirtualenv env\_name`**

**- Creates a new virtual environment named `env\_name`. This command automatically activates the virtual environment after creation.**

**2. rmvirtualenv - Remove Virtual Environment:**

**- Command: `rmvirtualenv env\_name`**

**- Deletes the virtual environment named `env\_name`. Be cautious, as this action is irreversible, and it permanently removes the virtual environment.**

**3. workon - Activate Virtual Environment:**

**- Command: `workon env\_name`**

**- Activates the specified virtual environment named `env\_name`. After activation, any subsequent package installations or Python commands will apply to this virtual environment.**

**4. deactivate - Deactivate Virtual Environment:**

**- Command: `deactivate`**

**- Deactivates the currently active virtual environment, returning to the system-wide Python environment.**

**What is HTTP and HTTP Methods?**

**The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers.**

**HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.**

**HTTP works as a request-response protocol between a client and server.**

**Example: A client (browser) sends an HTTP request to the server; then the server returns a response to the client. The response contains status information about the request and may also contain the requested content.**

**Each HTTP request method has a specific purpose, and they are used by clients to communicate with servers. Here are some commonly used HTTP request methods:**

**HTTP Methods**

1. **GET:retrieves requested resources.**
2. **POST:send data to the server to create or update resources**
3. **PUT:resource creation or replacement.**
4. **HEAD:gets headers and metadata about a resource.**
5. **DELETE:for deletions of specified resources.**
6. **PATCH: for updates,apply partial modifications to a resource.**
7. **OPTIONS:describes the communication options for the target resource.**
8. **CONNECT:start a two-way communications (a tunnel) with the requested resource.**
9. **TRACE:perform a message loop-back test that tests the path for the target resource (useful for debugging purposes).**

**The two most common HTTP methods are: GET and POST.**

**1.The GET Method**

**GET is used to request data from a specified resource.The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.**

**Some notes on GET requests:**

* **GET requests can be cached**
* **GET requests remain in the browser history**
* **GET requests can be bookmarked**
* **GET requests should never be used when dealing with sensitive data**
* **GET requests have length restrictions**
* **GET requests are only used to request data (not modify)**

