SET-5

PRACTICAL-1

❖ AIM : A program that creates a simple RESTful API that returns list of users in JSON format.

Problem Statement:

Develop a simple RESTful API that returns a list of users in JSON format. The API should provide endpoints for retrieving the list of users, adding new users, updating existing user information, and deleting users. Additionally, implement basic authentication to secure access to the API endpoints.

Program Description:

Creating a simple RESTful API to manage user data is a fundamental task in web development. The API serves as a backend service that allows clients to interact with user data through standardized HTTP methods and JSON-formatted payloads. The objective of this project is to design and implement such an API using a framework or library of your choice (e.g., Flask, Django, Express.js).

Procedure:

1) Dependencies:

- We'll use Flask, a lightweight web framework for Python, and the json library for working with JSON data. Install them using pip:

bash

```
pip install flask
```

2) Create a Python file and paste the code inside:

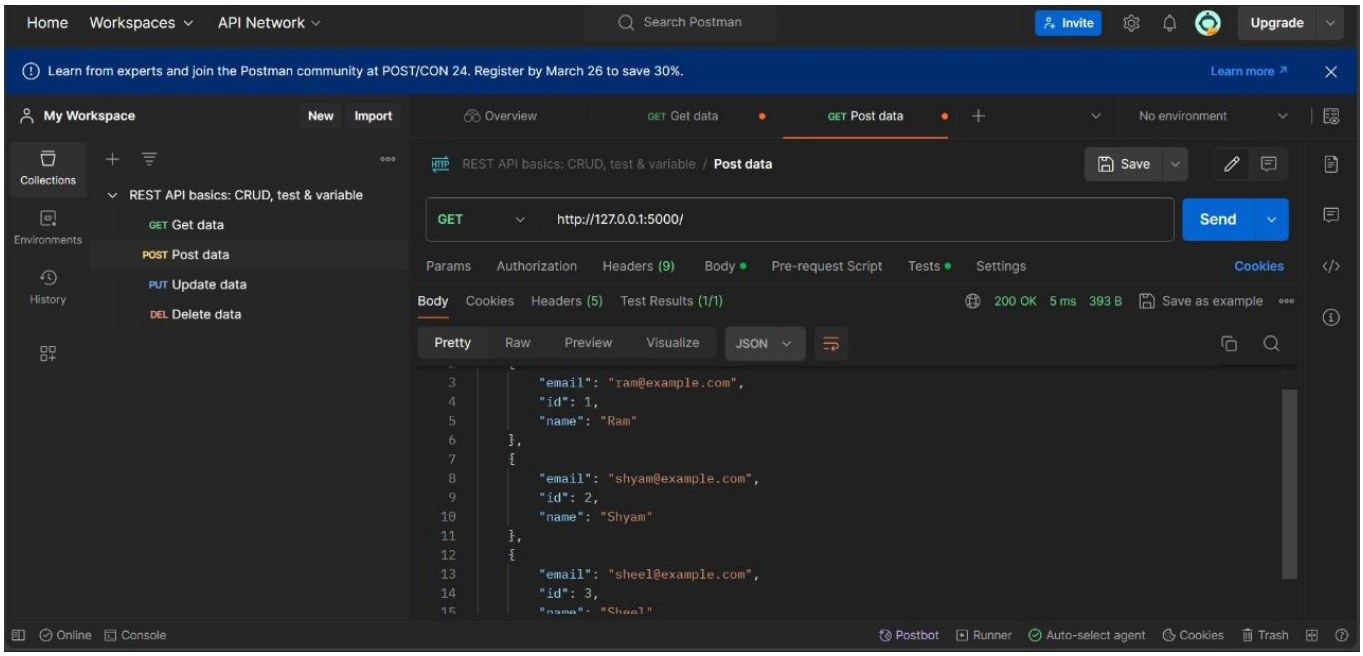
- Define your python app.

app.py

```
from flask import Flask, jsonify

app = Flask(__name__)

# Sample user data (replace with database connection if needed)

users = [

  {"id": 1, "name": "Ram", "email": "ram@example.com"},

  {"id": 2, "name": "Shyam", "email": "shyam@example.com"},
```

```
  {"id": 3, "name": "Sheel", "email": "sheel@example.com"},

]

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

def get_users():

 """Returns a list of users in JSON format"""

 return jsonify(users)


if __name__ == '__main__':

 app.run(debug=True)
```

Expected Output:



Result:

This program fulfills the goal of creating a simple RESTful API that acts as a bridge between user data and external applications. It achieves this by offering a standardized way (JSON format) to access a list of users through a well-defined API structure, promoting easy data exchange and potential future expansion.

PRACTICAL- 2

- AIM: A program that creates a RESTful API that allows users to create, read, update, and delete resource.

Problem Statement:

Develop a RESTful API that enables users to perform CRUD (Create, Read, Update, Delete) operations on a specific resource. The API should provide endpoints for creating new instances of the resource, retrieving existing instances, updating instance attributes, and deleting instances.

Program Description:

Creating a RESTful API that supports CRUD operations is a foundational task in web development. The API acts as a backend service, exposing endpoints through which clients can interact with the underlying resource. This project aims to design and implement such an API using a suitable framework or library.

Procedure:

1) Dependencies:

- We'll use Flask, a lightweight web framework for Python, and the json library for working with JSON data. Install them using pip:

bash

```
pip install flask
```

2) Create a Python file and paste the code inside:

