Theory Questions:- Restful API & Flask

Q1. What is a RESTful API?

Explanation:

A RESTful API (Representational State Transfer API) is a type of web service or interface that allows communication between systems over the internet using standard HTTP methods.

It is based on REST architecture principles, which emphasize simplicity, scalability, and stateless communication.

Key Features of RESTful API:

1. Stateless – Each request from the client to the server must contain all the information needed to process it; the server doesn't store session information.
2. Client-Server Architecture – Separation between the client (frontend) and the server (backend).
3. Uniform Interface – Uses standard HTTP methods like:

- GET → Retrieve data
- POST → Create new data
- PUT/PATCH → Update existing data
- DELETE → Remove data

4. Resource-Based – Everything is considered a resource (like users, products, or posts), identified by a URL/URI.

Example:

- GET /users → fetch all users
- GET /users/1 → fetch user with ID 1
- POST /users → create a new user

5. Representation – Resources can be represented in multiple formats (usually JSON, sometimes XML, or others).
6. Stateless & Cacheable – Responses can be cached to improve performance.

✅ Example: If you have a RESTful API for a blog:

- GET /posts ⟶ Get all blog posts
- POST /posts ⟶ Create a new post
- PUT /posts/5 ⟶ Update post with ID 5
- DELETE /posts/5 ⟶ Delete post with ID 5

Python Code Demo (using requests)

```
import requests

# Example: Fetching data from a public REST API (JSONPlaceholder)
url = "https://jsonplaceholder.typicode.com/posts/1"

response = requests.get(url)

# Print status code and JSON response
print("Status Code:", response.status_code)
print("Response JSON:", response.json())


# Output :
Status Code: 200
Response JSON: {
'userId': 1,
'id': 1,
'title': 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',
```

```
'body': 'quia et suscipit...'
}
```

Q2. Explain the concept of API Specification.

Explanation:

An API Specification is a detailed description of how an API works. It defines the rules, structure, and behavior that developers must follow to interact with the API.

It acts like a blueprint or contract between the API provider and the consumer, ensuring both sides know what to expect.

Key Points in an API Specification:

1. Endpoints (URLs/URIs) ⟶ Define where resources can be accessed. Example: /users, /orders/{id}
2. HTTP Methods ⟶ Define actions (GET, POST, PUT, DELETE, etc.).
3. Request Format ⟶ What data should be sent (headers, query params, body).
4. Response Format ⟶ What data will be returned (JSON, XML, status codes).
5. Authentication & Authorization ⟶ Security details (API keys, OAuth tokens).
6. Error Handling ⟶ Standard error codes and messages.

Example (OpenAPI/Swagger Style):

```
GET /users/{id}:
  description: Get details of a user by ID
 parameters:
   - name: id
     in: path
     required: true
     type: integer
 responses:
    200:
```

```
    description: User found
    schema:
      id: integer
       name: string
       email: string
  404:
    description: User not found
```

Here, the specification clearly defines:

- The endpoint (/users/{id})
- The HTTP method (GET)
- The input (path parameter id)
- The expected output (user details or an error).

Q3. What is Flask, and why is it popular for building APIs?

Explanation:

Flask is a lightweight, open-source Python web framework used for building web applications and APIs. It is based on the WSGI (Web Server Gateway Interface) standard and follows a minimalist design, meaning it provides the essentials to get started but allows developers to add extensions for extra features when needed.

Why Flask is Popular for Building APIs:

1. Simplicity & Minimalism – Flask is easy to learn and lets developers quickly set up APIs with just a few lines of code.
2. Flexibility – It doesn't force a specific project structure; developers have the freedom to design APIs as they like.
3. Built-in Development Server – Makes testing and debugging APIs straightforward.
4. Rich Ecosystem of Extensions – Support for authentication, databases, and serialization can be added as needed.
5. Integration with JSON – Easy to send and receive JSON data, which is widely used in RESTful APIs.
6. Scalability – Suitable for both small projects (prototypes) and production-level APIs.

Example: Simple Flask API

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/hello', methods=['GET'])
def hello():
    return jsonify({"message": "Hello, World!"})

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

👉 Running this will create an API endpoint /hello, and when accessed, it returns a JSON response:

```
{"message": "Hello, World!"}
```

✅ In summary: Flask is popular for APIs because it is lightweight, flexible, and beginner-friendly while still being powerful enough for production use.

Q4. What is Routing in Flask?

Explanation:

In Flask, Routing is the process of mapping a URL (endpoint) to a specific function in your application. When a client (like a browser or API consumer) sends a request to a URL, Flask uses the defined route to determine which function should handle that request.

How Routing Works

- You define routes using the @app.route() decorator.
- Each route is linked to a view function that returns a response (like HTML, JSON, or plain text).

- Routes can handle different HTTP methods (GET, POST, PUT, DELETE, etc.).

Example: Simple Routing

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"

@app.route('/about')
def about():
    return "This is the About Page."

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

👉 In this example:

- Visiting http://localhost:5000/ → shows "Welcome to the Home Page!"
- Visiting http://localhost:5000/about → shows "This is the About Page."

Example: Routing with Parameters

```
@app.route('/user/<name>')
def user(name):
    return f"Hello, {name}!"
```

👉 Visiting /user/Riya → returns "Hello, Riya!"

✅ In short: Routing in Flask is the mechanism that connects URLs to functions, allowing your app to respond to different requests with specific outputs.

Q5. How do you create a simple Flask application?

Explanation:

Creating a simple Flask application is very easy because Flask is lightweight and requires only a few lines of code to get started.

Steps to Create a Simple Flask App:

1. Install Flask

```
pip install flask
```

2. Write the Application Code

Create a file named app.py (or any name you like).

```
from flask import Flask

# Create a Flask application instance
app = Flask(__name__)

# Define a route
@app.route('/')
def home():
     return "Hello, Flask! This is my first app."

 # Run the application
 if __name__ == '__main__':
     app.run(debug=True)
```

3. Run the Application

   python app.py

4. Access in Browser

Go to → http://127.0.0.1:5000/ You'll see:

```
Hello, Flask! This is my first app.
```

✅ In short: To create a simple Flask app:

- Import Flask
- Create an instance of Flask
- Define routes using @app.route()
- Run the app with app.run()

Q6. What are HTTP Methods used in RESTful APIs?

Explanation:

In RESTful APIs, HTTP methods define the type of action to be performed on a resource (like data in a database). They follow the CRUD operations (Create, Read, Update, Delete).

Common HTTP Methods in RESTful APIs:

1. GET 🟢

- Used to retrieve data from the server.
- Does not change the state of the resource.
- Example:
  - GET /users → fetch all users
  - GET /users/1 → fetch user with ID 1

2. POST 🟠

- Used to create a new resource on the server.
- Example:
  - POST /users → create a new user (data sent in request body)

3. PUT 🔵

- Used to update/replace an existing resource completely.
- Example:
  - PUT /users/1 → update user with ID 1 (entire record replaced)

4. PATCH 🟣

- Used to partially update a resource (only some fields).
- Example:
  - PATCH /users/1 → update only specific fields of user 1

5. DELETE 🔴

- Used to remove a resource.
- Example:
  - DELETE /users/1 → delete user with ID 1

Extra (less common but useful):

- HEAD → Similar to GET but only retrieves headers, not the body.
- OPTIONS → Shows which HTTP methods are allowed for a resource.