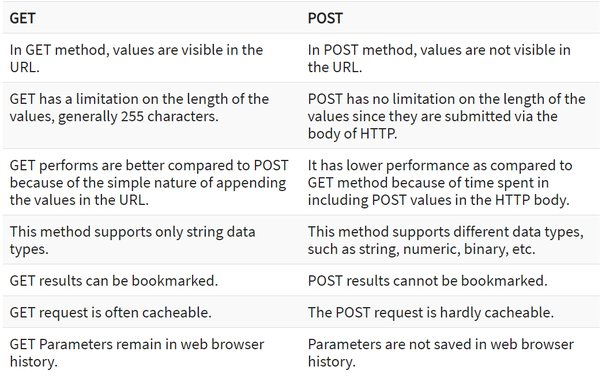
**The POST Method**

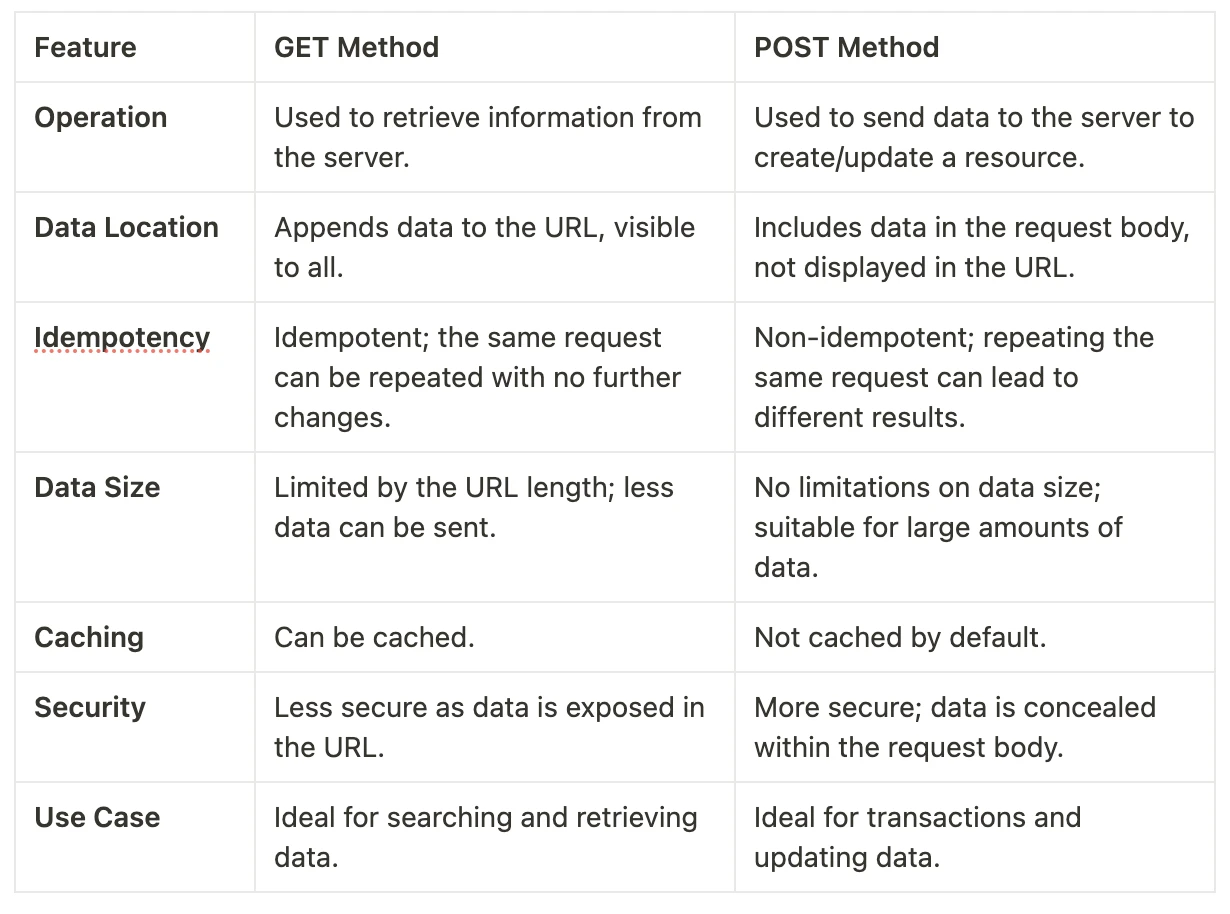
**POST is used to send data to a server to create/update a resource.The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.**

**Some notes on POST requests:**

* **POST requests are never cached**
* **POST requests do not remain in the browser history**
* **POST requests cannot be bookmarked**
* **POST requests have no restrictions on data length**

**Difference between post and get**

****

****

**Here are other HTTP request methods:**

**1. GET:**

**- The GET method is used to retrieve information from the server.**

**- It is a safe and idempotent method, meaning that multiple identical requests should have the same effect as a single request.**

**2. POST:**

**- The POST method is used to submit data to be processed to a specified resource.**

**- It is not idempotent, meaning that multiple identical requests may have different effects.**

**3. PUT:**

**- The PUT method is used to update or replace a resource or create a new resource if it does not exist.**

**- It is idempotent, meaning that multiple identical requests should have the same effect as a single request.**

**4. DELETE:**

**- The DELETE method is used to request that a resource be removed.**

**- It is idempotent, meaning that multiple identical requests should have the same effect as a single request.**

**5. PATCH:**

**- The PATCH method is used to apply partial modifications to a resource.**

**- It is not idempotent, meaning that multiple identical requests may have different effects.**

**6. HEAD:**

**- The HEAD method is similar to GET but does not return a message body. It is used to obtain the headers of a resource for information like the last modification time.**