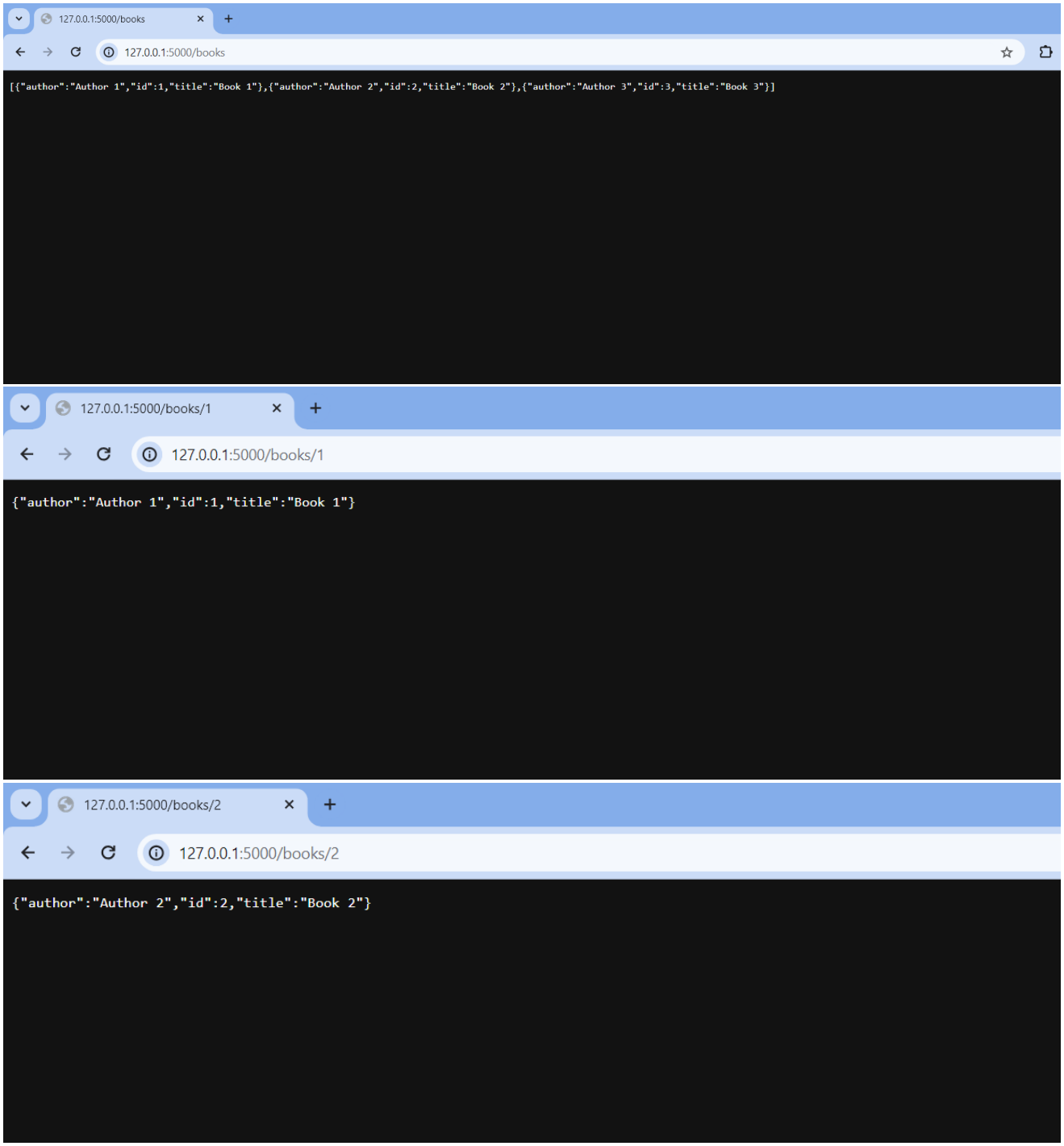
- Define your python app.