✅ In short:

RESTful APIs commonly use GET, POST, PUT, PATCH, and DELETE to perform CRUD operations on resources.

Q7. What is the purpose of the @app.route() decorator in Flask?

Explanation:

In Flask, the @app.route() decorator is used to map a URL (route/endpoint) to a Python function (called a view function).

When a client (like a browser or API consumer) visits that URL, Flask runs the associated function and returns its response.

Purpose of @app.route():

1. Defines Routes – Connects a specific URL to a function.
2. Handles Requests – Ensures that when a request comes to that URL, the correct function executes.
3. Supports Dynamic URLs – Can accept parameters from the URL.
4. Specifies HTTP Methods – Can restrict routes to certain methods (GET, POST, etc.).

Example 1: Basic Route

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"
```

👉 Visiting http://127.0.0.1:5000/ runs home() and shows the message.

Example 2: Route with Methods

```
@app.route('/submit', methods=['POST'])
def submit():
     return "Form Submitted!"
```

👉 Only responds to POST /submit.

Example 3: Dynamic Route

```
@app.route('/user/<name>')
def user(name):
    return f"Hello, {name}!"
```

👉 Visiting /user/Riya → shows "Hello, Riya!"

✅ In short:

The @app.route() decorator in Flask tells the app which function should run when a specific URL is requested.

Q8. What is the difference between GET and POST HTTP methods?

Explanation:

🔹 Difference between GET and POST HTTP Methods

Feature	GET	POST
Purpose	Used to **retrieve** data from the server.	Used to **send data** to the server to create or update resources.
Data Location	Data is sent in the **URL (query string)**.	Data is sent in the **request body**.
Visibility	Data is **visible in the URL** (e.g., `/search?name=piyush`).	Data is **hidden** in the body, not shown in the URL.
Security	Less secure (data appears in logs, browser history).	More secure for sensitive data (like passwords), but still should use HTTPS.
Caching	Can be cached by the browser.	Not usually cached.
Idempotency	Safe and idempotent (repeating the request doesn't change data).	Not idempotent (repeating may create multiple entries).
Data Size Limit	Limited by URL length (around 2000 characters in many browsers).	No significant size limit (depends on server settings).
Usage Example	Searching, fetching user profile, viewing products.	Submitting forms, login, creating a new user, uploading files.

🔹 Example in REST API

- GET /users/1 → Fetch details of user with ID 1.
- POST /users → Create a new user (data like name, email sent in body).

✅ In short:

- GET → Request data (read-only, safe, visible in URL).

- POST ⟶ Send data (create/update resources, hidden in body).

Q9. How do you handle errors in Flask APIs?

Explanation:

In Flask APIs, errors can be handled gracefully using error handlers and try-except blocks. Proper error handling ensures that the API returns meaningful responses with the correct HTTP status codes instead of crashing.

1. Using abort()

Flask provides the abort() function to trigger an HTTP error.

```
from flask import Flask, abort, jsonify

app = Flask(__name__)

@app.route('/user/<int:id>')
def get_user(id):
    users = {1: "Piyush", 2: "Ankit"}
    if id not in users:
        abort(404)  # Not Found
    return jsonify({"id": id, "name": users[id]})
```

- Visiting /user/3 ⟶ returns 404 Not Found.

2. Custom Error Handlers

You can define custom responses for specific errors.

```
@app.errorhandler(404)
def not_found(error):
    return jsonify({"error": "Resource not found"}), 404
```

```
@app.errorhandler(500)
def server_error(error):
    return jsonify({"error": "Internal Server Error"}), 500
```

- Now, any 404 or 500 error returns a JSON response instead of the default HTML page.

3. Using try-except in Routes

For catching exceptions during request handling:

```
@app.route('/divide/<int:a>/<int:b>')
def divide(a, b):
    try:
        result = a / b
        return jsonify({"result": result})
    except ZeroDivisionError:
        return jsonify({"error": "Division by zero is not allowed"}), 400
```

✅ Summary

- abort() ⟶ Immediately returns an HTTP error.
- @app.errorhandler() ⟶ Customize error messages for specific status codes.
- try-except ⟶ Catch exceptions and return meaningful API responses.

Q10. How do you connect Flask to a SQL database?

Explanation:

To connect Flask to a SQL database, you typically use Flask extensions like Flask-SQLAlchemy (an ORM) or a direct database connector. This allows your Flask application to interact with relational databases such as SQLite, MySQL, or PostgreSQL.

Using Flask-SQLAlchemy (Recommended)

1. Install Required Packages

```
pip install flask flask_sqlalchemy
```

2. Configure Flask App

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'  # SQLite database
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

3. Define a Database Model

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
```

4. Create Database Tables

```
with app.app_context():
    db.create_all()
```

5. Perform CRUD Operations

```
 # Example: Add a new user
 @app.route('/add_user')
 def add_user():
     user = User(name="Riya", email="Riya@example.com")
     db.session.add(user)
```

```
    db.session.commit()
    return "User added successfully!"
```

Alternative: Direct Database Connection (SQLite Example)

```
import sqlite3
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/users')
def get_users():
    conn = sqlite3.connect('users.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()
    conn.close()
    return jsonify(rows)
```

✅ Summary

- Flask-SQLAlchemy is recommended for easier database management and ORM features.
- Direct database connectors can be used for simpler or lightweight applications.
- Steps: Configure DB → Define Models → Create Tables → Perform CRUD operations.

Q11. What is the role of Flask-SQLAlchemy?

Explanation:

Flask-SQLAlchemy is a Flask extension that integrates the SQLAlchemy ORM (Object Relational Mapper) with Flask, making it easier to work with relational databases in a Pythonic way.

Role of Flask-SQLAlchemy:

1. Simplifies Database Interaction

- Allows you to interact with databases using Python objects instead of writing raw SQL queries.
- Example: User.query.all() instead of SELECT * FROM users.

2. Defines Database Models Easily

- Lets you define tables as Python classes.
- Columns and relationships are defined as class attributes.

3. Manages Database Sessions

- Handles connections, commits, and rollbacks automatically.

4. Supports Multiple Databases

- Works with SQLite, MySQL, PostgreSQL, Oracle, etc., without changing code structure.

5. Integrates Seamlessly with Flask

- Works with Flask's app context and routing system.
- Makes CRUD operations straightforward.

Example: Using Flask-SQLAlchemy

```
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
db = SQLAlchemy(app)

class User(db.Model):
     id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
```

```
# Create tables
with app.app_context():
    db.create_all()

# Add a user
@app.route('/add_user')
def add_user():
    user = User(name="Riya", email="Riya@example.com")
    db.session.add(user)
    db.session.commit()
    return jsonify({"message": "User added successfully!"})
```

✅ In short:

Flask-SQLAlchemy makes it easy to define models, manage database sessions, and perform CRUD operations in Flask applications without writing raw SQL.