**7. \*\*OPTIONS:\*\***

**- The OPTIONS method is used to describe the communication options for the target resource.**

**- It is often used to request information about the allowed methods for a resource.**

**8. \*\*TRACE:\*\***

**- The TRACE method is used to perform a message loop-back test along the path to the target resource.**

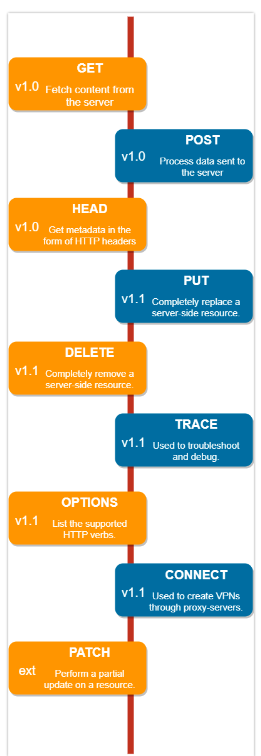
**- It is typically used for diagnostic purposes.**

**9. \*\*CONNECT:\*\***

**- The CONNECT method is used to establish a tunnel to the server identified by a given URI.**

**- It is often used for SSL/TLS encrypted communication.**

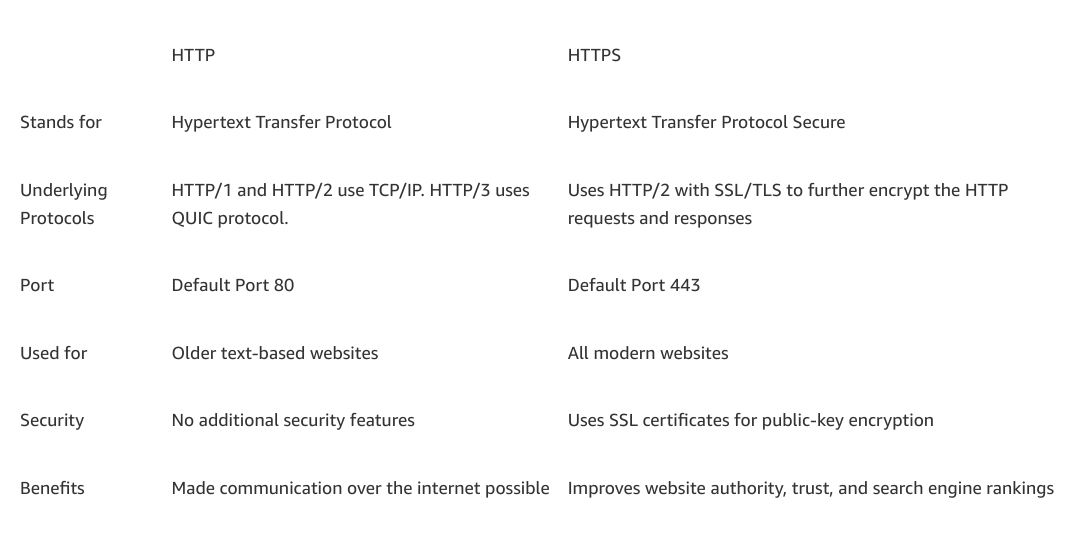
**These HTTP methods define the operations that can be performed on a resource. When a client sends an HTTP request to a server, it includes the request method to indicate the desired action. The server then processes the request and responds accordingly.**

****

**Put vs post**

**The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly have side effects of creating the same resource multiple times.**

**Http vs Https**

****

**Http Status Codes:**

**An HTTP status code is a server response to a browser’s request. When you visit a website, your browser sends a request to the site’s server, and the server then responds to the browser’s request with a three-digit code: the HTTP status code.**

**Status codes are grouped into five classes based on their first digit, and each class has a specific meaning.**

**The first digit of each three-digit status code begins with one of five numbers, 1 through 5; you may see this expressed as 1xx or 5xx to indicate status codes in that range. Each of those ranges encompasses a different class of server response.**

**Advantages of HTTP Status Codes:**

**Standardization: HTTP status codes provide a standardized way to communicate information about the outcome of a request. This standardization facilitates interoperability between clients and servers.**

**Diagnostic Information: Status codes help developers and administrators diagnose issues quickly by providing specific information about the success or failure of a request.**

**Error Handling: The use of status codes allows for effective error handling. Clients can take appropriate actions based on the received status code, whether it's retrying the request, redirecting, or displaying an error message.**

**Automation: Status codes enable automation of certain behaviors. For example, clients can automatically follow redirects, or certain errors can trigger automated error-handling processes.**