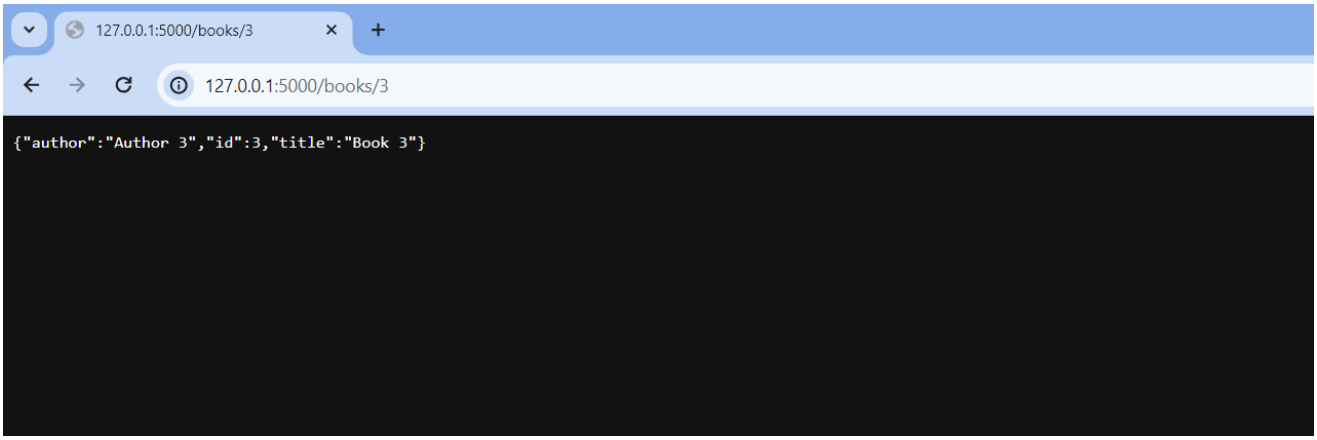
app.py

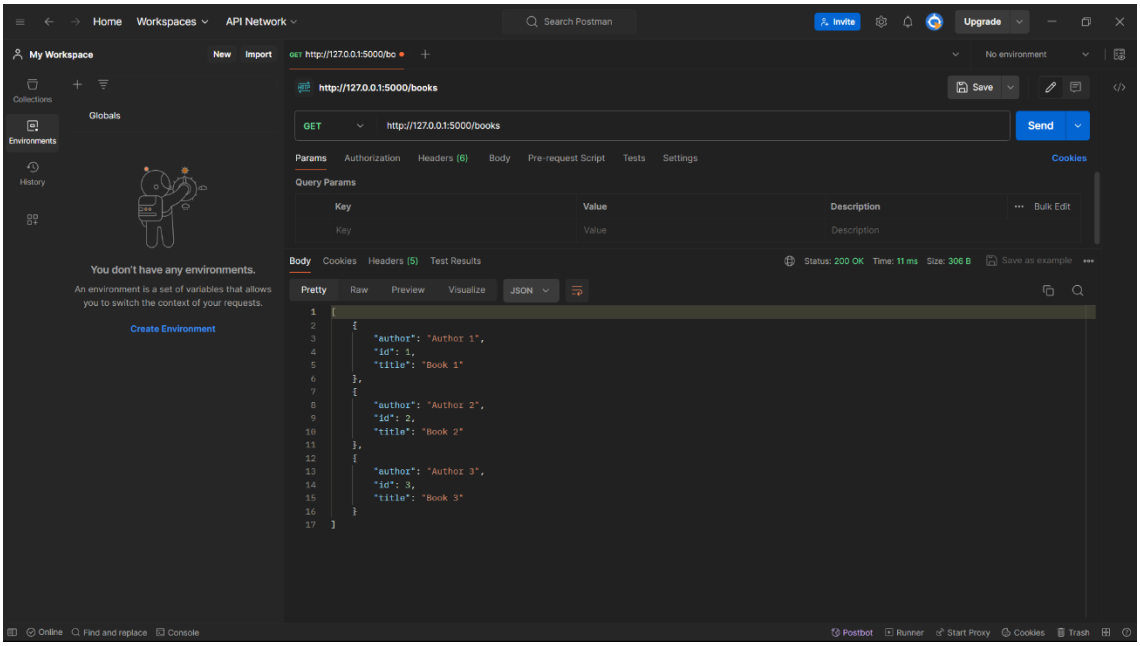
```
from flask import Flask, jsonify, request

app = Flask(__name__)
books = [
    {"id": 1, "title": "Book 1", "author": "Author 1"},
    {"id": 2, "title": "Book 2", "author": "Author 2"},
    {"id": 3, "title": "Book 3", "author": "Author 3"},
]


@app.route("/books", methods=["GET"])
def get_books():
    return jsonify(books)
```

```
@app.route("/books/<int:book_id>", methods=["GET"])
def get_book(book_id):
    book = next((b for b in books if b["id"] == book_id), None)

    if book:
        return jsonify(book)
    else:
        return jsonify({"error": "Book not found"}), 404


@app.route("/books", methods=["POST"])
def create_book():
    data = request.get_json()
    new_book = {"id": len(books) + 1, "title": data["title"],
"author": data["author"]}
    books.append(new_book)
    return jsonify(new_book), 201


@app.route("/books/<int:book_id>", methods=["PUT"])
def update_book(book_id):
    book = next((b for b in books if b["id"] == book_id), None)
    if book:
        data = request.get_json()
        book["title"] = data["title"]
        book["author"] = data["author"]
        return jsonify(book)
    else:
        return jsonify({"error": "Book not found"}), 404


@app.route("/books/<int:book_id>", methods=["DELETE"])
def delete_book(book_id):
    global books
    books = [b for b in books if b["id"] != book_id]
    return jsonify({"result": True})


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

