PRACTICAL -1

AIM: A program that creates a simple RESTful API that returns a 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.

Problem 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:

Step 1: Write the Flask App

Open **app.py** in a text editor and write the Flask application code:

app.py

```
from flask import Flask, jsonify
app = Flask(__name__)
users = [
{'id': 1, 'name': 'Arshad'},
{'id': 2, 'name': 'Vishnu'},
{'id': 3, 'name': 'Reddy'}
]
@app.route('/users', methods=['GET'])
def get_users():
      return jsonify(users)
if __name__== '__main__':
app.run(debug=True)
```

Output:

[{"id":1,"name":"John"},{"id":2,"name":"Jane"},{"id":3,"name":"Doe"}]

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.

Problem 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:

Step 1: Write the Flask App

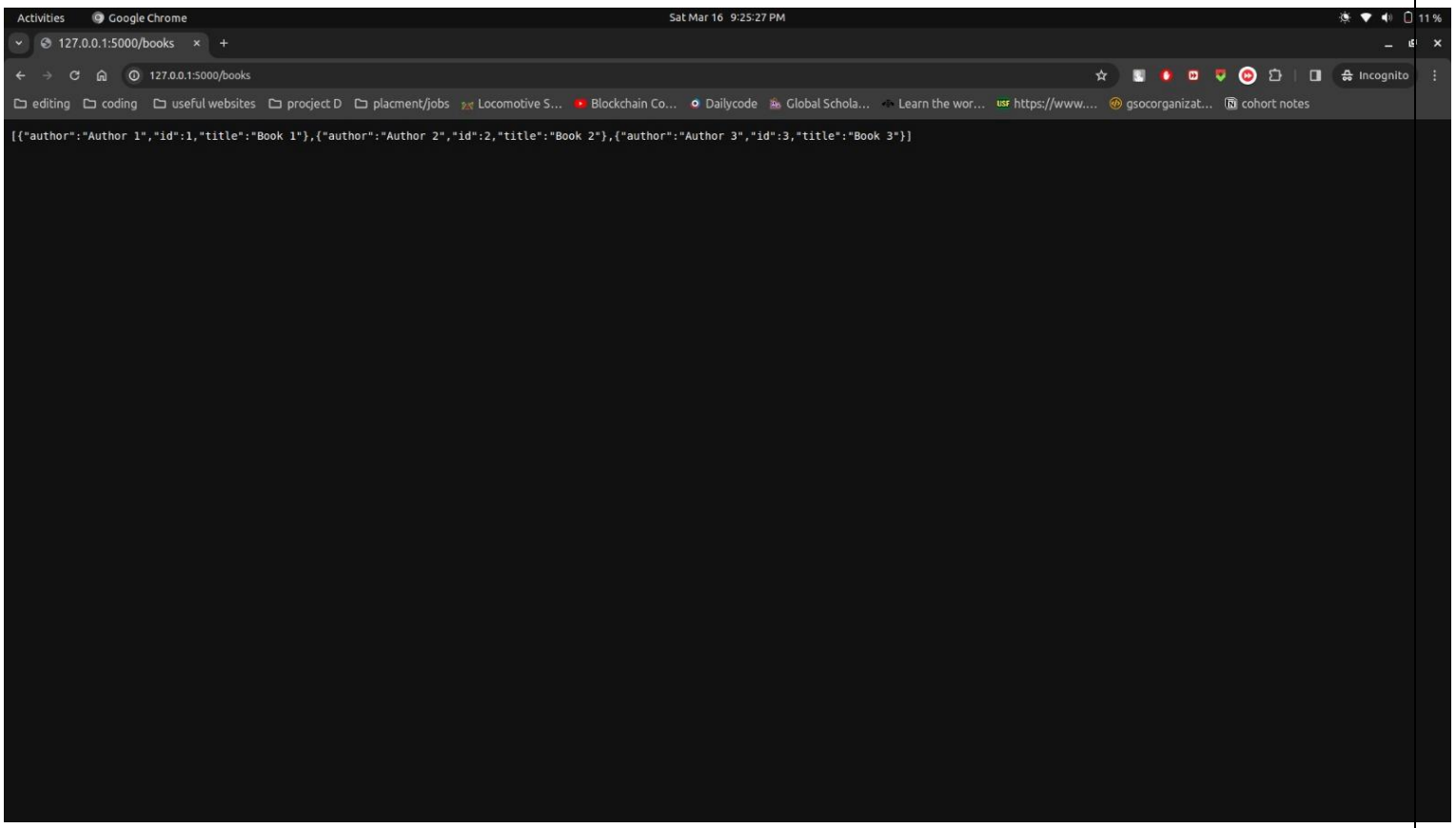
Open **app.py** in a text editor and write the Flask application code:

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)
```

Output:


Activities
Google Chrome
Sat Mar 16 9:25:27 PM
11 %
127.0.0.1:5000/books
127.0.0.1:5000/books
Incognito
editing
coding
useful websites
procject D
placment/jobs
Locomotive S...
Blockchain Co...
Dailycode
Global Schola...
Learn the wor...
https://www....
gsocorganizat...
cohort notes
[{"author":"Author 1","id":1,"title":"Book 1"},{"author":"Author 2","id":2,"title":"Book 2"},{"author":"Author 3","id":3,"title":"Book 3"}]

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.

Problem Description:

Debugging is an essential aspect of software development, especially in web applications where errors can occur due to various factors such as incorrect configurations, faulty code logic, or unexpected user inputs. Django, a high-level Python web framework, provides robust built-in debugging tools to assist developers in diagnosing and resolving issues efficiently.

Procedure:

Step 1: Write the Flask App

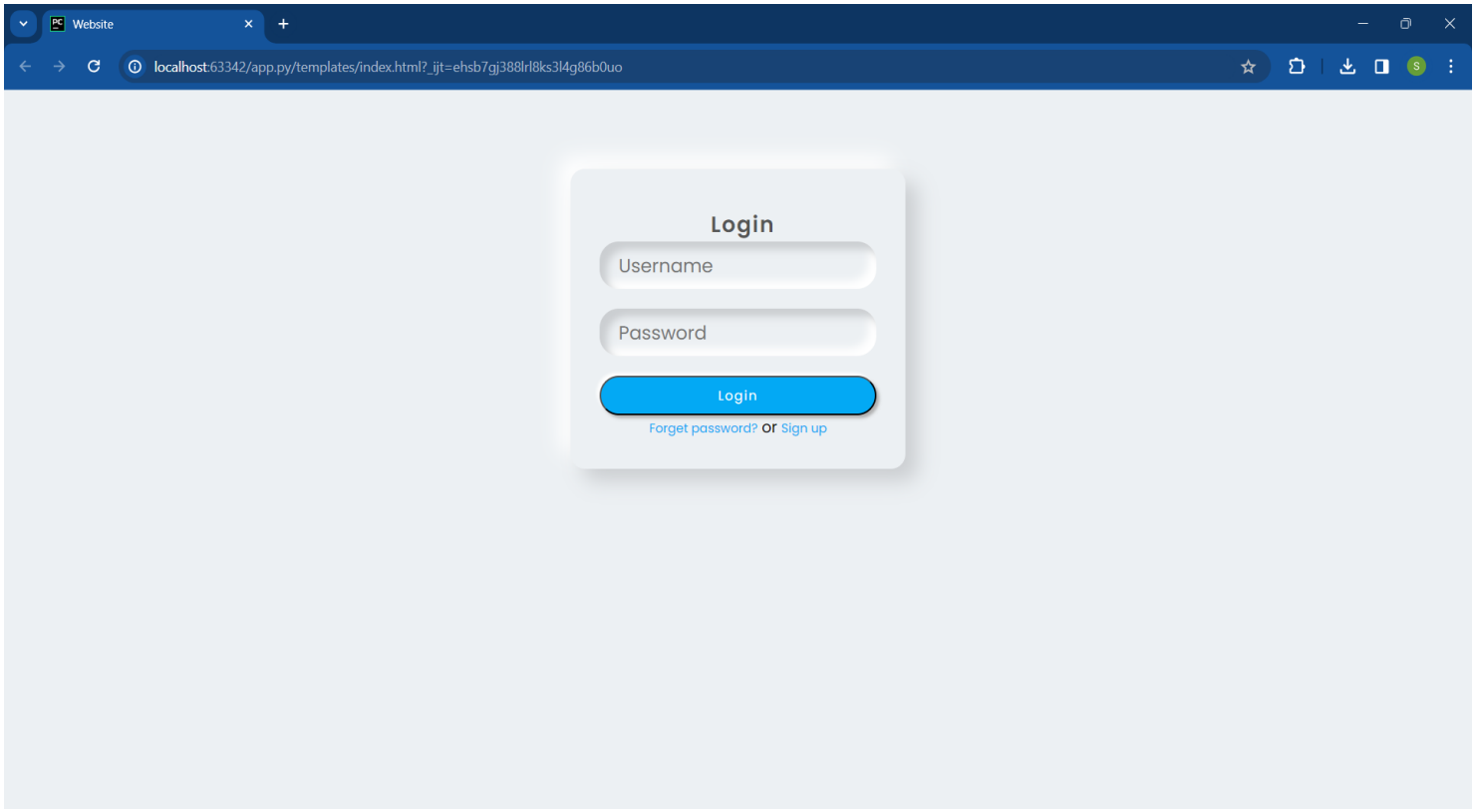
Open **app.py** in a text editor and write the Flask application code:

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 your
secret key
jwt = JWTManager(app)
# Dummy user data (replace with a proper user database in a real
application)
users = {
'user1': {'password': 'password1'},
'user2': {'password': 'password2'}
}
# 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)
```

