

ASSIGNMENT – 15.4

NAME: B.SaiGanesh

HT.NO: 2403A52094

BATCH NO: 04

TASK 1:

PROMPT:

Generate a simple Flask backend in Python with one endpoint that returns a JSON message “Welcome to AI-assisted API”.

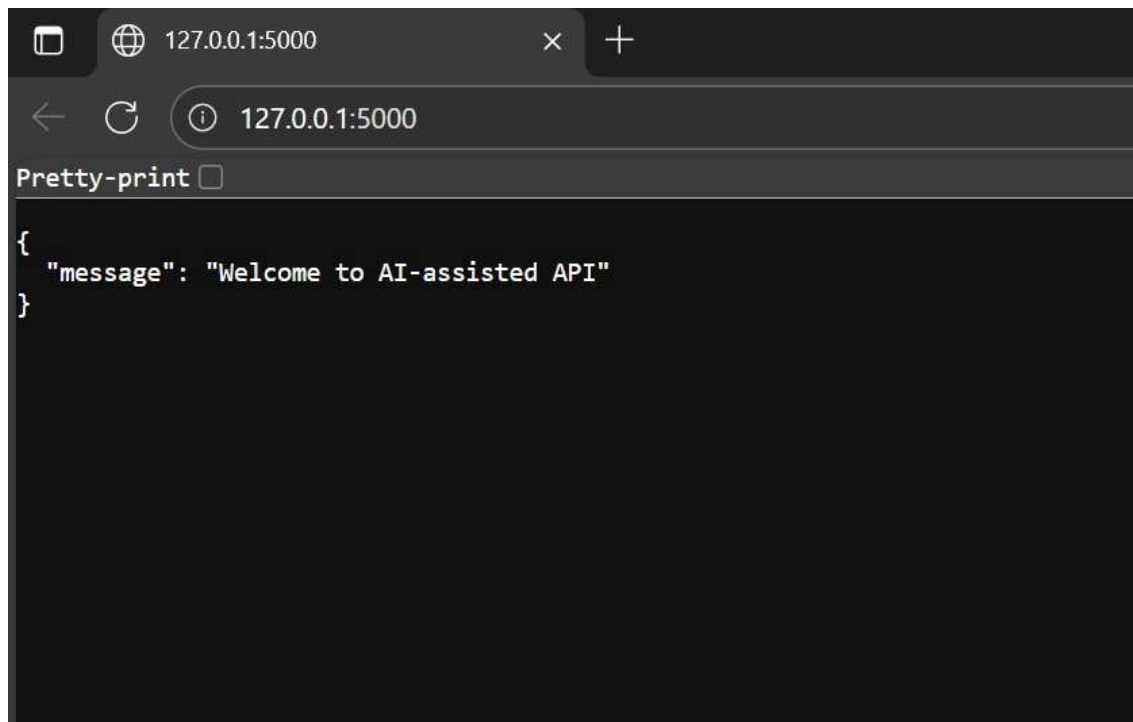
CODE:

```
lab-15 > task1.py > ...
1  from flask import Flask, jsonify
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def home():
7      return jsonify({"message": "Welcome to AI-assisted API"})
8
9  if __name__ == "__main__":
10     app.run(debug=True)
11
```

OUTPUT:

```
in\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding\lab-15\task1.py'
* Serving Flask app 'task1'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server in
stead.
* Running on http://127.0.0.1:5000
* Running on http://127.0.0.1:5000   arting with stat
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
```

In 11, Col 1, Sp



OBSERVATION:

In this code, I observed that Flask is used to create a simple web server with one endpoint. The / route returns a JSON message “Welcome to AI-assisted API” using the jsonify() function. The server runs in debug mode, and when accessed through `http://127.0.0.1:5000/`, it displays the message correctly in the browser or Postman.

TASK 2:

PROMPT:

Create a simple Flask CRUD API with Read (GET) and Create (POST) operations using an in-memory list.