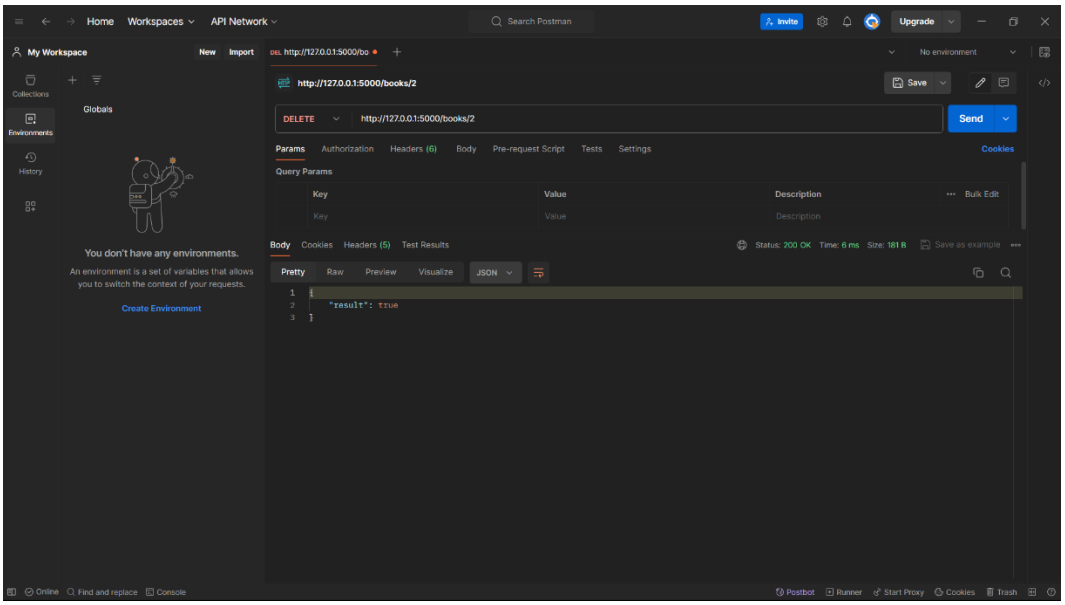
Expected Output:

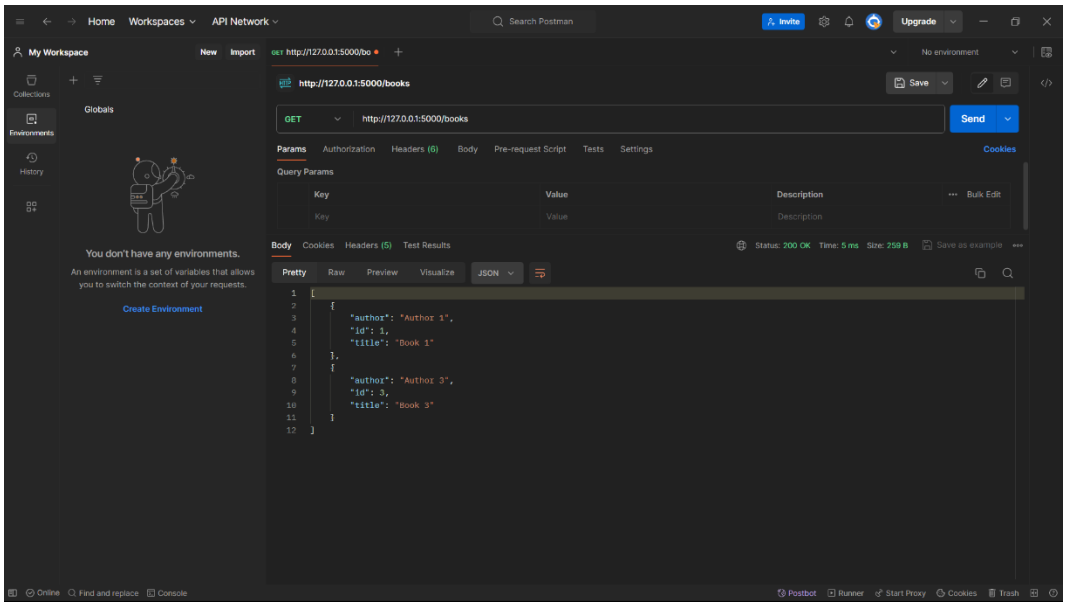




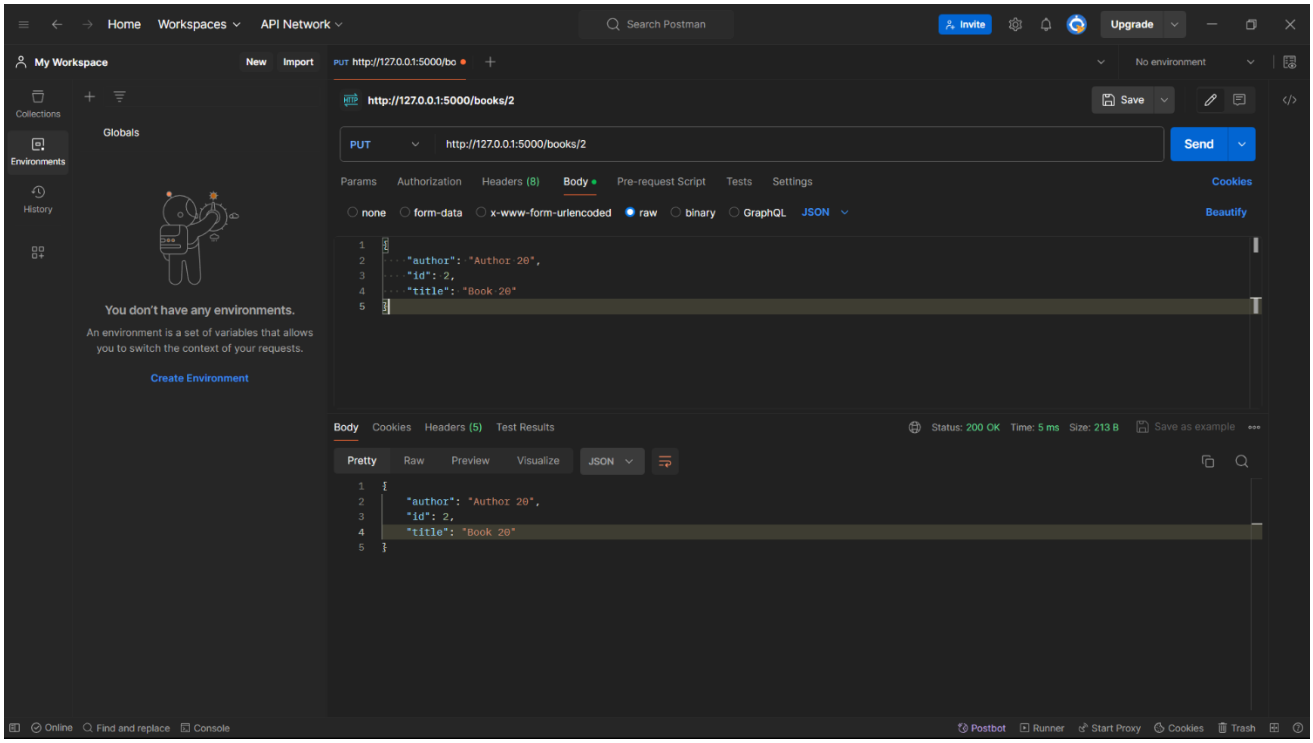


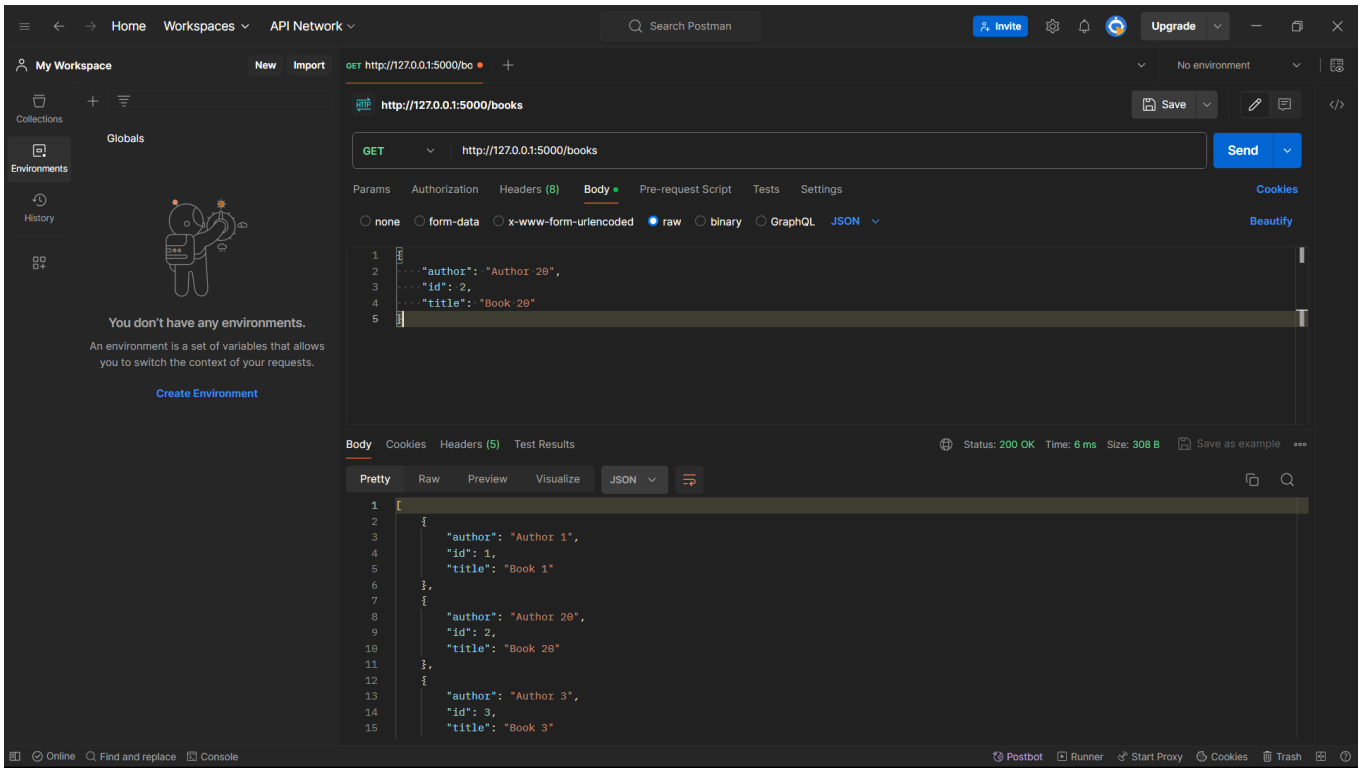
Delete operation:





Update (PUT method) operation:





Result:

This program successfully establishes a versatile RESTful API that empowers users with comprehensive resource management capabilities. By implementing CRUD operations (Create, Read, Update, Delete), it allows users to interact with data in a flexible and efficient manner. This program provides a solid foundation for building powerful APIs that facilitate data manipulation within a well-defined RESTful architecture.

PRACTICAL- 3

❖ AIM: A program that creates a RESTful API that authenticates users using a JSON Web Token.

Problem Statement:

Develop a RESTful API that authenticates users using JSON Web Tokens (JWTs). The API should provide endpoints for user registration, user authentication, and protected resources accessible only to authenticated users with valid JWTs.

Program Description:

This Flask API implements secure user access with JWTs. Users log in and obtain a temporary token. This token unlocks protected resources, verified by the server's secret key. It's a secure and efficient way to authenticate users without storing session data.

Procedure:

1) Dependencies:

- We'll use Flask, a lightweight web framework for Python, and the json library for working with JSON data. Install them using pip:

bash