Q12. What are Flask Blueprints, and How Are They Useful?

Explanation:

In Flask, a Blueprint is a way to organize a Flask application into modular components. Instead of defining all routes, views, and logic in a single file, you can split your app into smaller, reusable parts, making it easier to maintain and scale.

Key Features of Flask Blueprints:

1. Modularity – Divide your app into separate components (e.g., auth, blog, api).
2. Reusability – Blueprints can be reused across different projects.
3. Organized Codebase – Keeps routes, templates, and static files organized.
4. Flexible Registration – You can register a blueprint with a URL prefix, making route management easier.

Example: Using Blueprints

1. Create a blueprint (auth.py):

```
from flask import Blueprint, jsonify

auth_bp = Blueprint('auth', __name__)

@auth_bp.route('/login')
def login():
    return jsonify({"message": "Login Page"})

@auth_bp.route('/register')
def register():
    return jsonify({"message": "Register Page"})
```

2. Register the blueprint in the main app (app.py):

```
from flask import Flask
 from auth import auth_bp

app = Flask(__name__)
app.register_blueprint(auth_bp, url_prefix='/auth')

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

3. Accessing Routes:

- /auth/login → Returns {"message": "Login Page"}
- /auth/register → Returns {"message": "Register Page"}

✅ Why Blueprints Are Useful

- Makes large applications manageable by splitting functionality.

- Enables code reuse across projects.
- Simplifies route organization and URL prefixes.
- Helps maintain clean, readable code for APIs and web apps.

Q13. What is the purpose of Flask's request object?

Explanation:

In Flask, the request object is used to access incoming request data sent by the client (like a browser or API consumer). It contains all the information about the HTTP request, including form data, query parameters, headers, JSON data, files, and more.

Purpose of Flask's request object:

1.Access Request Data

- Retrieve form fields, query parameters, and JSON payloads.

2.Inspect HTTP Method

- Check if the request is GET, POST, PUT, etc.

3.Access Headers and Cookies

- Read custom headers, user-agent, or cookies sent by the client.

4.Handle File Uploads

- Access files sent with the request.

Example Usage

```
from flask import Flask, request, jsonify

app = Flask(__name__)

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

```
def greet():
    if request.method == 'POST':
        data = request.get_json()  # Get JSON data from request body
        name = data.get('name', 'Guest')
    else:
        name = request.args.get('name', 'Guest')  # Get query param from URL

    return jsonify({"message": f"Hello, {name}!"})

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

Usage:

- `GET /greet?name=Riya → {"message": "Hello, Riya!"}`
- `POST /greet with JSON {"name": "Ankit"} → {"message": "Hello, Ankit!"}`

✅ In short:

The request object provides a centralized way to access all client-sent data in Flask, making it essential for handling forms, query parameters, JSON, headers, and files.

Q14. How do you create a RESTful API endpoint using Flask?

Explanation:

Creating a RESTful API endpoint in Flask involves defining a route that responds to HTTP methods (GET, POST, PUT, DELETE) and usually returns data in JSON format.

Steps to Create a RESTful API Endpoint

1. Install Flask

```
pip install flask
```

2. Create a Flask App

```
from flask import Flask, jsonify, request

app = Flask(__name__)
```

3. Define a Resource Endpoint

Here's an example for managing a simple users resource:

```
users = [
    {"id": 1, "name": "Piyush"},
    {"id": 2, "name": "Ankit"}
]

# GET all users
@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)

# GET a specific user by ID
@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if user:
        return jsonify(user)
        return jsonify({"error": "User not found"}), 404

# POST a new user
@app.route('/users', methods=['POST'])
def create_user():
```

```
    data = request.get_json()
    new_user = {"id": len(users) + 1, "name": data["name"]}
    users.append(new_user)
    return jsonify(new_user), 201
```

4. Run the Flask App

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

How It Works

- GET /users ⟶ Returns all users.
- GET /users/1 ⟶ Returns the user with ID 1.
- POST /users ⟶ Adds a new user with JSON data in the request body.

✅ Summary

To create a RESTful API endpoint in Flask:

1. Import Flask, request, and jsonify.
2. Define routes using @app.route() with HTTP methods.
3. Use request to get input data and jsonify to return JSON responses.

Q15. What is the purpose of Flask's jsonify() function?

Explanation:

In Flask, the jsonify() function is used to convert Python data structures (like dictionaries or lists) into a JSON-formatted HTTP response. This is essential when building APIs, as JSON is the standard format for exchanging data between the server and client.

Purpose of jsonify():

1. Converts Python objects to JSON – Automatically transforms dictionaries, lists, or other serializable objects into JSON.
2. Sets the Correct MIME Type – Returns a response with Content-Type: application/json, which tells the client that the response is JSON.
3. Handles Status Codes – You can optionally provide an HTTP status code along with the JSON response.
4. Simplifies API Responses – Makes it easier to return structured data without manually converting it using json.dumps().

Example:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/user')
def get_user():
    user = {"id": 1, "name": "Riya", "email": "Riya@example.com"}
    return jsonify(user)  # Converts Python dict to JSON response

@app.route('/users')
def get_users():
    users = [
        {"id": 1, "name": "Riya"},
        {"id": 2, "name": "Ankit"}
   ]
    return jsonify(users), 200  # Can also specify HTTP status code
```

Output for /user:

```
{
  "id": 1,
  "name": "Riya",
```

```
  "email": "Riya@example.com"
}
```

✅ In short:

jsonify() is used in Flask to return data in JSON format with proper headers, making it the standard way to send API responses.

Q16. Explain Flask's url_for() function.

Explanation:

In Flask, the url_for() function is used to dynamically generate URLs for routes defined in your application. Instead of hardcoding URLs, url_for() ensures that links remain correct even if the route changes.

Purpose of url_for():

1.Dynamic URL Generation – Automatically generates the correct URL for a given endpoint (function name).

2.Avoid Hardcoding URLs – Makes your app more maintainable and less error-prone.

3.Supports Variable Routes – Can pass arguments for dynamic parts of the URL.

4.Useful in Templates – Often used in HTML templates for linking between pages.

Basic Syntax

```
url_for(endpoint, **values)
```

- endpoint ⟶ The name of the view function.
- values ⟶ Any arguments for dynamic parts of the route.

Example 1: Simple Route

```
from flask import Flask, url_for
```

```
app = Flask(__name__)

@app.route('/')
def home():
    return "Home Page"

@app.route('/about')
def about():
    return "About Page"

with app.test_request_context():
    print(url_for('home'))   # Output: '/'
    print(url_for('about'))  # Output: '/about'
```

Example 2: Route with Parameters

```
@app.route('/user/<username>')
def profile(username):
    return f"User: {username}"

with app.test_request_context():
    print(url_for('profile', username='Riya'))  # Output: '/user/Riya'
```

Example 3: Using in Templates

```
<a href="{{ url_for('about') }}">About Us</a>
<a href="{{ url_for('profile', username='Ankit') }}">Profile</a>
```

✅ In short:

Flask's url_for() function creates URLs dynamically based on route names, making your app more flexible, maintainable, and less prone to errors.