**Common HTTP status code classes:**

**1xxs – Informational responses: The server is thinking through the request.**

**These codes indicate that the request was received, continuing process, or the server is in the process of continuing the request.**

**2xxs – Success! The request was successfully completed and the server gave the browser the expected response.These codes indicate that the request was successfully received, understood, and accepted.**

**3xxs – Redirection: You got redirected somewhere else. The request was received, but there’s a redirect of some kind.**

**These codes indicate that further action needs to be taken to complete the request. They often involve redirection to a different resource or URL.**

**4xxs – Client errors: Page not found. The site or page couldn’t be reached. (The request was made, but the page isn’t valid — this is an error on the website’s side of the conversation and often appears when a page doesn’t exist on the site.)**

**These codes indicate that the client seems to have made an error in the request. Commonly seen when the client sends an invalid request.**

**5xxs – Server errors: Failure. A valid request was made by the client but the server failed to complete the request.**

**These codes indicate that the server failed to fulfill a valid request. It typically means there is an issue on the server side.**

**Common HTTP Status Codes:**

**200 OK: The request was successful.**

**201 Created: The request resulted in the creation of a new resource.**

**204 No Content: The request was successful, but there is no new information to send back.**

**400 Bad Request: The server did not understand the request.**

**401 Unauthorized: Authentication is required to access the resource.**

**403 Forbidden: The server understood the request but refuses to authorize it.**

**404 Not Found: The requested resource could not be found.**

**500 Internal Server Error: A generic error message returned when an unexpected condition was encountered on the server.**

**502 Bad Gateway: The server, while acting as a gateway or proxy, received an invalid response from an inbound server.**

**503 Service Unavailable: The server is not ready to handle the request.**

**Database CRUD:**

**CRUD stands for Create, Read, Update, and Delete, which are the basic operations that can be performed on data in a database. Here are examples of CRUD queries for a hypothetical database:**

**### 1. \*\*Create (C):\*\***

**- \*\*SQL (Structured Query Language):\*\***

**```sql**

**INSERT INTO users (username, email, age) VALUES ('john\_doe', 'john@example.com', 25);**

**```**

**This query creates a new record in the `users` table with the specified values for `username`, `email`, and `age`.**

**### 2. \*\*Read (R):\*\***

**- \*\*SQL:\*\***

**```sql**

**SELECT \* FROM users WHERE age > 18;**

**```**

**This query retrieves all columns (`\*`) from the `users` table for records where the `age` is greater than 18.**

**### 3. \*\*Update (U):\*\***

**- \*\*SQL:\*\***

**```sql**

**UPDATE users SET age = 26 WHERE username = 'john\_doe';**

**```**

**This query updates the `age` column to 26 in the `users` table for the record where the `username` is 'john\_doe'.**

**### 4. \*\*Delete (D):\*\***

**- \*\*SQL:\*\***

**```sql**

**DELETE FROM users WHERE username = 'john\_doe';**

**```**

**This query deletes the record from the `users` table where the `username` is 'john\_doe'.**

**These are basic examples, and the exact syntax may vary depending on the database management system (DBMS) you are using (e.g., MySQL, PostgreSQL, SQLite, etc.). Always refer to the specific documentation of the DBMS you are working with for accurate syntax and details. Additionally, remember to handle user input and SQL injection vulnerabilities properly in real-world applications.**