```
pip install flask flask_jwt_extended
```

3) Create a Python file and paste the code inside it:

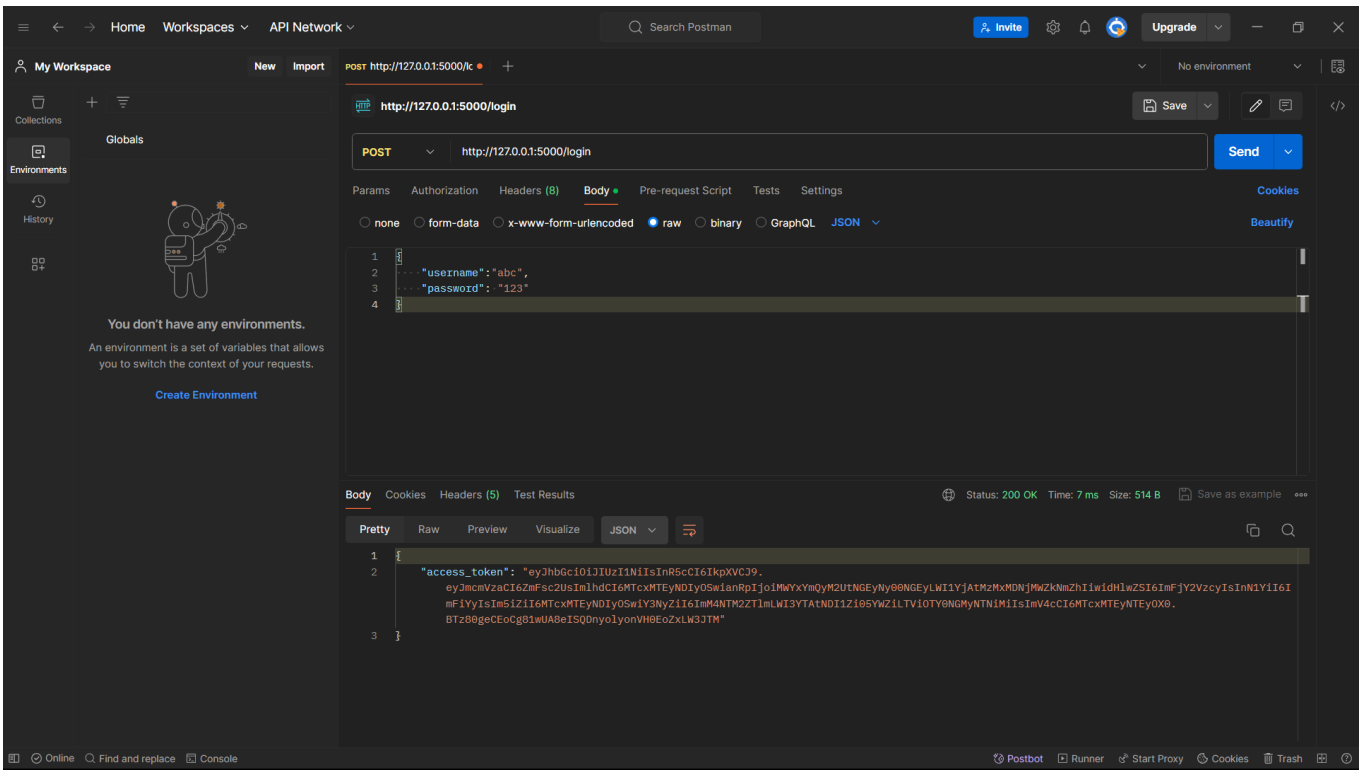
- Define your python app.

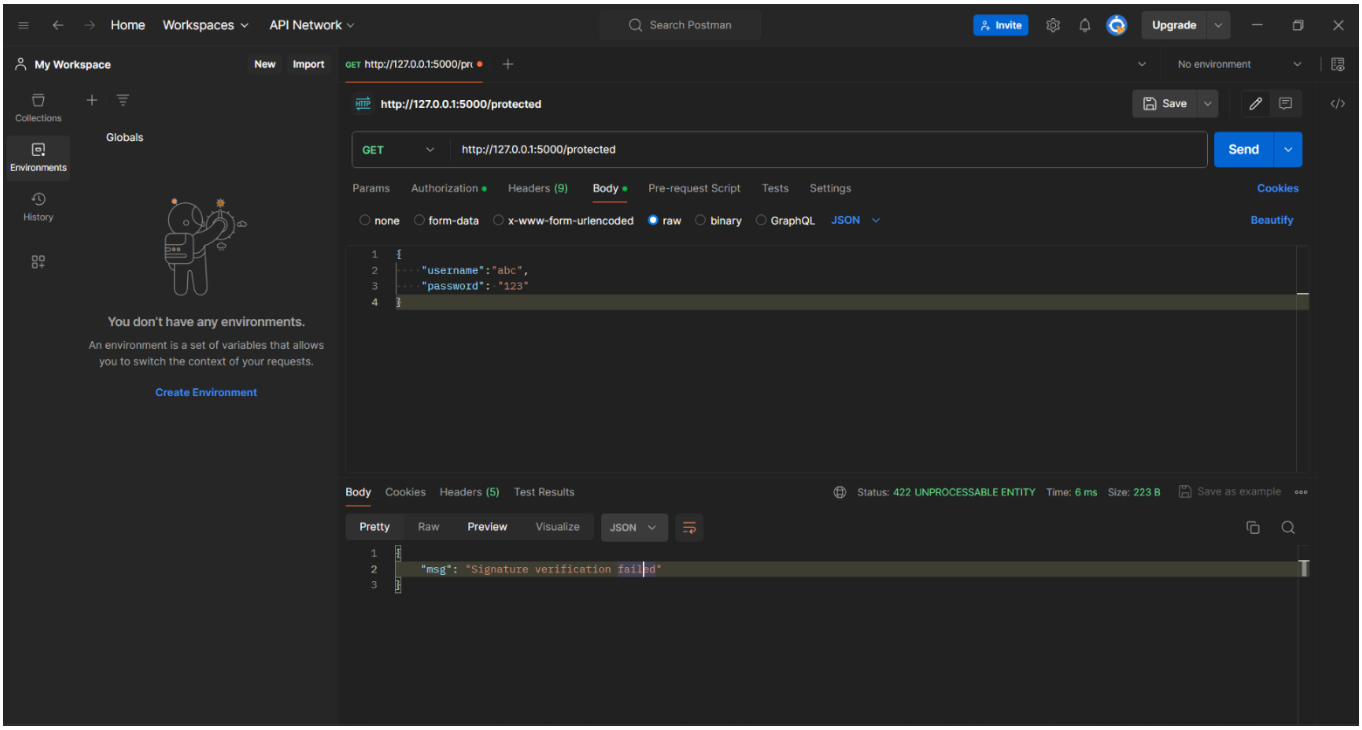
app.py