Q17. How does Flask handle static files (CSS, JavaScript, etc.)?

Explanation:

In Flask, static files like CSS, JavaScript, images, and other assets are served from a special folder called static. Flask provides an easy way to organize and access these files without manually handling routes for them.

Key Points about Static Files in Flask:

1.Default Folder – By default, Flask looks for static files in a folder named static inside your project directory.

2.Accessing Files – Static files are accessed via the /static/ URL path.

- Example: /static/style.css or /static/script.js.

3.url_for() for Static Files – Use url_for('static', filename='path/to/file') to generate URLs dynamically.

4.No Special Routes Needed – Flask automatically serves files from the static folder.

Project Structure Example

```
my_flask_app/
│
├── app.py
├── static/
│   ├── style.css
│   └── script.js
└── templates/
    └── index.html
```

Example: Using Static Files in Templates:

```
<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
```

```
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <script src="{{ url_for('static', filename='script.js') }}"></script>
</head>
<body>
    <h1>Welcome to Flask!</h1>
</body>
</html>
```

Accessing Directly in Browser:

- CSS: http://127.0.0.1:5000/static/style.css
- JS: http://127.0.0.1:5000/static/script.js

✅ In short:

Flask handles static files by serving them from a static folder, accessible via /static/ URLs, and you can use url_for('static', filename=...) to generate links dynamically in templates.

Q18. What is an API specification, and how does it help in building a Flask API?

Explanation:

An API specification is a formal document or blueprint that defines how an API behaves—what endpoints it provides, what data it accepts and returns, what HTTP methods are used, authentication requirements, and error responses. Essentially, it serves as a contract between the API developer and the API consumer.

How an API Specification Helps in Building a Flask API

1. Clear Structure

Defines routes, methods, and request/response formats in advance, making development more organized.

2. Consistency

Ensures all endpoints follow the same conventions, data formats (usually JSON), and error handling.

3. Easier Collaboration

Frontend developers, QA, and other teams can understand and test the API without waiting for backend implementation.

4. Documentation & Testing

Tools like OpenAPI (Swagger) can generate interactive docs directly from the specification.

Makes it easier to write automated tests and validate API responses.

5. Speeds Up Development

- Developers know exactly what to implement, reducing guesswork and errors.

Example: OpenAPI (Swagger) Specification for Flask Endpoint

```
paths:
  /users/{id}:
    get:
      summary: Get user details
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: User found
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                  name:
```

```
                type: string
      '404':
        description: User not found
```

In Flask, you would implement this as:

```
@app.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    # Implementation matching the API spec
    ...
```

✅ In short:

An API specification serves as a roadmap for building a Flask API. It ensures clarity, consistency, and faster development, while also enabling better collaboration, documentation, and testing.

Q19. What are HTTP status codes, and why are they important in a Flask API?

Explanation:

HTTP status codes are standardized codes sent by a server in response to a client's HTTP request. They indicate whether a request was successful, failed, or requires further action.

In a Flask API, status codes are crucial because they communicate the result of an API request clearly to the client (frontend, mobile app, or another service).

Categories of HTTP Status Codes

Code Range	Category	Meaning	Example in Flask
1xx	Informational	Request received, processing	Rarely used in APIs
2xx	Success	Request was successful	`200 OK`, `201 Created`
3xx	Redirection	Client needs to take additional action	`301 Moved Permanently`
4xx	Client Error	Request had a problem	`400 Bad Request`, `404 Not Found`
5xx	Server Error	Server failed to process request	`500 Internal Server Error`

Example in Flask:

```
from flask import Flask, jsonify

app = Flask(__name__)

users = [{"id": 1, "name": "Piyush"}]

@app.route('/users/<int:user_id>')
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if user:
        return jsonify(user), 200          # Success
    else:
        return jsonify({"error": "User not found"}), 404  # Client Error
```

Explanation:

- 200 OK ⟶ The request succeeded and data is returned.
- 404 Not Found ⟶ The requested resource doesn't exist.

✅ Why Status Codes Are Important:

1. Communicate Outcome – Clients can understand if the request was successful or failed.
2. Error Handling – Helps frontend or API consumers handle errors properly.
3. Standardization – Ensures APIs follow web standards, improving interoperability.
4. Debugging & Logging – Makes it easier to identify issues in development and production.

Q20. How do you handle POST requests in Flask?

Explanation:

In Flask, handling POST requests involves:

1. Defining a route that accepts the POST method.
2. Accessing the data sent by the client in the request body (usually JSON or form data).
3. Processing the data and returning a response, often in JSON format.

Step-by-Step Example:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Sample in-memory storage
users = []

# POST endpoint to add a new user
@app.route('/users', methods=['POST'])
def add_user():
    data = request.get_json()  # Get JSON data from request body
    if not data or 'name' not in data:
        return jsonify({"error": "Name is required"}), 400  # Bad Request

    new_user = {
        "id": len(users) + 1,
        "name": data['name']
    }
    users.append(new_user)
     return jsonify(new_user), 201  # Created

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

How It Works

1. Client sends a POST request to /users with JSON data:

```
{
  "name": "Riya"
}
```

2. Flask reads the data using request.get_json().
3. The server processes the data, adds it to the users list, and returns a JSON response with status code 201 Created.

✅ Key Points

- Use methods=['POST'] in @app.route() to accept POST requests.
- Use request.get_json() for JSON data or request.form for form data.
- Return meaningful HTTP status codes (201 Created, 400 Bad Request, etc.).

Q21. How would you secure a Flask API?

Explanation:

Securing a Flask API is crucial to protect sensitive data, prevent unauthorized access, and ensure the integrity of your application. Security involves authentication, authorization, data protection, and request validation.

1. Authentication & Authorization

- API Keys: Simple method to restrict access to trusted clients.

```
from flask import request, abort

API_KEY = "mysecretkey"

@app.before_request
def check_api_key():
```

```
    if request.headers.get('x-api-key') != API_KEY:
        abort(401)  # Unauthorized
```

- JWT (JSON Web Tokens): Secure token-based authentication.
    - Server generates a token after login; client sends it in headers (Authorization: Bearer).
- OAuth2: Standard for third-party integrations (e.g., Google, Facebook login).

2. Input Validation

- Validate incoming JSON, query parameters, and form data to prevent injection attacks.
- Use libraries like Marshmallow or pydantic to enforce schema validation.

```
from flask import request, jsonify
from marshmallow import Schema, fields, ValidationError

class UserSchema(Schema):
    name = fields.Str(required=True)
    email = fields.Email(required=True)

@app.route('/users', methods=['POST'])
def add_user():
     try:
        data = UserSchema().load(request.get_json())
    except ValidationError as err:
        return jsonify(err.messages), 400
    # proceed to save data
```

3. HTTPS (TLS/SSL)

- Always serve your API over HTTPS to encrypt data in transit and prevent man-in-the-middle attacks.