**-- Create the hello database**

**CREATE DATABASE hello;**

**-- Switch to the hello database**

**USE hello;**

**-- Create the user table**

**CREATE TABLE user (**

**id INT PRIMARY KEY NOT NULL AUTO\_INCREMENT,**

**first\_name VARCHAR(255),**

**last\_name VARCHAR(255),**

**email VARCHAR(255),**

**password VARCHAR(255),**

**status VARCHAR(255)**

**);**

**-- Create the treks table**

**CREATE TABLE treks (**

**trek\_id INT PRIMARY KEY NOT NULL AUTO\_INCREMENT,**

**trek\_name VARCHAR(255),**

**trek\_location VARCHAR(255),**

**difficulty\_level VARCHAR(50),**

**user\_id INT,**

**FOREIGN KEY (user\_id) REFERENCES user(id)**

**);**

**-- Sample data for the treks table**

**INSERT INTO treks (trek\_name, trek\_location, difficulty\_level, user\_id)**

**VALUES**

**('Mountain Peak Trek', 'Himalayas', 'Challenging', 1),**

**('Forest Adventure Trek', 'Amazon Rainforest', 'Moderate', 2),**

**('Coastal Exploration Trek', 'Great Barrier Reef', 'Easy', 3);**

**-- Perform a simple join between the user and treks tables**

**SELECT**

**user.id AS user\_id,**

**user.first\_name,**

**user.last\_name,**

**treks.trek\_name,**

**treks.trek\_location,**

**treks.difficulty\_level**

**FROM**

**user**

**JOIN**

**treks ON user.id = treks.user\_id;**

**from flask import Flask, jsonify, render\_template, request**

**# jsonify converts python object into JSON**

**# request is used to get data from form or json**

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

**# {userId : {name, age}}**

**users = {**

**1: {"name": "Ram", "age": 21},**

**2: {"name": "Sam", "age": 23},**

**}**

**@app.route("/get-all-user", methods=["GET"])**

**def getAllUser():**

**return jsonify(users)**

**@app.route("/user/<int:id>", methods=["GET"])**

**def getUser(id):**

**user = users.get(id)**

**return jsonify(user)**

**@app.route("/add-user", methods=["POST"])**

**def addUser():**

**req = request.json**

**name = req.get("name")**

**age = req.get("age")**

**users\_count = max(users.keys())**

**new\_user\_key = users\_count + 1**

**users[new\_user\_key] = {"name": name, "age": age}**

**return jsonify(users)**

**@app.route("/add-user-from-form", methods=["POST"])**

**def addUserFromForm():**

**name = request.form["name"]**

**age = request.form["age"]**

**users\_count = max(users.keys())**

**new\_user\_key = users\_count + 1**

**users[new\_user\_key] = {"name": name, "age": age}**

**return jsonify(users)**

**@app.route("/user/<int:id>", methods=["DELETE"])**

**def deleteUser(id):**

**user = users.pop(id)**

**return jsonify(users)**

**@app.route("/user/<int:id>", methods=["PUT"])**

**def updateUser(id):**

**req = request.json**

**age = req.get("age")**

**users[id] = {"name": users[id]["name"], "age": age}**

**return jsonify(users[id])**

**@app.route("/form")**

**def formPage():**

**return render\_template("form.html")**

**@app.route("/json")**

**def jsonPage():**

**return render\_template("json.html")**

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

**app.run(debug=True)**

**HTML Form for Submitting JSON Payload:**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>JSON Payload Submission</title>**