```
from flask import Flask, jsonify, request
from flask_jwt_extended import JWTManager, jwt_required,
create_access_token

app = Flask(__name__)

# Set up Flask-JWT-Extended
app.config["JWT_SECRET_KEY"] = "your-secret-key"  # Replace with
yoursecret key

jwt = JWTManager(app)
```

```
# Dummy user data (replace with a proper user database in a
realapplication)

users = {"abc": {"password": "123"}, "def": {"password": "456"}}

# Route to generate a JWT token upon login
@app.route("/login", methods=["POST"])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    if username in users and users[username]['password'] ==
password:
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token)
    else:
        return jsonify({'error': 'Invalid username or password'}),
401

# Protected route that requires a valid JWT token for access
@app.route('/protected', methods=['GET'])
@jwt_required()
def protected():
    current_user = jwt.get_jwt_identity()
    return jsonify(logged_in_as=current_user), 200

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

- **Expected Output:**





Result:

In conclusion, this Flask API offers a robust and scalable solution for user authentication. By leveraging JWTs, it ensures secure access to protected resources while eliminating the need for session management, ultimately enhancing the API's efficiency and security.

PRACTICAL- 4

❖ AIM: A program that creates a RESTful API that authenticates users using a JSON Web Token.

Problem Statement:

Develop a RESTful API that paginates the results of a query to improve performance and optimize resource usage. The API should allow clients to retrieve large datasets in smaller, manageable chunks by paginating the results and providing navigation controls to access subsequent pages.

Program Description:

When dealing with large datasets, returning all results in a single response can lead to performance issues, increased network traffic, and excessive resource consumption. Pagination is a common technique used to mitigate these issues by dividing the dataset into smaller pages and allowing clients to request and navigate through them incrementally. This project aims to design and implement a RESTful API that incorporates pagination to enhance performance and efficiency.

Procedure:

1) Dependencies:

- We'll use Flask, a lightweight web framework for Python, and the json library for working with JSON data. Install them using pip:

bash

```
pip install flask
```

2) Create a Python file and paste the code inside:

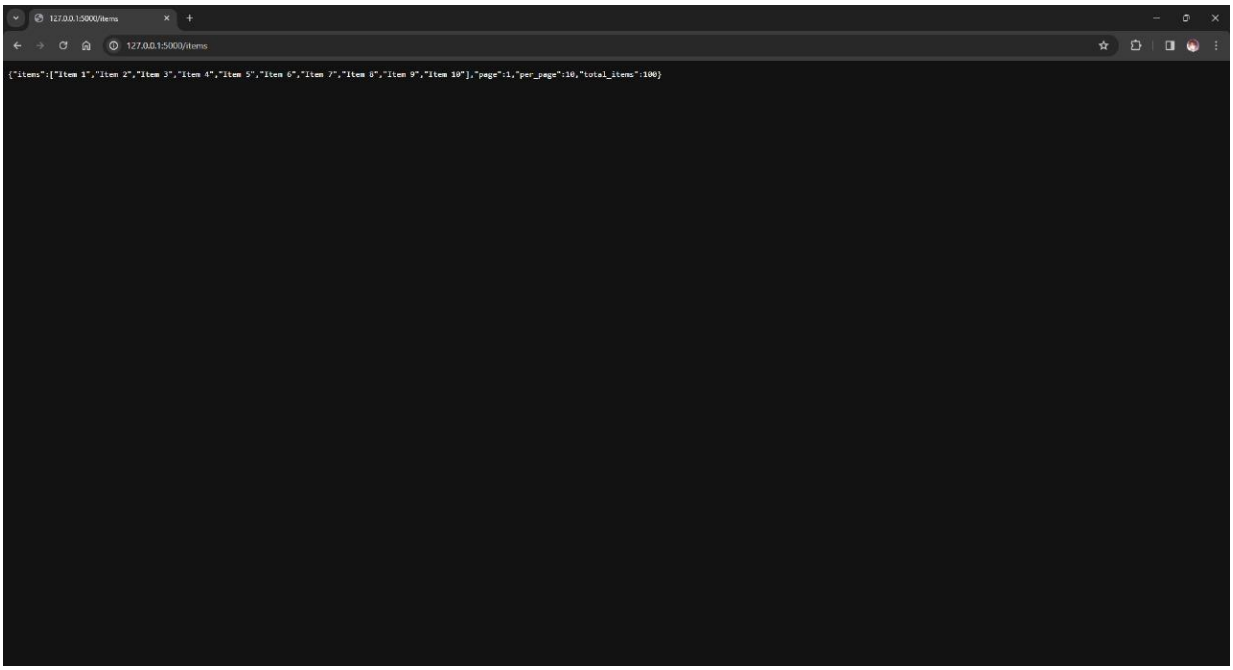
- Define your python app.