4. Rate Limiting

- Prevent abuse by limiting the number of requests a client can make.
- Use Flask-Limiter:

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(app, key_func=get_remote_address)
@app.route("/users")
@limiter.limit("5 per minute")
def get_users():
    return jsonify(users)
```

5. CORS (Cross-Origin Resource Sharing)

- Control which domains can access your API.
- Use Flask-CORS to allow only trusted origins.

```
from flask_cors import CORS
CORS(app, origins=["https://myfrontend.com"])
```

6. Secure Headers

- Protect against common attacks by setting headers like Content-Security-Policy, X-Frame-Options, X-XSS-Protection.
- Use Flask-Talisman:

```
from flask_talisman import Talisman
Talisman(app)
```

✅ Summary

To secure a Flask API:

1. Implement authentication & authorization (API keys, JWT, OAuth2).
2. Validate all incoming data.
3. Use HTTPS for encrypted communication.
4. Apply rate limiting to prevent abuse.
5. Control CORS for trusted domains.
6. Add secure headers to prevent attacks.

Q22. What is the significance of the Flask-RESTful extension?

Explanation:

Flask-RESTful is an extension for Flask that simplifies the process of building RESTful APIs. It provides tools and abstractions that make creating, organizing, and managing API endpoints much easier than using plain Flask alone.

Significance of Flask-RESTful

1. Simplifies API Development
- Provides a Resource class to define endpoints as Python classes.
- Supports HTTP methods (GET, POST, PUT, DELETE) as class methods.
2. Automatic Request Parsing
- Includes reqparse to parse and validate request data easily.
3. Structured Responses
- Makes it easy to return JSON responses with proper HTTP status codes.
4. Organizes Code Better
- Resources can be grouped, making the API modular and maintainable.
5. Integration with Flask
- Works seamlessly with existing Flask apps, routes, and extensions.

Example: Flask-RESTful API

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

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

users = []

# Request parser
parser = reqparse.RequestParser()
parser.add_argument('name', type=str, required=True, help="Name is   required")

# Resource class
class User(Resource):
    def get(self):
        return users, 200

     def post(self):
        args = parser.parse_args()
        user = {"id": len(users) + 1, "name": args['name']}
        users.append(user)
        return user, 201

# Adding resource to API
api.add_resource(User, '/users')

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

Usage:

- GET /users ⟶ Get all users
- POST /users ⟶ Add a new user with JSON data

✅ In short:

Flask-RESTful makes building REST APIs simpler, more organized, and maintainable by providing abstractions like Resource classes, request parsing, and structured responses, reducing boilerplate code.

Q23. What is the role of Flask's session object.

Explanation:

In Flask, the session object is used to store data that is specific to a user across multiple requests. It provides a way to keep track of user information, like login status or preferences, without having to pass data manually between pages or API calls.

Key Features of Flask's session Object:

1. Persistent Across Requests
- Data stored in session is available for the same user across multiple requests.
2. Server-Side Security (Signed Cookies)
- Flask stores session data in client-side cookies but signs them cryptographically to prevent tampering.
- Requires app.secret_key to sign session cookies.
3. Stores Small Amounts of Data
- Typically used for lightweight information like user IDs, roles, or flash messages.

Example Usage

```
from flask import Flask, session, redirect, url_for, request

app = Flask(__name__)
app.secret_key = 'supersecretkey'  # Required for session
```

```
@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    # Store username in session
    session['username'] = username
    return f"Logged in as {username}"

@app.route('/profile')
def profile():
    if 'username' in session:
        return f"Welcome, {session['username']}!"
    else:
        return "You are not logged in.", 401

@app.route('/logout')
def logout():
    session.pop('username', None)  # Remove username from session
    return "Logged out!"
```

How it Works:

- User logs in → session['username'] is set
- On /profile, Flask retrieves the username from the session
- User logs out → session data is cleared

✅ In short:

The session object in Flask stores per-user data across requests, enabling features like login sessions, user preferences, or temporary data storage, while keeping it secure using signed cookies.

Practical Question :-

Q1. How do you create a basic Flask application?

Explanation:

1. Install Flask

First, make sure Flask is installed:

```
pip install flask
```

2. Create the Flask App

Create a file called app.py:

```
from flask import Flask

# Step 1: Create a Flask instance
app = Flask(__name__)

# Step 2: Define a route
@app.route('/')
def home():
    return "Hello, Flask! This is my first app."

# Step 3: Run the app
if __name__ == '__main__':
    app.run(debug=True)
```

3. Run the Application

```
python app.py
```

- By default, the app runs at http://127.0.0.1:5000/.

- Visiting this URL in a browser shows:

```
Hello, Flask! This is my first app.
```

4. Explanation of the Code

- Flask(**name**) → Creates the Flask application.
- @app.route('/') → Defines the URL route / and maps it to the home function.
- app.run(debug=True) → Starts the server in debug mode, which reloads automatically on code changes.

✅ Summary

- Install Flask → pip install flask
- Create app.py with a Flask instance
- Define routes using @app.route()
- Run the app with app.run()

Q2. How do you serve static files like images or CSS in Flask?

Explanation:

In Flask, static files like images, CSS, JavaScript, or fonts are served from a special folder called static. Flask provides a simple way to organize and access these files without manually creating routes.

1. Project Structure Example

```
my_flask_app/
│
├── app.py
├── static/
│   ├── style.css
│   └── logo.png
```

```
└── templates/
    └── index.html
```

2. Accessing Static Files

- Flask automatically serves files from the static folder at the /static/ URL.
    - CSS: /static/style.css
    - Image: /static/logo.png

3. Using url_for in Templates

Use url_for('static', filename='...') to generate URLs dynamically:

```
<!-- templates/index.html -->
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
     <h1>Welcome to Flask!</h1>
    <img src="{{ url_for('static', filename='logo.png') }}" alt="Logo">
</body>
</html>
```

4. Example Flask App

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
```

```
    return render_template('index.html')

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

- Visiting / in the browser will load the HTML template with CSS and images served from the static folder.

✅ Summary

- Place all static files in a folder named static.
- Access them via /static/filename or use url_for('static', filename='...') in templates.
- No extra routes are needed; Flask serves them automatically.

Q3. How do you define different routes with different HTTP methods in Flask?

Explanation:

In Flask, you can define routes that respond to specific HTTP methods (like GET, POST, PUT, DELETE) using the methods parameter in the @app.route() decorator. By default, a route only responds to GET requests.

1. Basic Route (GET only)

```
from flask import Flask

app = Flask(__name__)