CODE:

```

lab-15 > task2.py > ...
1  from flask import Flask, jsonify, request
2  app = Flask(__name__)
3  items = []
4  @app.route('/')
5  def home():
6      return jsonify({"message": "Welcome to the CRUD API! Try /items"}), 200
7  # GET all items
8  @app.route('/items', methods=['GET'])
9  def get_items():
10     return jsonify(items), 200
11  # POST a new item
12  @app.route('/items', methods=['POST'])
13  def add_item():
14     data = request.get_json()
15     if not data or "name" not in data:
16         return jsonify({"error": "Invalid item data"}), 400
17     item = {
18         "id": len(items) + 1,
19         "name": data["name"]
20     }
21     items.append(item)
22     return jsonify({"message": "Item added", "item": item}), 201
23 if __name__ == "__main__":
24     app.run(debug=True)
25

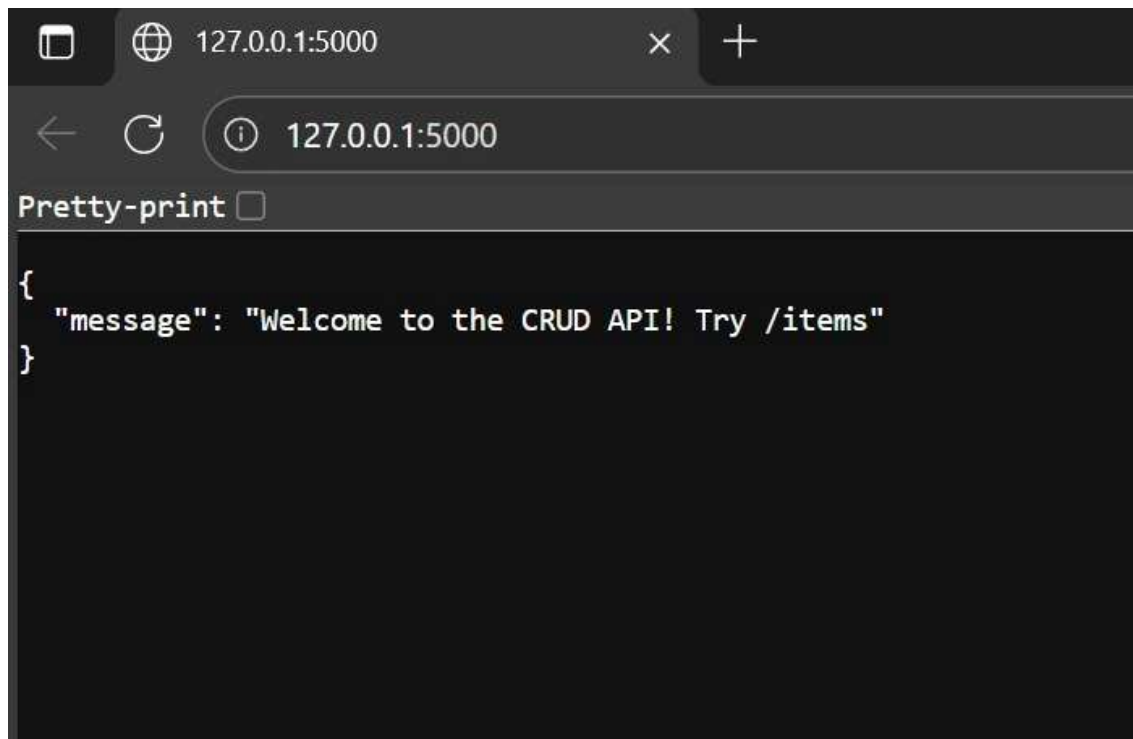
```

OUTPUT:

```

* Serving Flask app 'task2'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 747-951-939
127.0.0.1 - - [23/Oct/2025 12:24:11] "GET / HTTP/1.1" 200 -

```

A screenshot of a web browser window. The address bar shows '127.0.0.1:5000'. Below the address bar, there is a 'Pretty-print' checkbox. The main content area displays a JSON object:

```
{  "message": "Welcome to the CRUD API! Try /items"}
```

OBSERVATION:

In this code, I observed that Flask is used to create two API endpoints for reading and adding items. The /items route with the GET method returns all items stored in the in-memory list, while the POST method adds a new item received as JSON data. The added item is appended to the list and a success message is returned. Since the data is stored in memory, it will reset when the server restarts. This provides a simple demonstration of basic CRUD operations.

TASK 3:

PROMPT:

Create a PUT endpoint in Flask to update an existing item by its index.