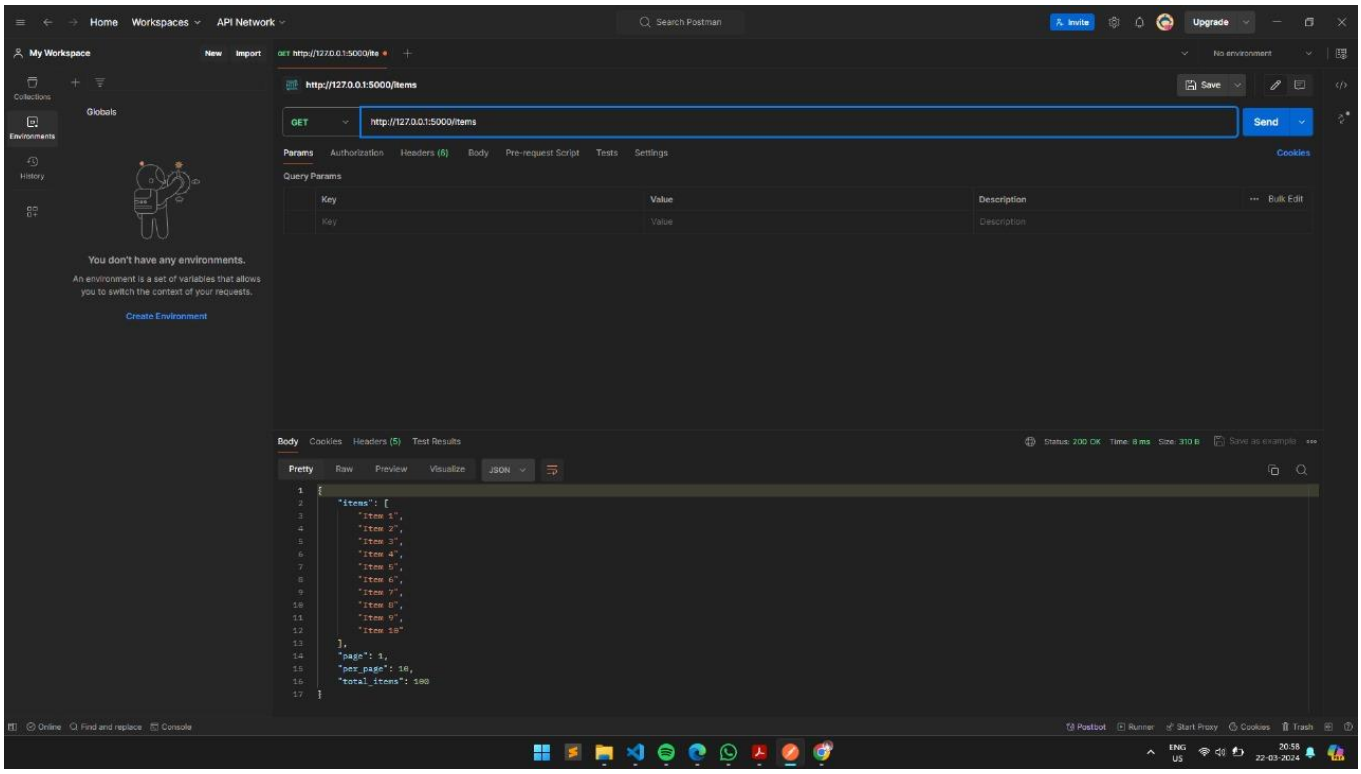
app.py

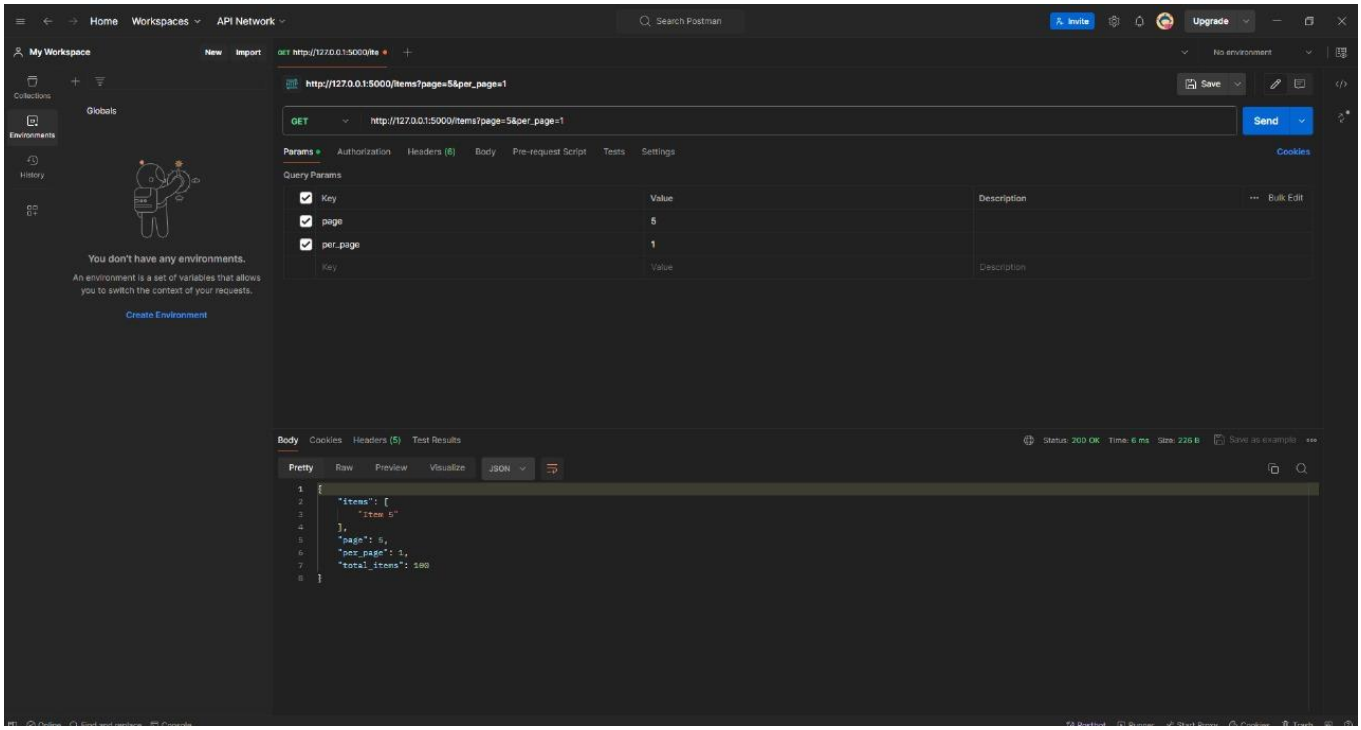
```
from flask import Flask, jsonify, request
from flask_jwt_extended import JWTManager, jwt_required,
create_access_token

app = Flask(__name__)

# Set up Flask-JWT-Extended
app.config["JWT_SECRET_KEY"] = ""  # Replace with yoursecret key

jwt = JWTManager(app)
```

```
# Dummy user data (replace with a proper user database in a
realapplication)

users = {"abc": {"password": "123"}, "def": {"password": "456"}}

# Route to generate a JWT token upon login
@app.route("/login", methods=["POST"])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')
    if username in users and users[username]['password'] ==
password:
        access_token = create_access_token(identity=username)
        return jsonify(access_token=access_token)
    else:
        return jsonify({'error': 'Invalid username or password'}),
401

# Protected route that requires a valid JWT token for access
@app.route('/protected', methods=['GET'])
@jwt_required()
def protected():
    current_user = jwt.get_jwt_identity()
    return jsonify(logged_in_as=current_user), 200

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

Expected Output:







Result:

This program successfully demonstrates the implementation of a secure and scalable RESTful API with user authentication using JSON Web Tokens (JWTs). The JWT approach provides stateless authentication, improved security through signed tokens, and flexibility for managing user access. This program serves as a valuable foundation for building robust and secure APIs with user authorization.

PRACTICAL- 5

❖ AIM: A program that creates a RESTful API that supports data validation and error handling.

Problem Statement:

Develop a RESTful API that supports data validation and error handling to ensure the integrity, consistency, and security of incoming requests and responses. The API should validate incoming data against specified constraints and provide informative error messages in case of validation failures or other errors.

Program Description:

Building a robust RESTful API involves not only defining endpoints and handling requests but also ensuring that the data being exchanged is valid and that errors are handled gracefully.

This project aims to design and implement a RESTful API that incorporates data validation and error handling mechanisms to enhance reliability and security.

Procedure:

2) Dependencies:

- We'll use Flask, a lightweight web framework for Python, and the json library for working with JSON data. Install them using pip:

bash