**</head>**

**<body>**

**<h1>Add User Data via JSON Payload</h1>**

**<label for="title">Name:</label>**

**<input type="text" id="name" required><br>**

**<label for="category">Age:</label>**

**<input type="text" id="age" required><br>**

**<button type="button" onclick="submitJson()">Submit JSON</button>**

**<script>**

**function submitJson() {**

**const data = {**

**"name": document.getElementById("name").value,**

**"age": document.getElementById("age").value,**

**};**

**fetch('/add-user', {**

**method: 'POST',**

**headers: {**

**'Content-Type': 'application/json',**

**},**

**body: JSON.stringify(data),**

**})**

**.then(response => response.json())**

**.then(data => {**

**console.log('Success:', data);**

**})**

**.catch((error) => {**

**console.error('Error:', error);**

**});**

**}**

**</script>**

**</body>**

**</html>**

### **HTML Form for Submitting Data:**

**<!DOCTYPE html>**

**<html lang="en">**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>Form Submission</title>**

**</head>**

**<body>**

**<h1>Submit Data via Form</h1>**

**<form action="/add-user-from-form" method="POST">**

**<label for="title">Name:</label>**

**<input type="text" id="title" name="name" required><br>**

**<label for="category">Age:</label>**

**<input type="text" id="category" name="age" required><br>**

**<input type="submit" value="Submit">**

**</form>**

**</body>**

**</html>**

**Web services API and Documentation**

**Endpoint 1: Get All Users**

**Description: Retrieve details of all users.**

**URL: /get-all-user**

**Method: GET**

**Parameters: None**

**Status Code: 200 OK**

**Example Request: curl -X GET http://localhost:5000/get-all-user**

**Response:**

**{**

**"1": { "name": "Ram", "age": 21 },**

**"2": { "name": "kushal", "age": 23 },**

**"3": { "name": "Rupesh", "age": 22 },**

**"4": { "name": "Sam", "age": 20 }**

**}**

**Endpoint 2: Get User by ID**

**Description: Retrieve details of a specific user by ID.**

**URL: /user/{id}**

**Method: GET**

**Parameters:**

**{id}: User ID (integer)**

**Status Code: 200 OK if user found, 404 Not Found if user not found**

**Example Request: curl -X GET http://localhost:5000/user/1**

**Response:**

**{ "name": "Ram", "age": 21 }**

**Endpoint 3: Add User**

**Description: Add a new user.**

**URL: /add-user**

**Method: POST**

**Parameters:**

**JSON payload with "name" and "age" fields**

**Status Code: 200 OK**

**Example Request:**

**bashCopy code**

**curl -X POST -H "Content-Type: application/json" -d '{"name": "NewUser", "age": 25}' http://localhost:5000/add-user**

**Response:**

**jsonCopy code**

**{**

**"1": { "name": "Ram", "age": 21 },**

**"2": { "name": "kushal", "age": 23 },**

**"3": { "name": "Rupesh", "age": 22 },**

**"4": { "name": "Sam", "age": 20 },**

**"5": { "name": "NewUser", "age": 25 }**

**}**

**Endpoint 4: Add User from Form**

**Description: Add a new user from an HTML form.**

**URL: /add-user-from-form**

**Method: POST**

**Parameters:**

**Form data with "name" and "age" fields**

**Status Code: 200 OK**

**Example Request: curl -X POST -d 'name=NewUser&age=25' http://localhost:5000/add-user-from-form**

**Response:**

**{**

**"1": { "name": "Ram", "age": 21 },**

**"2": { "name": "kushal", "age": 23 },**

**"3": { "name": "Rupesh", "age": 22 },**

**"4": { "name": "Sam", "age": 20 },**

**"5": { "name": "NewUser", "age": 25 }**

**}**

**Endpoint 5: Delete User by ID**

**Description: Delete a user by ID.**

**URL: /user/{id}**

**Method: DELETE**

**Parameters:**

**{id}: User ID (integer)**

**Status Code: 200 OK if user found and deleted, 404 Not Found if user not found**

**Example Request: curl -X DELETE http://localhost:5000/user/1**

**Response:**

**{**

**"2": { "name": "kushal", "age": 23 },**

**"3": { "name": "Rupesh", "age": 22 },**

**"4": { "name": "Sam", "age": 20 },**

**"5": { "name": "NewUser", "age": 25 }**

**}**

**Endpoint 6: Update User by ID**

**Description: Update a user's age by ID.**

**URL: /user/{id}**

**Method: PUT**

**Parameters:**

**{id}: User ID (integer)**

**JSON payload with "age" field**

**Status Code: 200 OK if user found and updated, 404 Not Found if user not found**

**Example Request: curl -X PUT -H "Content-Type: application/json" -d '{"age": 26}' http://localhost:5000/user/5**

**Response:**

**{**

**"2": { "name": "kushal", "age": 23 },**

**"3": { "name": "Rupesh", "age": 22 },**

**"4": { "name": "Sam", "age": 20 },**

**"5": { "name": "NewUser", "age": 26 }**

**}**

**Endpoint 7: Render Form Page**

**Description: Render an HTML form page.**

**URL: /form**

**Method: GET**

**Parameters: None**

**Status Code: 200 OK**

**Example Request:**

**curl -X GET http://localhost:5000/form**

**Response: HTML page rendering a form.**

**Endpoint 8: Render JSON Page**

**Description: Render an HTML page for JSON content.**

**URL: /json**

**Method: GET**

**Parameters: None**

**Status Code: 200 OK**

**Example Request:**

**curl -X GET http://localhost:5000/json**

**Response: HTML page rendering a form.**