  @app.route('/')
def home():
    return "Welcome to Flask!"
```

- This route only handles GET requests.

2. Route with Multiple HTTP Methods

You can specify which methods the route should accept:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/user', methods=['GET', 'POST'])
def user():
    if request.method == 'GET':
       # Handle GET request
        return jsonify({"message": "Fetching user data"})
    elif request.method == 'POST':
        # Handle POST request
        data = request.get_json()
         return jsonify({"message": "User created", "data": data}), 201

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

3. Explanation

- methods=['GET', 'POST'] → Specifies which HTTP methods the route can handle.
- request.method → Lets you check which method the client used.
- Each method can have its own logic within the same route.

4. Separate Routes for Different Methods (Optional)

You can also define different functions for the same URL but different methods using @app.route() separately:

```
@app.route('/user', methods=['GET'])
def get_user():
    return "Fetching user data"

@app.route('/user', methods=['POST'])
```

```
def create_user():
    return "Creating new user"
```

✅ Summary

- Use the methods parameter in @app.route() to specify allowed HTTP methods.
- Use request.method inside the function to handle different logic.
- You can also define separate functions for the same URL with different methods.

Q4. How do you render HTML templates in Flask?

Explanation:

In Flask, you can render HTML templates using the render_template() function. This allows you to separate your HTML code from Python logic, making your application more organized and maintainable.

1. Project Structure Example

```
my_flask_app/
│
├── app.py
├── templates/
│   └── index.html
└── static/
     └── style.css
```

- Flask looks for templates inside the templates folder by default.

2. Basic Template Rendering

app.py

```
from flask import Flask, render_template
```

```
app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')  # Render HTML template

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

templates/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>My Flask App</title>
</head>
<body>
<h1>Welcome to Flask!</h1>
</body>
</html>
```

- Visiting / in the browser will display the HTML content from index.html.

3. Passing Data to Templates

You can pass variables from Python to the template using render_template():

app.py

```
@app.route('/user/<name>')
def user(name):
    return render_template('user.html', username=name)
```

```
templates/user.html

<!DOCTYPE html>
<html>
<head>
    <title>User Page</title>
</head>
<body>
    <h1>Hello, {{ username }}!</h1>
</body>
</html>
```

- Visiting /user/Riya will display:

```
Hello, Riya!
```

✅ Summary

- Use render_template('filename.html', var=value) to render HTML pages.
- Templates go inside the templates folder.
- You can pass variables to templates using render_template.
- This separates HTML presentation from Python logic, making your app cleaner.

Q5. How can you generate URLs for routes in Flask using url_for?

Explanation:

In Flask, the url_for() function is used to dynamically generate URLs for routes instead of hardcoding them. This makes your application more maintainable because if a route changes, you don't have to update every link manually.

1. Basic Usage of url_for

```
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def home():
    return "Home Page"

@app.route('/about')
def about():
    return "About Page"

with app.test_request_context():
    print(url_for('home'))   # Output: '/'
    print(url_for('about'))  # Output: '/about'
```

- Argument: the name of the view function (home, about).
- Returns the URL path corresponding to that function.

2. Using url_for with Route Parameters

If your route has dynamic segments, you can pass values as keyword arguments:

```
@app.route('/user/<username>')
def profile(username):
    return f"User: {username}"

with app.test_request_context():
    print(url_for('profile', username='Piyush'))  # Output: '/user/Riya'
```

3. Using url_for in Templates

```
<a href="{{ url_for('home') }}">Home</a>
<a href="{{ url_for('profile', username='Ankit') }}">Profile</a>
```

- Automatically generates correct URLs for links in HTML templates.

4. Optional Query Parameters

You can also include query parameters in url_for:

```
print(url_for('profile', username='Piyush', page=2))
# Output: '/user/Piyush?page=2'
```

✅ Summary

- url_for() dynamically generates URLs based on view function names.
- Works with static routes, dynamic routes, and query parameters.
- Makes your app more maintainable and avoids hardcoding URLs.

Q6. How do you handle forms in Flask?

Explanation:

In Flask, handling forms involves receiving data submitted by the user, usually via HTML forms, and processing it on the server. Flask makes this simple with the request object.

1. Project Structure Example

```
my_flask_app/
│
├── app.py
└── templates/
    └── form.html
```

2. HTML Form Template

templates/form.html

```
<!DOCTYPE html>
<html>
<head>
     <title>Flask Form</title>
</head>
<body>
    <h1>Enter Your Name</h1>
    <form action="/submit" method="POST">
        <input type="text" name="username" placeholder="Your Name">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

3. Flask App to Handle the Form

app.py

```
from flask import Flask, render_template, request

app = Flask(__name__)

# Display the form
@app.route('/')
def index():
    return render_template('form.html')

# Handle form submission
@app.route('/submit', methods=['POST'])
 def submit():
     username = request.form.get('username')  # Get form data
```

```
    if username:
        return f"Hello, {username}!"
    else:
        return "Please enter a name!", 400

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

4. How it Works

- User visits / → sees the HTML form.
- User fills the form and clicks Submit → sends a POST request to /submit.
- Flask reads the data using request.form.get('field_name').
- Server processes the data and returns a response.

✅ Key Points

- Use method="POST" in the HTML form for sending data.
- Access submitted data in Flask with request.form (for form data) or request.get_json() (for JSON).
- Always validate input to avoid empty or malicious data.

Q7. How can you validate form data in Flask?

Explanation:

In Flask, validating form data is important to ensure that the user submits correct and safe information. You can do this manually using Python code, or use Flask extensions like Flask-WTF, which provide built-in validation and CSRF protection.

1. Manual Validation Using request.form

```
from flask import Flask, request, render_template, jsonify
```

```
app = Flask(__name__)

@app.route('/')
def index():
    return render_template('form.html')

@app.route('/submit', methods=['POST'])
def submit():
   username = request.form.get('username')
    email = request.form.get('email')

    # Simple validation
    errors = []
    if not username:
        errors.append("Username is required.")
    if not email or "@" not in email:
        errors.append("Valid email is required.")

    if errors:
         return jsonify({"errors": errors}), 400
    return f"Hello {username}, your email is {email}!"

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

Explanation:

- Check if fields are empty or incorrectly formatted.
- Return errors with HTTP 400 Bad Request if validation fails.

2. Using Flask-WTF for Advanced Validation

Step 1: Install Flask-WTF

```
pip install flask-wtf
```

Step 2: Create a Form Class

```
from flask import Flask, render_template
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import DataRequired, Email

app = Flask(__name__)
app.secret_key = 'supersecretkey'  # Required for CSRF protection

class UserForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
    submit = SubmitField('Submit')
```

Step 3: Use the Form in Routes

```
@app.route('/', methods=['GET', 'POST'])
def index():
    form = UserForm()
    if form.validate_on_submit():
        username = form.username.data
        email = form.email.data
        return f"Hello {username}, your email is {email}!"
    return render_template('form_wtf.html', form=form)
```

Step 4: HTML Template (form_wtf.html)

```
<form method="POST">
    {{ form.hidden_tag() }}
```

```
    {{ form.username.label }} {{ form.username() }}<br>
    {{ form.email.label }} {{ form.email() }}<br>
    {{ form.submit() }}
</form>
```

✅ Summary

- Manual Validation: Check request.form values and apply conditions.
- Flask-WTF: Provides built-in validators, CSRF protection, and cleaner code.
- Always return proper error messages and status codes if validation fails.