```
pip install flask Flask-RESTful
```

2) Create a Python file and paste the code inside:

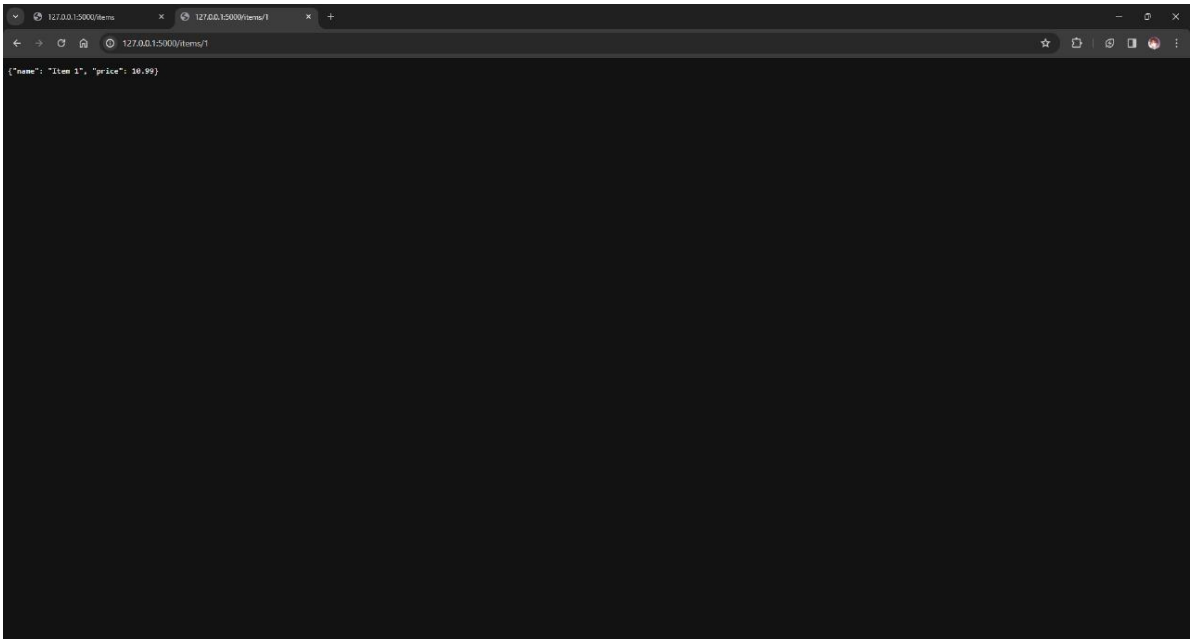
- Define your python app.

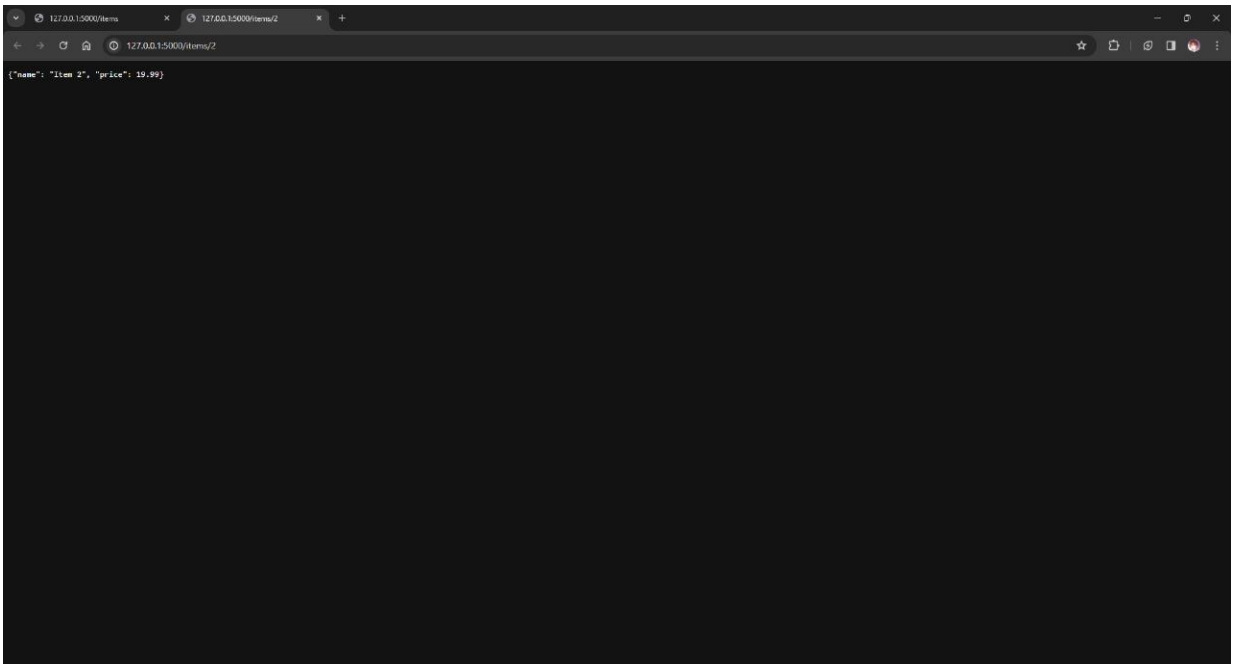
app.py

```
from flask import Flask
from flask_restful import Resource, Api, reqparse

app = Flask(__name__)
api = Api(app)

# Dummy data (replace with your actual data source)
items = {
    "1": {"name": "Item 1", "price": 10.99},
    "2": {"name": "Item 2", "price": 19.99},
```

```
}  # Request parser for input validation
parser = reqparse.RequestParser()
parser.add_argument("name", type=str, required=True, help="Name
cannot be blank")
parser.add_argument("price", type=float, required=True, help="Price
cannot be blank")

class ItemResource(Resource):
    def get(self, item_id) :
        item = items.get(item_id)
        if item:
            return item
        else:
            return {"error": "Item not found"}, 404

    def put(self, item_id):
        args = parser.parse_args()
        items[item_id] = {"name": args["name"], "price":
args["price"]}
        return items[item_id], 201

    def delete(self, item_id):
        if item_id in items:
            del items[item_id]
            return {"result": True}
        else:
            return {"error": "Item not found"}, 404

api.add_resource(ItemResource, "/items/<item_id>")

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

Expected Outcomes:





Result:

This program effectively establishes a robust RESTful API framework that prioritizes data integrity and user experience. By implementing data validation, it ensures that only valid data reaches the server, preventing potential errors and maintaining data consistency. Additionally, the error handling mechanism provides informative messages to users, facilitating debugging and improving overall API usability.