**LAB : 7**

Sure, let's break down the **Server.py** script snippet by snippet:

python

Copy code

from flask import Flask, request, jsonify app = Flask(\_\_name\_\_)

1. **Importing Flask**: This line imports the Flask class from the Flask module, which is a popular web framework for Python.
2. **Creating Flask App**: This line creates a Flask application instance named **app**.

python

Copy code

@app.route("/add", methods=["POST"]) def add(): data = request.get\_json() num1 = data['num1'] num2 = data['num2'] num3 = num1 + num2 return jsonify({"result":num3})

1. **Defining Route for Addition**: This snippet defines a route "/add" using the **@app.route()** decorator. It specifies that this route accepts POST requests.
2. **add() Function**: This function is executed when a POST request is made to the "/add" route. It retrieves JSON data from the request using **request.get\_json()**, extracts two numbers (**num1** and **num2**) from the JSON data, calculates their sum, and then returns the result as JSON using **jsonify()**.

python

Copy code

@app.route("/multiply", methods=["POST"]) def multiply(): data = request.get\_json() num1 = data['num1'] num2 = data['num2'] num3 = num1 \* num2 return jsonify({"result":num3})

1. **Defining Route for Multiplication**: Similar to the previous snippet, this defines a route "/multiply" for handling POST requests.
2. **multiply() Function**: This function is executed when a POST request is made to the "/multiply" route. It retrieves JSON data from the request, extracts two numbers (**num1** and **num2**), calculates their product, and returns the result as JSON.

python

Copy code

if \_\_name\_\_ == '\_\_main\_\_': app.run(debug=True)

1. **Running the Application**: This conditional block ensures that the Flask application is only run when the script is executed directly (not imported as a module). It starts the Flask development server with debugging enabled (**debug=True**), allowing for easier debugging during development.

Overall, this script creates a Flask application with two routes ("/add" and "/multiply") for performing addition and multiplication operations, respectively, and starts a development server to handle incoming requests.

Let's go through the **Client.py** script step by step:

python

Copy code

import requests

1. **Importing Requests Module**: This line imports the **requests** module, which is a popular HTTP library for making HTTP requests in Python.

python

Copy code

url = "http://127.0.0.1:5000/"

1. **Defining Server URL**: This line initializes a variable **url** with the base URL of the server where the Flask application is running. In this case, it's set to "<http://127.0.0.1:5000/>", which represents the local machine (127.0.0.1) on port 5000.

python

Copy code

def add(num1,num2): endpoint = url + "add" data = {"num1" : num1, "num2": num2} response = requests.post(endpoint, json=data) result = response.json()["result"] return result

1. **add() Function**: This function takes two numbers (**num1** and **num2**) as input parameters. It constructs the complete URL for the "/add" endpoint by appending "add" to the base URL (**url**). It then creates a dictionary **data** containing the numbers to be sent as JSON data in the request body. It makes a POST request to the specified endpoint using **requests.post()**, passing the JSON data. Finally, it extracts the result from the response JSON and returns it.

python

Copy code

def multiply(num1, num2): endpoint = url + "multiply" data = {"num1" : num1, "num2" : num2} response = requests.post(endpoint, json=data) result = response.json()["result"] return result

1. **multiply() Function**: Similar to the **add()** function, this function takes two numbers (**num1** and **num2**) as input parameters. It constructs the complete URL for the "/multiply" endpoint, creates a dictionary **data** with the numbers, makes a POST request to the endpoint, and extracts and returns the result from the response JSON.

python

Copy code

state = True while state: print("Enter the first number:") try: num1 = int(input()) print("Enter the second number:") num2 = int(input()) print("Do you want to:\n1. Add\n2. Multiply\n3. Exit") choice = int(input()) if choice == 1: print("Result:", add(num1,num2)) elif choice == 2: print("Result:", multiply(num1,num2)) elif choice == 3: print("Thank you for using the service") state = False except Exception as e: print("Encountered Error:", e) print("Restarting interface")

1. **User Interface Loop**: This part of the code presents a simple command-line interface for the user to interact with the client. It repeatedly asks the user for two numbers and the operation to perform (addition or multiplication) until the user chooses to exit. It catches any exceptions that may occur during user input and restarts the interface if an error occurs.
2. **Server-side Questions**:
   * Flask handles incoming requests by routing them to specific functions based on the URL route defined using the **@app.route()** decorator. This decorator maps a URL pattern to a Python function, allowing the function to be executed when a request matching that pattern is received. The **add()** and **multiply()** functions retrieve JSON data from the request body using **request.get\_json()**, which parses the JSON data sent by the client. Alternative methods for handling request data include accessing form data (**request.form**) for form submissions or request arguments (**request.args**) for URL query parameters.
3. **Client-side Questions**:
   * The **requests** library in Python provides a simple and convenient interface for making HTTP requests to web servers. It abstracts away the complexities of HTTP communication and provides methods like **requests.post()** to send POST requests with JSON data to specific endpoints on the server. Alternative methods for making HTTP requests include libraries like **urllib** or **http.client**, but **requests** is preferred for its ease of use and extensive functionality.
   * To handle user input in the client-side interface loop, precautions should be taken to validate input and handle errors gracefully. This can include using try-except blocks to catch exceptions, validating user input to ensure it's of the correct type, and providing informative error messages to the user in case of invalid input.
4. **Error Handling and Robustness**:
   * Potential points of failure or vulnerabilities in the codebase include invalid input from the user, network errors, or server-side issues such as database connectivity problems. These can be mitigated through robust error handling mechanisms, such as try-except blocks, input validation, and defensive programming techniques.
   * Reliability and robustness of client-server communication can be ensured through techniques like implementing retry mechanisms for failed requests, using timeouts to handle slow responses, and implementing error logging to track and debug issues as they occur.
5. **Scalability and Performance**:
   * Assessing the scalability of the current architecture involves considering factors like the server's capacity to handle concurrent users, response times under load, and resource utilization. To accommodate a larger number of users or higher request volumes, optimizations like load balancing, caching, and asynchronous processing can be implemented.
   * Performance optimizations can include techniques like minimizing unnecessary computation, optimizing database queries, caching frequently accessed data, and utilizing server-side or client-side caching mechanisms to reduce response times.
6. **Security and Authentication**:
   * Security vulnerabilities in the codebase include risks such as injection attacks, data interception, and unauthorized access to sensitive data. These can be addressed through measures like input validation, parameterized queries to prevent SQL injection, encryption of sensitive data in transit, and implementing authentication mechanisms like API keys or OAuth to restrict access to authorized users.
   * Risks associated with exposing the server to external access over the internet include potential DDoS attacks, brute force attacks, and unauthorized access to sensitive data. Mitigation strategies include implementing firewalls, rate limiting, IP whitelisting, and using HTTPS for secure communication.