Q8. How do you manage sessions in Flask?

Explanation:

In Flask, sessions are used to store user-specific data across multiple requests. Flask provides a session object, which is client-side (stored in cookies) but signed to prevent tampering. This is commonly used to manage login status, preferences, or temporary data.

1. Basic Usage of session

```
 from flask import Flask, session, redirect, url_for, request

app = Flask(__name__)
app.secret_key = 'supersecretkey'  # Required to sign session cookies

@app.route('/login', methods=['POST'])
def login():
    username = request.form.get('username')
    if username:
        session['username'] = username  # Store username in session
        return f"Logged in as {username}"
    return "Username required!", 400
```

```
@app.route('/profile')
def profile():
    if 'username' in session:
        return f"Welcome, {session['username']}!"
    else:
        return "You are not logged in.", 401

@app.route('/logout')
def logout():
    session.pop('username', None)  # Remove username from session
    return "Logged out!"
```

2. Key Points

1.app.secret_key

- Required to cryptographically sign session cookies.

2.Storing Data

- Use session['key'] = value to store data.

3.Accessing Data

- Use session.get('key') or session['key'].

4.Removing Data

- Use session.pop('key', None) or session.clear() to remove all session data.

5.Client-Side Storage

- Flask stores session data in cookies, signed to prevent modification. Avoid storing large amounts of data.

✅ Summary

- The session object in Flask allows persistent user-specific data across requests.

- Common use cases: login sessions, user preferences, flash messages.
- Always set secret_key for security and avoid storing sensitive or large data directly in the session.

Q9. How do you redirect to a different route in Flask?

Explanation:

In Flask, you can redirect a user to a different route using the redirect() function, often in combination with url_for() to generate the URL dynamically. This is commonly used after form submissions, login/logout, or other actions where you want to send the user to another page.

1. Basic Redirect from flask import Flask, redirect, url_for

```
app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"

@app.route('/login')
def login():
    # After login logic, redirect to home
    return redirect(url_for('home'))

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

Explanation:

- redirect() ⟶ Tells the browser to navigate to a new URL.
- url_for('home') ⟶ Dynamically generates the URL for the home route.

2. Redirect After Form Submission

```
from flask import Flask, request, redirect, url_for

app = Flask(__name__)

@app.route('/submit', methods=['POST'])
def submit():
    username = request.form.get('username')
    # Process data here...
    return redirect(url_for('thank_you', name=username))

@app.route('/thank-you/<name>')
def thank_you(name):
    return f"Thank you, {name}, for submitting!"
```

- After a POST request to /submit, the user is redirected to /thank-you/.

3. Key Points:

1.Always use url_for() instead of hardcoding URLs to keep routes maintainable.

2.redirect() returns a 302 Found status by default. You can specify other codes like 301 (permanent) if needed:

```
return redirect(url_for('home'), code=301)
```

3.Redirects are often used in Post/Redirect/Get pattern to prevent form resubmission.

✅ Summary

- Use redirect(url_for('route_name')) to navigate to another route.
- Helps in navigation, form handling, and flow control.
- Keeps URLs dynamic and maintainable using url_for().

Q10. How do you handle errors in Flask (e.g., 404)?

Explanation:

In Flask, you can handle errors like 404 Not Found, 500 Internal Server Error, or custom errors using error handlers. This lets you return custom responses or templates when something goes wrong, instead of the default Flask error page.

1. Using @app.errorhandler

```
from flask import Flask, jsonify, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return "Welcome to the Home Page!"

# Handle 404 errors
 @app.errorhandler(404)
def not_found(error):
    return jsonify({"error": "Page not found"}), 404

# Handle 500 errors
@app.errorhandler(500)
def server_error(error):
    return jsonify({"error": "Internal server error"}), 500

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

Explanation:

- @app.errorhandler(status_code) → Decorator to handle specific HTTP errors.
- You can return JSON, HTML templates, or any custom response.
- The second value in the return tuple sets the HTTP status code.

2. Handling 404 with HTML Template

```
@app.errorhandler(404)
def page_not_found(error):
    return render_template('404.html'), 404
```

templates/404.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Page Not Found</title>
</head>
<body>
    <h1>404 - Page Not Found</h1>
    <p>Sorry, the page you requested does not exist.</p>
</body>
</html>
```

This displays a friendly 404 page instead of the default Flask error page.

3. Raising Custom Errors

You can also raise errors manually in routes:

```
from flask import abort

@app.route('/user/<int:user_id>')
def get_user(user_id):
    users = [1, 2, 3]
    if user_id not in users:
        abort(404)  # Triggers the 404 error handler
     return f"User {user_id}"
```

✅ Summary

- Use @app.errorhandler() to handle specific HTTP errors.
- Return custom JSON or HTML templates for better user experience.
- Use abort() to manually trigger errors when needed.
- Helps in building robust and user-friendly APIs or websites.

Q11. How do you structure a Flask app using Blueprints?

Explanation:

In Flask, Blueprints allow you to organize your application into reusable and modular components. This is especially useful for large applications with multiple routes, templates, and static files. Blueprints make your app more maintainable and help avoid clutter in a single file.

1. What is a Blueprint?

- A Blueprint is like a mini Flask app that defines routes, templates, and static files.
- It can then be registered with the main Flask application.

2. Basic Project Structure Using Blueprints

```
my_flask_app/
│
├── app.py            # Main application
├── users/            # Users module
│   ├── __init__.py
│   └── routes.py
├── products/         # Products module
│   ├── __init__.py
│   └── routes.py
└── templates/
```

3. Example: Users Blueprint

users/routes.py

```
 from flask import Blueprint, jsonify

 users_bp = Blueprint('users', __name__, url_prefix='/users')

 @users_bp.route('/')
  def get_users():
    return jsonify([{"id": 1, "name": "Piyush"}, {"id": 2, "name": "Ankit"}])

@users_bp.route('/<int:user_id>')
def get_user(user_id):
    return jsonify({"id": user_id, "name": f"User {user_id}"})
```

users/init.py

```
from .routes import users_bp
```

4. Register Blueprint in Main App

app.py

```
from flask import Flask
from users import users_bp

app = Flask(__name__)
app.register_blueprint(users_bp)  # Register the blueprint

@app.route('/')
def home():
    return "Welcome to the Home Page!"
```

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

5. How It Works

- Blueprint object defines routes and can have its own URL prefix (/users in this case).
- app.register_blueprint() connects the Blueprint to the main Flask app.
- Allows each module (users, products, etc.) to be developed and maintained separately.

✅ Advantages of Using Blueprints

- 567-Modular Structure – Separate concerns by feature or module.
- Reusable Components – Can be reused across different projects.
- Easier Collaboration – Teams can work on different modules independently.
- Supports Templates & Static Files – Each blueprint can have its own templates and static folders

Q12. How do you define a custom Jinja filter in Flask?

Explanation:

In Flask, you can define a custom Jinja filter to transform or format data in your templates. Jinja filters are used inside templates with the | symbol, and custom filters let you add your own formatting logic.

1. Basic Example

app.py