Output:



PRACTICAL -4

AIM: A program that creates a RESTful API that paginates the results of a query to improve performance.

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.

Problem 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:

Step 1: Write the Flask App

Open **app.py** in a text editor and write the Flask application code:

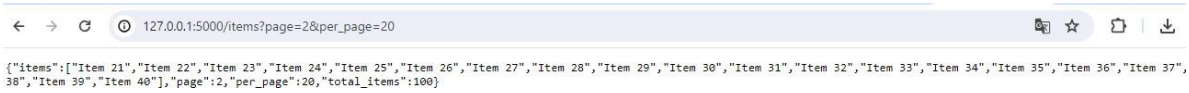
app.py

```
from flask import Flask, jsonify, request
app = Flask(__name__)
# Dummy data (replace with your actual data source)
items = [f'Item {i}' for i in range(1, 101)]
# Route that supports pagination
@app.route('/items', methods=['GET'])
def get_items():
page = int(request.args.get('page', 1))
per_page = int(request.args.get('per_page', 10))
start_index = (page - 1) * per_page
end_index = start_index + per_page
paginated_items = items[start_index:end_index]
return jsonify({'items': paginated_items, 'page': page,
'per_page': per_page, 'total_items': len(items)})
if __name__== '__main__':
```

Output:

{"items":["Item 1","Item 2","Item 3","Item 4","Item 5","Item 6","Item 7","Item 8","Item 9","Item 10"],"page":1,"per_page":10,"total_items":100}

Output



{"items":["Item 21","Item 22","Item 23","Item 24","Item 25","Item 26","Item 27","Item 28","Item 29","Item 30","Item 31","Item 32","Item 33","Item 34","Item 35","Item 36","Item 37",
38","Item 39","Item 40"],"page":2,"per_page":20,"total_items":100}

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.

Problem 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:

Step 1: Write the Flask App

Open app.py in a text editor and write the Flask application code:

app.py

```
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__':
```

Output