CODE:

```

lab-15 > task3.py > ...
1  from flask import Flask, jsonify, request
2  app = Flask(__name__)
3  # In-memory list
4  items = []
5  # Home route
6  @app.route('/')
7  def home():
8      return jsonify({"message": "Welcome to the CRUD API! Try /items"}), 200
9  # GET all items
10 @app.route('/items', methods=['GET'])
11 def get_items():
12     return jsonify(items), 200
13 # POST a new item
14 @app.route('/items', methods=['POST'])
15 def add_item():
16     data = request.get_json()
17     if not data:
18         return jsonify({"error": "Invalid item data"}), 400
19     items.append(data)
20     return jsonify({"message": "Item added", "item": data}), 201
21 # PUT /items/<int:index> - update an existing item
22 @app.route('/items/<int:index>', methods=['PUT'])
23 def update_item(index):
24     if index < 0 or index >= len(items):
25         return jsonify({"error": "Item not found"}), 404
26     data = request.get_json()
27     if not data:
28         return jsonify({"error": "Invalid item data"}), 400
29     items[index] = data
30     return jsonify({"message": "Item updated", "item": data}), 200
31 if __name__ == "__main__":
32     app.run(debug=True)

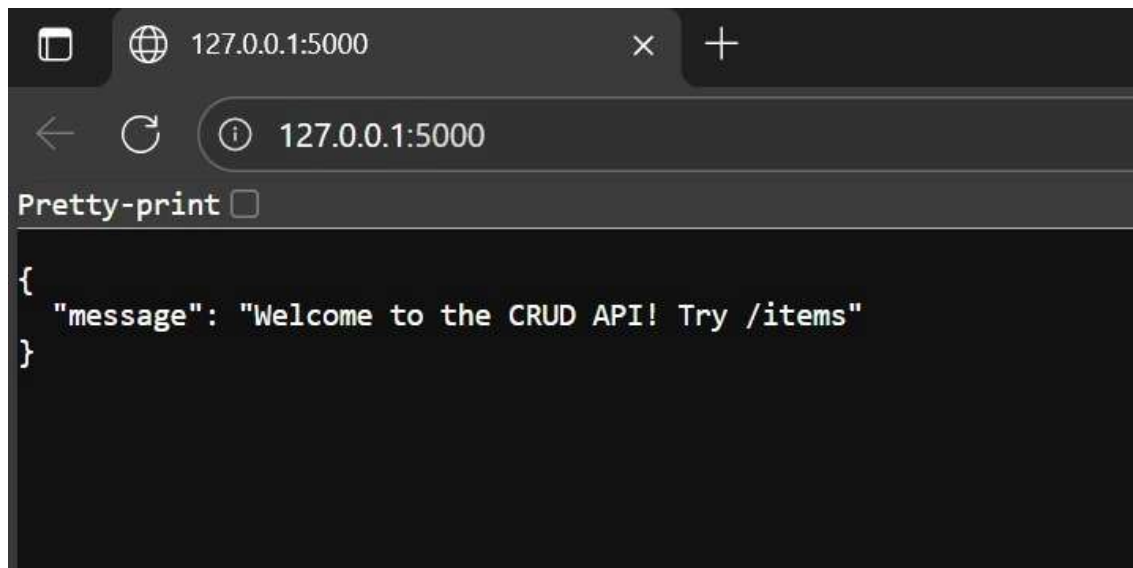
```

OUTPUT:

```

* Serving Flask app 'task3'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 747-951-939
127.0.0.1 - - [23/Oct/2025 12:30:05] "GET / HTTP/1.1" 200 -
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 747-951-939
127.0.0.1 - - [23/Oct/2025 12:30:05] "GET / HTTP/1.1" 200 -

```

A screenshot of a web browser window with a dark theme. The address bar shows '127.0.0.1:5000'. Below the address bar, there is a 'Pretty-print' toggle switch which is currently turned off. The main content area displays a JSON object:

```
{  "message": "Welcome to the CRUD API! Try /items"}
```

OBSERVATION:

This Flask code defines a PUT endpoint allows updating an existing item in the list based on its index. It first checks if the given index is valid to avoid errors when accessing non-existent items. The new data for the update is received as a JSON payload from the client request and replaces the old item in the list. Finally, the API returns a JSON response containing a success message along with the updated item details, confirming that the update was successful.