```
from flask import Flask, render_template

app = Flask(__name__)

# Define a custom filter function
```

```
def reverse_string(s):
    return s[::-1]

# Register the filter with Flask
app.jinja_env.filters['reverse'] = reverse_string

@app.route('/')
def home():
    name = "Riya"
    return render_template('index.html', name=name)

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

templates/index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Custom Filter Example</title>
</head>
<body>
    <h1>Original Name: {{ name }}</h1>
    <h1>Reversed Name: {{ name | reverse }}</h1>
</body>
</html>
```

Output:

```
Original Name: Riya
Reversed Name: hsuiyP
```

2. Key Steps

- Define a Python function that takes input and returns the transformed output.
- Register it in Flask using:

```
app.jinja_env.filters['filter_name'] = your_function
```

- Use the filter in templates with the | symbol:

```
{{ variable | filter_name }}
```

3. Example: Custom Date Formatter

```
from datetime import datetime

def format_date(value, format='%d-%m-%Y'):
    return value.strftime(format)

app.jinja_env.filters['format_date'] = format_date
```

Usage in Template

```
{{ current_date | format_date("%B %d, %Y") }}
```

✅ Summary

- Custom Jinja filters let you transform or format data in templates.
- Steps: define function → register as filter → use in template.
- Useful for reusable formatting logic like dates, strings, or numbers.

Q13. How can you redirect with query parameters in Flask?

Explanation:

In Flask, you can redirect a user to a different route with query parameters by using the redirect() function along with url_for(), passing the parameters as keyword arguments.

1. Basic Example

```
from flask import Flask, redirect, url_for, request

 app = Flask(__name__)

@app.route('/')
 def home():
      return "Welcome to the Home Page!"

 @app.route('/search')
def search():
     query = request.args.get('q', '')  # Get query parameter
     return f"Search results for: {query}"

 @app.route('/go-to-search')
  def go_to_search():
     # Redirect to /search?q=flask
     return redirect(url_for('search', q='flask'))

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

Explanation:

- url_for('search', q='flask') generates /search?q=flask.
- redirect() sends the client to that URL.
- request.args.get('q') retrieves the query parameter in the target route.

2. Using Multiple Query Parameters

```
    return redirect(url_for('search', q='flask', page=2))
```

- Generates /search?q=flask&page=2

**3. Key Points **

- Use url_for('route_name', key=value) to pass query parameters.
- redirect() performs an HTTP 302 redirect by default.
- In the target route, access query parameters with request.args.get('param').

✅ Summary

- Redirect with query parameters by passing them to url_for().
- Retrieve them in the destination route using request.args.
- Supports multiple parameters and keeps URLs dynamic and maintainable.

Q14. How do you return JSON responses in Flask?

Explanation:

In Flask, returning JSON responses is common for APIs. Flask provides the jsonify() function to convert Python dictionaries or lists into JSON with the correct content type (application/json).

1. Basic Example Using jsonify

```
from flask import Flask, jsonify

 app = Flask(__name__)

  @app.route('/user')
 def get_user():
```

```
    user = {
      "id": 1,
      "name": "Riya",
      "email": "Riya@example.com"
    }
    return jsonify(user)  # Convert dict to JSON response

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

Output (JSON):

```
{
  "id": 1,
  "name": "Riya",
  "email": "Riya@example.com"
}
```

- Flask automatically sets the Content-Type header to application/json.

2. Returning a List of Objects

```
@app.route('/users')
def get_users():
    users = [
       {"id": 1, "name": "Riya"},
        {"id": 2, "name": "Ankit"}
    ]
    return jsonify(users)
```

3. Returning JSON with Custom Status Code

```
@app.route('/create_user', methods=['POST'])
def create_user():
    user = {"id": 3, "name": "Rahul"}
    return jsonify(user), 201  # 201 Created
```

- The second value in the return tuple sets the HTTP status code.

4. Key Points

- Use jsonify() instead of json.dumps() for proper headers.
- You can return dict, list, or nested structures.
- Combine with HTTP status codes for API responses.

✅ Summary

- jsonify() converts Python objects to JSON responses.
- Automatically sets the Content-Type to application/json.
- Can include custom HTTP status codes for API responses.

```
**Q15. How do you capture URL parameters in Flask?**

Explanation:

In Flask, you can capture URL parameters (also called path
parameters) by defining dynamic segments in your route usin
>. Flask passes these values as arguments to your view func

**1. Basic Example**

        from flask import Flask

        app = Flask(__name__)
```

Q15. How do you capture URL parameters in Flask?

Explanation:

In Flask, you can capture URL parameters (also called path parameters) by defining dynamic segments in your route using < >. Flask passes these values as arguments to your view function.

1. Basic Example

```
        @app.route('/user/<username>')
        def show_user(username):
            return f"Hello, {username}!"

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


- Visiting /user/Riya → Output: Hello, Riya!

- <username> in the route becomes the username argument in
function.

**2. Capturing Different Data Types**

You can specify types in the route to ensure correct conver

         @app.route('/post/<int:post_id>')
          def show_post(post_id):
             return f"Post ID: {post_id}"


- Visiting /post/10 → Output: Post ID: 10

- <int:post_id> ensures post_id is an integer.

Other types:

- string (default) → /user/<string:name>

- int → /post/<int:id>

- float → /rating/<float:value>

- path → /path/<path:subpath> (captures slashes /)

**3. Multiple Parameters**

        @app.route('/user/<username>/post/<int:post id>')
```

```
from flask import Flask

app = Flask(__name__)

@app.route('/user/<username>')
def show_user(username):
    return f"Hello, {username}!"

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

- Visiting /user/Riya → Output: Hello, Riya!
- in the route becomes the username argument in the function.

2. Capturing Different Data Types

You can specify types in the route to ensure correct conversion:

```
@app.route('/post/<int:post_id>')
 def show_post(post_id):
    return f"Post ID: {post_id}"
```

- Visiting /post/10 → Output: Post ID: 10
- int:post_id ensures post_id is an integer.

Other types:

- string (default) → /user/string:name
- int → /post/int:id
- float → /rating/float:value

```
def user_post(username, post_id):
    return f"{username} is viewing post {post_id}"
```

- Visiting /user/Riya/post/5 → Output: Riya is viewing post

✅ Summary

- Use <parameter> in route to capture URL parameters.

- Specify types for automatic conversion (int, float, path)

- Flask passes captured parameters as function arguments.

- path → /path/path:subpath (captures slashes /)

3. Multiple Parameters

```
@app.route('/user/<username>/post/<int:post_id>')
def user_post(username, post_id):
    return f"{username} is viewing post {post_id}"
```

- Visiting /user/Riya/post/5 → Output: Riya is viewing post 5

✅ Summary